Data Structures and Algorithms

Roberto Sebastiani
roberto.sebastiani@disi.unitn.it
http://www.disi.unitn.it/~rseba

- Week 01 -
B.S. In Applied Computer Science
Free University of Bozen/Bolzano
academic year 2010-2011

Acknowledgements & Copyright Notice

Data Structures and Algorithms
Week 1

- Introduction, syllabus, administration
- Algorithms
- Recursion (factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

Data Structures and Algorithms
Week 1

- Introduction, syllabus, administration
- Algorithms
- Recursion (factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

The Goals of this Course

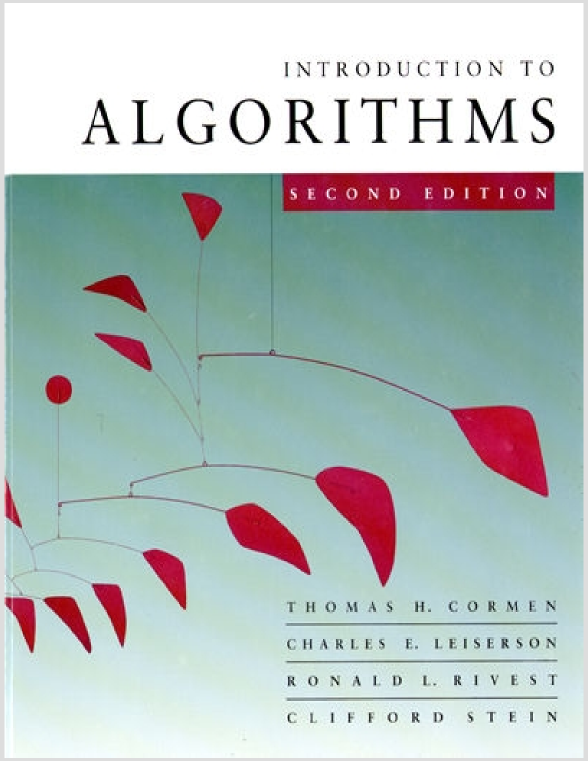
- In this course we will learn:
  - To *think algorithmically* and get the spirit of how algorithms are designed.
  - To get to know a *toolbox* of *classical* algorithms.
  - To learn a number of algorithm design *techniques* (such as divide-and-conquer).
  - To reason (in a precise and formal way)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

Syllabus (provisional)

1) Introduction, recursion (chap 1 in CLRS)
2) Correctness and complexity of algorithms (2, 3)
3) Divide and conquer, recurrences (4)
4) Sorting (1, 6, 7)
5) Pointers, lists, sets, abstract data types (10)
6) Trees, Red-black trees (12, 13)
7) Hash tables (11)
8) Dynamic programming (15)
9) Graph algorithms (22, 23, 24)
10) NP-Completeness (34)

References

- Notes from classes
- Slides (from course URL)
- Other material (from course URL)
- Book: Cormen, Leiserson, Rivest and Stein (CLRS), *Introduction to Algorithms*, Second Edition, MIT Press and McGraw-Hill, 2001.



Administration

- **Teaching assistants**
  - IT: Gennaro Iaccarino, gennaro.iaccarino@unibz.it
  - EN: Roberto Sebastiani, rseba@disi.unitn.it
  - GE: Johannes Martin, johannes.martin@es-tech.eu
- **Home page**
  - http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ/
  - Check the homepage frequently
- **Lectures** random room, TU & WE 10:30-12:30

Administration/2

- **Exercise classes (E331)**
  - WE 08.30-10:30 Italian, Iaccarino
  - WE 14:00-16:00 English, Sebastiani
  - TH 16:00-18:00 German, Martin
- **Office hours:**
  - TU & WE 09.30-10.30 and TU 12.30-13.30 **only by appointment!**
- **Exam**: written (details TBD)
- No midterm

General suggestions

- ATTEND CLASSES!!!!!
- During classes:
  - Interaction is welcome; ask questions.
  - Additional explanations and examples if desired.
  - Speed up/slow down the progress.
- Do not procrastinate studying:
  - Study each lesson before starting next one
  - Much more efficient exploitation of your efforts!!!

General suggestions/2

- When implementing algorithms:
  - Be simple and precise
  - Solve on paper (editor) first, then write code

Prerequisites

- Introduction to programming
  - Data types, operations
  - Conditional statements
  - Loops
  - Procedures and functions
- Computer lab (edit, compile, execute, navigate,...)
- Basic background in mathematics

Data Structures and Algorithms

Week 1

- Introduction, syllabus, administration
- Algorithms
- Recursion (factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

What is it all about?

- Solving problems
  - Get me from home to work
  - Balance my budget
  - Simulate a jet engine
  - Graduate from FUB
- To solve problems we have procedures, recipes, process descriptions – in one word ***algorithms.***

History

- Name: Persian mathematician Mohammed al-Khowarizmi, in Latin became Algorismus.
- First algorithm: Euclidean Algorithm, greatest common divisor, 400-300 B.C.
- 19th century – Charles Babbage, Ada Lovelace.
- 20th century – Alan Turing, Alonzo Church, John von Neumann.

Data Structures and Algorithms

- 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 specify the problem.
  - Designing programs
    - architecture
    - algorithms
  - Writing programs
  - Verifying (testing) programs

Data Structure and Algorithm Design Goals

Correctness

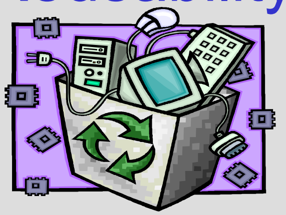
Efficiency

Implementation Goals

Robustness

Adaptability

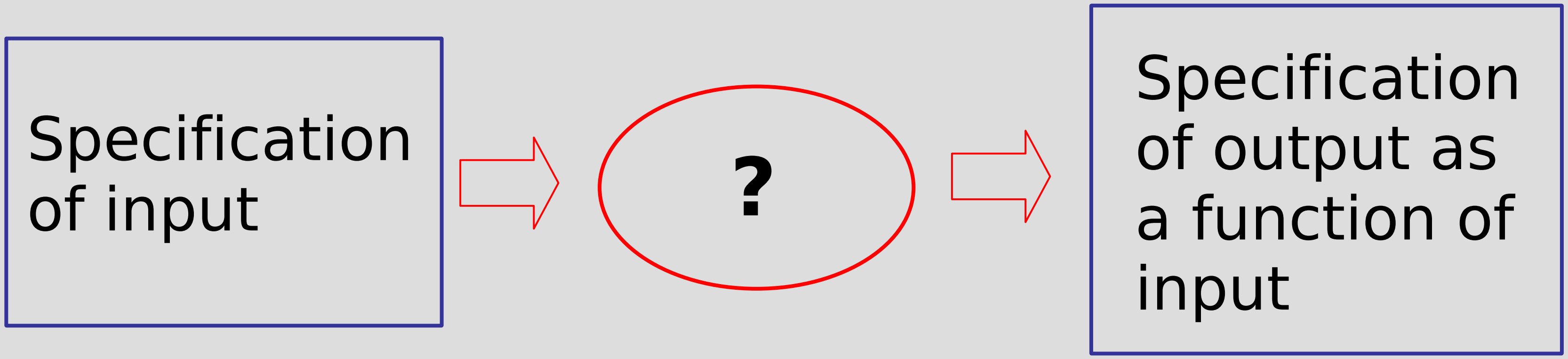
Reusability



Overall Picture/2

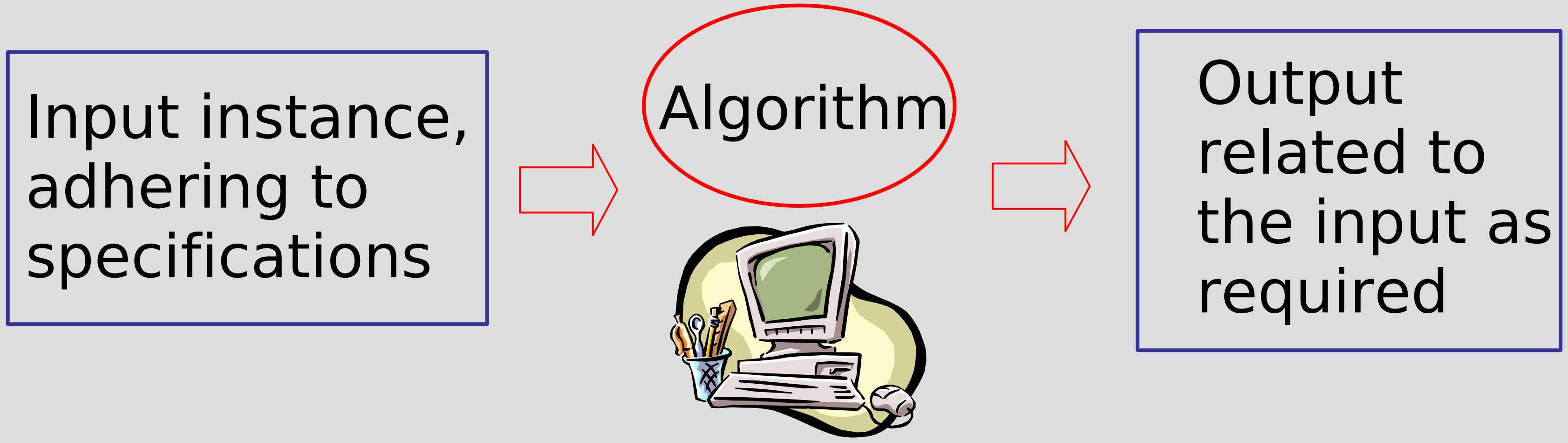
- This course is **not** about:
  - Programming languages
  - Computer architecture
  - Software architecture
  - SW design and implementation principles
    - Issues concerning small and large scale programming.
- We will only touch upon the theory of complexity and computability.

Algorithmic Problem



– Infinite number of input *instances* satisfying the specification. For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    – 1, 20, 908, 909, 100000, 1000000000.

Algorithmic Solution


Input instance, adhering to specifications
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 of an Algorithm

- An **algorithm** is a sequence of *unam-biguous* instructions for solving a problem, i.e., for obtaining a *required output* for any *legitimate input* in a finite amount of time.
- Important properties:
  - Correctness, Efficiency, Generality,
  - Precision, Determinism, Finiteness

How to Develop an Algorithm

- **Precisely define** the problem. Precisely specify the input and output. Consider all cases.
- Come up with a **simple plan** to solve the problem at hand.
  - The plan is language independent.
  - The precise problem specification influences the plan.
- Turn the plan into an implementation
  - The problem representation (data structure) influences the implementation.

Preconditions, Postconditions

- It is important to specify the preconditions and the postconditions of algorithms:
  - INPUT: precise specifications of what the algorithm gets as an input.
  - OUTPUT: precise specifications of what the algorithm produces as output, and how this relates to the input. The handling of special cases of the input should be described.

Example 1: Searching

INPUT

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

$a_1, a_2, a_3, \ldots, a_n;\ q \xrightarrow{} j$

2 5 6 10 11; 5 $\longrightarrow$ 2

2 5 6 10 11; 9 $\longrightarrow$ *NIL*

OUTPUT

- index of number q in sequence A or *NIL*

Searching/2, search1

search1
INPUT: A[1..n] (un)sorted array of integers, *q* an integer.
OUTPUT: index *j* such that A[*j*]=*q*. *NIL* if ∀*j*(1≤*j*≤*n*): A[*j*]≠q

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

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

Pseudo-code

- A la Pascal, C, Java or any other imperative language:
  - Control structures (if then else, while, and for loops)
  - Assignment: :=
  - Array element access: A[i]
  - Composite type (record or object) element access: A.b (CLRS uses b[A]).

Searching, C solution

```
#include <stdio.h>
#define n 5

int j, q;
int a[n] = { 11, 1, 4, -3, 22 };
int main() {
  j = 0;  q = -2;
  while (j < n && a[j] != q) { j++; }
  if (j < n) { printf("%d\n", j); }
  else { printf("NIL\n"); }
}
// compilation: gcc -o search search.c
// execution: ./search
```

How to approach C, etc.

- Do not study it (it is close to Java and we only use a small subset).
- Whenever you meet a new construct, learn and use it.
- Here:
  - `#define n 5`
    defines n as a constant with value 5
  - `printf("%d\n",i)`
    prints an integer argument (%d) followed by a new line (\n).
  - Arrays have a fixed size and start with index 0.

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 };

  int 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("NIL"); }
  }
}

// compilation: javac search.java
// execution: java search
```

Searching/3, search2

- Other solution to search?
- 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*. *NIL* if $\forall j(1 \leq j \leq n)$: A[*j*]≠*q*

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

search1 vs search2

- Are the solutions equivalent?
- No!
- Construct an example such that
  - search1 returns 3
  - search2 returns 7
- What's the problem?
  - Input and output were **not** precisely defined.

Searching/4, search3

- A third solution: run through the array and **return** the index of the value in the array.

search3
INPUT: A[1..n] – sorted array of integers, *q* – an integer.
OUTPUT: index *j* such that A[*j*]=*q*. *NIL* if $\forall j (1 \le j \le n)$: A[*j*]$\neq q$

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

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 is the most efficient.
- search3 does not finish the loop (as a general rule it is good to avoid this).

Beware: array indexes in Java/C/C++

- Users/pseudocode writers see array indexes ranging from 1 to length
- In Java/C/C++ array indexes range from 0 to length-1
- Examples:
  - “**for** j := 1 **to** n **do**” in pseudocode is written in java as :

    “`for (j=0;j<a.length;j++) {`“
  - “**for** j := n **to** 2 **do**” in pseudocode is written in java as :

    “`for (j=a.length-1;j>=1;j--) {`“

Suggested exercises

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

Data Structures and Algorithms
Week 1

- Introduction, syllabus, administration
- Algorithms
- Recursion (factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

Recursion

- An object is **recursive** if
  - it contains itself as part of itself, or
  - it is defined in terms of itself.

  Ex: matematical expressions ((3*7)-(9/3)):

  EXPR := VALUE | (EXPR OPERATOR EXPR)
- A **recursive** procedure: a procedure which calls itself

  Ex: Factorial: n!
  - n! = 1 * 2 * 3 *...* n
  - n! = n * (n-1)!

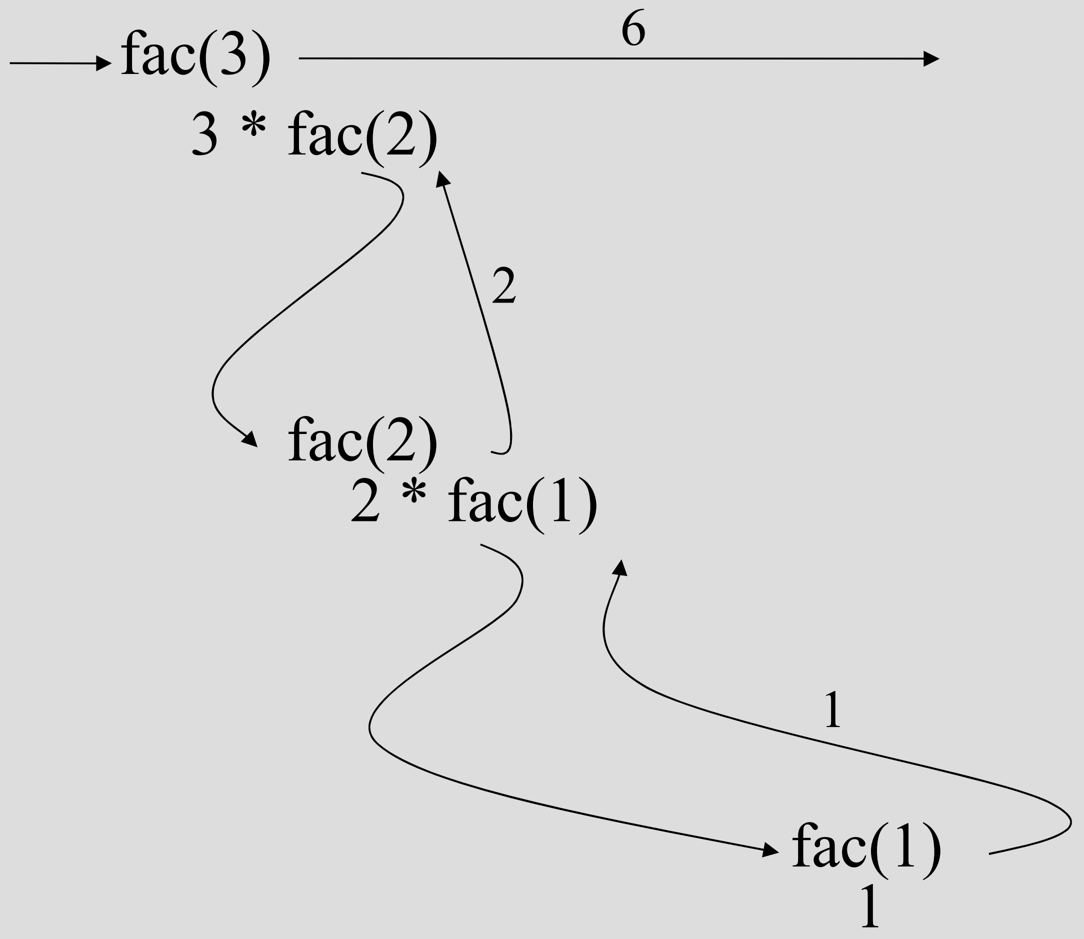
Factorial Function

- 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)
```

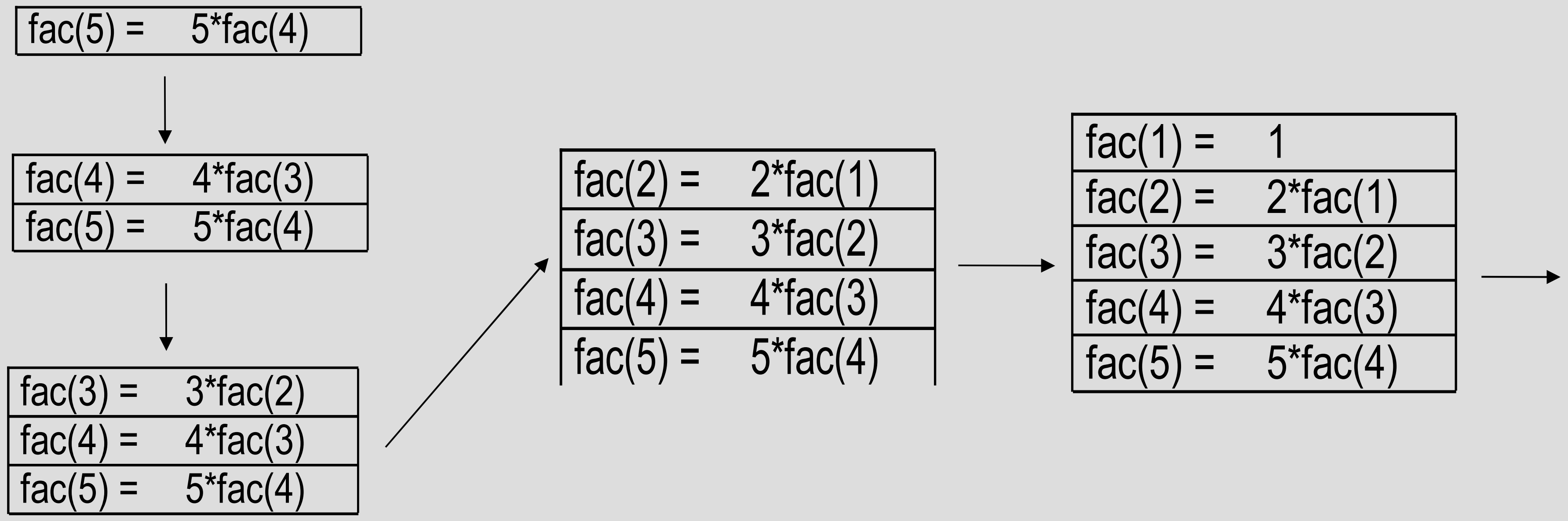
- A recursive procedure includes:
  - a termination condition (determines when and how to stop the recursion).
  - one (or more) recursive calls.

Tracing the Execution



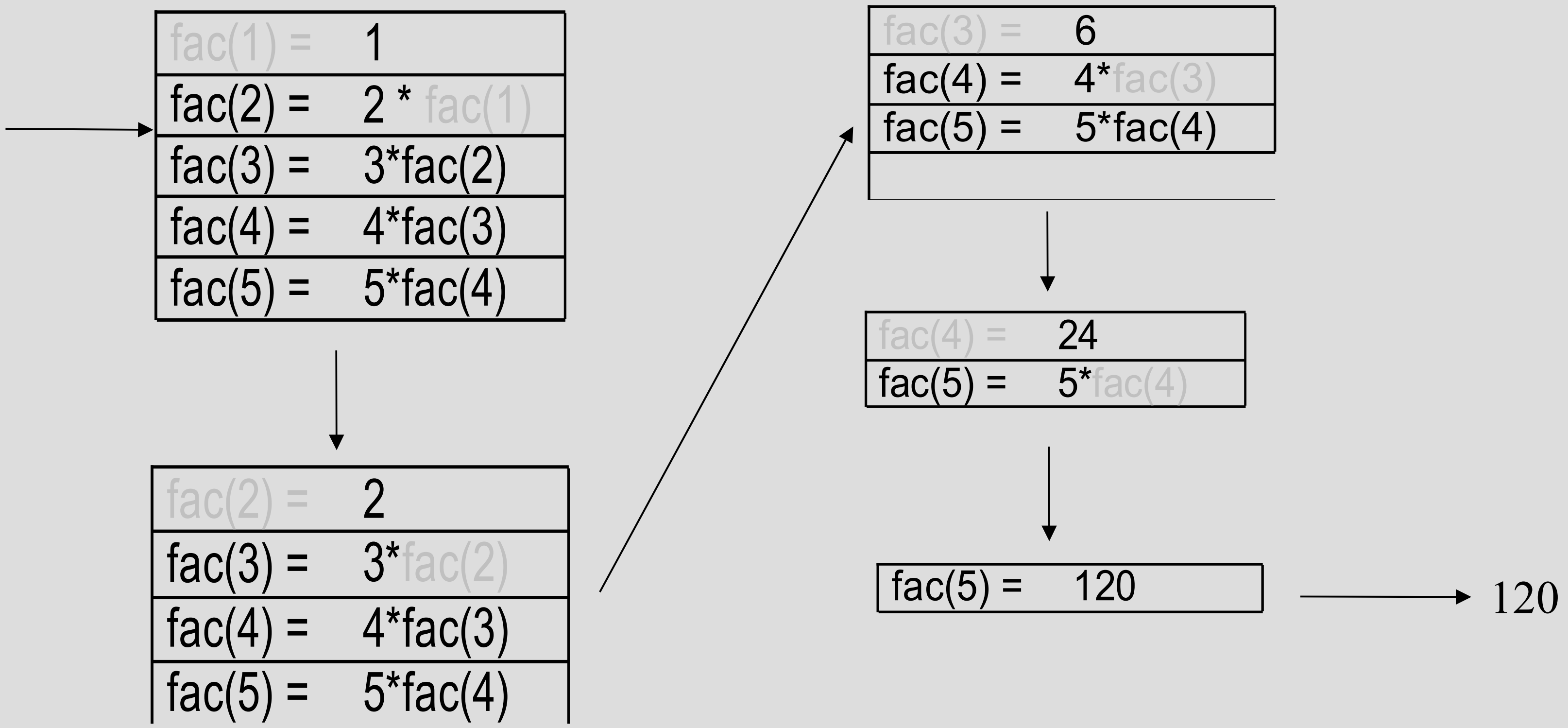
Bookkeeping

- The computer maintains an activation stack for active procedure calls (-> compiler construction). Example for fac(5).



Bookkeeping/2

- Reducing the activation stack



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 solutions

- 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 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)

```
fac2(n)
  if n <= 1 then return 1
  return n*(n-1)*fac4(n-2)
```

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

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

Analysis of the solutions

- 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.

Fibonacci Numbers

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

- Numbers in the series:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Implementation

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

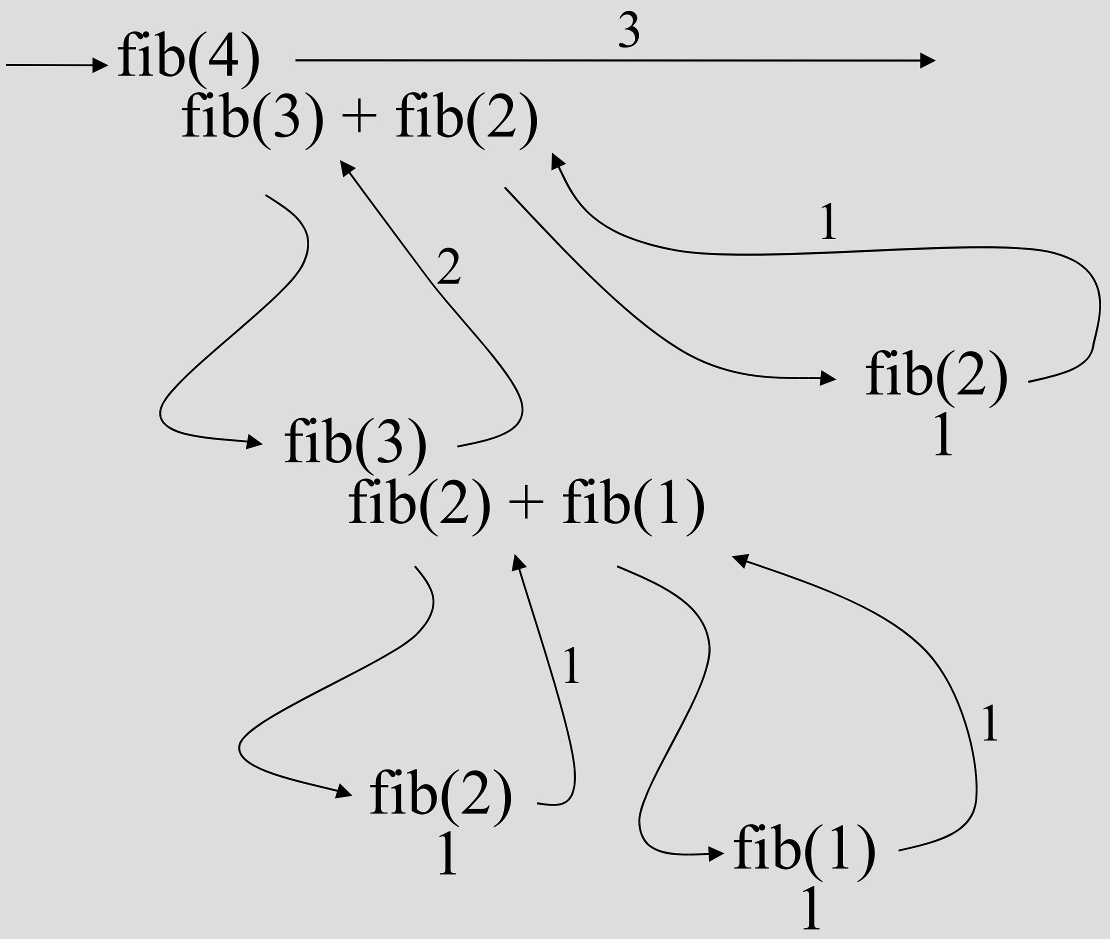
```
fib(n)
  if n ≤ 2 then return 1
  else return fib(n-1) + fib(n-2)
```

- Multiple recursive calls are possible.

Fibonacci Implementation/2

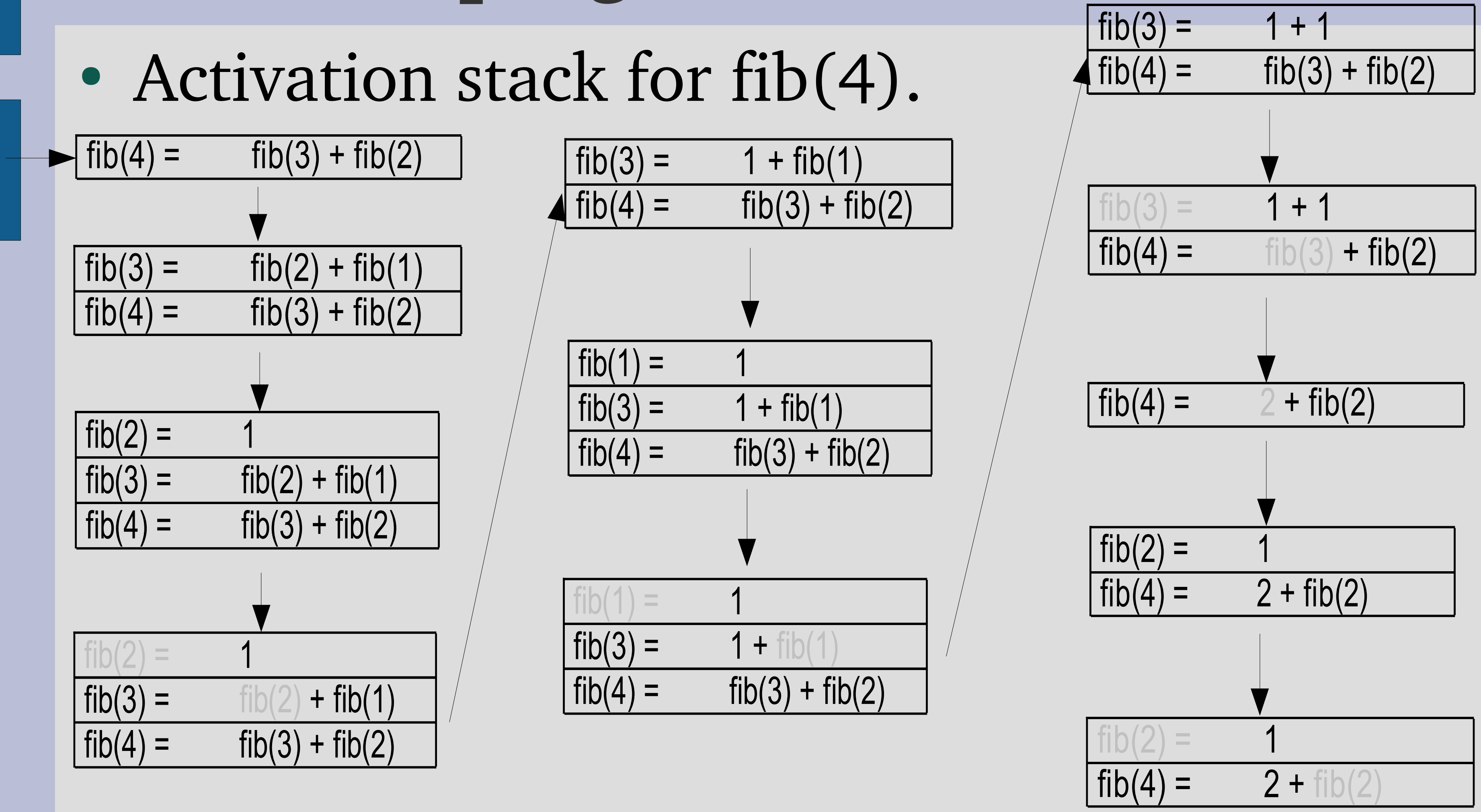
```
#include <stdio.h>
int fib(int i) {
  if (i <= 2) { return 1;}
  else { return fib(i-1) + fib(i-2); }
}

int main() {
  printf(“Fibonacci of 5 is %d\n", fib(5));
}

// compilation: gcc -o fib fib.c
// execution: ./fib
```

Tracing fib(4)



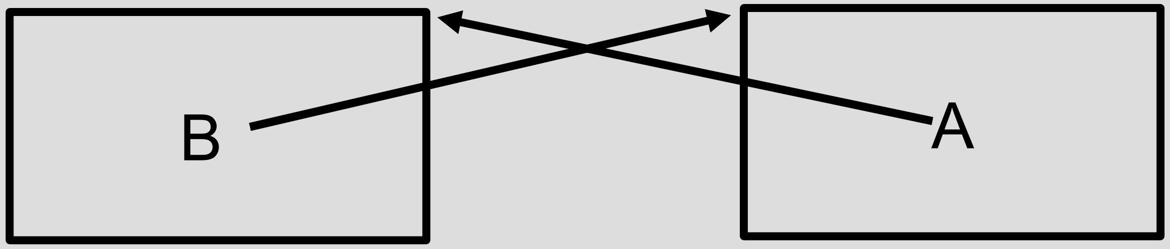
Bookkeeping

- Activation stack for fib(4).



Mutual Recursion

- Recursion does not always occur because a procedure calls itself.
- Mutual recursion occurs if two procedures call each other.



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

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

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

- Can it be used to determine whether a number 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 correspon-ding iterative solutions since bookkeeping is necessary.

Suggested exercises

- Implement (the various variants of) fact, Fibonacci and even&odd
- Add a counter counting the number of recursive calls to Fibonacci and print its final value: what happens?
- Implement a recursive function which reads and evaluates a simple arithmetical expression as in the example

Data Structures and Algorithms
Week 1

- Introduction, syllabus, administration
- Algorithms
- Recursion (factorial, Fibonacci)
- Sorting (bubble, insertion, selection)

A Quick Math Refresher

- Arithmetic progression

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(1+n)}{2}$$

intuition: $(1+n)+(2+n-1)+\ldots = (1+n)\ n/2$ *times*

- Geometric progression
  - given an integer *n* and a real number $0<a\neq 1$

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

intuition: $(1-a)(1+a+a^2+\ldots+a^n)=1-a^{n+1}$

  - geometric progressions exhibit exponential growth

Miscellaneous/1

- Manipulating summations:
  - $\sum_j c a_j = c \sum_j a_j$
  - $\sum_j (a_j + b_j) = \sum_j a_j + \sum_j b_j$

Miscellaneous/2

- Manipulating logarithms and powers:
  - $\log_a b = \log b / \log a$
  - $\log a^b = b \log a$
  - $\log(ab) = \log a + \log b$
  - $a^{mn} = (a^m)^n$
  - $a^m a^n = a^{m+n}$
  - $a^{\log_a(b)} = b$

Sorting

- Sorting is a classical and important algorithmic problem.
- We look at sorting arrays (in contrast to files, which restrict random access).
- A key constraint is the efficient management of the space: in-place sorting algorithms (no extra RAM).
- The efficiency 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

Example 2: Sorting

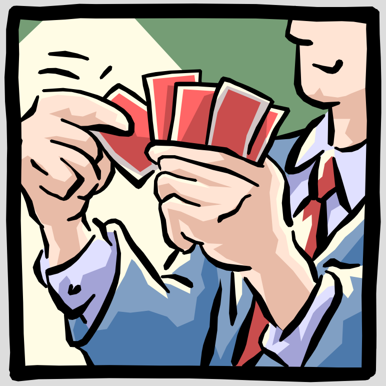
INPUT

sequence of *n* numbers

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

$\longrightarrow$

2 5 4 10 7



OUTPUT

a permutation of the input sequence of numbers

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

$\longrightarrow$

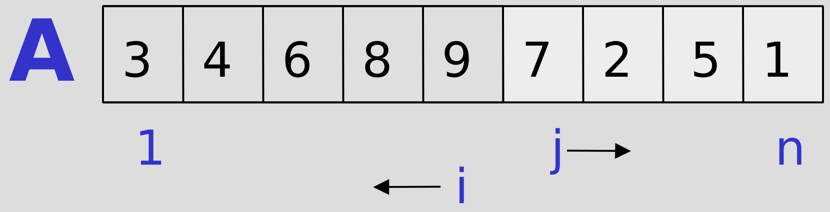
2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$
- $b_1, b_2, b_3, \ldots, b_n$ 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/2

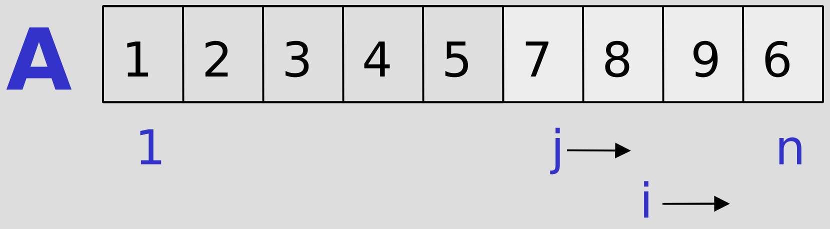
```
INPUT: A[1..n] – an array of integers
OUTPUT: permutation of A s.t. A[1]≤A[2]≤...≤A[n]
for j := 2 to n do // A[1..j-1] sorted
  key := A[j]; i := j-1;
  while i > 0 and A[i] > key do
    A[i+1] := A[i];  i--;
  A[i+1] := key
```

- The number of comparisons during the jth iteration is
  - at least 1: Cmin $=\sum_{j=2}^{n} 1 \quad = \text{n-1}$
  - at most j-1: Cmax $=\sum_{j=2}^{n} (j-1) =$ (n*n-n)/2

Insertion Sort/3

- The number of comparisons during the jth iteration is:
  - j/2 average: Cavg = $\sum_{j=2}^{n} j/2$ = (n*n+n–2)/4
- The number of movements is Ci+1:
  - Mmin = $\sum_{j=2}^{n} 2$ = 2*(n-1),
  - Mavg = $\sum_{j=2}^{n} j/2+1$ = (n*n+5n-6)/4
  - Mmax = $\sum_{j=2}^{n} j$ = (n*n+n-2)/2

Selection Sort



Strategy

- Start empty handed.
- Enlarge the sorted part by switching the first element of the unsorted part with the smallest 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/2

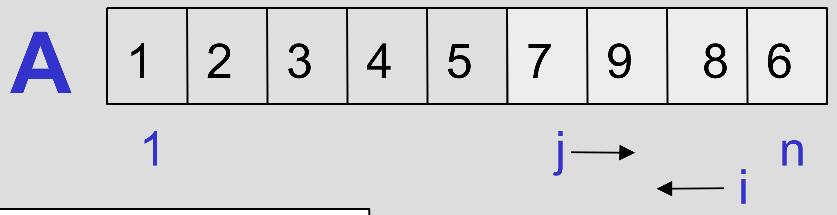
```
INPUT: A[1..n] – an array of integers
OUTPUT: a permutation of A such that A[1]≤A[2]≤…≤A[n]

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

- The number of comparisons is independent of the original ordering (this is 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:
  - Mmin = $\sum_{j=1}^{n-1} 3$ = 3*(n-1)
  - Mmax = $\sum_{j=1}^{n-1} n\text{-}j+3$ = (n*n–n)/2 + 3*(n-1)

Bubble Sort



Strategy

- Start from the back and compare pairs of adjacent elements.
- Switch the elements if the larger comes before the smaller.
- In each step 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/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 to j do
    if A[i-1] > A[i] then
      key := A[i-1];
      A[i-1] := A[i];
      A[i]:=key
```

- 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:
  - Mmin = 0
  - Mmax $=\sum_{j=2}^{n} 3(n-j+1)=$ 3*n*(n-1)/2
  - Mavg = $\sum_{j=2}^{n} 3(n-j+1)/2 =$ 3*n*(n-1)/4

Suggested exercises

- Implement insertion, selection and bubble sort.
- Implement them in reverse order.
- Add counters counting the number of comparisons and assignment and print the results: what happens?
- Implement a version of bubble sort which stops the external loop if the array is already sorted

Summary

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

Next Week

- Algorithmic complexity
- Asymptotic analysis, big *O* notation
- Correctness of algorithms
- Special case analysis