# Informatics II

## Introduction, Sorting, Recursion
## — SL01 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

# Table of contents — SL01

1. Introduction
   Introduction

2. Algorithms

3. Sorting

4. Recursion

# LITERATURE

- T. Cormen, C. Leiserson, R. Rivest and C. Stein (CLRS), Introduction to Algorithms, Fourth Edition, MIT Press, 2022.
- An excellent book that covers all topics of this course (as well as many others).
- Available as an ebook from inside the UZH network:

https://ebookcentral.proquest.com/lib/uzh/detail.action?docID=6925615

# Goals of this course

The main things we will learn in this course:

- To **think algorithmically** and get the spirit of how algorithms are designed.
- To get to know a **toolbox** of classical algorithms.
- To learn a number of algorithm **design techniques** (such as divide-and-conquer).
- To reason in a precise way about the **efficiency** and the **correctness** of algorithms.

# SYLLABUS

math + recursion + C

1. **Introduction, basic sorting, recursion** (chap 1 in CLRS)
2. **Complexity and correctness** (chap 2, 3 in CLRS)
3. **Divide and conquer, recurrences** (chap 4 in CLRS)
4. **Heap sort, quick sort** (chap 6, 7 in CLRS)
5. **Pointers, lists, sets, abstract data types** (chap 10 in CLRS)
6. **Trees, red-black trees** (chap 12, 13 in CLRS)
7. **Hash tables** (chap 11 in CLRS)
8. **Dynamic programming** (chap 15 in CLRS)
9. **Graph algorithms** (chap 22, 23, 24 in CLRS)

data structure + algorithms

C

data struct + algos

# Administration/1

**Home page**:

- https://lms.uzh.ch/url/RepositoryEntry/17335386476
- Check OLAT frequently and carefully.
- Official communications happen via OLAT email to all registered students.
- Official communications will also be posted in the OLAT forum.

# ADMINISTRATION/2

**Lectures**:

- Tuesday 14:00 - 15:45 in HAH-E-11 (Häldeliweg)
- Friday 12:15 - 13:45 in Y15-G-60 (Irchel)
- First lecture: Tuesday 21.2.2023
- Last lecture: Friday 2.6.2023
- 27 double-lectures that take place on-site.
- No lectures during the spring break:
  FR 7.4.2023, TU 11.4.2023, FR 14.4.2023

- On-site interactions during lectures are welcome.
- Annotated slides are provided on OLAT.
- Lectures are not streamed and no podcasts are provided.
- Old recordings (for personal use; distribution not allowed) at `https://seafile.ifi.uzh.ch/d/c0f2679473b94e52b160/`

# ADMINISTRATION/3

**Exam**:

- The assessment is the final exam. There is no re-exam.
- The exam is closed book without auxiliary material.
- The final exam date is Wednesday, 28.6.2023 between 8:00 and 12:00 (check VVZ and the official WWF web pages).
- The exam lasts 90 minutes and is a BYOD (bring your own device) exam.
- The mandatory mock exam will be held on-site in room HAH-E-3 on April 26, 2023 from 18:00 to 20:00.

- Prepare for the final exam throughout the semester by actively participating in the course and the exercises.
- There is not enough time to prepare everything during the exam period.

# ADMINISTRATION/4

**Assistants**:

- Qing Chen, `qing@ifi.uzh.ch`
- Muhammad Saad, `saad@ifi.uzh.ch`

Qing Chen

Muhammad Saad

- Please use the OLAT forum to contact the assistants if there are any problems or if you have general questions about lectures, exercises and exam.

# ADMINISTRATION/5

setup: edit + compile + run a program with gcc

**Exercises**:

- There are 1 + 12 weekly exercises.
- There is no hand in of exercises.
- An exercise is published one week before the solution is being discussed in a lab.
- You get solutions for the exercises one week after the exercise was published and before the exercise is discussed in the lab.

- Try to solve the exercises before the lab.
- Trying to solve exercises on your own is important and a good preparation for the exam.

# ADMINISTRATION/6

**TAs and tutors**:

- The TAs and tutors prepare the exercises and teach the labs.
  - Jonas Blum, `jonas.blum2@uzh.ch`
  - Running Hou, `running.hou@uzh.ch`
  - Zheng Luo, `zheng.luo@uzh.ch`
  - Christoph Mayer, `christoph.mayer2@uzh.ch`
  - David Jonas Scherrer, `davidjonas.scherrer@uzh.ch`
  - Xinyu Zhu, `xinyu.zhu@uzh.ch`
- The TAs and tutors are most happy to help you and answer questions related to the content of the exercises.
- Please interact with the TAs and tutors during the labs and by posting in the OLAT forum.

# Administration/7

**TAs and tutors:**

Jonas Blum

Running Hou

Zheng Luo

Christoph Mayer

David Jonas Scherrer

Xinyu Zhu

# ADMINISTRATION/8

**Labs** (exercise classes + office hours):

- Monday 16:15-18:00, BIN 0.B.06: David Scherrer
- Tuesday 10:15 - 12:00, BIN 0.B.06: Xinyu Zhu
- Wednesday 16:15 - 18:00, BIN 0.B.06: Zheng Luo
- Thursday 14:00 - 15:45, BIN 0.B.06: Jonas Blum
- Friday 9:00 - 11:00, BIN 0.B.06: Running Hou

- Exceptions: Fri Apr 7, Mon Apr 17, Mon May 1, Wed May 17, Thu May 18, and Mon May 29 will be rescheduled.
- The tutors will reschedule labs at least two weeks in advance and announce the rescheduling in the OLAT forum.

# ADMINISTRATION/9

**Office hours**:

- The office hours follow right after the exercise classes.

- The exercise class lasts 90 minutes.
- During the exercise class the solution of the exercise is discussed. Interaction is welcomed and important.
- The goal is to learn how to understand and approach tasks, how to come up with a plan for a solution, and how to implement the solution.

- The office hours follow the exercise classes (after a 15′ break) and last 60 minutes.
- During the office hours you can talk one-on-one with your tutor to solve individual problems.

# General remarks/1

- **Hands-on exercises** are an important part of this course: an abstract understanding of the concepts is not good enough
- You must be able to **apply** your knowledge to **new** examples. Use exercises to practice this during the semester.
- Often it is most effective to first solve algorithms on paper and later key them in on the computer.
- A very important thing is to be **simple** and **precise**.
- There will be tasks in the final exam that require you to either write C code or pseudocode.

# General remarks/2

- During lectures:
  - Interaction is welcome; ask questions.
  - Speed up/slow down the progress.
  - Additional explanations if desired.
- During the lectures we solve many examples. Please participate and take notes. Trying, failing and improving is good. Not trying is bad.
- The slides are designed as **working lecture notes**. Solutions to selected exercises are included at the end.

# Prerequisites

Introduction to programming (aka Informatik I)

- Data types, operations
- Conditional statements if stmt
- Loops for while
- Procedures and functions

Ability to edit, compile, and run a program

- editor (Sublime, Atom, VIM, Nano, gedit, notepad, ...)
- terminal (xterm, gnome-terminal, powershell, ...)
- IDE: VS Code, Visual Studio, Xcode, Eclipse, ...

# Editor + Terminal

editor (gedit)

Open ▾ search.c ~/Teaching/FS23/InfII/Code

```
int j, q;
int a[] = {11, 1, 4, -3, 22};

int main() {
  j = 0; q = -3;
  while (j < n && a[j] != q) { j++; }
  if (j < n) { printf("%d\n", j); }
  else { printf("NIL\n"); }
}

// gcc search.c -o search; ./search
```

compile & link

run

File Edit View Search Terminal Help

terminal

```
boehlen@a1:~/Teaching/FS23/InfII/Code> gcc search.c -o search; ./search
3
boehlen@a1:~/Teaching/FS23/InfII/Code>
```

# Visual Studio Code (VS Code)

File Edit Selection View Go Run Terminal Help

search.c U

home > boehlen > Teaching > FS23 > InfII > Code > search.c > ...

```c
#include <stdio.h>

#define n 5

int j, q;
int a[] = {11, 1, 4, -3, 22};

int main() {
  j = 0; q = -3;
  while (j < n && a[j] != q) { j++; }
  if (j < n) { printf("%d\n", j); }
  else { printf("NIL\n"); }
}

// gcc search.c -o search; ./search
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL bash - Code

compile & link

```
boehlen@a1:~> cd Teaching/FS23/InfII/Code/
boehlen@a1:~/Teaching/FS23/InfII/Code> gcc search.c -o search; ./search
3
boehlen@a1:~/Teaching/FS23/InfII/Code>
```

execute

main* 0 0 Ln 15, Col 36 (32 selected) Spaces: 2 UTF-8 LF C Linux

# Acknowledgments


- The slides are based on the textbook Introduction to Algorithms from T. Cormen, C. Leiserson, R. Rivest and C. Stein.
- A very early version of these slides was developed by Simonas Saltenis from Aalborg University, Denmark.
- The course is based on a course that Michael Böhlen taught at the Free University of Bolzano, Italy.
- Kurt Ranaltar created the initial Latex version of these slides.

# Table of contents — SL01

1. Introduction

2. Algorithms
   Algorithms
   Data structures

3. Sorting

4. Recursion

# What is an algorithm?

- Algorithms are about solving problems
  - Enrollment at UZH
  - Booking a module
  - Graduate from IfI@UZH
  - Get me from home to work
  - Earn lots of money
  - Simulate a jet engine
  - Human genome project: algorithms to store and analyze DNA
  - Electronic commerce: algorithms for public-key cryptography and digital signatures
  - Internet/WWW: algorithms to determine good routes for data and to quickly find data

# What is an algorithm?

- Algorithms are about solving problems
  - Enrollment at UZH
  - Booking a module
  - Graduate from IfI@UZH
  - Get me from home to work
  - Earn lots of money
  - Simulate a jet engine
  - Human genome project: algorithms to store and analyze DNA
  - Electronic commerce: algorithms for public-key cryptography and digital signatures
  - Internet/WWW: algorithms to determine good routes for data and to quickly find data
- Algorithms = procedures, recipes, process descriptions

# History

- Etymology: from Persian mathematician al-Khwarizmi
- 400-300 B.C.
  - First algorithm: Euclidean algorithm, greatest common divisor
- 19th century
  - Charles Babbage (first mechanical computer) relais add multip
  - Ada Lovelace (first computer program)
- 20th century
  - Alan Turing (Turing machine, cryptography)
  - Alonzo Church (decidability)
  - John von Neumann (Von Neumann architecture)

# Terminology

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure

# Terminology

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure
- **Algorithm**
  - Finite sequence of unambiguous instructions
  - Step-by-step outline of computational procedure
  - Independent from choice of programming language

# Terminology

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure
- **Algorithm**
  - Finite sequence of unambiguous instructions
  - Step-by-step outline of computational procedure
  - Independent from choice of programming language
- **Program**
  - Implementation of given computational procedure
  - Programming language, software engineering issues

# Overall picture/1

- Specification
  - Precisely specify problem

# Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency

# Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency
- Development
  - Implement algorithm in some language
  - Goals: robustness, reusability, adaptability

# Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency
- Development
  - Implement algorithm in some language
  - Goals: robustness, reusability, adaptability
- Testing
  - Verify that implementation meets specification

# Overall picture/2

What this course is **not** about

- Software architecture
- Computer architecture
- Programming languages
- Software engineering

# Overall picture/2

What this course is **not** about

- Software architecture
- Computer architecture
- Programming languages
- Software engineering

Other related topics that we touch upon

- Computability and complexity
- Efficiency vs NP-completeness

# ALGORITHMS/1

An **algorithm** is

- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  i.e., an unambiguous sequence of computational steps that transform the input into the output

# ALGORITHMS/1

An **algorithm** is

- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  i.e., an unambiguous sequence of computational steps that transform the input into the output
- A tool for solving a well-specified computational problem
  - Problem statement specifies the desired input/output relation
  - Algorithm describes a procedure for achieving such a relation

# ALGORITHMS/2

**Algorithmic problem**

Input/output relation

1. step understand the problem (= for each input be able to tell what output is)

input specification $\rightsquigarrow$ $\cdots$ $\rightsquigarrow$ output specification

Specification of output as function of specification of input

# ALGORITHMS/2

**Algorithmic problem**

Input/output relation

$$\boxed{\text{input specification}} \rightsquigarrow \boxed{\cdots} \rightsquigarrow \boxed{\text{output specification}}$$

Specification of output as function of specification of input

**Algorithmic solution**

Input/output relation

2. step: solve problem

$$\boxed{\text{input instance}} \rightsquigarrow \boxed{\text{algorithm}} \rightsquigarrow \boxed{\text{output instance}}$$

Procedure transforming input instance into output instance

# How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.

# How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.
2. Come up with a simple plan to solve the problem at hand.
   - The plan is language independent.
   - The precise problem specification influences the plan.

# How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.
2. Come up with a simple plan to solve the problem at hand.
   - The plan is language independent.
   - The precise problem specification influences the plan.
3. Turn the plan into an implementation
   - The problem representation (data structure) influences the implementation.

# Example of an algorithm/1

- Searching problem
  - Input: sequence (array) of $n$ numbers $A = [a_1, a_2, \ldots, a_n]$ and a value $v$
  - Output: index $i$ such that $v = a_i (= A[i])$ or the special value `NIL` if $v$ does not appear in $A$

# Example of an algorithm/1

- Searching problem
  - Input: sequence (array) of $n$ numbers $A = [a_1, a_2, \ldots, a_n]$ and a value $v$
  - Output: index $i$ such that $v = a_i (= A[i])$ or the special value `NIL` if $v$ does not appear in $A$
- Instance of searching (example)
  - Input: $A = [31, 59, 26, 41, 58]$, $v = 59$
  - Output: $i = 2$

is this precise?

$A = [1, 1, 1, 1]$

$v = 1$

Output?

# Example of an algorithm/2

Example: linear search

- Input:
  sequence of numbers $A = [a_1, a_2, \ldots, a_n]$ and value $v$
- Output:
  index $i$ such that $v == A[i]$ or NIL if $v$ does not appear in $A$

# Example of an algorithm/2

Example: linear search

- Input:
  sequence of numbers $A = [a_1, a_2, \ldots, a_n]$ and value $v$
- Output:
  index $i$ such that $v == A[i]$ or NIL if $v$ does not appear in $A$
- Algorithms

**Algo:** LinSearch1(A,v)

```
p = NIL;
for i = 1 to n do
    if A[i]==v then p = i;
return p;
```

index of last element equal to v

**Algo:** LinSearch2(A,v)

```
i = 1;
while i≤n ∧ A[i]≠v do i++;
if i≤n then return i;
else return NIL;
```

index of first element equal to v

# Example of an algorithm/3

## Comparing algorithms

- Assume that $A = [31, 41, 59, 26, 41, 58]$ and $v = 41$ Then:
  - `LinSearch1`$(A, v) = 5$
  - `LinSearch2`$(A, v) = 2$
  - `LinSearch1`$(A, v) \neq$ `LinSearch2`$(A, v)$
- Ambiguous problem statement: first or last index?

# Example of an algorithm/4

## Fundamental properties

- Efficiency of an algorithm
  - `LinSearch2` is more efficient in general
  - `LinSearch1` always scans the entire array

# Example of an algorithm/4

- n? (global variable)
- v (pseudocode for 11)

## Fundamental properties

- Efficiency of an algorithm
  - `LinSearch2` is more efficient in general
  - `LinSearch1` always scans the entire array
- Correctness of an algorithm

- n = 10
  i goes to 11
  ⇒ A[11] ≠ out of bound

**Algo:** LinSearch3(A,v)

i = 1;
**while** $i \leq n \lor A[i] \neq v$ **do**
  **if** $i > n$ **then** break **else** i = i+1;
**if** $i \leq n$ **then return** i **else return** NIL;

- `LinSearch3` is incorrect; it crashes for all inputs.

# EXAMPLE OF AN ALGORITHM/5

- What about the fourth solution for searching?

**Algo:** LinSearch4(A,v)

**for** $i = 1$ **to** $n$ **do**
  **if** $A[i]==v$ **then return** i;
**return** NIL;

- OK?

returns index of first element equal to v

(equivalent to LinSearch2)

# Example of an algorithm/5

- What about the fourth solution for searching?

**Algo:** LinSearch4(A,v)

**for** $i = 1$ **to** $n$ **do**
  **if** $A[i]==v$ **then return** i;
**return** NIL;

- OK?
- There is always more than one solution to a given problem.

# Pseudo code

Use of pseudocode

- Similar in many respects to C, Pascal, Java, Python (assignments, control structures, arrays, ...)
- Provides means to convey the essence of an algorithm in a concise fashion
- No fixed format for pseudocode. OK as long as it is unambiguous and breaks down the problem to the relevant basic steps.
- Pseudocode works out the important parts and abstracts the unimportant parts.
- It is important that you can write algorithms in pseudocode and C.

# EXERCISE SL01-1/1

A prime number is a natural number greater than 1 and divisible only by 1 and itself. Write a program that determines all prime numbers between 0 and $n$.

*Hint:* Implement the sieve of Eratosthenes. Use an array of length $n + 1$ that indicates whether the number corresponding to the index is prime. Start with an array where all numbers greater than 1 are marked as primes and then proceed as follows: first eliminate the multiples of 2, then the multiples of 3, then the multiples of 4, etc.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A = | F | F | T | T | T | T | T | T | T | T |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A = | F | F | T | T | F | T | F | T | F | T |

# EXERCISE SL01-1/2

# EXERCISE SL01-1/3

array after initialization =

| i \ j | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 2 | 2·2 | 2·3 | 2·4 | 2·5 | 2·6 | 2·7 |
| 3 | 3·2 | 3·3 | 3·4 | 3·5 | | |
| 4 | 4·2 | 4·3 | | | | |
| 5 | 5·2 | 5·3 | | | | |
| 6 | 6·2 | | | | | |
| 7 | 7·2 | | | | | |

symmetric

$n = 15$

$\sqrt{n}\sqrt{n} = n$

Algo: Era

```
A[0] = false;
A[1] = false;
for i = 2 to n do
    A[i] = true
```

```
for i = 2 to ⌊n/2⌋ do
    for j = 2 to ⌊n/i⌋ do
        A[i * j] = false
```

```
for i = 2 to ⌊√n⌋
    for j = i to ⌊n/i⌋
```

# SEARCHING, C SOLUTION

```
#include <stdio.h>

#define n 5

int i, v;
int a[] = { 11, 1, 4, -3, 22 };

int main() {
  i = 0;  v = -2;
  while (i < n && a[i] != v) { i++; }
  if (i < n) { printf("%d\n", i); }
  else { printf("NIL\n"); }
}
// gcc search.c -o search; ./search
```

# How to approach C/1

- Do not study it (it is close to Python and Java and we only use a small subset).
- Whenever you meet a new construct learn and use it.
- Here:
  - `#include <stdio.h>` includes IO library with printf function.
  - `#define n 5`
    defines n as a constant with value 5.
  - `printf("%d\n",i)`
    prints an integer argument (`%d` is replaced by the value of i) followed by a new line (`\n`).
  - Arrays have a fixed size and start with index 0.
  - `//` is the start of a one line comment.
  - `gcc -o search search.c; ./search` command to compile and run the program from the command line.

# How to approach C/2

- In C the formatting of code (e.g., line breaks) is irrelevant.
  - For humans the formatting is very important.
  - Make sure you format your code properly.
  - Look at code examples in the lecture slides.
- A block that belongs together is put into { ... }.
- Always use a small working code example for illustration.

- Declaring an integer array with 5 elements: `int a[5];`
  - The first element has index 0: `a[0] = 132;`
  - The last element has index 4: `a[4] = 12;`
- A string is a character array: `char s[25];`
  - The character `'\0'` marks the end of a string.
  - `s[0] = 'A';  s[1] = 'B';  s[2] = '\0';`

# Searching, Java solution

```
import java.io.*;

class search {
  static int n = 5;
  static int i, v;
  static int a[] = { 11, 1, 4, -3, 22 };

  public static void main(String args[]) {
    i = 0; v = 22;
    while (i < n && a[i] != v) { i++; }
    if (i < n) { System.out.println(i); }
    else { System.out.println("NIL"); }
  }
}
// javac search.java; java search
```

classes are too much overhead for us

# Searching, Python solution

```
a = [ 11, 1, 4, -3, 22 ]
v = -3

j = -1
for i in range (len(a)):
    if a[i] == v:
        j = i
print(j)

# python search.py
```

# Data structures

Informal definition

- A data structure is a way to store and organize data
- As such it facilitates access and modification of data
- Elementary data structures: lists, stacks, queues, trees

# Data structures

## Informal definition

- A data structure is a way to store and organize data
- As such it facilitates access and modification of data
- Elementary data structures: lists, stacks, queues, trees

## Things to consider

- No single data structure works well for all purposes
- Choice of appropriate data structure is a **crucial** issue
- Important to know strengths and limitations of a variety of data structures.

# Table of contents — SL01

1. Introduction

2. Algorithms

3. Sorting
   - Bubble sort
   - Selection sort
   - Insertion sort

4. Recursion

# Sorting/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques

# SORTING/1

## Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques

## Some observations

- Sorting of arrays: does provide random access (each element can be directly accessed)
- Efficient management of space: in-place sorting
- Efficiency is generally measured in terms of
  - number of comparisons and/or
  - number of exchange operations

# Sorting/2

Simple methods

- Bubble sort
- Selection sort
- Insertion sort
- Roughly $n^2$ comparisons

# Sorting/2

## Simple methods

- Bubble sort
- Selection sort
- Insertion sort
- Roughly $n^2$ comparisons

slow

## Fast methods

- Merge sort
- Heap sort
- Quick sort
- Roughly $n \log n$ comparisons

fast

# SORTING/3

## Problem specification

- Sorting problem
  - Input: a sequence of $n$ numbers $A = [a_1, a_2, \ldots, a_n]$
  - Output: a permutation (reordering) $[b_1, b_2, \ldots, b_n]$ of the input sequence such that $b_1 \leq b_2 \leq \cdots \leq b_n$

# SORTING/3

good?

nice. precise
no holes.

BUT no hint
how to solve
it

## Problem specification

- Sorting problem
  - Input: a sequence of $n$ numbers $A = [a_1, a_2, \ldots, a_n]$
  - Output: a permutation (reordering) $[b_1, b_2, \ldots, b_n]$ of the input sequence such that $b_1 \leq b_2 \leq \cdots \leq b_n$
- Instance of sorting
  - Input sequence: $[31, 41, 59, 26, 41, 58]$
  - Output sequence: $[26, 31, 41, 41, 58, 59]$

$\leq \quad \leq \quad \leq \quad \leq \quad \leq$

21.2

# Bubble sort/1

Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

# BUBBLE SORT / 1

A[0 .. n-1]
A[1 .. n]

## Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

## The algorithm

why not to 1

**Algo:** BubbleSort(A)

```
for i = n to 2 do
    for j = 2 to i do
        if A[j]<A[j-1] then
            t = A[j];
            A[j] = A[j-1];
            A[j-1] = t;
```

A[1]

A[0]

# Bubble sort / 1

## Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

## The algorithm

**Algo:** BubbleSort(A)

**for** $i = n$ **to** $2$ **do**
  **for** $j = 2$ **to** $i$ **do**
    **if** $A[j] < A[j-1]$ **then**
      t = A[j];
      A[j] = A[j-1];
      A[j-1] = t;

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| s | o | r | t | i | n | g | n | n-1 |
| o | r | s | i | n | g | t | n-1 | n-2 |
| o | r | i | n | g | s | t | n-2 | n-3 |
| o | i | n | g | r | s | t | | |
| i | n | g | o | r | s | t | 3 | |
| i | g | n | o | r | s | t | 2 | 1 |
| g | i | n | o | r | s | t | | |

# BUBBLE SORT/2

$1 + 2 + 3 + 4 + 5$

$6$ $6$ $6/2$

$\frac{5}{2} \cdot 6$

## Basic properties

- Number of comparisons:

$$C = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$$

- The number of comparisons is independent of the original ordering and the values.

$n-1 + n-2 + \dots 3-1 + 2-1$

$\parallel$

$1 + 2 + \dots n-1$

$n = 100$

$10'000$

# BUBBLE SORT/3

Number of moves:

no elements are moved (array is sorted in beginning)

$$Mmin = 0$$

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

worst case (we always exchange) (inverse order)

3 times # of comparisons

# EXERCISE SL01-2

loc where to place the smallest element

Modify the bubble sort algorithm such that, at each iteration of the outer loop, the smallest element of the corresponding subarray is moved to the left. Implement this variant of bubble sort and show how it works on a concrete example.

7 8 5 3

3 | 7 8 5

3 5 | 7 8

3 5 7 | 8

```
BubbSz(A)
for i = 1 to n-1
    for j = n-1 to i
        if A[j] > A[j+1] then
            A[j] ↔ A[j+1]
```

# Selection sort/1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

# SELECTION SORT/1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

The algorithm

**Algo:** SelectionSort(A)

```
for i = 1 to n-1 do
    k = i;
    for j = i+1 to n do
        if A[j]<A[k] then k = j;
    exchange A[i] and A[k];
```

# SELECTION SORT / 1

1 5 7 8

1 5 7 8

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

The algorithm

**Algo:** SelectionSort(A)

**for** $i = 1$ **to** $n$-1 **do**
  k = i;
  **for** $j = i+1$ **to** $n$ **do**
    **if** $A[j]<A[k]$ **then** k = j;
  exchange A[i] and A[k];

position where smallest elem goes to

position of smallest elem

place where smallest element is moved to

j

i

| | | | | | | |
|---|---|---|---|---|---|---|
| s | o | r | t | i | n | g |
| g | o | r | t | i | n | s |
| g | i | r | t | o | n | s |
| g | i | n | t | o | r | s |
| g | i | n | o | t | r | s |
| g | i | n | o | r | t | s |
| g | i | n | o | r | s | t |

# elem'

n-1

n-2

1

# SELECTION SORT/2

Basic properties

- Number of comparisons:

$$C = \sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

- The number of comparisons is independent of the original ordering.

# SELECTION SORT / 3

n = 100 elements; array is sorted

⇒ 297 moves

why? why not 0?

Basic properties

- Number of moves:

linear (good)

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

- Only the movement of elements is counted. Updating an index is not counted.

n = 100 ⇒ $100^2 = 10'000$ comp

297 moves

# EXERCISE SL01-3

Modify selection sort in such a way that, at each iteration of the outer loop, it selects the largest element and swaps it with the last element of the corresponding subarray. Implement this variant of selection sort and show how it works on a concrete example.

# Insertion sort/1

Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

# Insertion sort / 1

Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

The algorithm

**Algo:** InsertionSort(A)

```
for i = 2 to n do
    j = i-1;
    t = A[i];
    while j≥1 ∧ t<A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

# Insertion sort / 1

Rough idea 1 5 7 0

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

element that is inserted into the sorted part

The algorithm

**Algo:** InsertionSort(A)

**for** $i = 2$ **to** $n$ **do**
  j = i-1;
  t = A[i]; 1)
  **while** $j>1 \wedge t<A[j]$ **do**
    A[j+1] = A[j];
    j = j-1;
  A[j+1] = t; 6)

2) .. 5)

| | | | | | | |
|---|---|---|---|---|---|---|
| s | o | r | t | i | n | g |
| o | s | r | t | i | n | g |
| o | r | s | t | i | n | g |
| o | r | s | t | i | n | g |
| i | o | r | s | t | n | g |
| i | n | o | r | s | t | g |
| g | i | n | o | r | s | t |

# INSERTION SORT/2

## Basic properties

- Number of comparisons:

$$Cmin = \sum_{i=2}^{n} 1 = n - 1$$

element is the largest and stays at end

$$Cmax = \sum_{i=2}^{n} (i-1) = \frac{n^2 - n}{2}$$

element is smallest and goes to first place

# Insertion sort/3

## Basic properties

- Number of moves:

$$Mmin = \sum_{i=2}^{n} 2 = 2(n-1) = 2n-2$$

$$Mmax = \sum_{i=2}^{n} (i+1) = \frac{n^2+3n-4}{2}$$

# Table of contents — SL01

1. Introduction

2. Algorithms

3. Sorting

4. Recursion
   Marking a ruler

# RECURSION/1

Recursive object

- It contains itself as part of it
- It is defined in terms of itself

# Recursion/1

## Recursive object

- It contains itself as part of it
- It is defined in terms of itself

## Recursive procedure

- Procedure that calls itself
- Termination condition to stop recursion

# Recursion/1

## Recursive object

$5! = 5 \cdot 4!$

- It contains itself as part of it
- It is defined in terms of itself

## Recursive procedure

- Procedure that calls itself
- Termination condition to stop recursion

## Example: simple recursion

Factorial $$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fact(n-1) & \text{if } n > 0 \end{cases}$$

# RECURSION/2

Evaluate the following three implementations for computing the factorial of an integer $n$.

**Algo:** fact1(n)

**if** $n$==*0* **then return** 1 **else return** n * fact1(n-1);

**Algo:** fact2(n)

**if** $n$==*0* **then return** 1;
**return** fact2(n-1) * n;

**Algo:** fact3(n)

not good

**return** n * fact3(n-1);
**if** $n$==*0* **then return** 1;

∞ recursion

is too late

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

1 * `fact1`(0)

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

1 * `fact1`(0)

`fact1`(0)

# RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

1 * `fact1`(0)

`fact1`(0) | 1

# RECURSION/3

Tracing the call `fact1(3)`:

| | |
|---|---|
| `fact1`(3) | |
| 3 * `fact1`(2) | |
| 2 * `fact1`(1) | |
| 1 * `fact1`(0) | 1 |
| `fact1`(0) | 1 |

# RECURSION/3

Tracing the call `fact1(3)`:

| | |
|---|---|
| `fact1`(3) | |
| 3 * `fact1`(2) | |
| 2 * `fact1`(1) | 2 |
| 1 * `fact1`(0) | 1 |
| `fact1`(0) | 1 |

# RECURSION/3

Tracing the call `fact1(3)`:

| | |
|---|---|
| `fact1`(3) | |
| 3 * `fact1`(2) | 6 |
| 2 * `fact1`(1) | 2 |
| 1 * `fact1`(0) | 1 |
| `fact1`(0) | 1 |

# RECURSION/3

Tracing the call `fact1(3)`:

| | |
|---|---|
| `fact1`(3) | 6 |
| 3 * `fact1`(2) | 6 |
| 2 * `fact1`(1) | 2 |
| 1 * `fact1`(0) | 1 |
| `fact1`(0) | 1 |

# RECURSION/4

- There can be multiple recursions where a procedure calls itself multiple times.
- There can be mutual recursion where several procedures cal each other recursively.
- Example: mutual recursion

u is odd

Odd number $odd(n) = \begin{cases} false & \text{if } n = 0 \\ even(n-1) & \text{if } n > 0 \end{cases}$

Even number $even(n) = \begin{cases} true & \text{if } n = 0 \\ odd(n-1) & \text{if } n > 0 \end{cases}$

# EXERCISE SL01-4

logical operators || , &&

Design recursive algorithms for *odd* and *even*.

Algo even(n)
return n==0 || odd(n-1)

e(3) o(2) e(1) o(0)
||
False

Algo: odd(n)
return n>0 && even(n-1)

e(-3) o(-4) F ✓
o(-3) F f

Algo: odd(n)
return n<0 && odd(-n) ||
n>0 && even(n-1)

negative numbers

o(-3) o(3) e(2) o(1) f(0)

e(-3) o(-4) o(4) ...

# RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far

# RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack

# RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack
- In the above examples recursion is not necessary: there exist simple iterative solutions

# RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack
- In the above examples recursion is not necessary: there exist simple iterative solutions
- Recursion is more expensive than corresponding iterative solution: bookkeeping is necessary

# Marking a ruler/1

## Problem description

- Print the marks of a ruler
- Example:

# Marking a ruler/1

## Problem description

- Print the marks of a ruler
- Example:


## Recursive algorithm

**Algo:** Ruler(l,r,h) Ruler(0, 8, 4)

m = (r+l)/2;
**if** $h>0$ **then**
  Mark(m,h); middle
  Ruler(l,m,h-1); left
  Ruler(m,r,h-1); right

# Marking a ruler/2

Printing of marks:

`Ruler` $(0, 8, 3)$
`Mark` $(4, 3)$

# Marking a ruler/2

Printing of marks:

`Ruler` $(0, 8, 3)$
  `Mark` $(4, 3)$
  `Ruler` $(0, 4, 2)$
    `Mark` $(2, 2)$

# Marking a ruler/2

Printing of marks:

```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
```

# Marking a ruler/2

Printing of marks:

```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
    Ruler (2, 4, 1)
      Mark (3, 1)
      Ruler (2, 3, 0)
      Ruler (3, 4, 0)
  ...
```

24.2

# EXERCISE SL01-5/1

Consider the Sierpinski triangles of depths 2, 3, and 4 in the following pictures:

d = 2

d = 3

d = 4

- Determine the Sierpinski triangles of depths $d = 0$ and $d = 1$.
- Design a recursive algorithm that draws a Sierpinski triangle of depth $d$.
- Show how to call your algorithm.

call:
draw big triangle with (-1,-1), (1,-1), (0,1)
Sierpi(0,0,1,d)

# Exercise SL01-5/2

Algo SierpTr(x, y, h, d)

if d > 0 then

draw triangle with (x − h/2, y), (x + h/2, y), (x, y − h)

SierpTr(x − h/2, y − h/2, h/2, d − 1)

SierTr(x + h/2, y − h/2, h/2, d − 1)

SierTr(x, y + h/2, h/2, d − 1)

# HILBERT CURVE/1

Consider the following patterns:

- Hi is the Hilbert curve of order i (named after the inventor David Hilbert).
- Goal: write a program to produce Hi.

# HILBERT CURVE/2

- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.

curve length

$2^1-1$ $2^2-1$ $2^3-1$ $2^i-1$

# HILBERT CURVE/3

- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.

# HILBERT CURVE/4

Install FreeGLUT: `sudo apt install freeglut3-dev`

```
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

float x, y, u;
int d;

...

int main(int argc, char **argv) {
  glutInit(&argc, argv);
  glutCreateWindow("glut");
  sscanf(argv[1],"%d",&d);
  glutDisplayFunc(displayHandler);
  glutKeyboardFunc(keyboardHandler);
  glutMainLoop();
}

// gcc hilb.c -lglut -lGL -lGLU -lm -o hilb; ./hilb 3
```

# HILBERT CURVE/5

- GLUT offers event handler functions for dealing with display, keyboard, and mouse events.
- We define event handlers for the display and the keyboard.

```
void keyboardHandler(unsigned char k, int x, int y) {
  exit(0);
}

void displayHandler() {
  glClearColor(1,1,1,1);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0,0,0);
  x=0.8; y=0.8;
  u=1.6/(pow(2,d)-1);
  A(d);
}
```

$2^i - 1$

# HILBERT CURVE/6

## Organization of the screen:

## Drawing line (Ax,Ay) to (Bx,By):

```
glBegin(GL_LINES);
  glVertex2f(Ax,Ay);
  glVertex2f(Bx,By);
glEnd();
glutSwapBuffers();
```

```
void l(float dx, float dy) {
  glBegin(GL_LINES);
  glVertex2f(x,y);
    x=x+dx; y=y+dy;
    glVertex2f(x,y);
  glEnd();
  glutSwapBuffers();
}
```

# HILBERT CURVE/7

```
void A(int); void B(int); void C(int); void D(int);

void A(int i) {
  if (i>0) {
    D(i-1); l(-u,0);
    A(i-1); l(0,-u);
    A(i-1); l(u,0);
    B(i-1);
  }
}

void B(int i) { ...
  if (i>0) {
    C(i-1); l(0,u);
    B(i-1); l(u,0);
    B(i-1); l(0,-u);
    A(i-1);
  }
}

...
```

A = D ← A ↓ A → B

# EXERCISE SL01-6/1

A = A ↘ B → → D ↗ A

S = A ↘ B ↙ C ↖ D ↗

Consider the Sierpinski curve:

- The above pictures illustrate the Sierpinski curve of order 0, 1, 2 and 3.
- Identify the recursive pattern and sketch the code to draw the Sierpinski curve of order $i$.

# EXERCISE SL01-6/2

Algo: S(i)

if $i \leq 0$ then

A(i); l(u,u), B(i); l(-u,u);

....

Algo: A(i)

if $i \geq 0$ then

A(i-1); l(u,u); ...

# SUMMARY

- Algorithmic problem:
  - An algorithm transforms input data into output data
  - Precisely specify **input** and **output**; consider special cases; work out representative examples
  - A first step is to come up with a simple plan.
  - Split or revise complex plans until they get simple.
- Sorting algorithms
  - A classical and representative algorithmic problem.
  - Initialization and conditions in loops are important.
  - Make sure you get details of loops **correct by design**. Avoid trial and error.
- Recursive algorithms
  - Recursion is an important algorithmic technique.
  - Practice the design of recursive algorithms.
  - Consider **special cases**: termination, the first couple of recursive calls.

# Informatik II

## Complexity and Correctness
## — SL02 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

# TABLE OF CONTENTS — SL02

1. Algorithmic complexity
   - Efficiency
   - Runtime of insertion sort
   - Best, worst and average case
   - Runtime of binary search

2. Correctness

3. Asymptotic complexity

4. Special case analysis

detailed picture

large-scale picture

# EFFICIENCY/1

## Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage

# EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size

# EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size
- Various ways of determining the size of input data
  - Depends on the problem being studied
  - Number of items in the input (e.g. sorting)
  - Total number of bits (e.g. integer arithmetic)

# EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size
- Various ways of determining the size of input data
  - Depends on the problem being studied
  - Number of items in the input (e.g. sorting)
  - Total number of bits (e.g. integer arithmetic)
- Must have a model of the implementation technology

# EFFICIENCY/2

Random-access machine

- Generic one-processor, random-access model of computation

# EFFICIENCY/2

Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy

# EFFICIENCY/2

Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy
- Each instruction takes a constant amount of time

# EFFICIENCY/2

usually "nr of elements" and "nr of operations" is precise enough to understand performance

## Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy
- Each instruction takes a constant amount of time
- Data types: character, integer, and floating point

# Runtime of insertion sort/1

cost per execution

nr of executions

## Analysis of the algorithm

```
for i = 2 to n do
    j = i-1;
    t = A[i];
    while j>0 ∧ t<A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

best case: do not move elements

| | |
|---|---|
| $c_1$ | $n$ |
| $c_2$ | $n-1$ |
| $c_3$ | $n-1$ |
| $c_4$ | $\sum_{i=2}^{n} t_i$ |
| $c_5$ | $\sum_{i=2}^{n} (t_i - 1)$ |
| $c_6$ | $\sum_{i=2}^{n} (t_i - 1)$ |
| $c_7$ | $n-1$ |

- Execution of $k$-th line takes time $c_k$ (a constant)
- $t_i$: the number of times a nested line is executed

5 6 9 15

1 best case

⋮

n worst case

# Runtime of insertion sort/2

## Exact running time

Sum of products of rightmost columns (cost, times)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i$$
$$+ c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

# Runtime of insertion sort/2

$\sum_{i=2}^{n} 1 = n-1$

## Exact running time

Sum of products of rightmost columns (cost, times)

$1/i$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i$$
$$+ c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

$\sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1$

$= \frac{n(n+1)}{2} - 1$

## Some observations

- Running time seen as function of the input size $n$
- May depend on **which** input of size $n$ is given
- Such behavior is captured by the parameter $t_i$

# Runtime of insertion sort / 3

**Best case** running time:

Array is already sorted: $t_i = 1$

$$
\begin{aligned}
T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\
&= \underbrace{(c_1 + c_2 + c_3 + c_4 + c_7)}_{a} n - \underbrace{(c_2 + c_3 + c_4 + c_7)}_{b} \\
&= an + b
\end{aligned}
$$

linear in n

# Runtime of insertion sort / 3

**Best case** running time:

Array is already sorted: $t_i = 1$

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \end{aligned}$$

**Worst case** running time:

Array is in reverse sorted order: $t_i = i$

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + (c_5 + c_6)\frac{(n-1)n}{2} \;=\; \underbrace{an^2 + bn + c}_{\text{quadratic in } n} \end{aligned}$$

# Best, worst and average case/1

Worst vs average case

- Usually, one focuses on worst case running time

# Best, worst and average case/1

## Worst vs average case

- Usually, one focuses on worst case running time
- There are basically the following three reasons
  - Worst case running time provides an upper bound
    - **Guarantee** that algorithm will never take any longer
  - For some algorithms, worst case occurs fairly often
    - Example: searching for an element that is not present
  - Average case is often roughly as bad as worst case
    - Example: insertion sort is still quadratic on average

# Best, worst and average case/1

Worst vs average case

- Usually, one focuses on worst case running time
- There are basically the following three reasons
  - Worst case running time provides an upper bound
    - **Guarantee** that algorithm will never take any longer
  - For some algorithms, worst case occurs fairly often
    - Example: searching for an element that is not present
  - Average case is often roughly as bad as worst case
    - Example: insertion sort is still quadratic on average
- Might be quite difficult to find typical average case

# Best, worst and average case/2

For a specific size of input size $n$, investigate running times for different input instances:

# EXERCISE SL02-1

How can we modify almost any algorithm to have a good best-case running time?

- modify algo to first check the input for a special case and solve the special case
- chess (store solution to positions)
- valid and good approach (but it can be used for cheating in benchmarks)

best-case is not always a good metric

28.2

# Best, worst and average case/3

For inputs of all sizes:

# Best, worst and average case/4

## Concluding remarks

- Often, counting the number of iterations of the core (innermost) part is sufficient
  - No distinction between comparisons, assignments, etc.
  - Means that each of them has roughly the same cost
  - Does provide results with good/adequate precision
  - Simplifying abstraction: asymptotic analysis

# Best, worst and average case/4

## Concluding remarks

- Often, counting the number of iterations of the core (innermost) part is sufficient
  - No distinction between comparisons, assignments, etc.
  - Means that each of them has roughly the same cost
  - Does provide results with good/adequate precision
  - Simplifying abstraction: asymptotic analysis
- In certain cases the cost of a specific operation may dominate all other costs
  - Example: disk I/O versus RAM operations
- Today simple cost models that consider one (the "dominating") cost only have to be refined since their accuracy is insufficient.

# RUNTIME OF BINARY SEARCH/1

## Searching revisited

**Algo:** LinSearch2(A,v)

---

**Input:** sequence $A[1..n]$ of length $n$, value $v$
**Output:** either index $i$ such that $v = A[i]$ or `NIL`

i = 1;
**while** $i \leq n \wedge A[i] \neq v$ **do** i = i+1;
**if** $i \leq n$ **then return** i **else return** NIL;

# RUNTIME OF BINARY SEARCH/1

## Searching revisited

**Algo:** LinSearch2(A,v)

---

**Input:** sequence $A[1..n]$ of length $n$, value $v$
**Output:** either index $i$ such that $v = A[i]$ or `NIL`

i = 1;
**while** $i \leq n \wedge A[i] \neq v$ **do** i = i+1;
**if** $i \leq n$ **then return** i **else return** NIL;

- Running times:
  - worst case: $n$
  - average case: $n/2$
  - best case: 0

# RUNTIME OF BINARY SEARCH/1

## Searching revisited

**Algo:** LinSearch2(A,v)

**Input:** sequence $A[1..n]$ of length $n$, value $v$
**Output:** either index $i$ such that $v = A[i]$ or `NIL`

i = 1;
**while** $i \leq n \wedge A[i] \neq v$ **do** i = i+1;
**if** $i \leq n$ **then return** i **else return** NIL;

- Running times:
  - worst case: $n$
  - average case: $n/2$
  - best case: $0$
- Do things change if the sequence is sorted?

# Runtime of binary search/2

Sorted sequence

Consider middle element & narrow down the search space

# RUNTIME OF BINARY SEARCH/2

easy - - - - - - - - - difficult

## Sorted sequence

Consider middle element & narrow down the search space

## The algorithm

**Algo:** BinSearch1(A,v)

---

**Input:** sequence $A[1..n]$ of length $n$, value $v$
**Output:** either index $i$ such that $v == A[i]$ or `NIL`

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

# EXERCISE SL02-2

Illustrate the binary search algorithm for the following examples:

- A = [10, 20, 30, 40, 50, 60, 70], q = 30
- A = [10, 20, 30, 40, 50], q = 22

q = 30 u = 7
l = 1 r = 7 m = 4
l = 1 r = 3 m = 2
l = 3 r = 3

⇒ 3 ✓

q = 22 u = 5
l = 1 r = 5 m = 3
l = 1 r = 2 m = 1
l = 2 r = 2 m = 2
l = 3 r = 2

⇒ NIL ✓

# Runtime of binary search/3

Analysis of algorithm

- How many times is the loop executed?
  - Each iteration the difference between $r$, $l$ is cut in half
    - Initially the difference is $n$
    - Stops when it becomes $0$
  - How many times do we have to cut $n$ in half to get $0$?
  - $\log_2 n$ — better than brute-force linear search ($n$)

# RUNTIME OF BINARY SEARCH/3

Analysis of algorithm

- How many times is the loop executed?
  - Each iteration the difference between $r$, $l$ is cut in half
    - Initially the difference is $n$
    - Stops when it becomes $0$
  - How many times do we have to cut $n$ in half to get $0$?
  - $\log_2 n$ — better than brute-force linear search ($n$)
- Cost of linear versus cost of binary search
  - For $n = 0.2M$ one gets $\log_2 200'000 \approx 18$
  - For $n = 2M$ one gets $\log_2 2'000'000 \approx 21$
  - For $n = 20M$ one gets $\log_2 20'000'000 \approx 24$

log grows very slowly

# EXERCISE SL02-3

The inner loop of the insertion sort algorithm uses a linear search to search backward through the sorted array a[1..j-1] to find the place where to insert the new element. Can we use a binary search instead to improve the worst case running time of insertion sort?

- we can use binary search to find the position fast

- but we still have to move all elements larger than the one at position i

=> it does not improve performance

# EXERCISE SL02-4/1

Let $A_1$ be an unsorted array of length $n_1$ and $A_2$ a sorted array of length $n_2$. Both $A_1$ and $A_2$ store integers and do not contain duplicates.

Provide an algorithm that counts the number of integers in $A_1$ that are also in $A_2$. That is, given $A_1 = [5, 3, 7, 1, 9]$ and $A_2 = [1, 2, 3, 4]$, `cntOcc` ($A_1$, $A_2$) returns 2. The ~~asymptotic~~ complexity of your solution must be strictly less than $O(n_1 n_2)$. Furthermore, the arrays may not be changed and your solution may not use more than a constant amount of additional memory. Determine the ~~asymptotic~~ running time of your solution.

```
cnt = 0
for i=1 to n1 do
    for j=1 to n2 do
        if A1[i] = A2[j] then cnt++
```

complexity (# of comparisons) $= n_1 \cdot n_2$

# EXERCISE SL02-4/2

Alg: crtCec(A1, A2)

```
crt = 0
for i = 1 to n1 do
    if (Bi-Search(A2, A1[i]) ≠ NIL) then
        crt++
return crt;
```

runtime: $\sum_{i=1}^{n1} \log_2 n2 =$

$n1 \cdot \log_2 n2$

# EXERCISE SL02-4/3

# Table of contents — SL02

1. Algorithmic complexity

2. Correctness
   Correctness notions, assertions, invariants
   Correctness of binary search
   Correctness of insertion sort

3. Asymptotic complexity

4. Special case analysis

# CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input

# CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input
- Automatic proof of correctness is not possible

# CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input
- Automatic proof of correctness is not possible
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms

# CORRECTNESS/2

Partial versus total

$$\boxed{\text{legal input}} \rightsquigarrow \boxed{\text{algorithm}} \overset{\Downarrow}{\rightsquigarrow} \boxed{\text{desired output}}$$

# CORRECTNESS/2

Partial versus total

$$\boxed{\text{legal input}} \quad \rightsquigarrow \quad \boxed{\text{algorithm}} \quad \overset{\Downarrow}{\rightsquigarrow} \quad \boxed{\text{desired output}}$$

- **Partial correctness**
  If the point indicated with $\Downarrow$ is reached, then one gets the desired output

# CORRECTNESS/2

Partial versus total

$$\boxed{\text{legal input}} \rightsquigarrow \boxed{\text{algorithm}} \overset{\Downarrow}{\rightsquigarrow} \boxed{\text{desired output}}$$

- **Partial correctness**
  If the point indicated with $\Downarrow$ is reached, then one gets the desired output
- **Total correctness**
  Indeed the point indicated with $\Downarrow$ is reached, and one gets the desired output

total correctness =
partial correctness
+
termination

# CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence

# CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm

# CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm
- **Preconditions**: assertions that must be valid before the execution of an algorithm or a subroutine (input)

# CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm
- **Preconditions**: assertions that must be valid before the execution of an algorithm or a subroutine (input)
- **Postconditions**: assertions that must be valid after the execution of an algorithm or a subroutine (output)

# CORRECTNESS/4

A=[5,5,5]
A=[5]
A=[5,7,5,5,8]
A=[ ]

precise?

Example: Specify pre- and postconditions for an algorithm that determines the two smallest numbers in a sequence of numbers (given as an array).

# CORRECTNESS/4

A=[5] m1=5 m2=5
A=[5,5,5] m1=5 m2=5
A=[5,7,5,5,8] m1=5, m2=7
A=[] illegal input

Example: Specify pre- and postconditions for an algorithm that determines the two smallest numbers in a sequence of numbers (given as an array).

Precondition (INPUT): an array of integers $A[1..n]$, $n > 0$

Postcondition (OUTPUT): ($m1 \in A$, $m2 \in A$) such that either

- $m1 < m2$ if
  - $\forall i \in [1..n] : m1 \leq A[i]$
  - $\forall i \in [1..n] : A[i] \neq m1 \Rightarrow m2 \leq A[i]$, or
- $m2 = m1 = A[1]$ if
  $\forall i, j \in [1..n] : A[i] = A[j]$

m1: smaller equal than all numbers in A

m2: smaller equal than all number in A different from m1

m1 = m2 if all numbers are the same

# CORRECTNESS/5

Loop invariants

- To show correctness of loop statements

# CORRECTNESS/5

Loop invariants

- To show correctness of loop statements
- **Invariants**: assertions that are valid any time they are reached

# CORRECTNESS/5

## Loop invariants

- To show correctness of loop statements
- **Invariants**: assertions that are valid any time they are reached
- One must show three things about loop invariants
  - Initialization: it is true before the first iteration
  - Maintenance: if it is true before an iteration then it is true after that iteration
  - Termination: when the loop terminates, it gives a useful property that helps to show that the algorithm is correct

# CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if $v$ is not in $A$.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

# CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if $v$ is not in $A$.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$, and

all elements left of l are smaller than v

# CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if $v$ is not in $A$.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$, and
- $\forall i \in [r+1..n] : v < A[i]$ all elements to right of r are larger than v

# Correctness of binary search/2

Invariant:

- $\forall i \in [1..l-1]\ v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

**Initialization**: $l = 1, r = n$

The invariant holds because there are no elements to the left of $l$ and no elements to the right of $r$.

- $l = 1$ yields $\forall i \in [1..0] : A[i] < v$. This holds since the range [1..0] is empty.
- $r = n$ yields $\forall i \in [n+1..n] : A[i] > v$. This holds since the range [n+1..n] is empty.

# Correctness of binary search/3

r = m? OK (invariant holds)

r = m-2? not OK (one element, we know nothing about; invariant is not guaranteed)

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

**Maintenance**: $l, r, m = \lfloor(l+r)/2\rfloor$

Two cases:

- We have $A[m] \neq v$, $A[m] > v$, $r = m - 1$, A is sorted. This implies $\forall k \in [r+1..n] : A[k] > v$.
- We have $A[m] \neq v$, $A[m] < v$, $l = m + 1$, A is sorted. This implies $\forall k \in [1..l-1] : A[k] < v$.

# Correctness of binary search/4

r = m?
$\lfloor (l+r)/2 \rfloor \not< r$

Ex: A=[7], v=5
l=1, r=1, m=1
l=1, r=1
⇒ ∞ loop

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

now l is larger than l

**Termination**: $l, r, l \leq r$

Two cases:

- $l = m + 1$ allows to conclude $\lfloor (l+r)/2 \rfloor + 1 > l$
- $r = m - 1$ allows to conclude $\lfloor (l+r)/2 \rfloor - 1 < r$

at least l

⇒ our range gets smaller

Thus, the range gets smaller during each iteration and the loop will terminate when $l \leq r$ no longer holds.

now r is smaller than r

at most r

bin search with duplicates:
1 4 4 4 4 4
r=6

3.3

# EXERCISE SL02-5/1

1. Formulate the loop invariant for the outer loop of the selection sort algorithm.

=

```
for i = 1 to n-1 do
    k = i
    for j = i+1 to n do
```

n?

- $\forall j \in [1 .. i-2] : A[j] \leq A[j+1]$
- $\forall j \in [1 .. i-1] \; \forall k \in [i .. n] : A[j] \leq A[k]$

# EXERCISE SL02-5/2

2. What happens if in the selection sort algorithm the outer loop runs up to $n$?

- in last step of the loop we exchange the element with itself

- it is OK (last iteration is not needed)

# FOR LOOPS

up loop:

for i=1 to 5 5X
1 to 2 2X
1 to 1 1X
1 to 0 0X
10 to 1 0X

- The semantics of statements must be specified precisely for all cases. Consider `for i = 2 to` $n$ as an example.
- We have an **up loop** if the loop runs from low to high:
  - Example: Array A[1..n]; for j=1 to n do ...
  - If the start value of an index (start) is higher than the end value of an index (end) an up loop is executed zero times
  - Otherwise an up loop is executed (end - start + 1) times
- We have a **down loop** if the loop runs from high to low:
  - Example: Array A[1..n]; for j=n to 1 do ...
  - If the start value of an index (start) is lower than the end value of the index (end) a down loop is executed zero times
  - Otherwise a down loop is executed (start - end + 1) times
- The value of the loop variable is undefined after the loop.

# Correctness of insertion sort/1

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

# Correctness of insertion sort/1

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Loop invariant

- Outer loop: $\begin{cases} A[1 \ldots i-1] \text{ sorted} \\ A[1 \ldots i-1] \in A^{orig} \end{cases}$

# Correctness of insertion sort/1

outer loop

inner loop

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Loop invariant

- Outer loop: $\begin{cases} A[1 \ldots i-1] \text{ sorted} \\ A[1 \ldots i-1] \in A^{orig} \end{cases}$
- Inner loop: $\begin{cases} \overbrace{A[1 \ldots j]}^{a3}, \overbrace{t}^{a2}, \overbrace{A[j+1 \ldots i-1]}^{a1} \\ t < A[k] \text{ for } j+1 \leq k \leq i-1 \\ A[1 \ldots j] \circ A[j+1 \ldots i-1] \text{ sorted} \end{cases}$

# Correctness of insertion sort/2

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

**Initialization**:

- $i = 2$: invariant holds; A[1..1] is trivially sorted
- $j = i-1$: $A[1..i-1], t, A[i..i-1], t = A[i]$
  $A[i..i-1]$ is empty (and thus trivially sorted)
  $A[1..i-1]$ is sorted (invariant of outer loop)

large

all

empty

# Correctness of insertion sort/3

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

**Maintenance**:

- $A[1..i-1]$ is sorted + insert $A[i]$ implies $A[1..i]$ is sorted outer loop
- $A[1..j-1], t, A[j], j+1..i-1]$ satisfies condition since $A[j] > t$ and $A[1..i-1]$ is sorted inner loop

# Correctness of insertion sort/4

case 1 (if cond is false)

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

case 2 (cond is false)

**Termination**:

correct order

- outer loop: $i = n+1$ and $A[1..n]$ is sorted
- $A[j] \leq t$: $A[1..j], t, A[j+1..i-1] = A[1..i]$ is sorted case 2
- $j = 0$: $A[t, 1..i-1] = A[1..i]$ is sorted case 1

# Table of contents — SL02

1. Algorithmic complexity

2. Correctness

3. Asymptotic complexity
   Mathematics refresher
   Order of growth

4. Special case analysis

# Asymptotic complexity

## Asymptotic analysis

- Goal: simplify the analysis of running time

# Asymptotic complexity

## Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$

# Asymptotic complexity

$n \to \infty$ ($n$ very large)

## Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$
- Fundamental concern: how the running time increases with the size of the input in the limit

# Asymptotic complexity

## Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$
- Fundamental concern: how the running time increases with the size of the input in the limit
- Asymptotically more efficient algorithms are best for all but small inputs

# Mathematics refresher/1

## Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$

# Mathematics refresher/1

## Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$
- Arithmetic series
  - $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

# Mathematics refresher/1

Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$
- Arithmetic series
  - $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

# MATHEMATICS REFRESHER/2

- Geometric series
  - given an integer $n_0$ and a real number $0 < a \neq 1$:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

# MATHEMATICS REFRESHER/2

- Geometric series
  - given an integer $n_0$ and a real number $0 < a \neq 1$:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - given an integer $n_0$ and a real number $|a| < 1$:

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

# Mathematics refresher/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$

# Mathematics refresher/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$
- Inductive hypothesis: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$

# Mathematics refresher/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$
- Inductive hypothesis: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$
- Inductive step (uses inductive hypothesis):

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

inductive hypothes

# Mathematics refresher/4

## Logarithms and powers

- $\log_a(bc) = \log_a b + \log_a c$
- $\log_a b = \log_c b / \log_c a$
- $\log_a b^c = c \log_a b$
- $a^m a^n = a^{m+n}$
- $a^{mn} = (a^m)^n$
- $a^{\log_a b} = b$
- $\lg b = ldb = \log_2 b$
- Notation: $\log_a b = {}^a\log b$

$\boxed{{}^2\log n}$

$= {}^5\log n \,/\, {}^5\log 2$ $= c \approx 0.43$

$= \boxed{c' \cdot {}^5\log n}$

$\Rightarrow$ changing the base of log $\equiv$ constant factor in the complexity

# Mathematics refresher/5

polynomial complexity
$n^k$ for a positive $k$

Complexities of common functions
From $n = 10$ up to $n = 1M$

logarithmic compl

different time complexity classes

exponential

quadratic

low → high

linear

| $\log n$ | $\log^2 n$ | $\sqrt{n}$ | $n$ | $n \log n$ | $n \log^2 n$ | $n^{3/2}$ | $n^2$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 3 | 10 | 30 | 90 | 30 | 100 | 1e4 | 4e6 |
| 6 | 36 | 10 | 100 | 600 | 3600 | 1000 | 1e4 | 1e30 | 9e157 |
| 9 | 81 | 31 | 1000 | 9000 | 8e5 | 3e4 | 1e6 | 1e301 | $\infty$ |
| 13 | 169 | 100 | 1e4 | 1e5 | 2e6 | 1e6 | 1e8 | $\infty$ | $\infty$ |
| 16 | 256 | 316 | 1e5 | 2e6 | 3e7 | 3e7 | 1e10 | $\infty$ | $\infty$ |
| 19 | 361 | 1000 | 1e6 | 2e7 | 4e8 | 1e9 | 1e12 | $\infty$ | $\infty$ |

binary search

linear search

Bubble sort

very problematic

# Order of growth/1

Big-$O$ notation

- Asymptotic upper bound

# Order of growth/1

Big-$O$ notation

- Asymptotic upper bound
- Used for worst-case analysis

# Order of growth/1

Big-$O$ notation

- Asymptotic upper bound
- Used for worst-case analysis
- $f(n)$ and $g(n)$ are functions over non-negative integers

# Order of growth/1

## Big-$O$ notation

- Asymptotic upper bound
- Used for worst-case analysis
- $f(n)$ and $g(n)$ are functions over non-negative integers
- $f(n) = O(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

g is an upper bound for f

a constant multiple of g(n) is an upper bound for f(n)

# EXERCISE SL02-6

Prove or disprove the following statements:

1. $2^{n+1} = O(2^n)$
2. $\cancel{2^{2^n}} = O(2^n)$ $(2^2)^n$
3. $2n^{1.5} = O(n \lg n)$

1. $2^{n+1} \leq c\,2^n \quad \forall n \geq n_0$

$2 \cdot 2^n \leq c\,2^n$

holds for $c=2$, $n_0=1$

2. $(2^2)^n$

$2^{2^n} \leq c\,2^n$

$2^{2n} \leq c\,2^n$

$2^n \cdot 2^n \leq c\,2^n$

$2^n \leq c$ — constant

exists $n_0$

3. $2n^{3/2} \leq c\,n \log n$

$2\sqrt{n^3} \leq c\,n \log n$

$n^3 \leq \frac{c^2}{4} n^2 \log^2 n$

$n \leq \left(\frac{c^2}{4}\right) \log^2 n$ $\quad n_0$

# Order of growth/2

## Big-Ω notation

- Asymptotic lower bound

# Order of growth/2

Big-Ω notation

- Asymptotic lower bound
- Used for best-case analysis

# Order of growth/2

## Big-Ω notation

- Asymptotic lower bound
- Used for best-case analysis
- $f(n) = \Omega(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $cg(n) \leq f(n)$ for $n \geq n_0$

g is a lower bound for f

# Order of growth/2

## Big-Ω notation

- Asymptotic lower bound
- Used for best-case analysis
- $f(n) = \Omega(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $cg(n) \leq f(n)$ for $n \geq n_0$

- Little-$o/\omega$: non-tight analogues of big-oh/omega

$\leq \quad \rightsquigarrow \quad <$

# Order of growth/3

Big $\Theta$ notation

- Asymptotic tight bound

# Order of growth/3

## Big $\Theta$ notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$

g is a tight bound for f

# Order of growth/3

## Big $\Theta$ notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$

- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

# Order of growth/3

## Big $\Theta$ notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$

- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is sometimes (ab)used as notation for $\Theta(f(n))$

(do not do this)

# Exercise SL02-7

$4^{\log n / \log 10} = n^{\log 10} \approx n^{0.6}$

Determine the asymptotic complexities of the formulas: $(\Theta)$

1. $0.001n^2 + 70'000n$ $\Theta(n^2)$
2. $2^n + n^{10'000}$ $\Theta(2^n)$
3. $n^k + c^n$ $\Theta(c^n)$ $c > 1$
4. $\log_k n + n^e$ $\Theta(n^e)$
5. $2^n + 2^{n/2}$ $\Theta(2^n)$
6. $n^{\log c} + c^{\log n}$ $= \Theta(n^{\log c})$ $= \Theta(c^{\log n})$
7. $2n^3 + 100n \log n + 5$ $\Theta(n^3)$
8. $30 \log n + \log 2^n$ $\Theta(n)$ $(n \log 2)$
9. $4^{\log n} + n^2$ $\Theta(n^2)$

$n^{\log c} =$

$n^{\log c / \log x} =$

$c^{1/\log x} =$

$c^{\log n / \log x} =$

$c^{\log n}$

# Order of growth/4

Useful analogy

- $f(n) = O(g(n)) \quad \leftrightarrow \quad f \leq g$
- $f(n) = o(g(n)) \quad \leftrightarrow \quad f < g$
- $f(n) = \Omega(g(n)) \quad \leftrightarrow \quad f \geq g$
- $f(n) = \omega(g(n)) \quad \leftrightarrow \quad f > g$
- $f(n) = \Theta(g(n)) \quad \leftrightarrow \quad f = g$

# ORDER OF GROWTH/4

$O(3n+5) = \{4n, 60n, 5n, 100, \sqrt{n}, \log n, \ldots\}$

Useful analogy

- $f(n) = O(g(n)) \quad \leftrightarrow \quad f \leq g$
- $f(n) = o(g(n)) \quad \leftrightarrow \quad f < g$
- $f(n) = \Omega(g(n)) \quad \leftrightarrow \quad f \geq g$
- $f(n) = \omega(g(n)) \quad \leftrightarrow \quad f > g$
- $f(n) = \Theta(g(n)) \quad \leftrightarrow \quad f = g$

- Abuse of notation: $\boxed{f(n) = O(g(n))}$ actually means $f(n) \in O(g(n))$ since $O(g(n))$ is the set of all functions for which there exists constants $c > 0$ and $n_0 > 0$ such that ...

$O(g(n)) = \{ h(n) : h(n) \leq c\, g(n) \}$

= set of fcts for which $g(n)$ is an upper bound

# Order of growth/5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

# Order of growth/5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Lower-order terms are relatively insignificant

# Order of growth/5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Lower-order terms are relatively insignificant
- Although $50n \log n$ is $O(n^5)$, it is expected that an approximation is of smallest possible order

# Comparison of Running Times

$400n = 3600 \cdot 10^6 \mu s$

exceptions for small n

Determining the maximal problem size:

| Running Time T(n) in $\mu s$ | 1 second | 1 minute | 1 hour |
|---|---|---|---|
| $400n$ | 2500 | 150'000 | 9'000'000 |
| $20nlogn$ | 4096 | 166'666 | 7'826'087 |
| $2n^2$ | 707 | 5477 | 42'426 |
| $n^4$ | 31 | 88 | 244 |
| $2^n$ | 19 | 25 | 31 |

$a^n$

problem cannot be solved by increasing the runtime

$2^{2^n} = (2^2)^n$

7.3

# EXERCISE SL02-8/1

Provide an algorithm that selects the $k$-th smallest integer from an unsorted array $A$ of length $n$ that does not contain duplicates. For instance, given $k = 3$ and $A = [12, 4, 10, 8, 9, 33, 2, 18]$, `select_kth` $(A, k)$ returns 8. Determine the asymptotic complexity of your algorithm.

idea: (1) find k times the smallest element and remove it (k-1 times) (using linear search)

⟹ (2) sort (selection sort). in $k^{th}$ round we find the kth smallest element

(3) Bubble sort where smallest element goes to left (we do k steps)

# EXERCISE SL02-8/2

$k=1 \Rightarrow \Theta(n)$

$\vdots$

$k=n \Rightarrow \Theta(n^2)$

Algo: selectKth(A, k)

```
for i=1 to k do
    m := i
    for j=i+1 to n do
        if A[j] < A[m] then m=j
    exchange A[i] and A[m]
return A[k]
```

# of exec of innermost part:

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

$$= \Theta(kn)$$

# Table of contents — SL02

1. Algorithmic complexity

2. Correctness

3. Asymptotic complexity

4. Special case analysis
   - Special cases
   - Correctness of binary search variant

# Special case analysis/1

Basic approach for checking the correctness of a program with special case analysis:

- Consider all possible extreme cases

# Special case analysis/1

Basic approach for checking the correctness of a program with special case analysis:

- Consider all possible extreme cases
- Make sure the solution works in these cases

# Special case analysis/1

Basic approach for checking the correctness of a program with special case analysis:

- Consider all possible extreme cases
- Make sure the solution works in these cases
- Problem is reduced to identifying all the special cases

# Special case analysis/1

Basic approach for checking the correctness of a program with special case analysis:

- Consider all possible extreme cases
- Make sure the solution works in these cases
- Problem is reduced to identifying all the special cases
- Related to problem specification (input/output relation)

# Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled, duplicates, etc.

# Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled, duplicates, etc.
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.

# Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled, duplicates, etc.
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.
- Calling of functions (procedures)
  - Entering function, termination of function

# Special case analysis/2

fac(60)

→ fac(59)

fac(59)

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled, duplicates, etc.
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.

fac(0) →

- Calling of functions (procedures)
  - Entering function, termination of function
- Control and loop statements
  - Start of loop, end of loop, 1st iteration

# Special case analysis/3

## Simple example

**Algo:** Sorted1(A)

---

**Input:** array $A[1..n]$ of integers
**Output:** TRUE if $A$ is sorted, FALSE otherwise

**for** $i = 1$ **to** $n$ **do**
  **if** $A[i] \geq A[i+1]$ **then return** FALSE;
**return** TRUE

# Special case analysis/3

## Simple example

**Algo:** Sorted1(A)

---

**Input:** array $A[1..n]$ of integers
**Output:** TRUE if $A$ is sorted, FALSE otherwise

**for** $i = 1$ **to** $n$ **do**
  **if** $A[i] \geq A[i+1]$ **then return** FALSE;
**return** TRUE

- Loop special cases:
  - $i = 1$ (start)
  - $i = n$

# Special case analysis/3

- A = [1, 1, 2, 3]
⇒ FALSE

## Simple example

**Algo:** Sorted1(A)

**Input:** array $A[1..n]$ of integers
**Output:** TRUE if $A$ is sorted, FALSE otherwise

**for** $i = 1$ **to** $n$ **do**
  **if** $A[i] \geq A[i+1]$ **then return** FALSE;
**return** TRUE

- Loop special cases:
  - $i = 1$ (start)
  - $i = n$
- Data structure special cases:
  - empty array
  - single element array

- end of the loop, $i = n$
⇒ $A[n] \geq A[n+1]$
↯

# Special case analysis/4

**Algo:** Sorted2(A)

---

**Input:** array $A[1..n]$ of integers
**Output:** TRUE if $A$ is sorted, FALSE otherwise

**for** *i = 1* **to** *n-1* **do**
  **if** *A[i]>A[i+1]* **then return** FALSE;
**return** TRUE

# SPECIAL CASE ANALYSIS/4

**Algo:** Sorted2(A)

**Input:** array $A[1..n]$ of integers
**Output:** TRUE if $A$ is sorted, FALSE otherwise

**for** $i = 1$ **to** $n$-$1$ **do**
  **if** $A[i]>A[i+1]$ **then return** FALSE;
**return** TRUE

- start of loop, i=1: A[1] > A[2] ok
- end of loop, i=n-1: A[n-1] > A[n] ok
- first iteration from i=1 to i=2: A[2] > A[3] ok
- empty array, n=0, A=[]: for i=1 to -1 ok ✓
- one element array, n=1, A=[1]: for i=1 to 0 ok ✓
- two element array, n=2, A=[1,0]: for i=1 to 1 ok A[1] > A[2] ✓
- A=[1,1,1], A=[1,2,3], A=[-1,0,1,-3]: ✓
- A=[3,2,1]: FALSE ??

# CORRECTNESS OF BINARY SEARCH VARIANT/1

a. l=1 r=2 ] m=1
l=1 r=0 |
NIL ✓

b. l=1 r=2
1 ✓

d. l=1 r=2
l=2 r=2
NIL ✗

Use special cases analysis to analyze the following algorithm.

**Algo:** BinSearch2(A,v)

l = 1; r = n; l=1, r=2
**repeat**
  m = ⌊(l+r)/2⌋;
  **if** *A[m]==v* **then return** m;
  **else if** *A[m]>v* **then** r = m-1;
  **else** l = m+1;
**until** *l≥r*;
**return** NIL;

a. A=[5,7], v=3
b. A=[5,7], v=5
c. A=[5,7], v=6
d. — " — v=7
e. — " — v=8

– A with odd # of elements / A is empty –

# Correctness of binary search variant/2

a. A=[5,7], v=3

l=1 r=2

l=1 r=1

A[0] out of bounds

d. v=7

l=1 r=2

l=2 r=2

NIL

Use special cases analysis to analyze the following algorithm.

**Algo:** BinSearch3(A,v)

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

# CORRECTNESS OF BINARY SEARCH VARIANT/3

a. $A = [5, 7]$, $v = 3$

$l = 1$ $r = 2$

$l = 1$ $r = 1$

$l = 1$ $r = 1$

$\infty$ loop

Use special cases analysis to analyze the following algorithm.

**Algo:** BinSearch3(A,v)

```
l = 1; r = n;
while l≤r do
    m = ⌊(l+r)/2⌋;
    if A[m]≤v then l = m+1 else r = m;
if A[l-1]==v then return l-1 else return NIL;
```

# EXERCISE SL02-9

Use special case analysis to determine the correctness of BinarySearch4 and BinarySearch5.

**Algo:** BinSearch4(A,q)

```
l = 1; r = n;
while l≤r do
    m = ⌊(l+r)/2⌋;
    if A[m]==q then return m;
    else if A[m]>q then r = m-1;
    else l = m+1;
return NIL;
```

**Algo:** BinSearch5(A,q)

```
l = 1; r = n;
while l<r do
    m = ⌊(l+r)/2⌋;
    if A[m]==q then return m;
    else if A[m]>q then r = m-1;
    else l = m+1;
return NIL;
```

✓

f

d. A=[5,7], v=7

l=1 r=2

l=2 r=2

NIL f

# Outlook

This week

- Algorithmic complexity
- Correctness of algorithms
- Asymptotic analysis
- Special case analysis

# Outlook

This week

- Algorithmic complexity
- Correctness of algorithms
- Asymptotic analysis
- Special case analysis

Next week

- Divide and conquer
- Merge sort, tiling
- Recurrences

# Informatik II

## Divide and Conquer, Recurrences
## — SL03 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

# TABLE OF CONTENTS — SL03

principle
(how to program,
solve problems)

1. Divide and conquer
   Merge sort
   Tromino tiling

2. Recurrences

become difficult
practice

analysis of the runtime
(mechanical, simpler, recipes)

# Divide and conquer/1

Designing algorithms

- Recursive algorithms: divide-and-conquer approach

# Divide and conquer/1

Designing algorithms

- Recursive algorithms: divide-and-conquer approach
- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. **Divide** problem into a number of subproblems
  2. **Conquer** subproblems by solving them recursively
  3. **Combine** solutions of the subproblem into solution for the original problem

# Divide and conquer/2

## Finding decomposition

- Requires some practice and is the key part

# Divide and conquer/2

## Finding decomposition

- Requires some practice and is the key part
- Fundamental properties of a decomposition
  - It reduces the problem to smaller problems
  - Often, these smaller problems are the same as the original problem
  - A sequence of decompositions eventually yields the base case (trivial solution)
  - It must contribute to solving the original problem

# Merge sort/1

Rough idea

- Closely follows the divide-and-conquer paradigm

# Merge sort/1

## Rough idea

- Closely follows the divide-and-conquer paradigm
- To sort an $n$-element sequence proceed as follows
  - **Divide**: divide the sequence into two $n/2$-element sequences
  - **Conquer**: sort the two sequences recursively using merge sort
  - **Combine**: merge the two sorted sequences to produce the solution

# Merge sort/1

1 5 6 10 | 7 8 11 20

## Rough idea

- Closely follows the divide-and-conquer paradigm
- To sort an $n$-element sequence proceed as follows
  - **Divide**: divide the sequence into two $n/2$-element sequences
  - **Conquer**: sort the two sequences recursively using merge sort
  - **Combine**: merge the two sorted sequences to produce the solution
- Recursion stops when the sequence that must be sorted has length 1.

# Merge sort/2

Base case: array of size 1, i.e., $l = r$

The algorithm

**Algo:** MergeSort(A,l,r)

---

**Input:** array A[l..r]
**Output:** permuted array A[l..r] that is sorted in increasing order

**if** $l<r$ **then** more than 1 element

```
    m = ⌊(l+r)/2⌋;
    MergeSort(A,l,m);
    MergeSort(A,m+1,r);
    Merge(A,l,r,m);
```

$l = 1$ $m = 2$ $r = 3$

# Merge sort/3

Sorting $\langle 5, 2, 4, 7, 1, 3, 6, 2 \rangle$: overview

# Merge sort/4

Goal: $[7, 14, 15, 20], [3, 6, 16, 22] \Rightarrow [3, 6, 7, 14, 15, 16, 20, 22]$

**Algo:** Merge(A,l,r,m)

---

**Input:** array A; indexes l, r and m from MergeSort
**Output:** A[l..r] is sorted in increasing order

```
for i = l to m do B[i] = A[i];
for i = m+1 to r do B[r+m-i+1] = A[i];
i = l; j= r;
for k = l to r do
    if B[i]<B[j] then A[k] = B[i]; i = i+1;
    else A[k] = B[j]; j = j-1;
```

The following slides show the state at the beginning of the third for loop.

# Merge sort/5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

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

# Merge sort/5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

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

- $i = 1, j = 8, k = 1$

$B =$

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

# Merge sort/5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

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

- $i = 1, j = 8, k = 1$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| . | . | . | . | . | . | . | . |

# Merge sort/5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

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

- $i = 1, j = 8, k = 1$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| . | . | . | . | . | . | . | . |

- $i = 1, j = 7, k = 2$

$B =$

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

# Merge sort/5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

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

- $i = 1, j = 8, k = 1$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| . | . | . | . | . | . | . | . |

- $i = 1, j = 7, k = 2$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | . | . | . | . | . | . | . |

# Merge sort/6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

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

# Merge sort/6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | . | . | . | . | . | . |

# Merge sort/6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | . | . | . | . | . | . |

- $i = 2, j = 6, k = 4$

$B =$

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

# Merge sort/6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | i | | | | | j | | |
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | . | . | . | . | . | . |
| | | | k | | | | | |

- $i = 2, j = 6, k = 4$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | i | | | | j | | |
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | 2 | . | . | . | . | . |
| | | | | k | | | | |

# Merge sort/7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

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

# Merge sort/7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | . | . | . | . |

# Merge sort/7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | . | . | . | . |

- $i = 3, j = 5, k = 6$

$B =$

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

# MERGE SORT/7

1 2 3 7 | 2 18 17 3

if all elem from left part have been used then i points to largest element of right part ⇒ we take always from right

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | 2 | 3 | . | . | . | . |

- $i = 3, j = 5, k = 6$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | 2 | 3 | 4 | . | . | . |

# Merge sort/8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

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

# Merge sort/8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 4 | 5 | . | . |

# Merge sort/8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 4 | 5 | . | . |

- $i = 4, j = 4, k = 8$

$B =$

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

# Merge sort/8

j i
3 5 5 5 | 5 5 5 7

equal element across middle
"belong" to right part; correct
because we
stop after
n elements

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | 2 | 3 | 4 | 5 | . | . |

- $i = 4, j = 4, k = 8$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $B =$ | 2 | 4 | 5 | 7 | 6 | 3 | 2 | 1 |
| $A =$ | 1 | 2 | 2 | 3 | 4 | 5 | 6 | . |

# Merge sort/8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 4 | 5 | . | . |

- $i = 4, j = 4, k = 8$

$B =$

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

$A =$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 3 | 4 | 5 | 6 | . |

- After last iteration of for loop:

$A =$

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

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# Merge sort/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]

# EXERCISE SL03-1

$l = 1 \Rightarrow m = 2$
$r = 3$ $A[l..m]$ $A[m+1..r]$ $\lfloor (l+r)/2 \rfloor = m$

Assume $A = [4, 5, 3, 2, 5, 9, 4]$. Show the content of the array after the fourth call of Merge during MergeSort.

$\Rightarrow$ 2 3 4 5 5 9 4

# MERGE SORT / 10

## Running time

- Recursive algorithms: $T(n)$ described by a recurrence

runtime of problem of size n

cost to divide

cost to combine

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

# of subproblems

size of subproblems

- $n \leq c$: base case
- $a$ subproblems of size $1/b$ each
- $D(n)$: to divide problem
- $C(n)$: to combine solutions

# MERGE SORT/10

## Running time

- Recursive algorithms: $T(n)$ described by a recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

  - $n \leq c$: base case
  - $a$ subproblems of size $1/b$ each
  - $D(n)$: to divide problem
  - $C(n)$: to combine solutions

merge: look over all elements

- Merge sort: $a = b = 2$, $D(n) = \Theta(1)$, $C(n) = \Theta(n)$

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

# Merge sort/11

## The recurrence/1

- Assume that $n > 1$ is an exact power of 2

# Merge sort/11

$\Theta(n)$

$\Theta(2n) = \Theta(n)$

$\Theta(3n) = \Theta(n)$

$\Theta(\log n \cdot n)$

## The recurrence/1

- Assume that $n > 1$ is an exact power of 2
- $T(n) = 2T(n/2) + cn$ yields $T(n) = \Theta(n \lg n)$

# MERGE SORT/12

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

## The recurrence/2

$n/2$

- Alternatively, one may proceed as follows
  - For $n > 1$, let $\boxed{T(n) = 2T(n/2) + cn}$
  - Apply repeated backward substitution

$$T(n) = 2T(n/2) + cn \qquad \text{(substitute)}$$

# Merge sort/12

## The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)} \\
&= 2(\underbrace{2T(n/4) + cn/2}) + cn && \text{(expand)}
\end{aligned}
$$

# Merge sort/12

## The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)} \\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\
&= 4T(n/4) + 2cn && \text{(substitute)}
\end{aligned}
$$

# Merge sort/12

$T(n/4) = 2T(n/8) + c\,n/4$

## The recurrence/2

$n/4$

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}
\end{aligned}
$$

# Merge sort/12

## The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

$2^3$ $2^3$ $3$

$i =$ after 3 steps

# Merge sort/12

$\frac{n}{2^i} = 1$

$i = {}^2\log n = i_{max}$

## The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

- Note that $\boxed{T(n) = 2^i T(n/2^i) + icn}$ where $i_{max} = \lg n$

$$2^{\lg n}\underbrace{T(n/2^{\lg n})}_{1} + cn\lg n = n\underbrace{T(1)}_{\Theta(1)} + \boxed{cn\lg n} = \Theta(n\lg n)$$

$\Theta(n)$

# EXERCISE SL03-2

$T(1) = 1$

Solve the recurrence $\boxed{T(n) = 2T(n/2) + \Theta(1).}$

$$\frac{n}{2^i} = 1$$

$$i = \log_2 n = i_{max}$$

$$T(n) = 2T(n/2) + \Theta(1)$$
$$= 2(2T(n/4) + \Theta(1)) + \Theta(1)$$
$$= 2^2 T(n/4) + (2+1)\Theta(1)$$
$$= 2^2(2T(n/8) + \Theta 1) + (2+1)\Theta(1)$$
$$= 2^3 T(n/8) + (2^2 + 2^1 + 2^0)\Theta(1)$$
$$\boxed{= 2^i T(n/2^i) + (2^i - 1)\Theta(1)}$$
$$= 2^{\log n} T(n/n) + (2^{\log n} - 1)\Theta(1)$$
$$= n + n\Theta(1) - \Theta(1) = \Theta(n)$$

# Tromino tiling/1

## Problem description

- Tromino tiles

# Tromino tiling/1

## Problem description

- Tromino tiles
- Provide a tromino tiling for a $2^n \times 2^n$ board with a hole

# Tromino tiling/1

## Problem description

- Tromino tiles
- Provide a tromino tiling for a $2^n \times 2^n$ board with a hole

# TROMINO TILING/2

Trivial case: $n = 1$

- Tromino tiling for a $2^1 \times 2^1$ board with a hole

# TROMINO TILING/2

Trivial case: $n = 1$

- Tromino tiling for a $2^1 \times 2^1$ board with a hole
- Solution is easy

# Tromino tiling/2

Trivial case: $n = 1$

- Tromino tiling for a $2^1 \times 2^1$ board with a hole
- Solution is easy
- Idea for $2^n \times 2^n$ board: reduce size of the board

# TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$

# TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes

# TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes
- Solution: use a tromino to simulate missing holes

# TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes
- Solution: use a tromino to simulate missing holes

- One can repeat this until the $2^1 \times 2^1$ case is reached

# TROMINO TILING/4

## Sketch of algorithm

- Input: board size $n$ ($2^n \times 2^n$), location $l$ of the hole
- Output: tiling of $2^n \times 2^n$ board with hole in location $l$

# TROMINO TILING/4

## Sketch of algorithm

- Input: board size $n$ ($2^n \times 2^n$), location $l$ of the hole
- Output: tiling of $2^n \times 2^n$ board with hole in location $l$
- Rough outline of `tile` $(n, l)$
  - If $n = 1$ then tile with one tromino
  - Otherwise, proceed as follows
    - Divide the board into four equal-sized boards
    - Put a tromino to simulate the three missing holes
    - Call `tile`$(n-1, l_1)$, `tile`$(n-1, l_2)$, `tile`$(n-1, l_3)$, and `tile`$(n-1, l_4)$ where $l_1$, $l_2$, $l_3$, and $l_4$ are the locations of the four holes

# TROMINO TILING/5

**Algo:** Tile(n,L)

---

**Input:** $2^n x 2^n$ board, location L of hole
**Output:** tiling of the board

**if** *n==1* **then** tile with one tromino and **return**;
Divide board into four equal-sized boards;
Place one tromino at center to cover 3 holes;
Let L1, L2, L3, L4 be positions of the 4 holes;
Tile(n-1,L1);
Tile(n-1,L2);
Tile(n-1,L3);
Tile(n-1,L4);

# Tromino tiling/6

## Divide-and-conquer

- The algorithm is an instance of divide-and-conquer

# Tromino tiling/6

## Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size $2 \times 2$

# Tromino tiling/6

## Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size $2 \times 2$
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances

# Tromino tiling/6

## Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size $2 \times 2$
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances
- Conquer: tile the four smaller boards recursively

# Tromino tiling/6

## Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size $2 \times 2$
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances
- Conquer: tile the four smaller boards recursively
- Combine: put a tromino around the center position
  - Covers missing holes to solve the original problem

# Table of contents — SL03

1. Divide and conquer

2. Recurrences
   - Substitution
   - Recursion tree
   - Master method

# Recurrences/1

Informal definition

- Running times of algorithms with recursive calls can be described using recurrences

# RECURRENCES/1

Informal definition

- Running times of algorithms with recursive calls can be described using recurrences
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

# Recurrences/1

## Informal definition

- Running times of algorithms with recursive calls can be described using recurrences
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
- Example: the running time of merge sort is described by the following recurrence

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

# Recurrences/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern

# Recurrences/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method
   - Guess a bound and then use induction to prove that the guess is correct

# RECURRENCES/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method
   - Guess a bound and then use induction to prove that the guess is correct
3. Recursion trees
   - Convert a recurrence in a tree whose nodes represent the costs

# RECURRENCES/2

Solving recurrences

1. Repeated (backward) substitution pragmatic, intuitive
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method proof
   - Guess a bound and then use induction to prove that the guess is correct
3. Recursion trees graphical proof
   - Convert a recurrence in a tree whose nodes represent the costs
4. Master method recipe
   - Templates for different classes of recurrences

# RECURRENCES/3

## Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$

# Recurrences/3

## Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

# Recurrences/3

## Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

- From $8T(n/8) + 3cn$ one gets $2^iT(n/2^i) + icn$

# Recurrences/3

## Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

abstract

- From $8T(n/8) + 3cn$ one gets $2^iT(n/2^i) + icn$
- $i = \lg n$ yields $T(n) = nT(1) + cn \lg n = \Theta(n \lg n)$

# Recurrences/4

Repeated substitution

- Is not a strictly formal proof

# Recurrences/4

## Repeated substitution

- Is not a strictly formal proof
- The procedure is straightforward
  - Substitute, expand, substitute, expand, etc.
  - Observe a pattern and determine the expression after the $i$-th substitution
  - Find out what the highest value of $i$ should be to get the base case of the recurrence
  - Insert the cost of the base case and the expression for $i$ into the observed expression

# RECURRENCES/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

A → B copy n/2
B → A copy rev n/2
merge

3 function calls

# Recurrences/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$T(n) = 2T(n/2) + 2n + 3 \qquad \text{(substitute)}$$

# Recurrences/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \end{aligned}$$

# RECURRENCES/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \end{aligned}$$

# RECURRENCES/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \\ &= 4(2T(n/8) + n/2 + 3) + 4n + 9 && \text{(expand)} \end{aligned}$$

# RECURRENCES/5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \\ &= 4(2T(n/8) + n/2 + 3) + 4n + 9 && \text{(expand)} \\ &= 8T(n/8) + 3 \cdot 2n + (4 + 2 + 1)3 && \text{(find pattern)} \end{aligned}$$

# Recurrences/6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\sum_{j=0}^{i-1} 2^j \end{aligned}$$

# Recurrences/6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\sum_{j=0}^{i-1} 2^j \end{aligned}$$

$$\frac{1-2^{\lg n-1+1}}{1-2} = 2^{\lg n} - 1$$

- Upper bound for $i$ is $\lg n$
- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$T(n) = 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1)$$

# Recurrences/6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\textstyle\sum_{j=0}^{i-1} 2^j \end{aligned}$$

- Upper bound for $i$ is $\lg n$

- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$\begin{aligned} T(n) &= 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1) \\ &= \underline{4n + 2n \lg n - 3} = \underline{\Theta(n \lg n)} \end{aligned}$$

# RECURRENCES/6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\textstyle\sum_{j=0}^{i-1} 2^j \end{aligned}$$

- Upper bound for $i$ is $\lg n$
- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$\begin{aligned} T(n) &= 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1) \\ &= 4n + 2n \lg n - 3 = \Theta(n \lg n) \end{aligned}$$

- Same asymptotic bound as for

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

# EXERCISE SL03-3/1

$T(n-d)$

A program reads numbers into an array until a zero arrives. No assumption about the input can be made. The program starts with an array of size $n = 1$. If the array is full a larger array of size $n'$ is created, the data is copied, and the old array is destroyed. Consider the following strategies:

- $n' = n + d$
- $n' = k * n$

Determine the cost of each strategy.

$T(n) = T(n-d) + n-d$ copy

$T(n) = T(n/k) + n/k$

# EXERCISE SL03-3/2

$$T(n) = \begin{cases} 0 & \text{if } n=1 \text{ (we start with 1)} \\ T(n-d) + n-d \end{cases}$$

---

$$\begin{aligned}
T(n) &= T(n-d) + n-d \\
&= T(n-2d) + n-2d + n-d \\
&= T(n-3d) + 3n - d(1+2) \qquad n-id=1 \\
&= T(n-id) + in - d\sum_{j=1}^{i} j \qquad i_{max} = \frac{n-1}{d} \\
&= \frac{n^2 - dn + d - 1}{2d} = \Theta(n^2)
\end{aligned}$$

# EXERCISE SL03-3/3

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/k) + n/k & \text{if } n > 1 \end{cases}$$

$$T(n) = T(n/k) + n/k$$

$$= T(n/k^2) + n/k^2 + n/k$$

$$= T(n/k^i) + \underline{\sum_{j=1}^{i} n/k^j} \qquad \underline{i_{max} = \log_k n}$$

$$= \frac{n-1}{k-1} = \underline{\underline{\Theta(n)}} \qquad \left| \sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \right.$$

# EXERCISE SL03-3/4

# SUBSTITUTION/1

proof

## Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

intuition, recursion tree, partial solution

# SUBSTITUTION/1

## Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$

# SUBSTITUTION/1

## Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$
- Guessed solution: $T(n) = n^3$, i.e. $T(n)$ is of the form $n^3$

# SUBSTITUTION/1

## Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$
- Guessed solution: $T(n) = n^3$, i.e. $T(n)$ is of the form $n^3$
- Mathematical induction: need to show that $\boxed{T(n) \leq cn^3}$

# SUBSTITUTION/2

step $n/2 \to n$
assume it hold for $n/2$
we make

hypothesis

Proof by induction

- Need to show that $\boxed{T(n) \leq cn^3}$ for $T(n) = 4T(n/2) + n$

$$T(n) = 4T(n/2) + n \qquad \text{(recurrence)}$$

# SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$\begin{aligned} T(n) &= 4T(n/2) + n && \text{(recurrence)} \\ &\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \end{aligned}$$

# SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)}
\end{aligned}
$$

# SUBSTITUTION/2

## Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)} \\
&= cn^3 - \underbrace{(cn^3/2 - n)}_{\geq 0} && \text{(rearrangement)}
\end{aligned}
$$

for $c = 2$,

$$2n^3 - \underbrace{(n^3 - n)}_{\geq 0}$$

# SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)} \\
&= cn^3 - (cn^3/2 - n) && \text{(rearrangement)} \\
&\leq cn^3 && \text{(for } c \geq 2, n \geq 1\text{)}
\end{aligned}
$$

# SUBSTITUTION/2

$\Rightarrow T(1) = 0 \leq c1 \quad c \cdot 1^3$

$T(2) = 2 \leq c \cdot 8 \quad c \cdot 2^3$

$T(4) = 12 \leq c \cdot 64 \quad c \cdot 4^3$

ok for $c = 2$

$4 \cdot T(1) + 2$

$4 \cdot 0 + 2 = 2$

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$ if $n > 1$, $0$ if $n = 0$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)} \\
&= cn^3 - (cn^3/2 - n) && \text{(rearrangement)} \\
&\leq cn^3 && \text{(for } c \geq 2, n \geq 1\text{)}
\end{aligned}
$$

14.3

$T(2) = 4T(1) + 2$

$4 \cdot T(2) + 4$

$4 \cdot 2 + 4$

$12$

- It follows that $T(n) = O(n^3)$

# EXERCISE SL03-4

Use the substitution method to prove that by increasing an array by a factor $k$ each time it is filled up yields a linear asymptotic runtime complexity.

hypothesis: $T(n) \leq cn$ (linear complexity)

$$T(n) = T(n/k) + n/k$$

$$\leq c\,n/k + n/k$$

$$= \frac{n(c+1)}{k}$$

$$\leq cn$$

$$\frac{(c+1)n}{k} \overset{?}{\leq} cn$$

$$\Downarrow \; c = k$$

$$\underbrace{\frac{k+1}{k}}_{\leq 1.5} n \leq \underbrace{kn}_{\geq 2}$$

if $c = k$, $k \geq 2$

# SUBSTITUTION/3

## Getting tighter bound/1

- Try to show that $T(n) \leq cn^2$ for $\boxed{T(n) = 4T(n/2) + n}$

$$T(n) = 4T(n/2) + n \qquad \text{(recurrence)}$$

# SUBSTITUTION/3

## Getting tighter bound/1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)}
\end{aligned}
$$

# SUBSTITUTION/3

## Getting tighter bound/1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= cn^2 + n && \text{(simplification)}
\end{aligned}
$$

# SUBSTITUTION/3

## Getting tighter bound/1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= \underbrace{cn^2 + n} && \text{(simplification)} \\
&\not\leq cn^2 && \text{(contradiction)}
\end{aligned}
$$

# SUBSTITUTION/3

## Getting tighter bound/1

guess: quadratic complexity

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= cn^2 + n && \text{(simplification)} \\
&\not\leq cn^2 && \text{(contradiction)}
\end{aligned}
$$

- It seems that induction does not work!?

# SUBSTITUTION/4

## Getting tighter bound/2

- Underlying problem: trying to rewrite

$$T(n) = O(n^2) = cn^2 + (\text{something positive})$$

as $T(n) \leq cn^2$ does not work in the inductive proof

# SUBSTITUTION/4

## Getting tighter bound/2

- Underlying problem: trying to rewrite

$$T(n) = O(n^2) = cn^2 + (\text{something positive})$$

as $T(n) \leq cn^2$ does not work in the inductive proof

- Solution: strengthen hypothesis for inductive proof

$$T(n) \leq (\text{answer you want}) - (\text{something positive})$$

# SUBSTITUTION/5

## Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$T(n) = 4T(n/2) + n \qquad \text{(recurrence)}$$

# SUBSTITUTION/5

## Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$\begin{aligned} T(n) &= 4T(n/2)+n && \text{(recurrence)} \\ &\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)} \end{aligned}$$

# SUBSTITUTION/5

## Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)} \\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)} \\
&= cn^2 - 2dn + n && \text{(simplification)}
\end{aligned}
$$

# SUBSTITUTION/5

## Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)}\\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)}\\
&= cn^2 - 2dn + n && \text{(simplification)}\\
&= cn^2 - dn - \underbrace{(dn - n)}_{\geq 0 \text{ for } d \geq 1} && \text{(rearrangement)}
\end{aligned}
$$

# SUBSTITUTION/5

n/2

## Getting tighter bound/3

- Show that $\boxed{T(n) \leq cn^2 - dn}$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)}\\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)}\\
&= cn^2 - 2dn + n && \text{(simplification)}\\
&= cn^2 - dn - (dn - n) && \text{(rearrangement)}\\
&\leq cn^2 - dn && \text{(for } d \geq 1\text{)}
\end{aligned}
$$

# SUBSTITUTION/5

## Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)}\\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)}\\
&= cn^2 - 2dn + n && \text{(simplification)}\\
&= cn^2 - dn - (dn - n) && \text{(rearrangement)}\\
&\leq cn^2 - dn && \text{(for } d \geq 1\text{)}
\end{aligned}
$$

- It thus follows that $T(n) = O(n^2)$

# EXERCISE SL03-5/1

An array $A[1..n]$ is balanced if

- the sum of the elements in the first half is no more than double and no less than half the sum of the elements in the second half, and
- the first and second half are balanced.

Give an algorithm that determines if an array is balanced. Use the substitution method to prove the asymptotic complexity of the algorithm.

A1 = 2 5 | 3 5 | 7 not balanced

A2 = 1 | 2 | 3 5 | 7 balanced

# EXERCISE SL03-5/2

Algo Balanced(A, l, r)

if (l = r) then return(TRUE, A[l])

m = ⌊(l+r)/2⌋

(BalLeft, SumL) = Balanced(A, l, m).

(BalRight, SumR) = Balanced(A, m+1, r).

return (BalLeft && BalRight &&

SumL ≥ SumR/2

SumL ≤ SumR * 2, SumL + SumR)

# Exercise SL03-5/3

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + \underline{1} & \text{if } n > 1 \end{cases}$$

$T(1) = 0 \leq c - d$

$T(2) = 1 \leq 2c - d$

$c = 2$

$d = 1$

Guess: $T(n) = O(n)$

$T(n) \leq \underline{cn}$

$T(n) = 2\underbrace{T(n/2)} + 1$

$\leq 2cn/2 + 1$

$= cn + 1$

$\not\leq cn$

Guess: $T(n) = O(n)$

$T(n) \leq \underline{cn - d}$

$T(n) = 2\underline{T(n/2)} + 1$

$\leq 2(cn/2 - d) + 1$

$= cn - 2d + 1$

$\leq cn - d \underbrace{- (d - 1)}_{<0 \text{ if } d > 1}$

✓

# EXERCISE SL03-5/4

# Recursion tree/1

## Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated

# Recursion tree/1

Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences

# Recursion tree/1

## Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$

# Recursion tree/1

- terms with T at leaves (will be exponential)
- terms without T go to root

## Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$

# Recursion tree/1

## Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$

# Recursion tree/2

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

$T(n/2) = T(n/8) + T(n/4) + \left(\frac{n}{2}\right)^2$

Example recurrence/1

Sum it up

$$
\begin{array}{ccccccccc}
 & & & n^2 & & & & \rightsquigarrow n^2 & i=0 \\
 & & \left(\frac{n}{4}\right)^2 & & + & & \left(\frac{n}{2}\right)^2 & = & \rightsquigarrow \frac{5}{16}n^2 & i=1 \\
\left(\frac{n}{16}\right)^2 & + & \left(\frac{n}{8}\right)^2 & + & \left(\frac{n}{8}\right)^2 & + & \left(\frac{n}{4}\right)^2 & & \rightsquigarrow \frac{25}{256}n^2 & = \frac{5^2}{16^2}n^2 \quad i=2 \\
 & & \vdots & & \vdots & & \vdots & & \rightsquigarrow \boxed{\left(\frac{5}{16}\right)^i n^2} & i \\
\hline
 & & & & & & & & \rightsquigarrow \Theta(n^2) &
\end{array}
$$

shortest

longest

$$u^2 \sum_{i=0}^{i=\dots} \left(\frac{5}{16}\right)^i < u^2 \sum_{i=0}^{\infty} \left(\frac{5}{16}\right)^i = u^2 \frac{1}{1-5/16} = n^2 \frac{16}{11} \approx 1.5\, n^2$$

# Recursion tree/3

guess (hypothesis)
follows from recursion tree

## Example recurrence/2

- Show that $\boxed{T(n) \leq cn^2}$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$T(n) = T(n/2) + T(n/4) + n^2 \qquad \text{(recurrence)}$$

# Recursion tree/3

## Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \end{aligned}$$

# Recursion tree/3

## Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\ &= 5cn^2/16 + n^2 && \text{(simplification)} \end{aligned}$$

# Recursion tree/3

## Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\ &= 5cn^2/16 + n^2 && \text{(simplification)} \\ &= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \end{aligned}$$

# Recursion tree/3

## Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$
\begin{aligned}
T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\
&\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\
&= 5cn^2/16 + n^2 && \text{(simplification)} \\
&= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \\
&\leq cn^2 && \text{(for } c \geq 16/11\text{)}
\end{aligned}
$$

# Recursion tree/3

$\frac{5}{16}cn^2 + n^2$

$\left(\frac{16}{16} - \frac{11}{16}\right)cn^2 + n^2$

## Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$T(2)$ $T(1)$ $\rightarrow$ $T(4)$

$$
\begin{aligned}
T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\
&\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\
&= 5cn^2/16 + n^2 && \text{(simplification)} \\
&= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \\
&\leq cn^2 && \text{(for } c \geq 16/11\text{)}
\end{aligned}
$$

- It thus follows that $T(n) = O(n^2)$

# MASTER METHOD/1

recipe

- To solve a class of recurrences that have the form

$$\boxed{T(n) = aT(n/b) + f(n)}$$

# MASTER METHOD/1

- To solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- Assumptions:
  - $a \geq 1$,
  - $b > 1$, and
  - $f(n)$ asymptotically positive

# MASTER METHOD/1

- To solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- Assumptions:
  - $a \geq 1$,
  - $b > 1$, and
  - $f(n)$ asymptotically positive
- $T(n)$ describes running time of a divide-and-conquer algorithm
  - $a$ subproblems of size $n/b$ are solved recursively, each in time $T(n/b)$
  - $f(n)$ is the cost of dividing the problem and combining the solutions

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$T(n) = f(n) + aT(n/b)$$

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2)
\end{aligned}
$$

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= \underbrace{f(n) + af(n/b) + a^2 f(n/b^2) + \ldots}_{\text{non-leaves}} \\
&\quad + \underbrace{a^{\log_b n - 1} f(n/b^{\log_b n - 1})}_{} + \underbrace{a^{\log_b n} T(1)}_{\text{leaves}}
\end{aligned}
$$

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\
&= \underbrace{\textstyle\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)}_{\text{non-leaves}} + \underbrace{\Theta(n^{\log_b a})}_{\text{leaves}}
\end{aligned}
$$

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\
&= \textstyle\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a})
\end{aligned}
$$

- The first term is a division/combination cost (totaled across all levels of the tree)

# MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\
&= \textstyle\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a})
\end{aligned}
$$

- The first term is a division/combination cost (totaled across all levels of the tree)
- The second term is the cost of solving all the trivial subproblems (total of all work pushed to the leaves)

# Master method/3

## Expanding the tree

non-leaves

- Note:
  - $\log_b n$ levels,
  - $a^{\log_b n} = n^{\log_b a}$ leaves
- Running time: $T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$

leaves

non-leaves

# MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)

# MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term

# MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term
- In each case compare $f(n)$ with $n^{\log_b a}$

17.3

# MASTER METHOD/4

$$T(n) = \underbrace{3}_{a}\,T(\underbrace{n/2}_{b}) + \underbrace{n}_{f(n)}$$

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term
- In each case compare $\boxed{f(n)}$ with $\boxed{n^{\log_b a}}$

# MASTER METHOD/5

**Master theorem, case 2**

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)

# MASTER METHOD/5

**Master theorem, case 2**

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)
- The work is evenly distributed: $T(n) = \Theta(n^{\log_b a} \log_b n)$

# MASTER METHOD/5

**Master theorem, case 2**

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)
- The work is evenly distributed: $T(n) = \Theta(n^{\log_b a} \log_b n)$

Example: $\boxed{T(n) = 2T(n/2) + \Theta(n)}$ (merge sort)

- $a = 2$, $b = 2$, $f(n) = \Theta(n)$
- $n^{\log_2 2} = n$ $(= n^1)$
- $f(n)$ $\boxed{\Theta(n)} = \Theta(\boxed{n})$
- case 2: $T(n) = \Theta(n \lg n)$

# MASTER METHOD/6

**Master theorem, case 1**

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor $n^{\epsilon}$)

# MASTER METHOD/6

**Master theorem, case 1**

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor $n^\epsilon$)
- The work at the leaves dominates: $\boxed{T(n) = \Theta(n^{\log_b a})}$

# Master method/6

**Master theorem, case 1**

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor $n^\epsilon$)
- The work at the leaves dominates: $T(n) = \Theta(n^{\log_b a})$

Example: $T(n) = 9T(n/3) + n$

- $a = 9$, $b = 3$, $f(n) = n$
- $n^{\log_3 9} = n^2$
- $n = O(n^{\log_3 9 - 1})$
- case 1: $T(n) = \Theta(n^2)$

# MASTER METHOD/7

**Master theorem, case 3**

- $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor $n^\epsilon$)
  - Regularity: $\exists c < 1.\exists n_0 > 0.\forall n > n_0.af(n/b) \leq cf(n)$

# MASTER METHOD/7

**Master theorem, case 3**

- $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor $n^\epsilon$)
  - Regularity: $\exists c < 1.\exists n_0 > 0.\forall n > n_0.af(n/b) \leq cf(n)$
- The work at the root dominates: $\boxed{T(n) = \Theta(f(n))}$

# MASTER METHOD/7

**Master theorem, case 3**

- $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor $n^\epsilon$)
  - Regularity: $\exists c < 1.\exists n_0 > 0.\forall n > n_0.af(n/b) \leq cf(n)$
- The work at the root dominates: $T(n) = \Theta(f(n))$

Example: $T(n) = 3T(n/4) + n \lg n$

- $a = 3$, $b = 4$, $f(n) = n \lg n$
- $n^{\log_4 3} = n^{0.792}$
- $n \lg n = \Omega(n^{\overbrace{\log_4 3+0.208}^{1}})$
- case 3: $\boxed{T(n) = \Theta(n \lg n)}$
- Need also to verify regularity condition: $af(n/b) \leq cf(n)$

$$c = 3/4 : af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n$$

$f(n)$

$3\frac{n}{4}\log\frac{n}{4} \leq c\, n \log n$ $\quad \frac{3}{4}n\log\frac{n}{4} \leq \frac{3}{4}n\log n$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract $a$, $b$, and $f(n)$ from given recurrence

# Master method/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract $a$, $b$, and $f(n)$ from given recurrence
- Determine $n^{\log_b a}$; asymptotically compare it with $f(n)$

# MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract $a$, $b$, and $f(n)$ from given recurrence
- Determine $n^{\log_b a}$; asymptotically compare it with $f(n)$
- Determine and apply appropriate master theorem case
- There are recurrences $T(n) = aT(n/b) + f(n)$ where no Master theorem case applies (i.e., gaps between the cases).

# EXERCISE SL03-6/1

Use the Master Method to solve the recurrence

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

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

- $a = 2 \quad b = 2 \quad f(n) = n^3$
- $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
- $n^3 = \Omega(n^{1+\varepsilon})$ for $\varepsilon = 2$
- additional check: $\frac{n^3}{4} \leq c\, n^3$ holds for $c = \frac{1}{2}$
- $T(n) = \Theta(n^3)$

$$2\left(\frac{n}{2}\right)^3 \leq c\, n^3$$

$$\frac{n^3}{4} \leq c\, n^3$$

# SUMMARY/1

- Divide and conquer is an algorithm design paradigm that is based on (multiple) recursion.
- A divide and conquer algorithm recursively breaks down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are combined to give a solution to the original problem.
- Divide and conquer is the basis for many important problems (e.g., quicksort, merge sort, number multiplication, parsing, discrete Fourier transform).
- Understand and designing divide and conquer algorithms is a skill that takes time to learn (practice!).

# SUMMARY/2

- The correctness of a divide and conquer algorithm is proved by induction.
- The computational cost of a divide and conquer algorithm is determined by solving recurrence relations.
- A recurrence relation is an equation that recursively defines a sequence of values.
- Thus, from one or more initial terms, each further term of the sequence is defined as a function of the preceding terms.
- We have seen four techniques to solve recurrences:
  - Repeated substitution (not a formal proof)
  - Substitution method (formal proof)
  - Recursion trees
  - Master method (recipe for most common cases)

data structure

algorithmic

# Informatik II

## Heap Sort, Quick Sort

## — SL04 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

# Sorting/1

Why sorting?

- Most fundamental problem in the study of algorithms

# SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine

# SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques
  - Incremental: selection sort, insertion sort
  - Divide-and-conquer: merge sort, quick sort
  - Use of particular data structure: heap sort

# SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques
  - Incremental: selection sort, insertion sort
  - Divide-and-conquer: merge sort, quick sort
  - Use of particular data structure: heap sort
- Lower bound for sorting $\Omega(n \lg n)$ is used to prove lower bounds for other algorithmic problems

# SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$

# SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

# SORTING/2

## Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

## Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory

# SORTING/2

## Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

## Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory
- Heap sort
  - Worst case $\Theta(n \lg n)$, in place sorting algorithm

# SORTING/2

## Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

## Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory
- Heap sort
  - Worst case $\Theta(n \lg n)$, in place sorting algorithm
- Quick sort
  - Average case $\Theta(n \lg n)$, worst case $\Theta(n^2)$, in place

# Table of contents — SL04

1. Heap sort
   - Trees and heaps
   - Heapify
   - Building a heap
   - Heap sort algorithm

2. Quick sort

# Heap sort

## Basic intuition

- Sketch of the selection sort algorithm

**for** $i = 1$ **to** $n$ **do**
  find smallest element in A[i . . . n]; — $\Theta(n)$
  swap smallest element with A[i]; — $\Theta(1)$

# Heap sort

## Basic intuition

- Sketch of the selection sort algorithm

```
for i = 1 to n do
    find smallest element in A[i ... n];
    swap smallest element with A[i];
```

- Line 2 takes $\Theta(n)$, line 3 takes $\Theta(1)$: $\Theta(n^2)$ in total

# Heap sort

## Basic intuition

- Sketch of the selection sort algorithm

```
for i = 1 to n do
    find smallest element in A[i ... n];
    swap smallest element with A[i];
```

- Line 2 takes $\Theta(n)$, line 3 takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: smart data structure to
  - accomplish the task of lines 2 and 3 in $\Theta(1)$
  - maintain the data structure in time $O(\lg n)$
  - obtain a total running time of $O(n \lg n)$

# Trees and heaps/1

## Binary trees/1

- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3

# Trees and heaps/1

## Binary trees/1

- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9

# Trees and heaps/1

## Binary trees/1

- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9
- The root has no parent
  - 9 is the **root** of the tree

# Trees and heaps/1

## Binary trees/1

- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9
- The root has no parent
  - 9 is the **root** of the tree
- A leaf has no children
  - 6, 4, and 8 are **leaves**

# Trees and heaps/2

## Binary trees/2

- The depth (level) of a node $x$ is the length of the path from the root to $x$
  - The **depth** (**level**) of $1$ is $2$
  - The depth of $9$ is $0$

# Trees and heaps/2

## Binary trees/2

- The depth (level) of a node $x$ is the length of the path from the root to $x$
  - The **depth** (**level**) of $1$ is $2$
  - The depth of $9$ is $0$
- The height of a node $x$ is the length of the longest path from $x$ to a leaf
  - The **height** of $7$ is $2$
  - The height of $9$ is $3$

# Trees and heaps/2

## Binary trees/2

- The depth (level) of a node $x$ is the length of the path from the root to $x$
  - The **depth** (**level**) of $1$ is $2$
  - The depth of $9$ is $0$
- The height of a node $x$ is the length of the longest path from $x$ to a leaf
  - The **height** of $7$ is $2$
  - The height of $9$ is $3$
- The height of a tree is the height of its root

# Trees and heaps/3

## Binary trees/3

- The right subtree of a node $x$ is the tree rooted at the right child of $x$
  - The **right subtree** of 9 includes all blue nodes and connecting edges
  - The root of the right subtree of 9 is 7

# Trees and heaps/3

no ordering of nodes
(we will consider different orderings)

## Binary trees/3

- The right subtree of a node $x$ is the tree rooted at the right child of $x$
  - The **right subtree** of 9 includes all blue nodes and connecting edges
  - The root of the right subtree of 9 is 7
- The left subtree of a node $x$ is the tree rooted at the left child of $x$
  - The **left subtree** of 9 includes all red nodes and connecting edges
  - The root of the left subtree of 9 is 3
  - There is no left subtree of 3

# Trees and heaps/4

## Complete trees

- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children

# Trees and heaps/4

## Complete trees

- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children
- A **nearly complete binary tree** is a binary tree where
  - all levels of non-maximal depth $d$ are full (have $2^d$ nodes)
  - all the leaves with maximal depth are as far left as possible

# Trees and heaps/4

## Complete trees

- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children
- A **nearly complete binary tree** is a binary tree where
  - all levels of non-maximal depth $d$ are full (have $2^d$ nodes)
  - all the leaves with maximal depth are as far left as possible

## Definition of heap

- A binary tree is a (binary) **heap** if and only if
  - it is a nearly complete binary tree and, furthermore,
  - each node is greater than or equal to all its children

# EXERCISE SL04-1

Construct a complete binary tree, a nearly complete binary tree, and a heap with the same nodes as $T$.

CBT: impossible; requires $n = 2^h - 1$

NCBT:

# Trees and heaps/5

no holes (missing nodes)
⟹ tree can be represented as an array

Heaps and arrays are closely related:

| node | 9 | 8 | 5 | 7 | 4 | 1 | 3 | 6 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| ind/lev | 1/0 | 2/1 | 3/1 | 4/2 | 5/2 | 6/2 | 7/2 | 8/3 | 9/3 | 10/3 |

# Trees and heaps/6

## Fundamental properties

- Let $i$ be a node index.
- Heap property: $A[parent(i)] \geq A[i]$

# Trees and heaps/6

## Fundamental properties

- Let $i$ be a node index.
- Heap property: $A[parent(i)] \geq A[i]$
- A binary heap can be efficiently stored as an array
  - Because it is a nearly complete binary tree

# Trees and heaps/6

## Fundamental properties

- Let $i$ be a node index.
- Heap property: $A[parent(i)] \geq A[i]$
- A binary heap can be efficiently stored as an array
  - Because it is a nearly complete binary tree
- Finding parent, left child, and right child:

| **Algo:** Parent(i) |
|---|
| **return** $\lfloor i/2 \rfloor$; |

| **Algo:** Left(i) |
|---|
| **return** 2*i; |

| **Algo:** Right(i) |
|---|
| **return** 2*i+1; |

# EXERCISE SL04-2

Assume a heap where all elements are distinct. What is the minimum and maximum number of elements in a heap of height $h$? Where in a MaxHeap might the smallest element be?

- min # of elem: $2^0 + 2^1 + 2^2 + \dots + 1 = 2^k$
- max # of elem: $2^0 + 2^1 + 2^2 + \dots + 2^k = \underline{2^{k+1} - 1}$
- smallest element: must be a leaf

# Trees and heaps/7

- A heap where the largest element is at the root is a max-heap.
  - Used for sorting arrays
  - Max-heap property: $A[parent(i)] \geq A[i]$
- A heap where the smallest element is at the root is a min-heap.
  - Used for priority queues
  - Min-heap property: $A[parent(i)] \leq A[i]$
- A binary heap can be exploited for sorting arrays
  - Because of the organization of the data
  - Basic operations: `heapify`, `build_heap`

# HEAPIFY/1

- Input: index $i$ in array $A$, number $s$ of heap elements

# HEAPIFY/1

- Input: index $i$ in array $A$, number $s$ of heap elements
- Binary trees rooted at `left`($i$) and `right`($i$) are binary heaps

# HEAPIFY/1

- Input: index $i$ in array $A$, number $s$ of heap elements
- Binary trees rooted at `left`($i$) and `right`($i$) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

# HEAPIFY/1

- Input: index $i$ in array $A$, number $s$ of heap elements
- Binary trees rooted at `left`($i$) and `right`($i$) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

- `heapify`($A$, $i$, $s$) transforms the binary tree rooted at $i$ into a binary heap

# HEAPIFY/1

- Input: index $i$ in array $A$, number $s$ of heap elements
- Binary trees rooted at `left`($i$) and `right`($i$) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

- `heapify`($A$, $i$, $s$) transforms the binary tree rooted at $i$ into a binary heap
- Strategy: move $A[i]$ down the heap until the heap property is satisfied again

# HEAPIFY/2

m = 5
m = 10
( m = 11 )

## The algorithm

**Algo:** Heapify(A,i,s)

---

m = i;
l = Left(i); // 2*i
r = Right(i); // 2*i+1
**if** $l \leq s \wedge A[l] > A[m]$ **then** m = l;
**if** $r \leq s \wedge A[r] > A[m]$ **then** m = r;
**if** $i \neq m$ **then**
  exchange A[i] and A[m];
  Heapify(A,m,s);

left element exists

left elem is larger than parent

right elem exists

right is larger than m

# HEAPIFY/3

## An example/1

| node | 6 | 0 | 5 | 9 | 4 | 1 | 3 | 7 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# HEAPIFY/4

## An example/2

| node | 6 | 9 | 5 | 0 | 4 | 1 | 3 | 7 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# HEAPIFY/5

## An example/3

| node | 6 | 9 | 5 | 8 | 4 | 1 | 3 | 7 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# HEAPIFY/6

## Running time

- The running time of heapify on a subtree of size $n$ rooted at $i$ includes time to
  - determine relationship between elements: $\Theta(1)$
  - run heapify on a subtree rooted at one of $i$'s children

# HEAPIFY/6

- $a = 1$ $b = 3/2$ $f(n) = \Theta(1)$
- $n^{3/2 \log_1 1} = n^0 = 1$
- $\Theta(1) = O(1)$ case 2
- $\Theta(1 \cdot {}^{2/3} \log n) = \Theta(\log n)$

## Running time

- The running time of heapify on a subtree of size $n$ rooted at $i$ includes time to
  - determine relationship between elements: $\Theta(1)$
  - run heapify on a subtree rooted at one of $i$'s children
- $2n/3$ is the worst case size of the subtree (half filled bottom level) and thus

$$\boxed{T(n) \leq T(2n/3) + \Theta(1)} \text{ i.e. } T(n) = O(\lg n)$$

# EXERCISE SL04-3

Explain the derivation of the worst case running time of Heapify (size of tree as well as height of tree).

maximize unbalanced ness

# Building a heap/1

## Build a heap

- Convert an array $A$ with $n$ elements into a heap
- Note that the elements in $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are 1-element heaps

# Building a heap/1

## Build a heap

- Convert an array $A$ with $n$ elements into a heap
- Note that the elements in $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are 1-element heaps

## The algorithm

**Algo:** BuildHeap(A,n)

---

**for** $i$ = $\lfloor n/2 \rfloor$ **to** *1* **do** Heapify(A,i,n);

# Building a heap/2

## An example/1

| node | 6 | 0 | 5 | 9 | 4 | 1 | 3 | 7 | 8 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Building a heap/3

## An example/2

| node | 6 | 9 | 5 | 8 | 4 | 1 | 3 | 7 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Building a heap/4

## An example/3

| node | 9 | 8 | 5 | 7 | 4 | 1 | 3 | 6 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Building a heap/5

## Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

# Building a heap/5

## Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

## Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$

# Building a heap/5

## Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

## Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$
- Not an asymptotically tight bound — but good enough for an overall $O(n \lg n)$ bound for heap sort

# Building a heap/5

$h=2$, $\lceil 7/2^3 \rceil = 1$ node
$h=1$, $\lceil 7/2^2 \rceil = 2$ nodes
$h=0$, $\lceil 7/2^1 \rceil = 4$ nodes

## Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

## Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$
- Not an asymptotically tight bound — but good enough for an overall $O(n \lg n)$ bound for heap sort
- Intuition for tight bound: time for heapify to run at a node $i$ varies with the height of $i$
  - An $n$-element binary heap has height $\lg n$
  - The heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$
  - The cost for one call of heapify is $O(h)$

21.3

# Building a heap/6

## Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x|<1$
- ~~Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$~~
- The asymptotically tight bound:

$$T(n) = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h)$$

# Building a heap/6

## Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$T(n) = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h)$$
$$= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \quad \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots$$

# Building a heap/6

## Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$
\begin{aligned}
T(n) &= \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\
&= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \\
&\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\
&= O\left(n \sum_{h=0}^{\infty} h\left(\tfrac{1}{2}\right)^h\right)
\end{aligned}
$$

$k = h,\ x = 1/2$

# Building a heap/6

## Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$
\begin{aligned}
T(n) &= \textstyle\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\
&= O\Big(n \textstyle\sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\Big) \\
&\le O\Big(n \textstyle\sum_{h=0}^{\infty} \frac{h}{2^{h}}\Big) \\
&= O\Big(n \textstyle\sum_{h=0}^{\infty} h(\frac{1}{2})^h\Big) \\
&= O(n * 2) = \underline{O(n)}
\end{aligned}
$$

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

# HEAP SORT ALGORITHM/1

## The algorithm

**Algo:** HeapSort(A,n)

s = n; size of heap (= # of nodes on heap)
BuildHeap(A); $\Theta(n)$
**for** $i = n$ **to** $2$ **do**
  exchange A[i] and A[1]; first element $\Theta(1)$
  s = s-1; last element $\Theta(1)$
  Heapify(A,1,s); $\Theta(\log n)$

$\Theta(n \log n)$

restores heap

reduce the size of heap ("removes" largest element)

# Heap sort algorithm/1

## The algorithm

**Algo:** HeapSort(A,n)

```
s = n;
BuildHeap(A);
for i = n to 2 do
    exchange A[i] and A[1];
    s = s-1;
    Heapify(A,1,s);
```

## Running time

- Heap sort runs in time $O(n) + nO(\lg n) = O(n \lg n)$ guaranteed (also in worst case)

building heap

# Heap sort algorithm/2

## An example/1

| node | 9 | 8 | 7 | 6 | 4 | 1 | 3 | 5 | 0 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Heap sort algorithm/3

## An example/2

| node | 8 | 6 | 7 | 5 | 4 | 1 | 3 | 2 | 0 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Heap sort algorithm/4

## An example/3

| node | 7 | 6 | 3 | 5 | 4 | 1 | 0 | 2 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# EXERCISE SL04-4/1

Run the heap sort algorithm on the array
$A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.
Show the state of the array after BuildHeap and after three iterations of the loop in the HeapSport algorithm.

# EXERCISE SL03-4/2

A = [ 13 8 5 2 7 4 17 20 25 ]

# Table of contents — SL04

1. Heap sort

2. Quick sort
   - Partitioning
   - Performance

# Quick sort

## Rough idea

- A divide-and-conquer algorithm
  - Divide: partition array into two subarrays such that the items in the lower part $<$ the items in the upper part
  - Conquer: recursively sort the two subarrays
  - Combine: trivial since sorting is in place

# QUICK SORT

(i) $m > l$ $\quad l, m-1$ $\quad l+1$

$l, l+1-1$ 1 elem

## Rough idea

- A divide-and-conquer algorithm
  - Divide: partition array into two subarrays such that the items in the lower part $\leq$ the items in the upper part
  - Conquer: recursively sort the two subarrays
  - Combine: trivial since sorting is in place

## The algorithm

**Algo:** QuickSort(A,l,r)

**if** $l<r$ **then** 2 or more elements
  m = HoarePartition(A,l,r);
  QuickSort(A,l,m-1); smallest elements
  QuickSort(A,m,r); largest elements

$\Rightarrow$ no merging is required

# Hoare Partitioning/1

"middle" element (pivot): we (arbitrarily) take the rightmost element as pivot

Hoare partitioning

**Algo:** HoarePartition(A,l,r)

x = A[r]; i = l-1; j = r+1;
**while** *true* **do**
  **repeat** j = j-1 **until** $A[j] \leq x$;
  **repeat** i = i+1 **until** $A[i] \geq x$;
  **if** $i<j$ **then** exchange A[i] and A[j] **else return** i;

we search for a too small element in right part

we search for a too large element in the left part

swap the "wrong" elements

# Hoare Partitioning/2

pivot $x = 4$

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0$, $j = 9$

| | 1 | 2 | | | | | 8 |
|---|---|---|---|---|---|---|---|
| i | 6 | 5 | 2 | 7 | 9 | 3 | 0 | 4 | j |

# Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 2 | 7 | 9 | 3 | 0 | 4 |

- $i = 1, j = 8$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 2 | 7 | 9 | 3 | 0 | 6 |

# Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 2 | 7 | 9 | 3 | 0 | 4 |

- $i = 1, j = 8$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 2 | 7 | 9 | 3 | 0 | 6 |

- $i = 2, j = 7$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 7 | 9 | 3 | 5 | 6 |

# Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 2 | 7 | 9 | 3 | 0 | 4 |

- $i = 1, j = 8$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 2 | 7 | 9 | 3 | 0 | 6 |

- $i = 2, j = 7$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 7 | 9 | 3 | 5 | 6 |

- $i = 4, j = 6$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 3 | 9 | 7 | 5 | 6 |

# Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | 5 | 2 | 7 | 9 | 3 | 0 | 4 |

before the while loop

- $i = 1, j = 8$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 5 | 2 | 7 | 9 | 3 | 0 | 6 |

- $i = 2, j = 7$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 7 | 9 | 3 | 5 | 6 |

- $i = 4, j = 6$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 3 | 9 | 7 | 5 | 6 |

- $i = 5, j = 4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0 | 2 | 3 | 9 | 7 | 5 | 6 |
| | | | j | i | | | |

end of the while loop

large elements

small elements

# EXERCISE SL04-5

Demonstrate Hoare partitioning on the arrays
$A1 = [13, 19, 9, 11, 2, 6, 21]$ and $A2 = [21, 19, 9, 11, 2, 6, 13]$.

13 19 9 11 2 6 $\boxed{21}$ $i=0$ $j=8$ $x=21$
$i=7$ $j=7$ return 7 $j=5$

21 19 9 11 2 6 13 $i=0$ $j=8$ $x=13$
13 19 9 11 2 6 21 $i=1$ $j=7$
13 6 9 11 2 $\boxed{19}$ 21 $i=2$ $j=6$
$i=6$ $j=5$ return 6

# EXERCISE SL04-6

Formulate the invariant for the Hoare partitioning.

Invariant: end of loop before the if stmt

- $A[l..i-1] \leq x$
- $A[j+1..r] \geq x$

Terminate: $l < i \leq r$

$l \leq j \leq r$

| | | | |
|---|---|---|---|
| i | | | j |
| 8 | 7 | 3 | 1 |
| i | | | j |
| 1 | 7 | 3 | 8 |
| j | i | | |
| 1 | 7 | 3 | 8 |

$\Rightarrow i > l$

# EXERCISE SL04-7/1

What happens if

1. line 01 HoarePartition: $A[r]$ is changed to $A[l]$
2. line 05 HoarePartition: $i < j$ is changed to $i \leq j$
3. QS line 01: $l < r$ is changed to $l \leq r$
4. QS line 03: m-1 is changed to $m$
   QS line 04: m is changed to $m + 1$

QS([1,5,7,8], 1,4)

x=8
i=4, j=4

4

**Algo:** HoarePartition(A,l,r)

```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then
        exchange A[i] and A[j]
    else
        return i
```

**Algo:** QuickSort(A,l,r)

```
if l<r then
    m = HoarePartition(A,l,r);
    QuickSort(A,l,m-1);
    QuickSort(A,m,r);
```

1,3

4,4

# EXERCISE SL03-7/2

1. QS([3,5], 1, 2) → P([3,5], 1, 2) x = 3 i = 1 j = 1
   m = 1
   QS([3,5], 1, 0) ✓
   QS([3,5], 1, 2) ∞ loop

/24.3

2) QS([3,5], 1, 2) → P([3,5], 1, 2) x = 5 i = 2 j = 2
   i = 3 j = 1
   A[3] out of bound

3) QS([3], 1, 1) → P([3], 1, 1) x = 3 i = 1 j = 1
   m = 1
   QS([3], 1, 0) ✓
   QS([3], 1, 1) ∞ loop

4) QS([1,5,7,8], 1, 4) → P(...
   x = 8, j = 4, i = 4
   m = 4
   QS([1,5,7,8], 1, 4) ∞ loop

# Performance/1

Worst case

- Occurs when the array is already completely sorted

# Performance/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n - 1$ items and one with ~~0~~ 1 element~~s~~

# PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with $\cancel{0}$ 1 elements
- If such a partitioning arises at each recursive call then

$$\boxed{T(n) = T(n-1) + T(\cancel{0}\,1) + \Theta(n) = T(n-1) + \Theta(n)}$$

where $\Theta(n)$ is the cost of partitioning the array

$\Theta(n)$
$+$
$\Theta(n-1)$
$+$
$\Theta(n-2)$
$+$
$\vdots$
$5$

$\frac{n(n+1)}{2}$

# PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with $0$ elements
- If such a partitioning arises at each recursive call then

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

  where $\Theta(n)$ is the cost of partitioning the array
- Sum of the costs incurred at each level of the recursion yields an arithmetic series (i.e., $\sum_{i=0}^{n} i$)

# PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with $0$ elements
- If such a partitioning arises at each recursive call then

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

  where $\Theta(n)$ is the cost of partitioning the array
- Sum of the costs incurred at each level of the recursion yields an arithmetic series (i.e., $\sum_{i=0}^{n} i$)
- It thus follows that, in the worst case, $T(n) = \Theta(n^2)$

# PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$

# PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

# PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

# PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

Average case/1

- Actually much closer to best case than to worst case

# PERFORMANCE/2

## Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

## Average case/1

- Actually much closer to best case than to worst case
- Quick sort's behavior is determined by the ordering of the array elements

# PERFORMANCE/2

## Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

## Average case/1

- Actually much closer to best case than to worst case
- Quick sort's behavior is determined by the ordering of the array elements
- In average, there is a mix of "good" and "bad" splits

# Performance/3

## Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

# Performance/3

## Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$

# Performance/3

## Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$
- $G(n) = 2B(n/2) + \Theta(n)$
  $= 2G(n/2-1) + \Theta(n/2) + \Theta(n)$
  $= 2G(n/2-1) + \Theta(n)$

# Performance/3

## Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$
- $$\begin{aligned} G(n) &= 2B(n/2) + \Theta(n) \\ &= 2G(n/2-1) + \Theta(n/2) + \Theta(n) \\ &= 2G(n/2-1) + \Theta(n) \end{aligned}$$
- It follows that, on average, $T(n) = \Theta(n \lg n)$

# EXERCISE SL04-8

Consider an array where all elements are equal. Is this a worst case, best case or neither for quick sort with the following partitioning algorithm?

**Algo:** HoarePartition(A,l,r)

```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then
        exchange A[i] and A[j]
    else
        return i
```

A = [ 5, 5, 5 | 5, 5, 5 ]

best case since it splits in the middle

# PERFORMANCE/4

## Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely

# PERFORMANCE/4

## Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely
- Randomized algorithm: partition around random item (instead of last item)
  - Running time is independent of input ordering
  - No specific input triggers worst case behavior
  - Worst case determined only by output of the random-number generator

# Performance/4

## Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely
- Randomized algorithm: partition around random item (instead of last item)
  - Running time is independent of input ordering
  - No specific input triggers worst case behavior
  - Worst case determined only by output of the random-number generator
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity

# PERFORMANCE/5

wrapper around Hoare Partitioning

## Randomized algorithm

increase stability further

pick 3 random elements and choose middle one

**Algo:** RandomizedHoarePartition(A,l,r)

i = Random(l,r);
exchange A[i] and A[r];
**return** ~~Lomuto~~ Hoare Partition(A,l,r); — docs not change

**Algo:** RandomizedQuickSort(A,l,r)

**if** $l<r$ **then**
  m = RandomizedHoarePartition(A,l,r);
  RandomizedQuickSort(A,l,m-1);
  RandomizedQuickSort(A,m,r);

# SUMMARY/1

- Nearly complete binary trees
- Heap data structure
- Heapsort
  - based on heaps
  - worst case is n log n
- Quicksort:
  - partition based sort algorithm
  - popular algorithm
  - very fast on average
  - worst case performance is quadratic

# SUMMARY/2

- Comparison of sorting methods.
- Absolute runtime values are not important; always relate values to each other.
- Relate runtime values to the complexity ($n \log n$, $n^2$).
- Running time in seconds for $n = 2048$ and different sort orders:

$\Theta(n)\,C + \Theta(n)\,M$

$\Theta(n^2)\,C$
$\Theta(n^2)\,M$

$\Theta(n^2)\,C$
$\Theta(n)\,M$

$\Theta(n^2)\,C$
$\Theta(n)\,M$

| | ordered | random | inverse |
|---|---|---|---|
| insertion sort | 0.22 | 50.74 | 103.80 |
| selection sort | 58.18 | 58.34 | 73.46 |
| bubble sort | 80.18 | 128.84 | 178.66 |
| heap sort | 2.32 | 2.22 | 2.12 |
| quick sort (random pivot) | 0.72 | 1.22 | 0.76 |

$\Theta(n^2)$

$\Theta(n \log n)$

$\Theta(n^2)\,C + \Theta(0)\,M$

$\Theta(n^2)\,C + \Theta(n^2)\,M$

C

# Informatik II

## Pointers, Abstract Data Types

— SL05 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

- array: fixed size (limitation, fast)
- no fixed size is known ⇒ dynamic data structures (can grow & shrink)

much more expensive

# Table of contents — SL05

1. Dynamic data structures
   Pointers
   Linked lists

2. ADT: abstract data types

# Dynamic data structures/1

## Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items

# Dynamic data structures/1

## Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items
- But there are certain limitations or disadvantages
  - Dimension of an array is fixed and cannot be changed
  - Occupies an amount of memory that must be determined before program is run (i.e., can be too large or too small)
  - Inserting/removing items may be expensive operations

# Dynamic data structures/1

## Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items
- But there are certain limitations or disadvantages
  - Dimension of an array is fixed and cannot be changed
  - Occupies an amount of memory that must be determined before program is run (i.e., can be too large or too small)
  - Inserting/removing items may be expensive operations
- A possible way out: use of dynamic data structures
  - Resources are allocated only when they are needed
  - Items are represented by records/objects whose fields can be accessed if one has a pointer to the object
  - Change the way in which items are inserted/removed

# Dynamic data structures/2

Linked data structures

- Used to deal with collections of items dynamically

# Dynamic data structures/2

## Linked data structures

- Used to deal with collections of items dynamically
- Record contains fields that point to other records

  - Linked list: each item points to next item in the list
  - Binary tree: each node points to left & right subtree

# Dynamic data structures/2

## Linked data structures

- Used to deal with collections of items dynamically
- Record contains fields that point to other records

  - Linked list: each item points to next item in the list
  - Binary tree: each node points to left & right subtree
- Storage space of linked data structures is not known in advance
  - Depends on the number of items stored in the linked data structure.
  - Only known during runtime, i.e., program execution.
  - Linked data structures can grow and shrink in size

# Records in C

In C a `struct` is used to group fields:

```
struct rec { int a; int b; };
struct rec r;

int main() {
  r.a = 5; r.b = 8;
  printf("Sum a and b is %d\n", r.a + r.b);
}
// gcc dummy.c -o dummy ; ./dummy
```

# Recursive data structures

The counterpart of recursive functions are recursively defined data structures.

Example: list of integers = $\begin{cases} \text{integer} \\ \text{integer, list of integer} \end{cases}$

```
struct list {
  int value;
  struct list tail;
};
```

There must be a mechanism to constrain the initial storage space of recursive data structures (it is potentially infinite).
There must be a mechanism to grow and shrink the storage space of a recursive data structures during program execution.

# POINTERS/1

Informal definition of pointers

- A common technique is to allocate storage space dynamically

# POINTERS/1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime

# POINTERS/1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime
- Compiler reserves space only for addresses to these dynamic parts

# POINTERS/1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime
- Compiler reserves space only for addresses to these dynamic parts
- These addresses are usually called pointers

# Pointers/2

## Memory management

| | address | content |
|---|---|---|
| $i$ | $1af783$ | $22$ |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | $33$ |
| | $1af786$ | `NIL` |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | $44$ |

# POINTERS/2

## Memory management

| | address | content |
|---|---|---|
| $i$ | $1af783$ | $22$ |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | $33$ |
| | $1af786$ | `NIL` |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | $44$ |

- variable $i$: an integer (22)

# Pointers/2

## Memory management

| | address | content |
|---|---|---|
| $i$ | $1af783$ | 22 |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | 33 |
| | $1af786$ | NIL |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | 44 |

- variable $i$: an integer (22)
- variable $p$: pointer to an integer (44)

# POINTERS/2

## Memory management

| | address | content |
|---|---|---|
| $i$ | $1af783$ | 22 |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | 33 |
| | $1af786$ | `NIL` |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | 44 |

- variable $i$: an integer (22)
- variable $p$: pointer to an integer (44)
- variable $r$: record with 2 components
  - an integer (33)
  - a pointer to `NIL`

# POINTERS/2

## Memory management

| | address | content |
|---|---|---|
| $i$ | $1af783$ | $22$ |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | $33$ |
| | $1af786$ | `NIL` |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | $44$ |

address in memory array

values

- variable $i$: an integer (22)
- variable $p$: pointer to an integer (44)
- variable $r$: record with 2 components
  - an integer (33)
  - a pointer to `NIL`
- variable $s$: pointer to the record $r$

# EXERCISE SL05-1

What is the difference between an address and a pointer?

pointer = address + type

# POINTERS/3

## Alternative presentation

| | address | content |
|---|---|---|
| $i$ | $1af783$ | $22$ |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | $33$ |
| | $1af786$ | `NIL` |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | $44$ |

# POINTERS/3

r.a
r.b

memory array

## Alternative presentation

| | address | content |
|---|---|---|
| $i$ | $1af783$ | 22 |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | 33 |
| | $1af786$ | NIL |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | — |
| | $1af789$ | — |
| | $1af78a$ | 44 |

# POINTERS/4

## Concluding remarks

- Pointers are not the only way to implement dynamic data structures

# POINTERS/4

## Concluding remarks

- Pointers are not the only way to implement dynamic data structures
- The programmer does not have to be aware of their existence
  - Indeed, in Java objects are implemented with pointers
  - Creation of objects (i.e. new) allocates storage space
  - Accessing an object is done by following the pointer
  - Deallocation is done by so called garbage collector

# POINTERS/4

## Concluding remarks

- Pointers are not the only way to implement dynamic data structures
- The programmer does not have to be aware of their existence
  - Indeed, in Java objects are implemented with pointers
  - Creation of objects (i.e. new) allocates storage space
  - Accessing an object is done by following the pointer
  - Deallocation is done by so called garbage collector
- Certain programming languages (such as C) require that the storage space is managed explicitly

# Pointers in C/1

printf("%p", p)  int a[100]

0x7fff183c

Operations with pointers:

- To follow (chase, dereference) a pointer we write $*p$

```
*p = 12
```

very often

- To get the address of a variable $i$ we write $\&i$

```
p = &i
```

- To allocate memory we use *malloc(sizeof(Type))*

```
p = malloc(sizeof(int))
```

- To free storage space pointed to by a pointer $p$ we use *free*

```
free(p)
```

# Pointers in C/2

Declaration of pointers:

- To declare a pointer to type T we write T*

```
int* p
```

- Note that * is used for two purposes:
  - Declaring a pointer variable

    ```
    int* p
    ```

  - Following a pointer

    ```
    *p = 15
    ```

  In other languages these are syntactically different.

# Pointers in C/3

```
int i
i = 23

int* p
p = malloc(sizeof(int))
*p = 55

struct rec r
rec.a = 17
rec.b = 24

struct rec* s;
s = &r
```

| | address | content |
|---|---|---|
| $i$ | $1af783$ | 23 |
| $p$ | $1af784$ | $1af78a$ |
| $r$ | $1af785$ | 17 |
| | $1af786$ | 24 |
| $s$ | $1af787$ | $1af785$ |
| | $1af788$ | |
| | $1af789$ | |
| | $1af78a$ | 55 |

# EXERCISE SL05-2

Determine the output of the following program:

```
int i, j, *p, *q;
i = 10;
p = &j;
q = malloc(sizeof(int));
*q = i;
j = i;
q = p;
*q = 5;
printf("%d %d %d %d", i, j, *p, *q);
```

10 5 5 5

# LINKED LISTS/1

- A list of integers (short notation):

- A list of integers (detailed notation):

- Corresponding declaration in C:

```
struct node {
  int val;
  struct node* next;
}

struct node* root;
```

- Accessing a field: `(*p).a` or more convenient `p->a`

# LINKED LISTS/2

Populating a list with integers:

```
root = malloc(sizeof(struct node));
root->val = 88;
root->next = malloc(sizeof(struct node));

p = root->next;
p->val = 52;
p->next = malloc(sizeof(struct node));

p = p->next;
p->val = 12;
p->next = NULL;
```

# LINKED LISTS/3

Print all elements of a list (list traversal):

```
p = root;
while (p != NULL) {
  printf("%d,", p->val);
  p = p->next
}
printf("\n");
```

# EXERCISE SL05-3

Count the occurrences of an integer $x$ in a linked list.

```
cnt = 0;
p = root;
while (p ≠ NULL) {
    if (p->val == x) { cnt++; }
    p = p->next;
}
```

# LINKED LISTS/4

always use graphical illustration first when working with pointers

Inserting 43 at the beginning of a list:

```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```

4 operations (constant time)

# EXERCISE SL05-4

Give C code that appends linked list l1 to linked list l2.

```
if (l2 == NULL) l2 = l1
else {
    p = l2;
    while (p->next ≠ NULL) {
        p = p->next;
    }
    p->next = l1
}
```

special cases:

code works; it sets NULL to NULL ✓

✓

# LINKED LISTS/5

Insert 43 at the end of a list: *root* → 88 → 12 → 43

```
if (root == NULL) {
  root = malloc(sizeof(struct node));
  root->val = 43;
  root->next = NULL;
} else {
  q = root;
  while (q->next != NULL) { q = q->next; }
  q->next = malloc(sizeof(struct node));
  q->next->val = 43;
  q->next->next = NULL;
}
```

# LINKED LISTS/6

Insert 43 at the end of a list: *root* → 88 → 12 ⊣

Explain the following code fragment:

```
q = root;
while (q != NULL) { q = q->next; }
q = malloc(sizeof(struct node));
q->val = 43;
q->next = NULL;
```

# LINKED LISTS/7

Delete element x from a non-empty list: (assumption: x appears in the list; first instance is deleted)

```
p = root;
if (p->val == x) { // x is 1st element
  root = p->next;
  free(p);
} else { // x is not the 1st element
  while (p->next != NULL &&
         p->next->val != x) {
    p = p->next;
  }
  tmp = p->next;
  p->next = tmp->next
  free(tmp);
}
```

# EXERCISE SL05-5

Write C code to delete a linked list.

(root = ~~NULL~~)

we cannot delete p
(we loose rest of list)

we can delete q (since we have p)

```
p = root;
while (p ≠ NULL) {
    q = p;
    p = p->next;
    free(q)
}
```

```
p = root
while (p ≠ NULL)}
    free(p);
    p = p->next;
}
```

after free we can no longer access the memory

```
void del(struct node* p) {
    if (p ≠ NULL) {
        del(p->next);
        free(p);
    }
```

# LINKED LISTS/8

Cost of linked list operations:

- Insertion at beginning: $O(1)$
- Insert at end: $O(n)$
- isEmpty: $O(1)$
- Delete: $O(n)$
- Print: $O(n)$

The code fragments for working with the linked list access the list through global variable `root`. (not good in general)

- Thus, there can be at most one linked list.
- The code does not allow us to define and use multiple linked lists (see ADTs for a solution)

# Doubly linked lists

## Variants of linked lists

- Lists with explicit tail
  - Do have an extra pointer to the last item of the list
  - No need to scan the entire list if an operation applies only to the last item

# Doubly linked lists

## Variants of linked lists

- Lists with explicit tail
  - Do have an extra pointer to the last item of the list
  - No need to scan the entire list if an operation applies only to the last item
- Doubly linked list

  - Each node has a field with a pointer to the previous node of the linked list
  - Provides means to quickly navigate forth and back in the linked list

# EXERCISE SL05-6

Define a doubly linked list and a binary tree in C. Work out differences and similarities between a binary tree and a doubly linked list?

```
struct node {
    int val;
    struct node* lft;
    struct node* rgt;
}
struct node* root;
```

structures are very similar; but use of pointers is different

used very often

```
struct node {
    int val;
    struct node* next;
    struct node* prev;
}
struct node* head;
struct node* tail;
```

used less often

# Table of contents — SL05

1. Dynamic data structures

2. ADT: abstract data types
   - Stacks
   - Queues
   - Ordered lists

# Abstract data types/1

## Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).

# Abstract data types/1

Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).
- It is equipped with a specific interface, i.e., a collection of signatures for the operations that can be invoked on an instance

# Abstract data types/1

## Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).
- It is equipped with a specific interface, i.e., a collection of signatures for the operations that can be invoked on an instance
- Furthermore, it is equipped with a set of axioms (pre- and postconditions) that define the semantics of each operation (i.e. what the operations do to instances of the ADT, but not how)

# Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction

# Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them

# Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them
- Allow one to break work into pieces that can be seen as independent – without compromising correctness
  - Serve as specification of requirements for the building blocks of solutions to algorithmic problems

# Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them
- Allow one to break work into pieces that can be seen as independent – without compromising correctness
  - Serve as specification of requirements for the building blocks of solutions to algorithmic problems
- Provide means to separate the concerns of correctness and performance analysis
  - Design the algorithm using an ADT
  - Count the number of ADT operations
  - Choose implementation of operations

# Abstract data types/3

Similarity with OO-paradigm

- ADT = instance variables + procedures

# Abstract data types/3

Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

# Abstract data types/3

## Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

## Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle

# Abstract data types/3

## Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

## Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle
- Priority queues
  - Another application of heaps

# Abstract data types/3

## Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

## Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle
- Priority queues
  - Another application of heaps
- Ordered lists
  - Linked lists where items are ordered according to a key

LIFO Last In First Out
(useful for stacks)

FIFO First In First Out
(useful for queues)

# STACKS/1

## Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle

# STACKS/1

## Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion

# STACKS/1

## Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion
- Appropriate data structures for their implementation
  - arrays or singly linked lists
- Implemented by a data structure where items are
  - inserted (`push`) at the beginning (linked list) or end (array).
  - removed (`pop`) from the beginning (linked list) or end (array).

# STACKS/1

## Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion
- Appropriate data structures for their implementation
  - arrays or singly linked lists
- Implemented by a data structure where items are
  - inserted (`push`) at the beginning (linked list) or end (array).
  - removed (`pop`) from the beginning (linked list) or end (array).

We design a solution for a **single** stack (i.e., stack is implicit; not a parameter in operations) push(5); push(s1, 5)

# STACKS/2

## Array implementation

- Inserting and removing items from a stack

**Algo:** push(x)

```
S[t] = x;
t = t+1;
```

# STACKS/2

## Array implementation

- Inserting and removing items from a stack

| **Algo:** push(x) |
|---|
| S[t] = x;<br>t = t+1; |

| **Algo:** pop() |
|---|
| t = t-1;<br>**return** S[t]; |

# STACKS/2

## Array implementation

- Inserting and removing items from a stack

| **Algo:** push(x) |
|---|
| S[t] = x;<br>t = t+1; |

| **Algo:** pop() |
|---|
| t = t-1;<br>**return** S[t]; |

- The stack is implemented by an array $S[1..n]$ of length $n$

# STACKS/2

## Array implementation

- Inserting and removing items from a stack

| **Algo:** push(x) |
|---|
| S[t] = x;<br>t = t+1; |

| **Algo:** pop() |
|---|
| t = t-1;<br>**return** S[t]; |

- The stack is implemented by an array $S[1..n]$ of length $n$
- The variable $t$ functions as pointer to top of the stack
- $S$ and $t$ are global variables.

# STACKS/2

## Array implementation

- Inserting and removing items from a stack

| **Algo:** push(x) |
|---|
| S[t] = x;<br>t = t+1; |

| **Algo:** pop() |
|---|
| t = t-1;<br>**return** S[t]; |

- The stack is implemented by an array $S[1..n]$ of length $n$
- The variable $t$ functions as pointer to top of the stack
- $S$ and $t$ are global variables.
- Notice that if $t = 1$ then the stack is actually empty

# Stacks/2

## Array implementation

- Inserting and removing items from a stack

| **Algo:** push(x) | **Algo:** pop() |
|---|---|
| S[t] = x;<br>t = t+1; | t = t-1;<br>**return** S[t]; |

- The stack is implemented by an array $S[1..n]$ of length $n$
- The variable $t$ functions as pointer to top of the stack
- $S$ and $t$ are global variables.
- Notice that if $t = 1$ then the stack is actually empty
- Error checking for under/overflow has been omitted

31.3

# STACKS/3

## A simple example

- Stack $S$ containing 3 items ($t = 4$)

$$S = \begin{array}{|c|c|c|c|} \hline 3 & 8 & 4 & \cdot\,\cdot \\ \hline \end{array}$$

# STACKS/3

## A simple example

- Stack $S$ containing 3 items ($t = 4$)

$$S = \begin{array}{|c|c|c|c|} \hline 3 & 8 & 4 & \cdots \\ \hline \end{array}$$

- Stack $S$ after $2 \times$ `pop` () ($t = 2$)

$$S = \begin{array}{|c|c|c|c|} \hline 3 & 8 & 4 & \cdots \\ \hline \end{array}$$

# STACKS/3

## A simple example

- Stack $S$ containing 3 items ($t = 4$)

$$S = \begin{array}{|c|c|c|c|} \hline 3 & 8 & 4 & \cdots \\ \hline \end{array}$$

- Stack $S$ after $2 \times$ `pop` () ($t = 2$)

$$S = \begin{array}{|c|c|c|c|} \hline 3 & 8 & 4 & \cdots \\ \hline \end{array}$$

- Stack $S$ after `push` (5) ($t = 3$)

$$S = \begin{array}{|c|c|c|c|} \hline & & t & \\ \hline 3 & 5 & 4 & \cdots \\ \hline \end{array}$$

# STACKS/3

## A simple example

- Stack $S$ containing 3 items ($t = 4$) $\quad S = \boxed{3 \mid 8 \mid 4 \mid \cdots}$
- Stack $S$ after $2 \times$ `pop` () ($t = 2$) $\quad S = \boxed{3 \mid 8 \mid 4 \mid \cdots}$
- Stack $S$ after `push` (5) ($t = 3$) $\quad S = \boxed{3 \mid 5 \mid 4 \mid \cdots}$
- Stack $S$ after $2 \times$ `pop` () ($t = 1$) $\quad S = \boxed{3 \mid 5 \mid 4 \mid \cdots}$

t

empty stack

# QUEUES/1

## Properties of queues

- In a queue, insertions and deletions follow the first-in first-out (FIFO) principle

# QUEUES/1

## Properties of queues

- In a queue, insertions and deletions follow the first-in first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer

# QUEUES/1

## Properties of queues

- In a queue, insertions and deletions follow the first-in first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer
- Appropriate data structures for their implementation
  - arrays or singly linked lists with an explicit tail
- Implemented by a data structure where items are
  - inserted (`enqueue`) at the end
  - removed (`dequeue`) from the beginning

# QUEUES/1

## Properties of queues

- In a queue, insertions and deletions follow the first-in first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer
- Appropriate data structures for their implementation
  - arrays or singly linked lists with an explicit tail
- Implemented by a data structure where items are
  - inserted (`enqueue`) at the end
  - removed (`dequeue`) from the beginning

We design a solution for a **single** queue (i.e., queue is implicit; not a parameter in operations)

calls: enqueue(5); x = dequeue();

# QUEUES/2

## Array implementation

- Inserting and removing items from a queue

**Algo:** enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

# QUEUES/2

## Array implementation

- Inserting and removing items from a queue

**Algo:** enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

**Algo:** dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

# QUEUES/2

## Array implementation

- Inserting and removing items from a queue

**Algo:** enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

**Algo:** dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

- The queue is implemented by an array $Q[0..n-1]$ of length $n$ used in circular fashion

# QUEUES/2

## Array implementation

- Inserting and removing items from a queue

**Algo:** enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

**Algo:** dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

- The queue is implemented by an array $Q[0..n-1]$ of length $n$ used in circular fashion
- Variables $h$ and $t$ function as pointers to head and tail of the queue

# QUEUES/2

## Array implementation

- Inserting and removing items from a queue

| **Algo:** enqueue(x) |
|---|
| Q[t] = x;<br>t = t+1 mod n; |

| **Algo:** dequeue() |
|---|
| i = h;<br>h = h+1 mod n;<br>**return** Q[i]; |

- The queue is implemented by an array $Q[0..n-1]$ of length $n$ used in circular fashion mod function
- Variables $h$ and $t$ function as pointers to head and tail of the queue h = 1st element ; t = 1st free element
- Notice that if $h = t$ then the queue is actually empty

empty queue

# QUEUES/3

## A simple example

- Queue $Q$ containing 3 items ($h = n - 2$, $t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline \overset{0}{\cancel{4}} & \overset{1}{0} & \cdots & \cancel{3} & \overset{n-1}{\cancel{8}} \\ \hline \end{array}$$

t h

# QUEUES/3

## A simple example

- Queue $Q$ containing 3 items ($h = n - 2, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

- Queue $Q$ after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline \cancel{4} & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

h t

# QUEUES/3

## A simple example

- Queue $Q$ containing 3 items ($h = n - 2, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

- Queue $Q$ after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

- Queue $Q$ after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline \cancel{4} & \cancel{5} & \cdots & 3 & 8 \\ \hline \end{array}$$

h t

# QUEUES/3

## A simple example

- Queue $Q$ containing 3 items ($h = n - 2, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

- Queue $Q$ after 2 × `dequeue` () ($h = 0, t = 1$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 0 & \cdots & 3 & 8 \\ \hline \end{array}$$

- Queue $Q$ after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 5 & \cdots & 3 & 8 \\ \hline \end{array}$$

full queue

- Queue $Q$ after 2 × `dequeue` () ($h = t = 2$)

$$Q = \begin{array}{|c|c|c|c|c|} \hline 4 & 5 & \cdots & 3 & 8 \\ \hline \end{array}$$

h,t

# Single vs multiple instances of an ADT

- So far we have seen examples with **one** instance of the abstract data type, i.e., one stack, one queue, etc.
- If only one instance is needed the actual instance is implicit and not visible in procedure calls.

- If it shall be possible to have multiple instances we need to be able to create instances, specify which instance to use, etc.
- This is achieved by extending all procedures with an extra argument for the instance (e.g., stack1, stack2, etc).

- Below we give an example that allows multiple ordered lists. The same principle could also have been used for stacks and queues.

# ORDERED LISTS/1

- give a copy of a document to a procedure/person
- copy can be changed
- original cannot be changed

- In an ordered list elemenst are ordered according to a key.
- We design a solution for **multiple** ordered lists (i.e., the ordered list is an explicit parameter in operations).

- all arguments of procedures are copies
- pass a pointer to the original so that procedure can change the original

- Example functions on ordered list:
  - `struct node** init()`
  - `bool isEmpty(struct node** l)`
  - `int first(struct node** l)`
  - `int last(struct node** l)`
  - `void print(struct node** l)`
  - `void insert(struct node** l, int x)`

l is a pointer to a pointer to a node

why? the first never changes; only the second pointer changes

# ORDERED LISTS/2

Definition of data structure in library (we do not define a global instance variable root):

```
struct node {
  int val;
  struct node* next;
};
```

Definition of instances of ordered lists in application programs:

```
struct node** root1;
struct node** root2;
```

# ORDERED LISTS/3

call/use: struct node** r1;
r1 = init();

Initialization of an ordered list:

```
struct node** init() {
  struct node **l;
  l = malloc(sizeof(struct node*));
  *l = NULL;
  return l;
}
```

Retrieving the 1st element of an ordered list (-1 if list is empty):

```
int first(struct node** l) {
  if (*l == NULL) return -1;
  else return (*l)->val;
}
```

# ORDERED LISTS/4

Insertion into an ordered list:

```
void insert(struct node** l, int x) {
  struct node* p;
  struct node* q;

  if (*l == NULL || (*l)->val > x) {
    p = malloc(sizeof(struct node));
    p->val = x;
    p->next = *l;
    *l = p;
  } else {

...
```

l does not change

*l changes

insert(r1, 3)

# Ordered lists/5

Insertion into an ordered list (cnt'd):

```
...

  p = *l;
  while (p->next != NULL &&
         p->next->val < x) p = p->next;
  1) q = malloc(sizeof(struct node));
  2) q->val = x;
  3) q->next = p->next;
  4) p->next = q;
  }
}
```

# Dealing with changing roots/1

- The roots of ADT instances may change as a result of operations:
  - Initializing a list (before: root has no value; after: root has value NULL)
  - Inserting a node with value 3 into an empty list (before: root is NULL; after: root points to node with value 3)
- Changing the arguments of procedures has only a local effect. Changes are lost after the procedure terminates.
- Changes to the root must be made permanent.

- Solution is to have a data structure where the head, which is passed as an argument to procedures, does not change. The head points to the body, which might change.

# EXERCISE SL05-7/1

A first possibility to systematically deal with a changing root of a dynamic data structure is to use a **sentinel** (dummy element). Illustrate a sentinel for a list with zero, one and two elements.

No sentinel:

root changes!

bad because root changes

With sentinel (dummy element):

root never changes

call: struct node* l;
insert(l, 7)

# EXERCISE SL05-7/2

A second possibility to systematically deal with a changing root of a dynamic data structure is to use a **a pointer to a pointer** at the beginning of the list. Illustrate a list with a pointer to a pointer for zero, one and two elements.

No pointer to pointer:

root changes

Pointer to pointer

root never changes

```
struct node ** l;
insert(l, 7)
```

# EXERCISE SL05-7/3

A third possibility to systematically deal with a changing root of a dynamic data structure is to convert all procedures that possibly modify the root into **functions that return the modified object**. Illustrate the changes (signature and call of insert operation).

```
struct node* insert(struct node* r, int x) {
    .....
    return r;
}
```

```
l = insert(l, 7)
```

# Dealing with changing roots/2

- In C an array is implemented as a pointer to its elements:
  - If an array is used as a parameter only the pointer is passed.
  - Passing a pointer is very fast.
  - Changes to the array are global rather than local.
- Objects are also implemented as pointers to the actual content.

# EXERCISE SL05-8/1

Explain and illustrate the following piece of code:

```
void inc(int i) {i = i + 1;}
int main() { int x=7; inc(x); }
```

does not change x

Write a C procedure that can be used to increment an integer value that is passed as a parameter. Illustrate your solution.

```
void inc(int* i) { *i = *i + 1; }

main() {
    int x=7; inc(&x);
    int* pi = malloc(sizeof(int)); *pi = 7; inc(pi);
}
```

# SUMMARY

- Dynamic data structures are essential since they allow application to increase and decrease the memory space they use.
- The basic mechanism to implement dynamic data structures are pointers and the allocation and deallocation of memory.
- Memory allocation and deallocation is expensive.
- The most important dynamic data structures are lists, trees, stacks, and queues.
- Abstract data types encapsulate data structures and algorithms that operate on these data structures.
- They are the foundation of classes with methods.