

UNIVERSITY OF TWENTE.

# Pearls of Computer Science

Module 1.1, Study *Technical Computer Science*  
Academic year 2016/2017

DATE  
September 5, 2016  
PARTIAL EDITION

## Chapter 2: Pearl 001 — Algorithmics

### MODULE COORDINATOR

Maurice van Keulen

### DESIGN TEAM

Maurice van Keulen

Arend Rensink

Pieter-Tjerk de Boer

### INVOLVED TEACHERS

Harry Aarts

Rieks op den Akker

Pieter-Tjerk de Boer

Ansgar Fehnker

Marco Gerards

Celine Heijnen

Dirk Heylen

Maurice van Keulen

Frans de Kogel

André Kokkeler

Jan Kuper

Arend Rensink

Andreas Peter

Mannes Poel

Klaas Sikkell

Karen Slotman

### FORMER TEACHERS

Pascal van Eck

Anne Remke

Dolf Trieschnigg

### TEACHING ASSISTANTS

Kevin Alberts

Dex Bleeker

Tim Blok

Jeroen Bos

René Boschma

Dennis Cai

Sharbel Chmoun

Twan Coenraad

Frans van Dijk

Vincent Dunning

Thijs van Essen

Paula Felix

Noah Goldsmid

Aron van Harten

Joran Honig

Wim Kamerman

Lindsay Kempen

Etienne Khan

Dirk Koelewijn

Mark Kok

Jan-Jaap Korpershoek

Yoep Kortekaas

Remco de Man

Sander Meinderts

Ömer Sakar

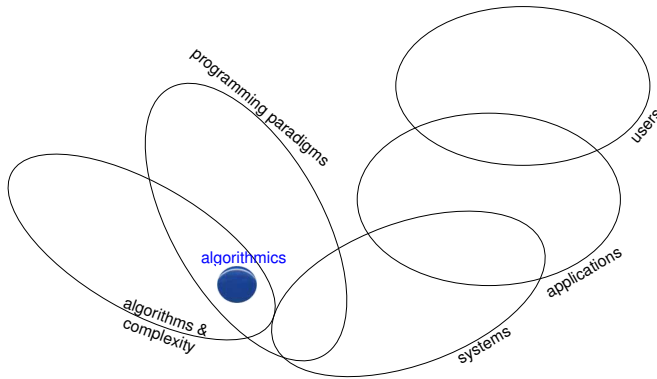
Jasper Sustronk

Wouter Timmermans

Jeroen Weener



The Blue Pearl is the subtle abode of the inner Self  
Swami Muktananda —*Does Death Really Exist?*



Week **2**

## Pearl 001: Algorithmics

### 2.1 Introduction

To make a computer do what you want it to do, you have to program it. A lot of issues are involved, but one of the main requirements is to have a good insight in the *algorithms* you can use in your programs. In this pearl you will meet a number of existing, fundamental algorithms for *sorting* and *searching*.

The working of an algorithm is independent of the programming language you use, but to practice we have to choose a language. For this week, the choice has fallen on PYTHON (version 3).

Last week you have also programmed a bit, but on a much lower level, where you had to include every single step the processor should take. PYTHON is a “higher-order language”, which makes the life of a programmer a whole lot easier.

#### 2.1.1 Global description of the pearl assignment

The pearl assignment will consist of programming a search engine: after typing in a search term, you will get a list of (names of) documents in which this term occurs. The assignment can be extended at will, for instance by returning the results in decreasing order of frequency, by allowing more complex search terms, or in some other way you may think of yourself. You will do the assignment in pairs.

#### 2.1.2 Study material and tools

The study material of this pearl consists of:

- Slides and this reader: on Blackboard
- Tutorials and language definition of PYTHON: on the Internet

The practical of this pearl uses the following tools and files:

- A PYTHON 3 interpreter. *Nota bene:* when installing the interpreter under Windows, the `PATH` variable should be set.

U	Ma	Di	Wo	Do	Vr	Abbr Form	Abbr. Part
1	M tst T∞	P lec T∞	M self A∞	P ass2 N	M self N	ass Assignment	P Pearl
2	M tst T∞	P prac T∞	M self A∞	P ass2 N	M self N	prac Practical	F Final project
3	M lec T∞	S tut T24	M tut T∞	P ass2 A∞	P ass2 A∞	lec Lecture	S Academic skills
4	M lec T∞	S tut T24	M tut T∞	P ass2 A∞	P ass2 A∞	pfb Peer feedback	M Math
6	P lec T∞	P prac A∞	P ass2 A∞	M tut T∞	P tst T∞	pj Project	Abbr Supervision
7	P prac T∞	P prac A∞	P ass2 A∞	M tut T∞	P tst T∞	qa Q&A	T Teacher
8	P prac A∞	P prac A∞	P self N	M self N	P ass2 N	self Self study	A Teaching assistant
9	P prac A∞	P prac A∞	P self N	M self N	P ass2 N	tst Test	N None
						tut Tutorial	

<b>Mon</b>	1–4	Math
	6	Introductory lecture: What are algorithms? What can you do with them? Why is it important to know about them? Why PYTHON?
	7–9	Practical: linear and binary search in PYTHON (Sections 2.3.1–2.3.2)
<b>Tue</b>	1	Introductory lecture: Sorting, fast and slow. What makes (searching and) sorting important and interesting?
	2	Practical: Various sorting algorithms in PYTHON (Sections 2.3.3–2.3.4)
	3–4	Academic skills
	6–9	Continuation of practical 2 <sup>nd</sup> hour
<b>Wed</b>	1–4	Math
	6–7	Starting the pearl assignment (Section 2.3.5)
	8–9	Self study: reading about algorithms, PYTHON, ...
<b>Thu</b>	1–4	Working on pearl assignment
	6–9	Math
<b>Fri</b>	1–2	Math self-study or pearl test preparation
	3–4	Finishing pearl assignment
	6–7	Test
	8–9	Finishing pearl assignment

Week 2 per session

- An arbitrary text editor; for instance `IDLE` that is packaged with `PYTHON`, but alternatively `NotePad++` is an excellent choice as well.
- Pre-defined `PYTHON`-files (on Blackboard)
- Pre-defined text files for the practical exercises and pearl assignment (on Blackboard)

You are entirely free to use the wealth of material on the Internet if and when you need more (practice) material on `PYTHON`. There are many, many beginner courses, with many, many examples and other material. Of course, your solutions for the practical exercises and pearl assignment should be your own; never just copy and paste.

### 2.1.3 Obligatory sessions and deliverables

Mandatory elements of this week are:

- Having the practical exercises signed off
- Participating in the Academic Skills-session
- Participating in the test
- Handing in the pearl assignment

## 2.2 Academic Skills

This week the first session for Academic Skills takes place. The session takes 1 lecture hour. Whether this is the 3rd or the 4th hour depends on your AS-group. See Blackboard to find out to which AS-group you are assigned and when you are expected and where.

**Study load:** 1.5hrs (incl. meeting 1)

**Learning objectives:** Apply “Hall’s Model of Cultural Differences” on your own situation and you can reflect on your personal experiences with other cultures.

**Related theory:** chapter 1 and 2 of “Cultural Sensitivity” (Nunez, Nunez et Popma, 2010).

- Read chapter 1 and 2 before the start up meeting
- Fill in the questionnaire. Follow the instructions of the teacher.
- Bring your book ‘Cultural Sensitivity’ to start up meeting (MT1)

## 2.3 Description of the sessions and lab assignments

The right way to do the exercises of this practical is:

**Read the entire text** and not just the exercises. The text in between the exercises serves a purpose: it contains explanations and hints. Make sure you understand everything you read, ask for help otherwise.

**Do all exercises** that are not marked “optional”. Not everything will be checked or signed off; it is in your own interest to also do those parts that are *not* signed off. These are often (kinds of) questions that may also be asked in the test.

**Have everything at hand** when having your solutions signed off, including tests you ran and the resulting outcomes.

### 2.3.1 Linear search

Suppose you get a printed list of student numbers, and you want to check if your number is on it. How would you do this? The most obvious way is to scan the list from beginning to end and to compare every

element of the list with your own number. This is called *linear search* (because you through the list in a straight line from beginning to end), and it is an example of an *algorithm*.

Writing a computer program largely consists of choosing and implementing algorithms, but the algorithms are themselves independent of the programming language you happen to be using. One can in fact give a “pure” description of an algorithm, without sticking to any actual programming language: this can aid understanding because the incidental syntax and design choices of the language do not play a role any more. This leads to so-called *pseudo-code*. In Algorithm 1 you will find the pseudo-code for linear search.

**Algorithm 1:** Linear search in an arbitrary list

**input** : List *data* and value *val*  
**output**: The index of *val* in *data*, or  $-1$  if *val* does not occur in *data*

```


1 Assign first index of data to i;
2 while i valid index in data and  $data[i] \neq val$  do
3   Increment i by one;
4 end
5 return If i is not a valid index in data then -1, otherwise i

```

To understand Algorithm 1, you have to know what is meant by a *list* and an *index*. A list is no more than a sequence of values — for instance, numbers or words. Those values can then be numbered in the order they occur in the list; this number is called the *index*. For historic reasons, numbering here often starts at 0 (rather than at 1); this is also the case in PYTHON.

Below you see an example list consisting of 8 words, with corresponding indexes 0 through 7. If we call the entire list *data*, then  $data[0]$  stands for the first element (“For”),  $data[1]$  for the second (“historic”), etc. In general,  $data[i]$  is the *i*-th element, where  $0 \leq i < 8$ . Note that the index is surrounded by *square brackets*, and not the more usual parentheses — we will meet those later as well.

Value	Index
For	0
historic	1
reasons	2
numbering	3
often	4
starts	5
at	6
0	7

 **2.1** Study Algorithm 1 until you completely understand it, for instance through a set-by-step calculation of what happens if you search for the words “often” and “never”. What result does the algorithm return for those cases? □

In the next exercises, your task is to program the linear search algorithm. As announced, in this pearl we use the programming language PYTHON, but we will use only a tiny part of this language. If you want to know more of the language, you can find all you want to know on the Internet — a good place to start is the Wikipedia page of PYTHON.

*Please note:* if you already know PYTHON well, you probably know there are a lot of shortcuts that make all the things we do here much easier. *Do not use those shortcuts* and stick to the fragment of PYTHON used in this pearl. The advanced language features partially or completely hide the algorithms that this pearl is about, so using them defeats the learning goals.

The practical exercises make use of a number of pre-defined PYTHON files (so-called “modules”). These are text files with the file extension `.py` which are treated as separate units by the PYTHON interpreter.

The following text fragment from the file `search.py` is a PYTHON function for linear search; in other words, it is the PYTHON version of the pseudo-code in Algorithm 1. A *function* is the appropriate way to package such an algorithm and give it a name. After defining a function, one can use the name to *call* the function, so that the algorithm is actually executed.

```

1 def linear(data, value):
2     """Return the index of 'value' in 'data', or -1 if it does not occur"""
3     # Go through the data list from index 0 upwards
4     i = 0
5     # continue until value found or index outside valid range
6     while i < len(data) and data[i] != value:
7         # increase the index to go to the next data value
8         i = i + 1
9     # test if we have found the value
10    if i == len(data):
11        # no, we went outside valid range; return -1
12        return -1
13    else:
14        # yes, we found the value; return the index
15        return i

```

To understand this piece of code, you need to know the following about PYTHON:

- **def** (line 1) starts a function definition, of a function named `linear`. The names between parentheses, `data` and `value`, are *parameters* of this function: they correspond to the structures this function can use to do its job.
- The second line (starting and ending with three quotes) describes the purpose of the function. This is documentation meant for a programmer who wants to *use* (i.e., call) the function; the computer completely ignores this line.
- Lines starting with `#` are comments. These are meant for a programmer who wants to *understand the code* of the function; these lines are also ignored by the computer.
- A line of the form `var = expression` (lines 4 and 8) is an *assignment* and means “*var becomes something*”: `var` is the name of a *variabile* which receives the value of the *something* (see below for a short explanation of what a variable is).
- A line of the form **while** `condition`: followed by a block of more deeply indented lines means that, if `condition` holds (i.e., is true), the whole block is executed; this is repeated again and again until `condition` does *not* hold.
- A line of the form **if** `condition`: followed by a block of more deeply indented lines means that, if `condition` holds, the whole block is executed, but *in contrast to while* just a single time.
- A line of the form **else**: followed by a block of more deeply indented lines belongs to the **if** just in front (on the same level of indentation as the **else**); the block is executed if the `condition` of that **if** did *not* hold.
- A line of the form **return** `expression` means that the function is finished, with the value of `expression` as its outcome.
- The signs “`<`” and “`!=`” (line 6) and “`==`” (line 10) each compare two values: the first variant is true if the value to the left of the sign is smaller than that to the right; the second variant is true if the values are not equal, the third if they are equal. *Do not confuse the third sign == with the single equality symbol, “=”: the latter is used for assignment, see above!*

In general, a *variable* is a name chosen by the programmer, which has an associated value. In PYTHON, a new value is associated to a variable by an assignment to that variable, of the form `var = expression` (“*var becomes expression*”), as discussed above. An example is `i = 0` in line 4: after executing this line, the variable `i` has the associated value 0. Moreover, a variable can be *used*, in an *expression* such as `data[i]` in line 6; here, `data` and `i` are both variables, the first of which is associated with a list and the second with a number: the expression as a whole stands for the value at index `i` of the list `data`, again as discussed above.


In the function above there are altogether three variables: `data`, `value` and `i`.

**data**: a list of elements (numbers, words, ...) in which we are searching. As explained above, the elements of a list have an index, starting at 0 for the first element. The element with index `x` can

be accessed by `data[x]` — again the index should be surrounded by square brackets. The length of the list (i.e., the number of elements) can be queried by `len(data)`. Note that this uses parentheses rather than square brackets; if we would write `len[data]`, this would mean that we expect `len` to be a list and `data` an index expression!<sup>1</sup> It follows that the last element of a list always has index `len(data)-1`.

**value:** the value to be looked up in the list. If `data` is a list of words, `value` also has to be a word; if `data` is a list of numbers, `value` has to be a number; et cetera.

**i:** the index in `data` where the search currently takes place. This index is initialised to 0 (line 4), and subsequently incremented (line 8); if it is larger than or equal to `len(data)` it is no longer a valid index (lines 6 and 10).

 **2.2** Execute the following on the computer. Call the PYTHON-interpreter from a command line window from the directory where you have put your PYTHON-files. Make sure that you use PYTHON version 3!<sup>2</sup>

(a) Try out the function `linear`. For instance, type

```
import search
search.linear([4, 2, 3, 2], 2)
search.linear([4, 2, 3, 2], 5)
search.linear(["a", "short", "sentence"], "sentence")
search.linear(["a", "short", "sentence"], 2)
```

The first line imports the module `search` containing the function `linear`, which has the effect that from that moment onwards, that function is known to the interpreter and can be used; the next lines each time call `linear` with different lists and search values.

(b) Explain the output of the interpreter in the previous step.

□

## Terminology

**Definition, call.** Functions such as `linear` are first *defined* and then *called*.

**Parameters.** In both a definition and a call, the function name is followed by parentheses surrounding a — possibly empty — comma-separated list of names (in case of a definition) and expressions (in case of a call); these are called *parameters*.


**Formal parameters.** In a definition, the parameters (in that case sometimes called *formal parameters*) are always *variables*, such as `data` and `value` in the case of `linear`.

**Actual parameters, arguments.** In a call, the parameters (in that case sometimes called *actual parameters* or *arguments*) can be arbitrary expressions, such as `[4, 2, 3, 2]` and `2` in `2` but also `data` in `len(data)`.

If you are getting bored having to type `search.linear` each time, instead use

```
from search import linear
linear([4, 2, 3, 2], 2)
```

It is one thing to understand a function that has already been defined, but another to modify it or write a function yourself. We will start with a few modifications in `linear`.

 **2.3** Using a text editor, change the function `linear` in the way described below. To use the modified module (i.e., PYTHON file) in your running interpreter, you first have to type

```
import imp
imp.reload(search)
```

(`imp` is a standard module of PYTHON that defines the function `reload`.) Apply the following changes, and test their effect:

<sup>1</sup>In fact, `len(data)` is a call to a function named `len`, with `data` as argument. In other words, `len` is the same kind of thing as the function `linear` we just defined, but this particular function is already defined in PYTHON by default.

<sup>2</sup>If you never did this before, find out how to open a command line window. The PYTHON version can be seen from the message you get when you call the interpreter. Under Linux (and therefore also on an Apple computer) you have to use the command `python3` to get the correct version of PYTHON.

- (a) Change the outcome in case the searched value does not occur in the list into 'does not occur'.
- (b) Start looking at index 1 instead of 0. Obviously, this introduces an error; what exactly goes wrong?
- (c) Continue up to *and including* `len(data)` (using `<=`) rather than *up to* (using `<`). What error does this introduce?
- (d) Search the list in reverse order. Does this always yield the same result?

□

When you have executed the same sequence of commands a few time, this too starts to be boring. You can alternatively write a PYTHON script to test the function `linear`, and invoke this script rather than the interpreter.

- ☞ **2.4** In a text editor, create a file `searchtest.py`, containing precisely the commands you used above to test `linear`, but now as an argument to the standard PYTHON function `print`: for instance

```
from search import linear
print(linear([4, 2, 3, 2], 2))
print(linear([4, 2, 3, 2], 5))
print(linear(["a", "short", "sentence"], "sentence"))
print(linear(["a", "short", "sentence"], 2))
```

(The `print`-function ensures that the result of calling `linear` is actually sent to the command line window.) Invoke this script from the command line using “`python searchtest.py`”.

If everything went right, you can now see the output of the commands in the script, but not the commands themselves. This is not always optimal for understandability. You can also use `print` to add some explanatory text to the output; for instance

```
from search import linear
print("Search 2 in [4, 2, 3, 2]; expected outcome = 1")
print(linear([4, 2, 3, 2], 2))
```

When invoking the script, you can now immediately see if the result of the function meets the expectation.

□

## 2.3.2 Binary search

If you would look for a word in an arbitrary book by a linear search, this would last forever. No one does that. But in case of a dictionary or telephone guide (if you know what those are) it used to be the whole purpose to find a word or name quickly! Fortunately, a lot of profit can be had because the words in a dictionary and the names in a telephone guide are *alphabetically ordered*. In an ordered list you can do a *binary search* instead of a linear one, which is much more efficient, as we will see.

Before going into this: even linear search can be improved if you assume that the list `data` is ordered. To wit, if at any point you notice that the element at `data[i]` is *larger* than the looked-for `value`, you can conclude that `value` does not occur in the list and you can stop searching!

- ☞ **2.5** Create a new module `ordsearch`, again containing a function `linear`, which is improved in the way described above. Test this function by writing a script `ordsearchtest`. Also show (in that test script) that the function can give wrong results if you use it on an argument list that is not ordered. □

- ☞ **2.6** What happens if you use `ordsearch.linear` to search for a number in a list of words, or vice versa? Explain the observed effect. □

Although linear search in an ordered list is a light improvement with respect to an unordered list (in case the searched word is not in the list), this does not really help much when the list contains a hundred thousand elements, as a dictionary might. Other solutions are fundamentally more efficient.

- ☞ **2.7** Imagine how you yourself would look for a word in a dictionary (or a name in a telephone book), and formulate this as an algorithm, in natural language. Use numbered lines to describe the steps in your algorithm, and when needed use terms such as “go to line *x*”. □

When searching in an ordered list, you can think of the *interval* within the list (given by a lower and upper index bound) within which the target value is certain to lie, if it is in the list at all. Initially, the lower bound and upper bounds are simply the lowest and highest indices in the list (which are those?). The interval is each time made shorter by *choosing the middle* and inspecting the value at that index. (If the interval contains an even number of elements, “the middle” can be either of the two central ones; it does not matter which one.) This inspection can result in any of three possible outcomes:

1. The element in the middle is exactly the target value; then we are done.
2. The element in the middle is larger than the target value; then the upper bound should be adjusted to the index just before the middle.
3. The element in the middle is smaller than the target value; then the lower bound should be adjusted to the index just after the middle.

After the upper has been adjusted (step 2), it may be the case that it has become smaller than the lower bound; and vic versa for the adjustment of the lower bound (step 3). In that case the interval within which we are searching has actually become *empty*, meaning that the target value does not occur in the list at all.

The above described the so-called *binary* search algorithm, called thus because there are always two ways to continue searching — left of the middle or right of the middle. In Algorithm 2 you will find the pseudo code of this algorithm.

**Algorithm 2:** Binary search in an ordered list

```

input : Ordered, non-empty data and target value val
output: The index of val in data, or  $-1$  if val does not occur in data

1 Assign to low and high the first and last index in data;
2 while interval between low and high non-empty and  $data[low] \neq val$  do
3   Choose mid in the middle between low and high ;
4   if  $data[mid] = val$  then
5     Assign to low the value of mid ;                               /* Found! */
6   else if  $val < data[mid]$  then
7     Assign to high the value of  $mid - 1$ ;                          /* val lies before index mid */
8   else
9     Assign to low the value of  $mid + 1$ ;                            /* val lies after index mid */
10  end
11 end
12 return if  $data[low] = val$  then low, otherwise  $-1$ 

```

- 🔗 **2.8** Add a function `binary` to the module `ordsearch`, containing the binary search algorithm according to Algorithm 2 in PYTHON. Test your implementation.

*Nota bene.* To program line 3 of Algorithm 2 you should take the average of *low* and *high*; this involves a division by 2. If you divide an odd number by 2 (for instance by typing  $n/2$  where  $n$  equals 5) this will result (in PYTHON) in a real number (in this example 2.5). You cannot use that as an index value; instead, you need to truncate it (in this example to 2). To get the truncated value you can either use  $n//2$  (where `//` is the PYTHON operator for integer division) or `int(n/2)` (where `int` is a standard PYTHON function that truncates a real number to an integer number). □

- 🔗 **2.9** Suppose you have a list of 200 elements, and you look for a value that eventually turns out to occur at index 62. Which are the values assigned to *low*, *high* and *mid* before the element is found? □

To test your algorithm more extensively, the pre-defined files of this practical include a dictionary `Unabr.dict` containing more than 200000 words. In order to use this file as the first parameter of `linear` or `binary`, it should first be converted to a list usable by PYTHON. For this purpose, the pre-defined files also include a module `util.py` containing two functions:


**lines(filename):** returns a list containing all individual lines in the file  
**words(filename):** returns a list containing all individual words in the file

You can now for instance type (in the interpreter):

```
>>> from ordsearch import binary
>>> from util import lines
>>> binary(lines("Unabr.dict"), "eagle")
-1
>>> binary(lines("Unabr.dict"), "zygose")
213492
```

(Check that “eagle” indeed does not occur in the dictionary!)


We have already claimed that binary search is more efficient than linear search. That claim should be supported both analytically and experimentally. For this purpose we take the most time-consuming case, namely that the target word does *not* occur in the list.

 **2.10 Analysis of the efficiency of linear and binary search.** Let us say that executing the `while`-block takes  $w$  seconds — in practice just a microsecond or less — and to make things easier, let us not make a distinction between the `while` in `linear` and `binary`.

- (a) Suppose that searching a non-existent word in a given list takes  $x$  seconds. Now we make the list twice as long.
  - How much time does a linear search take now?
  - How much time does a binary search take now?
- (b) Can you give a mathematical function that expresses how much time searching for a non-existent word takes, in terms of the length of the list?
  - For linear search;
  - For binary search.

□

To test the difference in efficiency experimentally, the lab files also include a script `searchmeasure.py`. This script measures the time for linear search versus binary search of a word in microseconds. The measurement is not very accurate, but good enough for our purpose. Call the script from the command line using “`python searchmeasure.py Unabr.dict`”.

 **2.11 Experimental observation of the efficiency of linear and binary search.**


- (a) Study the script `searchmeasure.py` and make sure you understand what goes on. Run the script and try out a number of input words.
- (b) What is the smallest time difference between linear and binary search that you can observe?
- (c) What is the largest time difference between linear and binary search that you can observe?

□

## 2.3.3 Sorting

The above clearly shows that it can be advantageous to work with sorted lists. This, then, makes it important to be able to quickly sort an (initially unsorted) list. In the next exercises you will meet two sorting algorithms.

The first of the two is Algorithm 3. This algorithm is called *bubble sort* to convey the corresponding mental image: larger values in the list “bubble” upwards, towards the end of the list. To understand in more detail what the algorithm does, you can best try it out manually using a (small) example.

 **2.12 Manual execution of bubble sort.** What happens if you perform Algorithm 3 on a list *data* with the following content (note that the list elements are now placed next to each other instead of on top of each other):

<b>Index:</b>	0	1	2	3
<b>Value:</b>	2	4	1	3

**Algorithm 3:** Ordering a list using *bubble sort*


```

input : List data
output: Sorted copy of list data
1 Assign a copy of data to res ;                               /* res is the result list */
2 Assign the last index in res to ui ;                       /* ui is the index of the last unsorted value */
3 while ui > 0 do                                           /* If ui equals 0, everything is sorted */
4   Assign 0 to si ;                                         /* si is the index of the last swapped value */
5   Assign 0 to i ;                                         /* i is the index that is currently processed */
6   while i < ui do                                         /* If ui is reached then the rest is ordered */
7     if res[i] > res[i + 1] then                             /* res[i] en res[i + 1] are inverted */
8       Swap res[i] and res[i + 1] ;
9       Assign i to si ;                                     /* We just swapped at index i */
10    end
11    Increment i by one;
12  end
13  Assign si to ui ;                                       /* The list is ordered from si+1 */
14 end
15 return res

```

Answer the question by manually going through the steps of the algorithm, and each time drawing the state of the variables. □

If you can answer the following question correctly, you are a long way towards understanding this algorithm well.

 **2.13 Analysis of the efficiency of bubble sort.** Consider the performance of the algorithm in the most extreme cases:

- How does it perform when applied to a list that is already correctly ordered?
- How does it perform when applied to a list that is entirely in reverse order?


In particular, in both of the above scenarios, answer the following questions:

- How often is the **while** in line 3 executed?
- How many times the **if** in line 7?
- How does the time taken by the algorithm depend on the length of the list?

□

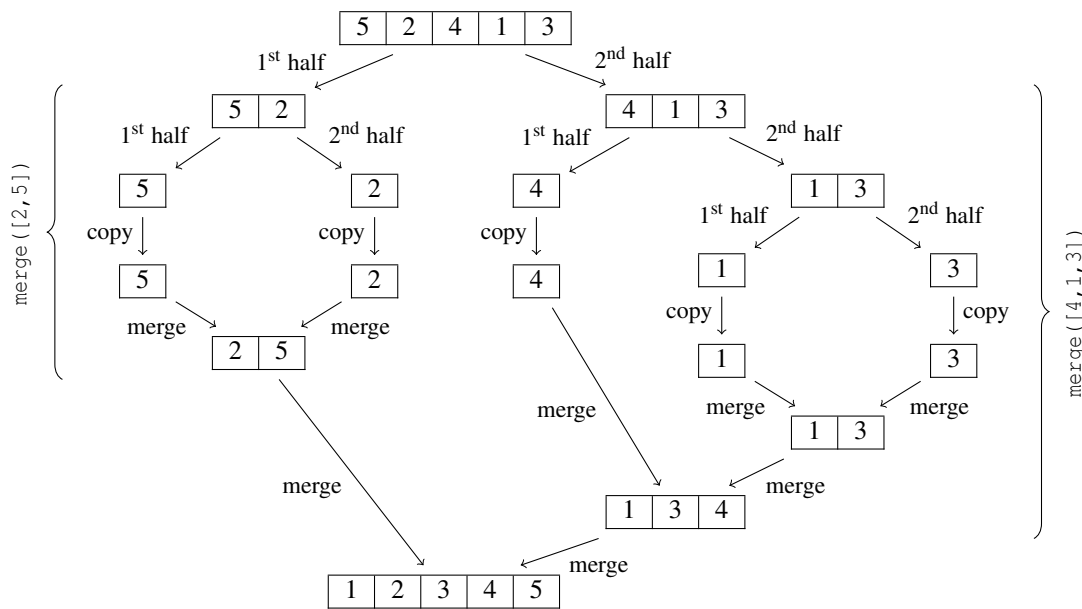
To program bubble sort in PYTHON, you need to know a few more things.

- To copy a list (line 1) in PYTHON, say the value of *data*, use *data[:]* as seen above; for instance, *result = data[:]* will assign a copy of the list *data* to *result*. The fact that it is a *copy* means that afterwards, *data* is unchanged when *result* is modified, and vice versa.
- To swap two values (line 8) in PYTHON, use a so-called *simultaneous assignment*: for instance, *x, y = a, b* has the effect that afterwards *x* has the value of *a* and *y* the value of *b*. This is especially useful in a construction such as *x, y = y, x* which swaps the values of *x* and *y*.

 **2.14** Program a PYTHON module *sort* with a function *bubble* implementing Algorithm 3. Test your implementation in the interpreter, with a few simple lists of numbers but also by sorting the words in the lab files *Unabr.dict* and *hacktest.txt*. (Use the previously discussed function *util.words* for converting these files to lists of words.) Which file takes more time to sort? Can you explain this? □

Bubble sort does not belong to the fastest class of sorting algorithms — that is, unless the list was already sorted to begin with. A smarter sorting algorithm can save as much time as a smarter searching algorithm. Many of the smarter algorithms use *recursion*: the original problem is first solved for a smaller list, and that solution is used within to solve the original problem.





This shows that the list is split each time into “1<sup>st</sup> half” and “2<sup>nd</sup> half” (lines 4–5 in Algorithm 4), which then are first sorted themselves (lines 7–21). Convince yourself you understand how the merging works, for instance by merging  $[2, 5]$  and  $[1, 3, 4]$  into  $[1, 2, 3, 4, 5]$  step by step, using pen and paper.

**2.15 Manual execution of merge sort** Visualise the execution of  $\text{merge}([4, 3, 1, 6, 2, 5])$  in the same way as above. □

To program merge sort in PYTHON, you need to know a few more things.

- In lines 4, 5, 18 and 20 you have to take a *fragment* of a list; not the entire list and not just a single element. Use `data[:i]` to copy the fragment from `data` from the beginning *up until* (but not *including*) index `i`, and `data[i:]` for the fragment from (including) `i` up until *and including* the last element. As we have already seen, `data[:]` also produces a copy of the entire list.<sup>3</sup>
- In lines 10 and 13 you have to append an element to a list. Use the standard PYTHON function `data.append(elem)` to append the element `elem` to the end of `data`.
- In lines 18 and 20 you have to glue a list to the end of another (existing) list. Use `data.extend(rest)` to glue the list `rest` to the end of `data`.

To properly understand these expressions, you can always try them out in the PYTHON interpreter.

**2.16** What is the difference between `a.append(b)` and `a.extend(b)`? First think what the answer should be and then try it out, for the cases

- `a=[1,2]` and `b=3`
- `a=[1,2]` and `b=[3,4]`

□


**2.17** Program merge sort in PYTHON (in the module `sort`) and test your result. (You should be able to show your tests during sign-off.) □

We will now again look at efficiency in practice. As you have already seen from the script `searchmeasure.py` (Question 2.11), you can measure the time a given PYTHON-command takes by using the function `time.process_time()`, in the following way:

```
# import standard module with clock function
import time
# measure start and end time when executing command
start = time.process_time(); command; end = time.process_time()
```

<sup>3</sup>Expressions of the form `data[i:j]` also exist, but we do not need them here.


```
# subtract start from end time; multiply by 10^6 and round to get micros
duration = round((end - start) * 1000000)
```

 **2.18 Experimental observation of the efficiency of bubble sort and merge sort.** Compare (by trying out) the execution time of bubble sort and merge sort:

- (a) Using `hacktest.txt`
- (b) Using `Unabr.dict`

What are your observations? Can you explain them?

The next question is tricky: at this moment in time you have not yet learned how to answer questions of this kind systematically (this will only be taught in the second year), but maybe you have an intuition for it. Do not spend too much time on this question now, if you get stuck with it; it is optional and does not need to be signed off.

 **2.19 Analysis of merge sort (optional).** If a list has  $n$  elements, then merge sort takes (in the order of)  $n \log_2 n$  steps to sort it — where “a step” may be an assignment or a test in PYTHON. Can you think of an argument why this is the case?

Through the combination of sorting and searching in a *sorted* list, we can now also more quickly search in an (initially) *unsorted* list: after importing `util`, `ordsearch` and `sort` you can perform the following call:

```
ordsearch.binary(sort.merge(util.words("grail.txt")), "swallow")
```

This sorts the words in `"grail.txt"` and searches the sorted list for the word “swallow”.


 **2.20 Fast search in an unsorted list.**

- (a) The search method explained above is *not* faster than linear search! Explain why not.
- (b) If we want to find multiple words in an unsorted list, then it may be faster to first sort the list. From approximately how many target words does this start to be the case, assuming the list contains 10000 words?

## 2.3.4 Duplicates and tuples

Before you can start with the pearl assignment, there are a few more concepts that will be useful.

Firstly: if you sort a realistic text file, it is immediately obvious that a lot of words occur more than once. In some cases you may want all those occurrences to be in the sorted list, but often it is better to keep only one copy of each word. The simplest solution is then to remove the duplicates from the list, simply by creating a new list to which you copy a single instance of each word. Algorithm 5 shows (in pseudo-code) how that can be achieved.

 **2.21** Program Algorithm 5 in PYTHON, as function `remove_dups` in a new module `dup`, and test your result.

 **2.22 Analysis of Algorithm 5.**

- (a) Write a small piece of PYTHON code that determines the number of different words in `hacktest.txt` and `grail.txt`.<sup>4</sup>
- (b) If you use this algorithm on an unsorted list, not all duplicates are removed. (Test this behaviour.) How do you explain this, and how would you describe what does happen?

<sup>4</sup>Note that `util.words` considers punctuation to be part of a word, and that uppercase and lowercase letters are considered to be different. For instance, “you”, “You” and “you?” are different words. You may just work with this outcome and do not have to “clean” it.

**Algorithm 5:** Removing duplicates

```

input : Sorted list data
output: Copy of data in which each value occurs exactly once

1 Assign an empty list to res;
  ;
2 if data is not empty then
3   Assign data[0] to fresh;
4   Assign 1 to i;
5   while i is a valid index in data do
6     if data[i] does not equal fresh then
7       Append fresh to res;
8       Assign to fresh the value of data[i];
9     end
10    Increment i by one;
11  end
12  Append fresh to res;
13 end
14 return res

```

The second new concept is work with lists not just containing simple values such as numbers or words, but containing *tuples*. A tuple is itself also a list, with a fixed length. A tuple of two elements is often called a *pair*.<sup>5</sup>


Since for the notion of a list it does not matter what kind of elements are in it, it is straightforward to construct and use lists of lists (or tuples). The only difference is that the order of the elements is now arranged somewhat differently. In case of numbers or words, you can test the ordering with

```

if data[i] < data[i+1]:
  do something

```

but if the elements of *data* are tuples, then you should ask yourself what “<” means.

 **2.23** What is the correct order for the following tuples:

```

["a", 3]  ["a", 2]  ["b", 1]  ["ab", 10]

```

□


When asked this question, your first response should be: what do you mean by “correct”? What ordering should we use? The most natural answer (but certainly not the only one) is to use the so-called *lexicographical* sorting criterion:  $[x_1, y_1]$  is smaller than  $[x_2, y_2]$  if either  $x_1 < x_2$ , or  $x_1 = x_2$  and  $y_1 < y_2$ .<sup>6</sup> If *data*[*i*] is itself a pair, then the elements of that pair can be addressed by *data*[*i*][0] and *data*[*i*][1]. To test the lexicographical order, we therefore have to use:

```

if ( data[i][0] < data[i+1][0]
  or data[i][0] == data[i+1][0] and data[i][1] < data[i+1][1]):
  do something

```


(Note the parentheses around the condition: these are required because there is a newline between the sub-conditions.)

 **2.24** Implement a function `sort.merge_pairs` that sorts a list of pairs lexicographically. Test your program, for instance with the list of Question 2.23. □

<sup>5</sup>This is a white lie: those who know PYTHON also know that tuples and lists are actually different in that language — the main difference being that tuples are immutable and are denoted with round rather than square brackets.

<sup>6</sup>It so happens that in PYTHON, “<” is actually defined over lists such that it implements the lexicographical order.

When *searching* in a list of pairs, another thing comes into play. Sometimes you might want to search only on the basis of the first element of the pair — for instance, if the list consists of pairs of student numbers and grades, and you do know your own number but not your grade.

 **2.25** Implement a function `ordsearch.binary_pairs` that, given an ordered list of pairs and a value, looks for this value *as first element of a pair* and returns the index of that pair in the list, or `-1` if the value does not occur in the list. For example:

```
>>> x = [{"Anne", 7}, {"John", 5}, {"Pete", 9}]
>>> ordsearch.binary_pairs(x, "Brenda")
-1
>>> ordsearch.binary_pairs(x, "John")
1
```


□

### 2.3.5 Pearls assignment: Build a search engine

You have now learned enough to carry out this week's pearl assignment. The assignment is about searching a given word in a set of text files — the basic functionality of search engines such as Google (apart from the fact that the text files have already been collected and do not have to be reaped from web sites all over the world).

An example collection of text files has been included in the material of this week; these are the files with extension `*.txt` on Blackboard

The first step is to read in the text files. This step has been pre-defined: the function `util.all_word_pairs()` returns a list of pairs, each consisting of a word from a text file together with the name of that file. Given this list, your task is to efficiently search for a word and return all the files in which it occurs.

 **2.26 Basic assignment.** Implement a new module `pearl` containing a function `make_table(pairs)`, which is invoked with as parameter a list such as the one returned by `util.all_word_pairs()`, and returns a new, sorted list — the so-called *search table* — with pairs consisting of:

- As first element a word (de potential search target)
- As second element a list of files in which this word occurs.

Every word should appear in the search table only once, and the corresponding list of files should not contain duplicates. *The order of the files in the list is not important in this basic assignment.* □


For instance, if we start with two files:<sup>7</sup>

- `brian.txt` containing only the words “eunt” (2 times) and “dead” (1 time)
- `grail.txt` containing only the words “dead” (3 times) and “spank” (7 times)

then the result of `pearl.make_table` should be the following list:

```
[ [ "dead", ["brian.txt", "grail.txt"] ],
  [ "eunt", ["brian.txt"] ],
  [ "spank", ["grail.txt"] ]
]
```

To test your result, there is a pre-defined PYTHON script `google.py`, which first calls `util.all_word_pairs` and `pearl.make_table` in succession, and then repeatedly asks for a search term, which is looked up in the search table. *Study this script to understand what is being asked: your solution to the basic assignment is only approved if `google.py` is error free.*

 **2.27 Bonus assignments.** The following extensions of the basic assignment can earn you additional grade points::

<sup>7</sup>*This is an example!* The files `brian.txt` and `grail.txt` on Blackboard contain the text of the two well-known movies and are therefore obviously much more extensive.

- (a) +0.5 *punt*. Program a variation `pearl.make_counted_table(pairs)` which for every search term does not list the files in which it occurs in random order, but in decreasing order of the number of times the word occurs in the file. That number may itself be part of the list. For instance, the result of `make_counted_table` for the example above would be:

```
[ [ "dead", [{"grail.txt", 3}, {"brian.txt", 1}] ],
  [ "eunt", [{"brian.txt", 2}] ],
  [ "spank", [{"grail.txt", 7}] ]
]
```

- (b) +0.5 *punt*. program a variation `pearl.make_density_table(pairs)` which lists the files in decreasing order of *frequency* — in other words, the number of occurrences divided by the total number of words in the file. The frequency itself may be part of the list. For instance, the result of `make_freq_table` for the example above would be:

```
[ [ "dead", [{"brian.txt", 0.333}, {"grail.txt", 0.3}] ],
  [ "eunt", [{"brian.txt", 0.666}] ],
  [ "spank", [{"grail.txt", 0.7}] ]
]
```

*Note: this second bonus question is a good deal more tricky than the first, because you have to have some way of recording how many words there are each text file in total!*

□

If you already have some expertise in PYTHON or in programming in general: do not hesitate to implement more functionality! For instance, maybe you want to allow searching for more than one target term at the same time.

**What should you hand in?** On Blackboard, submit a zip file containing:

- A text file called `README.txt`, describing
  - Whose solution this is: your name and student number (of both students if you did the assignment together)
  - What you implemented: just the basic assignment (Question 2.26) or also the bonus assignments (Question 2.27); in the latter case, which extensions you did precisely.
  - What you did to test your solution; that is, which commands you tried out and what the outcome was.
- The PYTHON module `pearl.py` as described in Question 2.26 and possibly 2.27, plus other PYTHON code you produced yourself insofar necessary to make your solution work. *Add comments to your code to make it understandable.*
- (Not required, but desirable.) A PYTHON-script in which your own test commands (see first bullet) have been pre-programmed.

**Assessmen** The main criteria for assessment are summarised below.

- A solution not containing the required elements (see above) — for instance, which does not contain the `README` file or does not say anything about testing your own solution — will not be approved.
- A solution of which the code is insufficiently documented runs a high risk of not being approved.
- A solution which passes your own tests (according to your `README`) but which turns out to contain errors upon further testing will only be approved if the handed-in code is decently documented and it turns out to be possible to trace and repair the error easily.

The algorithm Google uses to order the pages containing a given search term is secret. In any case, it is by no means as simple as in this pearl assignment: it is *not* just a question of counting how many times a word occurs! If you are interested, look up the term *page rank* (in Google!) — and look up *keyword stuffing* to understand why word frequencies are not sufficient on their own.

The reason why Google's algorithm is secret, is that there are enormous economic interests involved: every company would like to appear at the top of the screen! Research has shown that only very few people ever browse to the next page after executing a search command. If you would know Google's sorting criterion, you could analyse what is necessary to appear higher on the result page. In fact, improving the findability of web sites is itself booming business by now.

Conclusion: if you really master the concepts of searching and sorting, you have taken an important step in your career!

## 2.4 Example test questions

Here you will find a set of sample questions of the kind that may appear in the written test; answers are given in Section 2.4.2. Note that this is *not* a sample test: in a real test, the questions are chosen such that they are divided evenly over the topics of the lab, and can be answered in 60 minutes.<sup>8</sup>

In addition, the following questions from the lab may also appear in a test:

- Question 2.6 on the difference between numbers and text
- Question 2.7 on searching in a dictionary
- Question 2.9 on the precise execution of binary search
- Question 2.10 on the efficiency of linear and binary search
- Question 2.12 on manually executing bubble sort
- Question 2.13 on the efficiency of bubble sort
- Question 2.15 on manually executing merge sort
- Question 2.20 on efficient search in an unsorted list.

An important point about the test is that you will not be asked to produce large pieces of PYTHON code (though some ability to produce small code fragments is expected), but you will be asked to analyse small algorithms in PYTHON that you have not seen before. Also, you are expected to understand how to use lists, and how the various searching and sorting algorithms encountered during the practical work.

You do not have to independently carry out computations such as the ones in Question 2.20 and 2.19; but you should understand the quantitative difference between algorithms needing  $\log_2 n$  steps,  $n$  steps,  $n \log_2 n$  steps or  $n^2$  steps to process a list of  $n$  elements.

### 2.4.1 Questions

1. Suppose you have a PYTHON list `list` known to contain 3 elements; for instance `[10, 4, -1]`.
  - (a) Write a condition in PYTHON-syntax which tests if this list is entirely sorted in reverse (which is indeed the case for the example list above, but for instance not for `[10, 4, 7]`).
  - (b) Write the minimal PYTHON assignments necessary to make `list` correctly sorted (from small to large), assuming that it starts out being sorted in reverse (and still contains 3 elements).
2. Explain under which circumstances linear search in an ordered list can be faster than binary search.

Analyse the following PYTHON code:

<sup>8</sup>A real test (with solutions) can be found on Blackboard.

```

1 def insertsort(data):
2     # result list, starts out empty
3     result = data[:]
4     # bound is index of first unsorted element
5     grens = 1
6     # continue until the end of the list
7     while grens < len(result):
8         # elem is element we will now sort into place
9         elem = result[bound]
10        # go back from bound
11        index = bound
12        # look for the correct place of elem
13        while index > 0 and elem < result[index-1]:
14            # go back until correct place for elem is found
15            # shift elements to the right to make place
16            result[index] = result[index-1]
17            index = index - 1
18        # insert elem into the right place
19        result[index] = elem
20        bound = bound + 1
21    return result

```

3. State what happens at the call `insertsort([4,2,3,1])`, by giving the values of `grens`, `elem` and `result` each time line 10 is reached.
4. How many steps does this algorithm need if `data` is a completely unsorted list of  $n$  elements: approximately (“in the order of”)  $n$ , approximately  $n \log_2 n$  or approximately  $n^2$  steps? Explain your answer.

At home you have 100 LPs,<sup>9</sup> which you lend out to your friends on a regular basis. When an LP is returned, you put it back unseen. One fine day, you are fed up with having to search for LPs in your unsorted collection.

For the questions below, explain your answers.

5. How many LP sleeves do you have to look at *on the average* to find a specific one, if your collection is unsorted and does contain the one you are looking for?
6. Describe the algorithm you would use to sort your LPs in natural language. As in Question 2.7, formulate your algorithm in small steps, use numbered lines to describe the steps, and when needed use terms such as “go to line  $x$ ”.
7. How many LP sleeves do you have to look at *at the most* to find a specific one, if your collection is sorted and does contain the one you are looking for, and you search as efficiently as you can?

## 2.4.2 Answers

1. (a) `lijst[0] > lijst[1] and lijst[1] > lijst[2]`  
 (b) `lijst[0], lijst[2] = lijst[2], lijst[0]` (gelijktijdige toewijzing)
2. Linear search can be faster if the target value is “small”, meaning that it is or should be near the start of the list. In that case, it is found (or observed not to exist) after a few iterations only.

We can make this more precise: binary search takes in the order of  $\log_2 n$  steps, so linear search is faster if the target value or a larger one is found within the first  $\log_2 n$  elements of the list. *This more precise analysis is not required during the test!*

3. The values are in the following table.

<sup>9</sup>An LP is a round, usually black disk with a spiral groove on both sides, into which sound is encoded mechanically.

bound	elem	data
1	2	[4, 2, 3, 1]
2	3	[2, 4, 3, 1]
3	1	[2, 3, 4, 1]

The final result is [1,2,3,4].

4. The algorithm requires approximately  $n^2$  steps: the outer **while**-loop is iterated  $n - 1$ , and the inner loop on the average  $b/2$  times where  $b$  is the bound, i.e., the number of already sorted elements;  $b$  is therefore itself on the average  $n/2$ . This results in the execution of lines 16 and 17 roughly  $(n - 1)\frac{n}{4} = \frac{n^2 - n}{4}$  times, which is in the order of  $n^2$ . (During the test, it is not expected that the calculation is done as precisely as this, but it is expected that the outer and inner **while**-loops are distinguished and it is realised that the number of steps is the product of the iteration counts of those two loops.)
5. On the average you have to look at 50.5 LP sleeves, assuming that for every LP the chances that it is borrowed are equally large. Formally, this number is determined by summing up the number of compared sleeves for each LP in the collection and dividing the total by 100:

$$\left(\sum_{i=1}^{100} i\right)/100 = \frac{100}{2} \times 101/100 = \frac{101}{2} = 50,5$$

(The exact calculation is not required — that requires knowing the Gauss-formula  $\sum_{i=1}^n i = \frac{1}{2}(i+1)$  — but the number 50 does not get full marks. The consideration on the distribution of probabilities may be omitted.)

6. There are many ways to do this, but a credible algorithm is::
- Make a pile of sorted LPs
  - Take the next unsorted LP
  - Look for the position in the pile where this LP belongs. (This can be done using binary search.)
  - Put the new LP into its position, causing the pile to be 1 higher
  - If there are still unsorted LPs, return to line 6b.

It is, of course, inadvisable to pile LPs, so after you are done you should put them straight at once!

7. In the worst case, the LP you look for is in (for instance) the second position (index 1). Then the index you will look at ( $m$  in the binary search algorithm) is subsequently

49    24    11    5    2    0    1

The number of comparisons is then 7. You can reason that it can never be more than 7, since  $2^7 = 128 > 100$  (or in other words  $7 > \log_2 100$ ), so after 7 comparisons you have covered all positions. (Again, the calculation does not need to be done this precisely, but the answer 7 and the connection to powers of 2 should be mentioned.)

