Data Structures and Algorithms

Part 1

Werner Nutt
Acknowledgments

• The course follows the book “Introduction to Algorithms“, by Cormen, Leiserson, Rivest and Stein, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.

• These slides are based on those developed by Michael Böhlen for this 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, Part 1: Overview

• Introduction, syllabus, organisation

• Algorithms

• Recursion (principle, trace, factorial, Fibonacci)

• Sorting (bubble, insertion, selection)
DSA, Part 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 *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 analyze (in a precise and formal way) 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)
Literature


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

• Lectures: Tue 8:30-10:30, Fri 8:30-10:30

• Labs
  – English: Mouna Kacimi, Mouna.Kacimi@unibz.it
    Fri 10:30-12:30
  – German: Simon Razniewski, razniewski@inf.unibz.it
    Mon 8:30-10:30
  – Italian: Valeria Fionda, Valeria.Fionda@unibz.it
    Mon 14:00-16:00

• Home page:
  http://www.inf.unibz.it/~nutt/DSA1112/
Assignments

The assignments are a crucial part of the course

• **Each week** an assignment has to be solved
• Assignments will be **published on Monday** and have to be **handed in on the Monday** of the following week (in the exercise group or in the lecture)
• Late hand-ins are not accepted.
• 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.
Assignment, Exam and Course Mark

• There will be one written exam at the end of the course.
• Students who do not submit exercises will be assessed on the exam alone.
• For students who submit all assignments, the final mark will be a weighted average
  50% exam mark + 50% assignment mark
• If students submit fewer assignments, the percentage will be lower.
• Assignments for which the mark is lower than the mark of the written exam will not be considered.
• The assignment marks apply to three exam sessions.
General

• 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, Part 1:

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

Solving problems in everyday life

- **Travel** from Bolzano to Munich
- **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, find 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, find 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 التفريع بحساب الهند
  = *Kitāb al-Dschamʿ 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
  – Ada Lovelace: Program for Bernoulli numbers

• **20th century**
  – Alan Turing, Alonzo Church: formal models computation
  – John von Neumann: architecture of modern computers
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
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.
There is an infinite number of possible input \textit{instances} satisfying the specification.

For example: A sorted, non-decreasing sequence of natural numbers, on nonzero, finite length:

\begin{verbatim}
1, 20, 908, 909, 100000, 1000000000.
\end{verbatim}
Algorithmic Solution

- 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.

implies This presumes a mechanism to execute the algorithm

Properties of algorithms:
• Correctness, Termination, (Non-)Determinism, Run Time, …
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 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
Preconditions, Postconditions

Specify preconditions and postconditions of algorithms:

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, pointer nil, …
Example 1: Searching

**INPUT**
- A - (un)sorted sequence of \( n \) numbers \( (n > 0) \)
- q - a single number \( a_1, a_2, a_3, \ldots, a_n; q \)

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>j</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NIL</td>
</tr>
</tbody>
</table>
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 for all 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 pseudo-code 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 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 \( b[A] \)
Searching, Java Solution

```java
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("NIL"); }
    }
}
```
Searching, C Solution

```c
#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
```
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. NIL if for all j (1 ≤ j ≤ n): A[j] ≠ q

ptr := NIL;
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?
• No!

Can one construct an example such that, say,
• search1 returns 3
• search2 returns 7

But both solutions satisfy the specification (or don’t they?)
An third idea:

Run through the array and \textbf{return} the index of the value in the array.

\begin{minted}{algorithm}
search3
INPUT: A[1..n] (un)sorted array of integers, q an integer.
OUTPUT: index j such that A[j]=q. NIL if for all j (1 ≤ j ≤ n): A[j] ≠ q

for j := 1 to n do
  if a[j] = q then return j
return NIL
\end{minted}
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 to 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)
  – 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, Part 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)
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 ...

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
by Alex Allain - More by Alex Allain
Learn how to use recursion in C and C++, with example recursive programs.

Recursion -- from Wolfram MathWorld
Part 1

Introduction, Algorithms, Recursion, Sorting

Source: http://bluehawk.monmouth.edu/~relayton/web-pages/s11-503/recursion.jpg
Recursion/2

• A recursive definition: a concept is defined by referring to itself.
  E.g., arithmetical expressions (like \((3 \times 7) - (9 / 3)\)):
  
  \[
  
  \text{EXPR} := \text{VALUE | (EXPR OPERATOR EXPR)}
  
  \]

• A recursive procedure: a procedure which calls itself
  Classical example: factorial, that is \(n! = 1 \times 2 \times 3 \times \ldots \times n\)

  \[
  
  n! = n \times (n-1)!
  
  \]

  … or is there something missing?
The Factorial Function

Pseudocode of factorial:

```plaintext
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

fac(3) → 6

3 * fac(2)

2

2 * fac(1)

1

fac(1)

1

fac(2)

2 * fac(1)

1

3 * fac(2)

2

fac(2)

2 * fac(1)

1

fac(1)

1

6
Bookkeeping

The computer maintains an **activation stack** for active procedure calls (**→** compiler construction).

Example for fac(5). The stack is built up.

\[
\begin{align*}
\text{fac}(5) &= 5 \times \text{fac}(4) \\
\text{fac}(4) &= 4 \times \text{fac}(3) \\
\text{fac}(5) &= 5 \times \text{fac}(4) \\
\text{fac}(3) &= 3 \times \text{fac}(2) \\
\text{fac}(4) &= 4 \times \text{fac}(3) \\
\text{fac}(5) &= 5 \times \text{fac}(4) \\
\text{fac}(2) &= 2 \times \text{fac}(1) \\
\text{fac}(3) &= 3 \times \text{fac}(2) \\
\text{fac}(4) &= 4 \times \text{fac}(3) \\
\text{fac}(5) &= 5 \times \text{fac}(4) \\
\text{fac}(1) &= 1
\end{align*}
\]
Then the activation stack is reduced

<table>
<thead>
<tr>
<th>$\text{fac}(1)$</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fac}(2)$</td>
<td>$2 \times \text{fac}(1)$</td>
</tr>
<tr>
<td>$\text{fac}(3)$</td>
<td>$3 \times \text{fac}(2)$</td>
</tr>
<tr>
<td>$\text{fac}(4)$</td>
<td>$4 \times \text{fac}(3)$</td>
</tr>
<tr>
<td>$\text{fac}(5)$</td>
<td>$5 \times \text{fac}(4)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{fac}(3)$</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fac}(4)$</td>
<td>$4 \times \text{fac}(3)$</td>
</tr>
<tr>
<td>$\text{fac}(5)$</td>
<td>$5 \times \text{fac}(4)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{fac}(2)$</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fac}(3)$</td>
<td>$3 \times \text{fac}(2)$</td>
</tr>
<tr>
<td>$\text{fac}(4)$</td>
<td>$4 \times \text{fac}(3)$</td>
</tr>
<tr>
<td>$\text{fac}(5)$</td>
<td>$5 \times \text{fac}(4)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{fac}(4)$</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fac}(5)$</td>
<td>$5 \times \text{fac}(4)$</td>
</tr>
</tbody>
</table>

|  $\text{fac}(5)$ | 120 |
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:

  \[
  \text{fact}(3) \\
  = 3 \times 2 \times \text{fact}(1) \\
  = 3 \times 2 \times 1 \times 0 \times \text{fact}(-1) \\
  = \ldots
  \]
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 < 2 then return 1
Analysis of the Variants/2

\texttt{fac4} is correct

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

\texttt{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

time = 0  1  2  3  4  5

pairs = 1  1  2  3  5  8

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

Definition

• \( \text{fib}(1) = 1 \)
• \( \text{fib}(2) = 1 \)
• \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2), \ n > 2 \)

Numbers in the series:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci numbers occur all the time in nature:

• number of sections in apples, oranges, ...
• spirals in flowers and pine cones
Fibonacci Procedure

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)

A procedure with multiple recursive calls
Fibonacci Procedure/2

```java
public class fibclassic {

    static int fib(int n) {
        if (n <= 2) {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 $\text{fib}(4)$

$\text{fib}(4) \rightarrow \text{fib}(3) + \text{fib}(2)$

$\text{fib}(3) \rightarrow \text{fib}(2) + \text{fib}(1)$

$\text{fib}(2) \rightarrow 1$

$\text{fib}(1) \rightarrow 1$
Bookkeeping

\[\text{fib}(1) = 1\]
\[\text{fib}(2) = 1\]
\[\text{fib}(3) = \text{fib}(2) + \text{fib}(1)\]
\[\text{fib}(4) = \text{fib}(3) + \text{fib}(2)\]
Questions

• How many recursive calls are made to compute $fib(n)$?
• What is the maximal height of the recursion stack during the computation?
• How are number of calls and height of the stack related to the size of the input?
• Can there be a recursive procedure for fib with fewer calls?
• How is the size of the result $fib(n)$ related to the size of the input $n$?
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

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)

• 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.

• We shall see: recursion allows for elegant divide and conquer algorithms.
DSA, Part 1:

- Introduction, syllabus, organisation
- Algorithms
- Recursion (principle, trace, factorial, Fibonacci)
- Sorting (bubble, insertion, selection)
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 $n \times n$ comparisons
  – Insertion sort
  – Selection sort
  – Bubble sort

• **Fast** sorting methods use roughly $n \times \log n$ 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:

- $b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$
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
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.
**Insertion Sort/2**

**INPUT:** A[1..n] – an array of integers  

```plaintext
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: \( C_{\text{min}} = \sum_{j=2}^{n} 1 = n - 1 \)
- at most \( j - 1 \): \( C_{\text{max}} = \sum_{j=2}^{n} j - 1 = (n^2 - n)/2 \)
Insertion Sort/3

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

• The number of movements is \( C_{i+1} \):
  - \( M_{\text{min}} = \sum_{j=2}^{n} 2 = 2(n-1) \),
  - \( M_{\text{avg}} = \sum_{j=2}^{n} j/2 + 1 = (n^2 + 5n - 6)/4 \),
  - \( M_{\text{max}} = \sum_{j=2}^{n} j = (n^2 + n - 2)/2 \)
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.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Starting array:

44 55 12 42 94 18 06 67

After each step:

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 67 94
Selection Sort/2

**INPUT:** A[1..n] – an array of integers  

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];  

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

\[
C = \sum_{j=1}^{n-1} (n-j) = \sum_{k=1}^{n-1} k = \frac{n^2 - n}{2}
\]
Selection Sort/3

The number of movements is:

\[ \text{Mmin} = \sum_{j=1}^{n-1} 3 = 3(n-1) \]

\[ \text{Mmax} = \sum_{j=1}^{n-1} (n-j+3) = \frac{n^2 - n}{2} + 3(n-1) \]
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.

<table>
<thead>
<tr>
<th>A</th>
<th>1 2 3 4 5 7 9 8 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>j → n</td>
</tr>
</tbody>
</table>

| 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


```plaintext
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:

\[ C = \sum_{j=2}^{n} (n − j + 1) = \frac{n^2 - n}{2} \]
Bubble Sort/3

The number of movements is:

\[ M_{\text{min}} = 0 \]

\[ M_{\text{max}} = \sum_{j=2}^{n} 3(n - j + 1) = \frac{3n(n - 1)}{2} \]

\[ M_{\text{avg}} = \sum_{j=2}^{n} \frac{3(n - j + 1)}{2} = \frac{3n(n - 1)}{4} \]
### Sorting Algorithms: Properties

Which algorithm has which property?

<table>
<thead>
<tr>
<th></th>
<th>Adaptive</th>
<th>Stable</th>
<th>Online</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• Precise problem specification is crucial.

• Precisely specify Input and Output.

• Pseudocode, Java, C, … is largely equivalent for our purposes.

• Recursion: procedure/function that calls itself.

• Sorting: important problem with classic solutions.