

COMP.CS.300 Data structures and algorithms 1

Bibliography

These lecture notes are based on the notes for the course OHJ-2016 Utilization of Data Structures and TIE-20106 Data structures and algorithms. All editorial work is done by Terhi Kilamo and the content is based on the work of Professor Valmari and lecturer Minna Ruuska. Further additions by Matti Rintala.

Most algorithms are originally from the book Introduction to Algorithms; Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

In addition the following books have been used when completing this material:

- Introduction to The Design & Analysis of Algorithms; Anany Levitin
- Olioiden ohjelmointi C++:lla; Matti Rintala, Jyke Jokinen
- Tietorakenteet ja Algoritmit; Ilkka Kokkarinen, Kirsti Ala-Mutka

1 Introduction

Let's talk first about the motivation for studying data structures and algorithms

Algorithms in the world

1.1 Why?

What are the three most important algorithms that affect YOUR daily life?

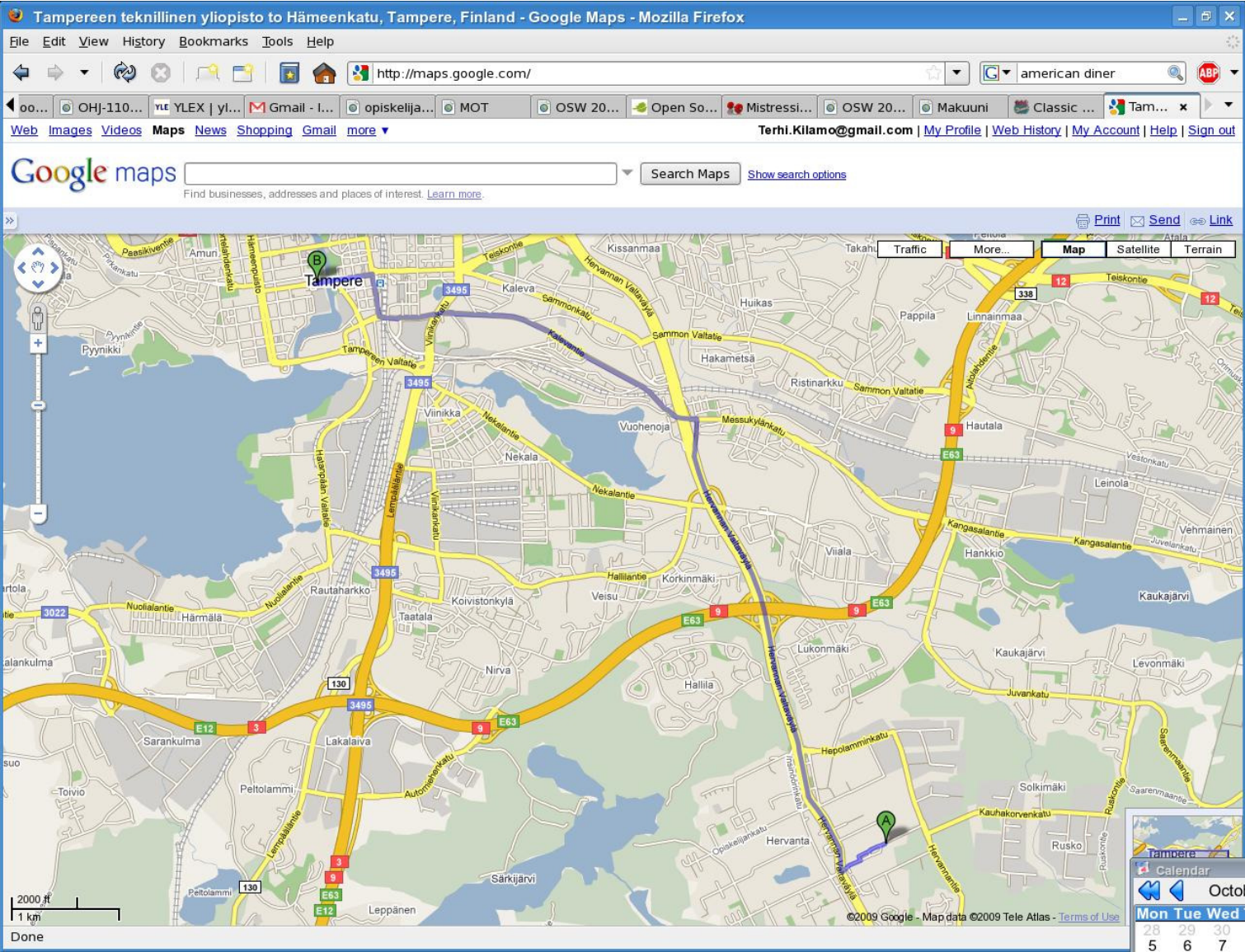


Picture: Chris Watt

There are no computer programs without algorithms

- algorithms make for example the following applications possible:





The image shows the momondo website interface for flight searches. The header includes the momondo logo and navigation links for flights, hotel, car rental, and holiday rentals. The main section is titled "Compare cheap flights and hotels" with a subtext: "Tell us where you're going and we find the best prices on flights and hotels. Enjoy your trip! Psst ... we're a free service, not a travel agency and we don't add any booking fees." There is a CNN TRAVEL+LEISURE logo and a "New! Price guarantee" banner.

The flight search section is active, showing "Flights" selected. The trip type is "Return Trip". The search parameters are:

- From: Helsinki (HEL), Finland
- To: Madrid (MAD), Spain
- Departure date: 03/20/2014
- Return date: 03/24/2014
- Adults: 1
- Children: 0
- Ticket Class: Economy

A date selection calendar is open, showing February 2014 and March 2014. The calendar is titled "SELECT YOUR END DATE". The date 03/24/2014 is highlighted in the calendar.

FEBRUARY 2014							MARCH 2014				
Wk	Su	Mo	Tu	We	Th	Fr	Sa	Wk	Su	Mo	Tu
5							1	9			
6	2	3	4	5	6	7	8	10	2	3	4
7	9	10	11	12	13	14	15	11	9	10	11
8	16	17	18	19	20	21	22	12	16	17	18
9	23	24	25	26	27	28		13	23	24	25
10								14	30	31	

algorithms are at work whenever a computer is used

Data structures are needed to store and access the data handled in the programs easily

- there are several different types of data structures and not all of them are suitable for all tasks
 - ⇒ it is the programmer's job to know which to choose
 - ⇒ the behaviour, strengths and weaknesses of the alternatives must be known

Modern programming languages provide easy to use library implementations for data structures (C++ standard library, JCF). Understanding the properties of these and the limitations there may be for using them requires theoretical knowledge on basic data structures.

Ever gotten frustrated on a program running slowly?

- functionality is naturally a top priority but efficiency and thus the usability and user experience are not meaningless side remarks
- it is important to take memory- and time consumption into account when making decisions in program implementation
- using a library implementation seems more straightforward than is really is

This course discusses these issues

2 Terminology and conventions

This chapter covers the terminology and the syntax of the algorithms used on the course.

The differences between algorithms represented in pseudocode and the actual solution in a programming language is discussed. The sorting algorithm INSERTION-SORT is used as an example.

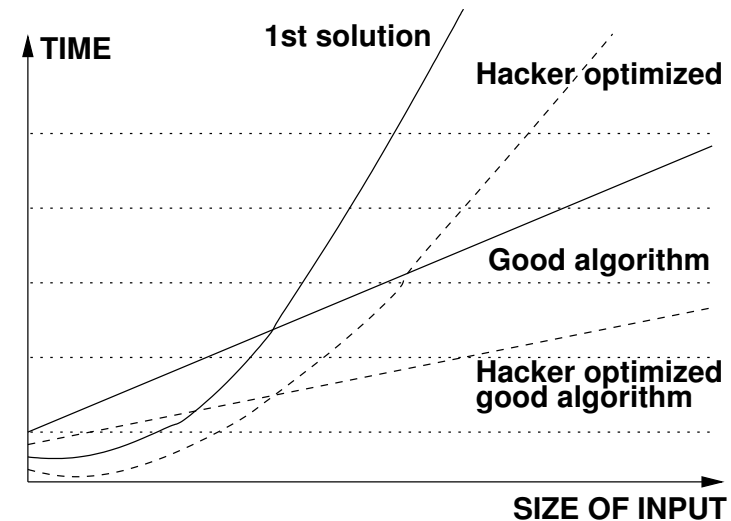
2.1 Goals of the course

As discussed earlier, the main goal of the course is to provide a sufficient knowledge on and the basic tools for choosing the most suitable solution to a given programming problem. The course also aims to give the student the ability to evaluate the decisions made during the design process on a basic level.

The data structures and algorithms commonly used in programming are covered.

- The course concentrates on choosing a suitable data structure for solving a given problem.
- In addition, common types of problems and the algorithms to solve them are covered.

- The course concentrates on the so called “good algorithms” shown in the picture on the right.
- The emphasis is on the time the algorithm uses to process the data as the size of the input gets larger. Less attention is paid to optimization details.









2.2 Terminology

A *data structure* is a collection of related data items stored in a segment of the computer's memory.

- data can be added and searched by using suitable algorithms.
- there can be several different levels in a data structure: a data structure can consist of other data structures.

An *algorithm* is a well defined set of instructions that takes in a set of input and produces a set of output, i.e. it gives a solution to a given problem.

	10:20 HEL – 10:55 TKU suora 0 t 35 min		22:00 TKU – 22:35 HEL suora 0 t 35 min		331 € Varaa
<small>Tiedot</small> tai varaa sivustoilta: travelstart 331 € Travellink 335 € Opodo 335 € Eticket.fi 336 € TravelPartner 337 € finnair 342 € budget 343 € bravofly 348 € Flyhi 352 €					
	14:00 HEL – 14:35 TKU suora 0 t 35 min		17:10 TKU – 17:45 HEL suora 0 t 35 min		331 € Varaa
<small>Tiedot</small> tai varaa sivustoilta: travelstart 331 € Travellink 335 € Opodo 335 € TravelPartner 337 € finnair 342 € budget 343 € bravofly 348 €					
<small>Supersaver</small> Tarkistettu 1t sitten					

- well defined =
 - each step is detailed enough for the reader (human or machine) to execute
 - each step is unambiguous
 - the same requirements apply to the execution order of the steps
 - the execution is finite, i.e. it ends after a finite amount of steps.

An algorithm solves a well defined problem.

- The relation between the results and the given input is determined by the problem
- for example:
 - sorting the contents of the array
 - input:** a sequence of numbers a_1, a_2, \dots, a_n
 - results:** numbers a_1, a_2, \dots, a_n sorted into an ascending order
 - finding flight connections
 - input:** a graph of flight connections, cities of departure and destination
 - results:** Flight numbers, connection and price information

- an instance of the problem is created by giving legal values to the elements of the problem's input
 - for example: an instance of the sorting problem: 31, 41, 59, 26, 41, 58

An algorithm is *correct*, if it halts and gives the correct output as the result each time it is given a legal input.

- A certain set of formally possible inputs can be forbidden by the definition of the algorithm or the problem

SAS	06:15 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 7 t 25 min	21:25 TKU – 14:30 HEL 2 vaihtoa ARN,CPH 17 t 05 min (+1)	3.200 € Varaa
Tiedot	tai varaa sivustolta: Flyhi 3.216 €		Eticket.fi ✓ päivitetty
SAS	07:45 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 5 t 55 min	21:25 TKU – 23:10 HEL 2 vaihtoa ARN,CPH 25 t 45 min (+1)	3.200 € Varaa
Tiedot	tai varaa sivustolta: Flyhi 3.216 €		Eticket.fi ✓ päivitetty
SAS + KLM	06:15 HEL – 13:40 TKU 2 vaihtoa ARN,CPH 7 t 25 min	21:25 TKU – 23:55 HEL 2 vaihtoa ARN,AMS 26 t 30 min (+1)	3.605 € Varaa
Tiedot	tai varaa sivustolta: Flyhi 3.621 €		Eticket.fi ✓ päivitetty
SAS + KLM	07:55 HEL – 13:40 TKU 1 vaihto CPH 5 t 45 min	21:25 TKU – 23:55 HEL 2 vaihtoa ARN,AMS 26 t 30 min (+1)	3.683 € Varaa
Tiedot	tai varaa sivustolta: Flyhi 3.699 €		Eticket.fi ✓ päivitetty

an algorithm can be incorrect in three different ways:

- it produces an incorrect result
- it crashes during execution
- it never halts, i.e. has infinite execution

an incorrect algorithm may sometimes be a very usefull one as long as a certain amount of errors is tolerated.

- for example, checking whether a number is prime

In principle any method of representing algorithms can be used as long as the result is precise and unambiguous

- usually algorithms are implemented as computer programs or in hardware
 - in practise, the implementation must take several “engineering viewpoints” into account
 - accomodation to the situation and environment
 - checking the legality of inputs
 - handling error situations
 - limitations of the programming language
 - speed limitations and practicality issues concerning the hardware and the programming language
 - maintenance issues \Rightarrow modularity etc.
- \Rightarrow the idea of the algorithm may get lost under the implementation details

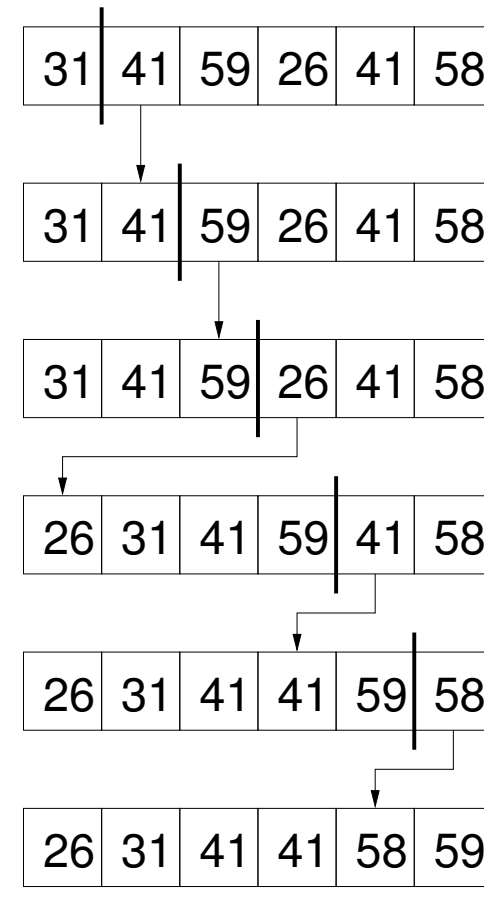
On this course we concentrate on the algorithmic ideas and therefore usually represent the algorithms in pseudocode without legality checks, error handling etc.

Let's take, for example, an algorithm suitable for sorting small arrays called INSERTION-SORT:



Figure 1: picture from Wikipedia

- the basic idea:
 - during execution the leftmost elements in the array are sorted and the rest are still unsorted
 - the algorithm starts from the second element and iteratively steps through the elements upto the end of the array
- on each step the algorithm searches for the point in the sorted part of the array, where the first element in the unsorted range should go to.
 - room is made for the new element by moving the larger elements one step to the right
 - the element is placed to it's correct position and the size of the sorted range in the beginning of the array is incremented by one.



In pseudocode used on the course INSERTION-SORT looks like this:

```
INSERTION-SORT(A)                                (input in array A)
1  for j := 2 to A.length do                    (increment the size of the sorted range)
2      key := A[j]                                  (handle the first unsorted element)
3      i := j - 1
4      while i > 0 and A[i] > key do (find the correct location for the new element)
5          A[i + 1] := A[i]                          (make room for the new element)
6          i := i - 1
7      A[i + 1] := key                              (set the new element to its correct location)
```

- indentation is used to indicate the range of conditions and loop structures
- (*comments*) are written in parentheses in italics
- the “:=” is used as the assignment operator (“=” is the comparison operator)
- the lines starting with the character ▷ give textual instructions

- members of structure elements (or objects) are referred to with the dot notation.
 - e.g. *student.name*, *student.number*
- the members of a structure accessed through a pointer x are referred to with the \rightarrow character
 - e.g. $x \rightarrow name$, $x \rightarrow number$
- variables are local unless mentioned otherwise
- a collection of elements, an array or a pointer, is a **reference** to the collection
 - larger data structures like the ones mentioned should always be passed by reference
- a pass-by-value mechanism is used for single parameters (just like C++ does)
- a pointer or a reference can also have no target: NIL

2.3 Implementing algorithms

In the real world you need to be able to use theoretical knowledge in practise.

For example: apply a given sorting algorithm ins a certain programming problem

- numbers are rarely sorted alone, we sort structures with
 - a *key*
 - *satellite data*
- the key sets the order
 - ⇒ it is used in the comparisons
- the satellite data is not used in the comparison, but it must be moved around together with the key

The INSERTION-SORT algorithm from the previous chapter would change as follows if there were some satellite data used:

```
1  for  $j := 2$  to  $A.length$  do  
2       $temp := A[j]$   
3       $i := j - 1$   
4      while  $i > 0$  and  $A[i].key > temp.key$  do  
5           $A[i + 1] := A[i]$   
6           $i := i - 1$   
7       $A[i + 1] := temp$ 
```

- An array of pointers to structures should be used with a lot of satellite data. The sorting is done with the pointers and the structures can then be moved directly to their correct locations.

The programming language and the problem to be solved also often dictate other implementation details, for example:

- Indexing starts from 0 (in pseudocode often from 1)
- Is indexing even used, or some other method of accessing data (or do we use arrays or some other data structures)
- (C++) Is the data really inside the array/datastructure, or somewhere else at the end of a pointer (in which case the data doesn't have to be moved and sharing it is easier). Many other programming languages always use pointers/references, so you don't have to choose.
- If you refer to the data indirectly from elsewhere, does it happen with
 - Pointers (or references)
 - Smart pointers (C++, `shared_ptr`)
 - Iterators (if the data is inside a datastructure)
 - Index (if the data is inside an array)
 - Search key (if the data is inside a data structure with fast search)

- Is recursion implemented really as recursion or as iteration
- Are algorithm "parameters" in pseudocode really parameters in code, or just variables

In order to make an executable program, additional information is needed to implement INSERTION-SORT

- an actual programming language must be used with its syntax for defining variables and functions
- a main program that takes care of reading the input, checking its legality and printing the results is also needed
 - it is common that the main is longer than the actual algorithm

The implementation of the program described above in C++:

```
#include <iostream>
#include <vector>
typedef std::vector<int> Array;

void insertionSort( Array & A ) {
    int key, i; unsigned int j;
    for( j = 1; j < A.size(); ++j ) {
        key = A.at(j); i = j-1;
        while( i >= 0 && A.at(i) > key ) {
            A.at(i+1) = A.at(i); --i;
        }
        A.at(i+1) = key;
    }
}

int main() {
    unsigned int i;
    // getting the amount of elements
    std::cout << "Give the size of the array 0...: "; std::cin >> i;
```

```
Array A(i); // creating the array
// reading in the elements
for( i = 0; i < A.size(); ++i ) {
    std::cout << "Give A[" << i+1 << "]: ";
    std::cin >> A.at(i);
}
insertionSort( A );    // sorting

// print nicely
for( i = 0; i < A.size(); ++i ) {
    if( i % 5 == 0 ) {
        std::cout << std::endl;
    }
    else {
        std::cout << " ";
    }
    std::cout << A.at(i);
}
std::cout << std::endl;
}
```

The program code is significantly longer than the pseudocode. It is also more difficult to see the central characteristics of the algorithm.

This course concentrates on the principles of algorithms and data structures. Therefore using program code doesn't serve the goals of the course.

⇒ From now on, program code implementations are not normally shown.