Algorithmic Problem Solving

6hp, vt2020

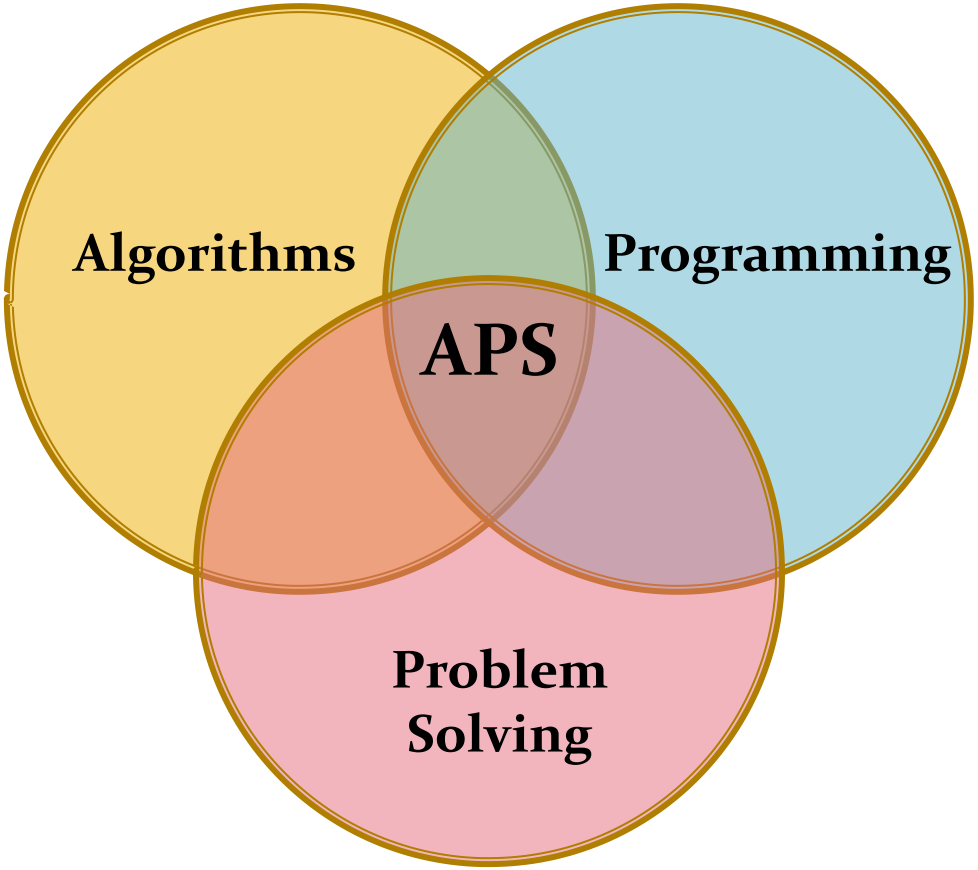
Fredrik Heintz
Dept of Computer and Information Science
Linköping University

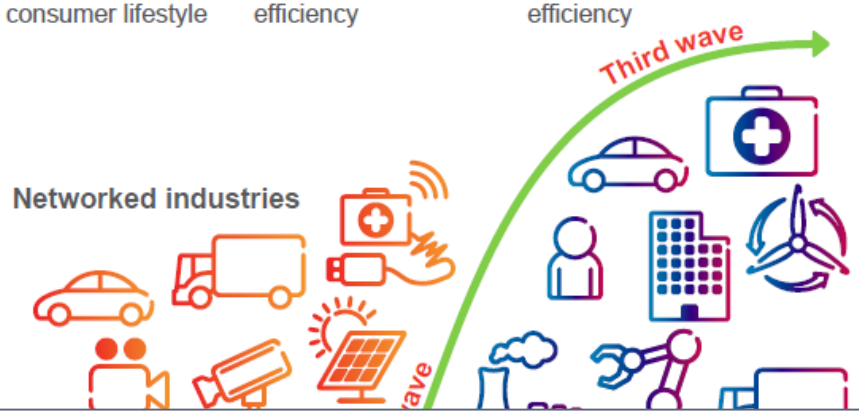
Outline

- What is algorithmic problem solving?
- Why is algorithmic problem solving important?
- What will be studied in this course?
- A method for algorithmic problem solving
- Common algorithmic problem solving approaches
- Common data structures and algorithms
- Pragmatic algorithmic problem solving using Kattis

What is Algorithmic Problem Solving?

- Algorithmic problem solving is about developing correct and working algorithms to solve classes of problems.
- The problems are normally very well defined and you know there is a solution, but they can still be very hard.





Swedsoft

ERICSSON

Mjukvaran är själen i svensk industri

Those that really understand and take advantage of software technology owns the future!

TM

facebook

Spotify

Google™

LINKÖPINGS UNIVERSITET

acm icpc

IBM event sponsor

international collegiate programming contest

The 3*n* + 1 problem

Background

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

The Problem

Consider the following algorithm:

```
1.    input n
2.    print n
3.    if n = 1 then STOP
4.          if n is odd then  n ⟵ 3n + 1
5.          else  n ⟵ n/2
6.    GOTO 2
```

Given the input 22, the following sequence of numbers will be printed 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers *n* such that $0 < n < 1{,}000{,}000$ (and, in fact, for many more numbers than this.)

Given an input *n*, it is possible to determine the number of numbers printed (including the 1). For a given *n* this is called the *cycle-length* of *n*. In the example above, the cycle length of 22 is 16.

For any two numbers *i* and *j* you are to determine the maximum cycle length over all numbers between *i* and *j*.

The Input

The input will consist of a series of pairs of integers *i* and *j*, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including *i* and *j*.

You can assume that no operation overflows a 32-bit integer.

The Output

For each pair of input integers *i* and *j* you should output *i*, *j*, and the maximum cycle length for integers between and including *i* and *j*. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers *i* and *j* must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

Example: The 3n+1 problem

100 The $3n + 1$ problem

Background

Problems in Computer Science are often classified as belonging to a certain class of problems (e.g., NP, Unsolvable, Recursive). In this problem you will be analyzing a property of an algorithm whose classification is not known for all possible inputs.

The Problem

Consider the following algorithm:

1. input n
2. print n
3. if $n = 1$ then STOP
4. if n is odd then $n \leftarrow 3n + 1$
5. else $n \leftarrow n/2$
6. GOTO 2

Given the input 22, the following sequence of numbers will be printed

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

It is conjectured that the algorithm above will terminate (when a 1 is printed) for any integral input value. Despite the simplicity of the algorithm, it is unknown whether this conjecture is true. It has been verified, however, for all integers n such that $0 < n < 1,000,000$ (and, in fact, for many more numbers than this.)

Given an input n, it is possible to determine the number of numbers printed before and including the 1 is printed. For a given n this is called the *cycle-length* of n. In the example above, the cycle length of 22 is 16.

For any two numbers i and j you are to determine the maximum cycle length over all numbers between and including both i and j.

Example: The 3n+1 problem

The Input

The input will consist of a series of pairs of integers i and j, one pair of integers per line. All integers will be less than 10,000 and greater than 0.

You should process all pairs of integers and for each pair determine the maximum cycle length over all integers between and including i and j.

The Output

For each pair of input integers i and j you should output i, j, and the maximum cycle length for integers between and including i and j. These three numbers should be separated by at least one space with all three numbers on one line and with one line of output for each line of input. The integers i and j must appear in the output in the same order in which they appeared in the input and should be followed by the maximum cycle length (on the same line).

Sample Input

```
1 10
100 200
201 210
900 1000
```

Sample Output

```
1 10 20
100 200 125
201 210 89
900 1000 174
```

Example: The 3n+1 problem

- Follow the instructions in the problem!
- Memoization to speed it up.
- Table lookup to solve it in constant time.
- Gotchas:
  - *j* can be smaller than *i*.
  - *j* can equal *i*.
  - The order of *i* and *j* in output must be the same as the input, even when *j* is smaller than *i*.

Course Goals

The goals of the course are you should be able to:

- analyze the efficiency of different approaches to solving a problem to determine which approaches will be reasonably efficient in a given situation,
- compare different problems in terms of their difficulty,
- use algorithm design techniques such as greedy algorithms, dynamic programming, divide and conquer, and combinatorial search to construct algorithms to solve given problems,
- strategies for testing and debugging algorithms and data structures,
- quickly and correctly implement a given specification of an algorithm or data structure,
- communicate and cooperate with other students during problem solving in groups.

Examination

- LAB1 4hp
  - individually solving the 4 lab assignments and
  - actively participating in at least 3 problem solving sessions.
- UPPG1 2hp,
  - individually solving the 14 weekly homework exercises, e.g.:
    - Data structures
    - Greedy Problems and Dynamic Programming
    - Graph Algorithms
    - Search
    - Math-related Problems
    - Computational Geometry.

The Schedule – VT 1

- 23/1 Practice problem solving session: 13.30-17.00 with discussion
- 24/1 Deadline Ex 1 (Greedy and DP 1) – Seminar Ex1 and Data structures
- 31/1 Deadline Ex2 (Data structures) – Seminar Ex2 and Arithmetic
- 7/2 Deadline Ex3 (Arithmetic) – Seminar Ex3 and Problem solving approaches
- 13/2 Deadline Lab Assignment 1 (Data structures, Greedy/Dynamic, Arithmetic)
- **13/2 Problem solving session (individual based on Lab 1)**
- 14/2 Deadline Ex4 (Greedy and DP II) – Seminar Ex 4 and Graphs I
- 21/2 Deadline Ex5 (Graphs I) – Seminar Ex5 and Graphs II
- 28/2 Deadline Ex6 (Graphs II) – Seminar Ex6 and Graphs III
- 5/3 Deadline Lab Assignment 2 (Graphs)
- **5/3 Problem solving session (individual based on Lab 2)**
- 6/3 Deadline Ex7 (Graphs III) – Seminar Ex7

The Schedule – VT 2

- 6/3 Deadline Ex 8 (Mixed problems) – Seminar Ex8 and Strings I
- **30/3 Problem solving session (groups based on Lab 1-2)**
- 3/4 Deadline Ex9 (Strings I) – Seminar Ex9 and Strings II
- 17/4 Deadline Ex10 (Strings II) – Seminar Ex10 and Combinatorial Search
- 20/4 Deadline Lab Assignment 3 (Strings, String Matching and Number Theory)
- **20/4 Problem solving session (individual based on Lab 3)**
- 24/4 Deadline Ex11 (Search) – Seminar Ex 11 and Number Theory
- 21/2 Deadline Ex12 (Number Theory) – Seminar Ex12 and Computational Geometry
- 28/2 Deadline Ex13 (Computational Geometry) – Seminar Ex13 and Combinatorics
- 18/5 Deadline Lab Assignment 4 (Computational Geometry)
- **18/5 Problem solving session (individual based on Lab 4)**
- 20/5 Deadline Ex14 (Combinatorics) – Seminar Ex14
- **25/5 Problem solving session (groups based on Lab 3-4)**

Steps in solving algorithmic problems

- Estimate the difficulty
  - Theory (size of inputs, known algorithms, known theorems, …)
  - Coding (size of program, many cases, complicated data structures, …)
  - Have you seen this problem before? Have you solved it before? Do you have useful code in your code library?
- **Understand the problem!**
  - What is being asked for? What is given? How large can instances be?
  - Can you draw a diagram to help you understand the problem?
  - Can you explain the problem in your own words?
  - Can you come up with good examples to test your understanding?

Steps in solving algorithmic problems

- Determine the right algorithm or algorithmic approach
  - Can you solve the problem using brute force?
  - Can you solve the problem using a greedy approach?
  - Can you solve the problem using dynamic programming?
  - Can you solve the problem using search?
  - Can you solve the problem using a known algorithm in your code library?
  - Can you modify an existing algorithm? Can you modify the problem to suite an existing algorithm?
  - Do you have to come up with your own algorithm?
- **Solve the problem! ☺**

Time Limits and Computational Complexity

- The normal time limit for a program is a few seconds.
- You may assume that your program can do about 100M operations within this time limit.

n	Worst AC Complexity	Comments
≤ [10..11]	$O(n!)$, $O(n^6)$	Enumerating permutations
≤ [15..18]	$O(2^n \times n^2)$	DP TSP
≤ [18..22]	$O(2^n \times n)$	DP with bitmask technique
≤ 100	$O(n^4)$	DP with 3 dimensions and O(n) loop
≤ 450	$O(n^3)$	Floyd Warshall's (APSP)
≤ 2K	$O(n^2 \log_2 n)$	2-nested loops + tree search
≤ 10K	$O(n^2)$	Bubble/Selection/Insertion sort
≤ 1M	$O(n \log_2 n)$	Merge Sort, Binary search
≤ 100M	$O(n)$, $O(\log_2)$, $O(1)$	Simulation, find average

Important Problem Solving Approaches

- Simulation/Ad hoc
  - Do what is stated in the problem
  - Example: Simulate a robot
- Greedy approaches
  - Find the optimal solution by extending a partial solution by making locally optimal decisions
  - Example: Minimal spanning trees, coin change in certain currencies
- Divide and conquer
  - Take a large problem and split it up in smaller parts that are solved individually
  - Example: Merge sort and Quick sort
- Dynamic programming
  - Find a recursive solution and compute it “backwards” or use memoization
  - Example: Finding the shortest path in a graph and coin change in all currencies
- Search
  - Create a search space and use a search algorithm to find a solution
  - Example: Exhaustive search (breadth or depth first search), binary search, heuristic search (A*, best first, branch and bound)

Important Data Structures and Algorithms

- Data structures
  - Standard library data structures
    - Vector, stack, queue, heap, priority queue, sets, maps
  - Other data structures
    - Graph (adjacency list and adjacency matrix), Union/find, Segment tree, Fenwick tree, Trie
- Sorting
  - Quick sort, Merge sort, Radix sort, Bucket sort
- Strings
  - String matching (Knuth Morris Pratt, Aho-Corasick), pattern matching, trie, suffix trees, suffix arrays, recursive decent parsing

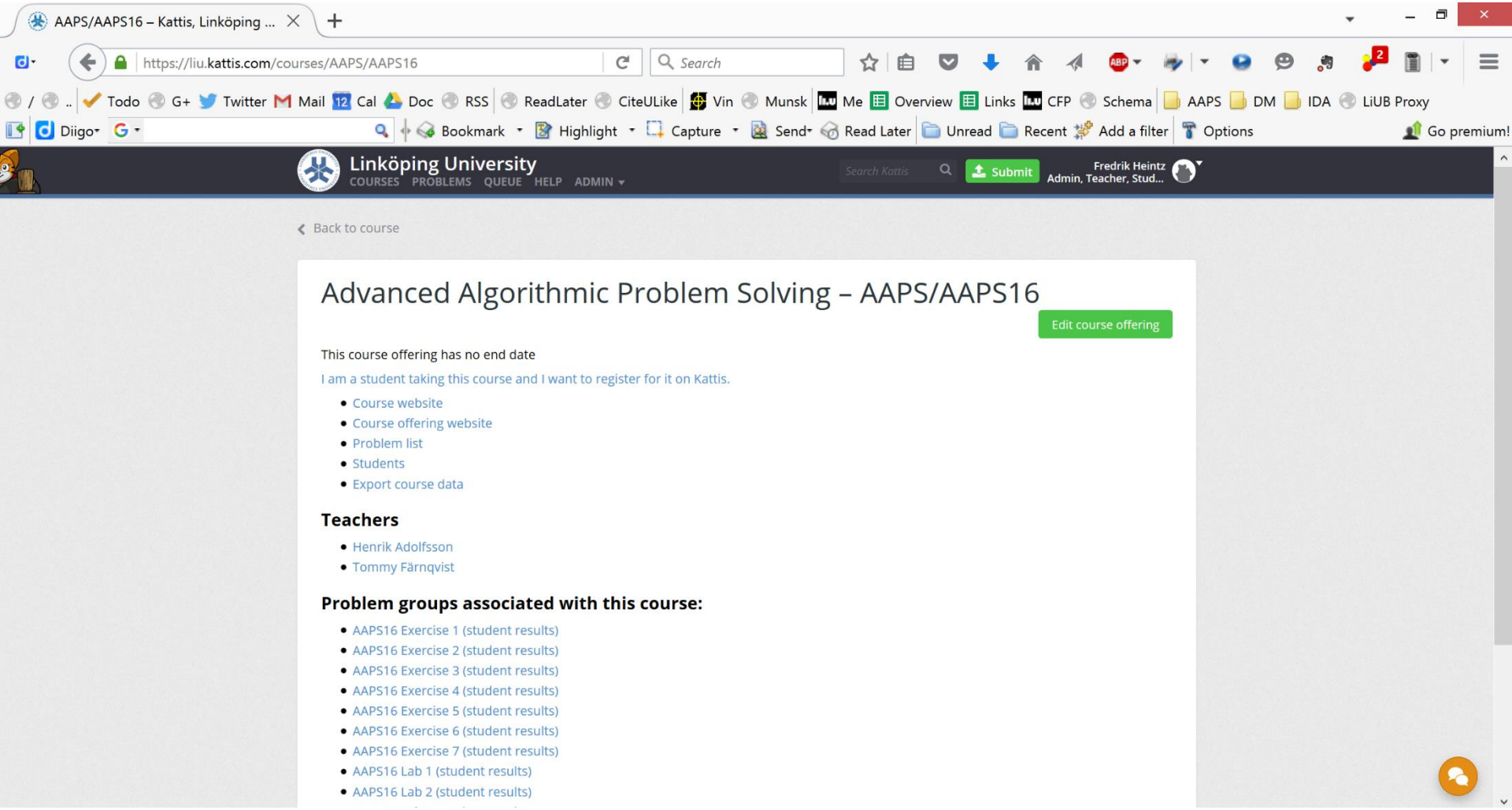
Important Data Structures and Algorithms

- Dynamic programming
  - Longest common subsequence, Longest increasing subsequence, 0/1 Knapsack, Coin Change, Matrix Chain Multiplication, Subset sum, Partitioning
- Graphs
  - Traversal (pre-, in- and post-order), finding cycles, finding connected components, finding articulation points, topological sort, flood fill, Euler cycle/Euler path, SSSP - Single source shortest path (Dijkstra, Bellman-Ford), APSP – All pairs shortest path (Floyd Warshall), transitive closure (Floyd Warshall), MST – Minimum spanning tree (Prim, Kruskal (using Union/find)), Maximal Bipartite Matching, Maximum flow, Maximum flow minimal cost, Minimal cut
- Search
  - Exhaustive search (depth-first, breadth-first search, backtracking), binary search (divide and conquer), greedy search (hill climbing), heuristic search (A*, branch and bound), search trees

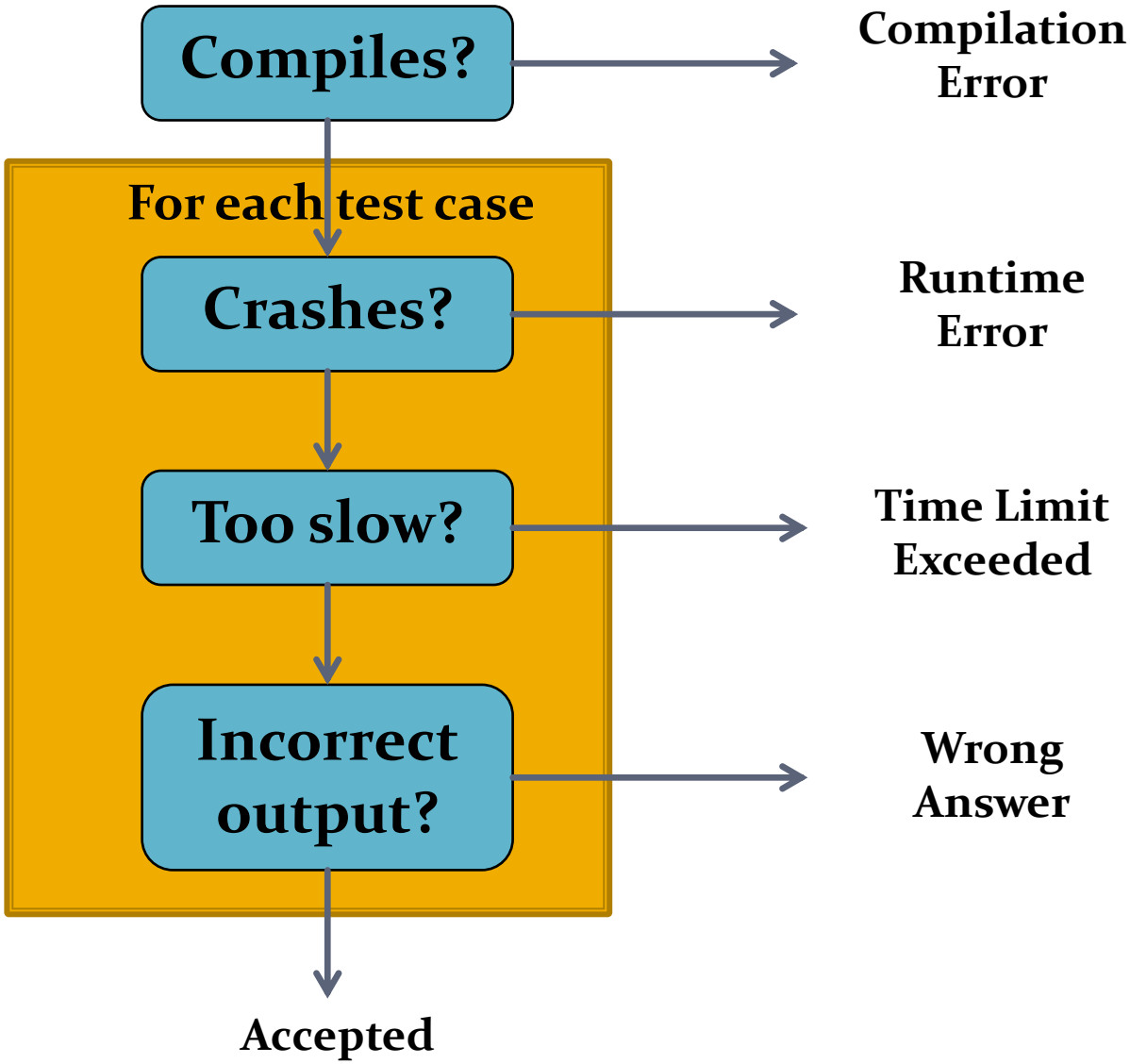
Important Data Structures and Algorithms

- Mathematics
  - Number theory (prime numbers, greatest common divisor (GCD), modulus), big integers, combinatorics (count permutations), number series (Fibonacci numbers, Catalan numbers, binomial coefficients), probabilities, linear algebra (matrix inversion, linear equations systems), finding roots to polynomial equations, diofantic equations, optimization (simplex)
- Computational geometry
  - Representations of points, lines, line segments, polygons, finding intersections, point localization, triangulation, Voronoi diagrams, area and volume calculations, convex hull (Graham scan), sweep line algorithms

Kattis (https://liu.kattis.com)



How Kattis checks a program



UVA Online Judge

UVa Online Judge

http://uva.onlinejudge.org/

Home

Google™ Custom Search | Search

Main Menu

- Home
- My Account
- Contact Us
- TOOLS on the Old UVa OJ Site
- ACM-ICPC Live Archive
- Logout

Online Judge

- Quick Submit
- Migrate submissions
- My Submissions
- My Statistics
- My uHunt with Virtual Contest Service
- Browse Problems

UVa OJ fundraising campaing

As you may already discovered by the widget shown on the left, we have started a fundraising campaing to create a whole new UVa Online Judge. Please, take a couple of minutes to read the reasons for this on the campaing website, by clicking on the widget.

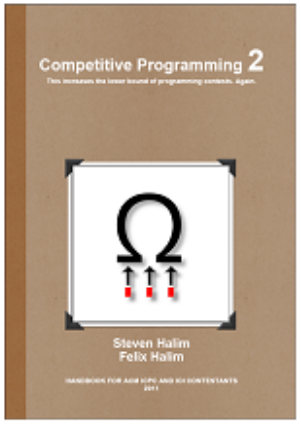
Welcome to the UVa Online Judge

Here you will find hundreds of problems. They are like the ones used during programming contests, and are available in HTML and PDF formats. You can submit your sources in a variety of languages, trying to solve any of the problems available in our database.

See the new Contest Rankings section at the Live Rankings link.

Now you can use the new Quick access, info and search option on the left menu for and easier navigation. (The tool will be updated next days

Categorized set of problems



This book contains a collection of relevant data structures, algorithms, and programming tips written for University students who want to be more competitive in the ACM International Collegiate Programming Contest (ICPC), high school students who are aspiring to be competitive in the International Olympiad in Informatics

Programming languages

- Allowed languages are C, C++, Java, and Python.
- C++ or Java is strongly recommended, use the language that you are most familiar with and want to learn more about.
- Get to know their standard libraries.
- Get to know input and output. Remember that I/O in Java is very slow, use Kattio. Remember that cout/cerr also is relatively slow, learn how to use scanf/printf if you use C++.
- Learn to use an appropriate IDE such as eclipse, emacs, or vim
- Create a problem template to speed up problem solving and to create a common format for your problems.


```
/* -*- Mode: C++ -*- */

/**
 * XXX NAME   C++   "Approach"
 *
 *    Started:
 *   Finished:
 * Total time:
 *
 * Submission 1:
 *
 * Comments:
 *
 * Lessons learned:
 */

#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <functional>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <map>
#include <set>
#include <queue>
#include <stack>
#include <string>
#include <utility>
#include <vector>

using namespace std;
typedef vector<int> vi;

int
main(int argc, char* argv[])
{

  return 0;
}
```

Testing and debugging

- Always create an example input (.in) and example output (.out) file with verbatim copies of the example input and output from the problem statement!
- For most problems it is enough to diff your output with the example output:
  ./prog < prog.in | diff - prog.out
- Create additional tests, such as:
  - Extreme inputs, i.e. smallest and largest values (0, 1, "", empty line, 2^31-1)
  - Small inputs that you can compute by hand
  - Potentially tricky cases such as when all inputs are equal, in the case of floating points numbers when you have to round both up and down
  - Very large cases, randomly generated to test that your program computes an answer fast enough (even though you might not know the correct answer).
- Use a correct but slow algorithm to compute answers.
- Print intermediate information, such as values of relevant variables.
  cout << "a=" << a << "; b=" << b << endl;
  Remember to remove the debug output before submitting! (or use cerr)

Summary

- What is algorithmic problem solving?
- Why is algorithmic problem solving important?
- What will be studied in this course?
- A method for algorithmic problem solving
- Common algorithmic problem solving approaches
- Common data structures and algorithms
- Pragmatic algorithmic problem solving using Kattis