Introduction to
Data Structures and Algorithms

Lecture with exercises (2+2)

URL: http://www7.informatik.uni-erlangen.de/~klehmet/teaching/SoSem/dsa/DSA_Script

Ulrich Klehmet

Email: klehmet@informatik.uni-erlangen.de

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Contents (1)

- Introduction and motivation
- Calculating Fibonacci numbers
  - recursive algorithm, iterative algorithm, iterative squaring
- Growth of functions --- asymptotic notation
- Sorting
  - insertion sort, merge sort, heapsort, quicksort
- Elementary data structures
  - stack, queue, linked list, tree

Contents (2)

- Hash tables
  - direct addressing, hashing, chaining, open addressing
- Binary search trees
  - definition, tree walks, querying, insertion, deletion, expected height
- Red-black trees
  - definition, balancedness, rotations, insertion, (deletion)
- Graph algorithms
  - representation of graphs, breadth-first search, depth-first search,

Introduction

- "Data Structures and Algorithms"
  - What is a Data Structure?
  - What is an Algorithm?
  - What does the combination of Data Structures and Algorithms mean?
  - How can we judge how useful a certain combination of Data Structures and Algorithms is?

Introduction

- A Data Structure is
  - is the method to store and organize data
    to facilitate access and modifications
  - the type of data
    - e.g. “stack”, “queue”, “tree”
  - the construction of complex domains
    using elementary domains
    - e.g. arrays, records, unions, sets,
      functions of elements of simple type
    - and arbitrary repetitions of such construction steps

Introduction

- Informally: (Cormen et al.)

  An *algorithm* is any well-defined computational procedure that takes some value (set of values), as *input* and produces some value (set of values) as *output*

- An algorithm is thus a sequence of *computational steps* that transform the input into the output

- An algorithm *must halt* after a final number of steps or time

- An algorithm is *correct* if, for every input instance, it halts with the correct output

Introduction

- An Algorithm
  - is a procedure for processing,
    that is formulated so precisely that it may be performed by
    a mechanical or electronic device
  - must be formulated so exactly that the sequence of the processing steps is completely clear
  - has to terminate
  - has well-defined semantics

- Typical examples for algorithms
  are computer programs
  written in a formal programming language

Introduction

- What does the combination of Data Structures and Algorithms mean?

  ⇨ “Algorithms + Data Structures = Programs”

  (This is the title of a book
  of the famous Swiss researcher Niklaus Wirth, well known
  as the inventor of the programming language “Pascal”)

- Good programs employ a “well suited combination” of Data Structures and Algorithms

Introduction

- How can we judge how useful a certain combination of Data Structures and Algorithms is?
  - We have to evaluate the <u>effort</u> that arises from performing a computation using this “certain combination of Data Structures and Algorithms”
  - This effort may be measured by
    - memory space used
    - cpu time used
    - or other suitable measures

Introduction to
Data Structures and Algorithms

Chapter: **Introduction and motivation**

- **Pseudocode for algorithms**

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Pseudocode for algorithms

- Ways of formulating Algorithms
  - Computer languages
    (→ intention: to be run on computers)
    - C
    - JAVA
    - Matlab
    - Basic
    - …
  - Pseudo code
    (→ intention: to describe algorithms on a high level, to be understood by human beings)

  - Remark: In both cases we have well-defined semantics!

Pseudocode for algorithms

- Example of algorithm in Pseudo code

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Pseudocode for algorithms

Rules for Pseudo code (1)

- Indentation indicates block structure
- Looping constructs (while, for, repeat) and conditional constructs (if, then, else) have interpretation similar to Pascal
  - Difference: the loop-counter of for-loops remains valid after exiting the loop
- Symbol ▷ or % indicates a comment
- Multiple assignment $k \leftarrow j \leftarrow e$ is equivalent to $j \leftarrow e$ and then $k \leftarrow j$

Pseudocode for algorithms

- Rules for Pseudo code (2)
  - Variables (such as *i*, *j*, and *key*) are local to the given procedure
  - Array elements are accessed by specifying the array name followed by the index in square brackets (e.g. *A[i]*)
    - *A[i..j]* indicates a range of values within an array (e.g. *A[1..n]* = *A[1]*, *A[2]*, …, *A[n]*)
  - Objects (= compound data) consist of fields or components: *abc[C]* is field *abc* of an object *C*.

Pseudocode for algorithms

- Rules for Pseudo code (3)
  - An array is treated as an object with field *length*. *length[A]* = number of elements of array *A*
  - A variable representing an array or object is treated as a pointer to the data representing the array or object.
  - *NIL* is the pointer that refers to no object at all
  - Parameters are passed by value: the called procedure receives a copy of its parameters, that are treated as local variables of the procedure

Pseudocode for algorithms

- Rules for Pseudo code (4)
  - The boolean operators “and” and “or” are “<u>short circuiting</u>”:
    - In an expression “*x* and *y*”, *x* is evaluated first
    - If x is FALSE the expression is FALSE, and y is not evaluated at all
    - In an expression “*x* or *y*”, *x* is evaluated first
    - If x is TRUE the expression is TRUE, and y is not evaluated at all
  - This allows writing of expressions e.g. as: “$x \neq NIL$ **and** $f[x] = y$”

Introduction to
Data Structures and Algorithms

Chapter: **Introduction and motivation**

- **Starting examples**

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl Informatik 7 (Prof. Dr.-Ing. Reinhard German)
Martensstraße 3, 91058 Erlangen

Starting examples

- The "sorting problem"
  - Input:

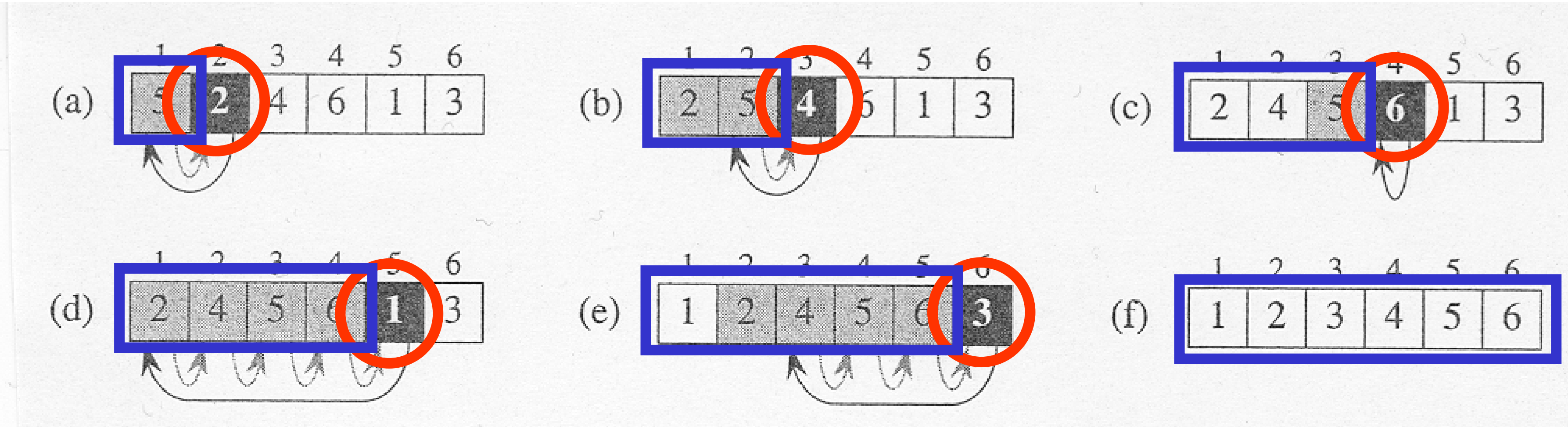
    A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

  - Output:

    A permutation (reordering) $(a_1', a_2', \ldots, a_n')$
    of the input sequence
    such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Starting examples

- Insertion sort



- Insertion sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    do key ← A[j]
       ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
       i ← j − 1
       while i > 0 and A[i] > key
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← key
```

Insertion sort

- Be t_j = number of times the while loop is executed for value j

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** $length[A]$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

Starting examples

- Insertion sort
  - “Running time in general”

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Running time = number of primitive operations or steps

Starting examples

- Insertion sort
  - Best case: “already sorted”
    (t_j = 1 for j = 2, …, n)

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

⇨ linear effort w.r.t. input parameter n

$$T(n) = a \cdot n + b; \quad a,b \in \mathbb{R}$$

Starting examples

- Insertion sort
  - Worst case: “sorted in reversed order” ($t_j = j$ for j = 2, …, n)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

⇨ Worst case running time is a quadratic function of *n*

Starting examples

Principle of recursion

<u>Expl:</u> **Computation of n! (n_factorial):** **$n! = n\,(n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n \cdot (n-1)!$**

```
fact(n)
  if n = 0
     then  n_factorial := 1
     else   n_factorial := n · fact(n –1)
```

fact(4) = 4· fact(3) = 4 · 6 = 24

fact(3) = 3· fact(2) = 3 · 2 = 6

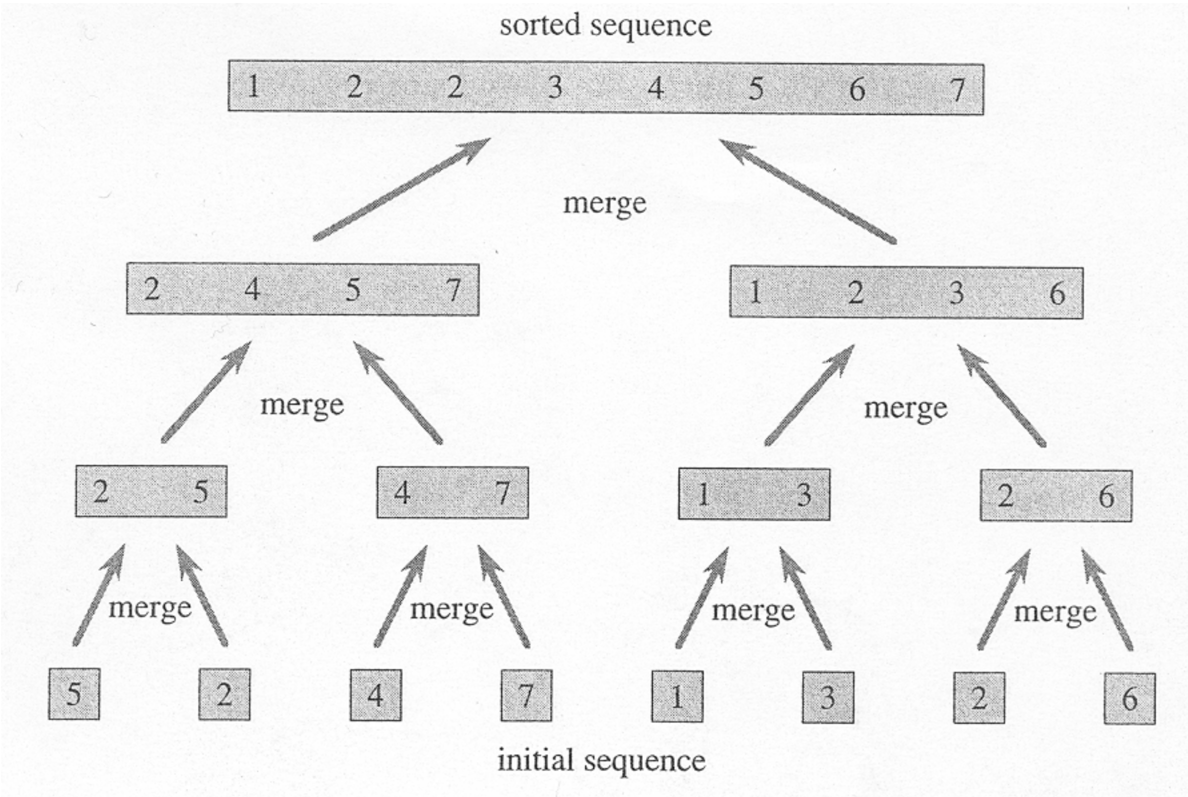
fact(2) = 2· fact(1) = 2· 1 = 2

fact(1) = 1· fact(0) = 1· 1 = 1

fact(0) = 1

Starting examples

- An example of a “recursive algorithm”: Merge sort



Starting examples

- Merge sort

MERGE-SORT(A, p, r)

if $p < r$

  then $q \leftarrow \lfloor (p + r)/2 \rfloor$

    MERGE-SORT(A, p, q)

    MERGE-SORT$(A, q + 1, r)$

    MERGE(A, p, q, r)