

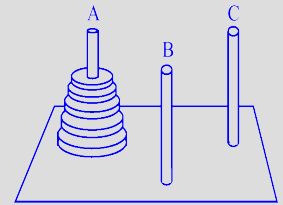
Data Structures and Algorithms

Week 11

1. Tractable and intractable problems.
2. What is a "reasonable" running time?
3. NP problems, examples
4. NP-complete problems
5. Polynomial reducibility

Towers of Hanoi/1

- **Goal:** transfer all n disks from peg A to peg C
- **Rules:**
 - move one disk at a time
 - never place larger disk above smaller one
- **Recursive solution:**
 - transfer $n - 1$ disks from A to B
 - move largest disk from A to C
 - transfer $n - 1$ disks from B to C
- **Total number of moves:**
 - $T(n) = 2T(n - 1) + 1$



DSA09

M. Böhlen

2

Towers of Hanoi/2

- **Recurrence:**
$$T(n) = 2 T(n-1) + 1$$
$$T(1) = 1$$
- **Solution by repeated substitution:**
$$T(n) = 2 (2 T(n - 2) + 1) + 1 =$$
$$= 4 T(n - 2) + 2 + 1 =$$
$$= 4 (2 T(n - 3) + 1) + 2 + 1 =$$
$$= 8 T(n - 3) + 4 + 2 + 1 = \dots$$
$$= 2^i T(n - i) + 2^{i-1} + 2^{i-2} + \dots + 2^1 + 2^0$$
- **The expansion stops ($n - i = 1$) when $i = n - 1$**
$$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$$

Towers of Hanoi/3

- This is a **geometric series** so that we have
$$T(n) = 2^n - 1 = O(2^n)$$
- The running time of this algorithm is **exponential** (k^n).
- **Good or bad news?**
 - the Tibetan priests were confronted with a tower problem of 64 rings.
 - Assuming the priests move **one ring per second**, it would take **~585 billion years** to complete the process.

DSA09

M. Böhlen

3

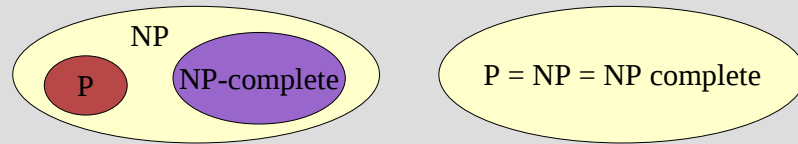
DSA09

M. Böhlen

4

Overview

- Problem class overview:
 - P: Problems solvable in polynomial time
 - NP: Problems verifiable in polynomial time
 - NPC: "reference problems" in NP (can be used to solve the other NP problems)
- Most important open problem in theoretical computer science.
- Clay institute of mathematics offered 1 million dollar millenium prize for solution!



DSA09

M. Böhlen

5

Decision Problems

- Are long running times linked to the size of the solution of an algorithm?
 - No! To show this we consider only TRUE/FALSE or yes/no problems in the following – **decision problems**
- We can usually transform an *optimization problem* into an **easier decision problem**:
 - *Optimization problem O* : “Find a shortest path between vertices u and v in a graph G .”
 - Related *decision problem D* : “Determine if there is a path between vertices u and v in a graph G shorter than k .”
 - If we have an easy way to solve O , we can use it to solve D .
 - If we show that D is hard, we know that O must be hard as well.

DSA09

M. Böhlen

6

LCS

- Longest common subsequence
 - $x = \text{"sariempiolcew"}$
 - $y = \text{"westigmupsalrte"}$
- Solution: Proceed from the end of the strings and
 - If $x_m = y_n$ append symbol to $\text{LCS}(x_{m-1}, y_{n-1})$
 - If $x_m \neq y_n$
 - Skip last symbol from x or
 - last symbol from y
 - Decide which symbol to skip by comparing $\text{LCS}(x_m, y_{n-1})$ and $\text{LCS}(x_{m-1}, y_n)$

DSA09

M. Böhlen

7

LCS Implementation/1

```
int lcsRec(int i, int j) {
    if (i==0 || j==0) return 0;
    else if (x[i] == y[j]) return lcsRec(i-1, j-1) + 1;
    else return max(lcsRec(i-1, j), lcsRec(i, j-1));
}
```

Recurrence for time complexity:

$$\begin{aligned} T(2n) &= 2T(2n-1) \\ &= 2(2T(2n-2)) \\ &= 4T(2n-2) \\ &= 4(2T(2n-3)) \\ &= 8T(2n-3) \\ &= 2^i T(2n-i) \\ &= 2^{2n} = 4^n \end{aligned}$$

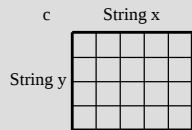
DSA09

M. Böhlen

8

LCS Implementations/2

```
int lcsMemo(int i, int j) {
    if (c[i][j] != -1) return c[i][j];
    else if (x[i] == y[j]) {
        c[i][j] = lcsMemo(i-1, j-1) + 1;
        return c[i][j];
    }
    else {
        c[i][j] = max(lcsMemo(i-1, j), lcsMemo(i, j-1));
        return c[i][j];
    }
}
```



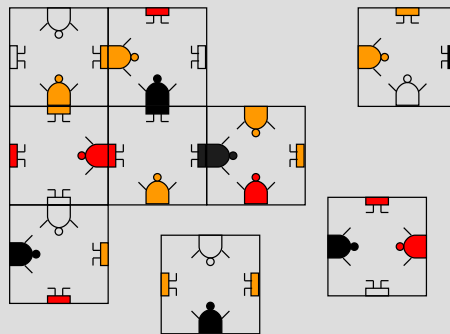
$$T(n) = O(n^2)$$

Runtime Comparison

N	Recursive	Memo
10	0.004	0.002
15	0.774	0.002
17	4.760	0.003
18	11.216	0.003
20	154.464	0.003
100		0.005

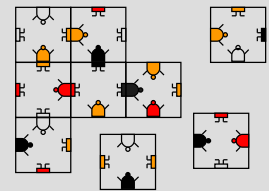
Monkey Puzzle/1

- Monkey puzzle: an example of a decision problem
- Nine square cards with imprinted “monkey halves”
- The goal is to arrange the cards in 3x3 square with matching halves.



Monkey Puzzle/2

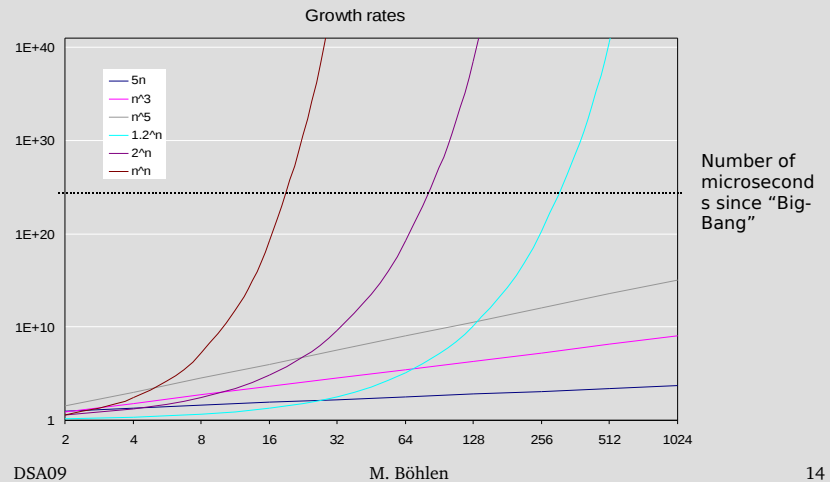
- Assumption: orientation is fixed
- Does any $M \times M$ arrangement exist that fulfills the matching criterion?
- Brute-force algorithm would take $n!$ time to verify whether a solution exists
 - assuming $n = 25$, it would take 490 billion years on a one-million-per-second arrangements computer to verify whether a solution exists



Monkey Puzzle/3

- Improving the algorithm
 - discarding partial arrangements
 - etc.
- A smart algorithm would still take a couple of thousand years in the worst case
- Is there an easier way to find solutions? MAYBE! But nobody has found them, yet!

Reasonable vs. Unreasonable/1

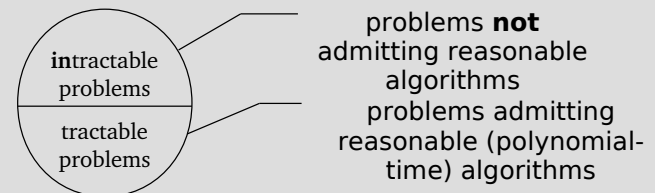


Reasonable vs. Unreasonable/2

n/ function	10	20	50	100	300	
Polynomial	n^2	1/10'000 second	1/2'500 second	1/400 second	1/100 second	9/100 second
	n^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
Exponential	2^n	1/1000 second	1 second	35.7 years	400 trillion centuries	a 75 digit-number of centuries
	n^n	2.8 hours	3.3 trillion years	a 70 digit-number of centuries	a 185 digit-number of centuries	a 728 digit-number of centuries

Reasonable vs. Unreasonable/3

- "Good", reasonable algorithms
 - algorithms bound by a polynomial function n^k
 - **Tractable problems**
- "Bad", unreasonable algorithms
 - algorithms whose running time is above n^k
 - **Intractable problems**



So What!

- Computers become faster every day
 - insignificant (a constant) compared to exponential running time
- Maybe the Monkey puzzle is just one specific problem, we could simply ignore
 - the monkey puzzle falls into a category of problems called NPC (NP complete) problems (~1000 problems)
 - all admit **unreasonable** solutions
 - **not known** to admit **reasonable** ones

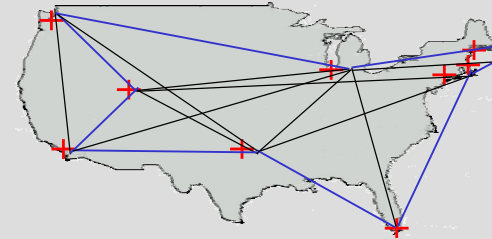
DSA09

M. Böhlen

17

Traveling Salesman Problem

- A traveling salesperson needs to visit n cities
- Is there a route of at most d length? (decision problem)
 - Optimization-version asks to find a shortest cycle visiting all vertices once in a weighted graph



DSA09

M. Böhlen

18

TSP Algorithms

- Naive solutions take $n!$ time in worst-case, where n is the number of edges of the graph
- No polynomial-time algorithms are known
 - TSP is an NP-complete problem
 - Simpler Hamiltonian Cycle problem is NP-complete
- Longest Path problem between A and B in a weighted graph is also NP-complete
 - Remember the running time for the shortest path problem

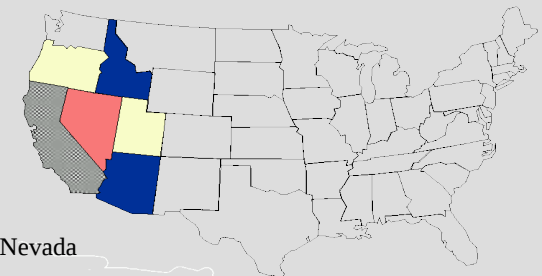
DSA09

M. Böhlen

19

Coloring Problem/1

- **3-color**: given a planar map, can it be colored using 3 colors so that no adjacent regions have the same color?



NO instance
Impossible to 3-color Nevada
and bordering states!

DSA09

M. Böhlen

20

Coloring Problem/2

- Any map can be **4-colored**
- Maps that contain no points that are the junctions of an odd number of states can be **2-colored**
- No polynomial algorithms are known to determine whether a map can be **3-colored**: it's an NP-complete problem

Determining Truth (SAT)/1

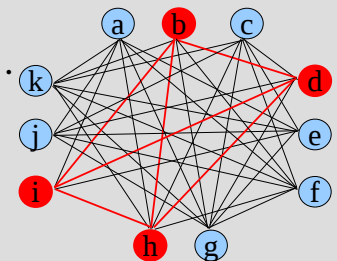
- Determine the truth or falsity of logical sentences in **propositional calculus**
- Using the logical operators (\wedge -and, \vee -or, \neg -not) we compose expressions such as the following $(\neg F \vee E) \wedge (F \vee D) \wedge (\neg E \vee F \vee \neg D)$
- The algorithmic problem calls for determining the **satisfiability** of such sentences
 - e.g., $D = \text{true}$, E and $F = \text{false}$

Determining Truth (SAT)/2

- Exponential time algorithm on $n =$ number of distinct variables ($O(2^n)$)
 - Best known solution for the general SAT problem
 - SAT problem is in the NP-complete class.
- Good performance if we only have Horn clauses (at most 1 positive literal)
 $(\neg x \vee \neg y \vee \neg z) \wedge (\neg x \vee x)$
- Good performance if all clauses have only two literals (2SAT)
 $(\neg F \vee E) \wedge (F \vee D) \wedge (\neg E \vee F)$

CLIQUE

- Given n people and their pairwise relationships, is there a group of s people (called *clique*) such that every pair in the group knows each other
 - people: a, b, c, \dots, k
 - friendships: $(a,e), (a,f), \dots$
 - clique size: $s = 4$?
 - YES, $\{b, d, i, h\}$ is a *certificate!*



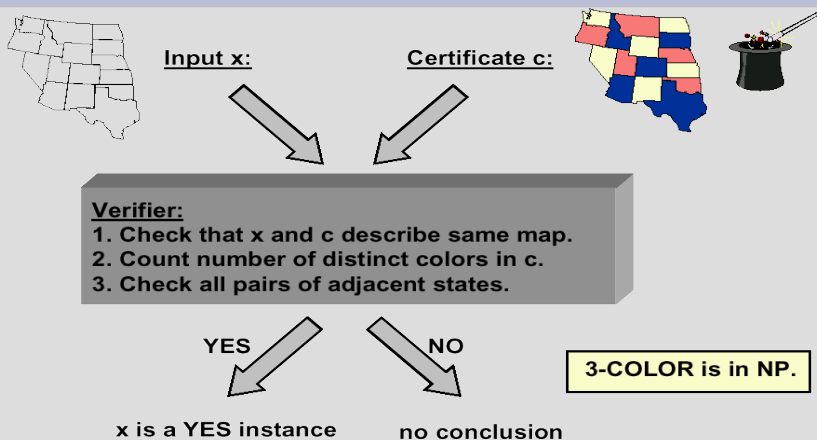
P

- Definition of P:
 - Set of all decision problems solvable in polynomial time on *real* computers
- Examples:
 - SHORTEST PATH: Is the shortest path between u and v in a graph shorter than k ?
 - RELPRIME: Are the integers x and y relatively prime?
 - YES: $(x, y) = (34, 39)$.
 - LCS: Given two strings x and y , is the length of their longest common subsequence $> k$?
 - YES: $(x, y, k) = ("CGTTAG", "GGTACG", 3)$

Certificates/1

- To find a solution for an NPC problem, we seem to be required to try out exponential amounts of partial solutions.
- Failing in extending a partial solution requires backtracking.
- However, once we found a solution, convincing someone of it is easy, if we keep a **certificate** (i.e., truth values for SAT, vertices for CLIQUE).
- The problem is finding an answer (exponential), but not verifying a potential solution (polynomial).

Certificates/2



Magic Coins and Oracles

- Assume we use a magic coin in the backtracking algorithm
 - whenever it is possible to extend a partial solutions in "two" ways, we toss a magic coin (two monkey cards, next truth assignment, etc.)
 - the outcome of this "act" determines further actions: we use magical insight.
- Such algorithms are termed "**non-deterministic**"
 - they **guess** which option is better, **rather** than employing some **deterministic procedure** to go through the alternatives

NP/1

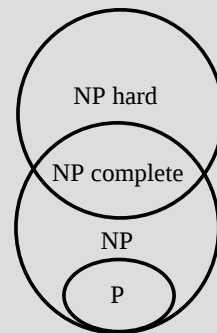
- Definition of NP:
 - Set of all decision problems solvable in polynomial time on a *non-deterministic* computer
- Definition important because it links many fundamental problems
- Useful alternative definition
 - Set of all decision problems with efficient verification algorithms
 - efficient = polynomial number of steps on deterministic TM
 - Verifier: algorithm for decision problem with extra input

NP/2

- NP = set of decision problems with efficient verification algorithms
- Why doesn't this imply that all problems in NP can be solved efficiently?
 - **PROBLEM:** need to know certificate ahead of time
 - real computers can simulate by guessing all possible certificates and verifying
 - naïve simulation takes exponential time unless you get "lucky"

NP-Completeness/1

- **NP-complete** problems are the toughest (hardest) problems in NP.
- **NP-hard** problems
 - are at least as hard as the hardest problems in NP and
 - an NP-complete problem can be reduced in polynomial time.



NP-Completeness/2

- Each NPC problem's faith is tightly coupled to all the others (complete set of problems).
- Finding a **polynomial time algorithm for one NPC problem** would **automatically** yield a polynomial time algorithm **for all NP problems**
- Proving that one NP-complete problem has an **exponential lower bound** would **automatically** prove that **all other NP-complete** problems have exponential lower bounds.

Polynomial Time Reduction

- *How can we prove such a statement?*
- **Polynomial time reduction:**
 - given two problems
 - it is an algorithm running in polynomial time that reduces one problem to the other such that
 - given input X to the first and asking for a yes/no answer
 - we transform X into input Y to the second problem such that its answer matches the answer of the first problem

Reduction Example/1

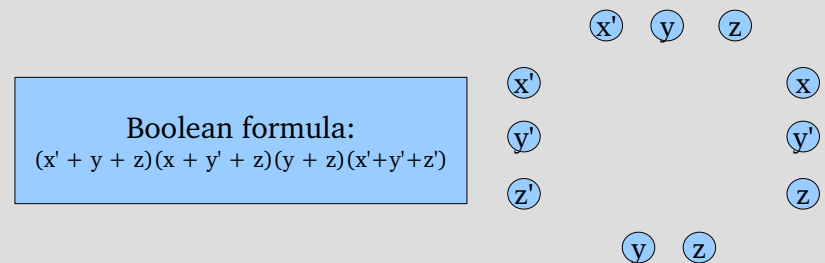
- Reduction is a general technique for showing that one problem is harder (easier) than another
 - For problems A and B , we can often show: if A can be solved efficiently, then so can B
 - In this case, we say B reduces to A (B is "easier" than A , or, B cannot be "worse" than A)
- NP-complete problem A :
 - It is NP problem
 - Any NP problem B can be *polynomially* reduced to A

Redcution Example/2

- SAT reduces to CLIQUE
 - Given any input to SAT, we create a corresponding input to CLIQUE that will help us solve the original SAT problem
 - Specifically, for a SAT formula with K clauses, we construct a CLIQUE input that has a clique of size K if and only if the original Boolean formula is satisfiable
 - If we had an efficient algorithm for CLIQUE, we could apply our transformation, solve the associated CLIQUE problem, and obtain the yes/no answer for the original SAT problem

Reduction Example/3

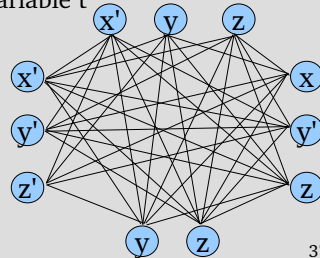
- SAT reduces to CLIQUE
 - Associate a person to each variable occurrence in each clause



Reduction Example/4

- SAT reduces to CLIQUE
 - Associate a person to each variable occurrence in each clause
 - "Two people" know each other except if:
 - they come from the same clause
 - they represent t and t' for some variable t

Boolean formula:
 $(x' + y + z)(x + y' + z)(y + z)(x' + y' + z')$



DSA09

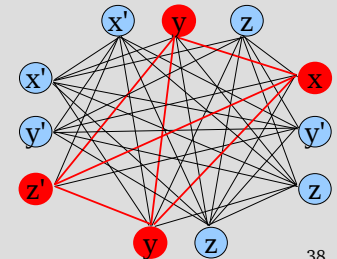
M. Böhlen

37

Reduction Example/5

- SAT reduces to CLIQUE
 - Two people know each other except if:
 - they come from the same clause
 - they represent t and t' for some variable t
 - Clique of size 4 \Rightarrow satisfiable assignment
 - set variable in clique to "true"
 - $(x, y, z) = (\text{true}, \text{true}, \text{false})$

Boolean formula:
 $(x' + y + z)(x + y' + z)(y + z)(x' + y' + z')$



DSA09

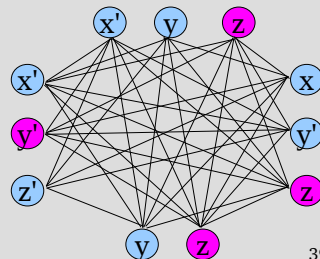
M. Böhlen

38

Reduction Example/6

- SAT reduces to CLIQUE
 - Two people know each other except if:
 - they come from the same clause
 - they represent t and t' for some variable t
 - Clique of size 4 \Rightarrow satisfiable assignment
 - Satisfiable assignment \Rightarrow clique of size 4
 - $(x, y, z) = (\text{false}, \text{false}, \text{true})$
 - choose one true literal from each clause

Boolean formula:
 $(x' + y + z)(x + y' + z)(y + z)(x' + y' + z')$



DSA09

M. Böhlen

39

CLIQUE is NP-complete

- CLIQUE is NP-complete
 - CLIQUE is in NP
 - SAT is in NP-complete
 - SAT reduces to CLIQUE
- Hundreds of problems can be shown to be NP-complete that way

DSA09

M. Böhlen

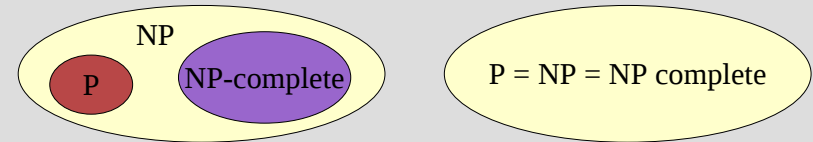
40

The Start

- The world's first NP-complete problem
- SAT is NP-complete (Cook-Levin, 1960's).

The Main Question/1

- Is $P = NP$?
 - Is the original DECISION problem as easy as VERIFICATION?
- Most important open problem in theoretical computer science. Millenium prize from Clay institute.



The Main Question/2

- If $P=NP$, then:
 - Efficient algorithms for 3-COLOR, TSP, and factoring.
 - Modern cryptography breaks down on conventional machines
- If no then:
 - Can't hope to write efficient algorithms for NP-complete problems
 - But maybe efficient algorithm still exists for testing the primality of a number – i.e., there are some problems that are NP, but not NP-complete

The Main Question/3

- Probably no, since:
 - Thousands of researchers have spent four decades in search of polynomial algorithms for many fundamental NP-complete problems without success
 - Consensus opinion: $P \neq NP$
- But maybe yes, since:
 - No success in proving $P \neq NP$ either

Dealing with NP-Completeness

- Hope that a worst case doesn't occur
 - Complexity theory deals with worst case behavior. The instance(s) you want to solve may be "easy"
 - TSP where all points are on a line or circle
 - 13,509 US city TSP problem solved (Cook et. al., 1998)
- Change the problem
 - Develop a heuristic, and hope it produces a good solution.
 - Design an approximation algorithm: algorithm that is guaranteed to find a high-quality solution in polynomial time
 - active area of research, but not always possible
- Keep trying to prove $P = NP$.

The Big Picture

- Summarizing: it is not known whether NP problems are tractable or intractable
- But, there exist provably intractable problems
 - Even worse – there exist problems with running times unimaginably worse than exponential!
- More bad news: there are **provably non-computable (undecidable)** problems
 - There are no (and there will not ever be) algorithms to solve these problems

Syllabus

- 1) Correctness and complexity of algorithms
- 2) Divide and conquer, recurrences
- 3) Sorting
- 4) Pointers, lists, sets, abstract data types
- 5) Trees, red-black trees
- 6) Hash tables
- 7) Dynamic programming
- 8) Graph algorithms

The Course – Syllabus/1

- **Data structures**
 - Simple data structures
 - *array, linked lists, stacks, queues, trees, heaps*
 - Dictionaries
 - *hash tables*
 - *binary search trees (unbalanced)*
 - *red-black trees*
 - *Graphs*
 - memory representation
 - DAG

The Course – Syllabus/2

- **Algorithmic techniques**
 - Divide and Conquer
 - *Merge sort, quicksort, binary search, ...*
 - Dynamic programming
 - *Matrix chain multiplication, longest common subsequence*
 - Search (graph) algorithms
 - *DFS, BFS*
 - *Prim, Kruskal, Dijkstra, Bellman*
 - *Topological sorting, SCC, Erdős*
 - Greedy algorithms
 - *Prim, Kruskal, Dijkstra*

The Course – Syllabus/3

- **Analyzing algorithms**
 - Correctness of algorithms
 - *pre/post-conditions*
 - *input/output*
 - *Invariants*
 - *case analysis*
 - Exact runtime
 - Asymptotic complexity
 - Recurrences

The Course – Syllabus/4

- **Constructing algorithms**
 - Break up into small pieces
 - ADT concept
 - Precise definitions of data structures and operations (signatures of procedures)
 - specification of input and output
 - Representative example of input and output

The Exam/1

- The exam will be written, you will have **two hours**. Roughly half an hour per exercise.
- Auxiliary material: **1 A4 sheet** with notes
- In the exam you have to **apply** things you learned during the course.
- Try to solve all problems. Do not get lost/stuck in details.

The Exam/2

- Template for solution:
 - 1) understand and explain problem (example)
 - 2) write down plan for solution; reduce to known techniques
 - 3) come up with algorithms
 - 4) analyze algorithms if required

The Exam/3

- *Good solution = simple and short solution that is understandable*
- *Being precise is important*
- *Solve step by step*

Preparation for the Exam

- To prepare for the exam:
 - Solve all assignments (do not start by studying/memorizing/learning an example solution)
 - Practice the first step
 - Do not only study the textbook (although you have to know the concepts).
 - Go to TAs or come to my office if you have questions

Thanks!

All the Best!!