Algorithms -
the peak of computing science

Peter Damaschke
ptr at chalmers.se

and

Erland Holmström
erland at chalmers.se

Autumn **2017**

CTH TIN093 / GU DIT602

Course home page:
www.cse.chalmers.se/edu/course/tin093

Lecture 1 – Introduction

Reading reference: Chap 1+2 + slides

- Introduction, homepage, literature mm
- What's it all about?
- A problem to a program –
  example: traffic lights
- How to describe algorithms
- Pseudocode

Prerequisites:

- programming knowledge (Java, C, C++, ...)
- a course on data structures, incl. recursion
- elementary discrete mathematics
  including proofs, logarithms, sums, ...

Next lecture:

Analyzing complexity, slides + chap 2
(important subject +
not very good description in book)

This are copies of my slides - not lecture notes.

Lot's of stuff appears only on the blackboard and you may not be able to fully appreciate the slides without that.

(Jag trycker dem för att de som är på föreläsningen skall slippa (försöka) anteckna

Detta är alltså inte föreläsninganteckningar utan saknar mycket av det som sägs, allt det som skrivs på tavlan och tex de färger som finns på OH bilderna. Är man på föreläsning kan man komplettera med detta själv. Jag trycker så att alla som kommer på föreläsning skall kunna få kopior. När det är slut där så är det slut men materialet finns ibland också på hemsidan.)

Literature:

Algorithm Design: Kleinberg, Tardos

(book is also used in the adv. Alg. course)

On the net price seem to be ≈ 556-1542:-

Programming assignments

with electronic submissions

You work in groups of ?? student

Preliminary schedule

Reading week	1	2	3	4	5	6	7	8
lectures	1	2	2	2	2	2	2	0
exercises	-	1	1	1	1	1	1	{1}
assignments	-	1	1	1	1	1	1	-
supervision		?	?		?			?

Exam in the end of reading week 8

re-exam in december

+ exam also in maj, august

check helping aids before each exam

Problem to Computer program

1. Understand the problem, exactly what is the solution supposed to do?
2. Formulate/specify the problem, in a mathematical model.

   (Half the job is to understand what problem to solve and what model to use.)
3. Design (and describe) one (or several) algorithms and corresponding data structures.

   Informal solutions in pseudocode.

   Understand why you might not succeed and what to do then.
4. Prove it correct (or incorrect)
5. Analyze its resource requirements
6. Develop it more, e.g. make it: more efficient, use less memory, easier to understand...

   ---
7. Implement the algorithm (Chose computer language etc.)
8. Document, test... forever :-)

(Compare with G. Polya: How to solve it 1945)

Goals with the Algorithms course

You will be able to

- **Model problems** Formulate a clear mathematical model of real world problems.
- **Design algorithms** Construct algorithms with several algorithm design methods like divide&conquer, dynamic programming, greedy, backtrack and be able to choose appropriate data structures and abstractions.
- **Describe** your algorithms and their qualities
- **Prove correctness** of your algorithms
- **Analyze** algorithms: perform an objective evaluation of the performance and be able to compare it to other algorithms performance
- **Develop** your algorithm - make it better
- **Recognise intractable problems** and other classes of problems like P, NP, NPC

You will also know something about

- **Problem analysis**: how to analyze the complexity of a problem e.g. how much resources are **required** to solve a problem.
- **Approximation algorithms**: how to solve problems with exponential complexity.

In short: what you should do before you start programming on a computer to ensure that what you get afterwards is good quality.

Algorithmic ideas are pervasive...
in computer science and beyond

- Some of the major shifts in internet routing standards
- The secret of life - bioinformatics: How do your DNA gene sequence determine you?
- Economics talking about electronic auctions, eBay
- Hypersearching the Web: How do you search the web for information?
- Aircraft crew scheduling
- Testing for reliability of hardware and software

- search the web for "algorithm" and find out (Approx. 127 000 000 results in 0,56 seconds)

Google - how do they do it?

Founder Larry Page: "The perfect search engine" is defined as something that

- "understands exactly what you mean and
- gives you exactly what you want"

1998:

25 million pages (million== 10^6, billion = 10^9)

"Yesterday":

25 billion pages, 1,3 billion pictures

- **Google File System** - a datastructure for files
- **Distributed hardware** - Many computer centers around the world with thousands of rack mounted low-price PCs
- **MapReduce** - server software that automatically split jobs on different computers
- **PageRank** - an algorithm to give a page a "rank"
- **Algorithms** for analyzing hypertext-matchings

What is an Algorithm? (A)

- Intuitive: a method to solve a problem
- A precise, unambiguous and executable set of instructions that (usually) are supposed to terminate (supposed to be "computerised").
- Rules for computation

What's then a Problem? (Π)

- A general question to be answered or
- a description of the goal to achieve, usually with unknown parameters.

You describe a problem by describing its input and what characteristics the solution must fulfill.

All input given => an Instance (I)

By specifying all parameters you get an **instance** of the problem.

An algorithm **solves** a problem if it can be applied to **every** instance of the problem and always gives a solution.

An algorithm like quicksort solves the problem of sorting and a problem usually has many known algorithms.

Problem - Instance - Algorithm

Problem 1:

Given is a graph with n nodes and e edges (description of parameters) how many colors are needed to color the graph if two adjacent nodes must not have the same color (requirements on the solution)

Instance: You get an instance of the problem by specifying all nodes and edges of the graph.

Algorithm: perhaps the greedy one we will see in a moment

Problem 2:

Given is an array with n numbers (description of parameters), sort the numbers in increasing order (requirements on the solution).

Instance: You get an instance of the problem by specifying all the numbers.

Algorithms: quicksort, mergesort, bubblesort,...

Problem to Computer program
Let's look at an example without digging into to many details

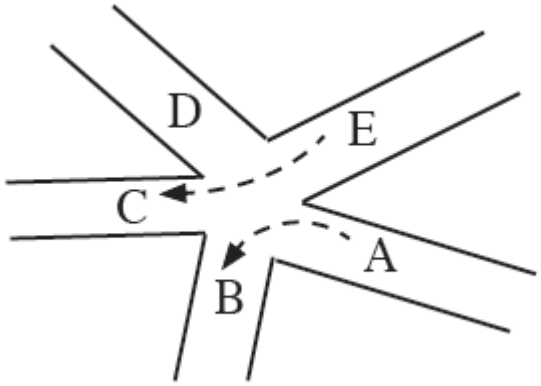
Problem: Given are a set of allowed turns and which of them that can't be performed simultaneously.

How many phases must the light in a traffic crossing have and which turns can happen simultaneously?

Allowed turns:

{AB,BC,DA,EA,AC,BC,DB,EB,AD,BA,DC,EC,ED}

Can't happen simultaneously: AD-EB, AC-DB,...



Ex: AB and EC are possible simultaneous turns

How to model a problem

- It's a "simplification" and formalization of the real world/problem being solved:
how to represent the problem in the "solution domain"?

  - A good model represent the world in a suitable way for solving the problem at hand.
  - A model is a partial rather than a complete representation, compare with a caricature.

    It means that one should eliminate all unnecessary information not relating to the problem that is being analyzed.

And - a model that is inadequate under one set of circumstances may be the best that you can come up with under another set of circumstances.

- The models quality depends on the question.

A book can be modeled/represented by:

- a list of the chapters (descriptive)
- its thickness
- by the formula T = t*p where p=number of pages and t the time it takes to read a page. (mathematical, predictive)

Q: How long time to read a book?

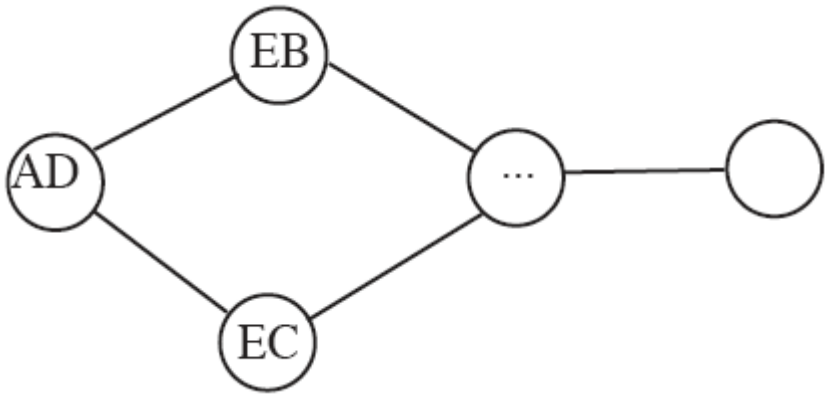
Q: How many books can we fit in a bookshelf?

- Time and accuracy influence your model.
- A model impose what you can/can't do.
- Models are useful in their own right.

they allow for a formal definition of a problem and can be useful in thinking about a problem, they are "a laboratory for the imagination".

Mathematical model

We can use a graph:



Let nodes represent turns and arcs represent turns that can't happen simultaneously

We need to find a solution with as few phases as possible with simultaneous turns.

This actually turns out to be a very well known problem in computer science, coloring of a graph, belonging to a very large class of problems called "NPC".

Unfortunately, no one has found a polynomial algorithm to solve any one of them!

Let's try a greedy approach...

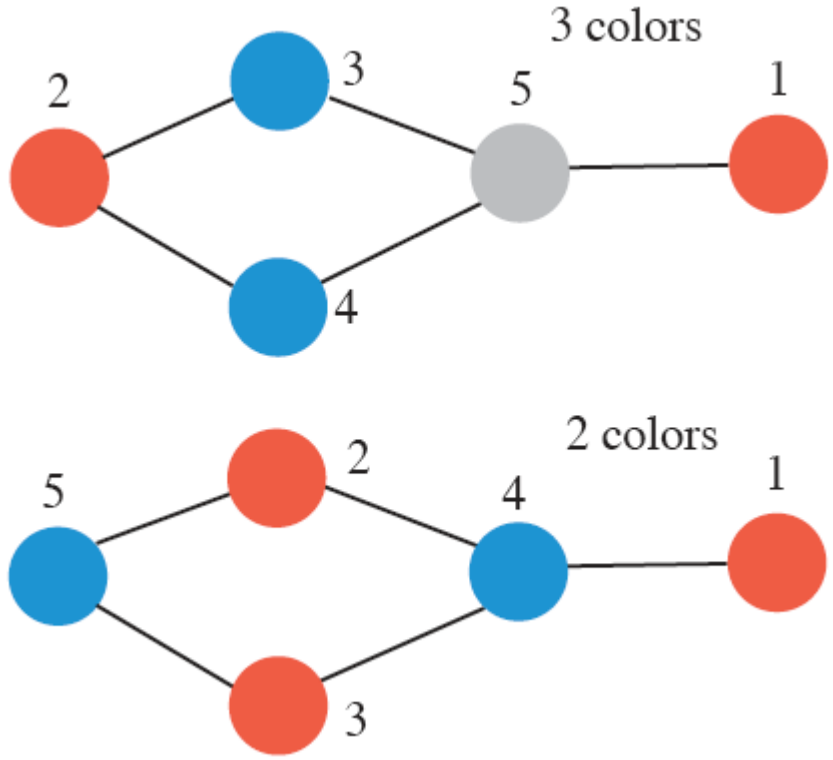
How to present algorithms

1. **Explain how it works**
   Use words, high level pseudocode and illustrations. The important thing is to reveal the underlying idea and as a help to understand the pseudocode in the next step. A "dry swim" of some steps is very good here.
2. **Give an abstract algorithm in pseudocode**
   This is mixture of programming language, mathematics, English, Swedish and other suitable notation. You can use abstract data types (for instance graphs, sets, lists...) and sentences like "for every node w on EL(v) loop"
3. **Prove that it works**
4. Analyze **its complexity**
5. **Describe the different parts** of your algorithm if that is needed for step 4 or 3 (To show that your algorithm can be implemented and to be able to analyze it in step 4.)
6. Implement more details

During this course you are expected to do step 1+2+3+4(+5) where (+5) is needed only if needed in step 4 or 3. Exceptions to this are always explicitly stated like "you only need to do step x".

How the algorithm works

Try to color as many nodes as possible with the first color, then as many as possible with the second color and so on.



This is a so called greedy algorithm. It colors without looking at the consequences later on. In this case it will not always (seldom?) find the optimal solution.

It delivers an approximate solution in short time.

Pseudocode

Pseudocode should:

- give a (human-)readable description of the algorithm and its structure
- be precise enough so that it can be understood, analyzed and translated to programming language code - but without to many details
- balance between understandability and precision

Generally you must describe your algorithm in several steps so it is important to find good abstractions and to divide the algorithm in parts that can be described separately.

Not very good pseudocode (not precise enough) handed in to describe mergesort:

"Split in two equal parts, put them together so that they are sorted, repeat recursively".

Pseudocode

```
color(graph: G) return set
    set: newclr = ∅
    for every uncolored node v in G loop
        if v isn't connected to any
                other node in newclr then
            mark v colored
            add v to newclr
        end if
    end loop
    return newclr
end color
```

This algorithm are called several times until all nodes are colored, then the empty set is returned. Returned at every call are a set of possible simultaneous turns.----------

We also need to describe the data structures (graph and set) and their performance (to be able to analyze our algorithm), explain how to do things like "`mark v colored`" etc., prove that the algorithm works and analyze its resource requirements. More on this later.

Example: Good level of pseudocode

This is Kruskals algorithm for computing minimum spanning trees from Horowitz, Sahni, Rajasekaran:

Computer Algorithms 1998, page 224

```
// E is the set of all edges in the graph,
// t is the growing minimum spanning tree
t = Ø
while ( t has < n-1 edges and E≠Ø ) do
   Chose an edge (v,w) from E
                              of lowest cost
   Delete (v,w) from E
   if (v,w) does not create a cycle in t
      add (v,w) to t
   else
      discard (v,w)
}
```

This is a very good start, it's readable, has visible structure, but many things are missing. I.e. it needs developing a bit further.

- What are the inputs to the algorithm?
- How are the graph and the set represented?
- How to find the edge of "lowest cost"
- How to check if there is a cycle?

Develop the algorithm

- How are the graph and the set represented?

Let the input to the algorithm be the set of edges (could be a list) and the cost of the edges (a cost matrix, construct this before the algorithm starts) and don't care more about the graph representation here.

- What is the minimum cost?

Let the algorithm return the minimum cost **and** the mst. Or compute afterwords.

- How to find the edge of "lowest cost"?

Place the edges in a priority queue. Then it's easy to find the minimum in log n time.

- How to check if there is a cycle?

The cycle part is perhaps a bit tricky. Use Merge - Find sets to hold the connected components constructed by Kruskals

Kruskals algorithm more details, same source

```
algorithm Kruskal (E, cost, n, t)
   // E is the set of edges in G with n vertices
   // cost(u,v) is the cost of edge (u,v),
   // t is the set of edges in the mst,
   // the final cost is returned
 Construct a heap out of the edge cost
 Each vertex is in a different set
   i := 0, mincost := 0.0;
   while((i<n-1) and (heap !empty)) do
      Delete a minimum cost edge (u,v)
      from the heap and reheapify
      j := Find(u); k := Find(v);
      if (j ≠ k) then
         i := i+1;
         insert((u,v), t);
         mincost := mincost+ cost[u,v];
         Union(j,k);
      }
   }
   if (i≠n-1) then
      write("No span. tree");
   else return mincost;
   }
```

Still some parts are missing (heap), that's alright since we know enough to analyze the algorithm. Some parts are perhaps to detailed, like counting (i, mincost)

Next lecture:
How do we evaluate algorithms?

How good is an algorithm?

How to make i better?

Does it work?

How do we compare two algorithms?

Is it optimal (in speed, memory etc.)

How do we evaluate algorithms?

1. It must be correct - it must work.
2. Must be easy to understand, code, maintain etc.
3. Should use computer resources well (speed, memory,...) e.g. should be "effective"