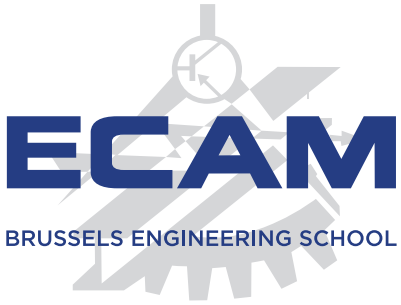


I410A Algorithm and Optimisation

Session 1

From Problem to Solution with Algorithm and Program

Sébastien Combéfis

Fall 2019

Objectives

- Discover the principles of algorithmic problem solving

  Understand what are the steps to solve a problem

- Understand the links between problem and algorithm
  - Computational and implementation problems
  - Algorithm correctness and complexity
- Overview of global strategies to solve a problem
  - Specific attitude for the case of algorithmic problems
  - Use the most relevant data structures for each problem


Algorithmic Problem

Algorithmic Problem

- Art of formulating efficient methods that solve problems
  - Popular intellectual pursuit during the last few thousand years
  - Purely mathematical endeavour, algorithms executed by hand
  - Algorithms to be implemented and executed by a computer
- Every problem is unique, but problems can be similar

  Similar observation can be done for solutions to problems...

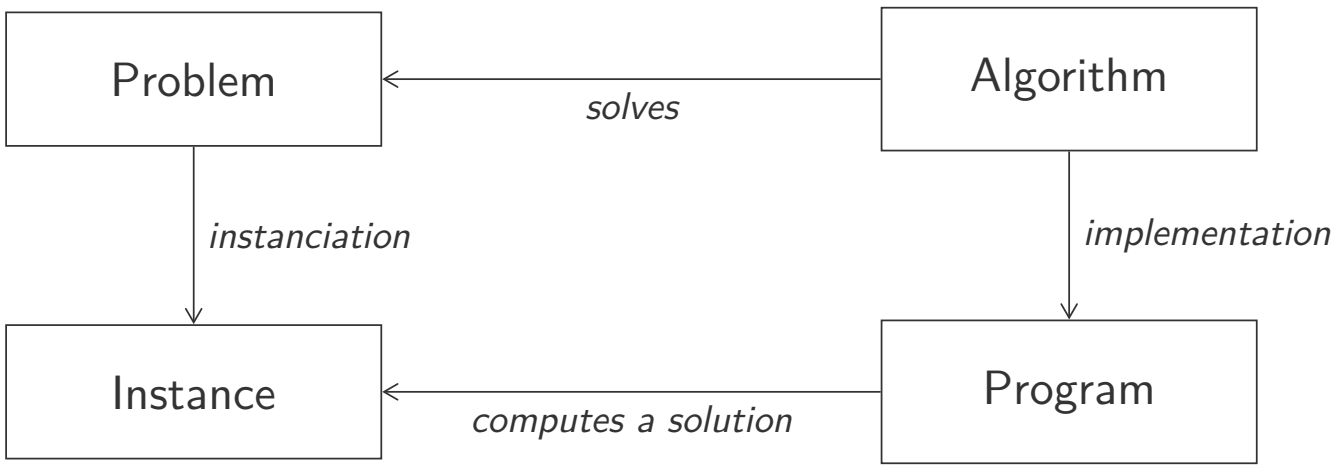
Algorithm vs Program

- The goal of an algorithm is to solve a problem

  Expressed in the form of a function to compute

- The algorithm is implemented in the form of a program

  Used to concretely solve instances of the problem



Algorithmic Problem Solving

- Writing instructions called code in a programming language

  Art of developing a solution to a computational problem

- Abstract method described by a code is called an algorithm
  - Can be **described** in several ways

    Informal, natural language, pseudo-code, etc.
  - Can be **implemented** in a concrete programming language

    "Translation" from algorithm to program can be direct or not
  - Can be **improved** to get better performances

    With different data structures, algorithm paradigm, etc.

Computational Problem

- Problem description generally consists of two parts
  - **Input** described as a data type or mathematical object
  - **Output** describes the goal to be accomplished
- Constraints can be added to the inputs

  Possible consequence on choices in competitive programming
- Challenging and difficult to find the right algorithm

  Must define/choose criteria to compare possible algorithms

Problem and Instance

- Sorting problem requires to sort a sequence of integers

  For example in descending order, from the smallest to largest

- Sorting the list $[3, 8, 4, -1, 2, 2]$ is an instance example...

  ...whose expected output is $[-1, 2, 2, 3, 4, 8]$

Problem Description
The task consists in sorting a sequence of integers in descending order, that is, from the smallest one to the largest one.

Input
A sequence of N integers $a_0, a_1, ..., a_{N-1}$.

Output
A permutation a' of the sequence a, such that $a'_0 \leq a'_1 \leq ... \leq a'_{N-1}$

Algorithm

- An algorithm is a solution to a computational problem

  Several different algorithms can solve the same problem

- Systematic method to use input to produce expected output

  Finding algorithms is a research area in itself

Selection Sort
We construct the sorted sequence iteratively one element at a time, starting with the smallest one.

After K elements have been sorted, that is, the K smallest elements of the original sequence have been chosen, the smallest element in the remaining sequence must be the $(K+1)^{th}$ smallest element.

This process is repeated N times, selecting each time the *"next smallest element"* of the output, giving rise to the elements of the original sequence, but sorted.

Abstraction Levels

- Not always easy to translate high-level description to code...

  ...until the description is sufficiently detailed

- Abstract high-level operations are easy for humans
  - They are required for the human to think about the algorithm
  - They have to be refined to lower the abstraction level
  - They must be translated to instructions for stupid computers

- English language can be extremely ambiguous

  Best to use a rigorous context/model or pseudo-code

Correctness and Complexity

Correctness

- An algorithm needs to be correct regarding the solved problem

  Very important property for an algorithm

- Two different levels of correctness for an algorithm
  - **Total** correctness if termination in a finite time
  - Otherwise only **partial** correctness
- Possible to make a compromise between the two possibilities

  Can be enough to find approximate solution within a finite time

Exactness and Termination

- Algorithm must be precisely described without uncertainty

  Description should be unambiguous for the machine

- Intuitive high-level instructions should be avoided

  Unless the detailed steps are precisely described

- An algorithm must finish and provide a result

  Must contains a finite number of steps in its execution

Soundness and Completeness

- An algorithm is sound if it never includes a wrong answer

  It may miss some answers though

- An algorithm is complete if it includes all the right answers

  It might include a few wrong answers

- A sound and complete algorithm is the best to have

  It only includes all the right answers, and you got them all

Abstraction

- An algorithm must be as general as possible
  - It should solve all instances of the problem
  - Or it can also be usable with a class of problemss
- An algorithm emphasises on the what and not on the how

  Details are left for the concrete implementation

Pseudo-Code

- Independent description of an algorithm with pseudo-code

  Not quite actual code, must more precise than natural language

- Convey most important points and structure of an algorithm

  Can be easily translated into a programming language

- Pseudo-code reads somewhat like English language

  Actions broken down into smaller pieces

Programming Language

- Write a solution that is executable by a computer

  Difference between what and how...

- Several possible choices for the chosen programming language
  - Levels of expressiveness (microcode, machine code, assembly, etc.)
  - Different available tools (compiler, checker, prover, etc.)
  - Programming paradigm (imperative, declarative, OO, etc.)

Time and Space Complexities

- Determining how fast an algorithm is with time complexity

  Even before the algorithm has been implemented

- Some hypothesis to analyse running time of an algorithm
  - *"Small operations"* take the same amount of time
  - Consider the worst case for loops (maximal iterations)
  - Asymptotic notation gives complexity as its arguments grow

Implementation Problem

- "Simplest" kind of problem is just implementation issue
  - Typically performing some simple calculation
  - Or simulating some process based on a list of rules

The Recipe (*Swedish Olympiad in Informatics 2011, School Qualifiers*)
You have decided to cook some food. The dish you are going to make requires N different ingredients. For every ingredient, you know the amount you have at home, how much you need for the dish, and how much it costs to buy (per unit).
If you do not have a sufficient amount of some ingredient you need to buy the remainder from the store. Your task is to compute the cost of buying the remaining ingredients.

Input
The first line of input is an integer $N \leq 10$, the number of ingredients in the dish.
The next N lines contain the information about the ingredients, one per line. An ingredient is given by three space-separated integers $0 \leq h, n, c \leq 200$ — the amount you have, the amount you need, and the cost per unit for this ingredient.

Output
Output a single integer — the cost for purchasing the remaining ingredients needed to make the dish.

Optimisation Problem

- Finding the best solution for a given problem
  - The problem is characterised by a **solution set** S
  - Solutions can be compared with **value function** f
- The goal is to optimise the value function

$$x_* = \arg\max_{x \in S} f(x)$$

Kattis Online Judge

Kattis
PROBLEMS CONTESTS RANKLISTS JOBS HELP
Search Kattis
Submit
Sébastien Combéfis
Score: 63.1, Rank: 6340

Solving for Carrots

Carrots are good for you! First of all, they give you good night vision. Instead of having your lights on at home, you could eat carrots and save energy! Ethnomedically, it has also been shown that the roots of carrots can be used to treat digestive problems. In this contest, you also earn a carrot for each difficult problem you solve, or huffle-puff problems as we prefer to call them.

Photo by niznoz

You will be given the number of contestants in a hypothetical contest, the number of huffle-puff problems that people solved in the contest and a description of each contestant. Now, find the number of carrots that will be handed out during the contest.

Input

Input starts with two integers $1 \leq N, P \leq 1\,000$ on a single line, denoting the number of contestants in the contest and the number of huffle-puff problems solved in total. Then follow N lines, each consisting of a single non-empty line in which the contestant describes him or herself. You may assume that the contestants are good at describing themselves, in a way such that an arbitrary 5-year-old with hearing problems could understand it.

Output

Output should consist of a single integer: the number of carrots that will be handed out during the contest.

Sample Input 1

```
2 1
carrots?
bunnies
```

Sample Output 1

```
1
```

Sample Input 2

```
1 5
sovl problmz
```

Sample Output 2

```
5
```

Submit | Stats

My Submissions

Problem ID: carrots
CPU Time limit: 1 second
Memory limit: 1024 MB
Difficulty: 1.3
Download:
Sample data files

Authors: Johan Sannemo and Oskar Werkelin Ahlin
Source: KTH Training
License: (cc) BY-SA

Assistance


Problem Solving Strategy

General Strategy

- Do not dive in directly but think first about the problem

  General problem solving approaches as a starting point

- Approach the problem with two phases: think then execute

  1. Thoroughly understand the problem and analyse it
  2. Write the code and execute the algorithm implementation

Understand the Problem

- Gather information on the problem and problem domain

  Purpose of the algorithm and targeted end users

- Identify the inputs and desired outputs of the problem

  Highlight the critical keywords and required domain information

LeetCode #26

Given a sorted array nums, remove the duplicates in-place such that each element appear only once and return the new length.

Do not allocate extra space for another array, you must do this by modifying the input array in-place with $\mathcal{O}(1)$ extra memory.

Restate the Problem

- Rephrase the problem with different words

  Make the problem meaningful for the guy who is solving it

- May be necessary to use a glossary

  To be sure to use right vocabulary according to the domain

LeetCode #26
Given a sorted array of numbers, passed by reference, destructively modify the original array in-place by removing duplicate, so that each value only appears once. Return the length of the modified array.

Instance Example

- Select some problem instances to think about the problem

  Mapping inputs to outputs by defining tests cases

- Can help to think about a first naive solution

  Based on a pen and paper resolution of the problem

Instance	**Input**	**Output**	**Side effect**
#1	[]	0	$in_1 \rightarrow []$
#2	[1]	1	$in_1 \rightarrow [1]$
#3	[1, 1, 2, 3, 4, 4, 4, 5]	5	$in_1 \rightarrow [1, 2, 3, 4, 5]$
#4	[1, 1, 1, 1, 1]	1	$in_1 \rightarrow [1]$

Challenging Problem

- What makes a problem a problem is the challenging aspect
  - There are obstacles to its resolution
  - The problem involves several challenges
- Constraints can be imposed to a solution for the problem
  - External constraints depending on the operating environment
  - Time and space complexities

Write Pseudo-Code

- Write a pseudo-code for a candidate solution

  Human high-level description of a resolution approach

- Run test instances sample inputs through the pseudo-code

  Check correctness, identify flaws, etc.

```
removeDuplicates(arr):
    if len(arr) == 0 or 1 => return len(arr)
    for each elem in arr:
        compare elem to next element
        repeat until false:
            if next element == elem:
                remove next element
        move on to the next element in arr
        stop once the second to last element has been reached
    return len(arr)
```

Implement Code

- Translate the pseudo-code into a programming language

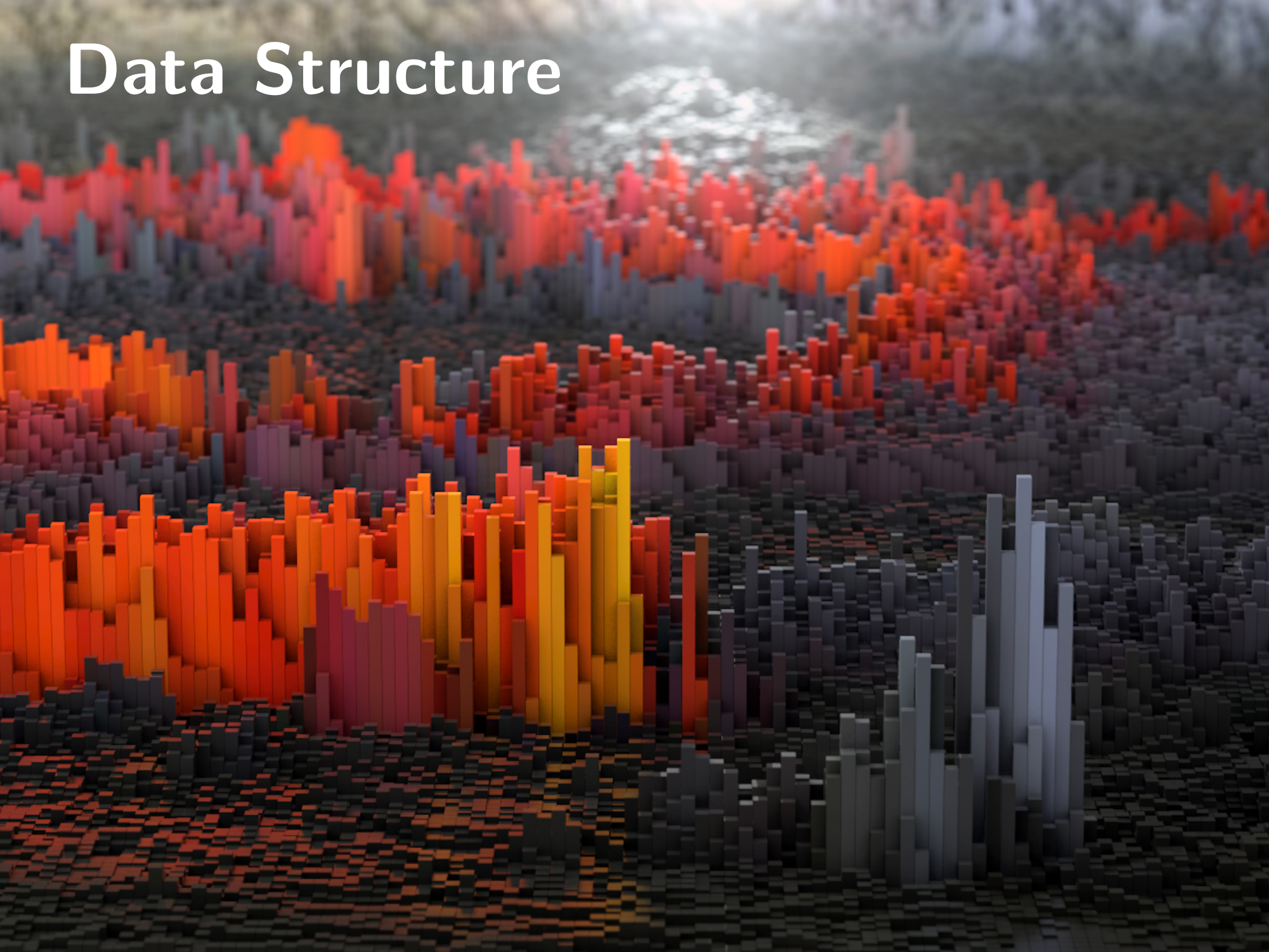
  While translating the high-level abstraction into routines

- Test the code and then refactor it once it is working

  Use features and design pattern from the language

```
1  def remove_duplicates(arr):
2      n = len(arr)
3      if n < 2:
4          return n
5      for i in range(n):
6          while i < n-1 and arr[i] == arr[i + 1]:
7              arr[i:i+1] = []
8              n -= 1
9      return n
```

Data Structure



Data Structure

- Need to deal with objects to solve an algorithm problem

  Objects are data that are manipulated by the algorithm

- Program = Algorithm + Data Structure
  - Algorithm provides the logic used to solve the problem
  - Data structure provides the values used by the algorithm

Sequence

- Sequences used to represent linear collection of elements

  Stack, *Queue*, *Deque*, *Vector*, *List*, *etc.*

- Several variants have different properties
  - Possibility to have a rank or only navigate with positions
  - Keeping elements sorted can be offered
  - Possible methods can be limited to have good performances

Dynamic Array

- Represents a continuous range of memory zones

  Often available by default in main programming language

- Fast read and write access to this kind of memory
  - Very useful for in-place algorithms (sort, for example)
  - Easy to manipulate
  - Can represent 2D maps in an efficient way (map from 1D)

Tree

- Tree used to represent an hierarchy of elements

  Can be structural or behavioural

- Several trees types with different properties
  - Binary tree, sorting tree, quad-tree, etc.
  - Typically used with recursive algorithms and backtracking

Graph

- Graph used to represent relations between entities

  Can model paths or flows between entities

- Several graphs with different properties
  - A graph can be sparse or dense, connected or not
  - A lot of problems can be expressed as a graph problem

References

- Johan Sannemo, *Principles of Algorithmic Problem Solving*, October 24, 2018.
- Jay, *Algorithmic Problem Solving for Programmers*, February 24, 2018.
  https://www.thecodingdelight.com/algorithmic-problem-solving-programmers
- Ezra Schwepker, *Algorithm Problem Solving Strategies*, April 10, 2019.
  https://dev.to/moresaltmorelemon/algorithm-problem-solving-strategies-21cp

Credits

- woodleywonderworks, August 21, 2008, https://www.flickr.com/photos/wwworks/2786241330.
- Vanessa, September 27, 2010, https://www.flickr.com/photos/takeitez/5037310758.
- Francisco Valenzuela, September 6, 2017, https://www.flickr.com/photos/146140331@N02/36923957252.
- Philippe Put, September 18, 2014, https://www.flickr.com/photos/34547181@N00/15281365335.