Informatics II

Introduction, Sorting, Recursion
— SL01 —



Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL01

1. Introduction
   Introduction

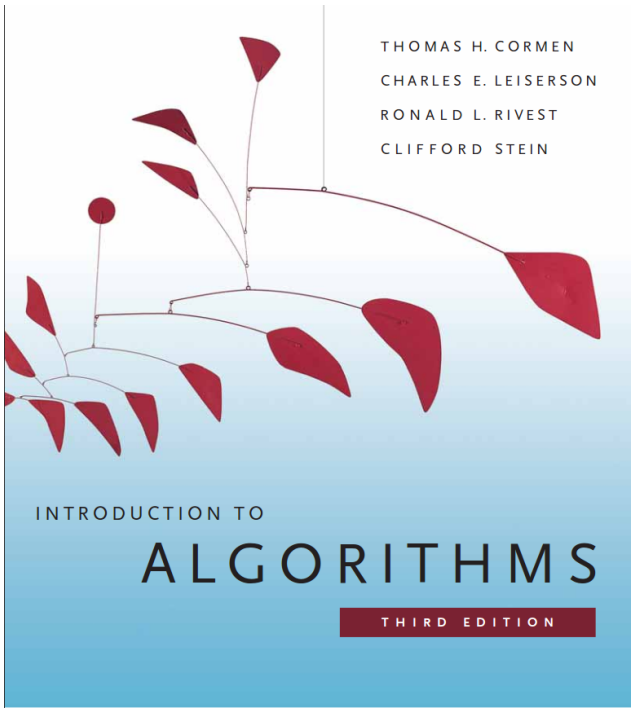
2. Algorithms

3. Sorting

4. Recursion

LITERATURE

- T. Cormen, C. Leiserson, R. Rivest and C. Stein (CLRS), Introduction to Algorithms, Third Edition, MIT Press, 2009.
- Available in IfI library (two books on display; 6 books for borrowing)
- Available as ebook from inside the UZH network:




```
http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=
343613&site=ehost-live
```

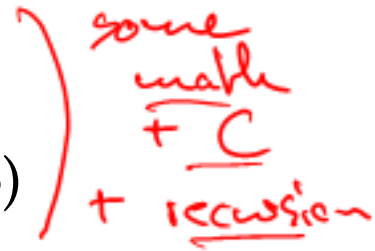
GOALS OF THIS COURSE

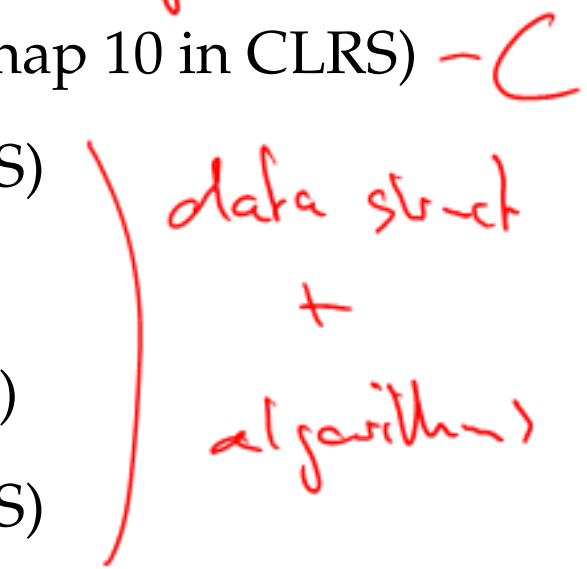
The main things we will learn in this course:

- 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

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)





ADMINISTRATION/1



Home page

- `http://www.ifi.uzh.ch/dbtg`
- `https://lms.uzh.ch/url/RepositoryEntry/16718364817`
- Check OLAT frequently

Course book

- Introduction to Algorithms, 3rd edition, Cormen et al.

Assistant

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

TA:

- Abhinav Aggarwal, `abhinav.aggarwal@uzh.ch`

ADMINISTRATION/2

Lectures

- Tuesday 14:00 - 15:45 in HAH-E-11 (Häldeliweg 2)
- Fridays 12:15 - 13:45 in Y15-G-60 (Irchel)
- First lecture: Tuesday 18.2.2020
- Last lecture: Friday 22.5.2020
- Exceptions are the following dates:
  - FR 10.4.2020 12:15 - 13:45 no lecture (Spring break)
  - TU 14.4.2019 14:00 - 15:45 no lecture (Spring break)
  - FR 17.4.2019 12:15 - 13:45 no lecture (Spring break)
  - FR 1.4.2019 14:00 - 15:45 no lecture (labor day)

ADMINISTRATION/3

Exercises:

- There are 12 weekly exercises
- An exercise is published one week before it is being discussed in the exercise classes.
- There is no hand in of exercises.
- Attending exercises is not mandatory.
- It is strongly recommended that you solve the exercise before the exercise class.

Install before Dev C++ (see GLAT)

ADMINISTRATION/3

Exercise classes:

- Monday 16:15 - 17:45, BIN-2.A.10
- Monday 16:15 - 17:45, BIN-0.K.11/12/13
- Wednesday 14:15 - 15:45, BIN-0.K.02
- Wednesday 16:15 - 17:45, BIN-0.B.06
- Friday 14:15 - 15:45, Y35-F-32 (Irchel)



- Exceptions: (same time slot, different room)
  - Lab 2, 23.03.2020, 16:15 - 17:45, AND-3-02/06
  - Lab 2, 20.04.2020, 16:15 - 17:45, AND-3-02/06
  - Lab 2, 11.05.2020, 16:15 - 17:45, AND-3-02/06
  - Lab 4, 25.03.2020, 16.15 - 17.45 , BIN-0.K.02

ADMINISTRATION/4

Tutors:

- Adam Klebus, `adam.klebus@uzh.ch`
- Anton Crazzolara, `anton.crazzolara@uzh.ch`
- Christoph Vogel, `christoph.vogel@uzh.ch`
- Johann Schwabe, `johann.schwabe@uzh.ch`
- Raphael Haemmerli, `raphael.haemmerli@uzh.ch`

ADMINISTRATION/5

Exams: The course assessment consists of two midterms and a final exam.

Exam dates (check VVZ and official WWF web pages):

- Midterm 1 (MT1, 90min): Monday, 23.3.2020, 12:15 - 13:45, HAH-E-03 (Häldeliweg 2).
- Midterm 2 (MT2, 90min): Monday, 27.4.2019, 12:15 - 13:45, HAH-E-03 (Häldeliweg 2)
- Final exam (FE) date: Wednesday 27.5.2019, 14:00 - 15:30

Your final grade is calculated as follows:

```
MAX(MT1,FE)*0.2 + MAX(MT2,FE)*0.2 + FE*0.6
```

General remarks/1

- Enrollment in the labs takes place in OLAT in the section "Team Enrollment". Please make sure you enroll as soon as possible. Deadline for enrollment: Saturday 29/02/2020 23:00.
- Note that there will be parts in the midterms and the final exam where you have to write working C code.
- One A4 sheet with your personal notes (both sides) is allowed in all Informatics II exams.

General remarks/2

- **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**.

General remarks/3

- 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.
- This script is designed as a **working script**. Solutions to selected exercises are included at the end.

PREREQUISITES

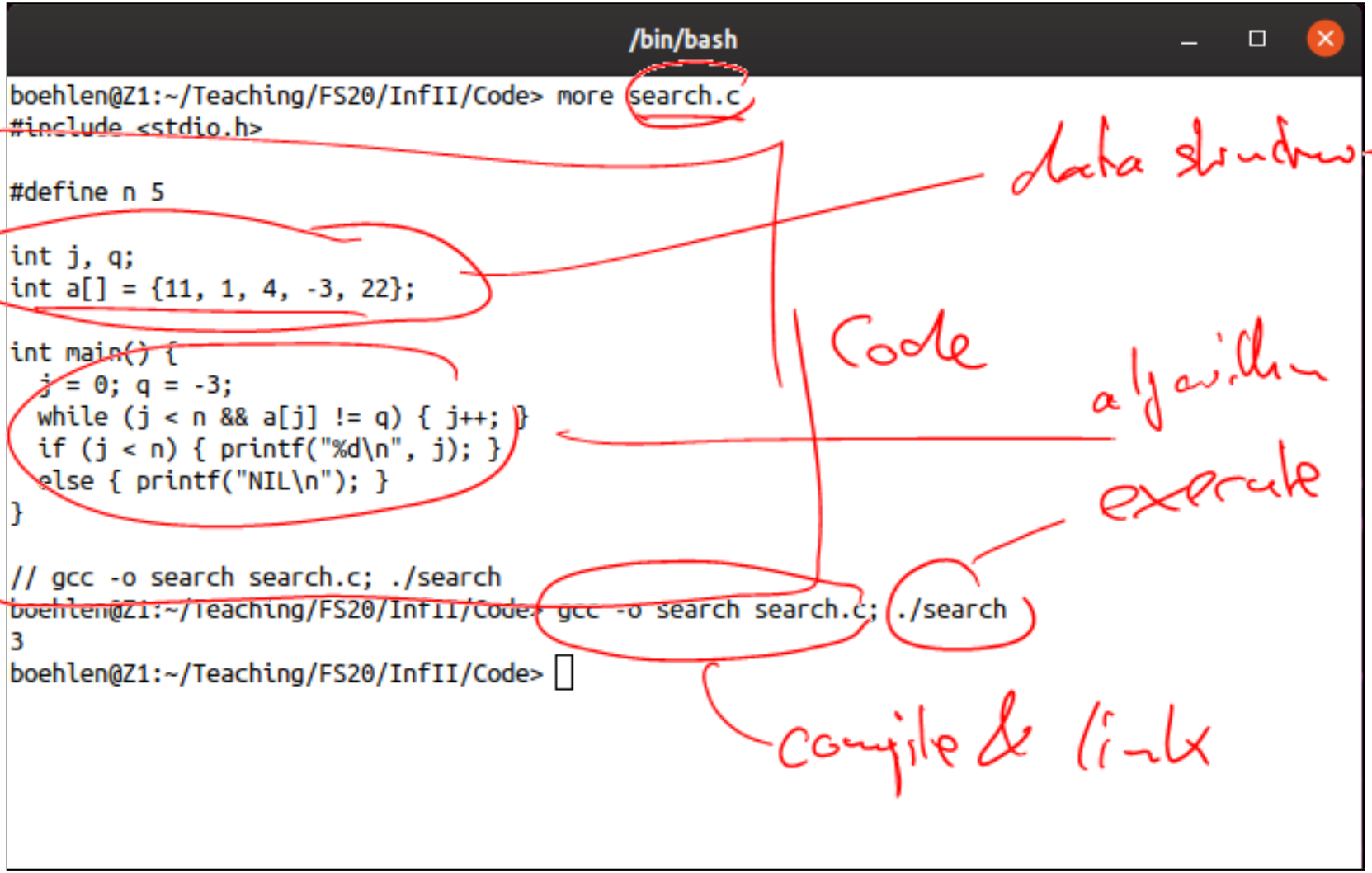
Introduction to programming (aka Informatik I)

- Data types, operations
- Conditional statements
- Loops
- Procedures and functions

Ability to edit, compile, and execute a program

- editor and terminal
- Xcode, Eclipse or equivalent

SHELL


/bin/bash
boehlen@Z1:~/Teaching/FS20/InfII/Code> more search.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 -o search search.c; ./search
boehlen@Z1:~/Teaching/FS20/InfII/Code> gcc -o search search.c; ./search
3
boehlen@Z1:~/Teaching/FS20/InfII/Code>
data structure
Code
algorithm
execute
compile & link

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 I 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 human 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 human 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)
  - Ada Lovelace (first computer program)
- 20th century
  - Alan Turing (Turing machine, cryptography)
  - Alonzo Church (decideability)
  - John von Neumann (Von Neumann architecture)

Code + data in the same place

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

understand what must be done in all cases

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 AlgoDat is **not** about

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

Overall picture/2

What AlgoDat 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

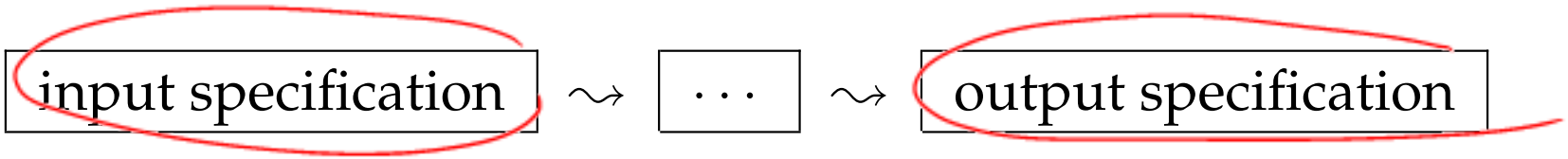
- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  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



Specification of output as function of specification of input

ALGORITHMS/2

Algorithmic problem

Input/output relation

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

Specification of output as function of specification of input

Algorithmic solution

Input/output relation

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

precise ?

EXAMPLE OF AN ALGORITHM/1

- Searching problem
  - Input: a sequence (array) of n numbers $A = [a_1, a_2, \ldots, a_n]$ and a value v
  - Output: an 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: $i = 2$

Input: $A = [1, 1]$, $v = 1$

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;
```

gives index of last equal element

Algo: LinSearch2(A,v)

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

gives index of first equal element

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

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

Fundamental properties

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



Algo: LinSearch3(A,v)

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

because of v we go to n+1
when i = n+1 then
A[n+1] is accessed
which does not exist

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

EXAMPLE OF AN ALGORITHM/5

- What about the fourth solution for searching?
- Run through the array and return the index of the value in the array.

Algo: LinSearch4(A,v)

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

- OK?

or it returns index of first equal element
(same result as LinSearch2)

EXAMPLE OF AN ALGORITHM/6

Metaphor: shopping behavior when buying a beer.

- `LinSearch1`: scan products until you get to the exit; if during the process you find a beer put it into the basket (and remove the rest).
- `LinSearch2`: scan products; stop as soon as a beer is found and go to the exit.
- `LinSearch4`: scan products; stop as soon as a beer is found and exit through next door.

PSEUDO CODE

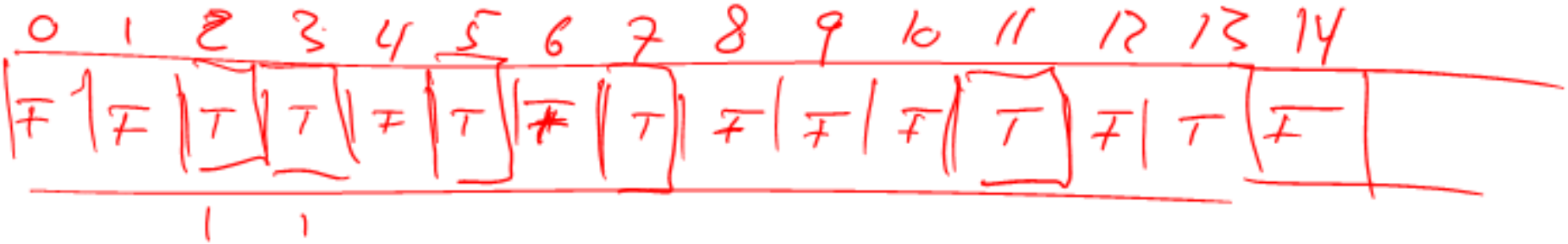
Use of pseudocode

- 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.

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.



Exercise SL01-1/2

Algo: Era(A[0..n])

A[0] = FALSE; A[1] = FALSE;

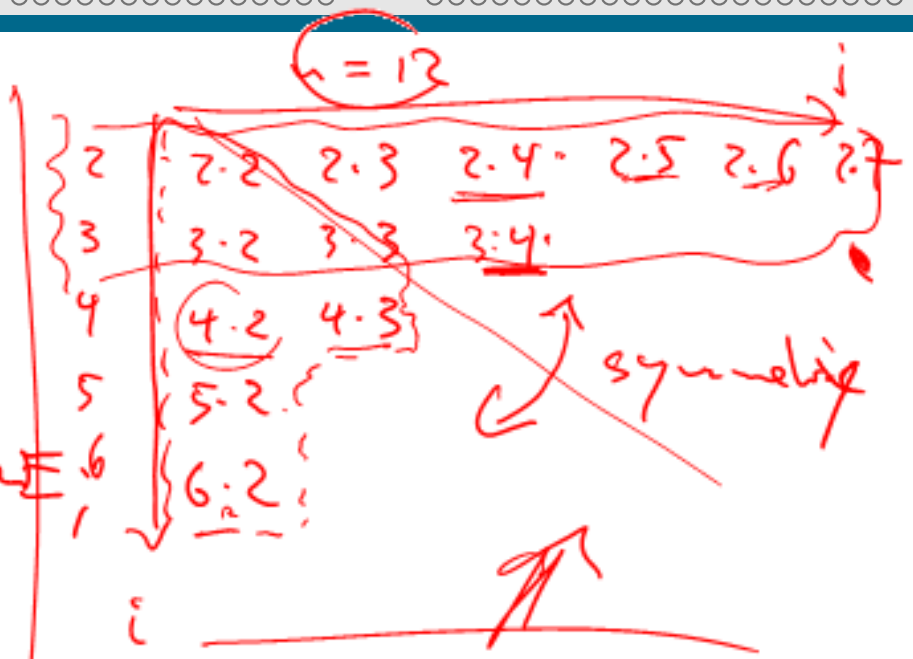
for i = 2 to n do A[i] = TRUE

4.32

for i = 2 to $\lfloor n/2 \rfloor$ do

for j = 2 to $\lfloor n/i \rfloor$ do

A[i*j] = FALSE



- $i \leq j$

for i = 2 to $\lfloor \sqrt{n} \rfloor$ do

for j = 2 to $\lfloor n/i \rfloor$ do

---

for i = 2 to $\lfloor n/2 \rfloor$ do

for j = 2 to min($\lfloor n/i \rfloor$, i)

SEARCHING, C SOLUTION

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

How to approach C, etc.

- Do not study it (it is close to Java or Python 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

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
```

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

$O(n^2)$

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

very bad

Fast methods

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

very good

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

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: $[26, 31, 41, 41, 58, 59]$

permutation

$\leq \leq \leq \leq$

BUBBLE SORT / 1

Rough idea

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

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;
```

BUBBLE SORT / 1

Rough idea

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

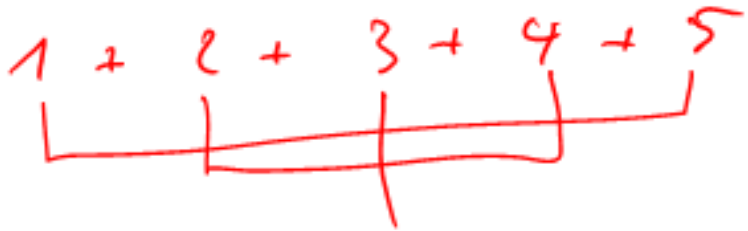
The algorithm

```
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
o	r	s	i	n	g	t
o	r	i	n	g	s	t
o	i	n	g	r	s	t
i	n	g	o	r	s	t
i	g	n	o	r	s	t
g	i	n	o	r	s	t

BUBBLE SORT/2





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.

BUBBLE SORT / 3

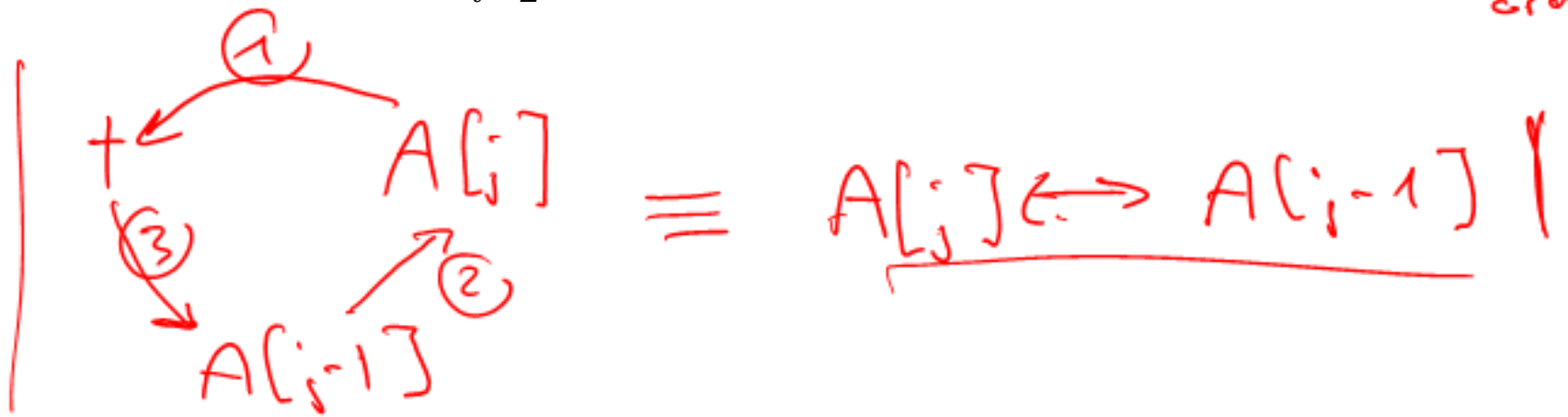
Number of exchanges:

$$Mmin = 0$$

we never have to swap elements (array is already sorted)

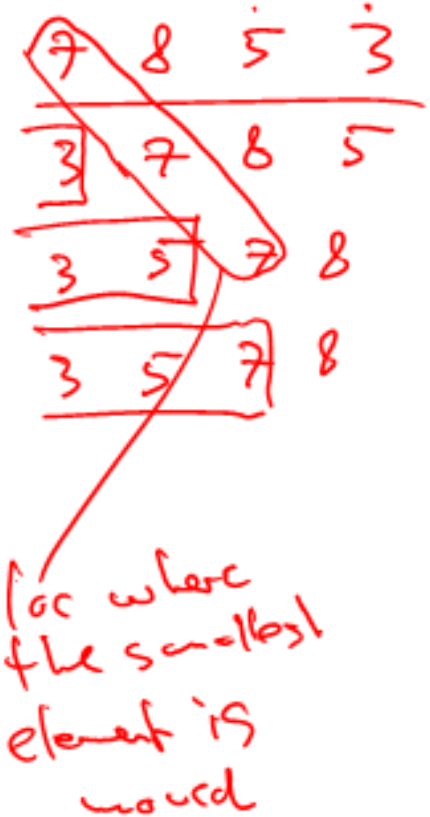
$$Mmax = \sum_{i=2}^{n} 3(i-1) = \frac{3n(n-1)}{2} = \frac{3n^2 - 3n}{2}$$

(array is in reverse order)



EXERCISE SL01-2

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.



BubbleSort2(A)

for i = 1 to n-1 do
  for j = n-1 to i do
    if A[j+1] < A[j] then
      A[j+1] ⟷ A[j]

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

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];
```

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

SELECTION SORT / 2

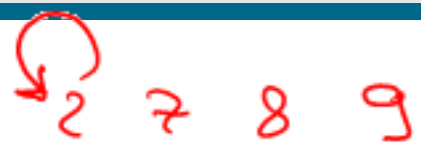
Basic properties

- 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



Basic properties

- Number of exchanges:

swap 1st with smallest

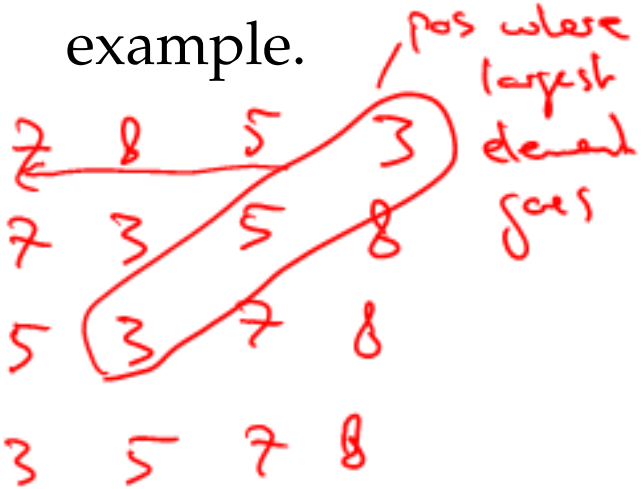
$$M = \sum_{i=1}^{n-1} 3 = 3(n-1) = 3n - 3$$ linear

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

- why not 0 in best case?

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.




```
SelSort2(A)

for i = n to 2
    k = i
    for j = i-1 to 1
        if A[j] > A[k] then k = j
    exchange A[i] and A[k]
```

✓? for j = 1 to i-1 do

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

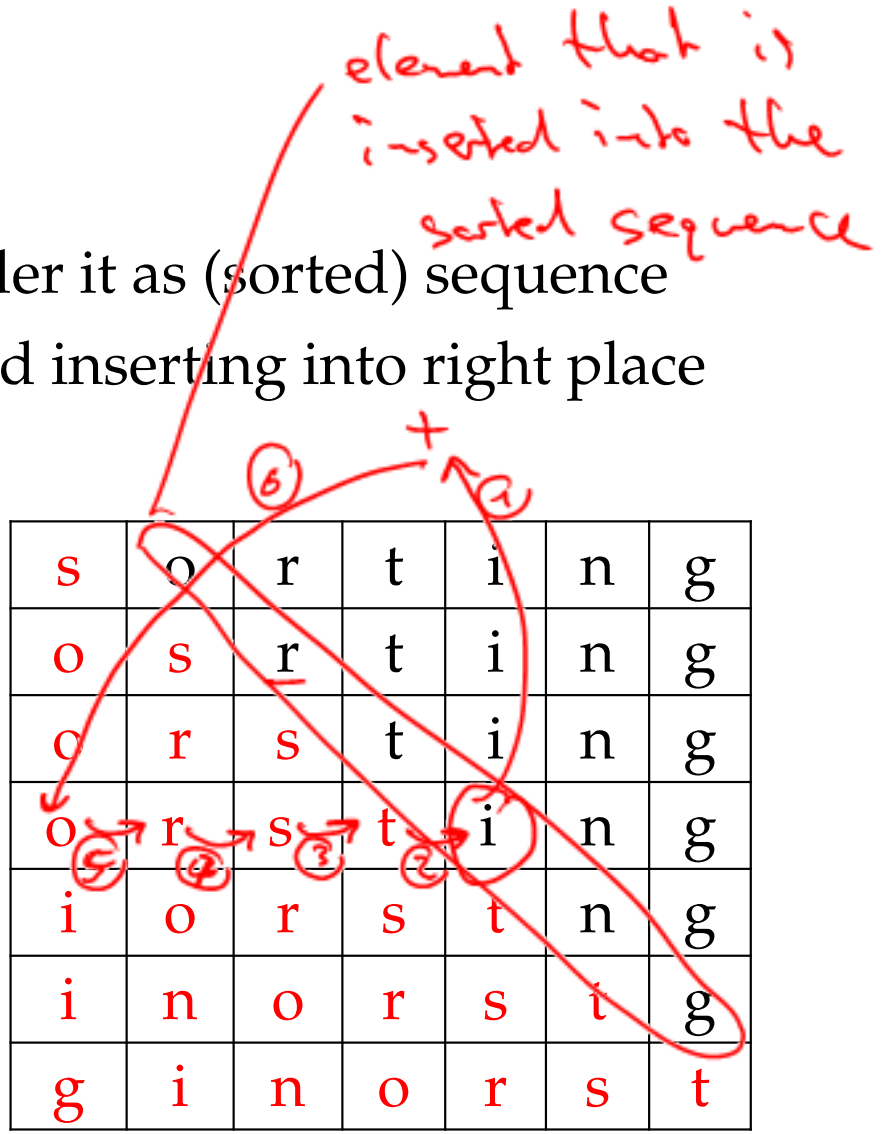
Rough idea

- 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 / 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 the smallest and goes to the beginning

INSERTION SORT / 3

Basic properties

- Number of exchanges:

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

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

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
- Multiple & mutual recursive calls
- Termination condition to stop recursion

RECURSION/1

Recursive object

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

Recursive procedure

- Procedure that calls itself
- Multiple & mutual recursive calls
- Termination condition to stop recursion

$n! = 1 \cdot 2 \cdot 3 \cdots \cdot n$

Example: simple recursion

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

termination

recursion

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)

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

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)

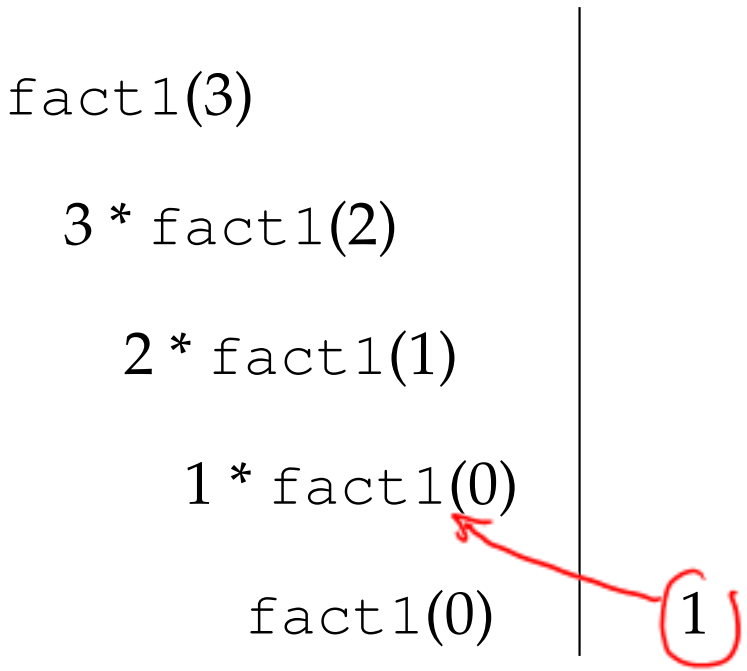
RECURSION/3

Tracing the call `fact1(3)`:

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

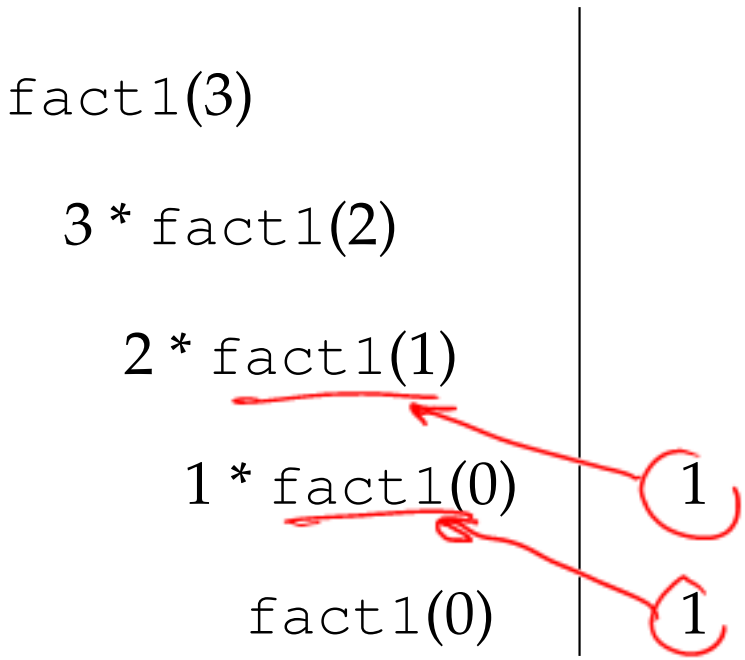
RECURSION/3

Tracing the call `fact1(3)`:



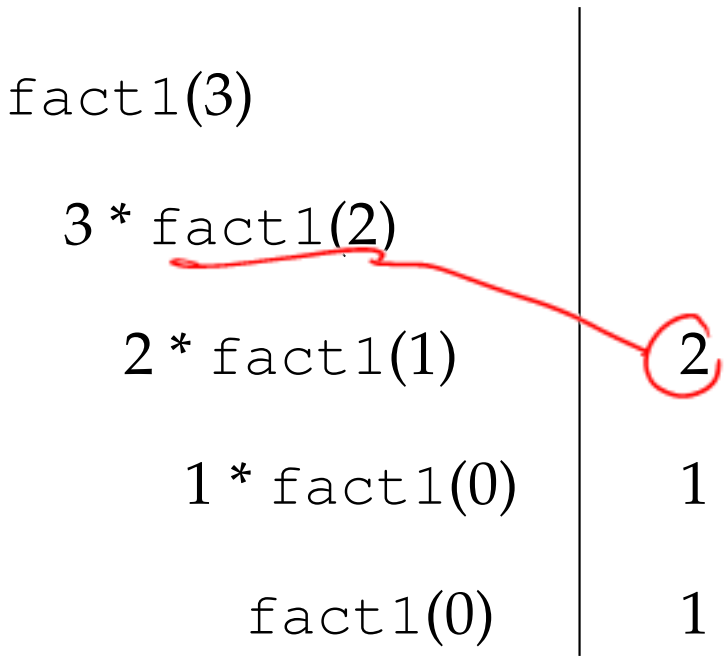
RECURSION/3

Tracing the call `fact1(3)`:



RECURSION/3

Tracing the call `fact1(3)`:



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

Example: multiple recursion

$$\text{Fibonacci } fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

RECURSION / 4

Example: multiple recursion

Fibonacci $$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Example: mutual recursion

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

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

odd(n) when is odd true

return n>0 && even(n-1)

even(n)

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

negative numbers?

odd

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

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

`fib`(3) + `fib`(2)

RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`

`fib(3)` + `fib(2)`

`fib(2)` + `fib(1)`

RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`

`fib(3) + fib(2)`

`fib(2) + fib(1)`

`fib(1) + fib(0)`

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

`fib`(3) + `fib`(2)

`fib`(2) + `fib`(1)

`fib`(1) + `fib`(0)

`fib`(1)

RECURSION/5

Tracing the call `fib(4)`:

```
fib(4)
   fib(3) + fib(2)
      fib(2) + fib(1)
         fib(1) + fib(0)
            fib(1)                  | 1
```

RECURSION/5

Tracing the call `fib(4)`:

```
fib(4)
   fib(3) + fib(2)
      fib(2) + fib(1)
         fib(1) + fib(0)
            fib(1)                  1
            fib(0)
```

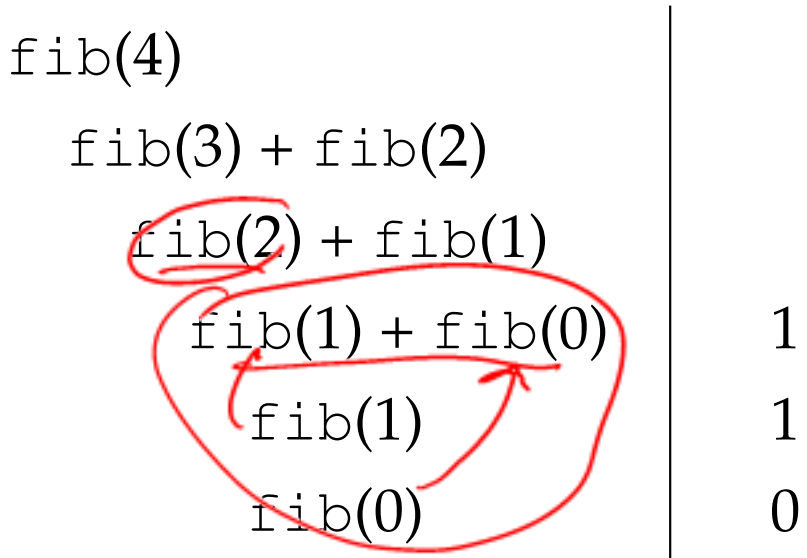
RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:



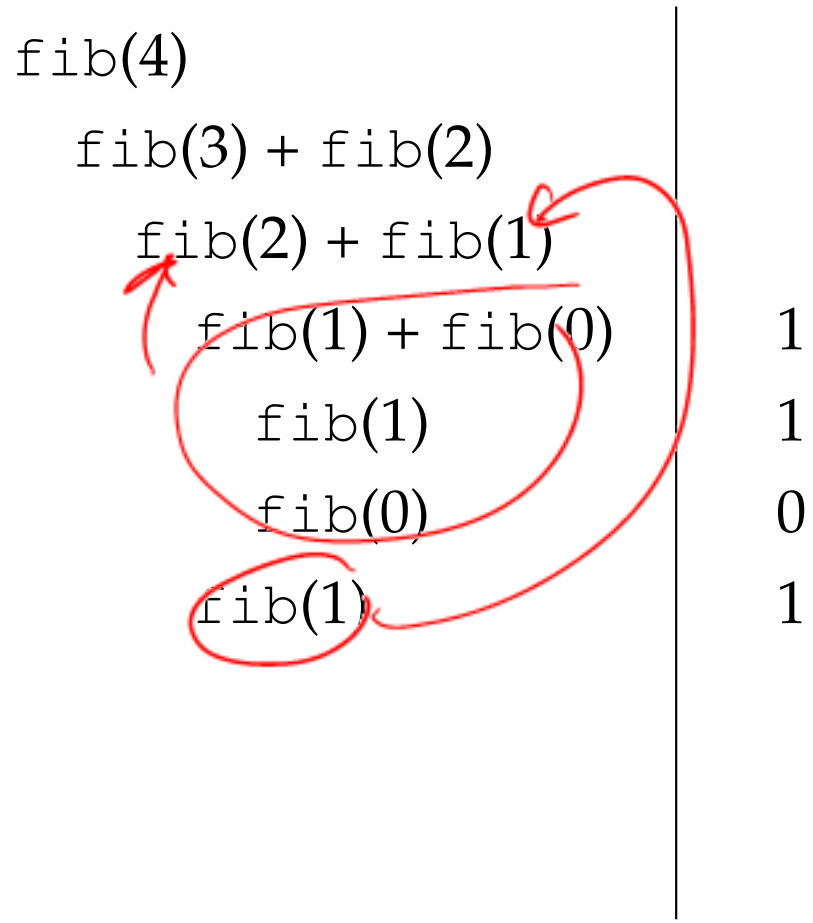
RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	

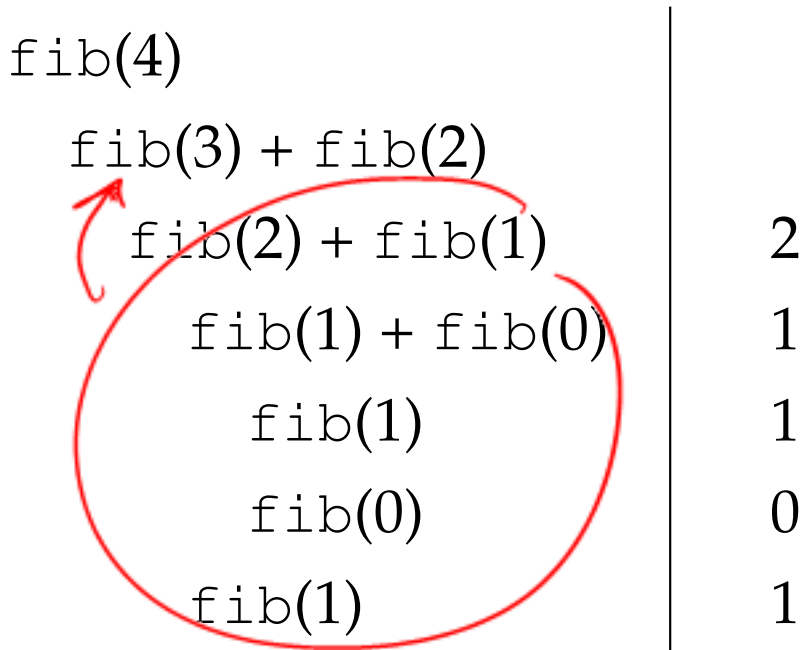
RECURSION/5

Tracing the call `fib(4)`:



RECURSION/5

Tracing the call `fib(4)`:



RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`	
`fib(3) + fib(2)`	3
`fib(2) + fib(1)`	2
`fib(1) + fib(0)`	1
`fib(1)`	1
`fib(0)`	0
`fib(1)`	1
`fib(1) + fib(0)`	1
`fib(1)`	1
`fib(0)`	0

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	3
`fib`(3) + `fib`(2)	3
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

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

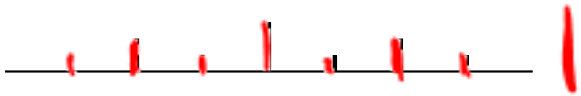
RECURSION/6

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

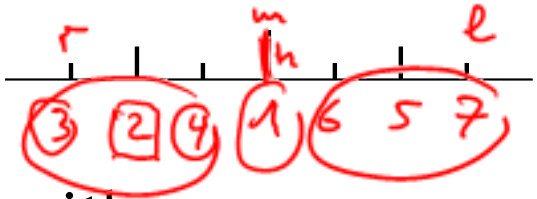
Problem description

- 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)

```
m = (r+l)/2;
if h>0 then
    Mark(m,h);
    Ruler(l,m,h-1);
    Ruler(m,r,h-1);
```

MARKING A RULER/2

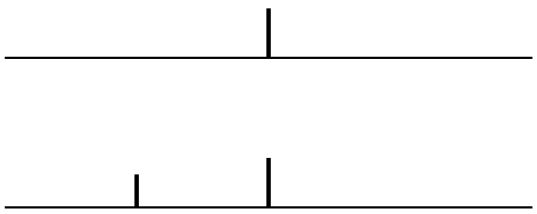
Printing of marks:

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

MARKING A RULER/2

Printing of marks:

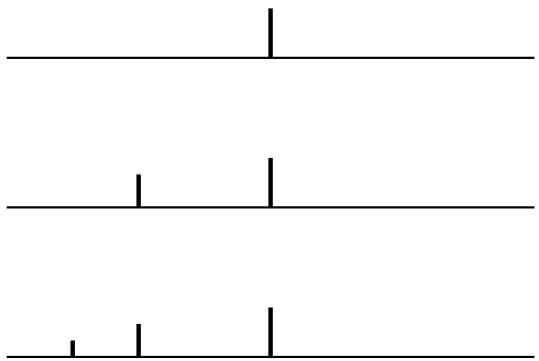
`Ruler` $(0, 8, 3)$
  `Mark` $(4, 3)$
  `Ruler` $(0, 4, 2)$
    `Mark` $(2, 2)$



MARKING A RULER/2

Printing of marks:

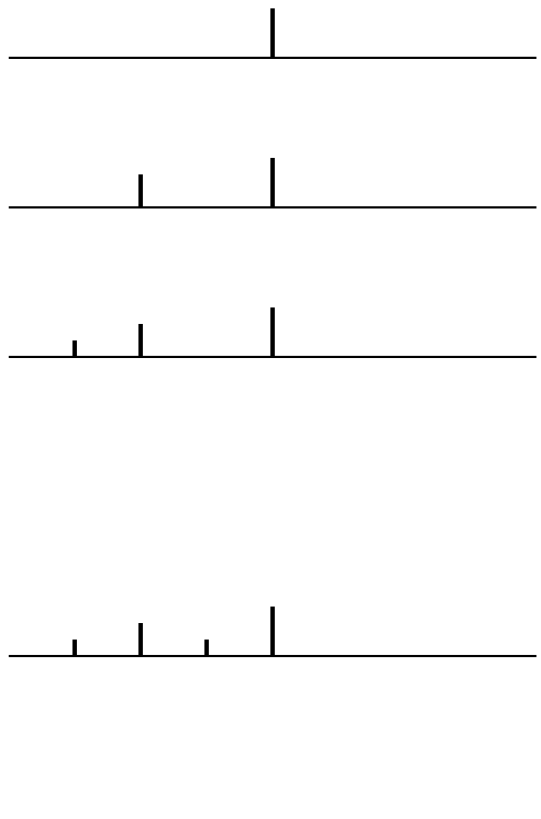
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
```



Marking a ruler/2

Printing of marks:

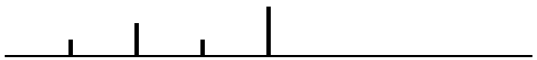
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
    Ruler (2, 4, 1)
      Mark (3, 1)
      Ruler (2, 3, 0)
      Ruler (3, 4, 0)
```



MARKING A RULER/3

Printing of marks:

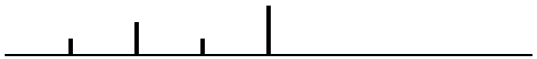
$[\cdots]$



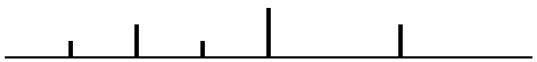
MARKING A RULER/3

Printing of marks:

$[\cdots]$



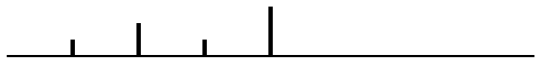
`Ruler` $(4, 8, 2)$
`Mark` $(6, 2)$



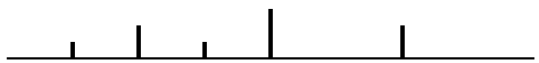
MARKING A RULER/3

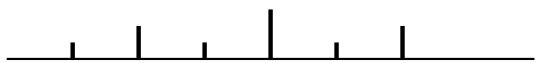
Printing of marks:

$[\cdots]$



`Ruler` $(4, 8, 2)$
  `Mark` $(6, 2)$
  `Ruler` $(4, 6, 1)$
    `Mark` $(5, 1)$
    `Ruler` $(4, 5, 0)$
    `Ruler` $(5, 6, 0)$



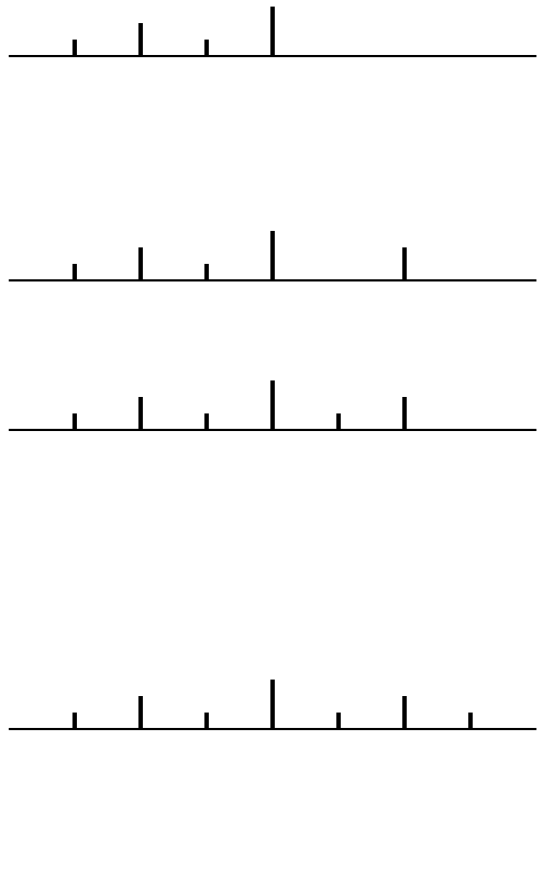


MARKING A RULER/3

Printing of marks:

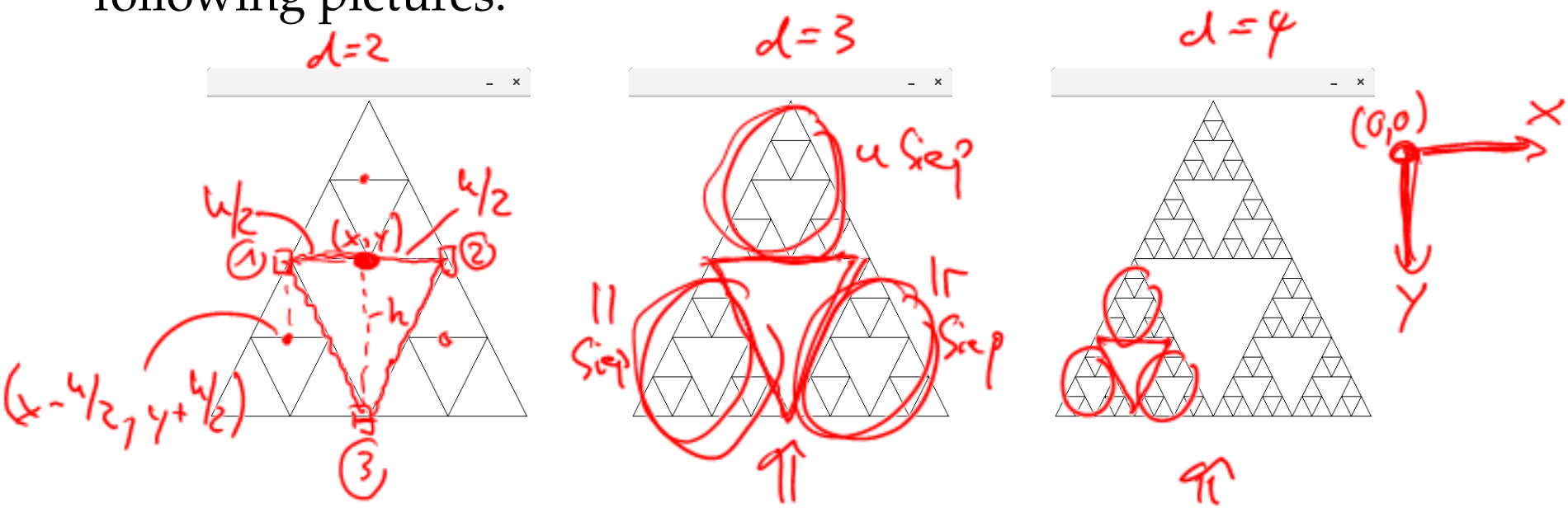
[· · ·]

```
Ruler (4, 8, 2)
  Mark (6, 2)
  Ruler (4, 6, 1)
    Mark (5, 1)
    Ruler (4, 5, 0)
    Ruler (5, 6, 0)
  Ruler (6, 8, 1)
    Mark (7, 1)
    Ruler (6, 7, 0)
    Ruler (7, 8, 0)
```



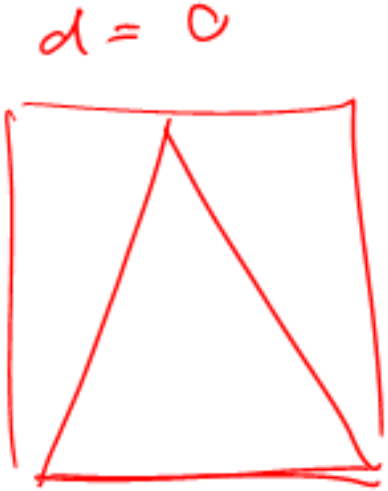
EXERCISE SL01-5/1

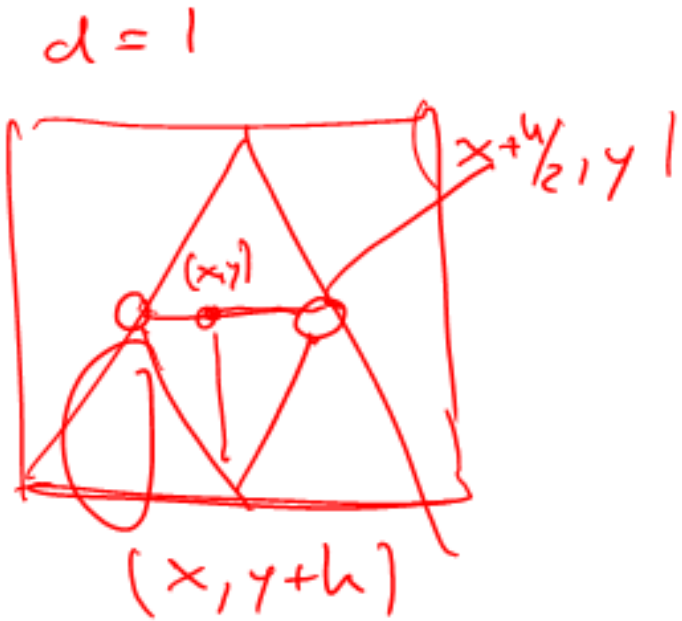
Consider the Sierpinski triangles of depths 2, 3, and 4 in the following pictures:



- 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.

EXERCISE SL01-5/2



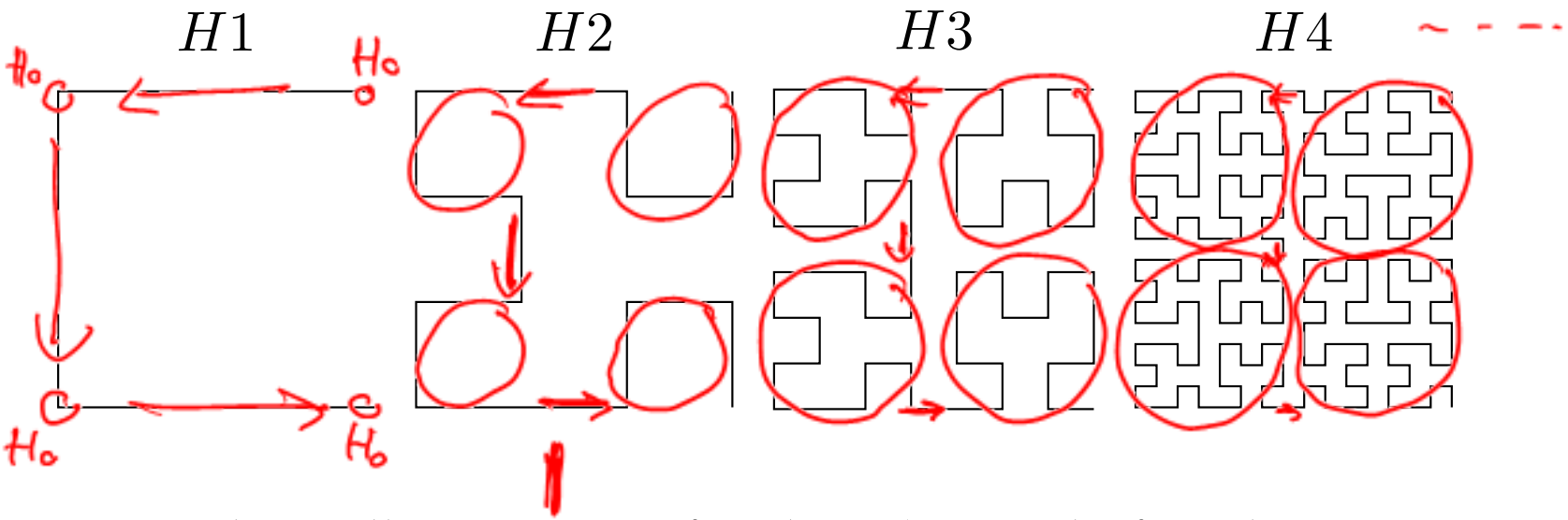



```
SierpΔ(x, y, h, d)
  if d > 0
    draw Δ with (x-h/2, y) ①,
                (x+h/2, y) ②,
                (x, y+h) ③
    SierpΔ(x-h/2, y+h/2, h/2, d-1)
    SierpΔ(x+h/2, y+h/2, h/2, d-1)
    SierpΔ(x, y-h, h/2, d-1)
```

HILBERT CURVE/1

space filling curve

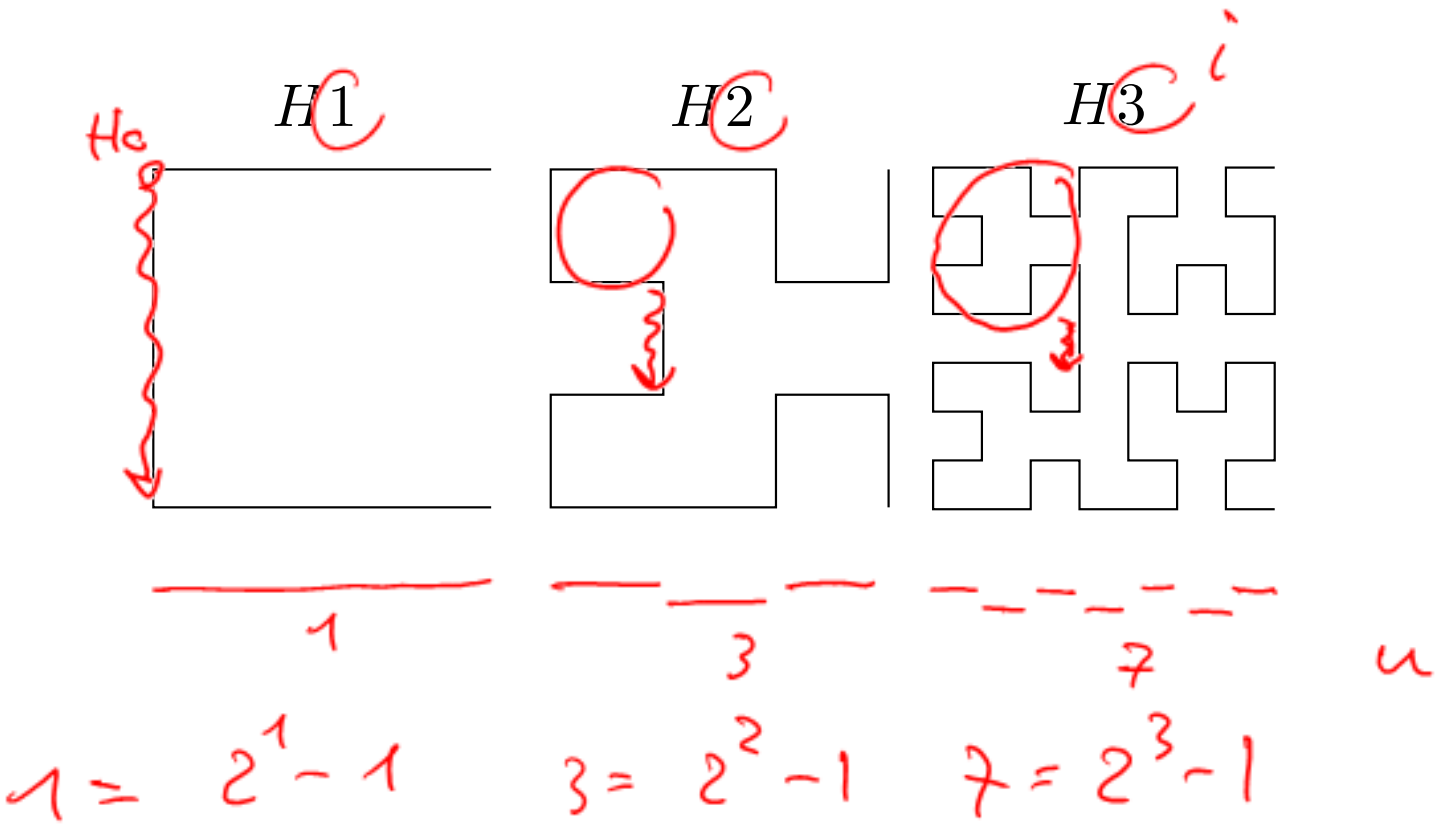
Consider the following patterns:



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

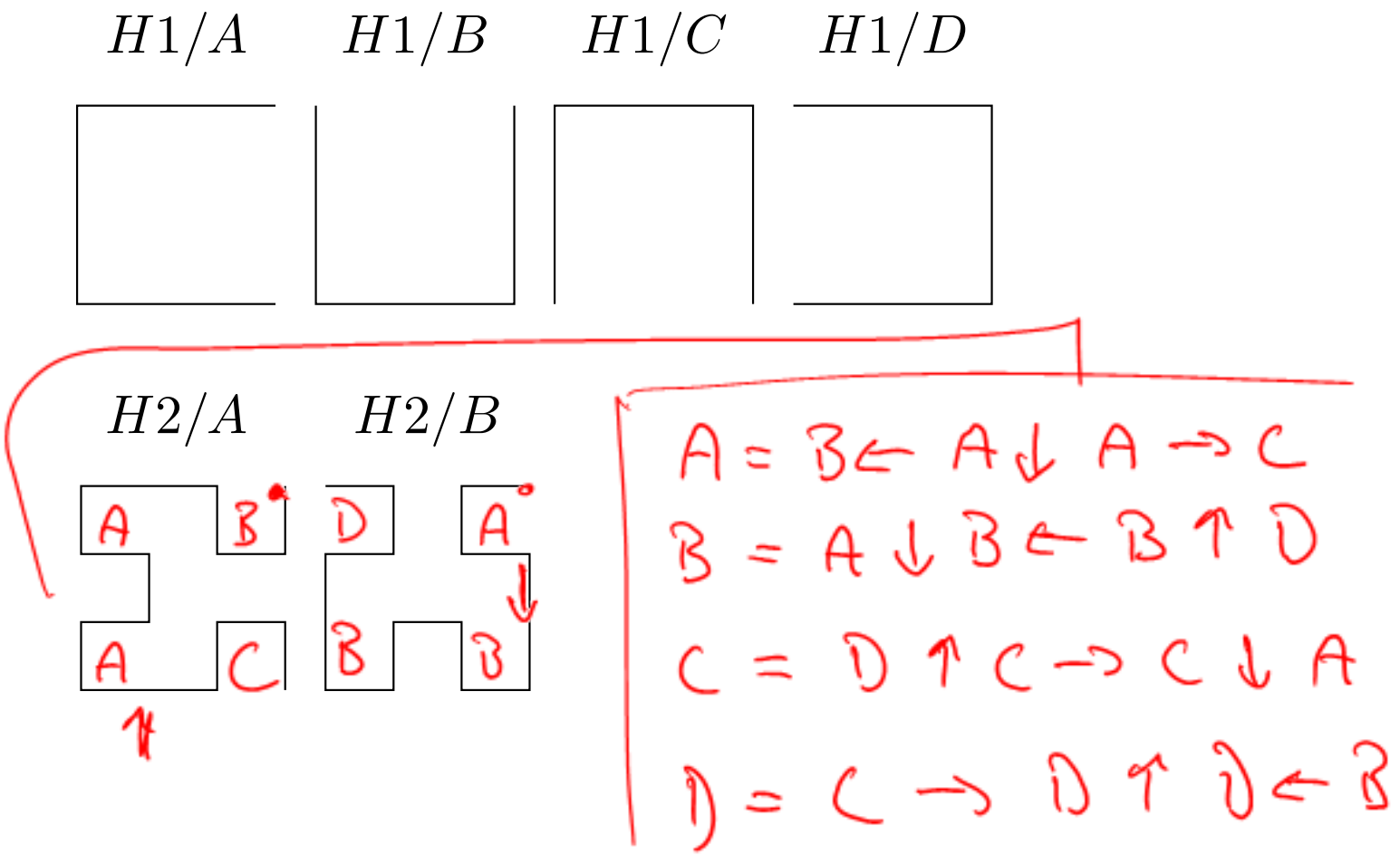
HILBERT CURVE/2

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



HILBERT CURVE/3

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



HILBERT CURVE/4

The first part of the code:

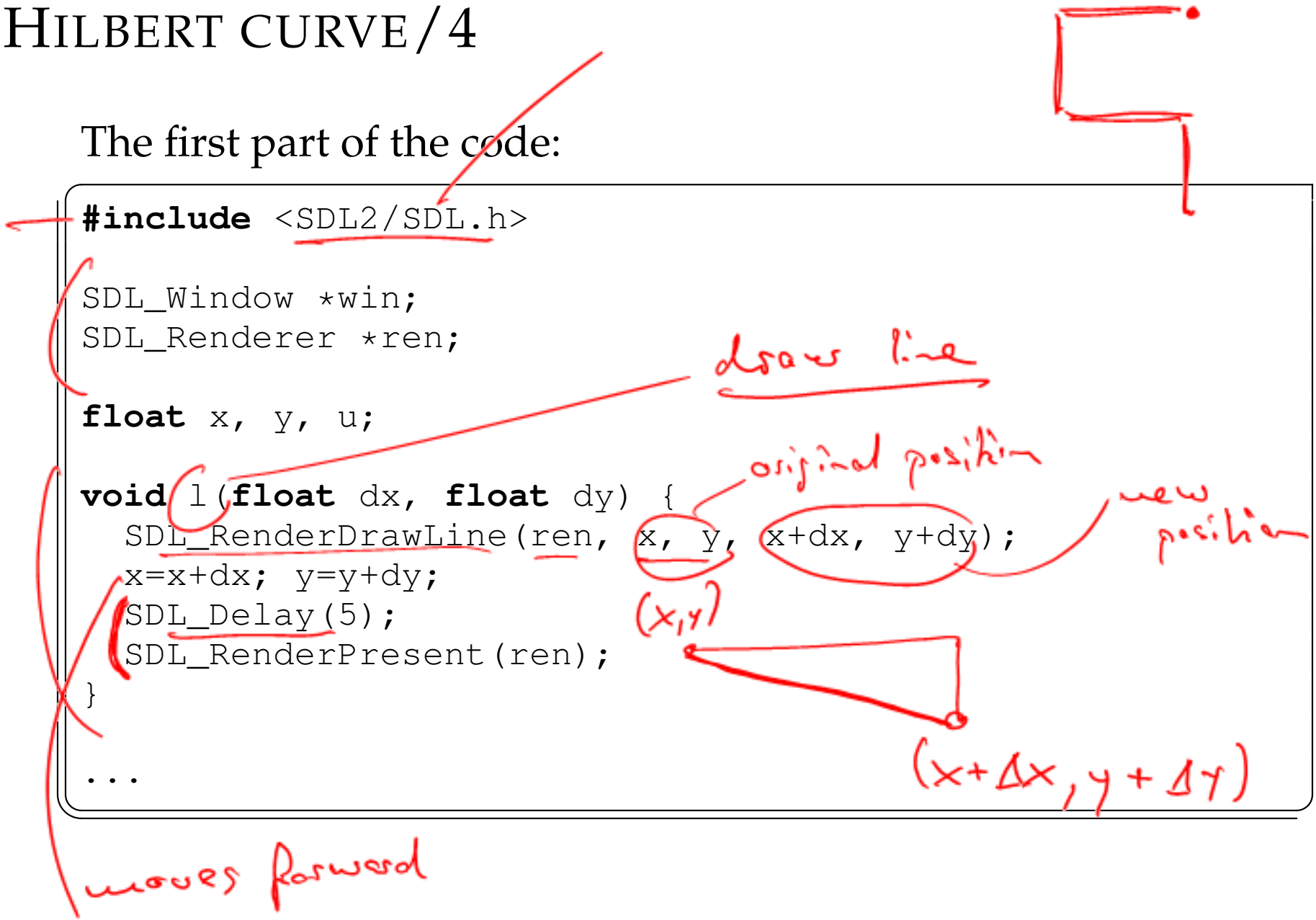
```
#include <SDL2/SDL.h>

SDL_Window *win;
SDL_Renderer *ren;

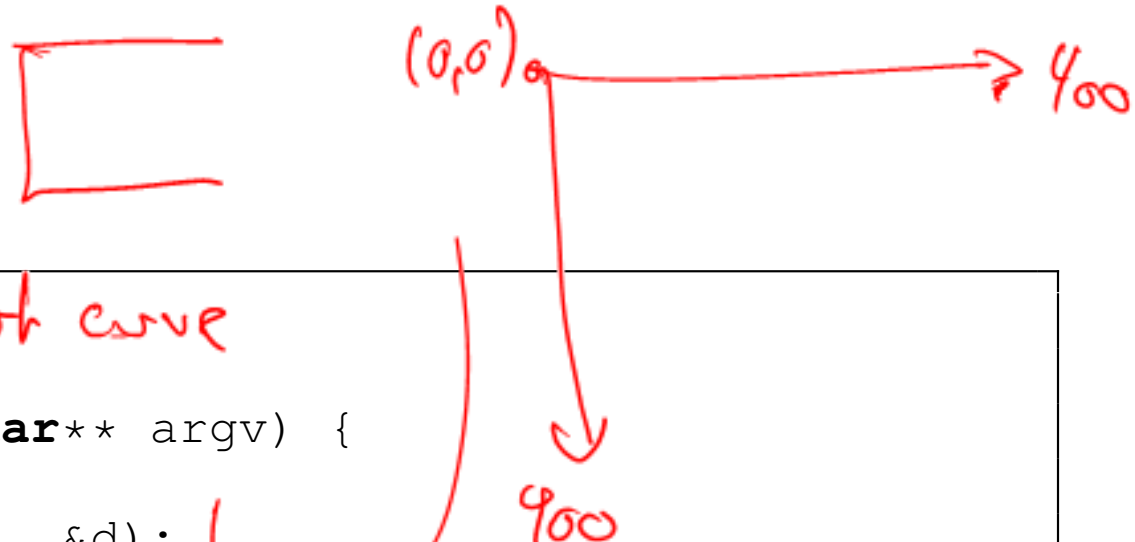
float x, y, u;

void l(float dx, float dy) {
  SDL_RenderDrawLine(ren, x, y, x+dx, y+dy);
  x=x+dx; y=y+dy;
  SDL_Delay(5);
  SDL_RenderPresent(ren);
}

...
```



HILBERT CURVE/5



The last part of the code:

```
...

int main (int argc, char** argv) {
  int d;
  sscanf(argv[1], "%d", &d);

  SDL_CreateWindowAndRenderer(400, 400, 0, &win, &ren);
  SDL_SetRenderDrawColor(ren, 255, 255, 255, 0);
  x=395; y=5; u=390/(pow(2,d)-1);
  A(d);

  SDL_Event e;
  do { SDL_PollEvent(&e); } while (e.type != SDL_QUIT);
}

// gcc hilbert.c -o hilbert -lm -lSDL2; ./hilbert 2
```

HILBERT CURVE/6

The middle part of the code:

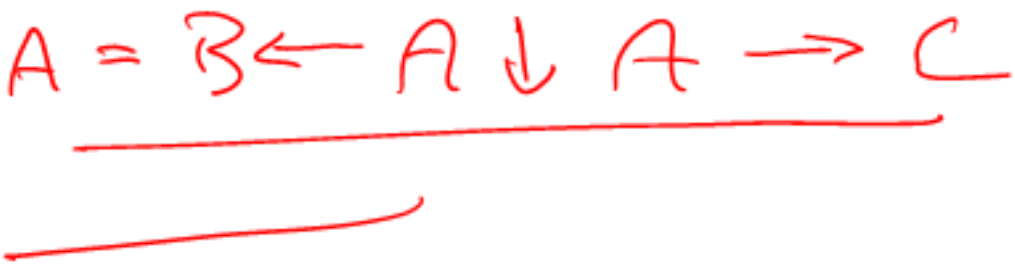
```
...

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

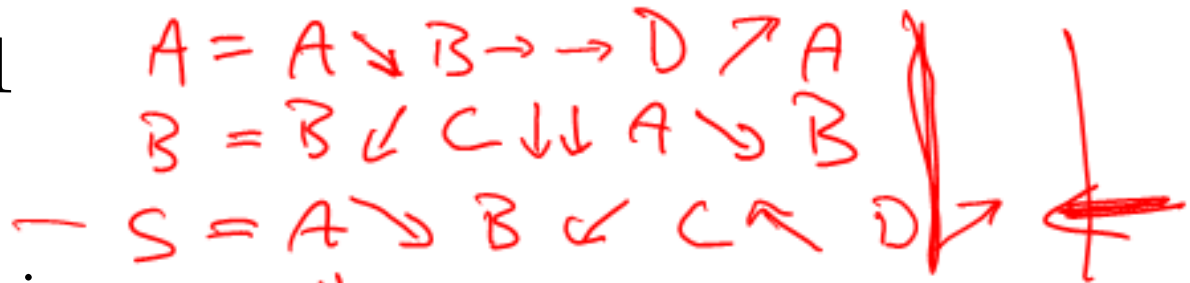
void A(int i) {
  if (i>0) {
    B(i-1); l(-u,0);
    A(i-1); l(0,u);
    A(i-1); l(u,0);
    C(i-1);
  }
}


void B(int i) { ...

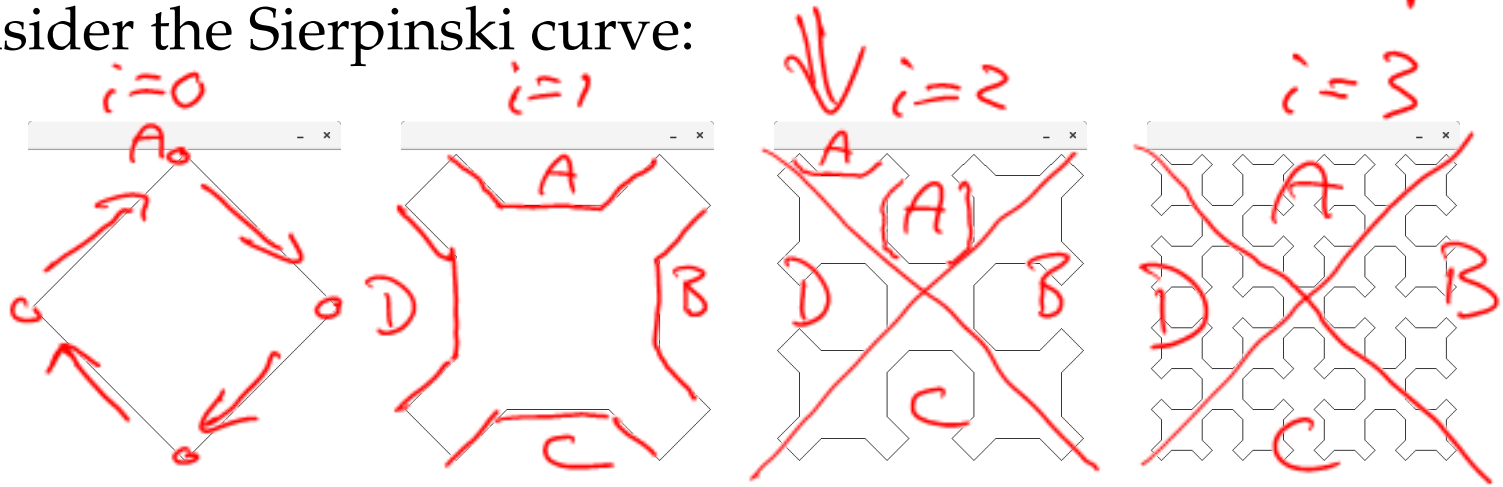
...
```



EXERCISE SL01-6/1



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 > 0) then

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

C(i); l(-u,-u); D(i); l(u,-u).

Algo: A(i)

if i > 0 then

A(i-1); l(u,u); B(i-1); l(2*u,0);

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

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

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

usually "nr of elements" and "nr of operations" is good enough
- decide what to consider / not consider
- all details is not good / possible

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

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

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;
```

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



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

Exact running time

Sum of products of rightmost columns (cost, times)

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

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: $t_i = 1$

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

all n together

a

b

linear cost (linearly on 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: $t_i = i$

$$\begin{aligned} T(n) = c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1\right) \\ + (c_5 + c_6)\frac{(n-1)n}{2} = an^2 + bn + c \end{aligned}$$

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

quadratic in n

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

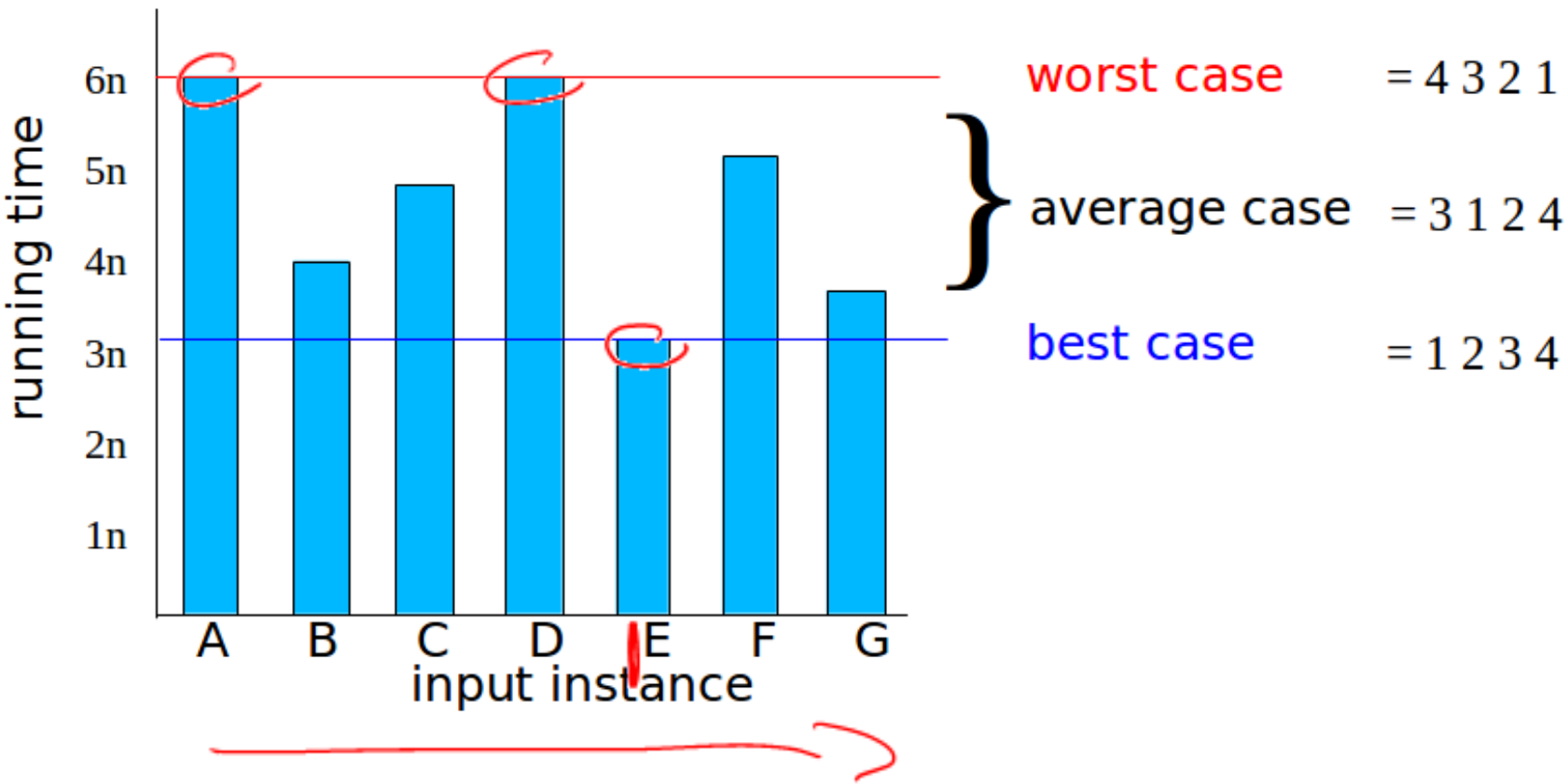
Best, worst and average case/1

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

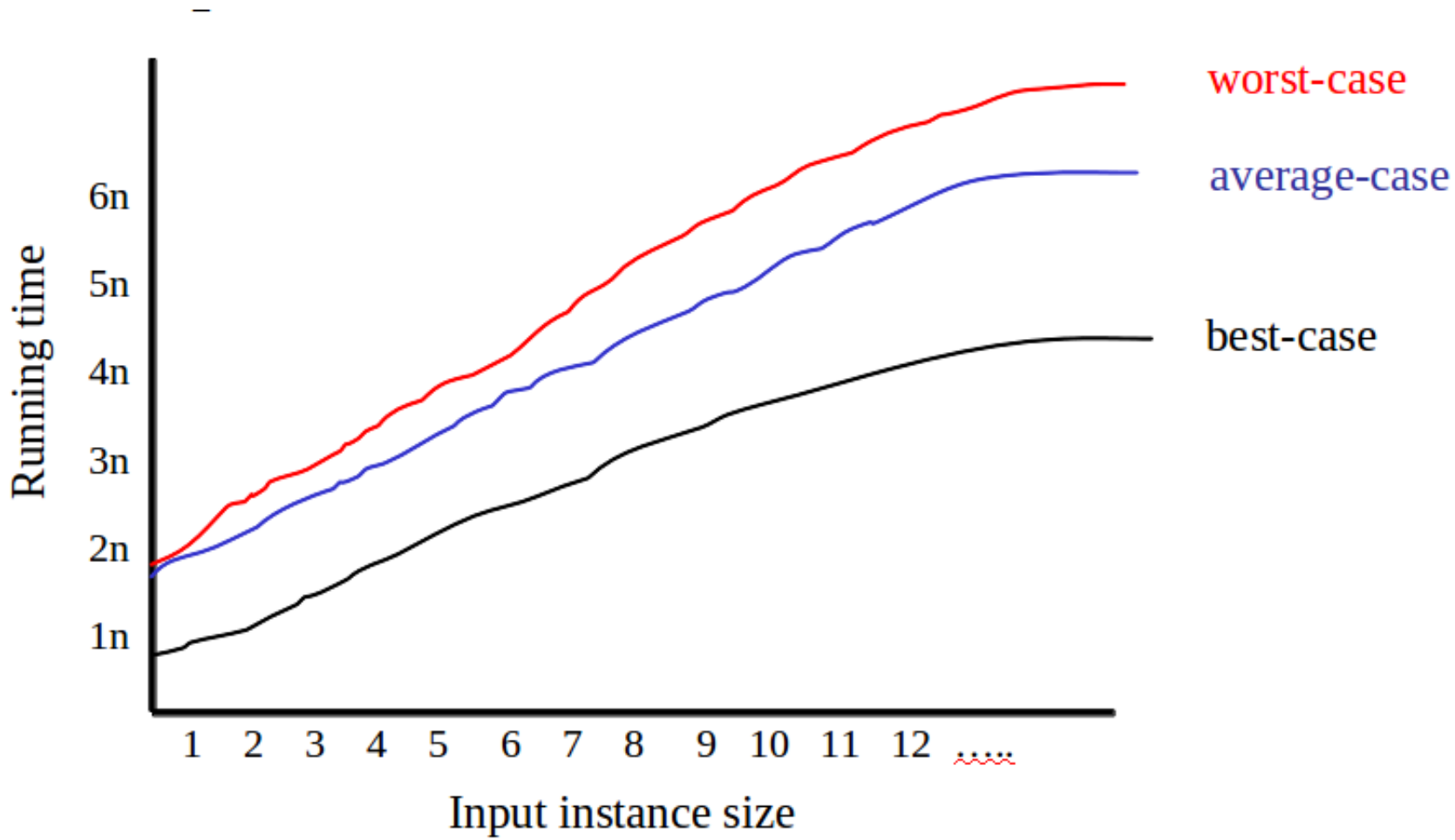
How can we modify almost any algorithm to have a good best-case running time?

- ex: check if array is sorted and do nothing (linear)
- reverse sorted
- chess: store many positions & solutions

- general: first test for special case and do solution for special case
- good and valid approach (but it can be used for dealing with benchmarks)

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≤n ∧ A[i]≠v do i = i+1;
if i≤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≤n ∧ A[i]≠v do i = i+1;
if i≤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

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

output is 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

output 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 $\log_2 20'000'000 \approx 24$

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 find position fast
- we still still have to move elements
- bin search does not improve runtime

EXERCISE SL02-4

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
  for j=1 to n2
    if A1[i] = A[i] then cnt++

cost $n_1 * n_2$

EXERCISE SL02-4

Plan: go through all elements of unsorted array and use binary search to look for element in sorted array

runtime: $\sum_{i=1}^{n1} \log_2(n2) = n1 \cdot \log_2(n_2)$

```
Algo: cntOcc(A1, A2)
      cnt = 0
      for i = 1 to n1
          if Bi_Search(A2, A1[i]) ≠ NIL then cnt++
      return cnt
```

Duplicates? A1 = [5, 3, 5], A2 = [1, 3, 3, 5]
what should result be? cnt?
different problem! define problem!

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



CORRECTNESS/2

Partial versus total



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

CORRECTNESS/2

Partial versus total



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

⟹ use proofs on small parts of the code

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

A=[5,5,5] A=[]

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

precise? no!

CORRECTNESS/4

• $A=[5]$ $m1=5, m2=5$ • $A=[5,7,5,5,8]$ $m1=5$ $m2=7$

• $A=[5,5,5]$ $m1=5, m2=5$ • $A=[\,]$ 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 is smaller equal than all numbers in A

m2 is smaller equal than all numbers in A different from m1

all numbers in A are equal

precise

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 $i \rightarrow i+1$
  - 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

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

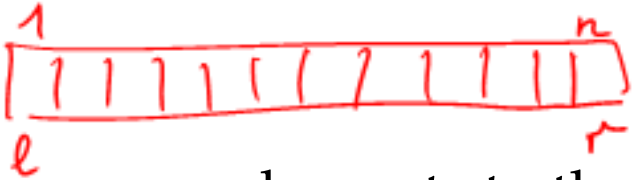
CORRECTNESS OF BINARY SEARCH/2

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

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;
```



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

```
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:

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

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 $\lfloor (l+r)/2 \rfloor - 1 < r$

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

EXERCISE SL02-5

for i=1 to (n-1) do
k=i
...



Formulate the loop invariant for the outer loop of the selection sort algorithm. What happens if in the selection sort algorithm the outer loop runs up to n?

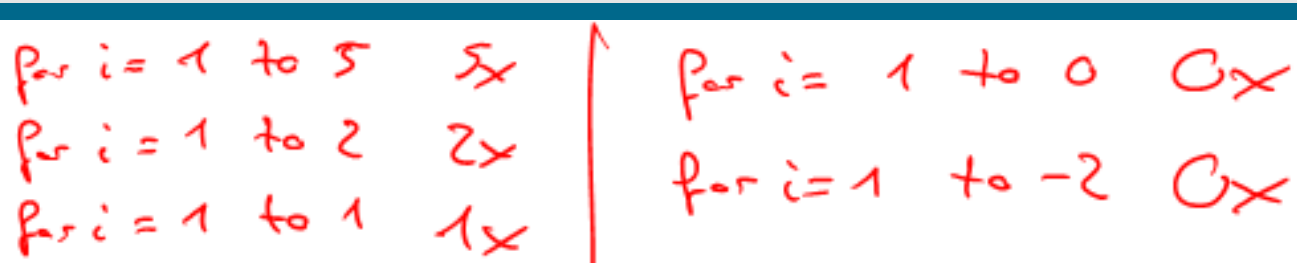
Invariant: elements in A[1..i-1] are already sorted and they are smaller than all other elements in A

$\forall j \in [1..i-2]: A[j] \leq A[j+1]$ (= A[1..i-1] is sorted)

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



FOR LOOPS



- In pseudo code the for loop can run **up** (we call this *up loop*) or **down** (we call this *down loop*).
- We have an up loop if the loop runs from low to high:
  - Example of an up loop: Array A[1..n]; for j=i 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 of a down loop: Array A[1..n]; for j=n to i 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

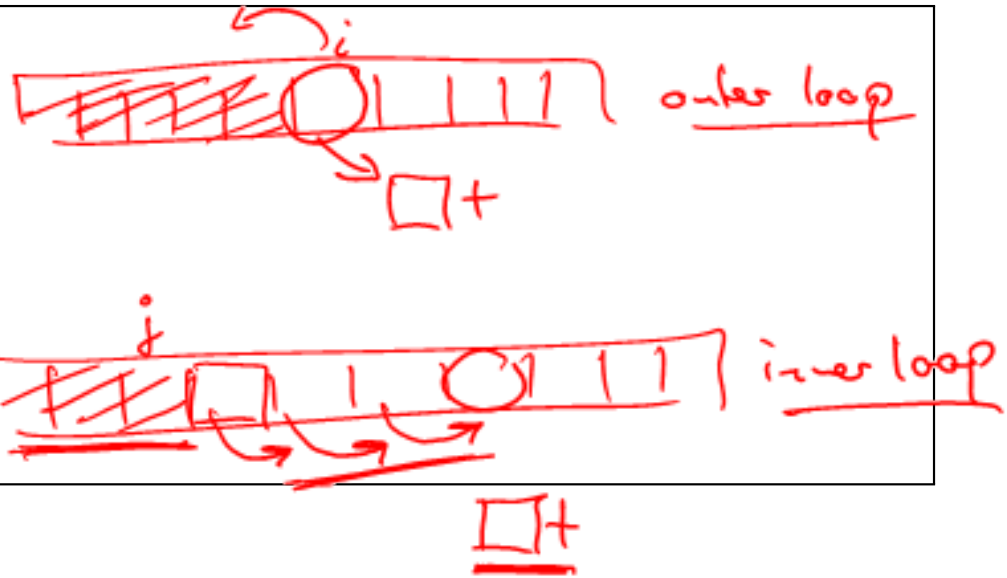
```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

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

```
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:
  - $A[1 \ldots i-1]$ sorted
  - $A[1 \ldots i-1] \in A^{orig}$
- Inner loop:
  - $A[1 \ldots j], t, A[j+1 \ldots i-1]$
  - $t < A[k]$ for $j+1 \leq k \leq i-1$
  - $A[1 \ldots j] \circ A[j+1 \ldots i-1]$ sorted

concatenation

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)

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
- $A[1..j-1], t, A[j, j+1..i-1]$ satisfies condition since $A[j] > t$ and $A[1..i-1]$ is sorted

Correctness of insertion sort / 4

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;
```

Termination:

- 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
- $j = 0$: $A[t, 1..i-1] = A[1..i]$ is sorted

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

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

MATHEMATICS REFRESHER/2

- Geometric progression
  - 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 progression
  - 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):

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

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: $\log_a b = {}^a\log b$

$$\log_2(n) = \log_5(n) / \log_5(2) = \frac{1}{c} \log_5(n) = c \log_5(n)$$

changing the base of log = constant factor c in asymptotic

MATHEMATICS REFRESHER/5

Some typical functions
From $n = 10$ up to $n = 1M$



$\log n$	$\log^2 n$	$\sqrt{n}$	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	∞
13	169	100	1e4	1e5	2e6	1e6	1e8	∞	∞
16	256	316	1e5	2e6	3e7	3e7	1e10	∞	∞
19	361	1000	1e6	2e7	4e8	1e9	1e12	∞	∞

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

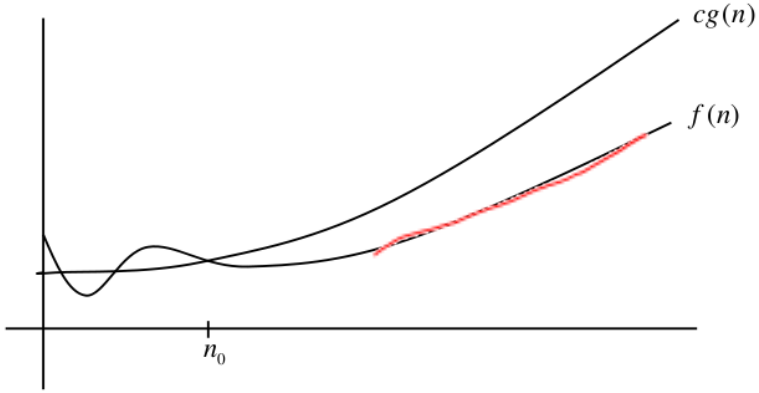
Big-O notation

- 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(n) multiplied with a constant is an upper bound for f(n)

EXERCISE SL02-6

Prove or disprove the following statements:

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

1. $2^{n+1} \stackrel{?}{\leq} c\,2^n$

$2 \cdot 2^n \stackrel{?}{\leq} c\,2^n$

true for, e.g., $c = 3$ and $n_0 = 1$

2. $2^{2^n} \leq c \cdot 2^n$

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

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

$2^n \not\leq c$ — constant

exponential $\not\in O$

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

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

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

$n \not\leq \frac{c^2}{4} \log^2 n$

ORDER OF GROWTH / 2

Big-Ω notation

- Asymptotic lower bound

Order of growth / 2

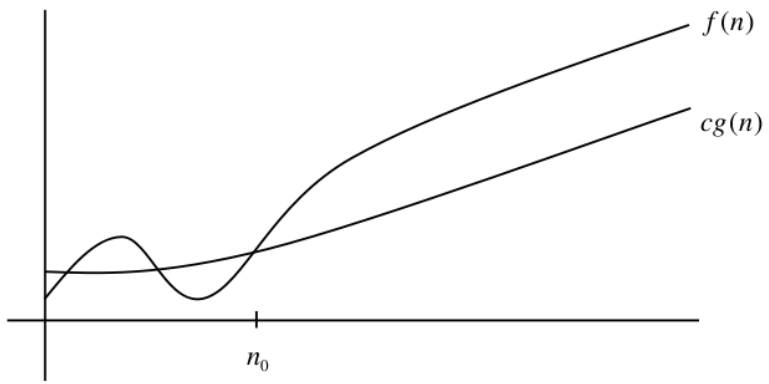
Big-Ω notation

- 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 $n \geq n_0$



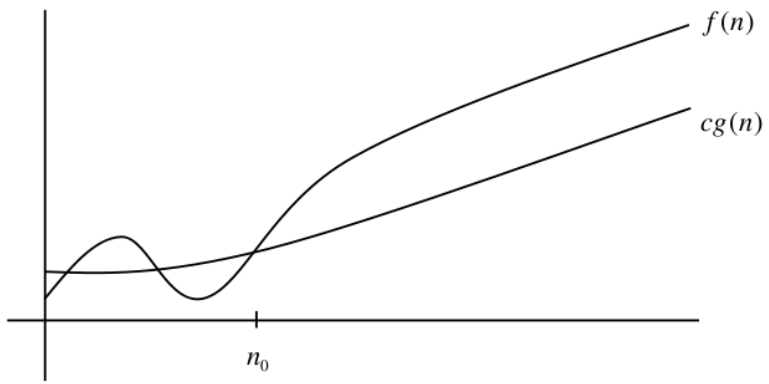
ORDER OF GROWTH/2

Ω: largest possible ∫

ω: possible ∫

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/ω: non-tight analogues of big-oh/omega

ORDER OF GROWTH / 3

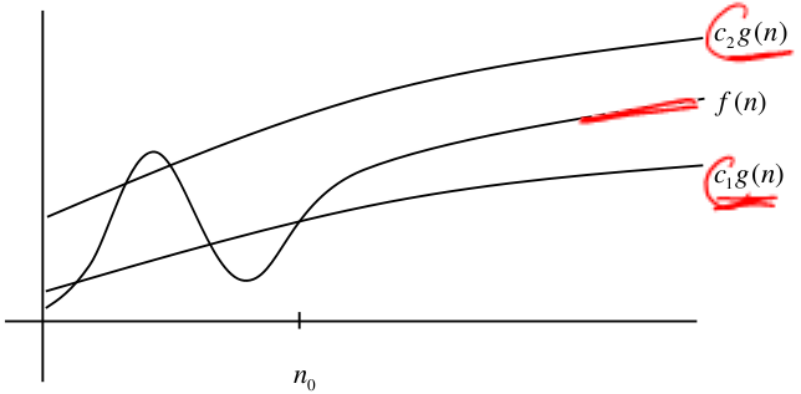
Big Θ notation

- Asymptotic tight bound

ORDER OF GROWTH / 3

Big Θ notation

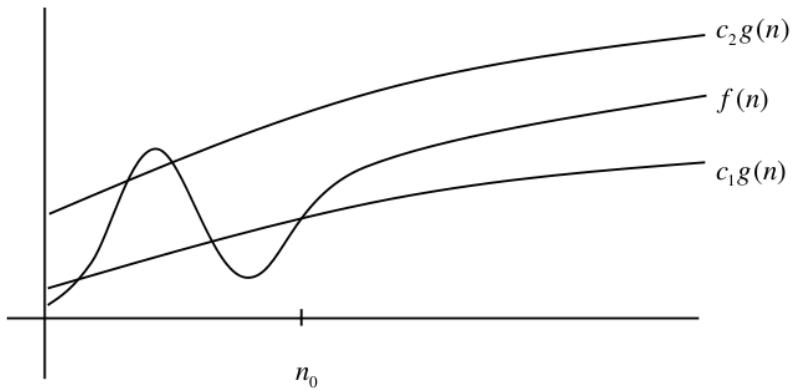
- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$



ORDER OF GROWTH / 3

Big Θ 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 $n \geq n_0$

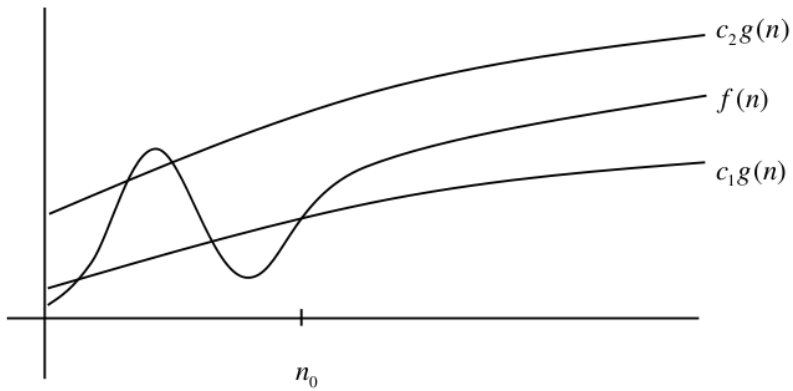


- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

ORDER OF GROWTH / 3

Big Θ 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

($n \to \infty$, a_k are constants)

Determine the asymptotic complexities of the formulas:

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

$n^{\log c} = 1$

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

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

$4^{\log_4 n / \log_4 10} = n^{1/\log_4(10)} \approx n^{0.6}$

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

$O(3n+5) = \{n, 100, \lceil n \rceil, 3on, n-6, \log n, n/7, \ldots\}$

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: $f(n) = O(g(n))$ 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 ...

$O(g(n)) = \{h(n) : \text{exists } c, n_0 \text{ such that } h(n) \leq c \cdot g(n)\}$

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 $O(n^2 \log n)$

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

Determining the maximal problem size:

Running Time T(n) in μs	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

EXERCISE SL02-8

bubble, selection, insertion

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

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.

$k=n$: $\frac{n^2}{2} - \frac{n}{2}$ (= sel sort)

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 acc of innermost part (if cmp)

$$\sum_{i=1}^{k} \sum_{j=i+1}^{n} 1$$

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

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

$i=1$: $n-1$

$i=2$: $n-2$

⋮

$i=k$: $n-k$

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

- Checking correctness of a program

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases
- Make sure the solution works in these cases

Special case analysis/1

Basic approach

- Checking correctness of a program
- 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

- Checking correctness of a program
- 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

Special case analysis/2

Special cases

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

Special case analysis/2

Special cases

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

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

A = [1, 7, 3]
n = 3

- empty array: n = 0, OK
(✓) possibly correct

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

CORRECTNESS OF BINARY SEARCH VARIANT / 1

other special cases:
empty array
1 elem array
odd # of elem

Use special cases analysis to analyze the following algorithm.

Algo: BinSearch2(A,v)

```
l = 1; r = n;
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 — l=1, r=2; 1 ✓
b. A=[5,7], v=5 — l=1, r=2; 1 ✓
c. v=6 — l=1, r=2; l=2, r=2; NIL ✓
d. v=7 — l=1, r=2; l=2, r=2; NIL ✗
e. v=8

CORRECTNESS OF BINARY SEARCH VARIANT/2

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;
```

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

l=1, r=2

l=1, r=1

A[0] is accessed

d. A=[5,7], v=7 l=1, r=2

l=2, r=2

NIL

CORRECTNESS OF BINARY SEARCH VARIANT/3

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;
```

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

$l=1$, $r=2$

$l=1$, $r=1$

$l=1$, $r=1$

∞ loop !

EXERCISE SL02-9

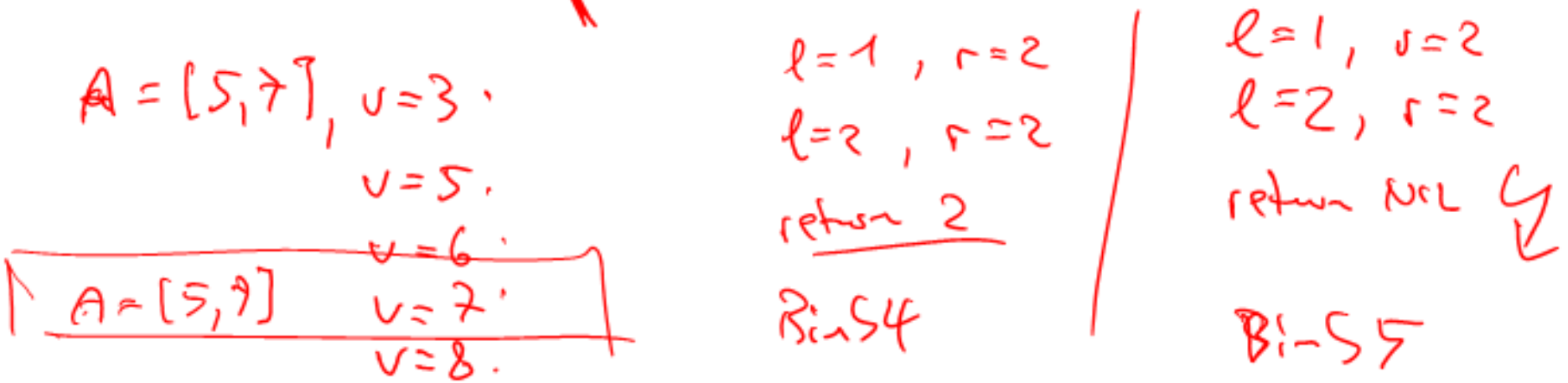
Use special case analysis to determine the correctness of BinarySearch4 and BinarySearch5.

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;
```



Outlook

This week

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

count # of exec

invariants (statements about which holds)

still count (for $n \to \infty$)

looking & running

represent examples

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 —

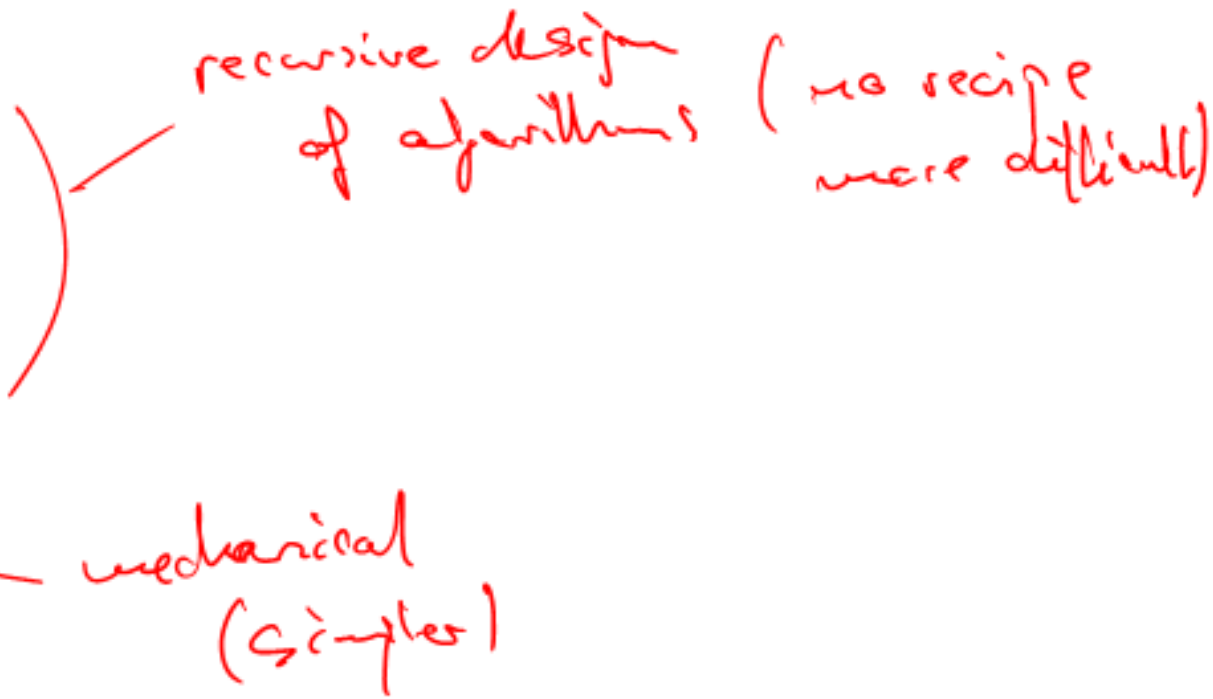
Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL03

1. Divide and conquer
   Merge sort
   Tromino tiling

2. Recurrences



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

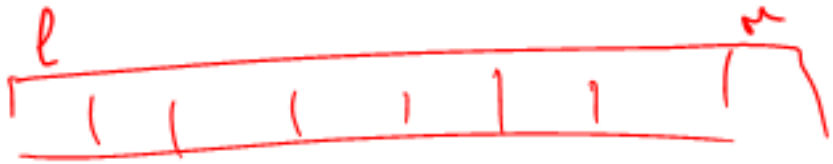
- Closely follows the divide-and-conquer paradigm
- To sort an n-element sequence proceed as follows
  - **Divide**: divide the sequence into two $n/2$-element sequences
  - **Conquer**: sort the two sequences recursively using merge sort
  - **Combine**: merge the two sorted sequences to produce the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

MERGE SORT/1

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

left right

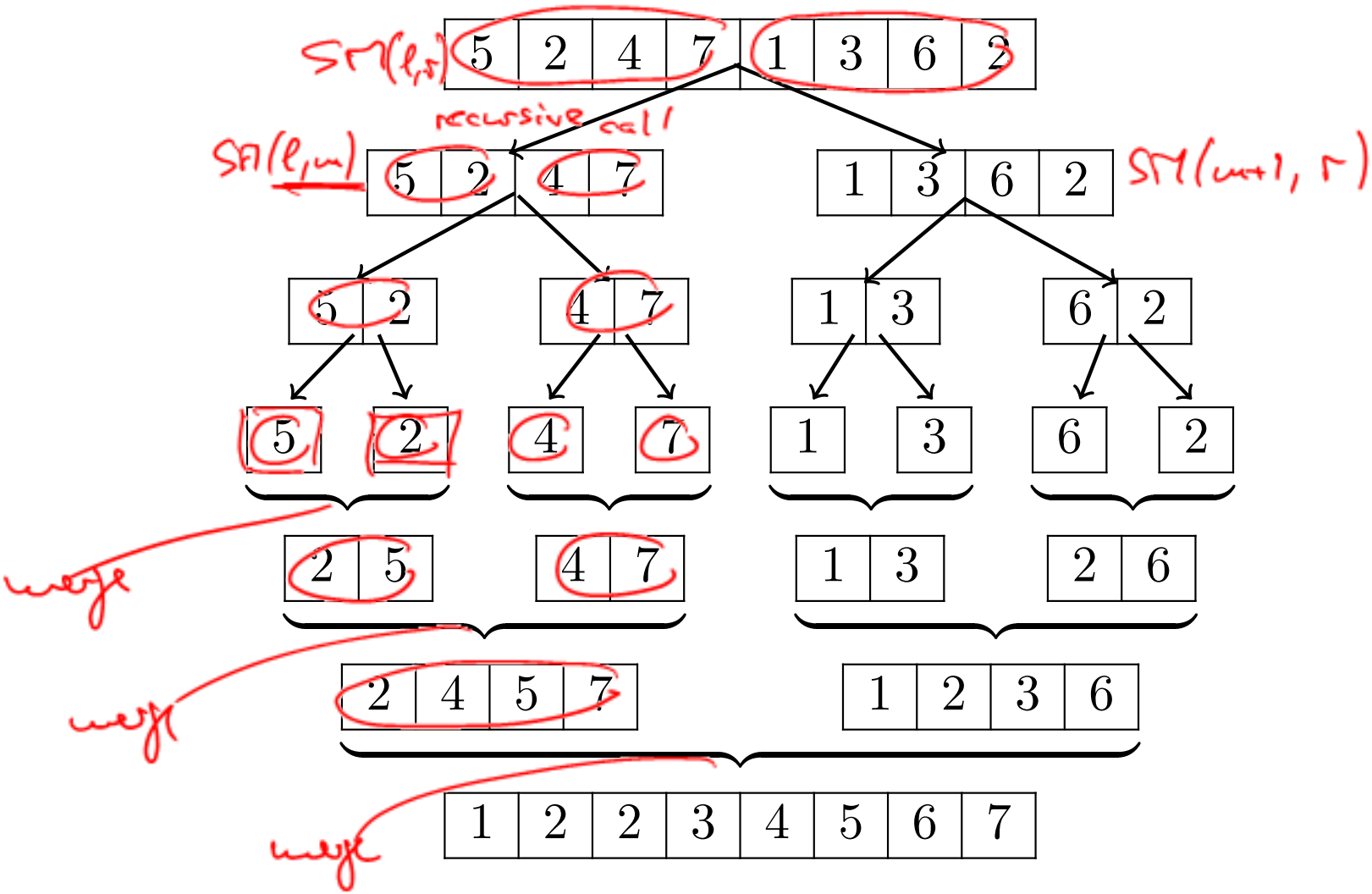
Algo: MergeSort(A,l,r)

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);                  ← key part (difficult part)
```

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

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

at beginning of for loop (1st iteration

$B =$

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

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

$B =$

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

$A =$

1	2	2	.	.	.	.	.

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

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

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

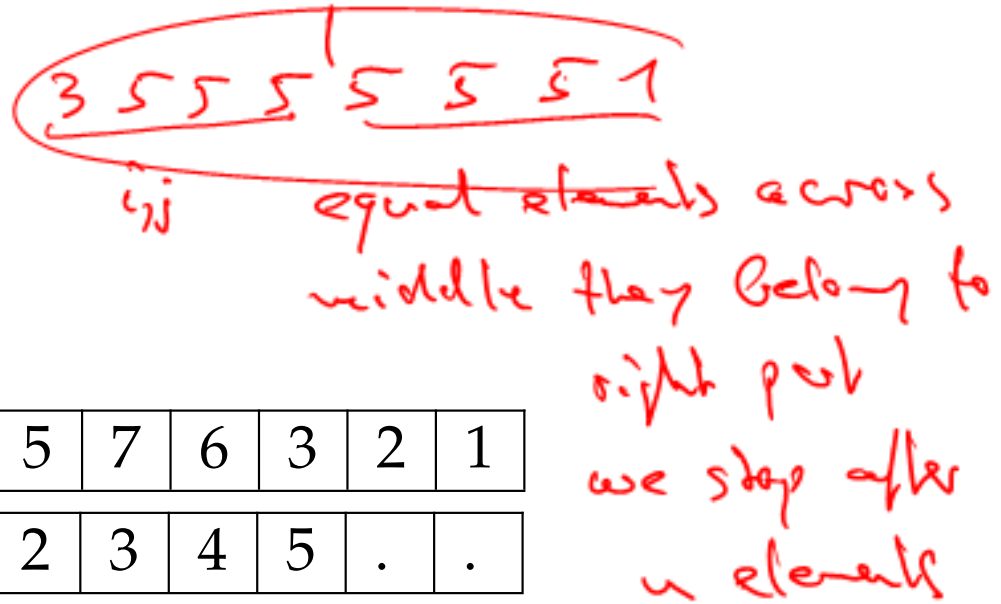
- $i = 4, j = 5, k = 7$

$B =$	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	.	.

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

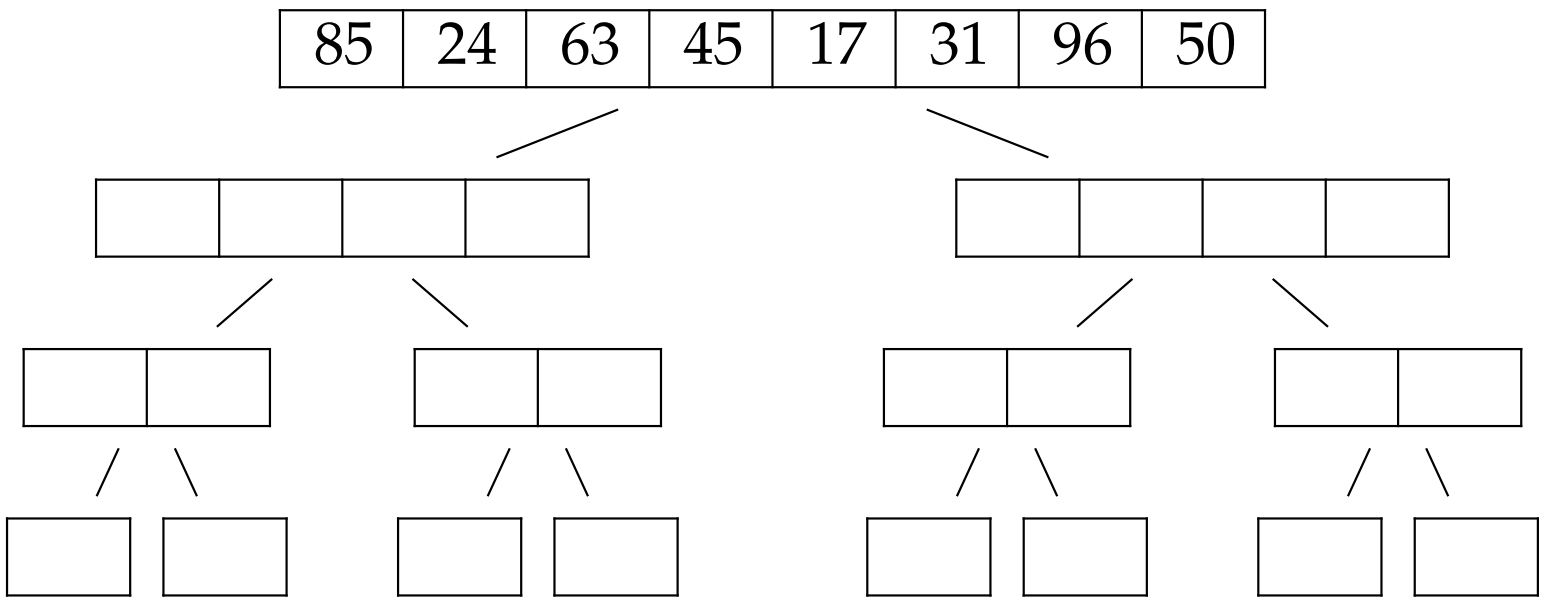
$B =$	2	4	5	7	6	3	2	1
$A =$	1	2	2	3	4	5	6	.

- After last iteration of for loop:

$A =$	1	2	2	3	4	5	6	7

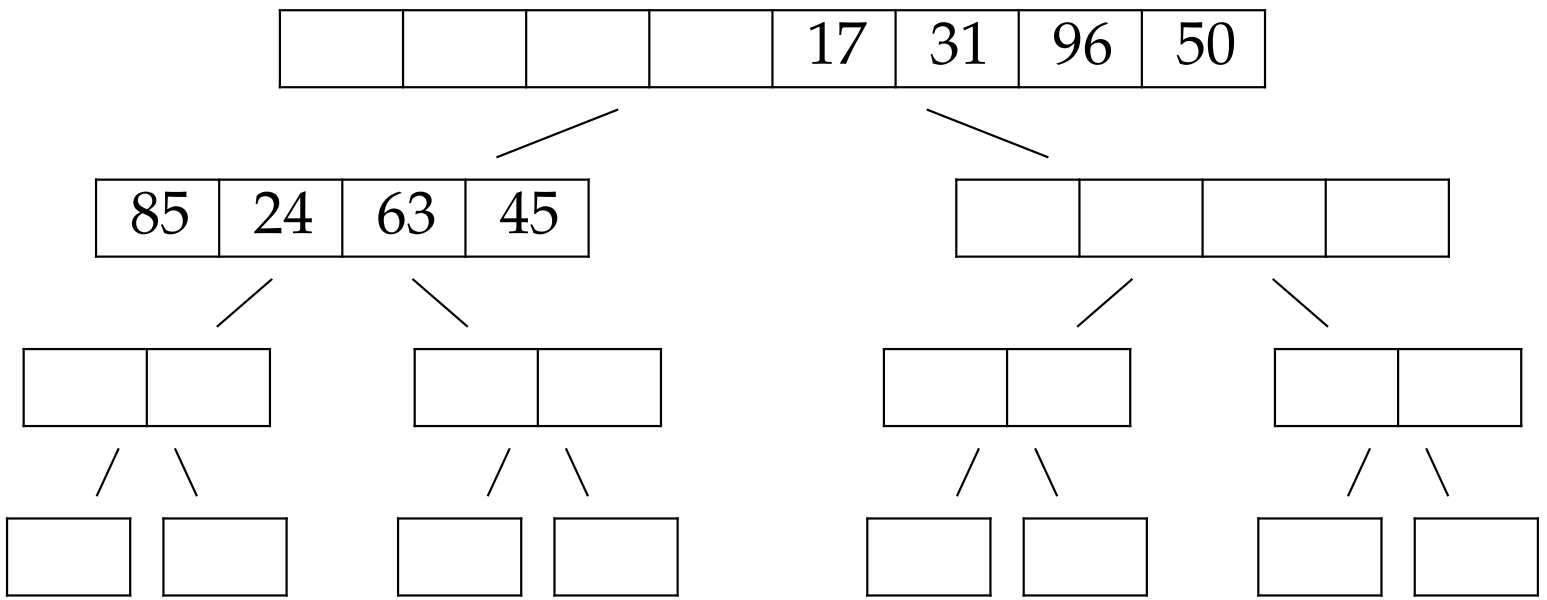
MERGE SORT/9

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



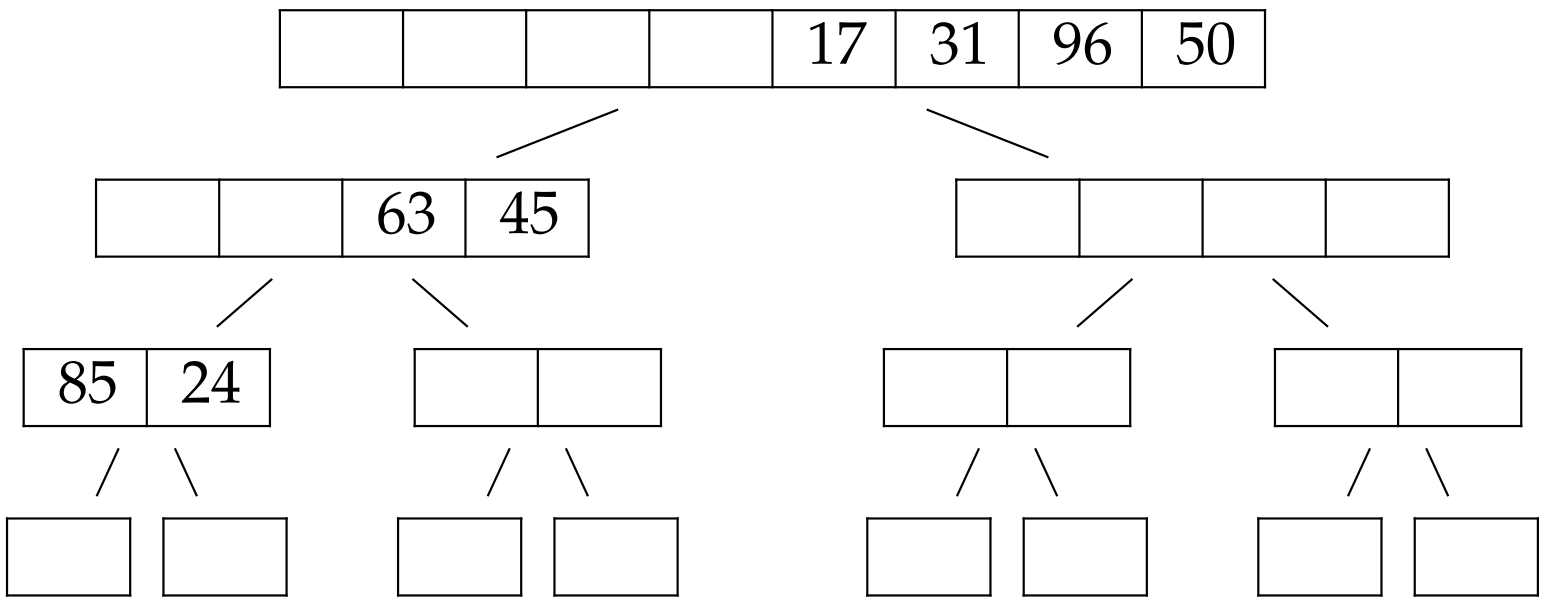
MERGE SORT/9

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



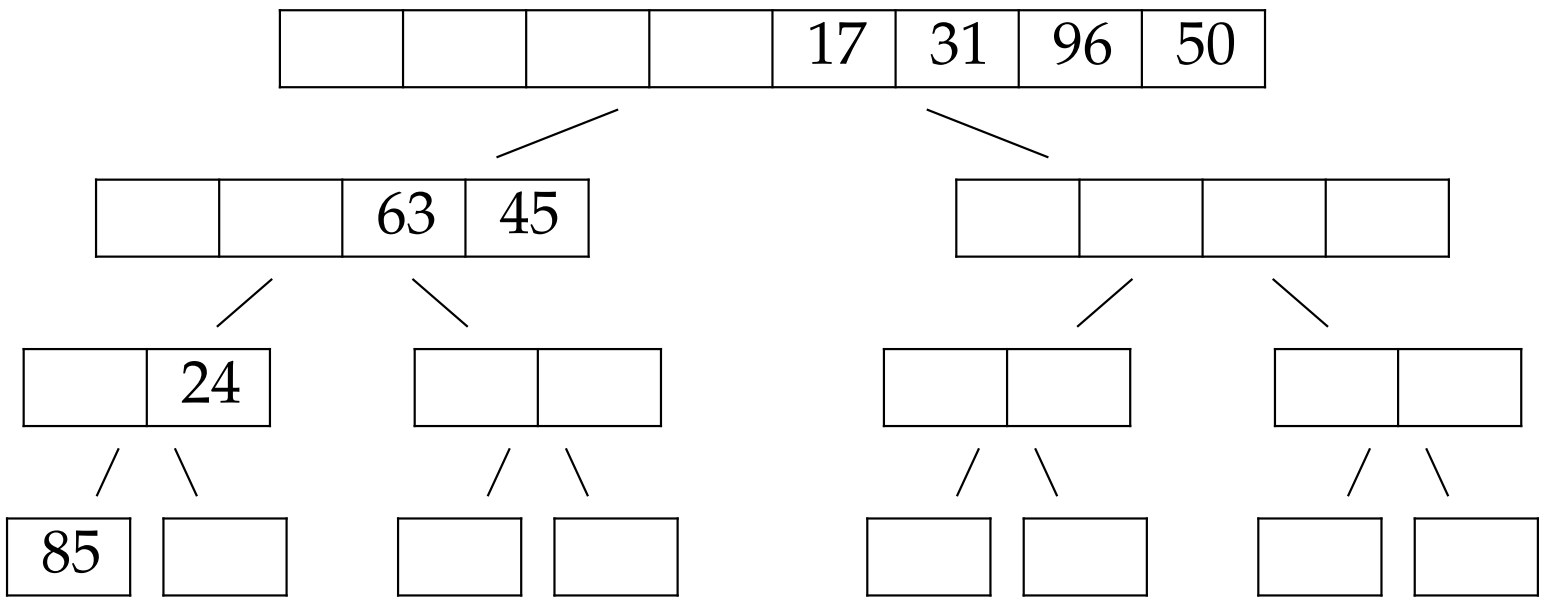
MERGE SORT/9

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



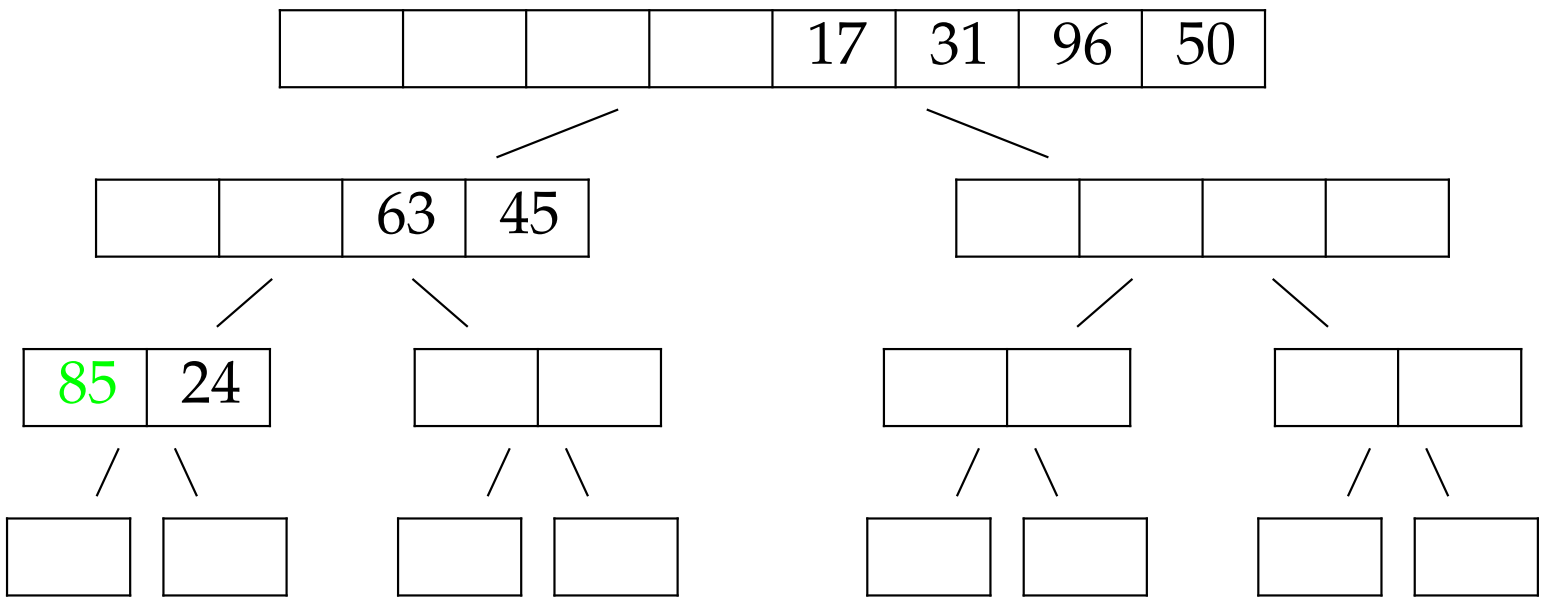
MERGE SORT/9

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



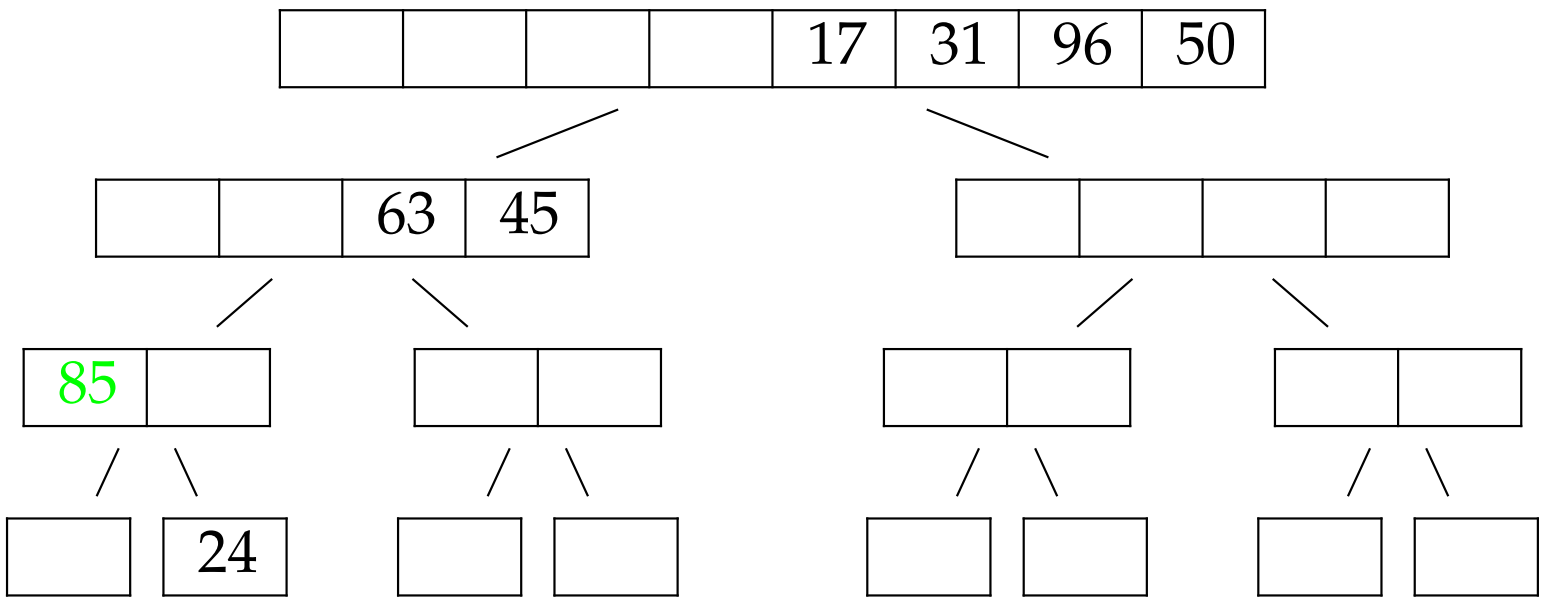
MERGE SORT/9

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



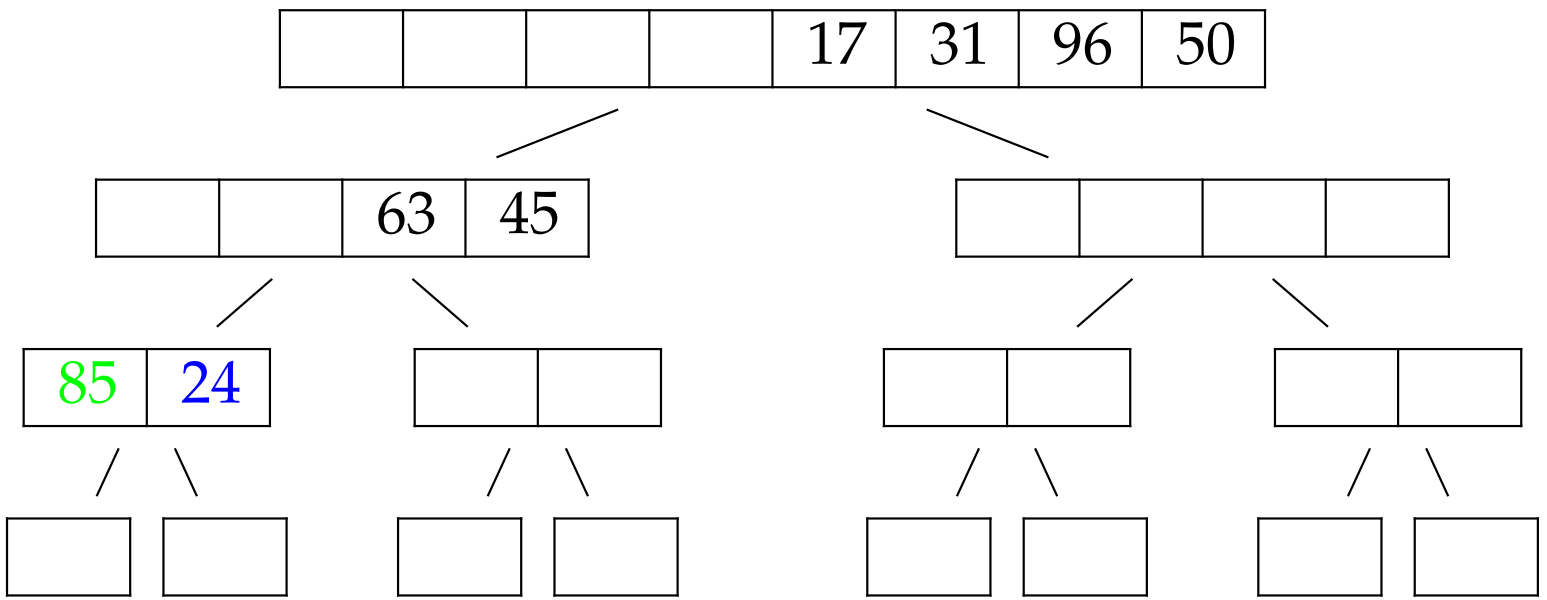
MERGE SORT/9

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



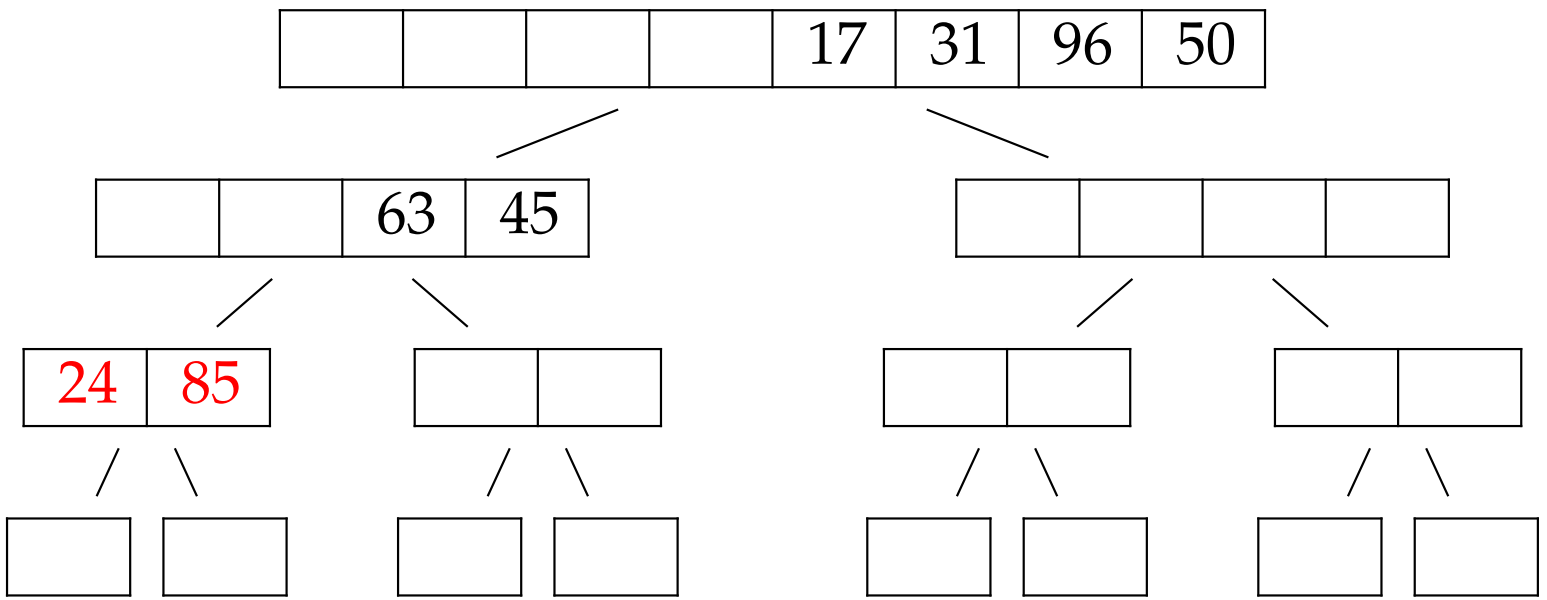
MERGE SORT/9

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



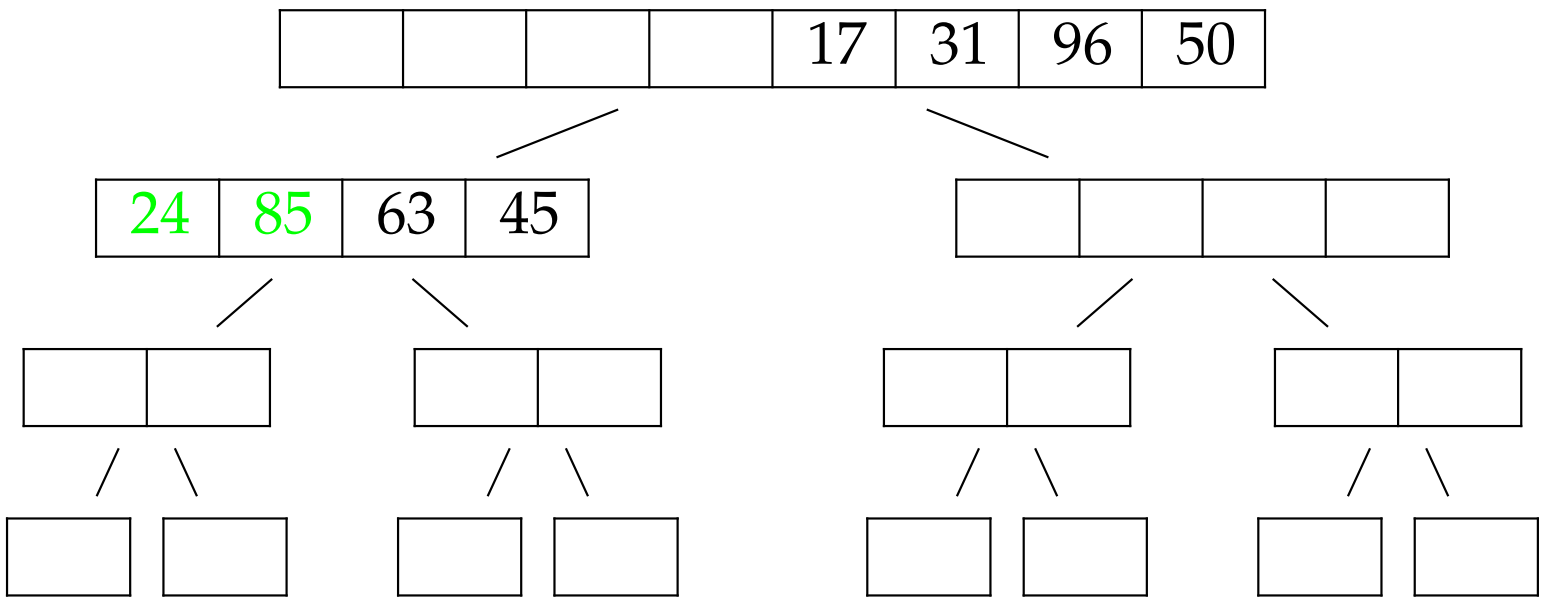
MERGE SORT/9

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



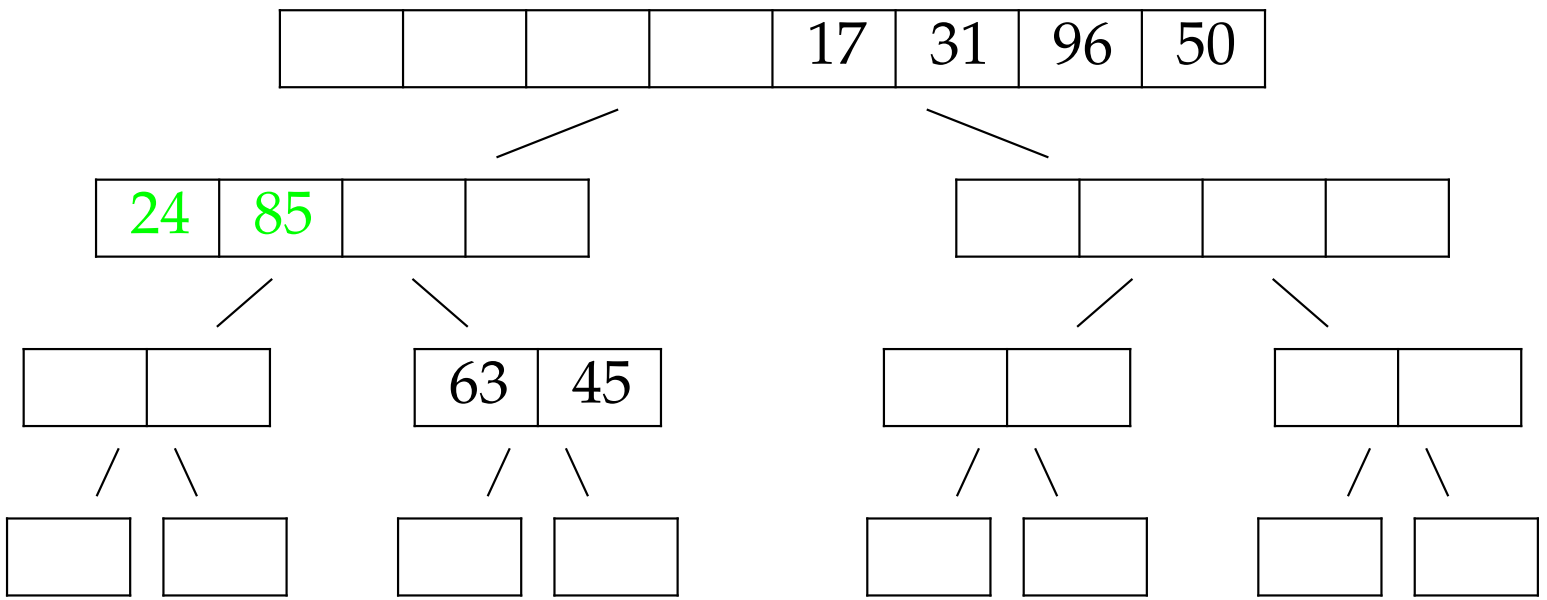
MERGE SORT/9

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



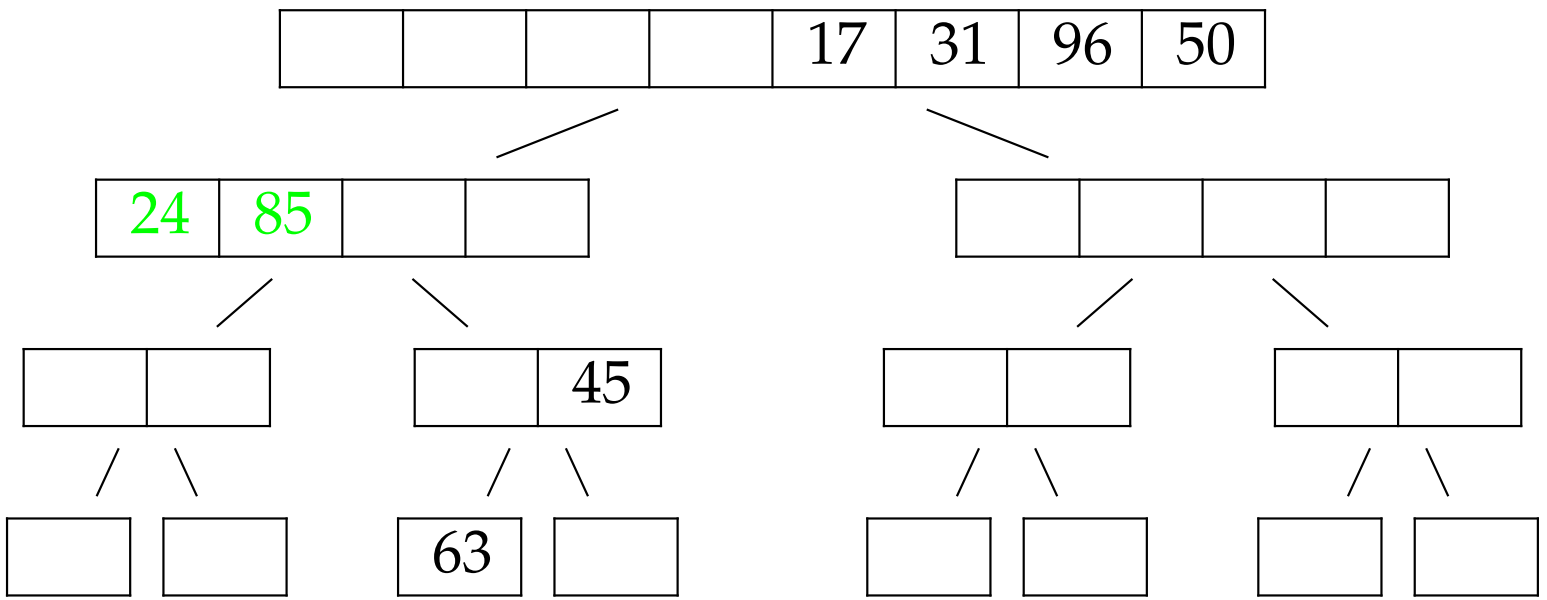
MERGE SORT/9

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



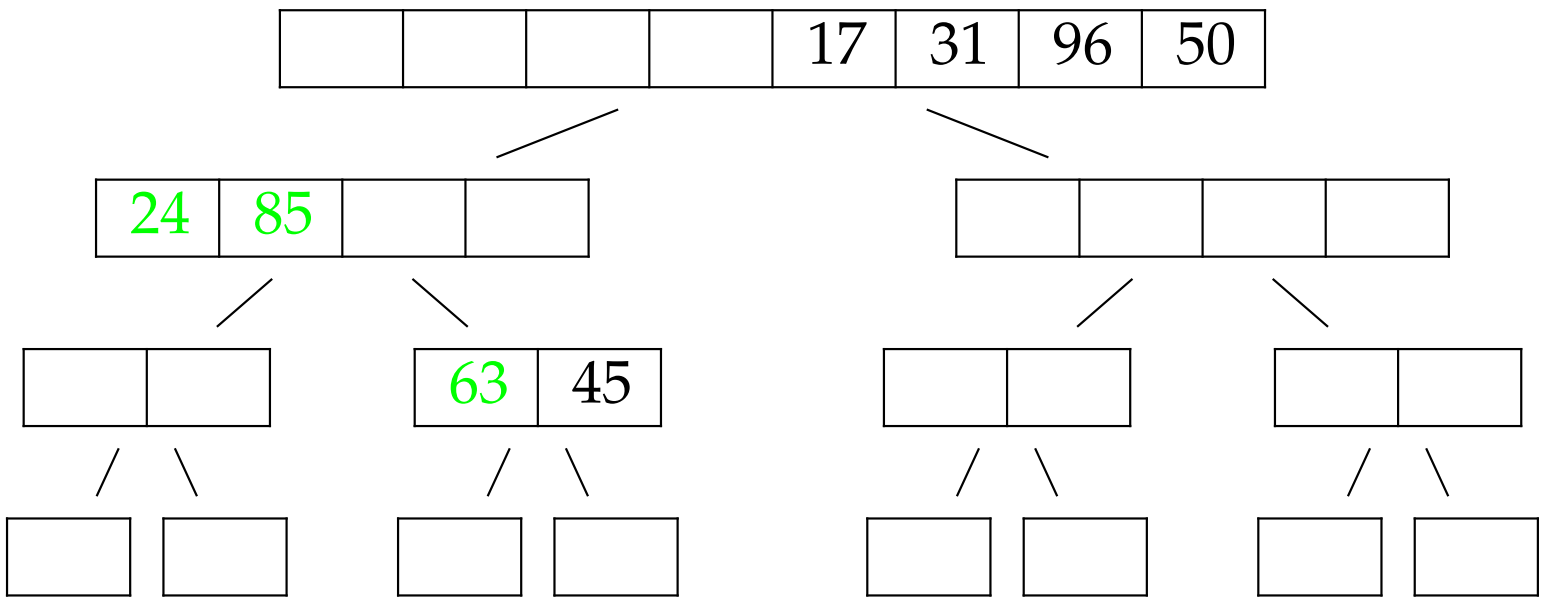
MERGE SORT/9

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



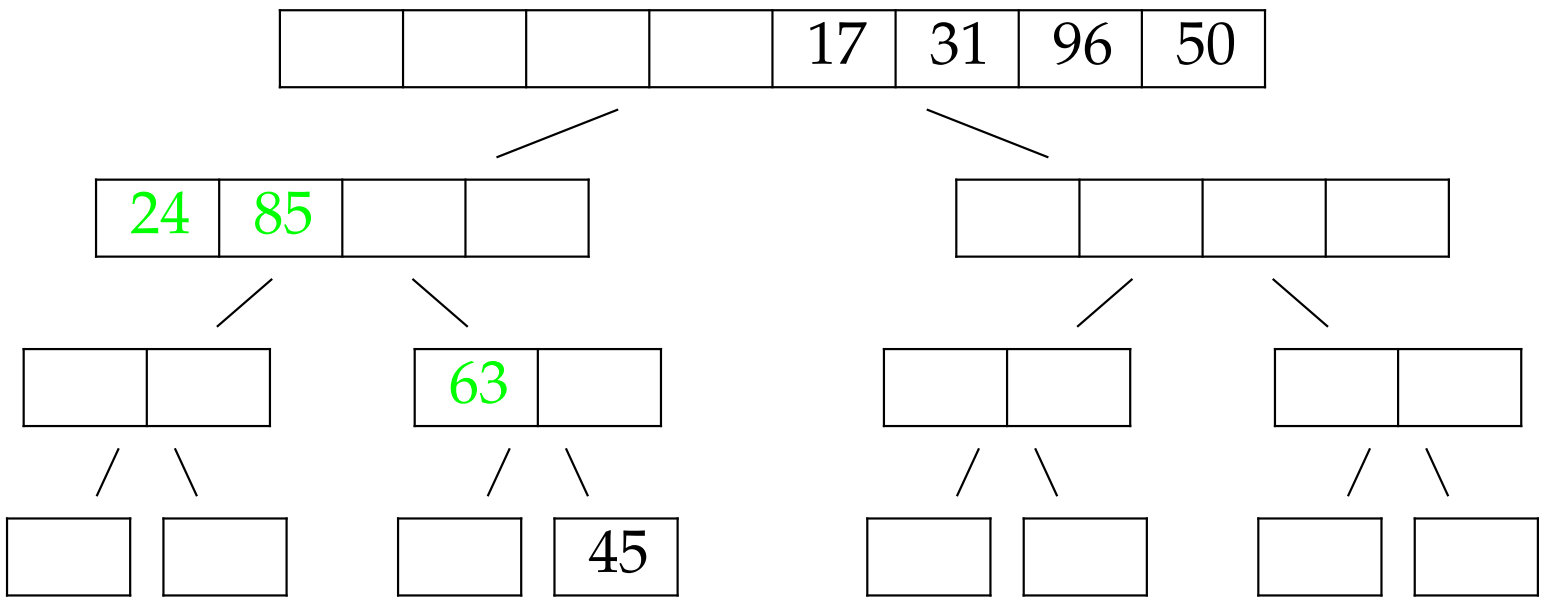
MERGE SORT/9

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



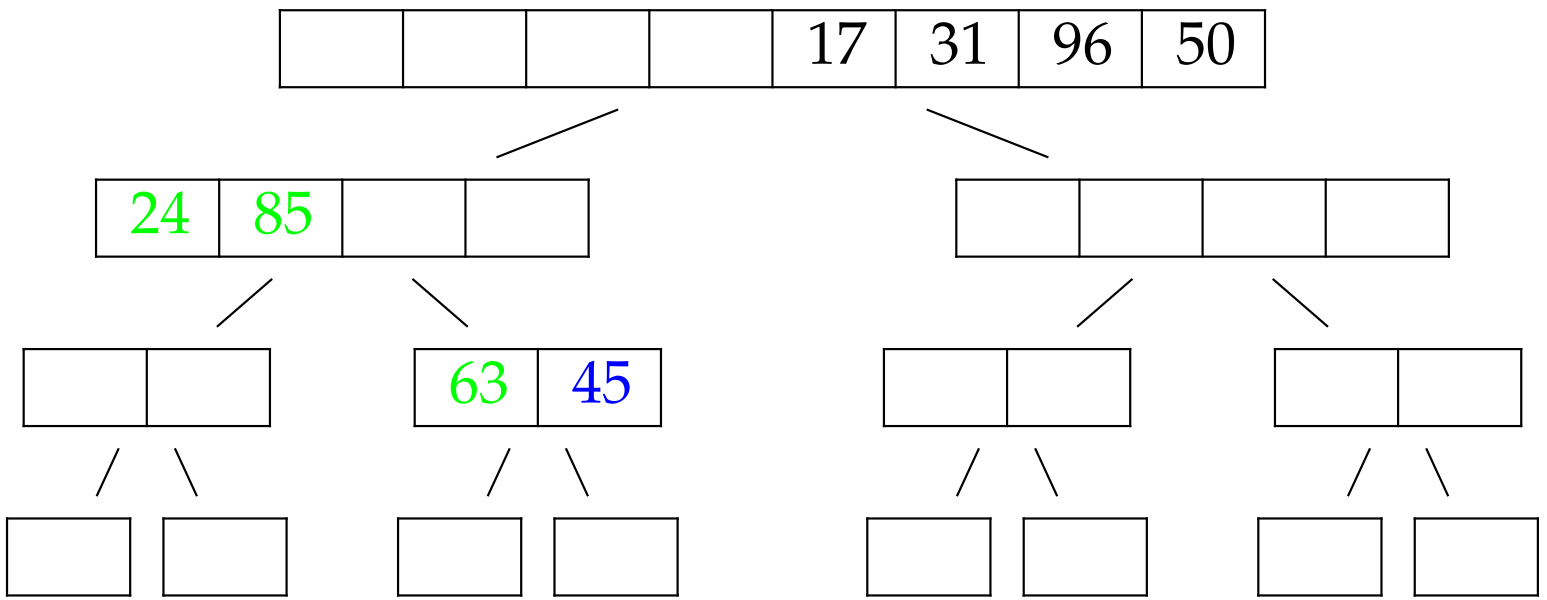
MERGE SORT/9

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



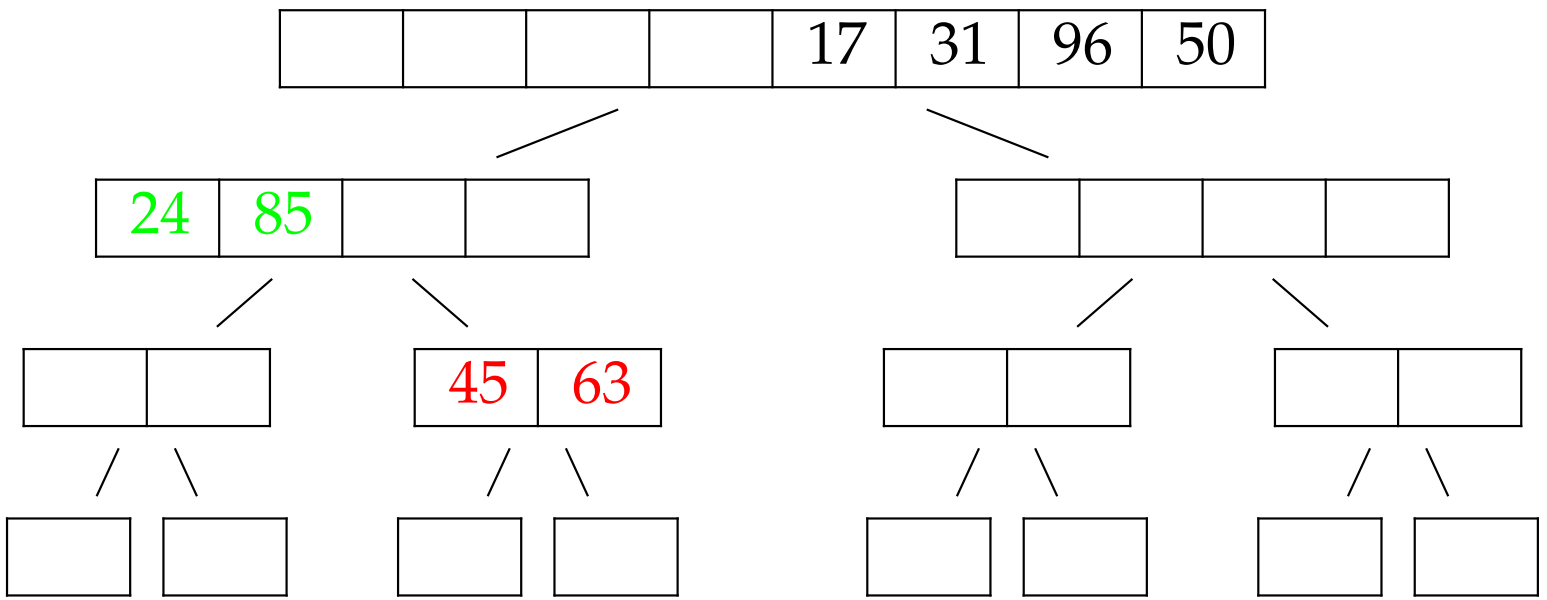
MERGE SORT/9

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



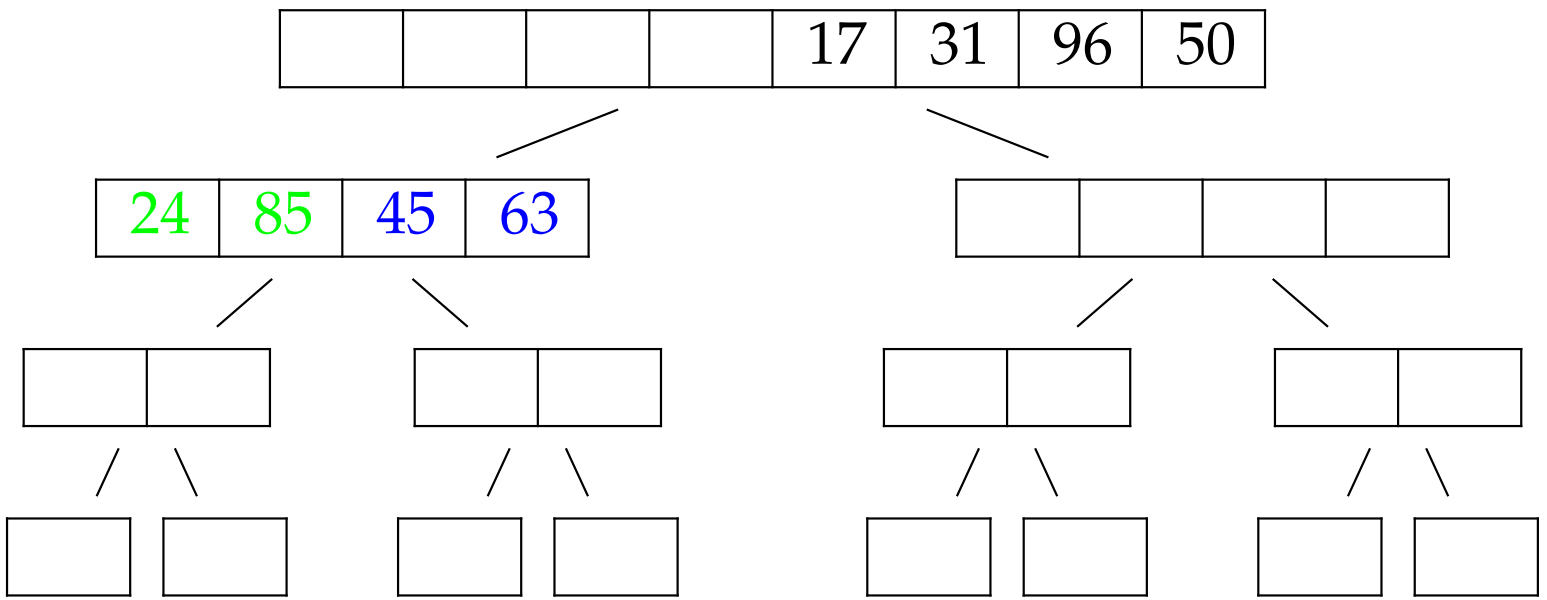
MERGE SORT/9

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



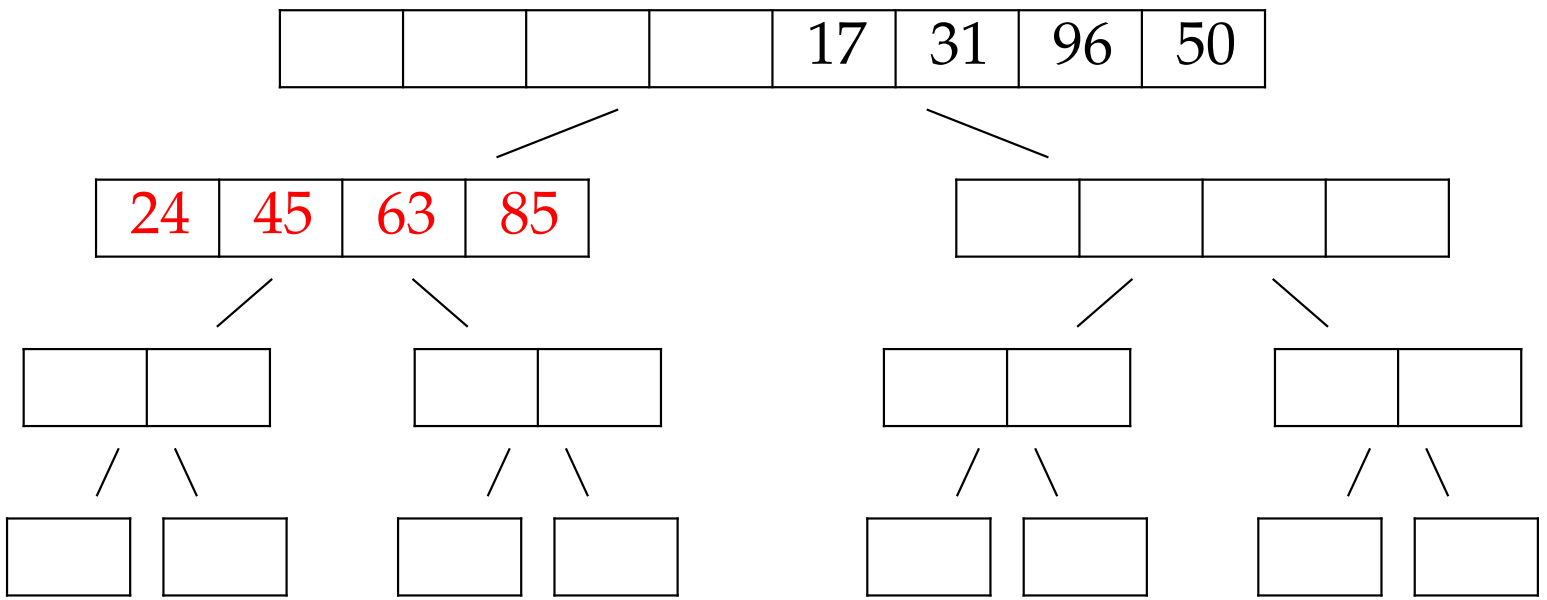
MERGE SORT/9

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



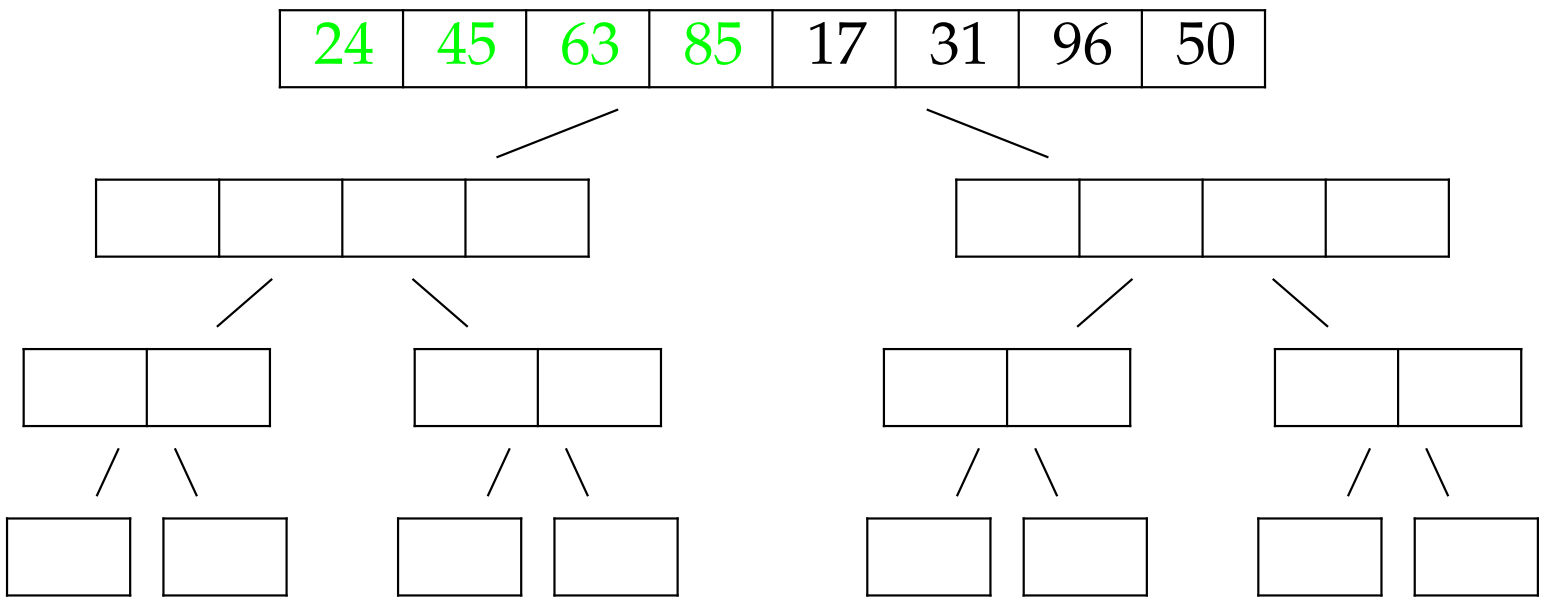
MERGE SORT/9

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



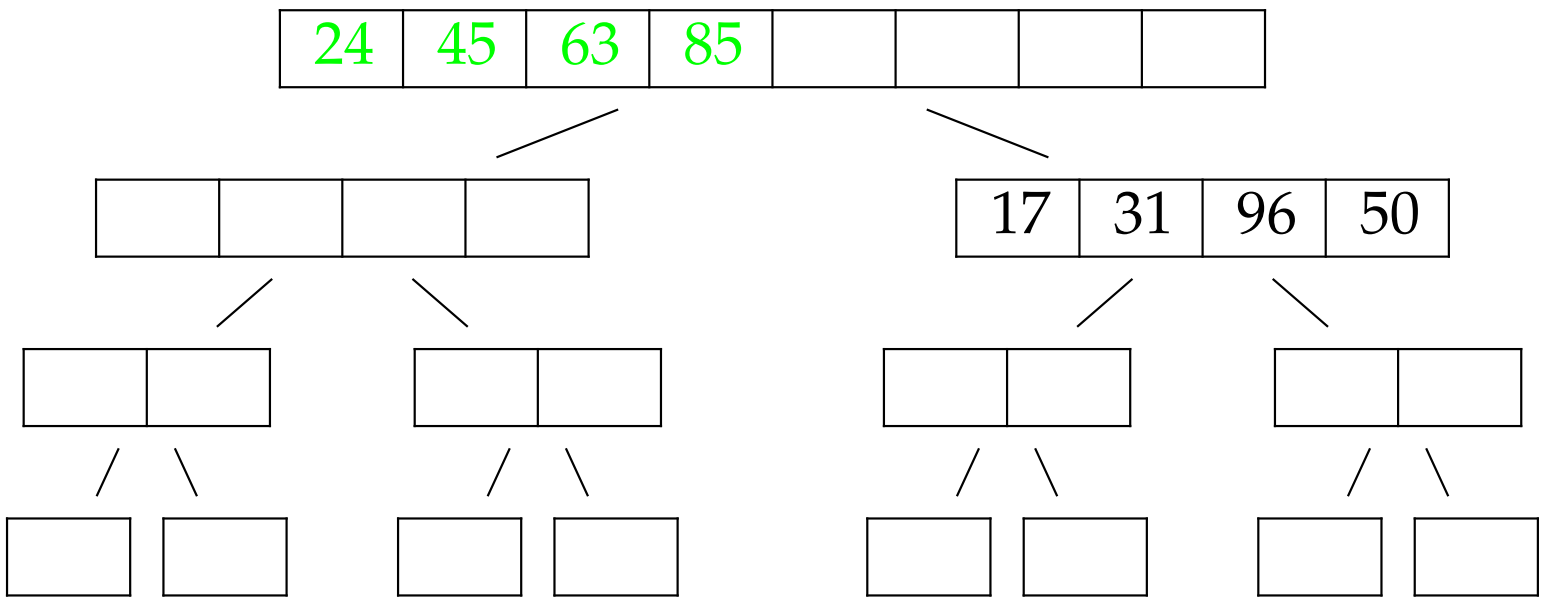
MERGE SORT/9

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



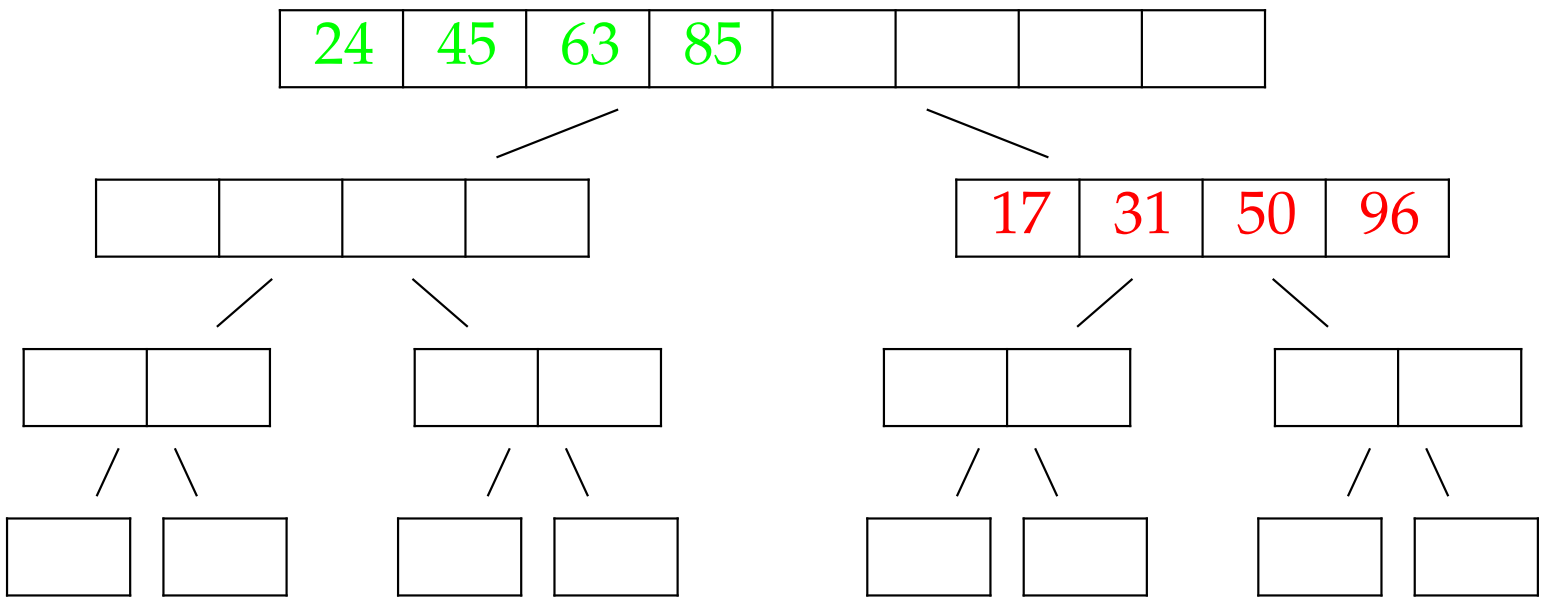
MERGE SORT/9

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



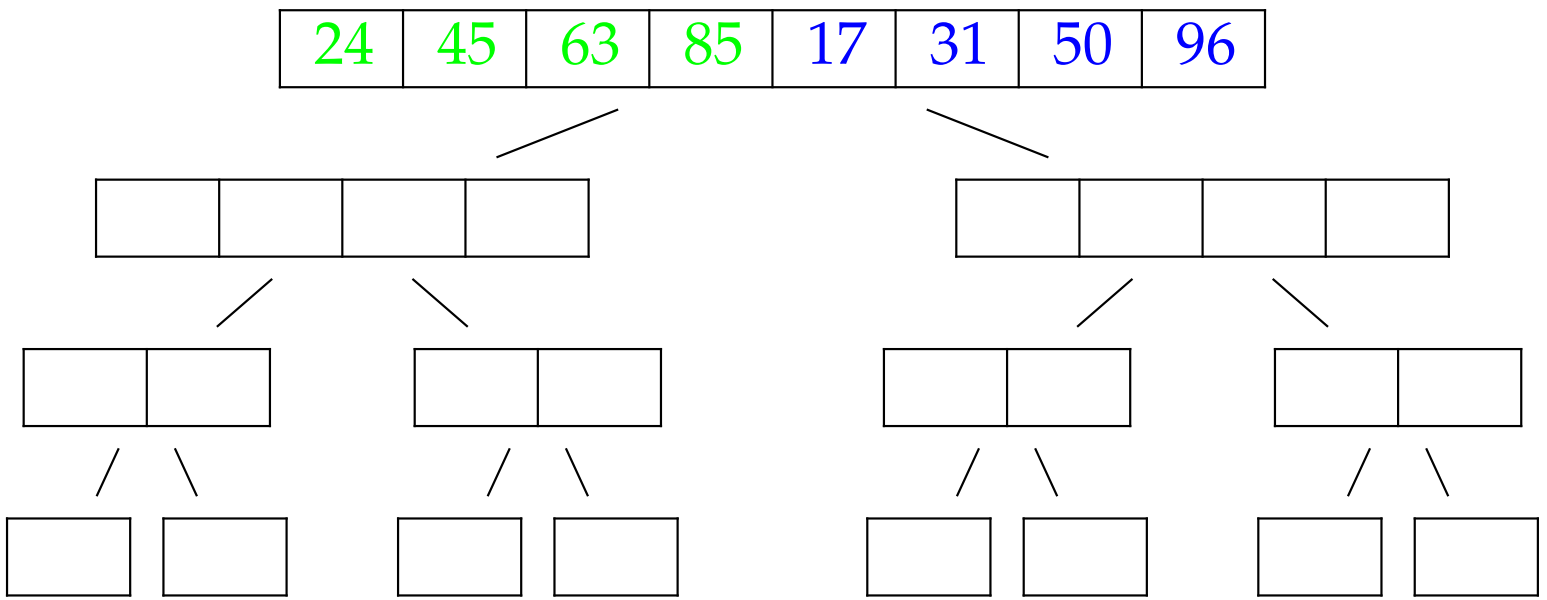
MERGE SORT/9

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



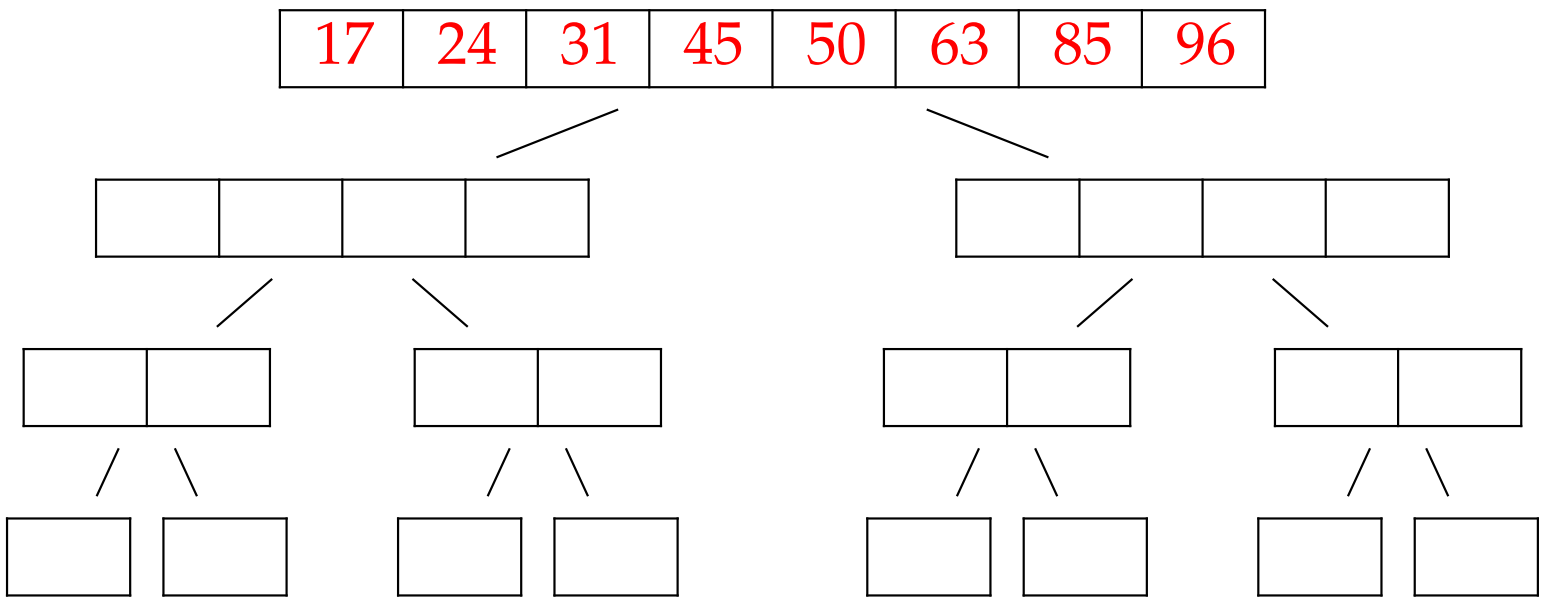
MERGE SORT/9

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]

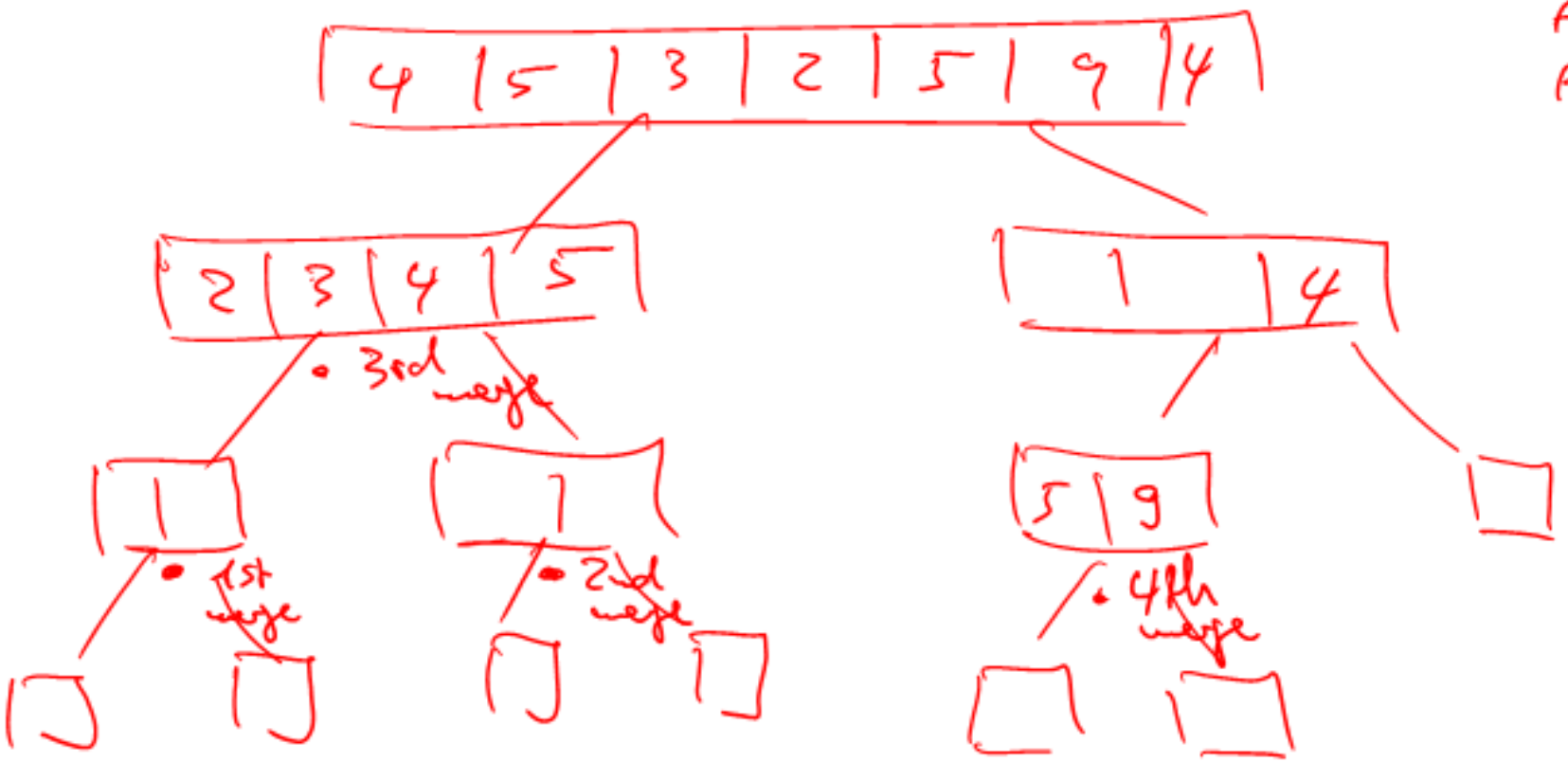


EXERCISE SL03-1

$m = \lfloor (\ell + r)/2 \rfloor$ $A[\ell..m]$, $A[m+1, r]$

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

$\ell = 1$, $r = 3$
$m = 2$
$A[1..2]$
$A[3..3]$



$\Rightarrow A = [2, 3, 4, 5, 5, 9, 4]$

MERGE SORT / 10

Running time

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

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

size of the original problem
size of the subproblem
(1 in our example)
of subproblems
cost to divide
cost to combine

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

Compute $C(n)$ + recursive call

iterate over n elements

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

MERGE SORT/11

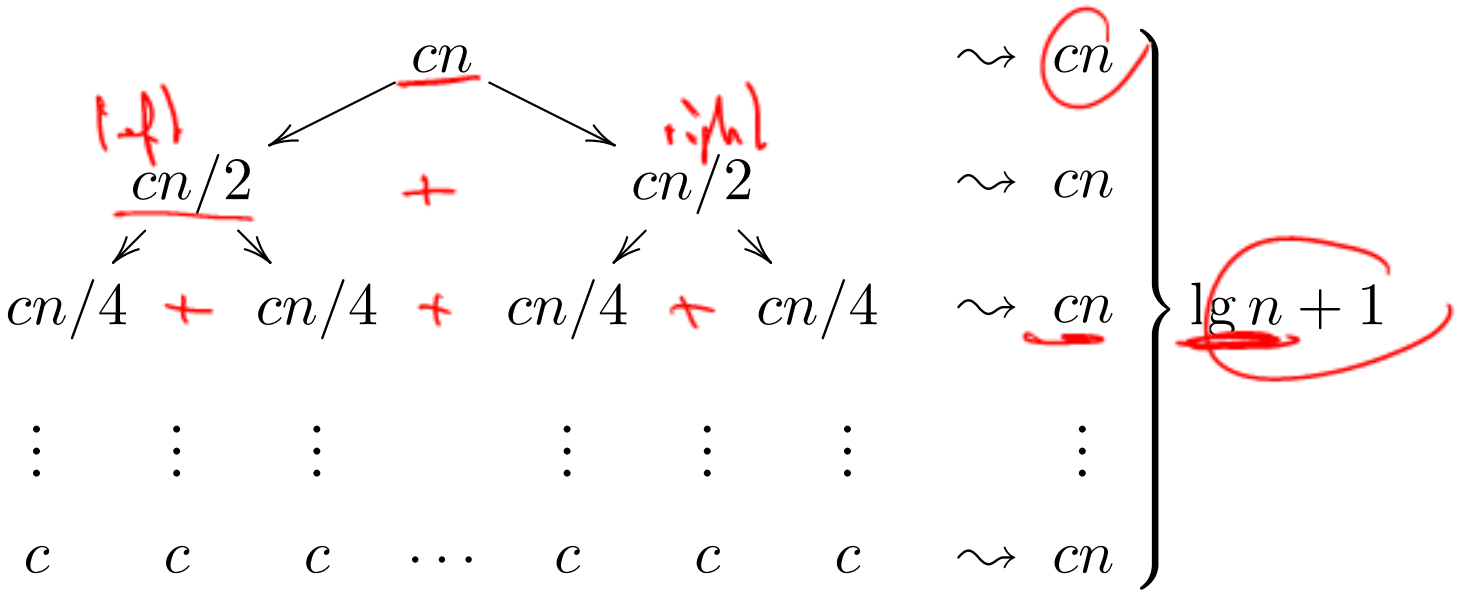
The recurrence/1

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

Merge sort / 11

The recurrence / 1

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



$= cn(\log n + 1)$

$= \Theta(n \log n)$

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

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

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

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

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

The recurrence/2

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

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

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

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

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

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

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

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

EXERCISE SL03-2

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

$$T(n) = 2T(n/2) + \Theta(1)$$

$$= 2(2T(n/4) + \Theta(1)) + \Theta(1)$$

$$= 2^2 T(n/4) + (2+1)\Theta(1)$$

$$= 2^2(2T(n/8) + \Theta(1)) + (2+1)\Theta(1)$$

$$= 2^3 T(n/8) + (2^2 + 2^1 + 2^0)\Theta(1)$$

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

$$= 2^{\log n} T(n/n) + 2^{\lceil \log n \rceil}\Theta(1) - \Theta(1)$$

$$= n + n\Theta(1) - \Theta(1) = \Theta(n)$$

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

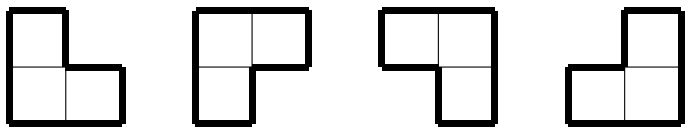
2^i

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

$i_{max} = \log n$

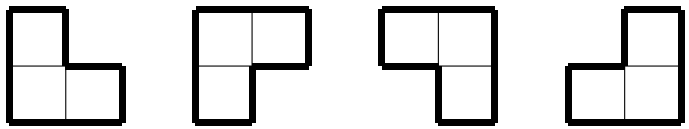
TROMINO TILING/1

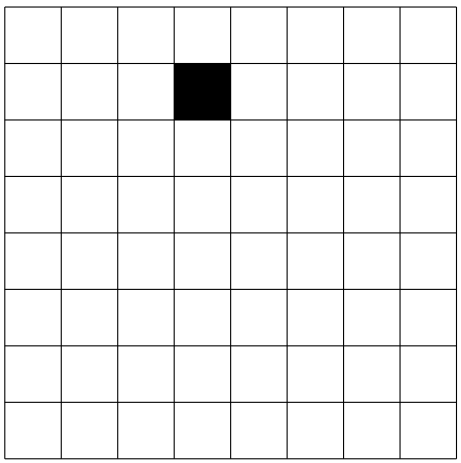
Problem description

- Tromino tiles 

TROMINO TILING/1

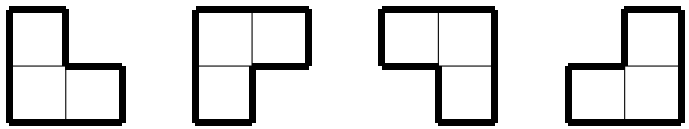
Problem description

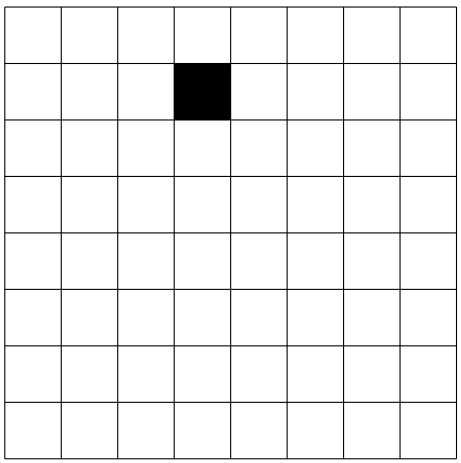
- Tromino tiles 
- Provide a tromino tiling for a $2^n \times 2^n$ board with a hole

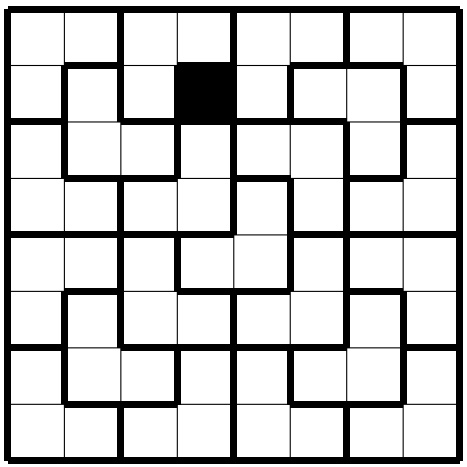


TROMINO TILING/1

Problem description

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





TROMINO TILING/2

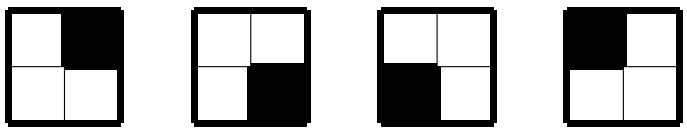
Trivial case: $n = 1$

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

TROMINO TILING/2

Trivial case: $n = 1$

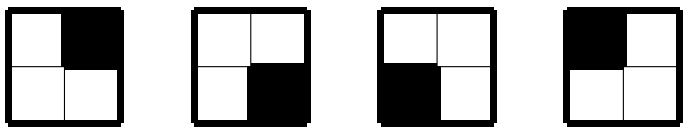
- Tromino tiling for a $2^1 \times 2^1$ board with a hole
- Solution is easy



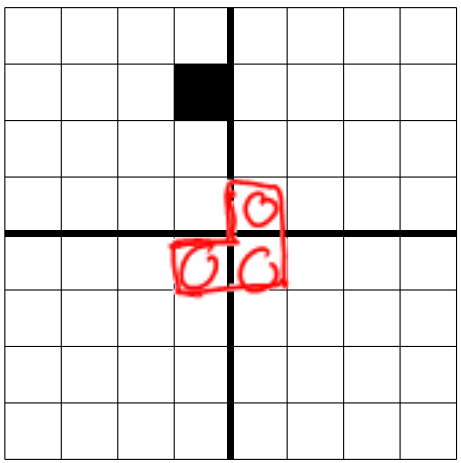
TROMINO TILING/2

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

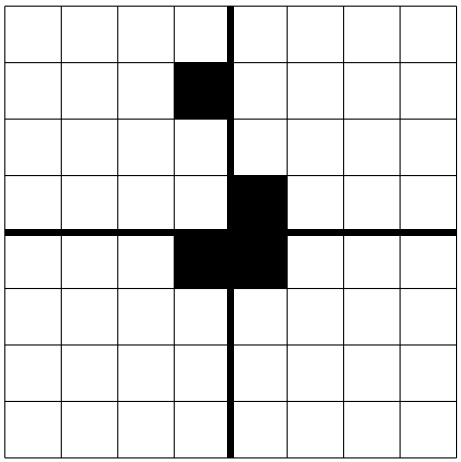
General case: $n > 1$

- 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: $n > 1$

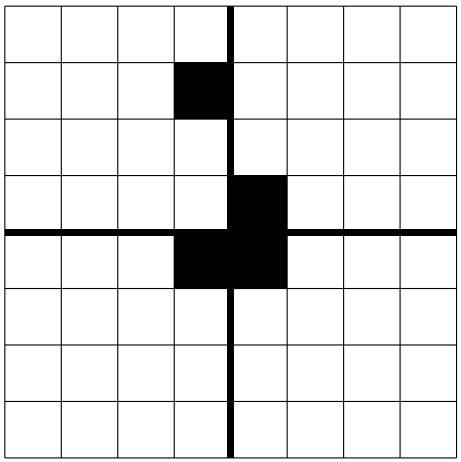
- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes
- Solution: use a tromino to simulate missing holes



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);

(x, y) coordinate

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 2×2

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 2×2
- 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 2×2
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances
- Conquer: tile the four smaller boards recursively

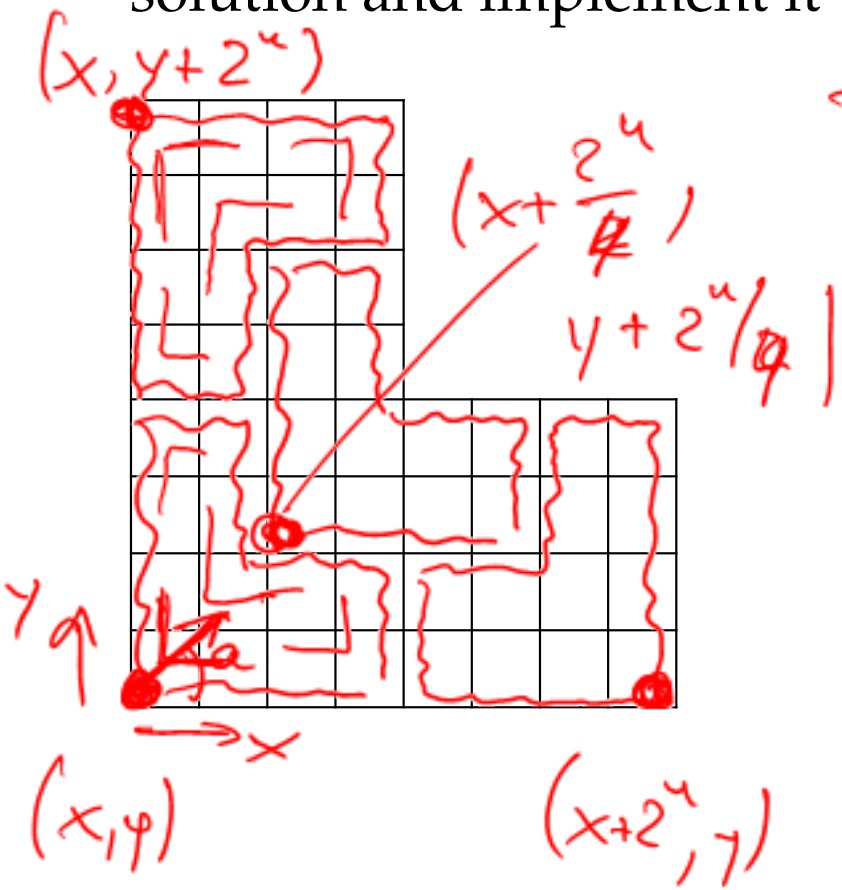
TROMINO TILING/6

Divide-and-conquer

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

EXERCISE SL03-3/1

Use a trominos to tile a right angle. Describe the idea for the solution and implement it with pseudocode.




```
Tile(k, P)
  if (k==1) then done
  Divide L-shape into 4 L-shapes with half length
  Tile(k-1, P1) — (x, y, a)
  Tile(k-1, P2)
  Tile(k-1, P3)
  Tile(k-1, P4)
```

EXERCISE SL03-3/2

TABLE OF CONTENTS — SL03

1. Divide and conquer

2. Recurrences
   - Substitution
   - Recursion tree
   - Master method

analyzing the runtime of recursive algorithms

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
   - 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
4. Master method
   - Templates for different classes of recurrences

RECURRENCES / 3

Merge sort revisited

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

cost to merge two sorted parts (go through all elements once)

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

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

- 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

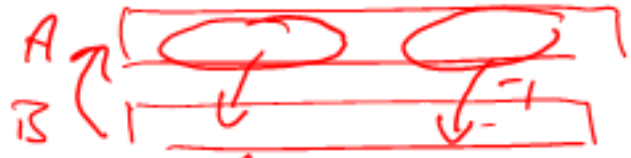
- Is not a strictly formal proof

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

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

copy to auxiliary array and back

3 procedure calls (+ join in middle)

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

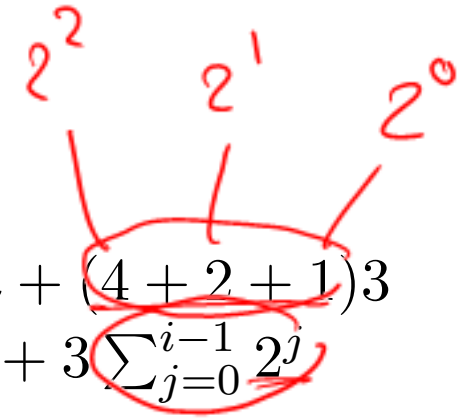
$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

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

Recurrences / 6

- Find pattern:



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

RECURRENCES / 6

- Find pattern:

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

- 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:

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

- Upper bound for i is $\lg n$

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

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

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

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

EXERCISE SL03-4/1

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.

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

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

EXERCISE SL03-4/2

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

$n - id = 1$

$\Rightarrow i = \frac{n-1}{d}$

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

$$= T(n-2d) + n - 2d + n - d$$

$$= T(n-2d) + 2n - d(1+2)$$

$$= T(n-id) + in - d \sum_{j=1}^{i} j \qquad i_{max} = \frac{n-1}{d}$$

$$= \frac{n^2 - dn + d - 1}{2d} = \Theta(n^2)$$

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

EXERCISE SL03-4/3

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

remember:

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

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

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

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

$$= \frac{n-1}{k-1} = \Theta(n)$$

$n/k^i = 1 \;/\; n = k^i \;/\; i = \log_k n$

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

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 $T(n) \leq cn^3$

SUBSTITUTION/2

Proof by induction

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

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

$c = 2$

$= 2n^3 - (n^3 - n)$

SUBSTITUTION / 2

Proof by induction

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

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

$(cn^3/2 - n) \geq 0$

inductive step

$T(1) = 0 \leq c$

$T(2) = 2 \leq c \cdot 8$

$T(4) = 12 \leq c \cdot 27$

OK for $c = 2$

base case

SUBSTITUTION/2

Proof by induction

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

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

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

EXERCISE SL03-5

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.

$T(n) = T(n/k) + n/k, \quad T(1) = c$

hypothesis: $T(n) \leq cn$

$$T(n) = T(n/k) + n/k$$
$$\leq cn/k + n/k$$
$$= n\frac{c+1}{k}$$
$$\leq cn \quad \text{yes} \quad k=2, \; c=2$$

$\frac{3}{2}n \leq 2n$ ✓

SUBSTITUTION / 3

Getting tighter bound / 1

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

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

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

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

SUBSTITUTION / 3

Getting tighter bound / 1

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

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

- 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

$\Theta(n^2) = \{ n^2, 3n^2, n^2 - n, n^2 + n, \ldots \}$

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

hypothesis

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

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

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

SUBSTITUTION / 5

Getting tighter bound / 3

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

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

SUBSTITUTION/5

Getting tighter bound/3

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

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

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

EXERCISE SL03-6/1

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

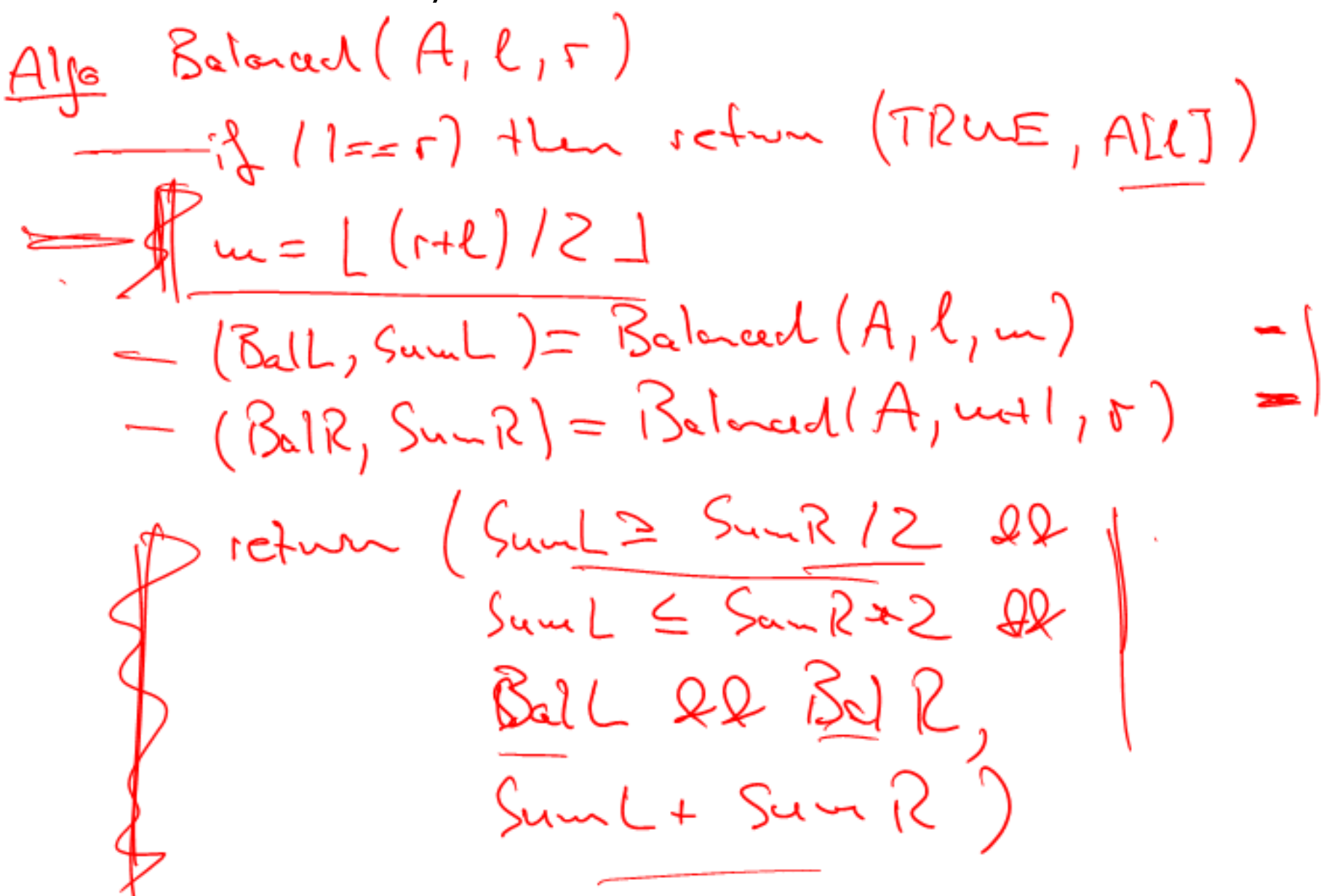
- the sum of the elements in the first half is no more than double and no less than half the sum of the elements in the second half, and
- the first and second half are balanced.

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 SL03-6/2



EXERCISE SL03-6/3

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

$T(2) = 2 \cdot T(1) + 1$

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

$T(n) \leq cn$

$T(n) = 2T(n/2) + 1$

$\leq 2cn/2 + 1$

$= cn + 1$

$\not\leq cn$

$T(n) \leq c_1 n - c_2$

$T(n) = 2T(n/2) + 1$

$\leq 2(c_1 n/2 - c_2) + 1$

$= c_1 n - 2c_2 + 1$

$= c_1 n - c_2 - (c_2 - 1)$

$\leq c_1 n - c_2$ for $c_2 > 1$

$T(1) = 0$	$\leq c_1 - c_2$	
$T(2) = 1$	$\leq 2c_2 - c_2$	OK for $c_1 = 2, c_2 = 1$
$T(4) = 3$	$\leq 4c_1 - c_2$	

$n \geq n_0$

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

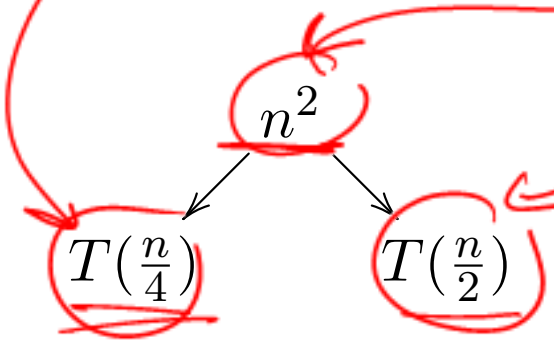
Basic methodology

- 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

Basic methodology

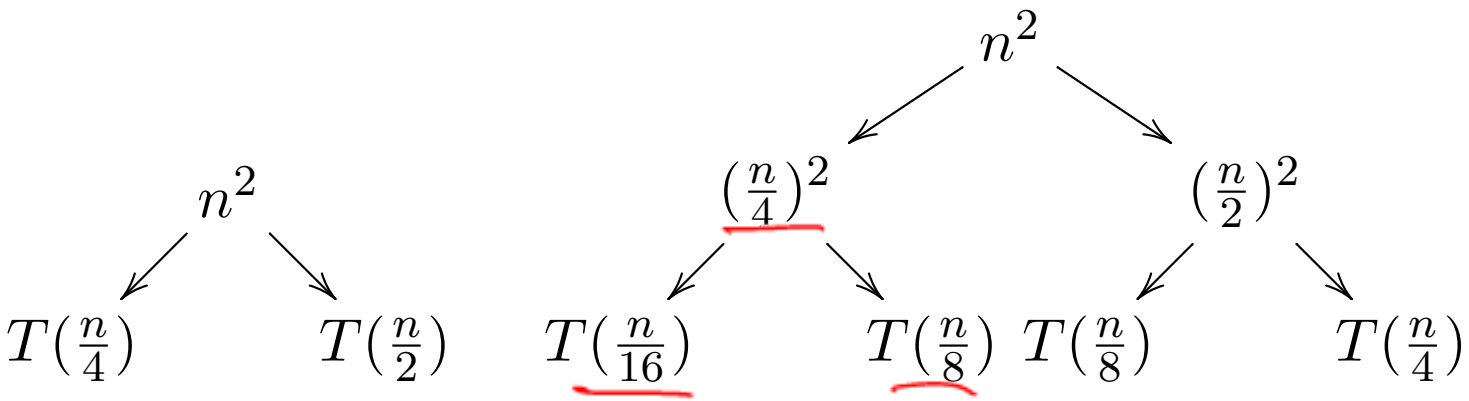
- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$



Recursion tree/1

Basic methodology

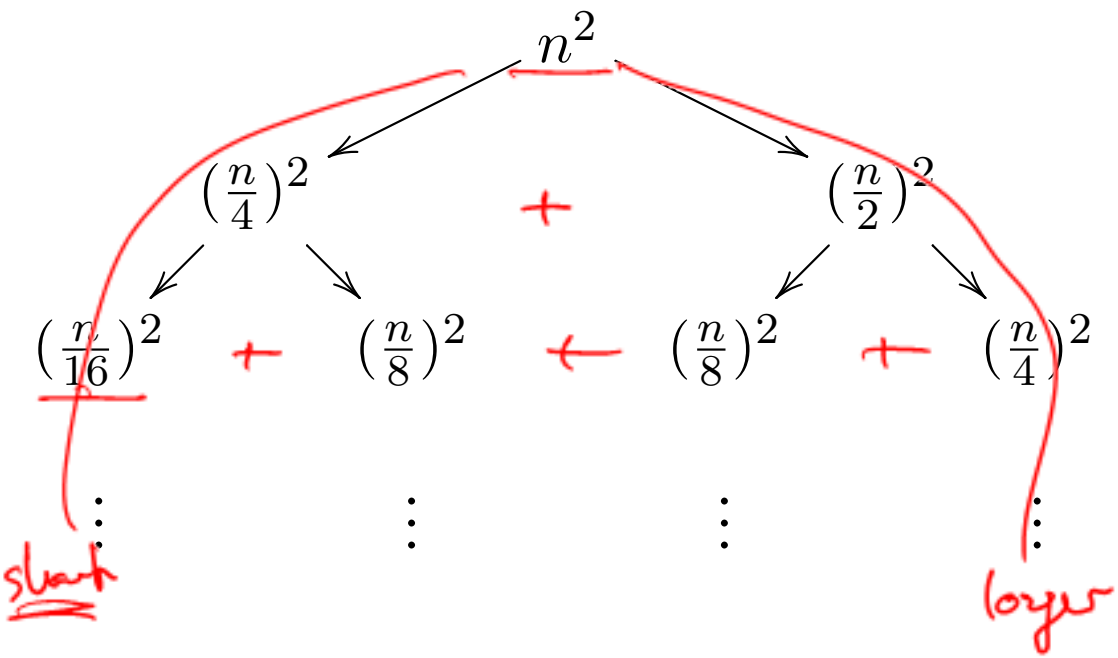
- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$



Recursion tree/2

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \text{if } |a| < 1$$

Example recurrence/1



$\rightsquigarrow n^2$ $i=0$

$\rightsquigarrow \frac{5}{16}n^2$ $i=1$

$\rightsquigarrow \frac{25}{256}n^2 = (\frac{5}{16})^2 n^2$ $i=2$

$\rightsquigarrow (\frac{5}{16})^i n^2$ i

$\rightsquigarrow \Theta(n^2)$

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

Recursion tree/3

Example recurrence/2

guess (follows from recursion tree)

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

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

$11cn^2/16 \geq 0$

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

Example recurrence/2

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

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

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

MASTER METHOD/1

- To solve a class of recurrences that have the form

$$T(n) = aT(n/b) + f(n)$$

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

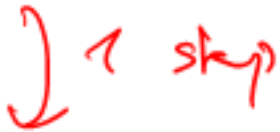
- Iterating the recurrence (expanding the tree) yields

$$T(n) = f(n) + aT(n/b)$$

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

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1)
\end{aligned}
$$

non-leaves

leaves

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\
&= \underbrace{\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)}_{\text{non-leaves}} + \underbrace{\Theta(n^{\log_b a})}_{\text{leaves}}
\end{aligned}
$$

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a}) \end{aligned}$$

- The first term is a division/combination cost (totaled across all levels of the tree)

MASTER METHOD/2

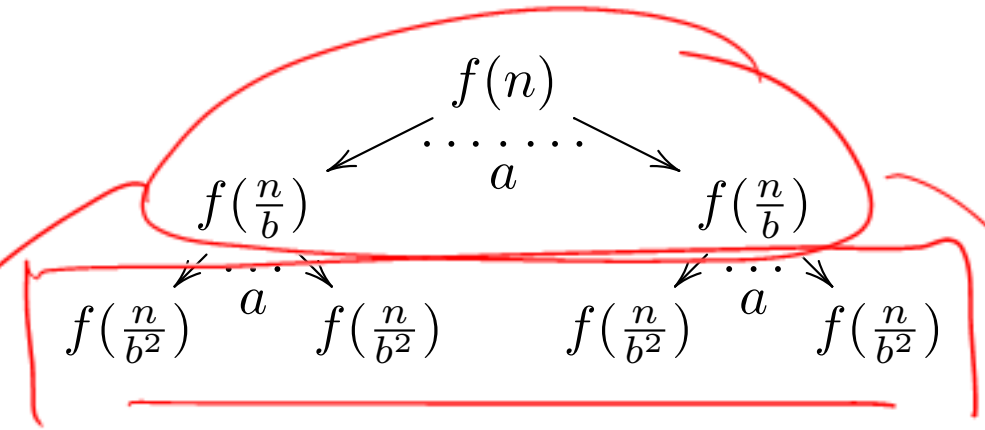
- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a}) \end{aligned}$$

- 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



$\rightsquigarrow f(n)$

$\rightsquigarrow a f(\frac{n}{b})$

$\rightsquigarrow a^2 f(\frac{n}{b^2})$

$\rightsquigarrow a^i f(\frac{n}{b^i})$

- Note:
  - $\log_b n$ levels,
  - $a^{\log_b n} = n^{\log_b a}$ leaves
- Running time: $T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$

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

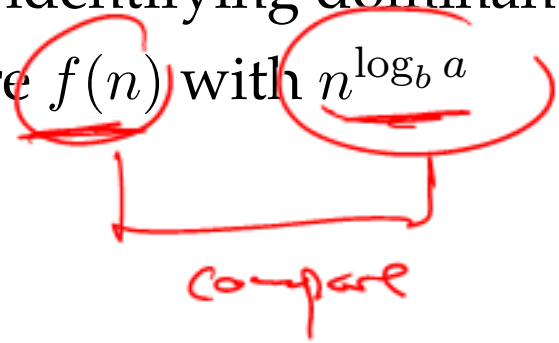
Underlying intuitions

- 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 $n^{\log_b a}$



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 $n^{\log_b a}$

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: $T(n) = 2T(n/2) + \Theta(n)$ (merge sort)

- $a = 2, b = 2, f(n) = \Theta(n)$
- $n^{\log_2 2} = n$
- $\Theta(n) = \Theta(n)$
- 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^ϵ)

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^ϵ)
- The work at the leaves dominates: $T(n) = \Theta(n^{\log_b a})$

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^ϵ)
- The work at the leaves dominates: $T(n) = \Theta(n^{\log_b a})$

Example: $T(n) = 9T(n/3) + n$

- $a = 9, b = 3, f(n) = n$
- $n^{\log_3 9} = n^2$
- $n = O(n^{\log_3 9 - 1}) = O(n^1) = O(n)$
- 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^ϵ)
  - 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^ϵ)
  - Regularity: $\exists c < 1 . \exists n_0 > 0 . \forall n > n_0 . af(n/b) \leq cf(n)$
- The work at the root dominates: $T(n) = \Theta(f(n))$

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^ϵ)
  - 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: $T(n) = 3T(n/4) + n \lg n$

- $a = 3$, $b = 4$, $f(n) = n \lg n$
- $n^{\log_4 3} = n^{0.792}$
- $n \lg n = \Omega(n^{\log_4 3+0.208})$
- case 3: $T(n) = \Theta(n \lg n)$
- Need also to verify regularity condition: $af(n/b) \leq cf(n)$

$$c = 3/4 : af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n$$

$\frac{3}{4} n \log \frac{n}{4} \leq \frac{3}{4} n \log n$

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

EXERCISE SL03-7/1

Use the Master Method to solve the recurrence

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

- $a = 2$, $b = 2$, $f(n) = n^3$
- $n^{\log_b a} = n^{\log_2 2} = n$
- $n^3 = \Omega(n^{1+\varepsilon})$ for $\varepsilon = 2$
- $2(\frac{n}{2})^3 \leq cn^3$ holds for $c = 1/2$, $c = 1/4$
- case 3: $T(n) = \Theta(n^3)$

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)

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

fastest known sorting algo; used in todays computers

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];
    swap smallest element with A[i];
```

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

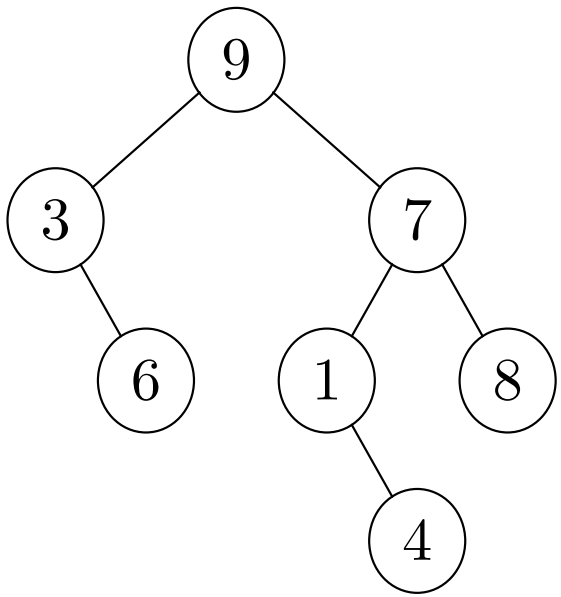
```
1 for i = 1 to n do
2     find smallest element in A[i ... n];
3     swap smallest element with A[i];
```

- 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

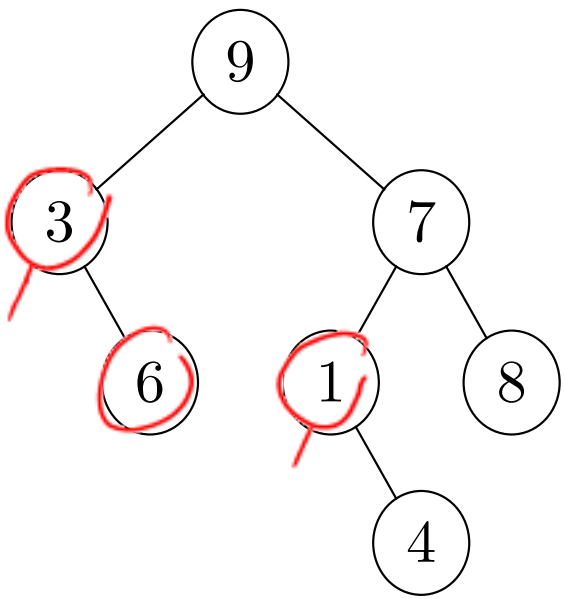
each node has two children

Binary trees/1



TREES AND HEAPS/1

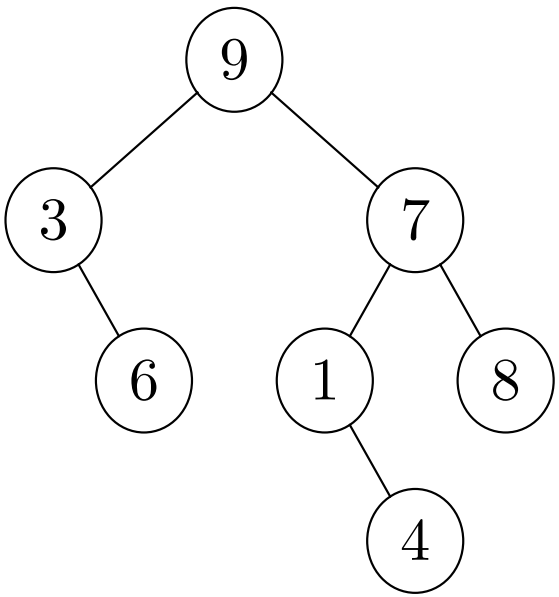
Binary trees/1



- 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

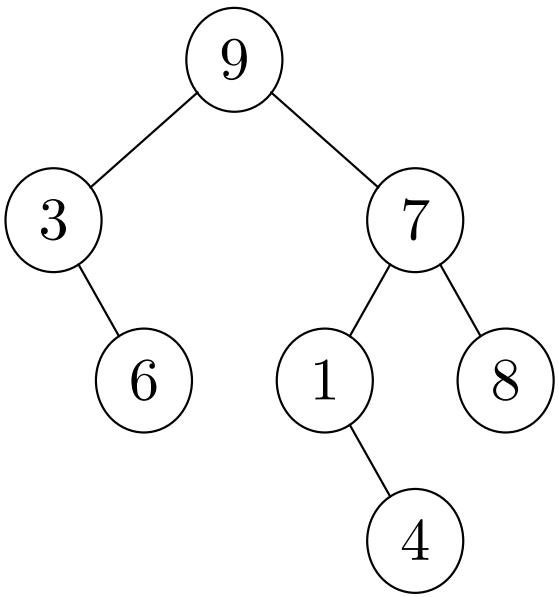
Binary trees/1



- 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

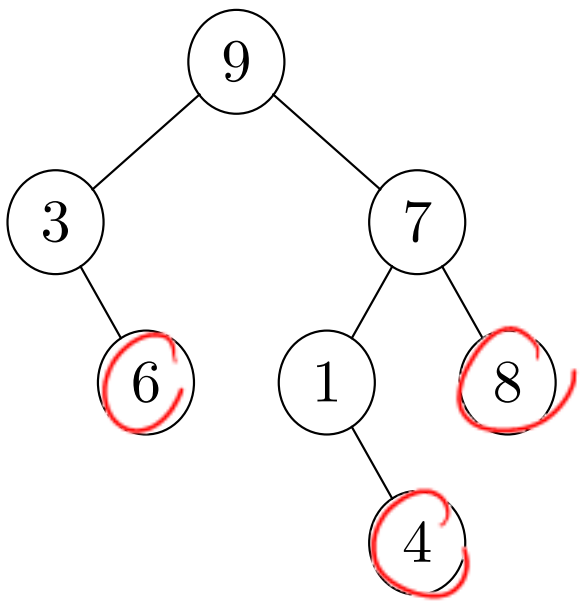
Binary trees/1



- 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

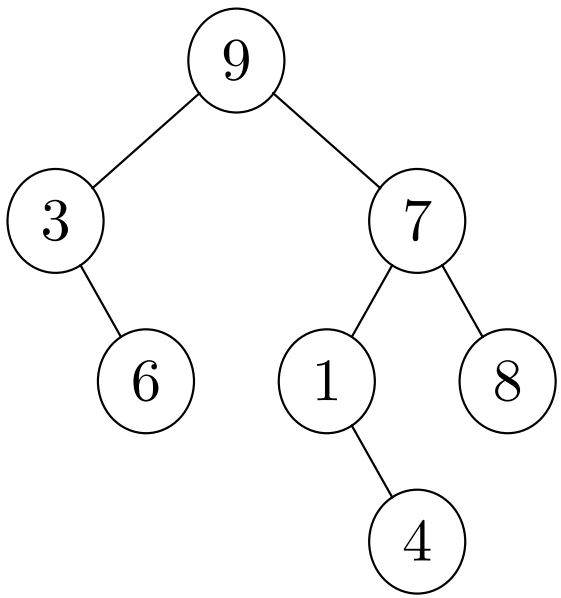
Binary trees/1



- 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**

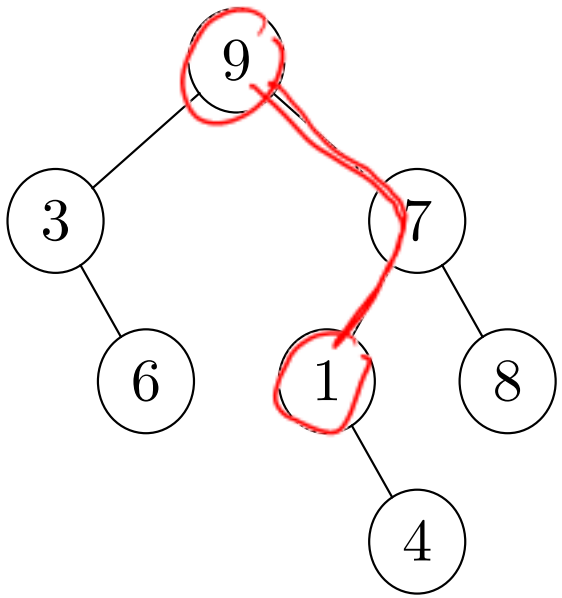
TREES AND HEAPS/2

Binary trees/2



TREES AND HEAPS/2

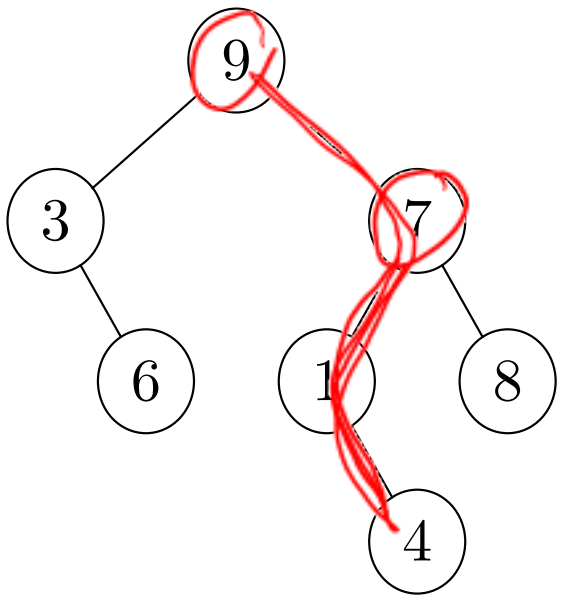
Binary trees/2



- 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

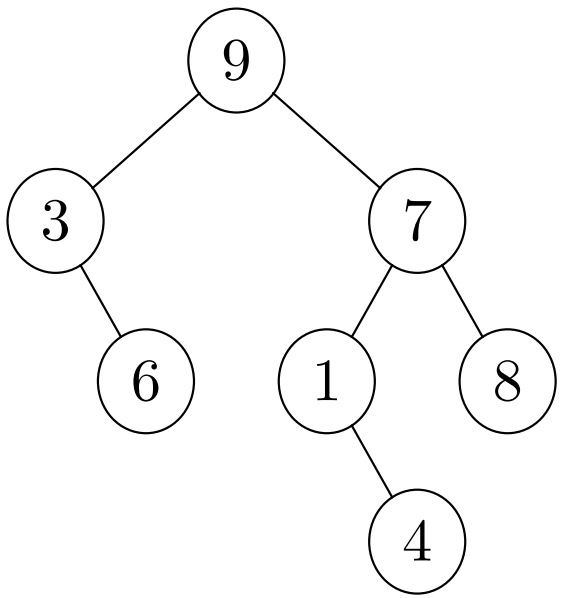
Binary trees/2



- 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

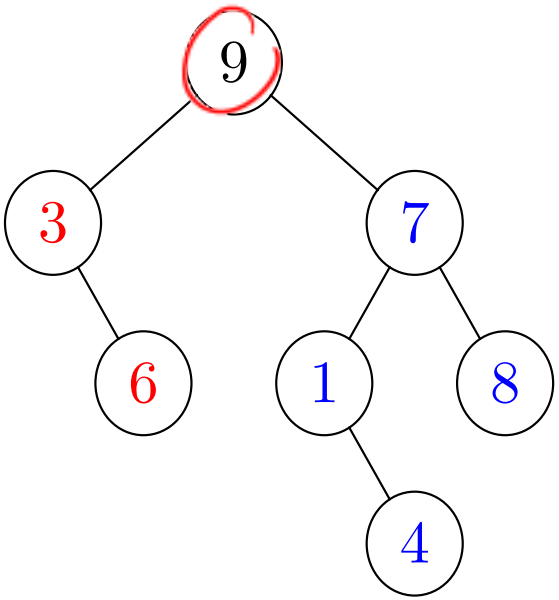


- The depth (level) of a node x is the length of the path from the root to x
  - The **depth** (**level**) of 1 is 2
  - The depth of 9 is 0
- The height of a node x is the length of the longest path from x to a leaf
  - The **height** of 7 is 2
  - The height of 9 is 3
- The height of a tree is the height of its root

TREES AND HEAPS/3

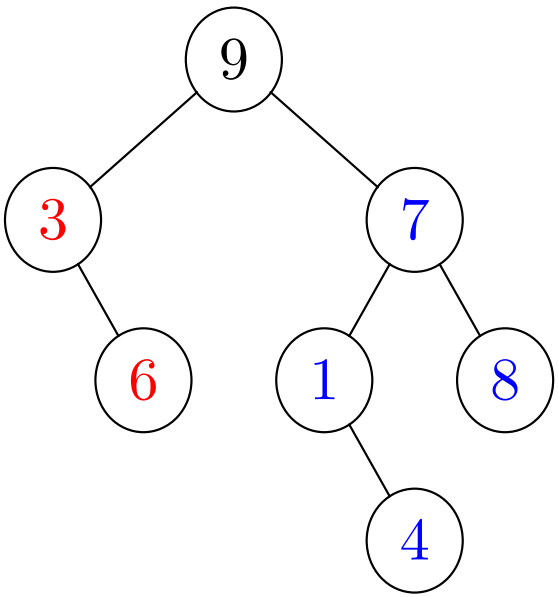
Binary trees/3

no ordering of nodes
(we consider different orderings and exploit these)



TREES AND HEAPS/3

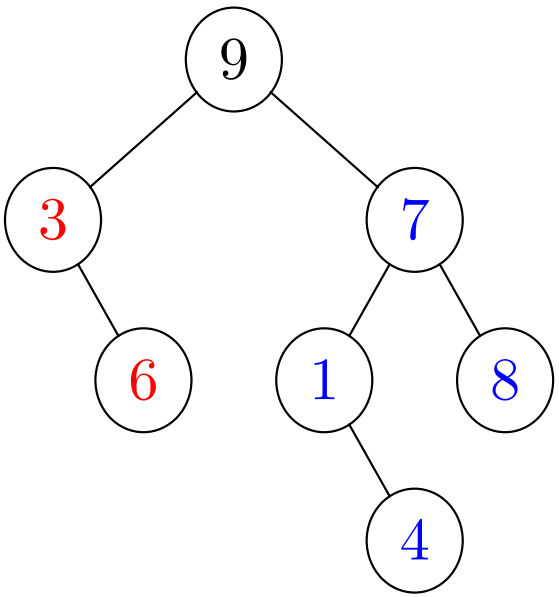
Binary trees/3



- 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

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

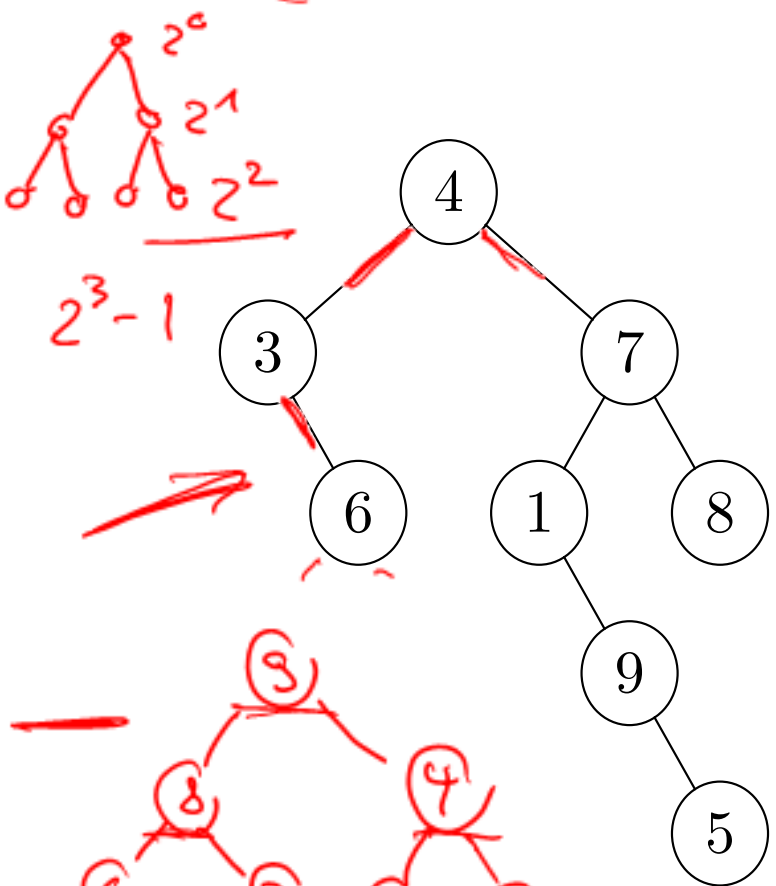
- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children
- A **nearly complete binary tree** is a binary tree where
  - all levels of non-maximal depth d are full (have 2^d nodes)
  - all the leaves with maximal depth are as far 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

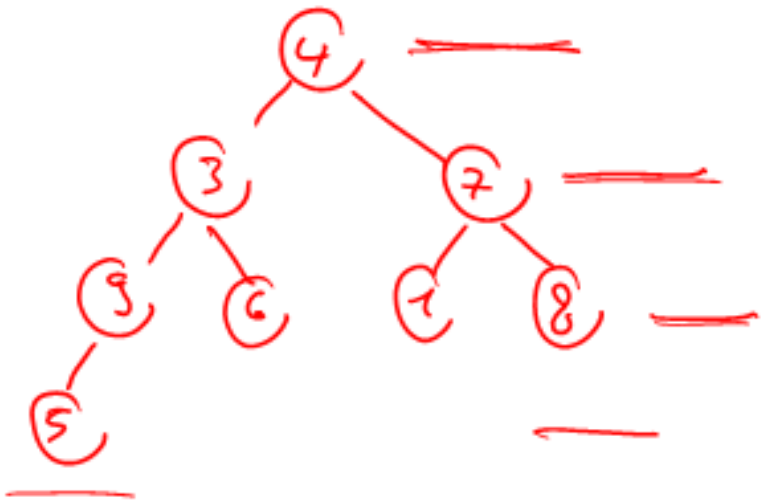
EXERCISE SL04-1

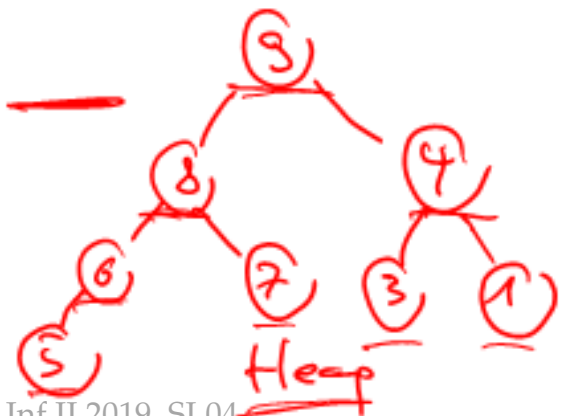
Construct a complete binary tree, a nearly complete binary tree, and a heap with the same nodes as T.



CBT: not possible (number of nodes makes this impossible, $2^x - 1$ nodes are required)

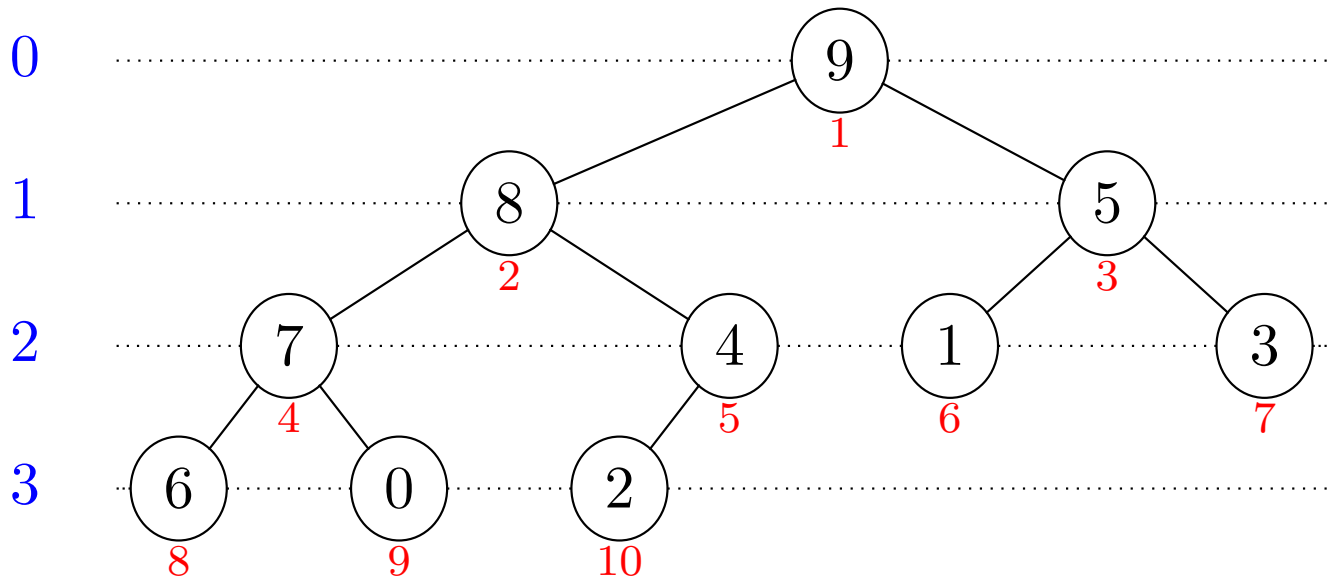
NCBT:





Trees and heaps/5

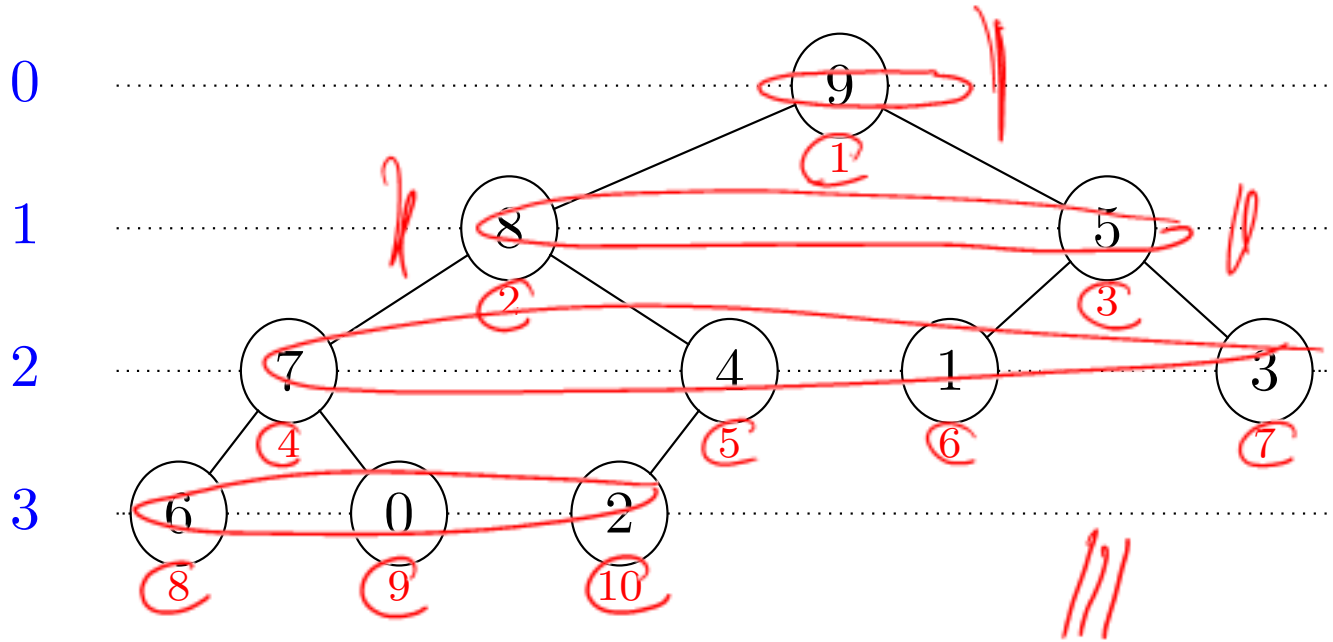
Heaps and arrays are closely related:



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

use array to represent a heap

Trees and heaps/6

Fundamental properties

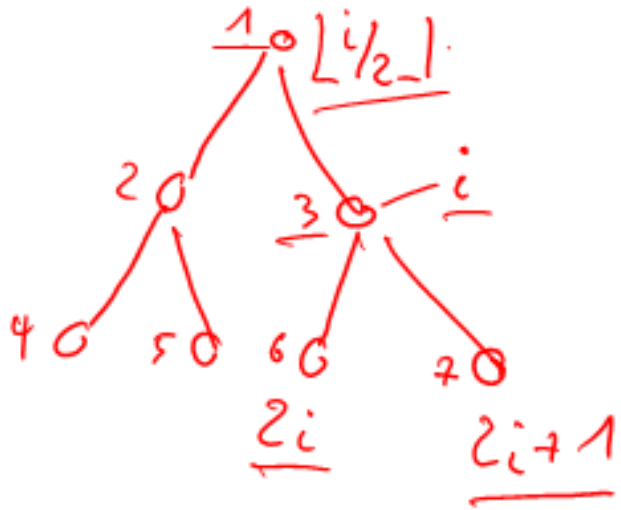
- Let i be a node index.
- Heap property: $A[parent(i)] \geq A[i]$

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?

- minimum number of elements if only 1 element at bottom level: $2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1 = \left(\sum_{i=0}^{h-1} 2^i\right) + 1 = \underline{2^h}$

- max number of elements if bottom level is full: $2^0 + 2^1 + 2^2 + \ldots + 2^h = \underline{2^{h+1} - 1}$

-



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

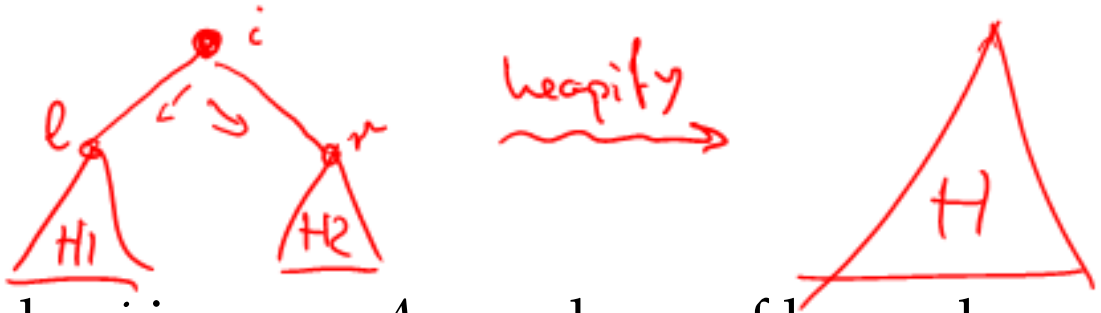
- Input: index i in array A, number s of heap elements
- Binary trees rooted at `left`(i) and `right`(i) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

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

The algorithm

size of the heap (= # of elements)

Algo: Heapify(A,i,s)

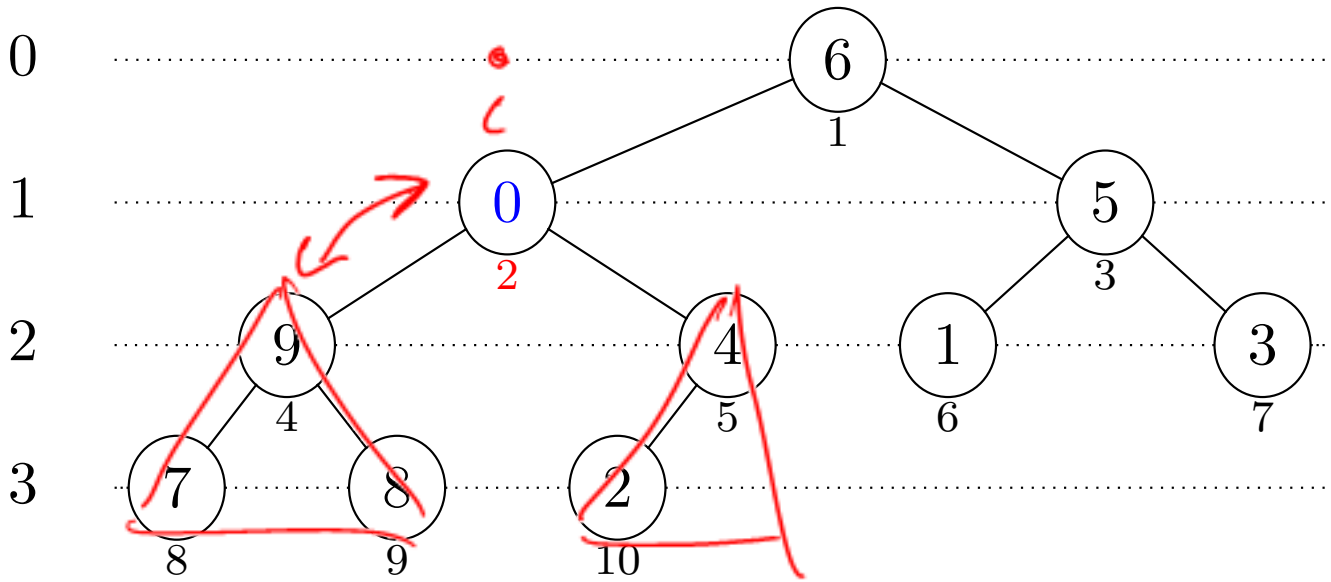
```
m = i;
l = Left(i);
r = Right(i);
if l≤s ∧ A[l]>A[m] then m = l;
if r≤s ∧ A[r]>A[m] then m = r;
if i≠m then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

left element exists

left element is larger than parent

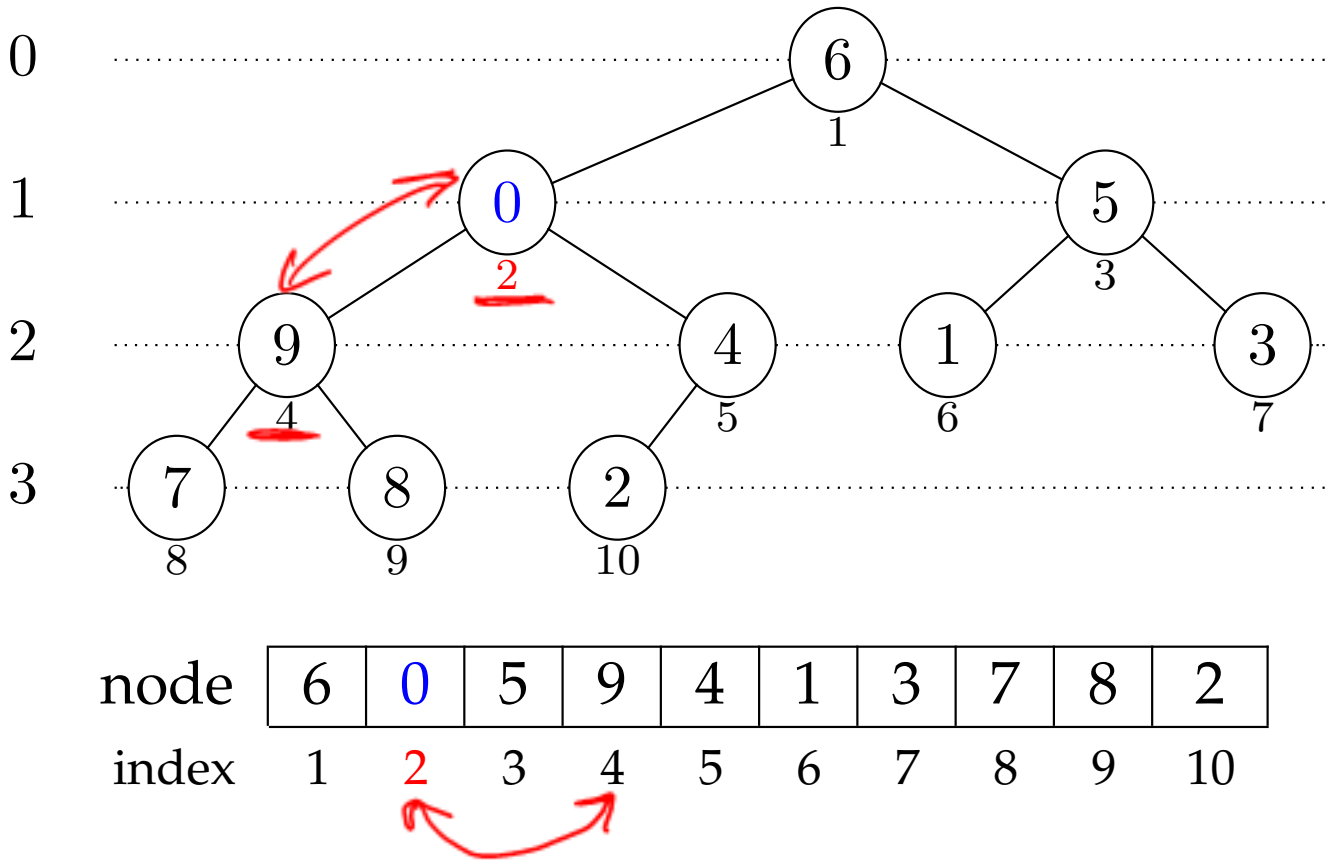
HEAPIFY/3

An example/1



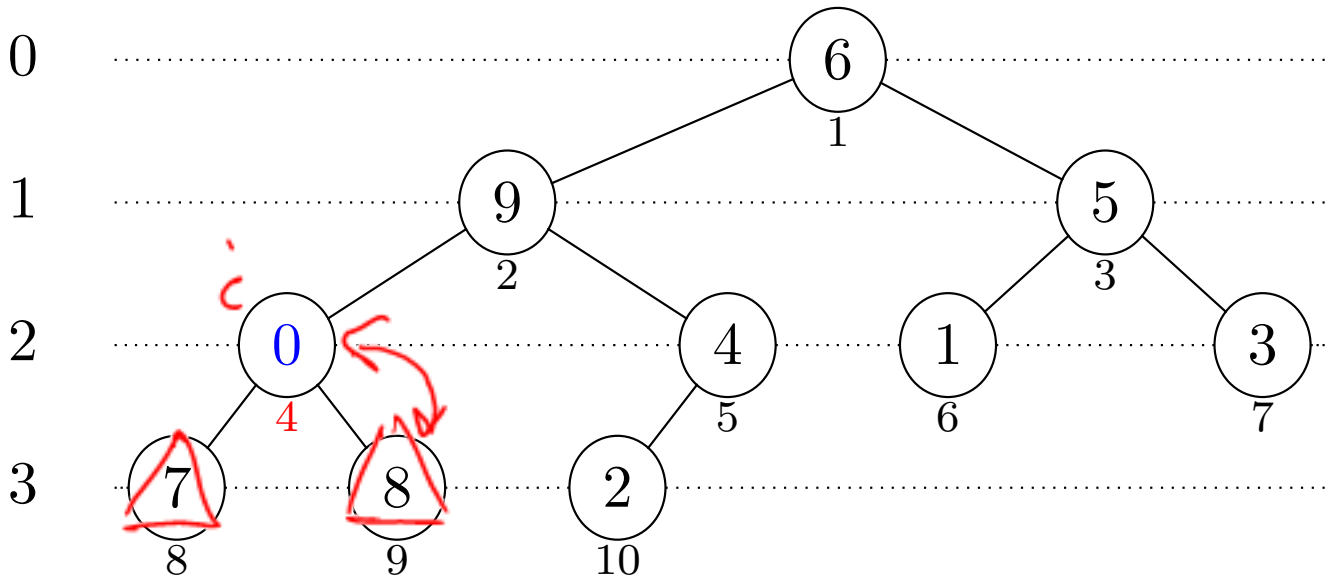
HEAPIFY/3

An example/1



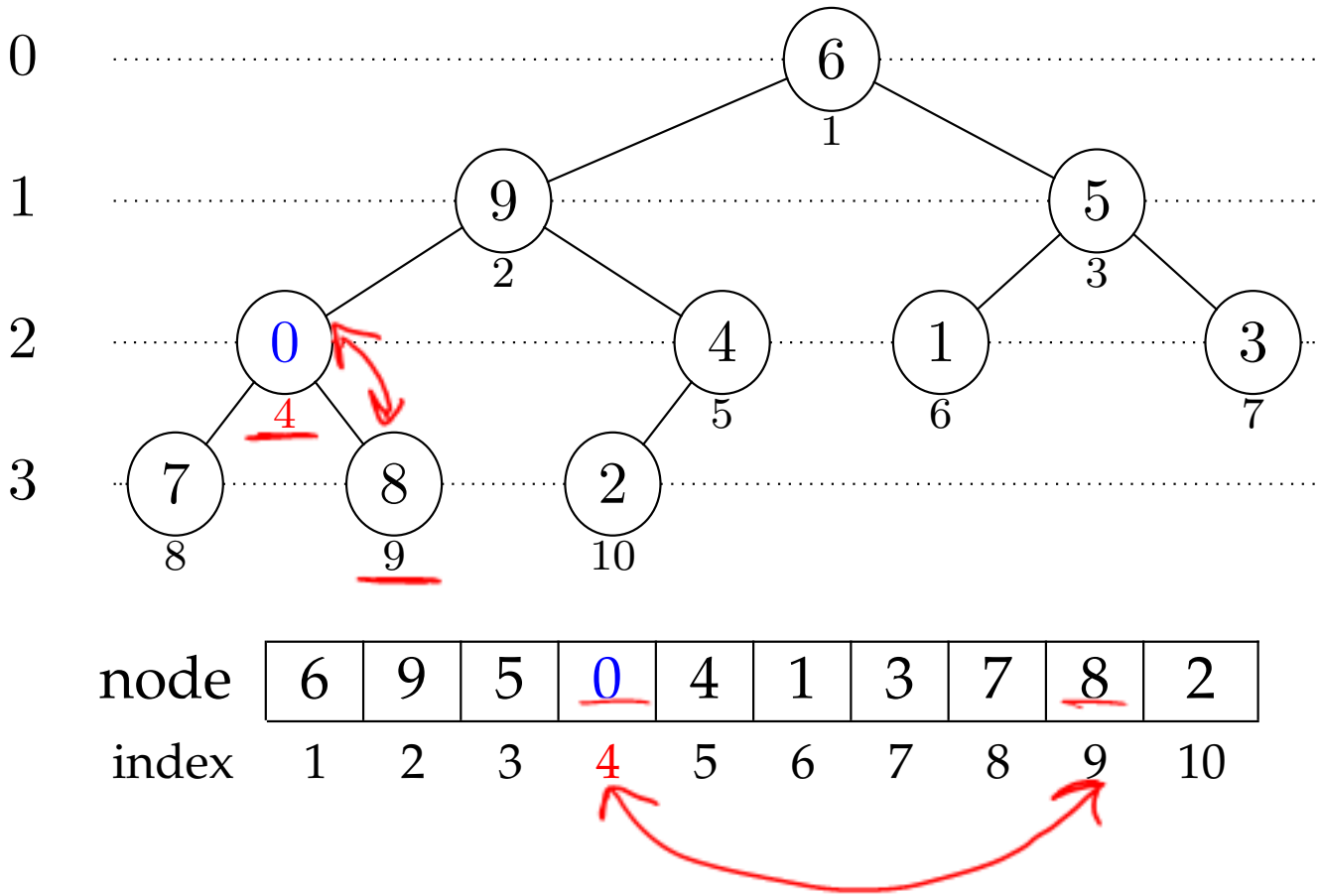
HEAPIFY/4

An example/2



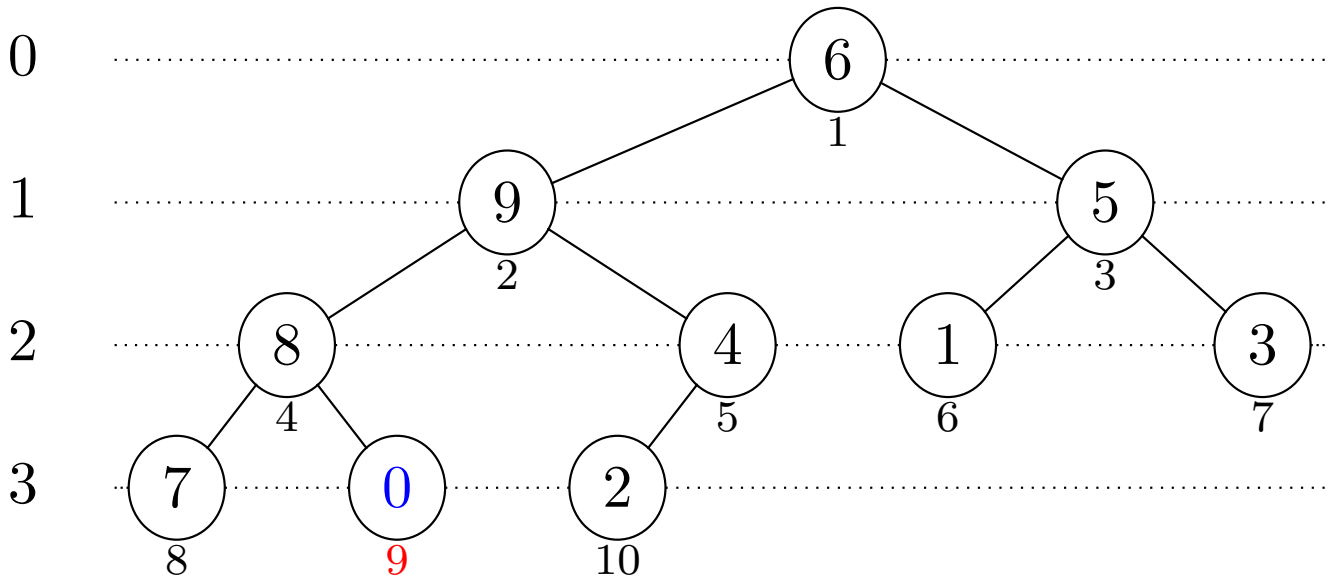
HEAPIFY/4

An example/2



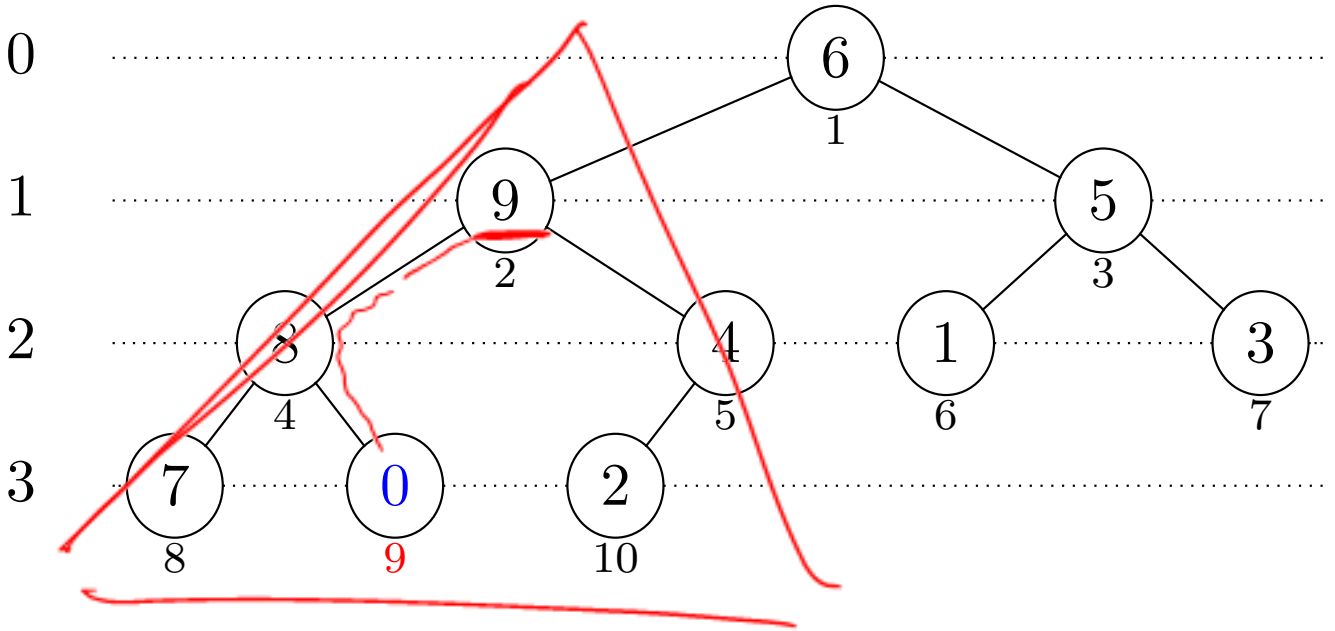
HEAPIFY/5

An example/3



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

- $a = 1$, $b = 2/3$, $f(n) = \Theta(1)$
- $n^{\log_{3/2} 1} = 1$
- $\Theta(1) = \Theta(1)$ case 2
- $\Theta(n^{\log_{3/2} 1} \cdot \log n) = \Theta(\log n)$

Running time

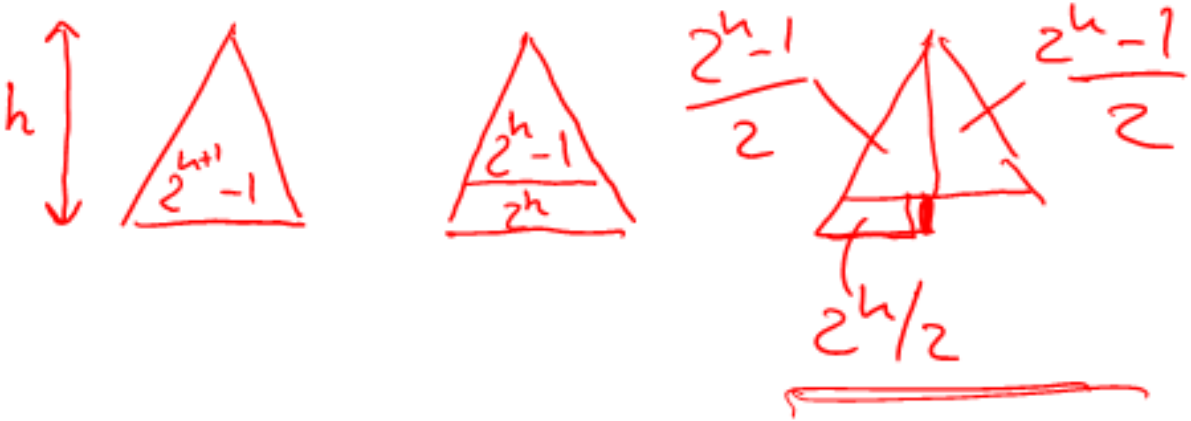
- The running time of heapify on a subtree of size n rooted at i includes time to
  - determine relationship between elements: $\Theta(1)$
  - run heapify on a subtree rooted at one of i's children
- $2n/3$ is the worst case size of the subtree (half filled bottom level) and thus

$$T(n) \leq T(2n/3) + \Theta(1), \text{ i.e. } T(n) = O(\lg n)$$

EXERCISE SL04-3

Explain the derivation of the worst case running time of Heapify (size of tree as well as height of tree).

worst case runtime if tree is as unbalanced as possible.



$$2^h - 1 + 2^h/2 = n$$

$$2 \cdot 2^h - 2 + 2^h = 2n$$

$$2^h = 2/3\, n + 2/3$$

$$h = \log(2/3\, n + 2/3)$$

$$\approx \log(2/3\, n)$$

$$= \log(2/3) + \log(n)$$

$$\approx \log(n)$$

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

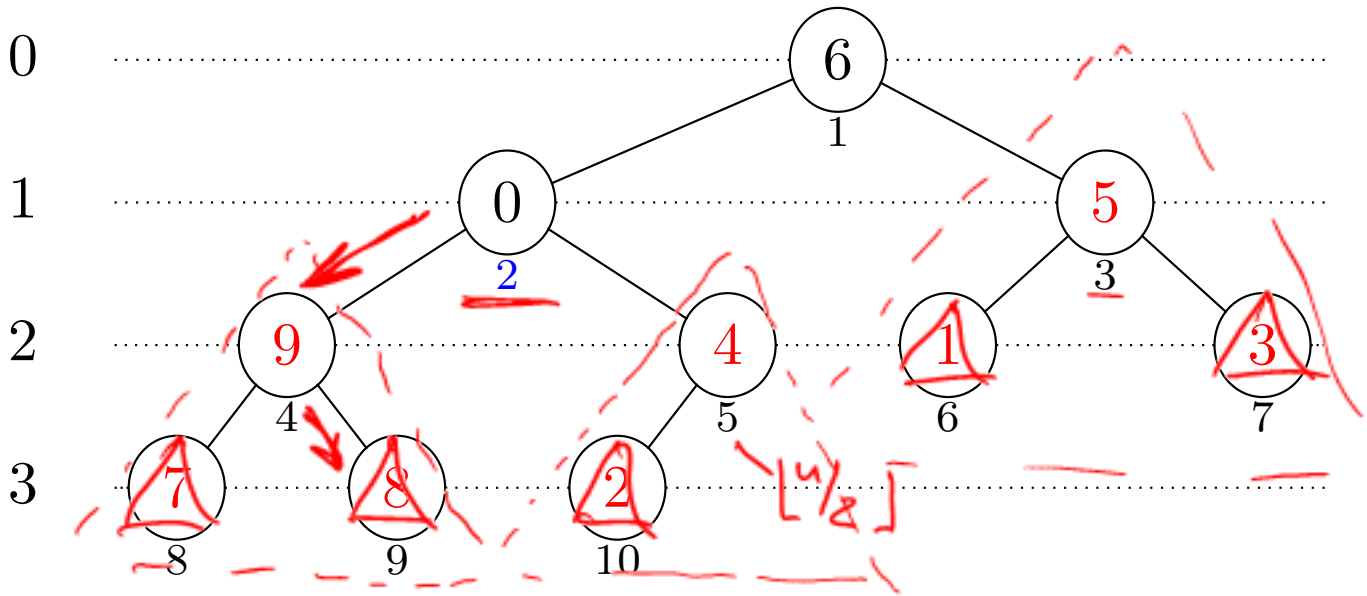
Algo: BuildHeap(A,n)

---

for $i = \lfloor n/2 \rfloor$ **to** *1* **do** Heapify(A,i,n);

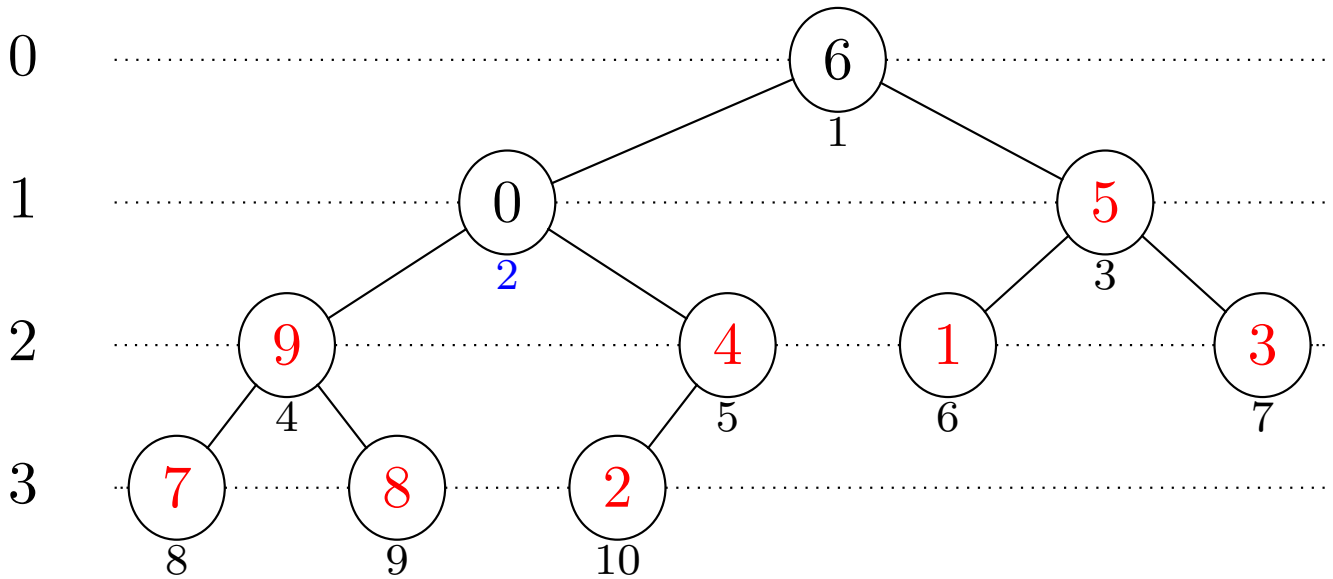
Building a heap/8

An example/1



BUILDING A HEAP/8

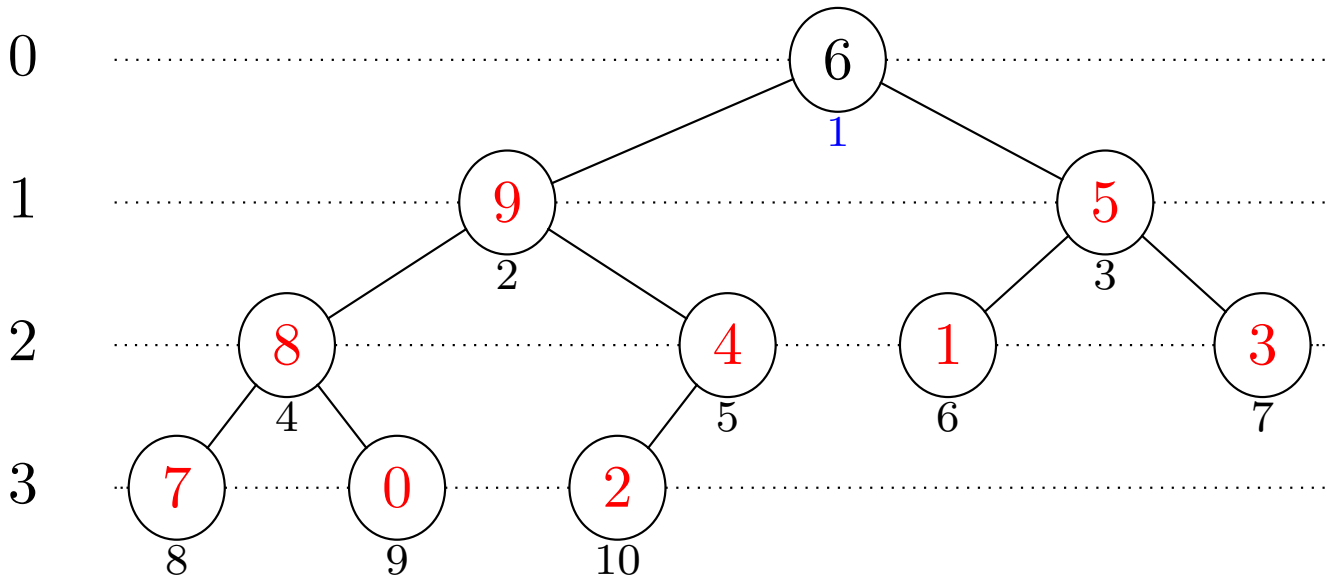
An example/1



node	6	0	5	9	4	1	3	7	8	2
index	1	2	3	4	5	6	7	8	9	10

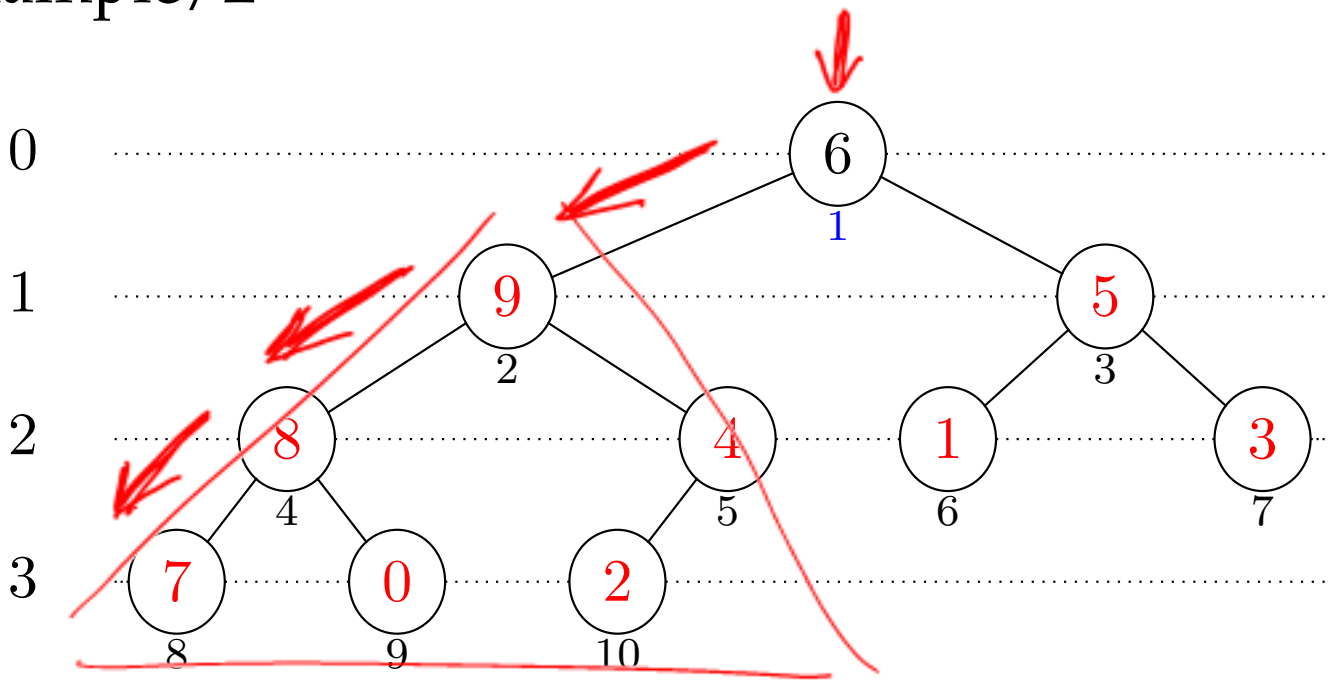
Building a heap/3

An example/2



Building a heap/3

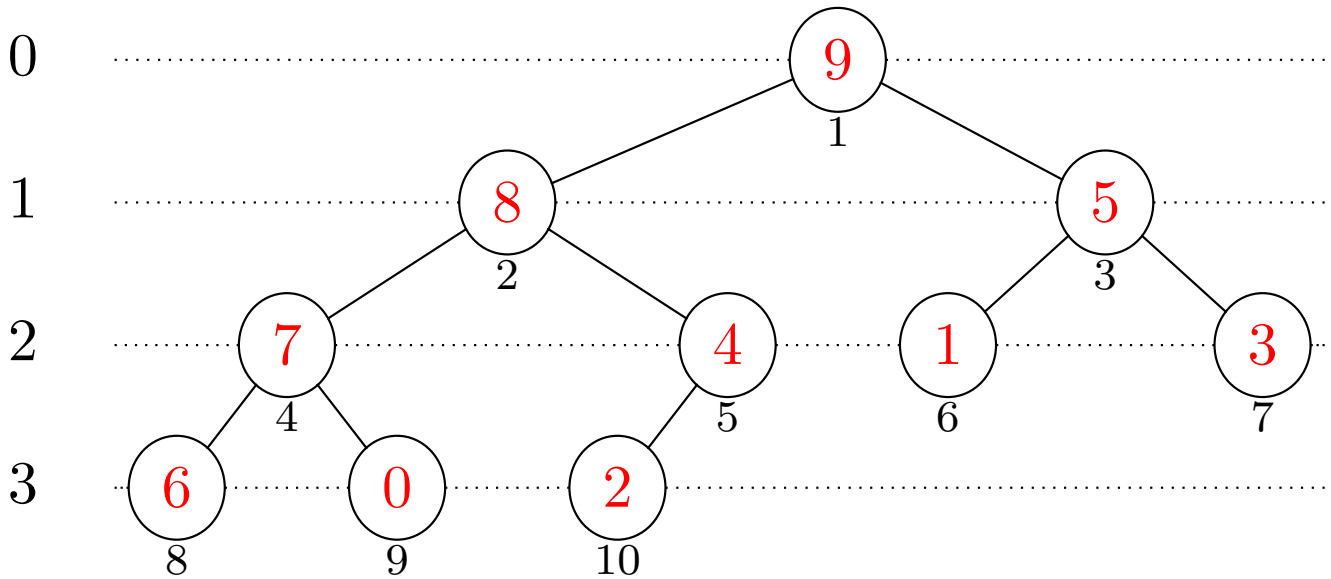
An example/2



node	6	9	5	8	4	1	3	7	0	2
index	1	2	3	4	5	6	7	8	9	10

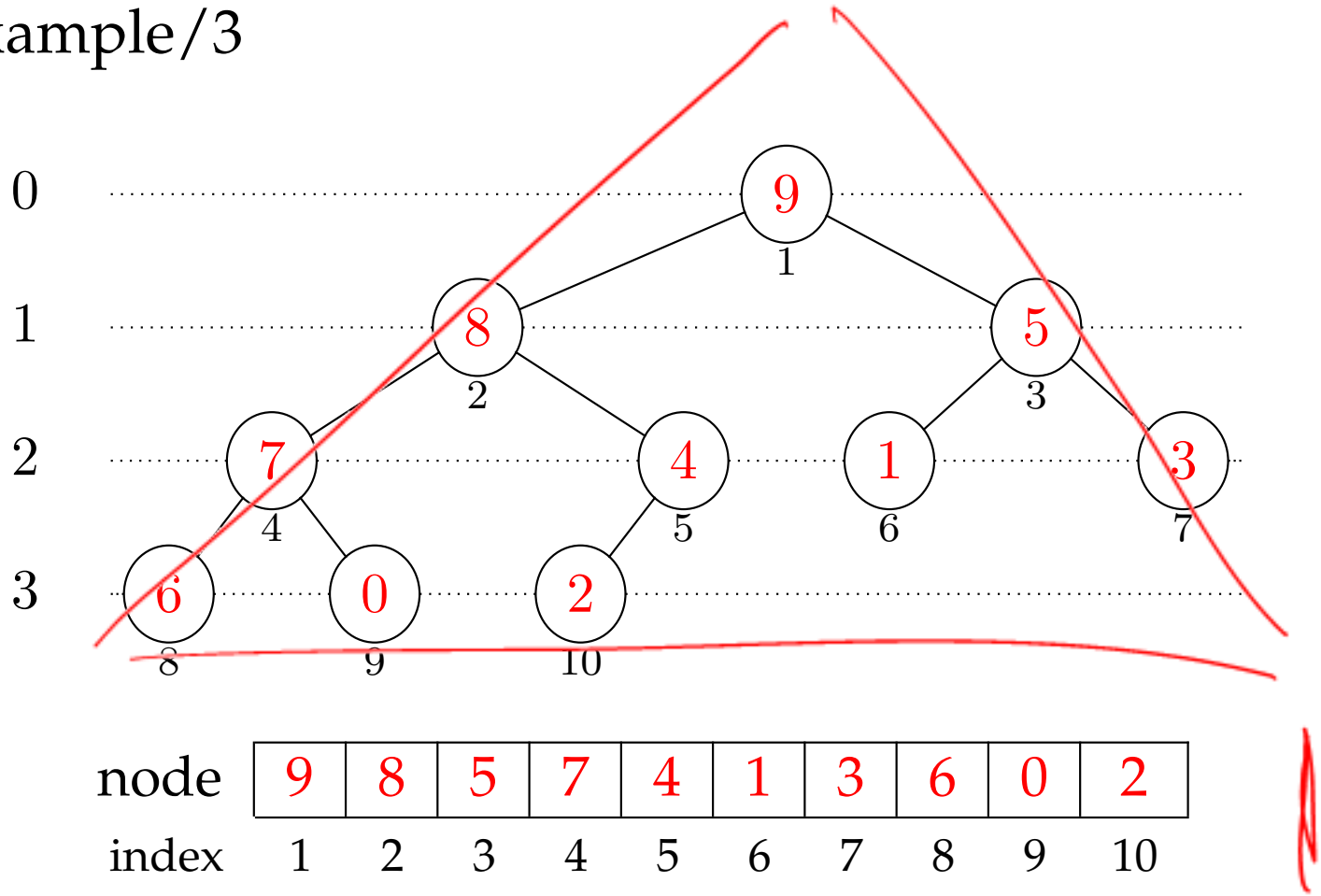
Building a heap/4

An example/3



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

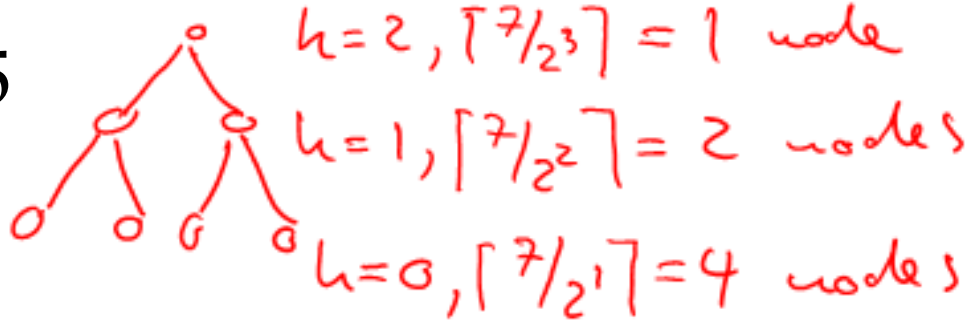
Correctness

- 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



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 $O(h)$

BUILDING A HEAP/6

Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$ for $x > 1$
- 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:

$$T(n) = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h)$$
$$= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right)$$

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:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\ &= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \\ &\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right) \end{aligned}$$

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:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\ &= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \\ &\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} h(\tfrac{1}{2})^h\right) \\ &= O(n * 2) = O(n) \end{aligned}$$

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);
```

position where the largest element is moved

last element

largest element

"removes" the largest element (makes heap smaller)

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);
```

O(n)

n times

n log n

log n

Running time

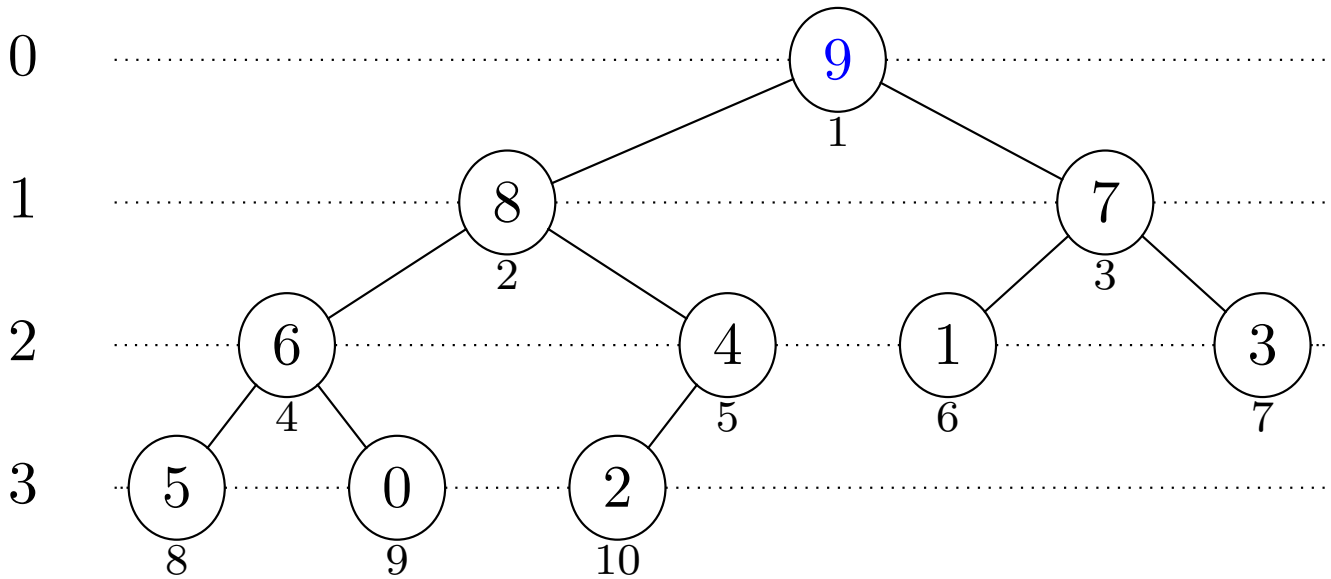
- Heap sort runs in time $O(n) + nO(\lg n) = O(n \lg n)$

build heap

guaranteed (also in worst case)

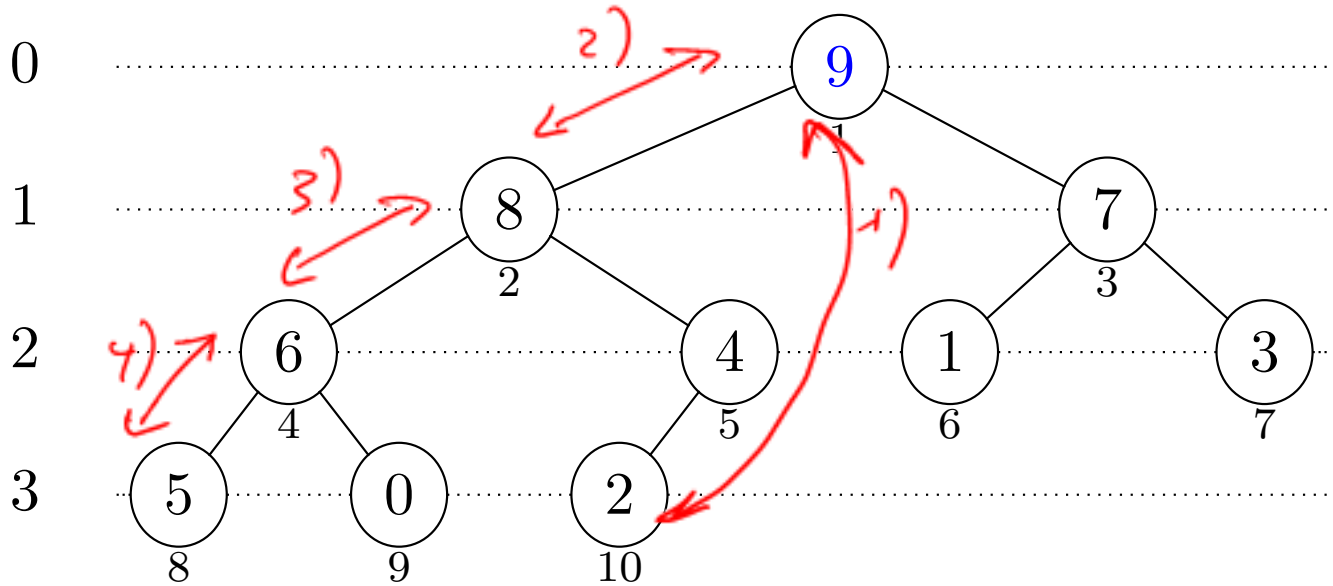
Heap sort algorithm/2

An example/1



HEAP SORT ALGORITHM/2

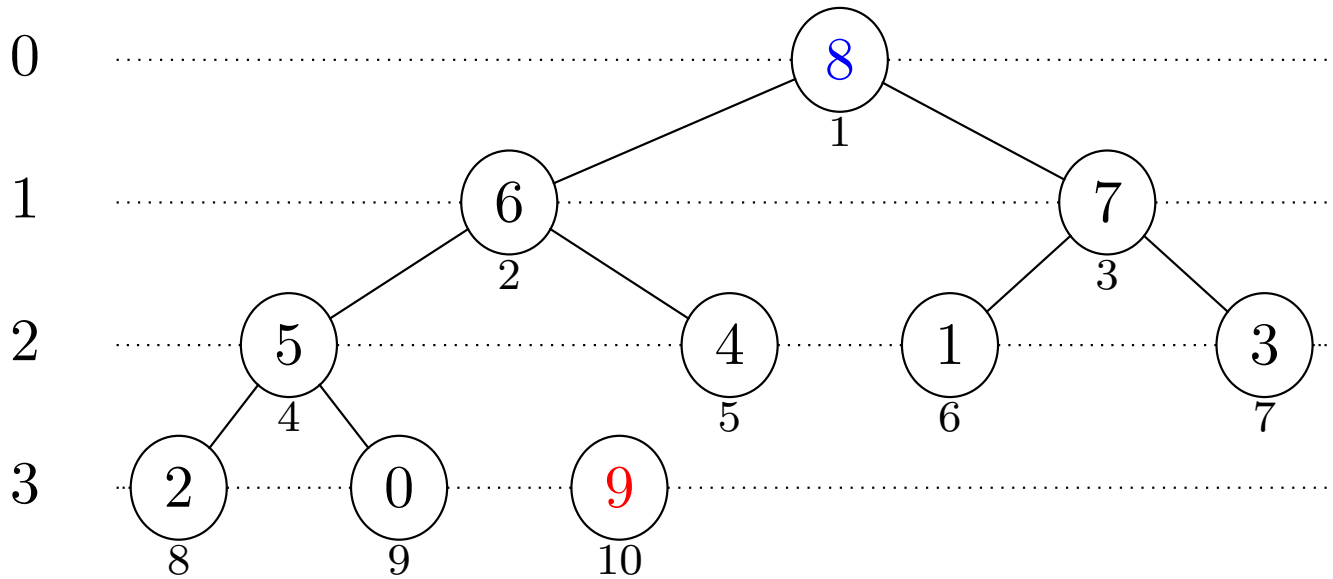
An example/1



node	9	8	7	6	4	1	3	5	0	2
index	1	2	3	4	5	6	7	8	9	10

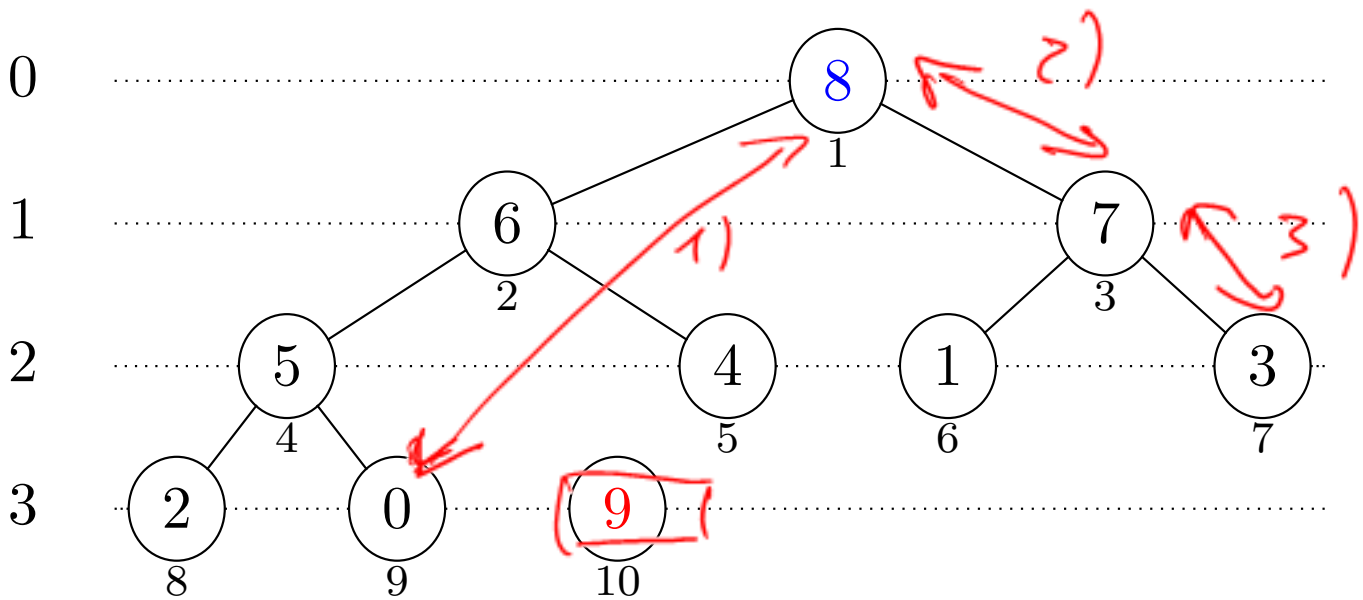
Heap sort algorithm/3

An example/2



HEAP SORT ALGORITHM/3

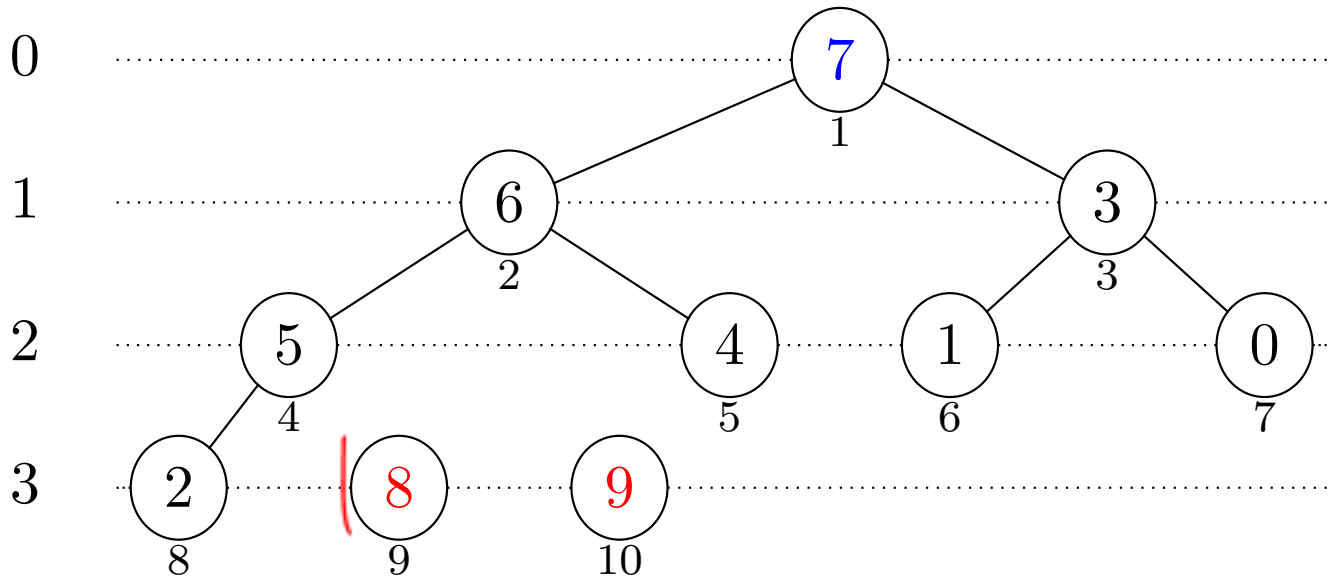
An example/2



node	8	6	7	5	4	1	3	2	0	9
index	1	2	3	4	5	6	7	8	9	10

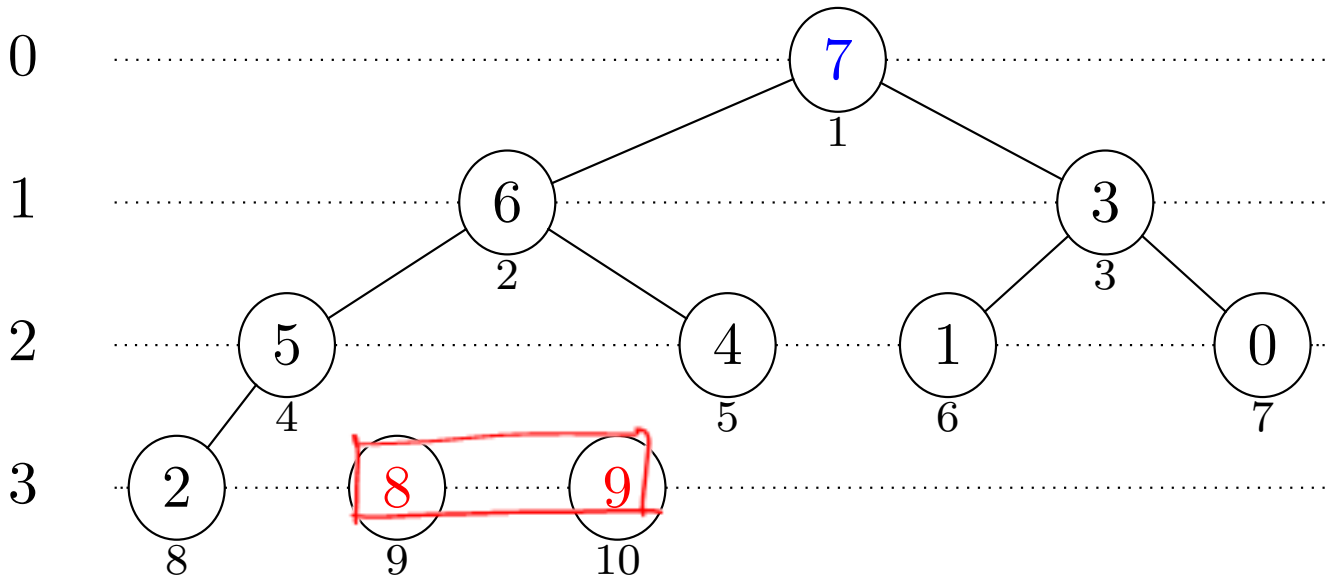
Heap sort algorithm/4

An example/3



Heap sort algorithm/4

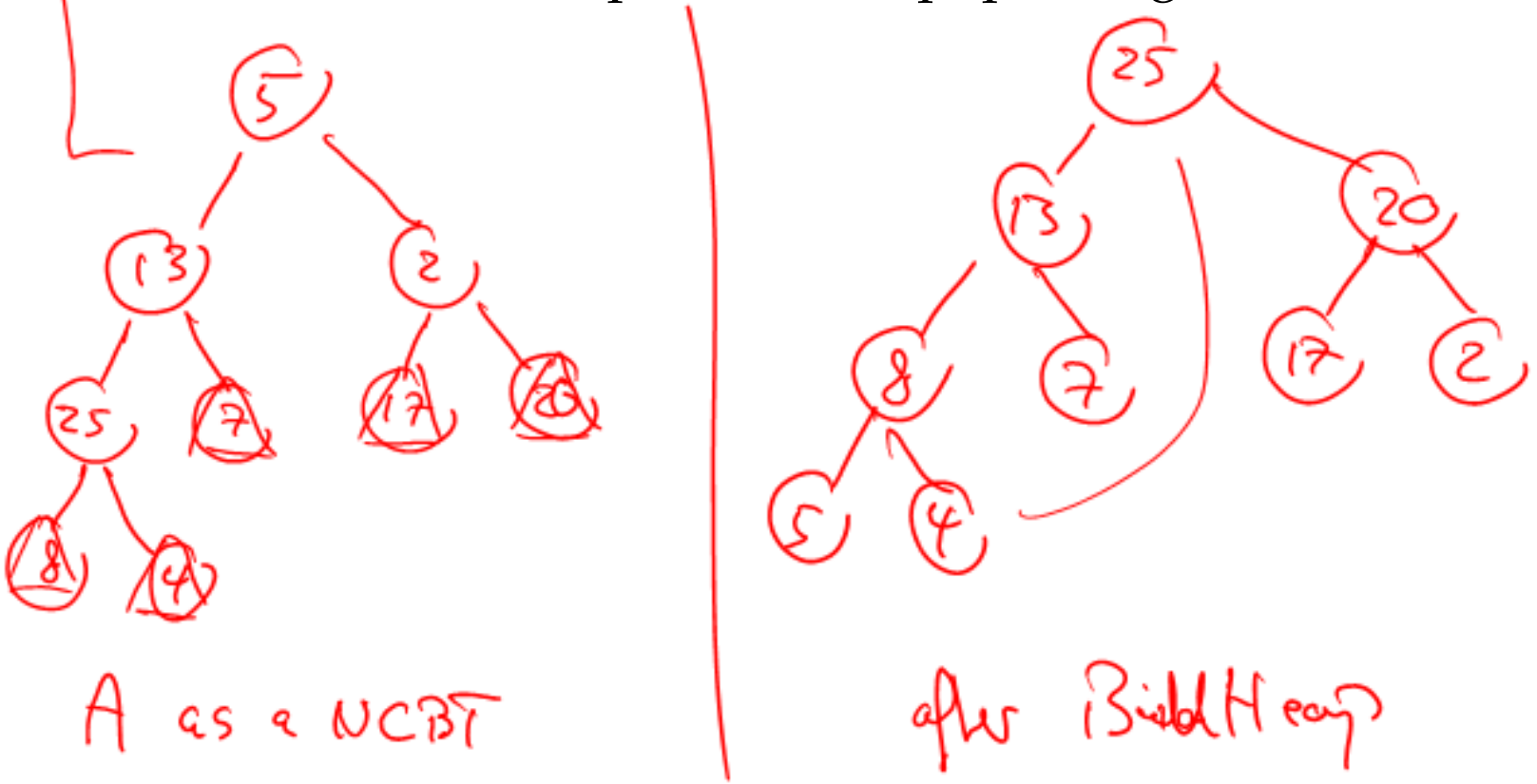
An example/3



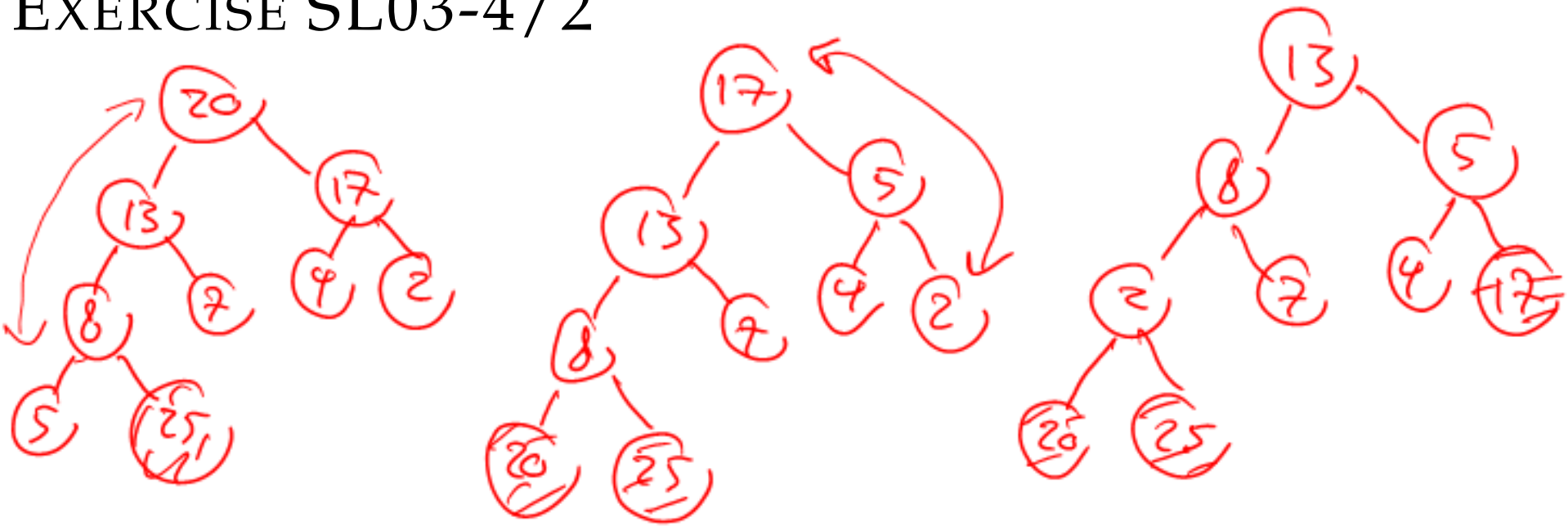
node	7	6	3	5	4	1	0	2	8	9
index	1	2	3	4	5	6	7	8	9	10

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



[25, 13, 20, 8, 7, 17, 2, 5, 4]

[20, 13, 17, 8, 7, 4, 2, 5, 25]

[17, 13, 5, 8, 7, 4, 2, 20, 25]

[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 $\leq$ the items in the upper part
  - Conquer: recursively sort the two subarrays
  - Combine: trivial since sorting is in place

QUICK SORT

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**
  m = LomutoPartition(A,l,r);
  QuickSort(A,l,m-1);
  QuickSort(A,m+1,r);

smallest elements

largest elements

⇒ no merge at the end

key: how to partition? (correctly & efficiently)

LOMUTO PARTITIONING / 1

Lomuto partitioning

```
Algo: LomutoPartition(A,l,r)
x = A[r];
i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

middle element (pivot)
(we take rightmost element as middle element)

elements l..r-1 are inserted into either the smaller or larger part

smaller part grows by one

Lomuto Partitioning/2

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

- $i = 0, j = 1$

2	7	6	1	3	5	4

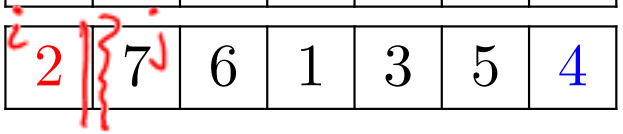
Lomuto Partitioning / 2

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

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$



Lomuto Partitioning / 2

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

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

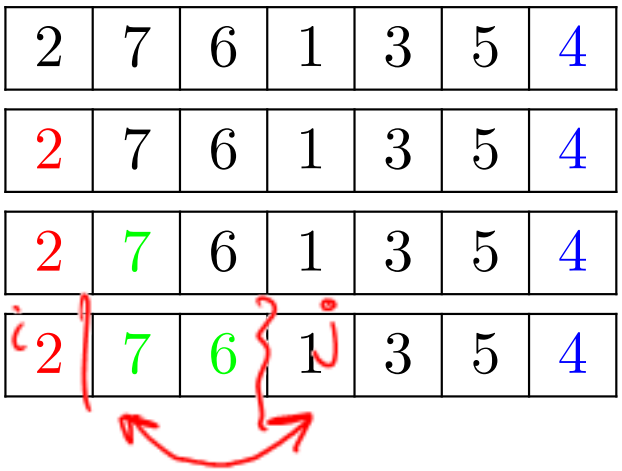
- $i = 1, j = 3$

2	7	6	1	3	5	4

Lomuto Partitioning / 2

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

- $i = 0, j = 1$
- $i = 1, j = 2$
- $i = 1, j = 3$
- $i = 1, j = 4$



Lomuto Partitioning / 2

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

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4

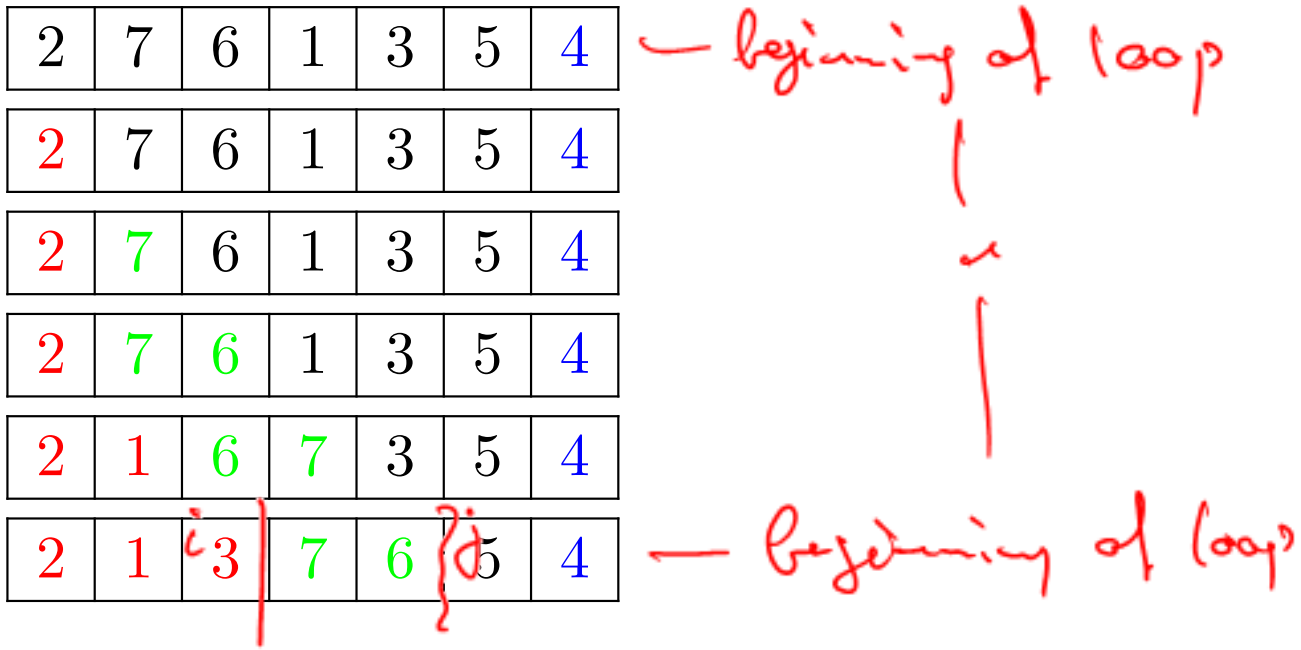
- $i = 2, j = 5$

2	1	6	7	3	5	4

Lomuto Partitioning / 2

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

- $i = 0, j = 1$
- $i = 1, j = 2$
- $i = 1, j = 3$
- $i = 1, j = 4$
- $i = 2, j = 5$
- $i = 3, j = 6$



Lomuto Partitioning/2

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

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4

- $i = 2, j = 5$

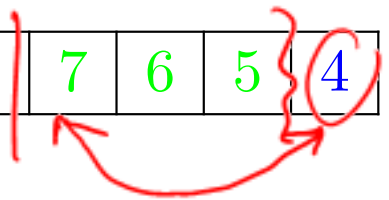
2	1	6	7	3	5	4

- $i = 3, j = 6$

2	1	3	7	6	5	4

- $i = 3, j = 6$

2	1	3	7	6	5	4



end of loop

Lomuto Partitioning/2

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

- $i = 0, j = 1$ | 2 | 7 | 6 | 1 | 3 | 5 | 4 |
- $i = 1, j = 2$ | 2 | 7 | 6 | 1 | 3 | 5 | 4 |
- $i = 1, j = 3$ | 2 | 7 | 6 | 1 | 3 | 5 | 4 |
- $i = 1, j = 4$ | 2 | 7 | 6 | 1 | 3 | 5 | 4 |
- $i = 2, j = 5$ | 2 | 1 | 6 | 7 | 3 | 5 | 4 |
- $i = 3, j = 6$ | 2 | 1 | 3 | 7 | 6 | 5 | 4 |

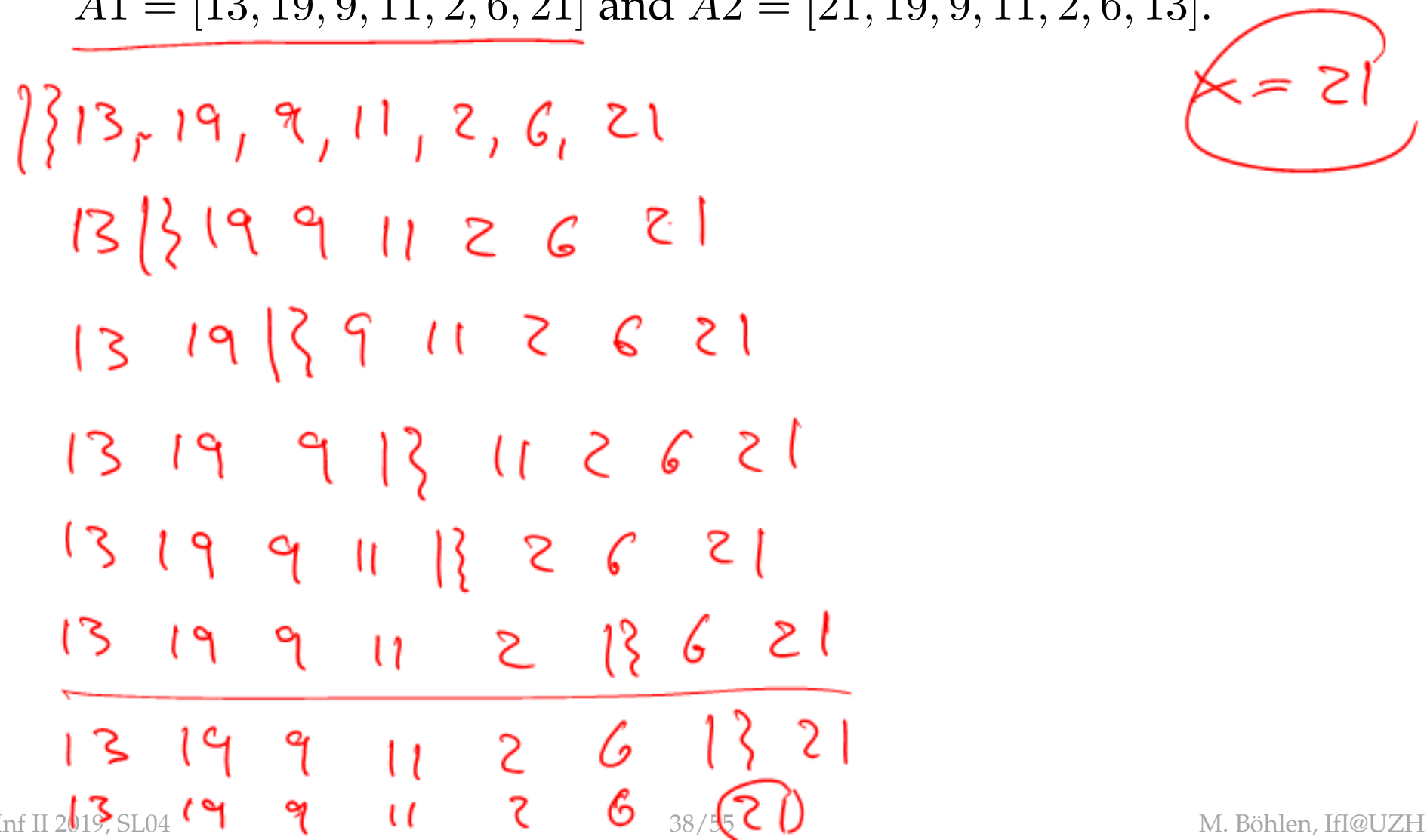
- $i = 3, j = 6$ | 2 | 1 | 3 | 7 | 6 | 5 | 4 |
- $i = 3, j = 6$ | 2 | 1 | 3 | 4 | 6 | 5 | 7 |

smallest elements

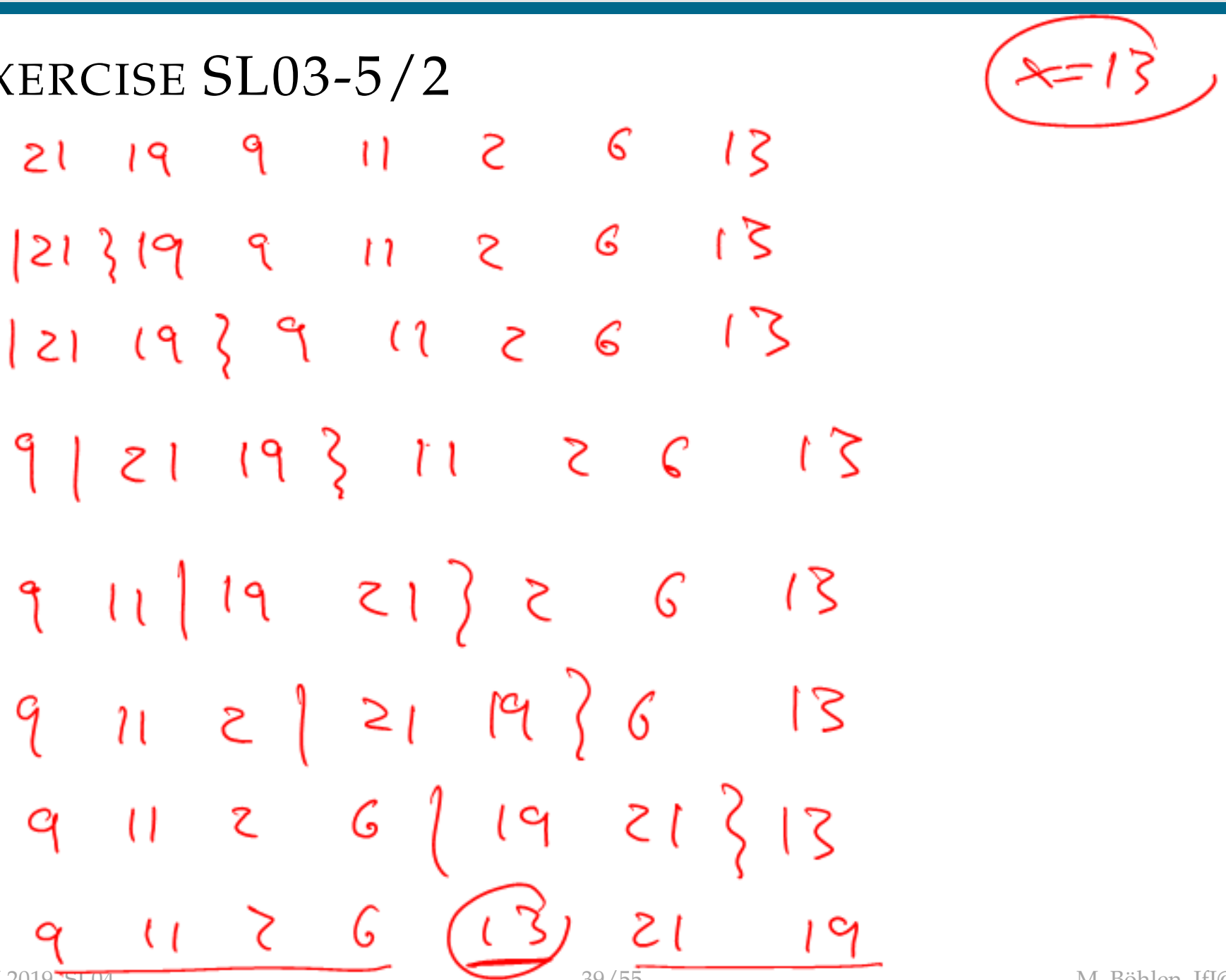
largest elements

EXERCISE SL04-5/1

Demonstrate Lomuto partitioning on the arrays
$A1 = [13, 19, 9, 11, 2, 6, 21]$ and $A2 = [21, 19, 9, 11, 2, 6, 13]$.

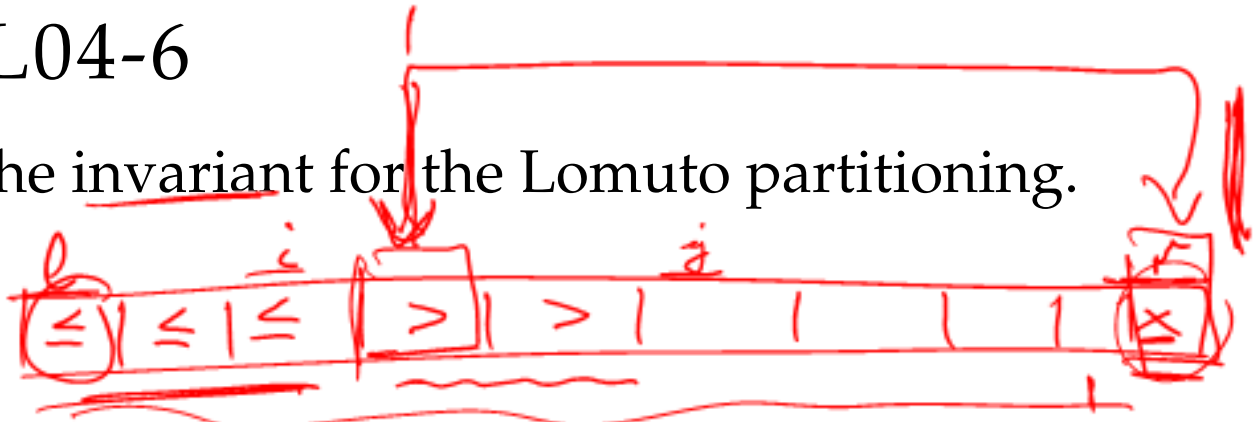


EXERCISE SL03-5/2



EXERCISE SL04-6

Formulate the invariant for the Lomuto partitioning.



invariant (start of loop)

- $A[\ell..i] \leq x$
- $A[i+1..j-1] > x$
- $A[r] = x$

termination:

- $\ell - 1 \leq i \leq r$
- $i = \ell - 1$: $A[\ell..\ell-1]$ is empty
- $i = r - 1$: $A[r..r-1]$ is empty

elem ?! why ok?

because there is a middle element left middle elem right

- $i = j = r-1$: $A[r] \leftrightarrow A[r]$
- $i < j = r-1$: $A[i+1] \leftrightarrow A[r]$ (first elem of large part)
- $i = 0 < j = r-1$: $A[1] \leftrightarrow A[r]$

Hoare Partitioning/1

searches in right part for a wrong (too small) element

searches in left part for a wrong (too large) element

swap of "wrong" elements

Hoare partitioning

Algo: HoarePartition(A,l,r)

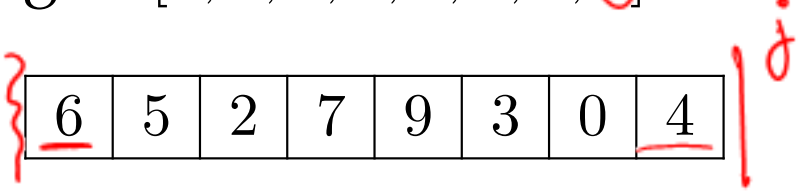
```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then exchange A[i] and A[j] else return i;
```

Hoare Partitioning/2



Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$



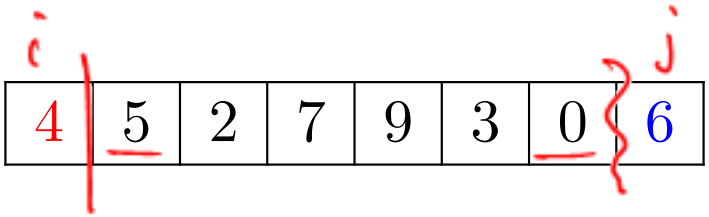
Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

6	5	2	7	9	3	0	4

- $i = 1, j = 8$



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

- $i = 1, j = 8$

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

EXERCISE SL04-7

Demonstrate Hoare partitioning on the arrays
$A1 = [13, 19, 9, 11, 2, 6, 21]$ and $A2 = [21, 19, 9, 11, 2, 6, 13]$.

A1: 13 19 9 11 2 6 21 i=0, j=8, x=21

13 19 9 11 2 6 21 i=7, j=7

i=7 is returned

A2: 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 19 21 i=2, j=6

13 6 9 11 2 19 21 i=6, j=5

i=6 is returned

EXERCISE SL04-8

Formulate the invariant for the Hoare partitioning.

invariant: - $A[l..i-1] \leq x$
- $A[j+1..r] \geq x$

} not disjoint

termination: $l < i \leq r$

5 4 3 2 $x=2$ $i=1$ $j=4$
2 4 3 5 $i=2$ $j=1$

2 3 4 5 $x=5$ $i=4$ $j=4$

$A[l..i-1]$ is never empty

$A[l..i-1]$ and $A[i..r]$ are both non-empty

EXERCISE SL04-9

Specify the quick sort algorithm for Hoare partitioning.

```
Alg  QuickSort(A, l, r)
       if l < r then // more than 1 element
           m = HoarePartition(A, l, r)
           QuickSort(A, l, m-1)
           QuickSort(A, m, r)
```

EXERCISE SL04-10/1

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 $m + 1$

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);
```

EXERCISE SL03-10/2

1. QS([1,5], 1, 2) → P([1,5], 1, 2), x = 1, w = 1
   QS([1,5], 1, 0) ✓
   QS([1,5], 1, 2) ∞ loop

2. QS([1,5], 1, 2) → P([1,5], 1, 2) x = 5
   i = 2, j = 2, w = 2
   j = 1
   i = 3 A[3] ↯ index out of bound

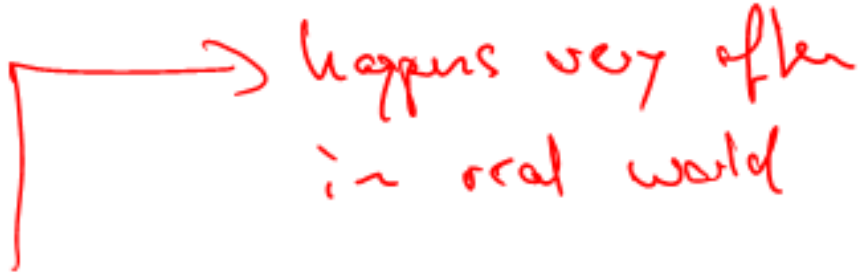
3. QS([3], 1, 1) →
   P([3], 1, 1), x = 3, i = 1, j = 1
   QS([3], 1, 0), x = A[0] ↯ index out of bound

4. QS([1,5], 1, 2) → P([1,5], 1, 2), x = 5
   i = 2, j = 2
   → QS([1,5], 1, 2) ∞ recursion

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 elements (counts partitioning)

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

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., $\sum_{i=0}^{n} i$)

$+\Theta(n-1)$
$+\Theta(n-2)$
$\vdots$
$+\Theta(1)$

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

$$T(n) = 2T(n/2) + \Theta(n)$$

same as merge sort or heap sort

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, $T(n) = \Theta(n \lg n)$

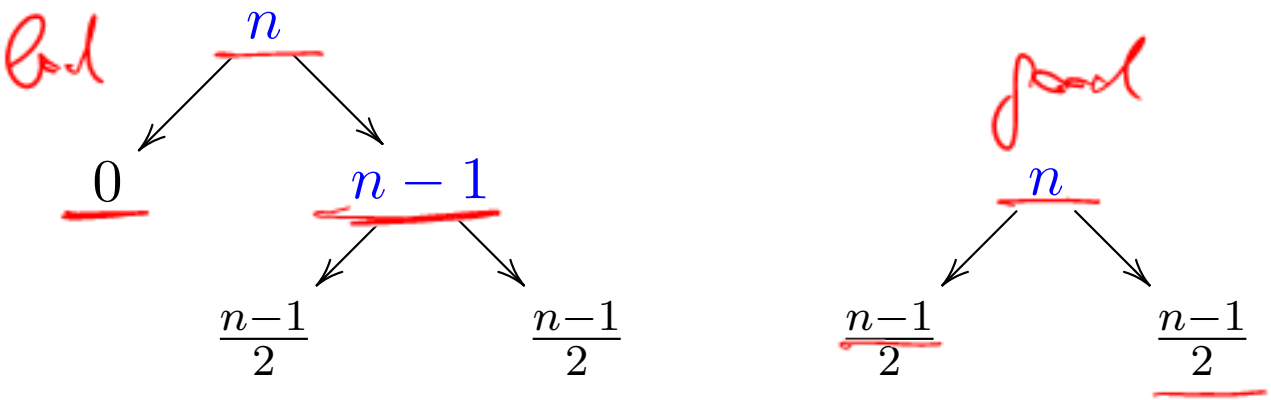
Average case/1

- 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

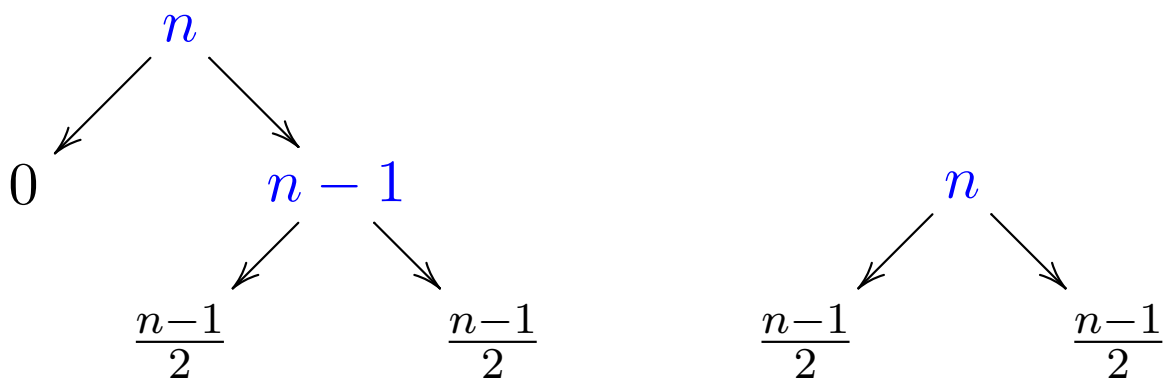
- Assume that "bad" (B) and "good" (G) splits alternate



Performance/3

Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

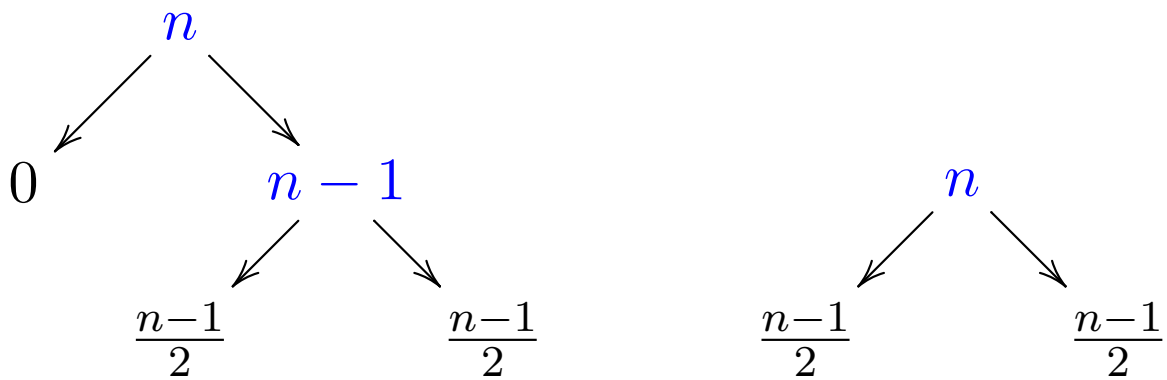


- $G(n) = 2B(n/2) + \Theta(n)$, and
- $B(n) = G(n-1) + \Theta(n)$

PERFORMANCE/3

Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

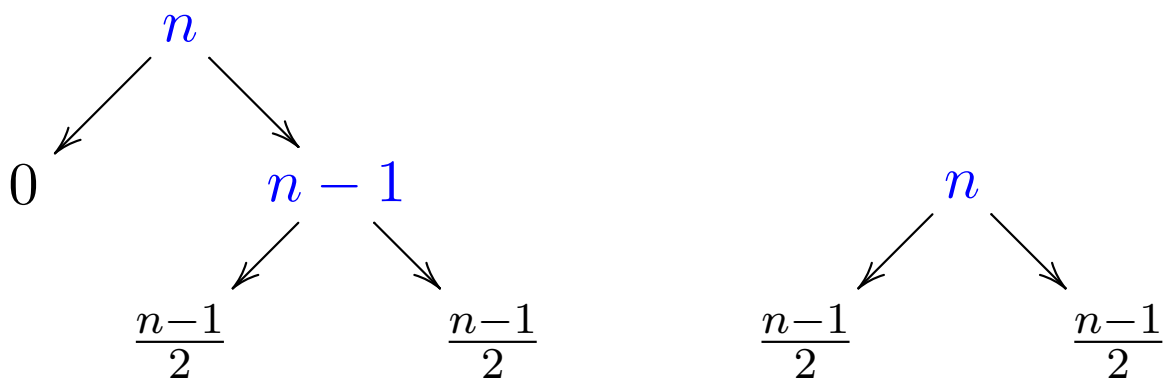


- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$
- $G(n) = 2B(n/2) + \Theta(n)$
  $= 2G(n/2-1) + \Theta(n/2) + \Theta(n)$
  $= 2G(n/2-1) + \Theta(n) \leq 2G(n/2) + \Theta(n)$

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 SL04-11

A = [5, 5, 5, 5, 5]

Consider an array where all elements are equal. Is this a worst case, best case or neither for quick sort?

Algo: LomutoPartition(A,l,r)

```
x = A[r]; i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

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
```

always use Hoare if equal elements are possible

A = [≤|≤|≤|≤|≤|>|>|>|]

worst case for Lomuto

splits: n-1, 1, 0

A = [≤|≤|≤|≥|≥|≥|≥|≥|]

Best case for Hoare

split 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

Randomized algorithm

Algo: RandomizedLomutoPartition(A,l,r)

```
i = Random(l,r);
exchange A[i] and A[r];
return LomutoPartition(A,l,r);
```

Algo: RandomizedQuickSort(A,l,r)

```
if l<r then
    m = RandomizedLomutoPartition(A,l,r);
    RandomizedQuickSort(A,l,m-1);
    RandomizedQuickSort(A,m+1,r);
```

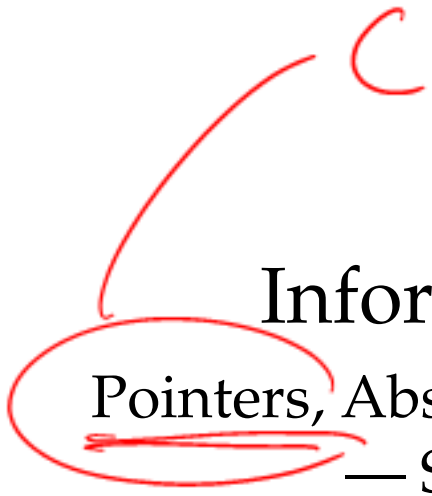
SUMMARY / 1

- 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:

	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



Informatik II

Pointers, Abstract Data Types

— SL05 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

- arrays: fixed size
- no fixed size is known → dynamic structures

TABLE OF CONTENTS — SL05

1. Dynamic data structures
   Pointers
   Linked lists

2. ADT: abstract data types

one mechanism to be dynamic

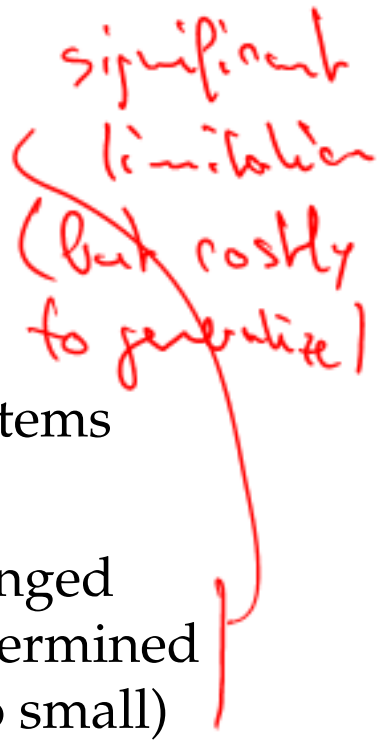
DYNAMIC DATA STRUCTURES/1

Arrays as data structures

aka dictionary

- 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

memory (space)

DYNAMIC DATA STRUCTURES/2

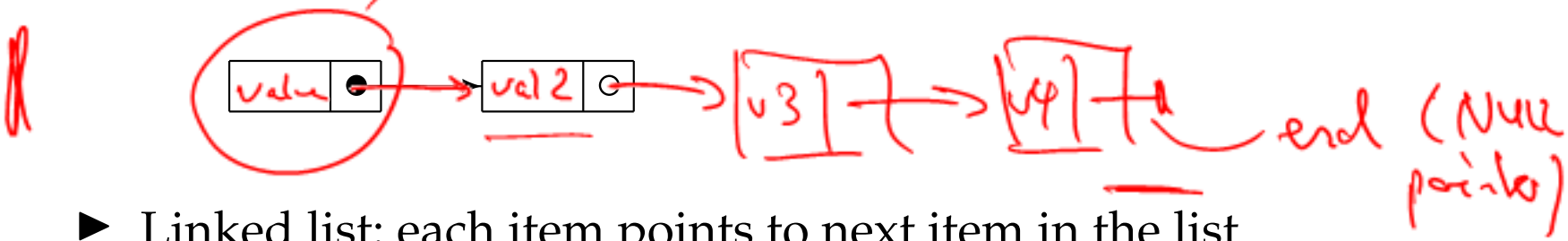
Linked data structures

- 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

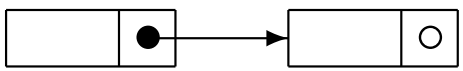


- Linked list: each item points to next item in the list
- Binary tree: each node points to left & right subtree

DYNAMIC DATA STRUCTURES/2

Linked data structures

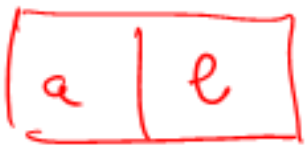
- Used to deal with collections of items dynamically
- Record contains fields that point to other records



  - 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 (LDS) is not known in advance
  - Depends on the number of items stored in the LDS
  - Only known during runtime, i.e. program execution
  - Linked data structures can grow and shrink in size

allocating/deallocating space is expensive

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 -o dummy dummy.c ; ./dummy
```

r = a | e

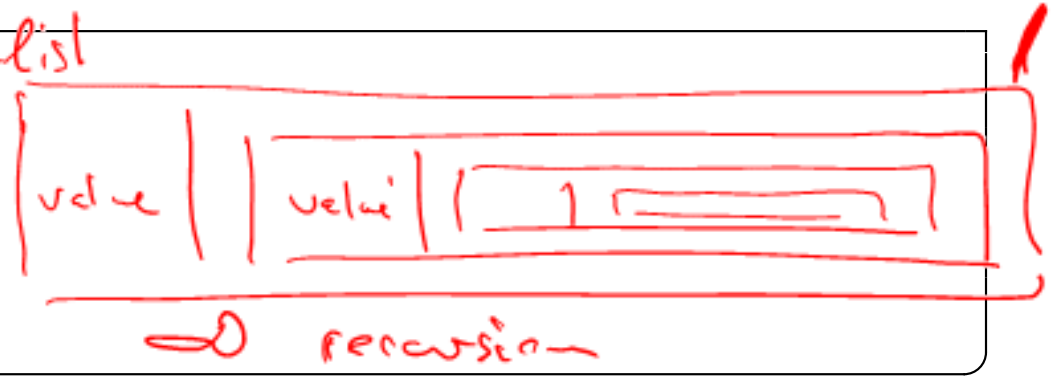
access to component b of record r

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

computer memory = (huge) array

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

index (hex numbers)

value

- 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 (as NULL)

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
- 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

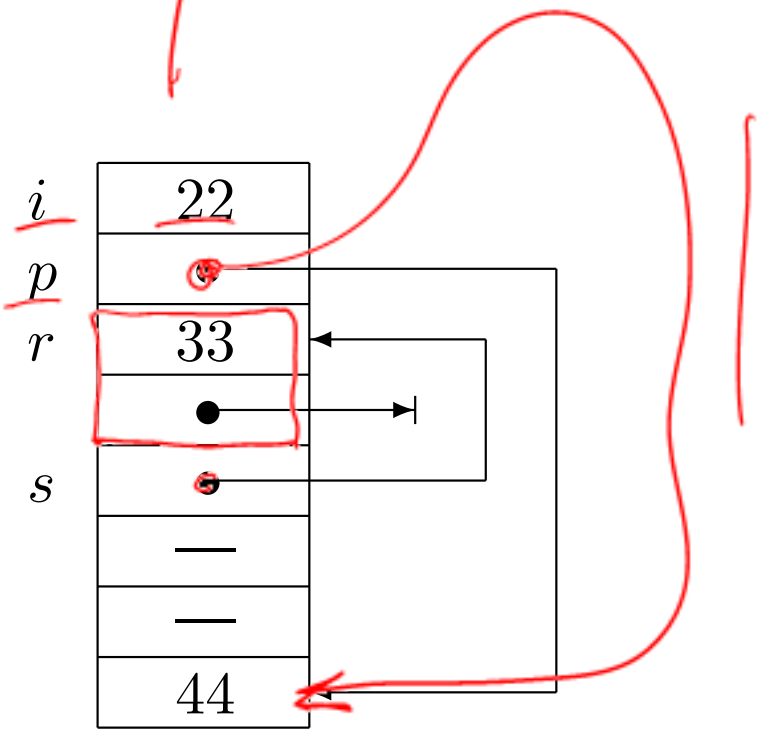
	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	`NIL`
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

POINTERS / 3

simpler, easier to understand

Alternative presentation

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44



always work with pictures when using pointers

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

operations with pointers:

- To follow (chase, dereference) a pointer we write *p

```
*p = 12
```

used very often

- To get the address of a variable i we write &i

```
p = &i
```

used much less often

- 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
```

declare a pointer to an integer

- 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

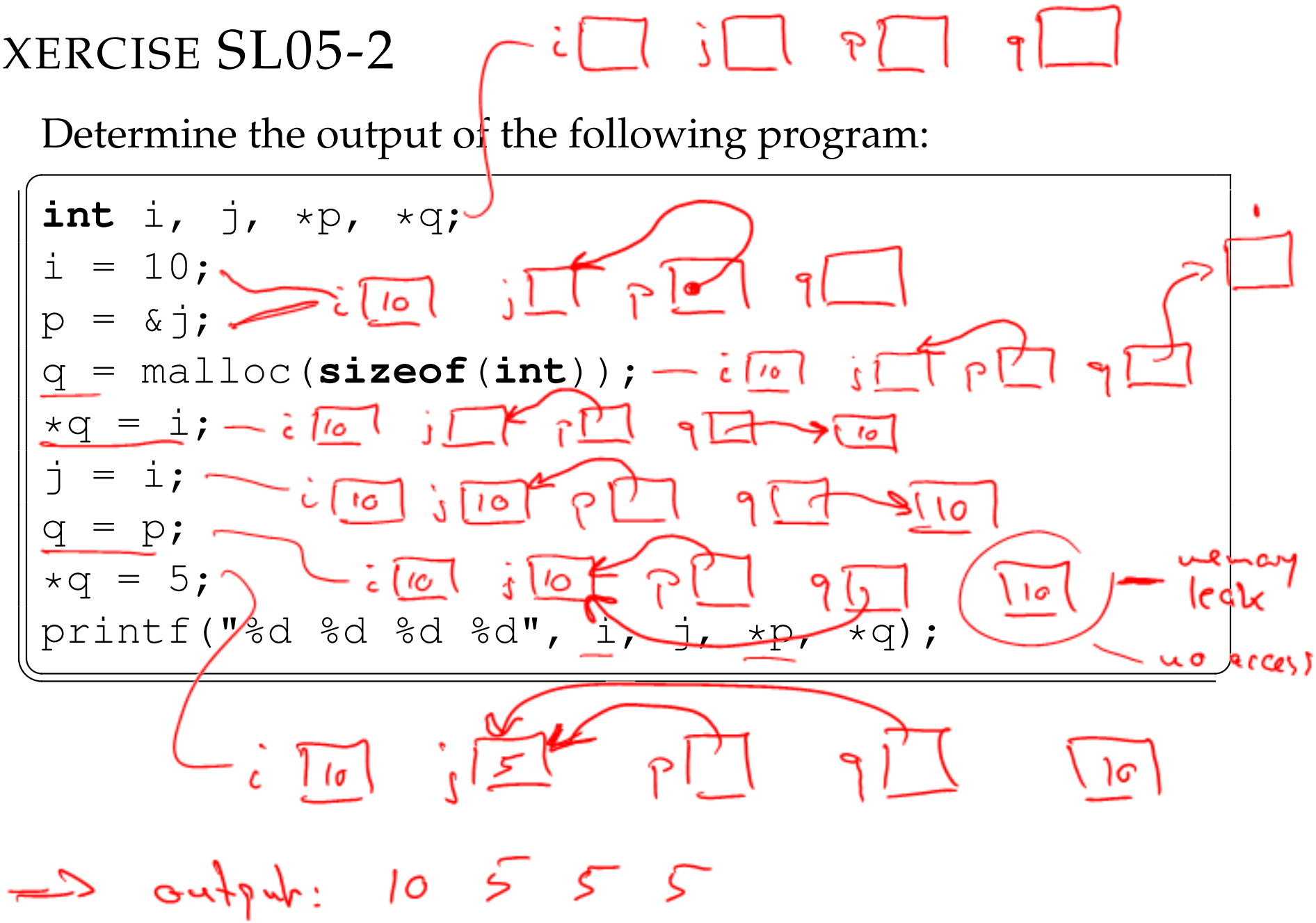
struct rec* s;
s = &r
```

	address	content
i	$1af783$	23
p	$1af784$	$1af78a$
r	$1af785$	17
	$1af786$	24
s	$1af787$	$1af785$
	$1af788$	
	$1af789$	
	$1af78a$	55

EXERCISE SL05-2

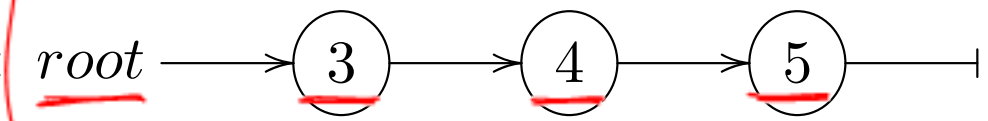
Determine the output of the following program:

```
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);
```



LINKED LISTS/1

- A list of integer:



- Notational convention:



- Corresponding declaration in C:

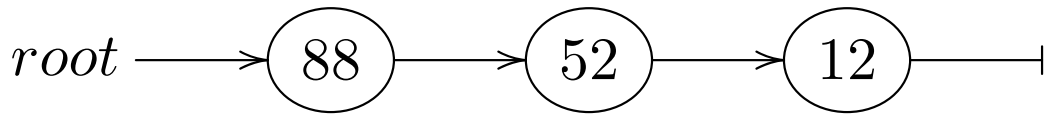
```
struct node {
  int val;
  struct node* next;
};

struct node* root;
```

- Accessing a field: `(*p).a` or more convenient `p->a`

LINKED LISTS/2

Populating a list with integers:



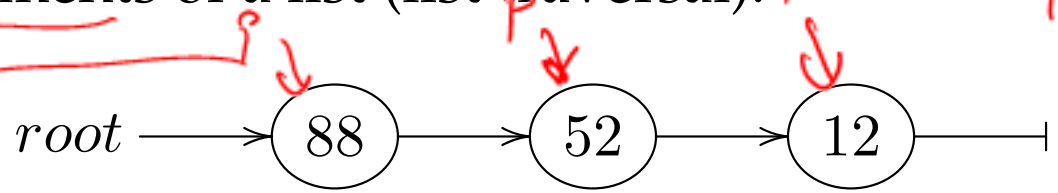

```
root = malloc(sizeof(struct node));
root->val = 88;
root->next = malloc(sizeof(struct node));

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");
```

88 52 12

EXERCISE SL05-3

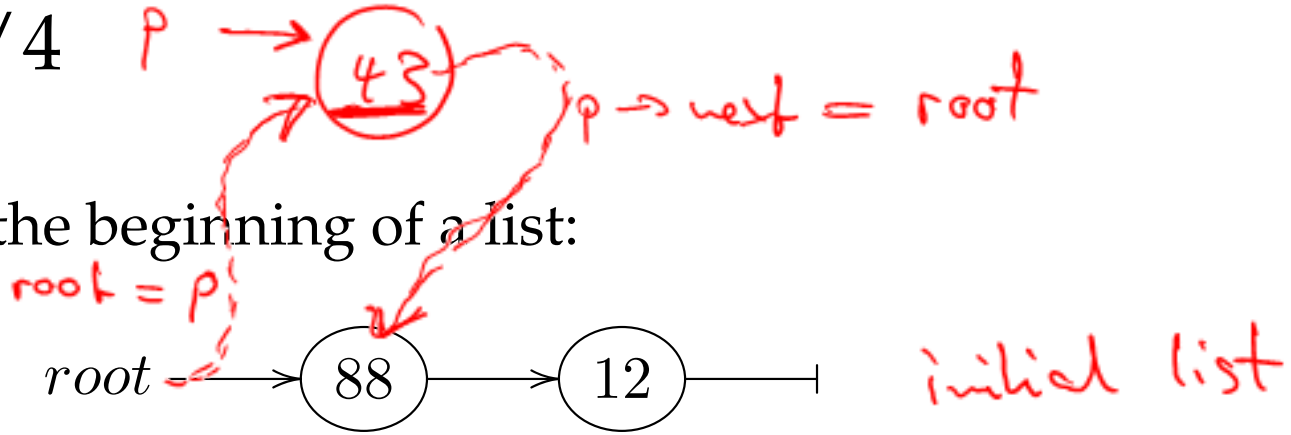
Count the occurrences of an integer x in a linked list.




```
cnt=0;
p = root;
while ( p ≠ NULL ) {
    if (p→val ==x) { cnt++; }
    p = p→next;
}
```

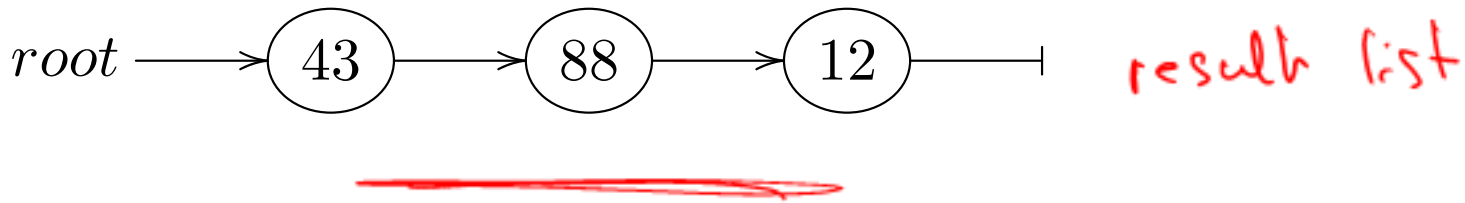
LINKED LISTS/4

Inserting 43 at the beginning of a list:



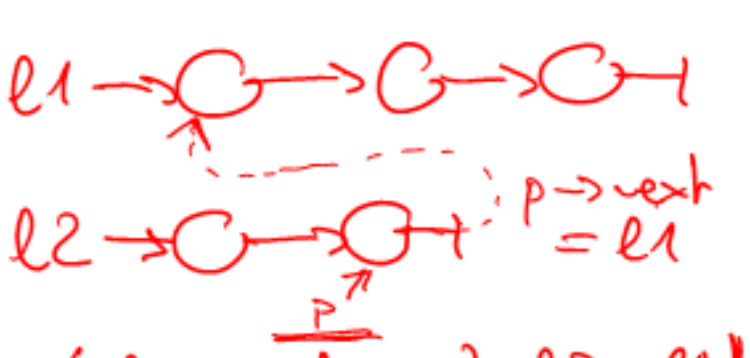

```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```

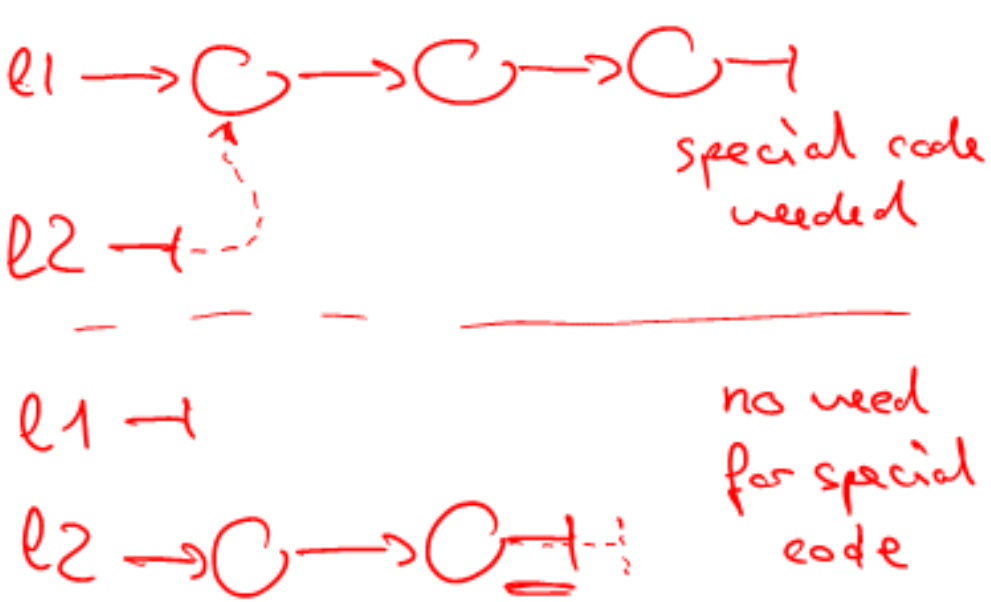
4 operations
constant time



EXERCISE SL05-4

Give C code that appends linked list l1 to linked list l2.

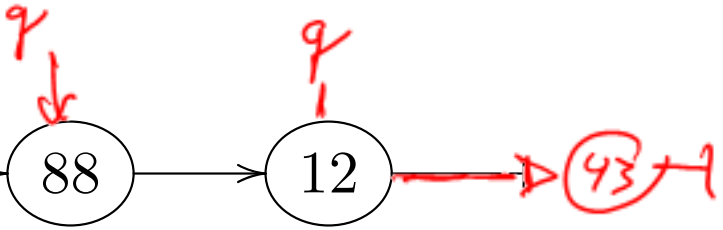





```
if (l2 == NULL) l2 = l1
else {
    p = l2
    while (p->next != NULL) p = p->next;
    p->next = l1
}
```

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;
}
```

special case

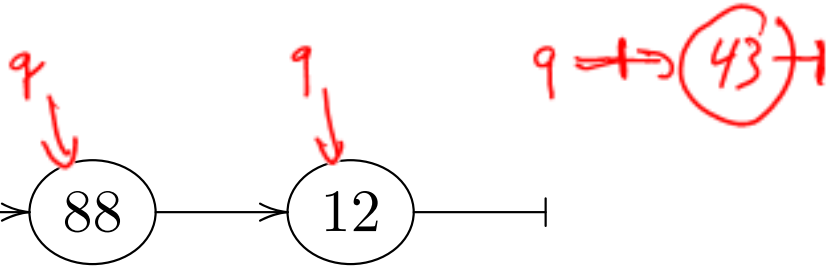
default case

as long as we are not at last element

root → 43

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;
```

new node is disconnected and not reachable through root

look ahead: q->next ≠ NULL

problem: we went too far while traversing the list

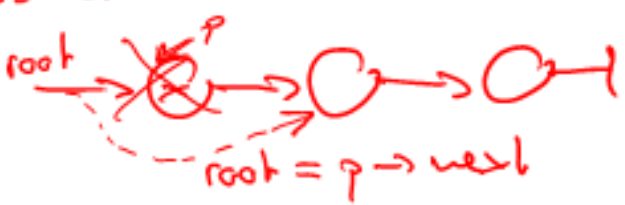
LINKED LISTS/7

assumption: x appears in the list

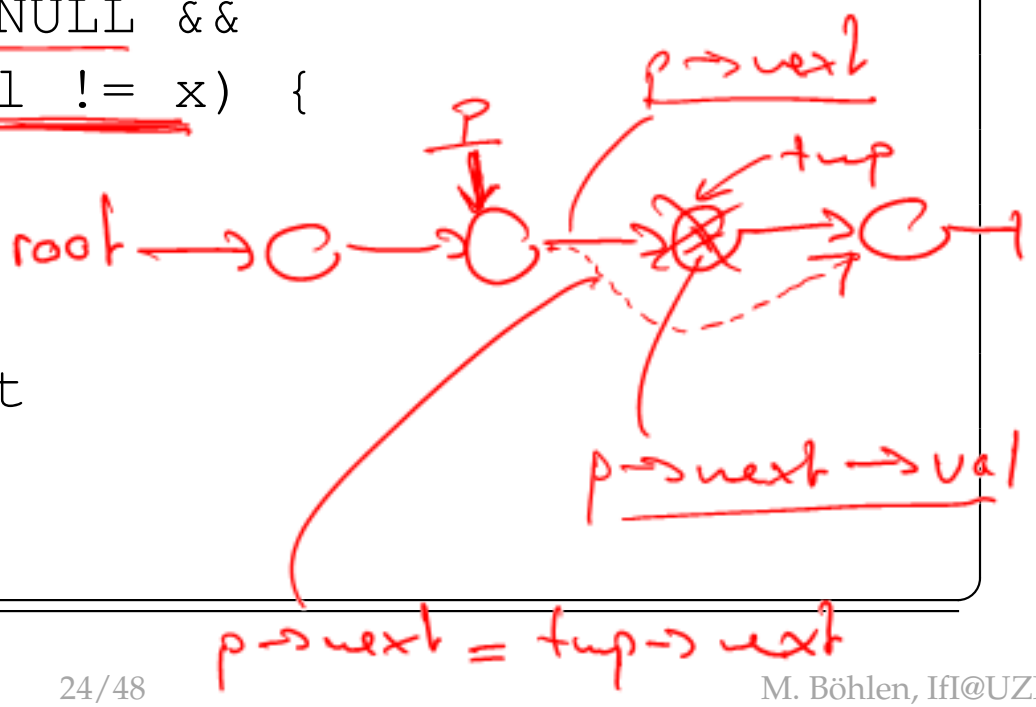
Delete element x from a non-empty list:

```
p = root;
if (p->val == x) {
  root = p->next;
  free(p);
} else {
  while (p->next != NULL &&
         p->next->val != x) {
    p = p->next;
  }
  tmp = p->next;
  p->next = tmp->next
  free(tmp);
}
```

x is first element

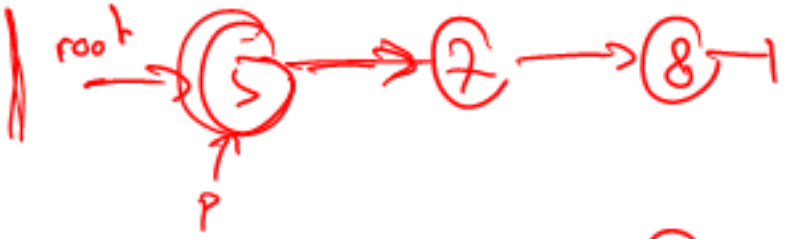


x is not first element



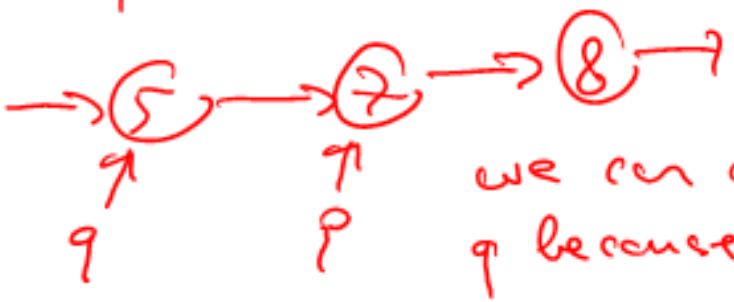
EXERCISE SL05-5

Write C code to delete a linked list.




```
p = root;
while (p ≠ NULL) {
  free(p);
  p = p->next;
}
```

wrong ☹ after free we cannot access space anymore



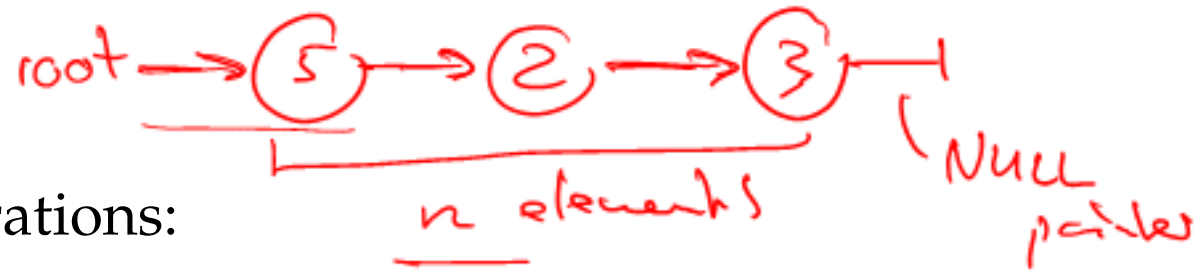
we can delete q because we have p

(q is one step behind p)

recursive solution:

```
p = root;
while (p ≠ NULL) {
  - q = p
    p = p->next
    free(q)
```

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`.

- 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

- more pointers (only few)
- different use of pointers

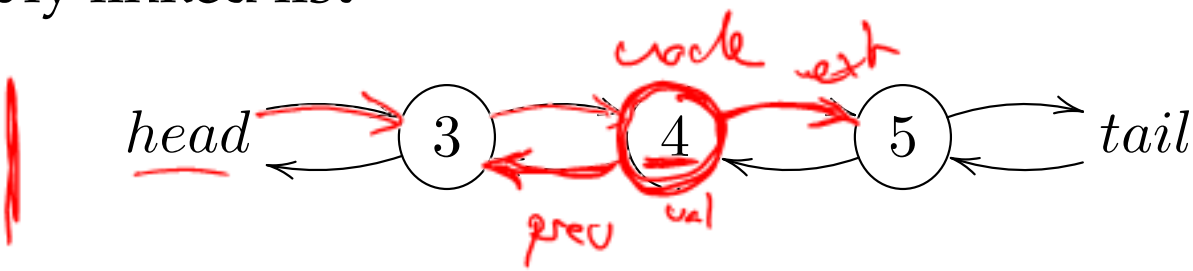
Variants of linked lists

- 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

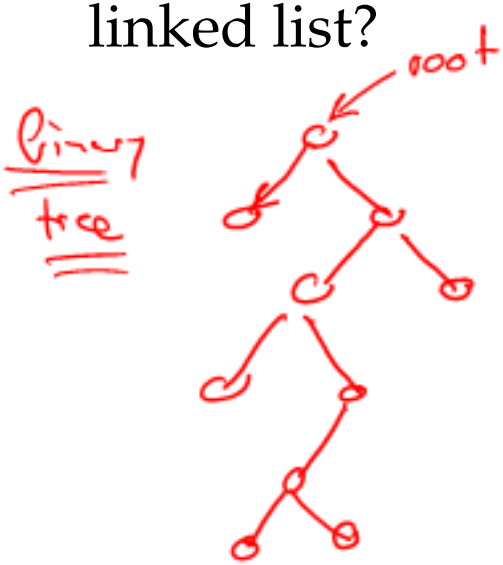


  - Each node has a field with a pointer to the previous node of the linked list
  - Provides means to quickly navigate forth and back in the linked list

EXERCISE SL05-6

very similar; but use of pointers is different

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* left;
    struct node* rgt;
};

struct node* root;
```

used very often



doubly linked list

```
struct node {
    int val;
    struct node* next;
    struct node* prev;
};

struct node* head;
struct node* tail;
```

used less often

benefits are not huge

TABLE OF CONTENTS — SL05

1. Dynamic data structures

2. ADT: abstract data types
   - Stacks
   - Queues
   - Ordered lists

ABSTRACT DATA TYPES/1

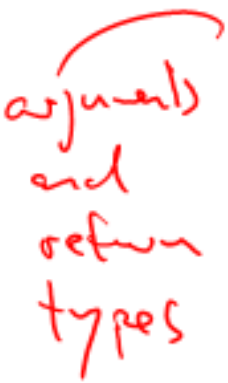
Informal definition

- 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

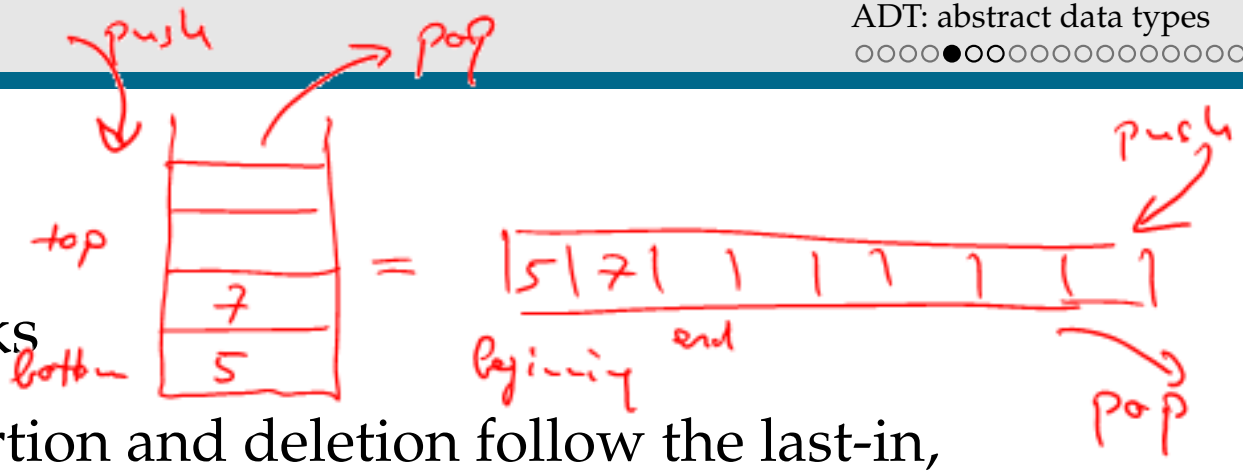
Similarity with OO-paradigm

- 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

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
- Implemented by a data structure where items are
  - inserted at the end/beginning (`push`)
  - removed from the end/beginning (`pop`)

top
top

at the same side of the stack

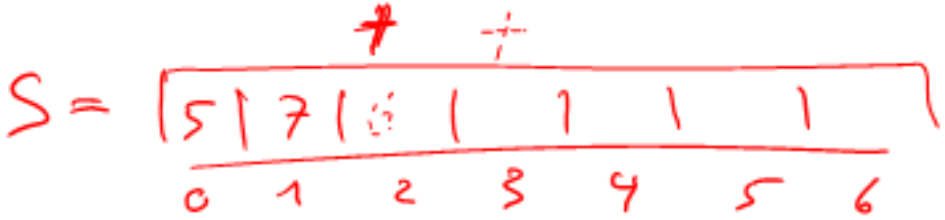
STACKS/1

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
- Implemented by a data structure where items are
  - inserted at the end/beginning (`push`)
  - removed from the end/beginning (`pop`)
- Appropriate data structures for their implementation
  - arrays (end) or singly linked lists (beginning)

We design a solution for a single stack (i.e., stack is implicit; not a parameter in operations)

STACKS/2



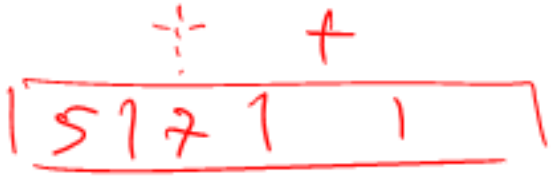
Array implementation

- 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; t = t+1;

Algo: pop()
t = t-1; **return** S[t];

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)
S[t] = x; t = t+1;

Algo: pop()
t = t-1; **return** S[t];

- The stack is implemented by an array $S[1..n]$ of length n

Pseudocode

(or S[0..n-1])C

STACKS / 2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)

```
S[t] = x;
t = t+1;
```

Algo: pop()

```
t = t-1;
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.

t points to first free element

(t can also point to last inserted element)

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)	**Algo:** pop()
S[t] = x; t = t+1;	t = t-1; **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

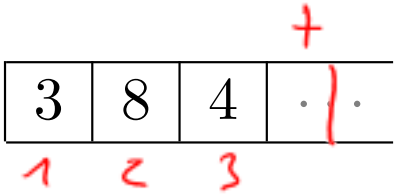
Algo: push(x)	**Algo:** pop()
S[t] = x; t = t+1;	t = t-1; **return** S[t];

- 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

STACKS / 3

A simple example

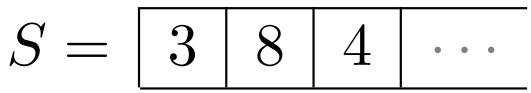
- Stack S containing 3 items ($t = 4$)

$S =$ 

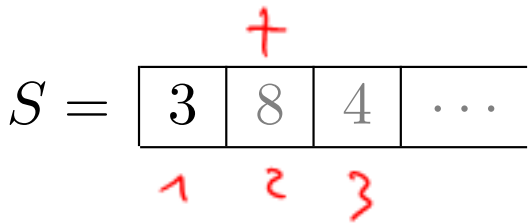
STACKS / 3

A simple example

- Stack S containing 3 items ($t = 4$)

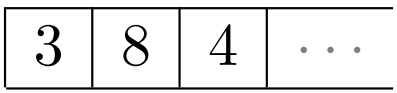


- Stack S after $2 \times$ `pop` () ($t = 2$)



STACKS / 3

A simple example

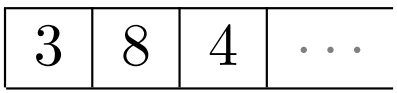
- Stack S containing 3 items ($t = 4$) $S =$ 

- Stack S after 2 × `pop` () ($t = 2$) $S =$ 

- Stack S after `push` (5) ($t = 3$) $S =$ 

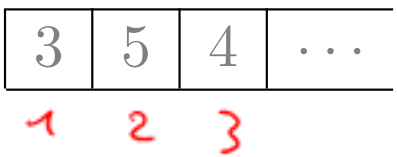
STACKS / 3

A simple example

- Stack S containing 3 items ($t = 4$) $S =$ 

- Stack S after 2 × `pop` () ($t = 2$) $S =$ 

- Stack S after `push` (5) ($t = 3$) $S =$ 

- Stack S after 2 × `pop` () ($t = 1$) $S =$ 

QUEUES/1

Properties of queues

- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle

QUEUES/1

Properties of queues

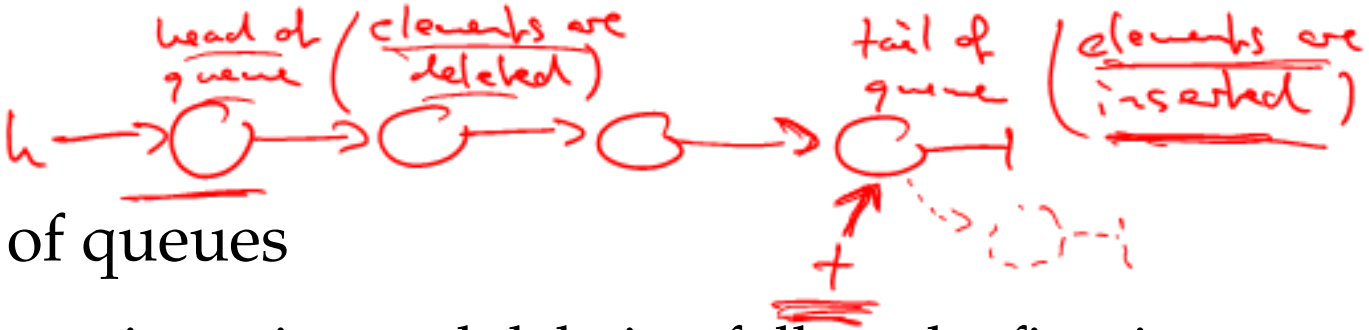
- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer

QUEUES/1

Properties of queues

- In a queue, insertion and deletion 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
- Implemented by a data structure where items are
  - inserted at the end/beginning (`enqueue`)
  - removed from the beginning/end (`dequeue`)

QUEUES/1



Properties of queues

- In a queue, insertion and deletion 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
- Implemented by a data structure where items are
  - inserted at the end/beginning (`enqueue`)
  - removed from the beginning/end (`dequeue`)

  at different sides of the queue
- Appropriate data structures for their implementation
  - arrays or singly linked lists with an explicit tail

We design a solution for a single queue (i.e., queue is implicit; not a parameter in operations)

QUEUES/2

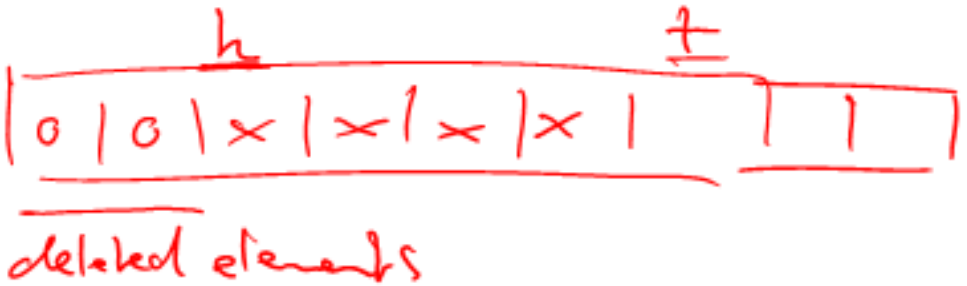
Array implementation

- 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 mod function
- Variables h and t function as pointers to head and tail of the queue

h = 1st element, t = 1st free element

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

Algo: dequeue()

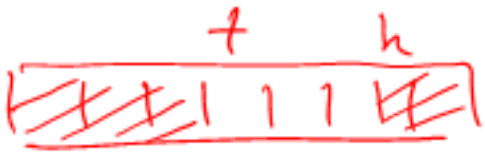
```
i = h;
h = h+1 mod n;
return Q[i];
```

- 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
- Notice that if $h = t$ then the queue is actually empty



QUEUES/3





A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4 \mid 0 \mid \cdots \mid 3 \mid 8}$$

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after 2 $\times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after 2 $\times$ `dequeue` () ($h = t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

Full queue:

t h

n-1 elements

h
t

ORDERED LISTS/1

- give a copy of doc to somebody
- copy is changed
- original is still unchanged

- In an ordered list ~~elemenst are ordered according~~ to a key.
- We design a solution for multiple ordered queues (i.e., ordered queue is an explicit parameter in operations).

- 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)`

pointer to a pointer to a node

why? ensures that l never changes

all arguments in procedures/functions are copies!

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

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;
}
```

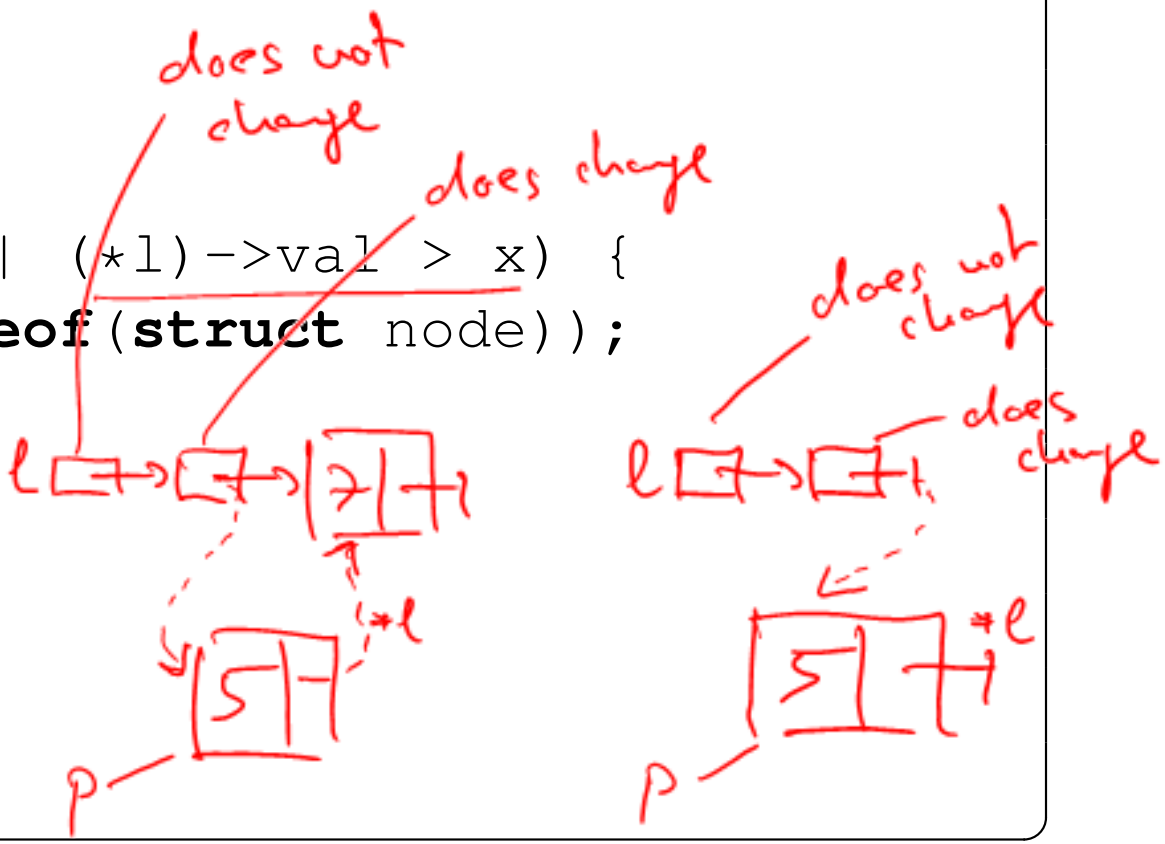
ORDERED LISTS/4

Insertion into an ordered list:

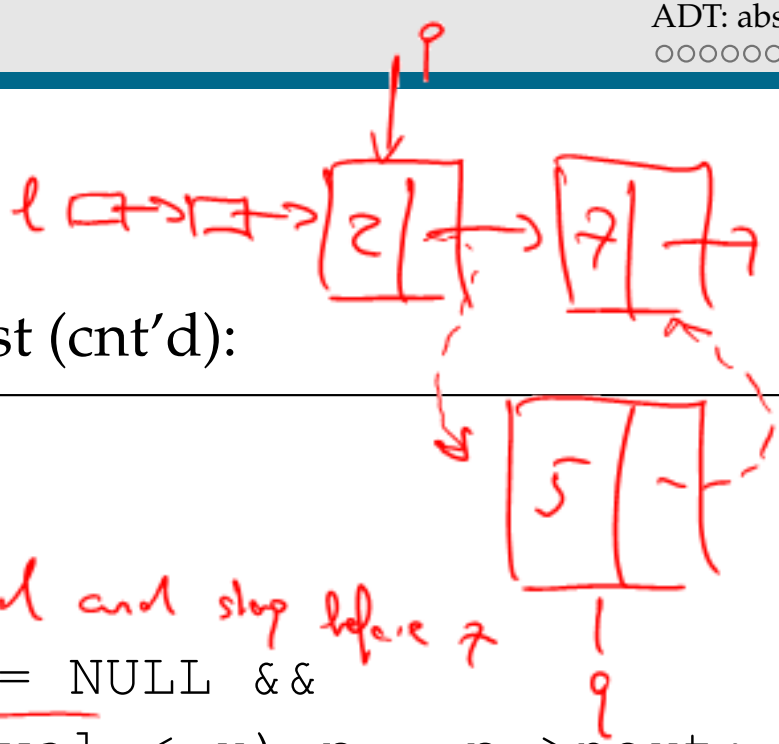
```
void insert(struct node** l, int x) {
  struct node* p;
  struct node* q;

  if (*l == NULL || (*l)->val > x) {
    p = malloc(sizeof(struct node));
    p->val = x;
    p->next = *l;
    *l = p;
  } else {

...
```



ORDERED LISTS/5



Insertion into an ordered list (cnt'd):

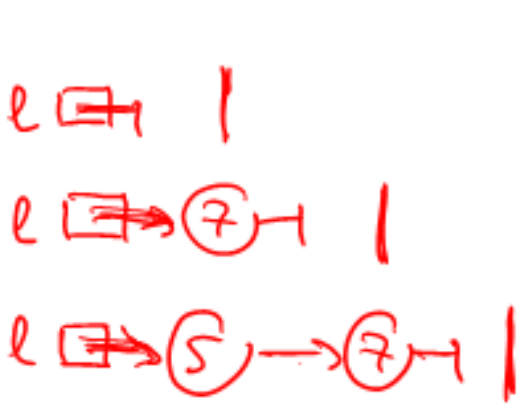
```
...

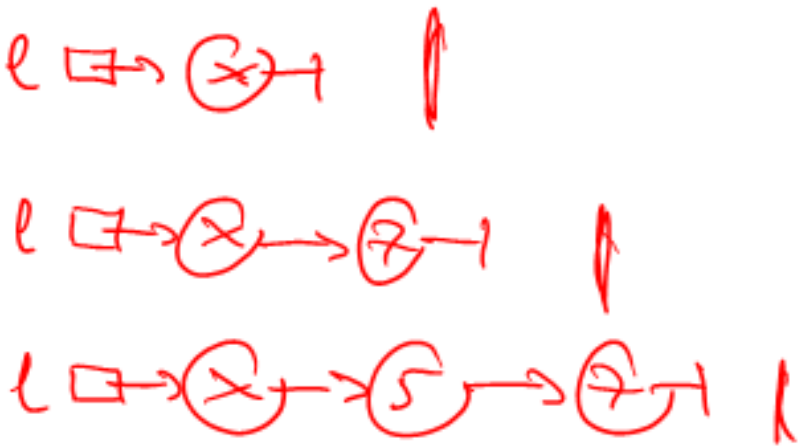
    p = *l;  look ahead and stop before x
    while (p->next != NULL &&
           p->next->val < x) p = p->next;
    q = malloc(sizeof(struct node));
    q->val = x;
    q->next = p->next;
    p->next = q;
  }
}
```

EXERCISE SL05-7/1

insert(s1, 5)

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: bad because root changes

sentinel: good because root never changes

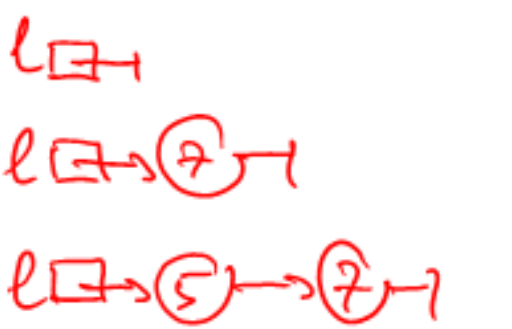
l changes

l does not change

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.

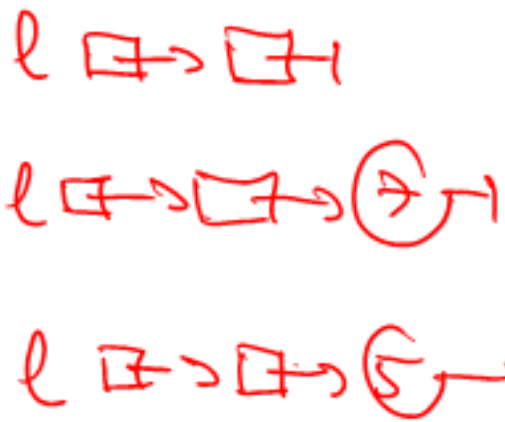
no pointer to a pointer:



l changes

slist node * l;

pointer to a pointer:



l never changes

slist node ** l

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).

1. modified object (the copy) is not returned; bad because changes are lost

void insert(struct node* l, int x) { .... l = ... }

2. modified object is returned and replaces the old (outdated) object

struct node* insert(struct node* l, int x) { ....

call: r = insert(r, 5)

l = ...; .... return l;}

EXERCISE SL05-8/1

Write a C procedure that can be used to increment an integer value that is passed as a parameter.

```
void inc (int i) {
  i++; }

int x;
inc(x);
```

does not change x

Solution: pass a pointer to x to the procedure

```
void inc(int* i) {*i = *i+1;}

int x;
x=7;
inc(&x);
```

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.