Data Structures and Algorithms

Werner Nutt

Acknowledgments

- The course follows the book "Introduction to Algorithms", by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed on these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for his course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Chapter 1: Overview

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

DSA, Chapter 1:

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

Learning Outcomes

The main things we will learn in this course:

- To *distinguish* between a *problem* and an *algorithm* that solves it
- To get to know a *toolbox* of *classical* algorithms
- To *think algorithmically* and get the spirit of how algorithms are designed
- To learn a number of algorithm design *techniques* (such as divide-and-conquer)
- To analyze (in a precise and formal way) the *efficiency* and the *correctness* of algorithms.

Syllabus

1. Introduction, recursion (chap 1 in CLRS)
2. Correctness and complexity of algorithms (2, 3)
3. Divide and conquer, recurrences (4)
4. Heapsort, Quicksort (6, 7)
5. Dynamic data structures, abstract data types (10)
6. Binary search trees, red-black trees (12, 13)
7. Hash tables (11)
8. Dynamic programming (15)
9. Graphs: Principles and graph traversal (22)
10. Minimum spanning tree and shortest path (23, 24)

Literature

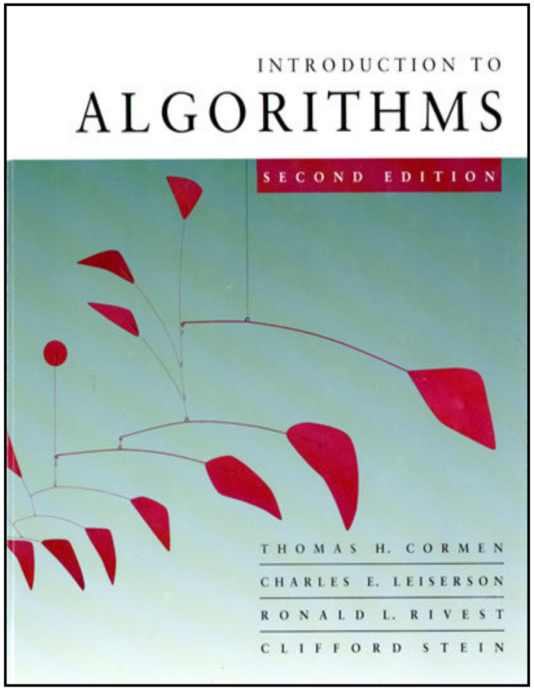
Cormen, Leiserson, Rivest and Stein (CLRS), *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, 2001

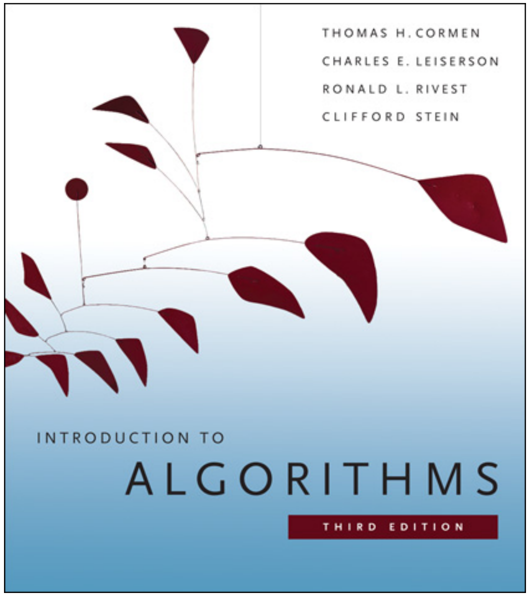
and

Third Edition, MIT Press, 2009

(See http://mitpress.mit.edu/algorithms/)

Course is based on this book





Other Literature

Kurt Mehlhorn and Peter Sanders

Algorithms and Data Structures - The Basic Toolbox

Offers alternate presentation of topics of the course

Free download from

http://www.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf

Course Organization

- Lectures: Wed 10:45-12:45, Fri 8:30-10:30
  Office hours: Wed 14:00-16:00 (but let me know if you want to come)
- Labs (starting next week): Tue 16:00-18:00
- Teaching Assistants
  - Radityo Eko Prasoyo, RPrasojo@unibz.it
  - Rafael Penaloza, Rafael.Penaloza@unibz.it
  - Guohui Xiao, guohui.xiao@unibz.it
- Home page:
  http://www.inf.unibz.it/~nutt/Teaching/DSA1516/

Assignments

The assignments are a crucial part of the course

- Each week an assignment has to be solved
- The schedule for the publication and the handing in of the assignments will be announced at the next lecture.
- A number of assignments include programming tasks. It is strongly recommended that you implement and run all programming exercises.
- Assignments will be marked. The assignment mark will count towards the course mark.
- Any attempt at plagiarism (copying from the web or copying from other students) leads to a 0 mark for all assignments.

Assignments, Midterm Exam, Final Exam, and Course Mark

- There will be
  - one written exam at the end of the course
  - one midterm exam around the middle of the course
  - assignments
- To pass the course, one has to pass the written exam.
- Students who do not submit exercises and do not take part in the midterm (or fail the midterm) will be marked on the final exam alone.
- For students who submit all assignments, and take part in the midterm, the final mark will be a weighted average

  40% exam mark + 20% midterm
  + 40% assignment mark

Assignments, Midterm Exam, Final Exam, and Course Mark

- If students submit fewer assignments, or do not take part in the midterm, the percentage will be lower.
- Assignments for which the mark is lower than the mark of the written exam will not be considered.
- Similarly, the midterm will not be considered if the mark is lower than the mark of the final exam.
- The midterm and assignment marks apply to three exam sessions.

Organisation of Labs

- You will attend always the lab of the same teaching assistant (TA) during the course
- The TA will mark your assignments and be your first contact person for questions on the assignments
- To help us organize the labs, send an email to Rafael.Penaloza@unibz.it containing a group of students that would like to attend the same lab
- One mail per group is enough
- The mail should contain for each student of the group
  - name
  - email address
  - student number

General Remarks

- Algorithms are first designed on paper
  … and later keyed in on the computer.

- The most important thing is to be simple and precise .

- During lectures:
  - Interaction is welcome; ask questions
    (I will ask you anyway ☺)
  - Additional explanations and examples if desired
  - Speed up/slow down the progress

DSA, Chapter 1:

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

What are Algorithms About?

There are problems we solve in everyday life

- Travel from Bolzano to Berlin
- Cook Spaghetti alla Bolognese *(I know, not in Italy,…)*
- Register for a Bachelor thesis at FUB

For all these problems, there are

- instructions
- recipes
- procedures,

which describe a complex operation in terms of

- elementary operations *("beat well …")*
- control structures and conditions *("… until fluffy")*

Algorithms

Problems involving numbers, strings, mathematical objects:

- for two numbers, determine their sum, product, …
- for two numbers, compute their greatest common divisor
- for a sequence of strings,
  find an alphabetically sorted permutation of the sequence
- for two arithmetic expressions, find out if they are equivalent
- for a program in Java,
  create an equivalent program in byte code
- on a map, find for a given house the closest bus stop

We call instructions, recipes, for such problems *algorithms*

What have algorithms in common with recipes?
How are they different?

History

- *First algorithm:* Euclidean Algorithm,
  greatest common divisor, 400-300 B.C.
- *Name:* Persian mathematician Mohammed al-Khowarizmi, in Latin became “Algorismus”
  كتاب الجمع و التفريق بحساب الهند
  Kitāb al-Dscham' wa-l-tafrīq bi-ḥisāb al-Hind =
  = Book on connecting and taking apart in the calculation of India
- *19th century*
  - Charles Babbage: Difference and Analytical Engine
- *20th century*
  - Alan Turing, Alonzo Church: formal models computation
  - John von Neumann: architecture of modern computers

Data Structures, Algorithms, and Programs

- Data structure
  - Organization of data to solve the problem at hand
- Algorithm
  - Outline, the essence of a computational procedure, step-by-step instructions
- Program
  - implementation of an algorithm in some programming language

Overall Picture

Using a computer to help solve problems:

- Precisely specifying the problem
- Designing programs
  - architecture
  - algorithms
- Writing programs
- Verifying (testing) programs

Data Structure and Algorithm Design Goals

Correctness

Efficiency

Implementation Goals

Robustness

Reusability

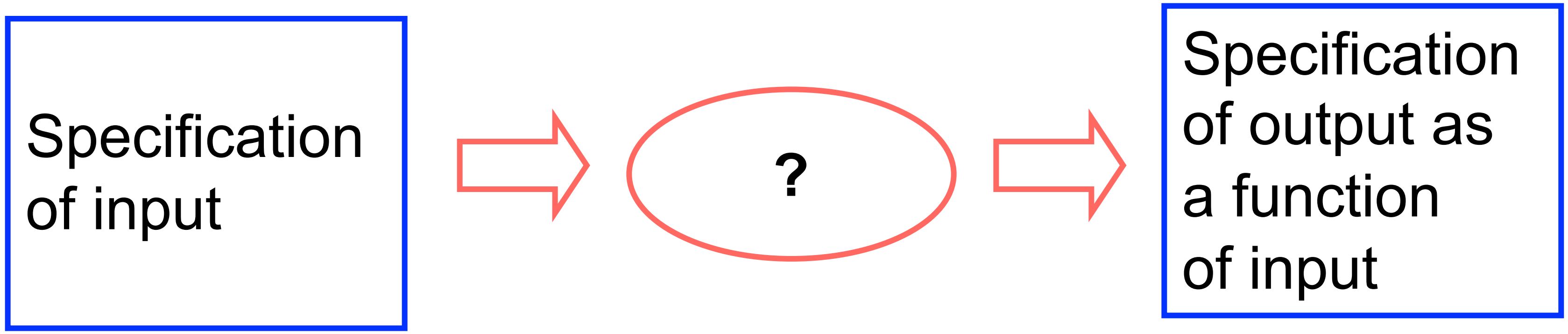
Adaptability

This course is not about:

- Programming languages
- Computer architecture
- Software architecture
- SW design and implementation principles

We will only touch upon
the theory of complexity
and computability.

Algorithmic Problem



There is an infinite number of possible input *instances* satisfying the specification.

For example: An array of distinct integer numbers:

[-909, -1, -20, 908, 1000000000, 100000]

Question from a Google Interview

You are given an array of distinct numbers.

You need to return an index to a "local minimum" element, which is defined as an element that is smaller than both its adjacent elements.

In the case of the array edges, the condition is reduced to one adjacent element.

If there are multiple "local minima",
returning any one of them is fine.

Question from a Google Interview/2

You are given an unsorted sequence of integers *A*.

Find the longest subsequence *B* such that
elements of this subsequence are strictly increasing numbers.

Elements in the subsequence *B* must appear
in the same relative order as in the sequence *A*.

Example:

input: *A* = [-1, 2, 100, 100, 101, 3, 4, 5, -7]

output: *B* = [-1, 2, 3, 4, 5]

Question from a Google Interview/3

You have a sorted array containing
the age of every person on Earth

[0, 0, 0, 0, ..., 1, 1, ..., 28, 28, ..., 110, ...].

Find out how many people have each age.

Question from a Google Interview/4

You are given a text file that has list of dependencies between (any) two projects in a source code repository.

Write an algorithm to determine the build order, i.e., which project needs to be built first, followed by which project, based on the dependencies.

You get a bonus point, if you can detect any circular dependencies and throw an exception if found.

Example: `ProjectDependencies.txt`

a → *b* (means "*a* depends on *b*", so *b* needs to be built first and then *a*)

b → *c*

b → *d*

c → *d*

Then the build order can be *d*, *c*, *b*, *a*, in that order.

Algorithmic Solution

Input instance adhering to specification → Algorithm → Output related to the input as required

- Algorithm describes actions on the input instance
- There may be many correct algorithms for the same algorithmic problem.

Definition

An algorithm is a sequence of *unambiguous* instructions for solving a problem, i.e.,

- for obtaining a *required output*
- for any *legitimate input*

in a finite amount of time.

➔ This presumes a mechanism to execute the algorithm

Properties of algorithms:

- Termination, Correctness, (Non-)Determinism, RunningTime, …

How to Develop an Algorithm

- Precisely define the problem.
  Precisely specify the input and output.
  Consider all cases.

- Come up with a simple (= abstract ?) plan
  to solve the problem at hand.
  - The plan is independent of a (programming) language
  - The precise problem specification influences the plan.

- Turn the plan into an implementation
  - The problem representation (data structure) influences the implementation

Example 1: Searching

INPUT

- A - (un)sorted sequence/array of *n* numbers, ($n > 0$)
- q - a single number

$a_1, a_2, a_3, \ldots, a_n;\ q$ →

2 5 6 10 11; 5

2 5 6 10 11; 9

OUTPUT

- index of number q in sequence/array A, or *-1*

→ j

2

-1

Searching/2, search1

search1
INPUT: A[1..n] (un)sorted array of integers, *q* an integer.
OUTPUT: index *j* such that A[*j*]=*q* or *-1* if A[*j*] ≠ *q* for all *j* ($1 \le j \le n$)

```
j := 1
while j ≤ n and A[j] ≠ q do
   j++
if j ≤ n
   then return j
   else return -1
```

- The code is written in *pseudo-code* and INPUT and OUTPUT of the algorithm are specified.
- The algorithm uses a *brute-force* technique, i.e., scans the input sequentially.

Preconditions, Postconditions

Precondition:

- what does the algorithm get as input?

Postcondition:

- what does the algorithm produce as output?
- … how does this relate to the input?

Make sure you have considered the special cases:

- empty set, number 0, empty reference *NULL*, …

Pseudo-code

Like Java, Pascal, C, or any other imperative language

- Control structures:

  (if then else, while, and for loops)

- Assignment: :=
- Array element access: A[i]
- Access to element of composite type (record or object):

  A.b

CLRS uses b[A]

Control Structures in Pseudo-code: Examples

- for-to:

```
for i := 1 to n do
    A[i] := A[i] + 1
```

- for-downto:

```
for i := n downto 1 do
    A[i] := A[i] + 2
```

- while:

```
while A[i] > 0 do
    A[i] := A[i] - i
```

- if-then-else:

```
if A[i] > 0
   then pos := true
   else pos := false
```

Searching, Java Solution

```
import  java.io.*;

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

  public static void main(String args[]) {
    j = 0; q = 22;
    while (j < n && a[j] != q) { j++; }
    if (j < n) { System.out.println(j); }
    else { System.out.println(-1); }
  }
}
```

Searching, Java Solution

```
import  java.io.*;

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

  public static void main(String args[]) {
    int j = 0;
    while (j < n && a[j] != q) { j++; }
    if (j < n) { System.out.println(j); }
    else { System.out.println(-1); }
  }
}
```

Searching/3, search2

Another idea:

Run through the array
and set a pointer if the value is found.

search2
INPUT: A[1..n] (un)sorted array of integers, *q* an integer.
OUTPUT: index *j* such that A[*j*]=*q* or *-1* if A[*j*] ≠ *q* for all *j* ($1 \le j \le n$)

```
ptr := -1;
for j := 1 to n do
  if a[j] = q then ptr := j
return ptr;
```

Does it work?

search1 vs search2

Are the solutions equivalent?

Can one construct an example such that, say,

- search1 returns 3
- search2 returns 7 ?

But both solutions satisfy the specification (or don't they?)

Searching/4, search3

A third idea:

Run through the array and
return the index of the value in the array.

search3
INPUT: A[1..n] (un)sorted array of integers, *q* an integer.
OUTPUT: index *j* such that A[*j*]=*q* or *-1* if A[*j*] $\neq$ *q* for all *j* ($1 \leq j \leq n$)

```
for j := 1 to n do
  if a[j] = q then return j
return -1
```

Comparison of Solutions

Metaphor: shopping behavior when buying a beer:

- **`search1`**: scan products;
  stop as soon as a beer is found and go to the exit.
- **`search2`**: scan products until you get to the exit;
  if during the process you find a beer,
  put it into the basket
  (instead of the previous one, if any).
- **`search3`**: scan products;
  stop as soon as a beer is found
  and exit through next window.

Comparison of Solutions/2

- search1 and search3 return *the same result* (index of the first occurrence of the search value)
- search2 returns the index of the last occurrence of the search value
- search3 does not finish the loop
  (as a general rule, you better avoid this)

Beware: Array Indexes in Java/C/C++

- In pseudo-code, array indexes range from 1 to length
- In Java/C/C++, array indexes range from 0 to length-1
- Examples:
  - Pseudo-code

    ```
    for j := 1 to n do
    ```

    Java:

    ```
    for (j=0; j < a.length; j++) { …
    ```
  - Pseudo-code

    ```
    for j := n downto 2 do
    ```

    Java:

    ```
    for (j=a.length-1; j >= 1; j--) { …
    ```

Suggested Exercises

- Implement the three variants of search (with input and output of arrays)
  - Create random arrays for different lengths
  - Compare the results
  - Add a counter for the number of cycles and return it, compare the result
- Implement them to scan the array in reverse order

DSA, Chapter 1:

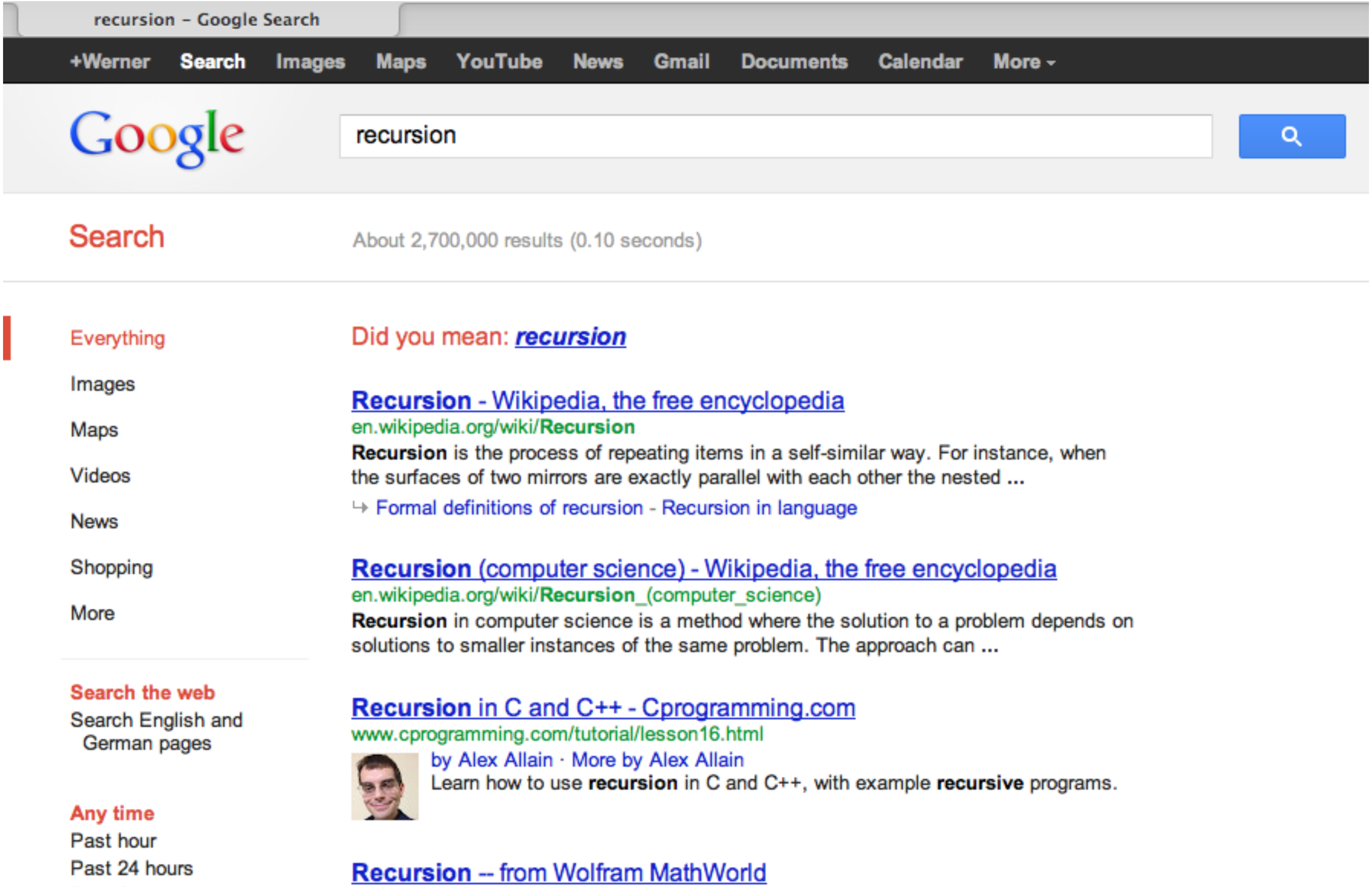
- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

Recursion

An object is recursive if

- a part of the object refers to the entire object, or
- one part refers to another part and vice versa

(mutual recursion)


recursion – Google Search
+Werner Search Images Maps YouTube News Gmail Documents Calendar More
Google
recursion
Search
About 2,700,000 results (0.10 seconds)
Everything
Images
Maps
Videos
News
Shopping
More
Search the web
Search English and German pages
Any time
Past hour
Past 24 hours
Did you mean: recursion
Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...
Formal definitions of recursion - Recursion in language
Recursion (computer science) - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion_(computer_science)
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. The approach can ...
Recursion in C and C++ - Cprogramming.com
www.cprogramming.com/tutorial/lesson16.html
by Alex Allain · More by Alex Allain
Learn how to use recursion in C and C++, with example recursive programs.
Recursion -- from Wolfram MathWorld

Source: http://bluehawk.monmouth.edu/~rclayton/web-pages/s11-503/recursion.jpg

Recursion/2

- A recursive definition: a concept is defined by referring to itself.

  E.g., arithmetical expressions (like (3 * 7) - (9 / 3)):

  EXPR := VALUE | (EXPR OPERATOR EXPR)

- A recursive procedure: a procedure that calls itself

  Classical example: factorial, that is n! = 1 * 2 * 3 *...* n

  n! = n * (n-1)!

  … or is there something missing?

The Factorial Function

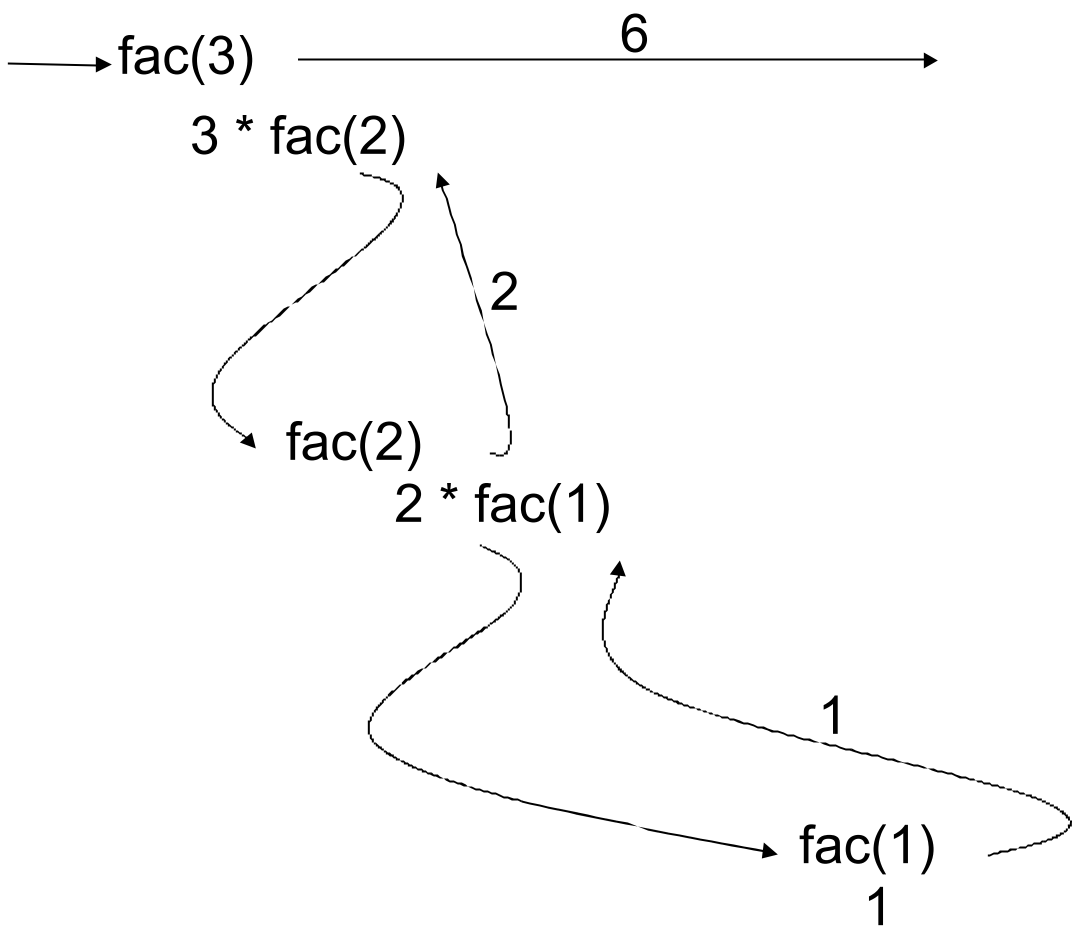
Pseudocode of factorial:

```
fac1
INPUT: n – a natural number.
OUTPUT: n! (factorial of n)

fac1(n)
  if n < 2 then return 1
  else return n * fac1(n-1)
```

This is a recursive procedure. A recursive procedure has

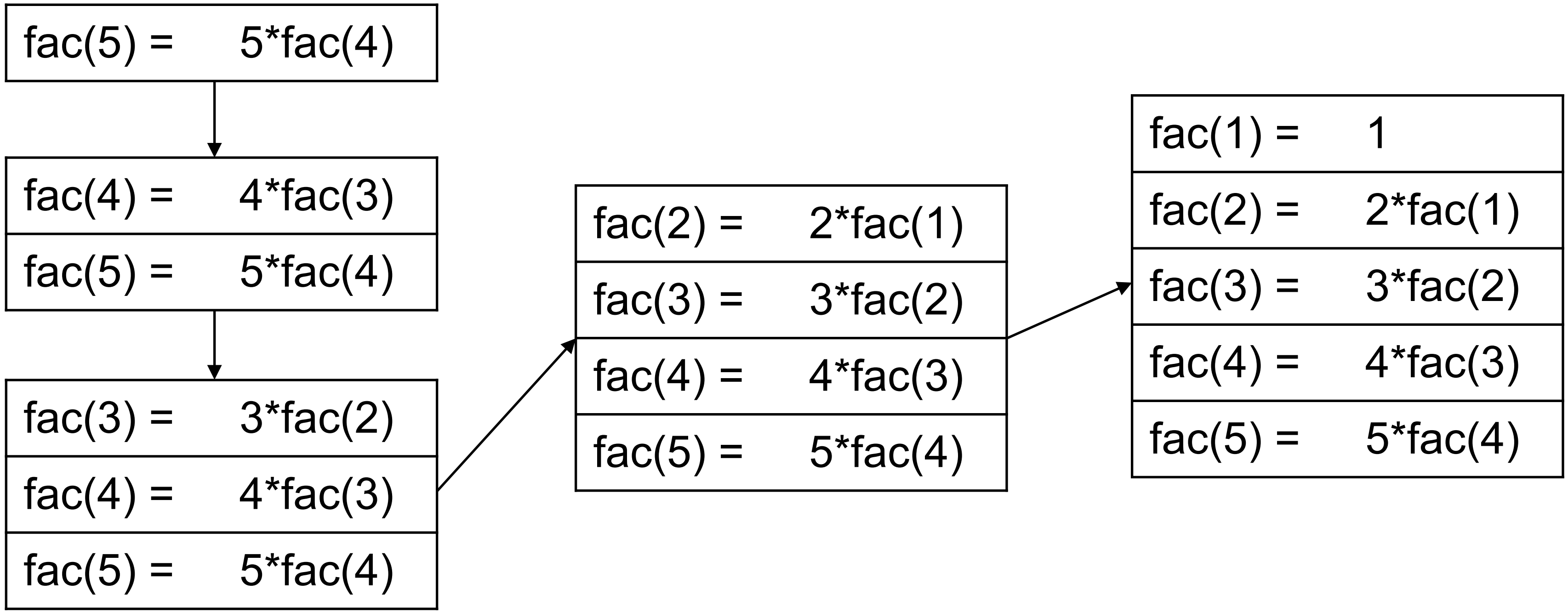
- a termination condition
  (determines when and how to stop the recursion).
- one (or more) recursive calls.

Tracing the Execution



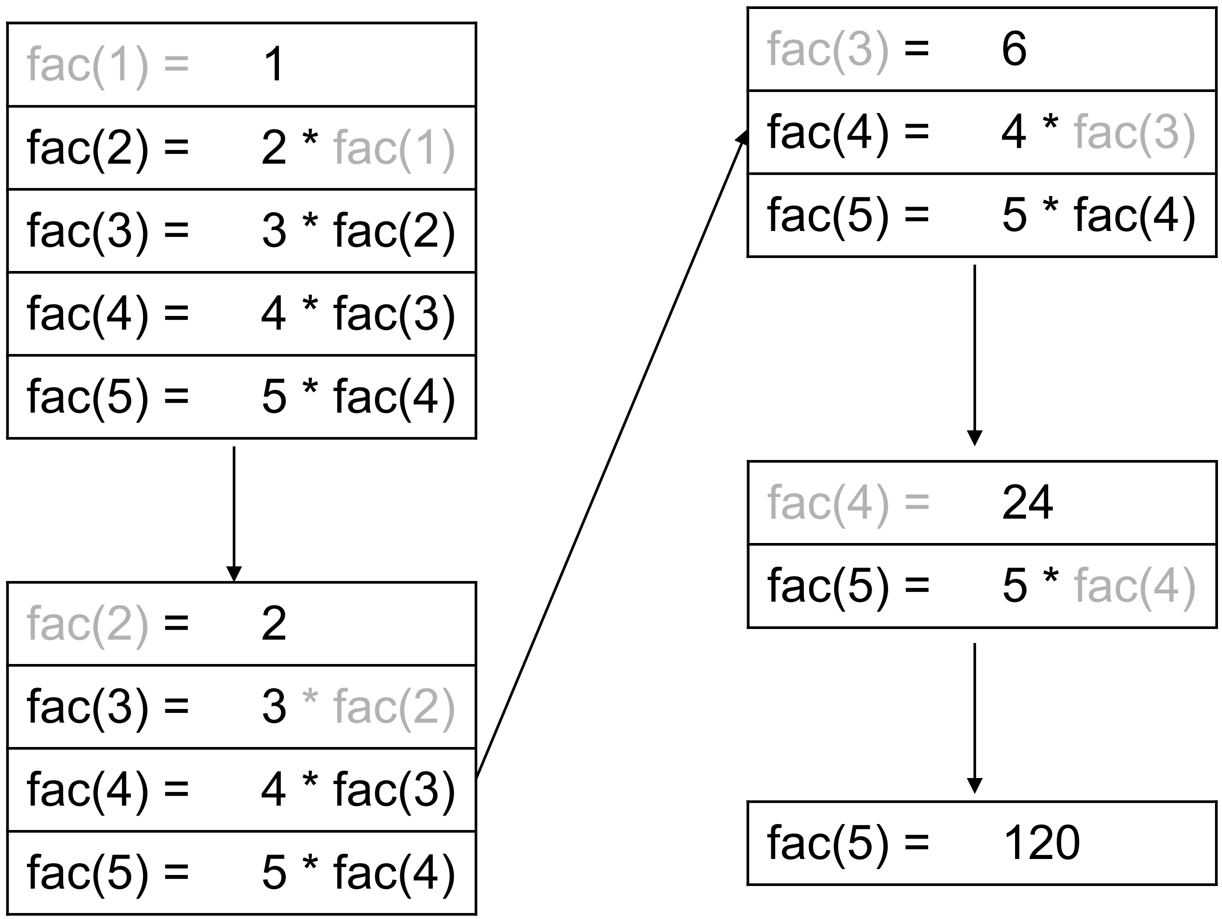
Bookkeeping

The computer maintains an activation stack
for active procedure calls (→ compiler construction).
Example for fac(5). The stack is built up.



Bookkeeping/2

Then the activation stack is reduced



Variants of Factorial

```
fac2
INPUT: n – a natural number.
OUTPUT: n! (factorial of n)

fac2(n)
  if n = 0 then return 1
  return n * fac2(n-1)
```

```
fac3
INPUT: n – a natural number.
OUTPUT: n! (factorial of n)

fac3(n)
  if n = 0 then return 1
  return n * (n-1) * fac3(n-2)
```

Analysis of the Variants

`fac2` is correct

- The return statement in the if clause terminates the function and, thus, the entire recursion.

`fac3` is incorrect

- Infinite recursion.
  The termination condition is never reached if n is odd:

```
fact(3)
= 3*2*fact(1)
= 3*2*1*0*fact(-1)
= ...
```

Variants of Factorial/2

```
fac4
INPUT: n – a natural number.
OUTPUT: n! (factorial of n)

fac4(n)
  if n <= 1 then return 1
  return n*(n-1)*fac4(n-2)
```

```
fac5
INPUT: n – a natural number.
OUTPUT: n! (factorial of n)

fac5(n)
  return n * fac5(n-1)
  if n <= 1 then return 1
```

Analysis of the Variants/2

`fac4` is correct

- The return statement in the if clause terminates the function and, thus, the entire recursion.

`fac5` is incorrect

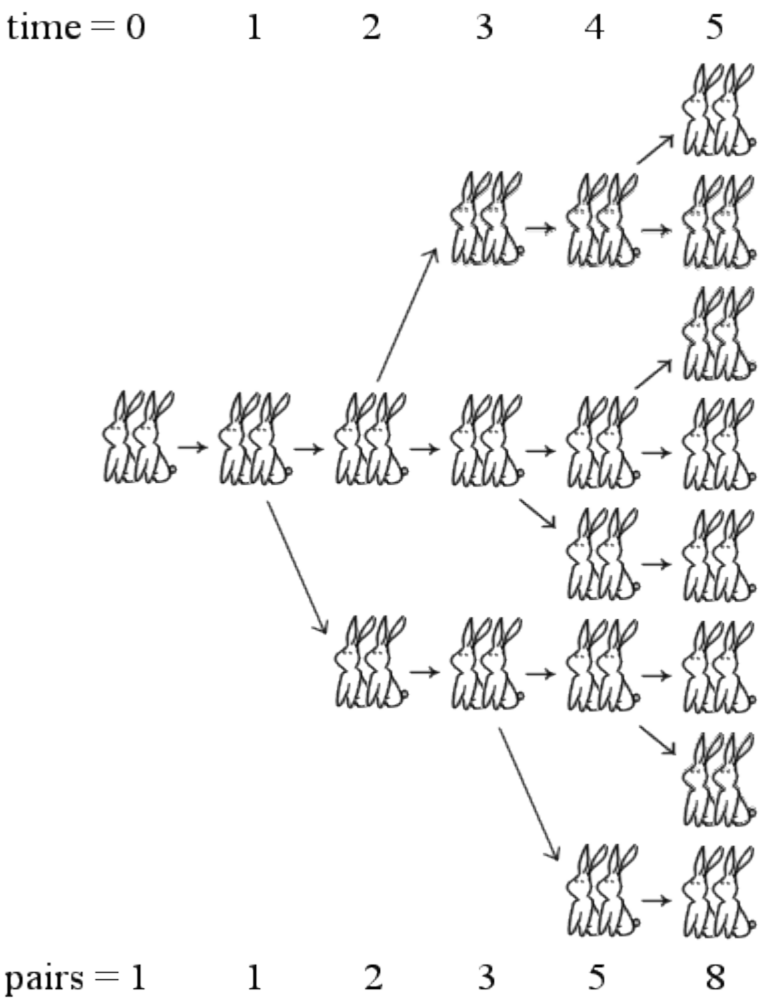
- Infinite recursion.
  The termination condition is never reached.

Counting Rabbits

Someone placed a pair of rabbits
in a certain place,
enclosed on all sides by a wall,
so as to find out
how many pairs of rabbits
will be born there in the course of one year,
it being assumed
that every month
a pair of rabbits produces another pair,
and that rabbits begin to bear
young two months after their own birth.

Leonardo di Pisa ("Fibonacci"),
Liber abacci, 1202

Counting Rabbits/2



Source: http://www.jimloy.com/algebra/fibo.htm

Fibonacci Numbers

Definition

- fib(0) = 1
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2), n>1

Numbers in the series:

1, 1, 2, 3, 5, 8, 13, 21, 34, …

Fibonacci Procedure

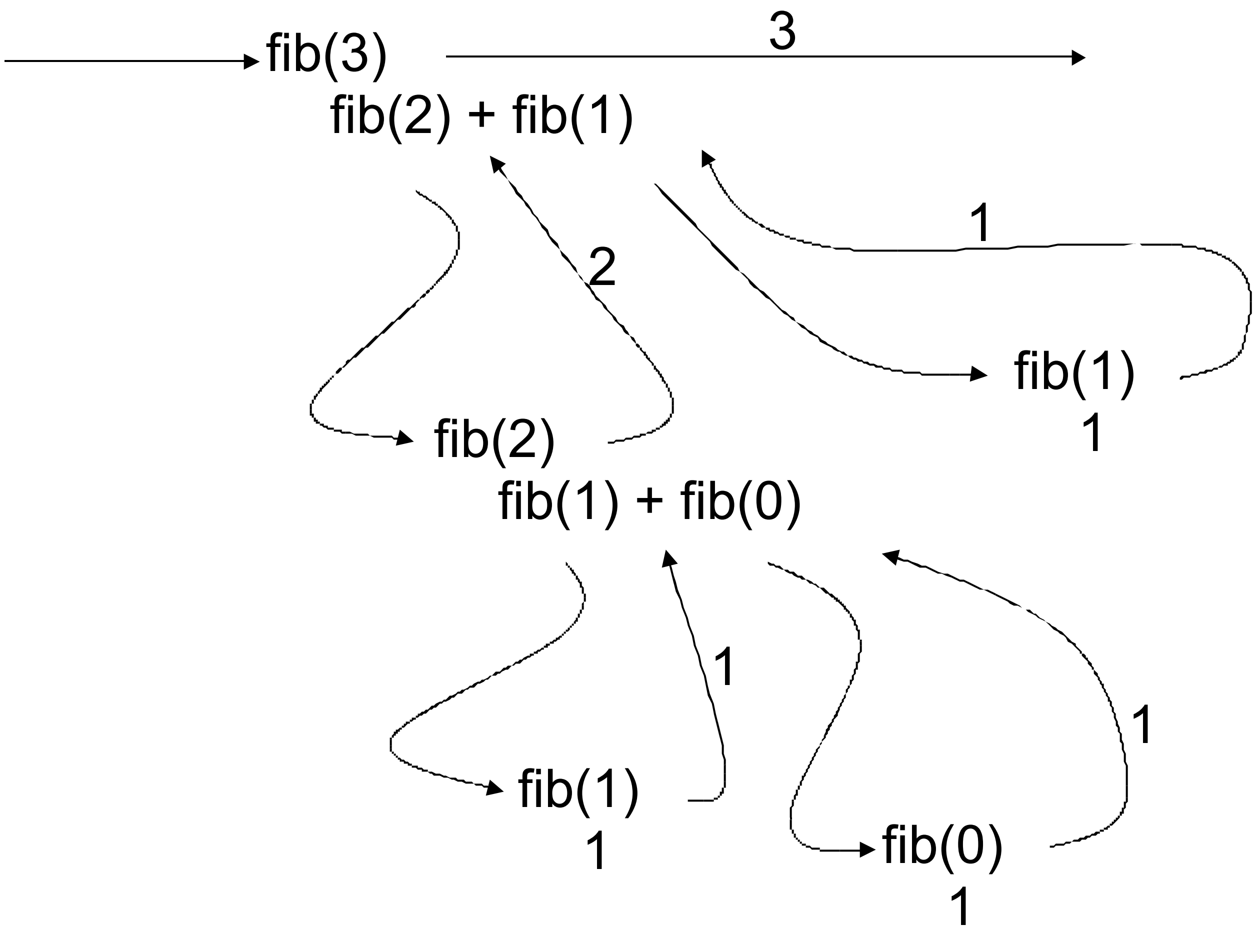
```
fib
INPUT: n – a natural number greater or equal than 0.
OUTPUT: fib(n), the nth Fibonacci number.

fib(n)
  if n ≤ 1 then return 1
  else return fib(n-1) + fib(n-2)
```

A procedure with multiple recursive calls

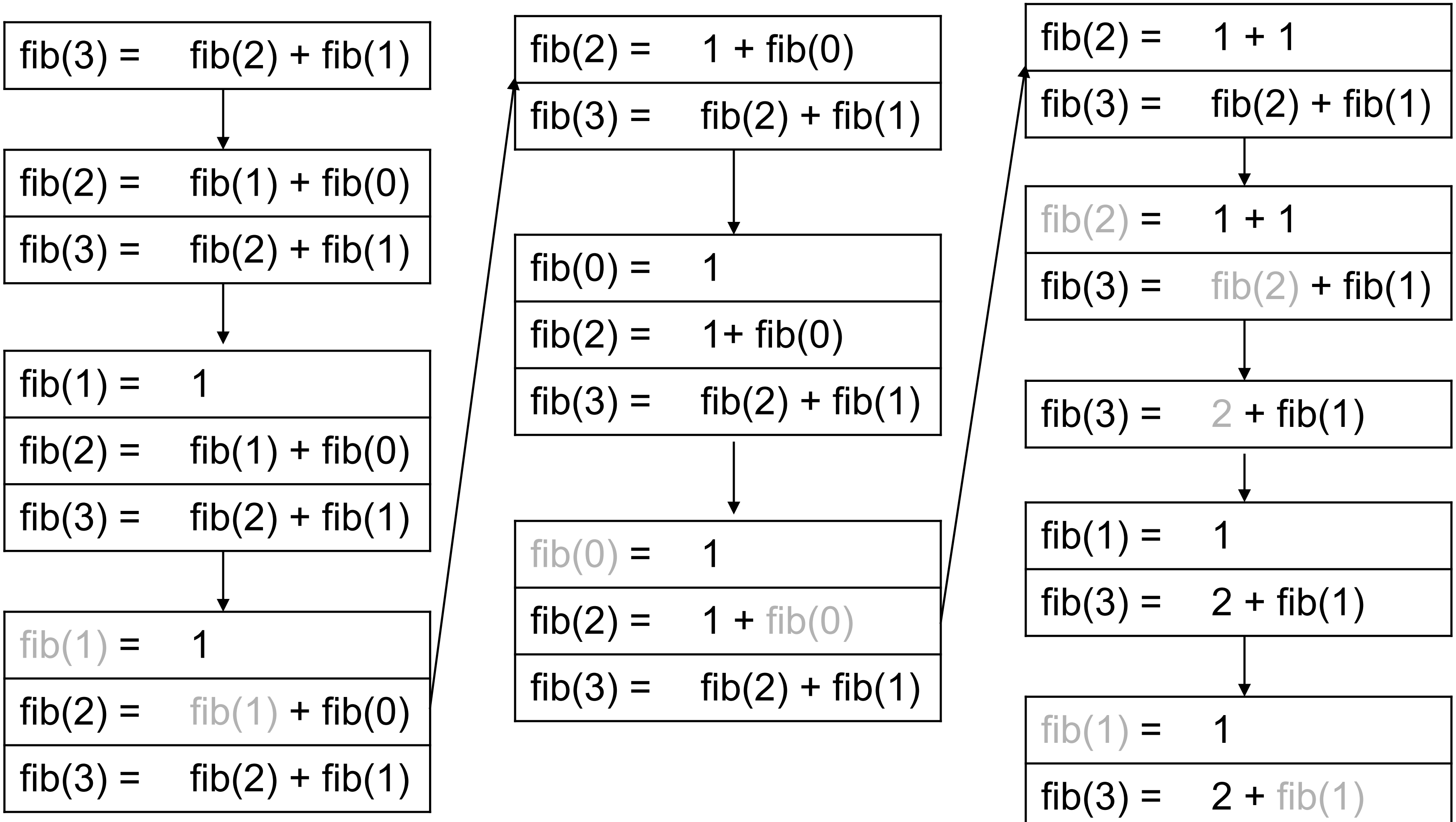
Fibonacci Procedure/2

```
public class fibclassic {

  static int fib(int n) {
    if (n <= 1) {return 1;}
    else {return fib(n - 1) + fib(n - 2);}
  }

  public static void main(String args[]) {
    System.out.println("Fibonacci of 5 is "
                            + fib(5));
  }
}
```

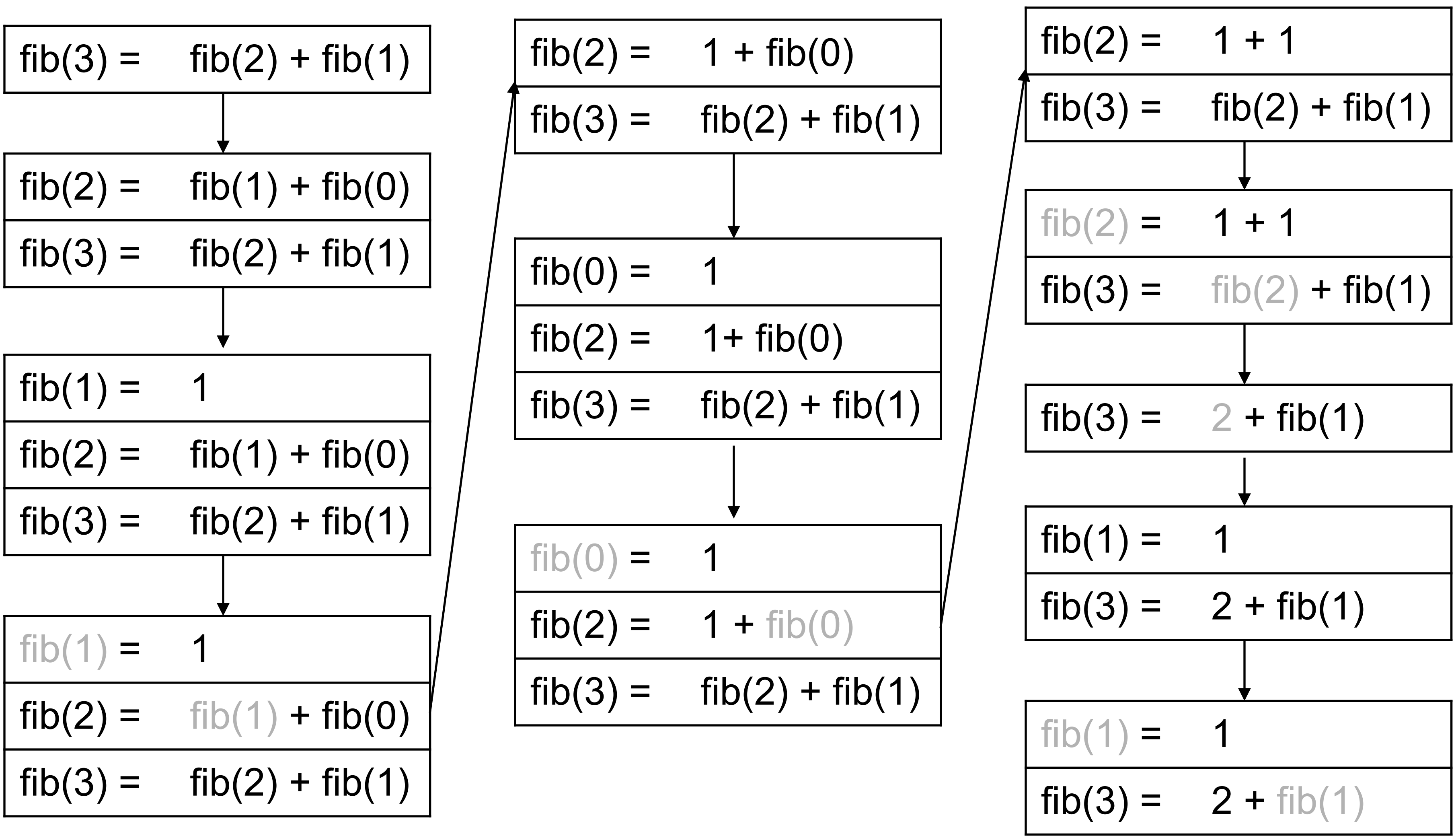
Tracing fib(3)



Bookkeeping



Bookkeeping



Questions

- What is the maximal height of the recursion stack during the computation of *fib(n)*?
- How many recursive calls are made to compute *fib(n)*?
- What does the tree of recursive calls (recursion tree) look like?
- Can we derive a lower bound for the number of calls from that?
- Can there be a procedure for *fib* with fewer operations?
- How is the size of the result *fib(n)* related to the size of the input *n*?

Saving Intermediate Values of fib(n)

fib
INPUT: n – a natural number greater or equal than 0.
OUTPUT: fib(n), the nth Fibonacci number.

```
fib(n)
  if n ≤ 1 then return 1
  int[] fibVal = new int[n+1]
     // let's assume in this case that the array
     // boundaries start with 0 :-(
  fibVal[0] := 1; fibVal[1] := 1;
  for j=2 to n do
     fibVal[j] := fibVal[j-1]+fibVal[j-2]
  return fibVal[n];
```

… but we only need the last two values of fib to compute the next one

Iterative Computation of fib(n)

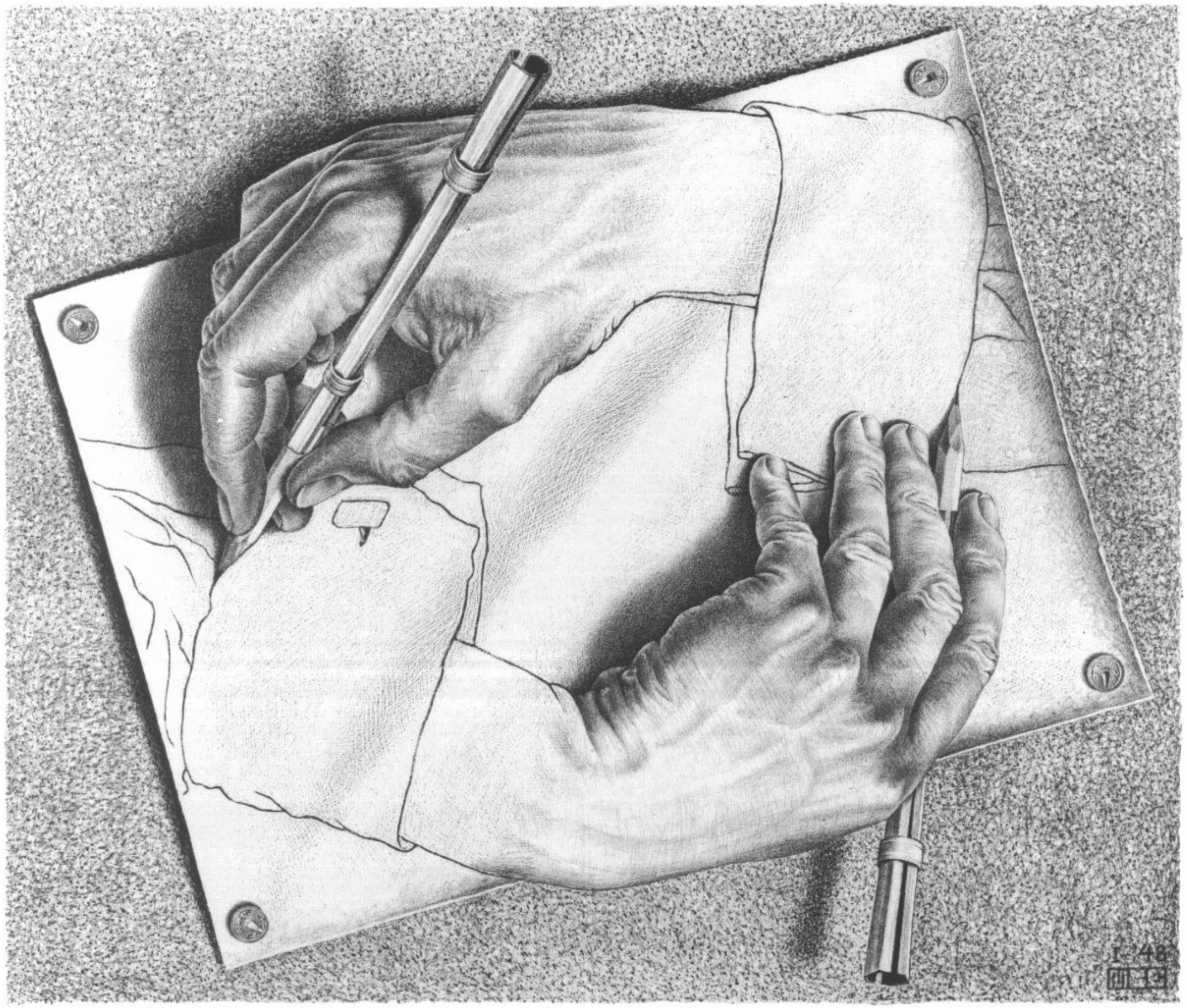
fib

INPUT: n – a natural number greater or equal than 0.

OUTPUT: fib(n), the nth Fibonacci number.

```
fib(n)
  if n ≤ 1 then return 1
  int f, f1, f2;
  f1 := 1; f2 := 1;
  for j=2 to n do
     f  := f1+f2
     f2 := f1
     f1 := f
  return f;
```

Mutual Recursion



Source: http://britton.disted.camosun.bc.ca/escher/drawing_hands.jpg

Mutual Recursion Example

- Problem: Determine whether a natural number is even
- Definition of even:
  - 0 is even
  - n is even if n - 1 is odd
  - n is odd if n - 1 is even

Implementation of `even`

even
INPUT: n – a natural number.
OUTPUT: true if n is even; false otherwise

```
even(n)
  if n = 0 then return TRUE
  else return odd(n-1)

odd(n)
  if n = 0 then return FALSE
  return even(n-1)
```

How can we determine whether n is odd?

Is Recursion Necessary?

- Theory: You can always resort to iteration and explicitly maintain a recursion stack.
- Practice: Recursion is elegant and in some cases the best solution by far.
- In the previous examples recursion was never appropriate since there exist simple iterative solutions.
- Recursion is more expensive than corresponding iterative solutions since bookkeeping is necessary.

Data Structures and Algorithms

Chapter 1.4

Werner Nutt

DSA, Chapter 1:

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (insertion, selection, bubble)

Sorting

- Sorting is a classical and important algorithmic problem.
  - For which operations is sorting needed?
  - Which systems implement sorting?
- We look at sorting arrays
  (in contrast to files, which restrict random access)
- A key constraint are the restrictions on the space:
  in-place sorting algorithms (no extra RAM).
- The run-time comparison is based on
  - the number of comparisons (C) and
  - the number of movements (M).

Sorting

- Simple sorting methods use roughly ***n * n*** comparisons
  - Insertion sort
  - Selection sort
  - Bubble sort

- Fast sorting methods use roughly ***n * log n*** comparisons
  - Merge sort
  - Heap sort
  - Quicksort

What's the point of studying those simple methods?

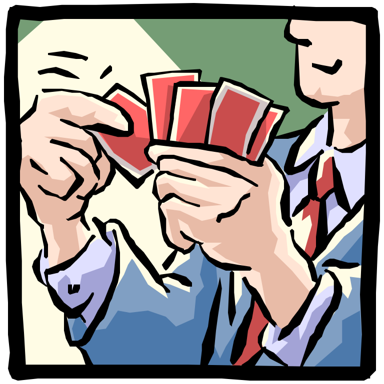
Example 2: Sorting

INPUT

sequence of *n* numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the input sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

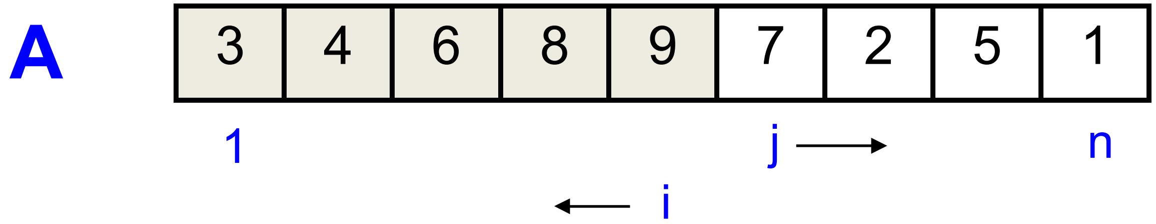
2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 \le b_2 \le b_3 \le \ldots \le b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Insertion Sort



Strategy

- Start with one sorted card.
- Insert an unsorted card at the correct position in the sorted part.
- Continue until all unsorted cards are inserted/sorted.

44	**55**	**12**	**42**	**94**	**18**	**06**	**67**
44	55	**12**	**42**	**94**	**18**	**06**	**67**
12	44	55	**42**	**94**	**18**	**06**	**67**
12	42	44	55	**94**	**18**	**06**	**67**
12	42	44	55	94	**18**	**06**	**67**
12	18	42	44	55	94	**06**	**67**
06	12	18	42	44	55	94	**67**
06	12	18	42	44	55	67	94

Insertion Sort: Principles

- Idea: stepwise, increase sorted part.
  Initially, A[1..1] is sorted
- Control structure: increase stepwise from left to right
  => iteration
- Insertion into sorted part: check until position is found
  => while-loop
  Number to be inserted: key:= A[j]
  Move sorted part to right, until correct position found

Insertion Sort/2

```
INPUT:  A[1..n] - an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

for j := 2 to n do // A[1..j-1] sorted
  key := A[j]; i := j-1;
  while i > 0 and A[i] > key do
    A[i+1] := A[i];  i--;
  A[i+1] := key
```

The number of comparisons during the *j*th iteration is

- at least 1: $C_{min} = \sum_{j=2}^{n} 1 =$
- at most *j*-1: $C_{max} = \sum_{j=2}^{n} j-1 =$

Insertion Sort/2

```
INPUT:  A[1..n] - an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]

for j := 2 to n do // A[1..j-1] sorted
  key := A[j]; i := j-1;
  while i > 0 and A[i] > key do
    A[i+1] := A[i];  i--;
  A[i+1] := key
```

The number of comparisons during the *j*th iteration is

– at least 1: $C_{min} = \sum_{j=2}^{n} 1 = n - 1$

– at most *j*-1: $C_{max} = \sum_{j=2}^{n} j-1 = (n*n - n)/2$

Insertion Sort/3

- The number of comparisons during the jth iteration is:
  - j/2 on average: $C_{avg} = \sum_{j=2}^{n} j/2$ = (n*n + n – 2)/4

- The number of movements M_i is $(C_i-1)+2 = C_i+1$:
  - $M_{min} = \sum_{j=2}^{n} 2$ = 2*(n-1),
  - $M_{avg} = \sum_{j=2}^{n} j/2+1$ = (n*n + 5n - 6)/4
  - $M_{max} = \sum_{j=2}^{n} j$ = (n*n +n - 2)/2

Ideas of Insertion Sort

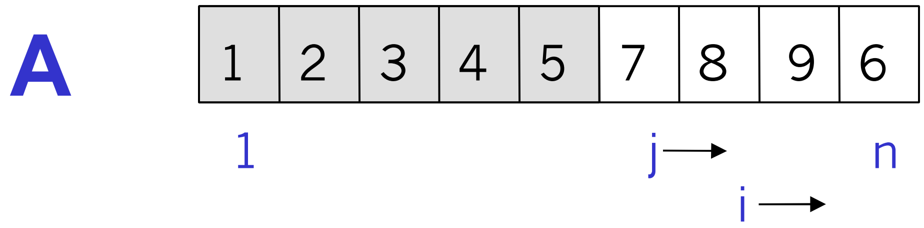
- Start with something that is a trivial partial solution
  - what is the initial (trivial) partial solution?
  - what could be another trivial partial solution?

- Stepwise extend each partial solution to a bigger partial solution
  … until it is full solution
  - in which way are the results of each (outer) iteration partial solutions?

Loop Invariants

- Which property (in terms of A and j) is true whenever the execution reaches the for-loop?
- Why is it true initially?
- Why does it continue to be true later on?
- What does this property mean when the for-loop is reached the last time?

Selection Sort



Strategy

- Start empty handed.
- Enlarge the sorted part by swapping the *least* element of the unsorted part with the *first* element of the unsorted part.
- Continue until the unsorted part consists of one element only.

44	55	12	42	94	18	06	67
06	55	12	42	94	18	44	67
06	12	55	42	94	18	44	67
06	12	18	42	94	55	44	67
06	12	18	42	94	55	44	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	94	67
06	12	18	42	44	55	67	94

Selection Sort: Principles

- Idea: increase the sorted part by adding the minimum of the unsorted part.
- Initially, the empty segment A[1..0] is sorted and contains the 0 minimal elements
- Control structure: iteration over j,
  find min in A[j..n] and put it into position j
-

Selection Sort: Abstract Version

```
INPUT: A[1..n] - an array of integers
OUTPUT: a permutation of A such that A[1]≤A[2]≤…≤A[n]


for j := 1 to n-1 do
  // A[1..j-1] is sorted and contains the
  // j-1 minimal elements of the array
  minpos := findMinPos(A,j,n);
  swap(A,j,minpos)
```

Selection Sort: Principles

- Idea: increase the sorted part by adding the minimum of the unsorted part.
- Initially, the empty segment A[1..0] is sorted and contains the 0 minimal elements
- Control structure: iteration over j,
  find min in A[j..n] and put it into position j
- Inner loop: find the min in the rest A[j..n]
  Hypothesis: min is A[j], revise during inner loop.
  Control structure: iteration

Selection Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1] ≤ A[2] ≤ … ≤A[n]

for j := 1 to n-1 do // A[1..j-1] sorted and minimum
  min := A[j]; minpos := j
  for i := j+1 to n do
    if A[i] < min then min := A[i]; minpos := i;
  A[minpos] := A[j]; A[j] := min
```

Selection Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤A[2]≤…≤A[n]

for j := 1 to n-1 do // A[1..j-1] sorted and minimum
  min := A[j]; minpos := j
  for i := j+1 to n do
    if A[i] < min then min := A[i]; minpos := i;
  A[minpos] := A[j]; A[j] := min
```

The number of comparisons is independent of the original ordering (this is a less natural behavior than insertion sort):

$$C = \sum_{j=1}^{n-1} (n-j) = \sum_{k=1}^{n-1} k = (n*n - n)/2$$

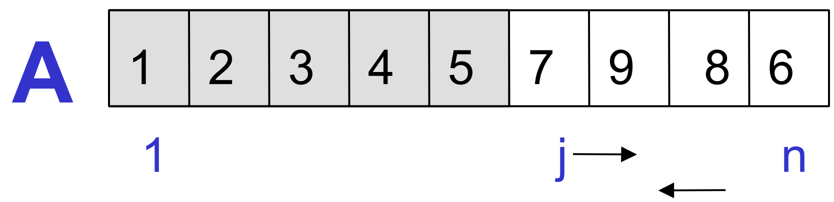
Selection Sort/3

The number of movements is:

$$M_{min} = \sum_{j=1}^{n-1} 3 = 3*(n–1)$$

$$M_{max} = \sum_{j=1}^{n-1} n - j + 3 = (n*n – n)/2 + 3*(n–1)$$

Bubble Sort



Strategy

- Start from the back and compare pairs of adjacent elements.
- Swap the elements if the larger comes before the smaller.
- In each iteration the smallest element of the unsorted part is moved to the beginning of the unsorted part and the sorted part grows by one.

44	55	12	42	94	18	06	67
06	44	55	12	42	94	18	67
06	12	44	55	18	42	94	67
06	12	18	44	55	42	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94
06	12	18	42	44	55	67	94

Bubble Sort: Principles

- Idea: let small elements move down (= to left).
  Effect: initial array segment is sorted.
- Control structure: Initially, the empty array A[1..0] is sorted, then the sorted part grows by one element per round
  => Iteration
- Sinking down: lesser elements are swapped with greater ones
  => Iteration

Bubble Sort

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1] ≤ A[2] ≤ … ≤ A[n]

for j := 2 to n do // A[1..j-2] sorted and minimum
  for i := n downto j do
    if A[i-1] > A[i] then
      swap(A,i,i-1)
```

Bubble Sort/2

```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤… ≤A[n]

for j := 2 to n do // A[1..j-2] sorted and minimum
  for i := n downto j do
    if A[i-1] > A[i] then
      val := A[i-1];
      A[i-1] := A[i];
      A[i]:= val
```

The number of comparisons is independent of the original ordering:

$$C = \sum_{j=2}^{n} (n - j + 1) = (n*n - n)/2$$

Bubble Sort/3

The number of movements is:

$M_{min} = 0$

$M_{max} = \sum_{j=2}^{n} 3(n-j+1) = 3*n*(n - 1)/2$

$M_{avg} = \sum_{j=2}^{n} 3(n-j+1)/2 = 3*n*(n - 1)/4$

Properties of a Sorting Algorithm

- Efficient: has low (worst case) runtime
- In place: needs (almost) no additional space (fixed number of scalar variables)
- Adaptive: performs little work if the array is already (mostly) sorted
- Stable: does not change the order of elements with equal key values
- Online: can sort data as it receives them

Sorting Algorithms: Properties

Which algorithm has which property?

	Adaptive	Stable	Online
Insertion Sort	Yes	Yes	Yes
Selection Sort	No	Yes, if ...	No
Bubble Sort	No		No

Sorting Algorithms: Properties

Which algorithm has which property?

	Adaptive	Stable	Online
Insertion Sort	Yes	Yes	Yes
Selection Sort	No	Yes (if we select the first minimum)	No
Bubble Sort	No	Yes	No

Summary

- Precise problem specification is crucial
- Precisely specify input and output
- Pseudocode, Java, C, … are largely equivalent for our purposes
- Recursion: procedure/function that calls itself
- Sorting: important problem with classic solutions