Module B1. Algorithms and Data Structures 1

Last updated: March 3, 2021.

Summary

Algorithms are at the heart of any computer application solving non-trivial problems in transportation, healthcare, education, social media etc. Writing efficient algorithms (because nobody likes slow apps and stack overflows..) requires patience, analytical talent, abstract thinking and creativity!

This course will help you train these skills by examining several classic algorithms for sorting, searching and optimization problems. You will learn to reason about algorithm complexity with measures like the big-Oh notation, and to use efficient data structures like hashtables, trees etc. Examples of optimization strategies discussed are greedy, backtracking, dynamic programming.

Learning outcomes

Learning outcome 1. Analysis

You are able to investigate, analyze and translate real world problems into abstract / mathematical models and solve them methodically and creatively.
You learn to estimate algorithm complexity, to recognize appropriate algorithmic strategies for solving real problems etc.

Learning outcome 2. Design

You are able to design abstract solutions (like algorithms and data structures) and able to reason about their validity/scope.
You design efficient algorithms and reason about their correctness and complexity.

Learning outcome 3. Realization

You are able to translate abstract/mathematical solutions into concrete implementations and reason about the correctness of the implementation.
You implement your algorithmic designs into actual working solutions.

Learning outcome 4. Testing

You are able to design and implement tests on the basis of abstract/mathematical specifications and reason about concepts like the coverage.
You design and apply unit tests to guarantee correctness of the solutions, and apply performance tests as well.

Learning outcome 5. Research

You are able to find and compare ideas and solutions from academic sources, such as research papers and are able to translate those abstract models/concepts into sustainable solutions.
Throughout the course, you adopt a curious attitude and take charge of your learning process. You are invited to investigate in more depth algorithms that you find interesting.

Learning outcome 6. Interaction

You are able to effectively interact and communicate with academically trained people on abstract problems and solutions.
There will be enough opportunity for interaction with your teachers and colleagues.

Learning outcome 7. Communication and Leadership

You are able to communicate your ideas to other people on an academic level both in presentations and in writing.
You describe and justify, mostly in writing, the correctness and complexity of your algorithms.

Teaching and grading

4 assignments + 1 graded (Canvas) quiz

Topics and activities per week

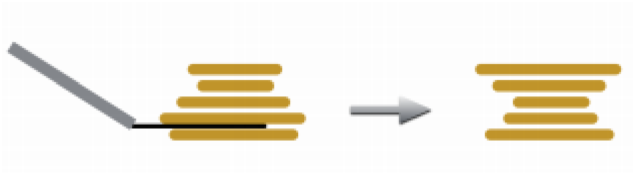
w	Topics	Resources and Activities **bold** = teacher meetings *italic* = preparation activities by you blue = the game assignment (in pairs)
1	**Algorithm efficiency** . More than "it works" . (Theoretical) time/space complexity. Big-Oh notation . Sorting algorithms	. **intro slides** . assignment 1 (see section below) . prepare week 2
2	**Binary search trees** . Searching algorithms . Smart data structures	. motivation and resources slides – *study in advance!* . intro assignment 4 (a multiplayer game) **. Q&A** – *prepare questions!* . assignment 2
3	**Algorithm design strategies** . Divide-and-conquer . Greedy . Backtracking	. motivation and resources slides – *study in advance!* . **Q&A** – *prepare questions!* . ask feedback . assignments 3 and 4 . game work
4	**Algorithm design strategies** . Dynamic Programming	. motivation and resources slides – *study in advance!* . **Q&A** – *prepare questions!* . ask feedback . assignments 3 and 4 . game work
5	**(Advanced) optimization algorithms** . games: shortest-path, minimax . bin-packing	**. inspiration session on some advanced algorithms** . catchup . game work
6	**Playtime**	. game demos / tournament . Final (Canvas) Quiz

Assignments

1. Reverse sorting

Given a pile of discs of various sizes, the challenge is to write an algorithm that rearranges them from largest (bottom) to smallest (top) by only using pairwise comparisons of the discs - **max**(i,j) - and the **`reverse(i)`** operation. That is, no swaps, no auxiliary lists.

The input is a list of integer sizes, S1…Sn. A call to **`reverse(i)`** reverses discs 1… i as shown in the image below. The output should be the increasingly ordered list and the sequence of reverse operations used to achieve the result.



Concrete steps:

(a) Write the reverse function
(b) Design and implement your reverse sorting algorithm
(c) .. while guarding correctness with at least 5 unit tests
(d) Evaluate the time complexity of your algorithm *assuming reverse is O(1)*
(e) But the reverse operation is probably more complex than O(1). What is the actual time complexity of your reverse implementation and what is then the actual time complexity of the whole algorithm?

2. Searching…

Consider a simple text document in English, of any size from a sentence to the whole Göteburg library collection 😊

Write a function **`LinearCount(word)`** that searches this textfile *sequentially* and returns the number of occurrences of the given word. The count should be case-insensitive. Then also write a function **`BinaryCount(word)`** that searches this textfile using a binary tree data structure.

For both approaches, implement any setup and breakdown operations you might need *outside the search functions*.

(a) Time the search performance of both functions for various document length and structure. Is there a difference? Can you visualize the difference on a timeplot?
(b) What is the maximum document size you can handle? What kind of problems do you run into?
(c) Is the diversity of the document vocabulary of any influence?
(d) Smart ways to improve the time and space limits?

At least (a) and (b) should be completed.

3. Pick your battles

Below are 5 problems with potentially interesting algorithmic solutions. For this assignment, look closely at each problem and shortly explain how you would approach it. Then choose one to solve completely, that is: design an algorithm for it, analyze, implement and test it.

Problem A. The way out

Given a maze MxN, with cells 0=wall, 1=available and 2=exit door, and a start position, find the shortest path to a way out.

Problem B. Send more money!

SEND + MORE = MONEY is one well-known example of a *cryptarythmetic puzzle*, where each letter represents a digit (0..9) and the challenge is to find the letter-digit assignment that makes the equation true.

Can you write an algorithm that can solve (when a solution exists!) cryptarithmetic puzzles of the form *string1* + *string2* = *string3* ? Assume that the input strings are always in capital letters. Feel free to combine a standard algorithm design strategy with sensible heuristics to limit the size of the explored solution space.

Problem C. From X to Y

Given 2 numbers X and Y and a third number m, convert X into Y by a sequence of the following operations:
(1) X := X * m
(2) X = X – 2
(3) X = X - 1

You can perform these operations in any order and any number of times. However, the problem requests to find the conversion sequence with the *minimum* number of operations.

Problem D. Shortest paths

Given a directed graph with a distance function on the arcs, and a source vertex, compute the shortest paths from the source to all the other vertices.

Problem E. Longest Common Subsequence

Given two character sequences S[1..n] and T[1..m], find their longest common subsequence (LCS). A *subsequence* is a list of characters occurring in the original sequence on positions of ascending, but not necessarily consecutive, order. For instance, “BAD” is a subsequence of “BCDAACD”, although BAD does not occur as a substring in BCDAACD.

Example: LCS(“BCDAACD”,”ACDBAC”) = “CDAC”

4. Peking Express

This digital multi-player game is inspired by the reality TV game Peking Express (https://en.wikipedia.org/wiki/Peking_Express), where participant couples engage in a hitchhiking race to the end destination Beijing (Peking).

You form a couple with another student and are together responsible for one character in the game. A map is given as a graph with *locations* and *connections,* where Peking is the destination location. Also given is a *budget* – same for each couple, and a *start* location - which is possibly different for each couple. In every turn, your character can move from the current location to a new one, along an available connection, or can choose to wait a round in the current location. Each move from a location to another has a cost (ticket price) associated with it. Some *critical* locations can only accommodate one player at a time, which means that the location, if occupied, is temporarily unavailable.

The challenge is to design and implement an algorithm that drives the moves of your character from the start location to Peking, in a minimal number of moves. Of course, the total amount spent on tickets for the whole route should fit in the budget.

The following functionalities / endpoints need to be implemented:

- **`initializeMap`** (`Json` map)
  - read the input map, which is provided in Json format with the following structure:

```
{
        "locations": {
                "number": 4,                    <- total number of locations on the map, max. 88
                "critical": [3]                 <- the list of critical locations
        },
        "connections": {
                "source": [1, 1, 1, 2, 3],         <- source[i] – target[i] is a connection
                "target": [2, 3, 88, 3, 88],          and the transport cost is price[i]
                "price": [1, 3, 7, 1, 1]
        }
}
```

- **`setStartLocation`** (`int` startlocation)
  - initialize the given startlocation as the initial position of your character.
- **`updateCompetitorLocation`** (`int` groupid, `int` location) – check the EDIT below
  - whenever another group moves, update their location on your locally maintained map. This is necessary for the availability checks of the critical locations.
- **`nextMove`**() : `int`
  - when it is your turn to move, compute and return the new location of your character

Note 1: Ideally, you implement the behaviour of your character as a REST service, with the endpoints as above, in order to keep open the possibility of one common front-end where the game could be played by all couples on the same map. This is not a strict requirement, consider it a best-effort experimental feature.

Note 2: You can make couples yourself, in Canvas, ALG1-groups. If you prefer to travel alone, that is also fine!

EDIT

Due to feasibility limitations, the game will not be played in a truly distributed manner, but rather simulated locally. See below the exact provided **Input**, required **Output** and the main structure of your algorithm using the required building blocks functionalities.

Important note: replace the initially required

`updateCompetitorLocation` (`int` groupid, `int` location)

with

`updateOccupiedLocations (locationList)`
- after every move of your own, this function should update the currently occupied locations on your locally maintained map. This is necessary for the availability checks of the critical locations.

Note: If you already implemented a REST service: nice, it will count as bonus.

Input

The (text) test files for this assignment will have the following structure:

`Json {..}` = the map

`int` = the startlocation

`OccupiedLocations` = a list of the occupied locations per turn

An occupied location is a location where, at the end of the current turn, at least one group is present. When the update list ends, all locations except your current one count as free (unoccupied).

Example testfile:

```
{"locations": {"number":4, "critical": [3]},
"connections": {
     "source": [1, 1, 1, 2, 3],
     "target": [2, 3, 88, 3, 88],
     "price": [1, 3, 7, 1, 1]
}}

1

[[2,3],[3],[88],[88]]
```

Note: Because the turns are simulated, the set of given occupied locations will sometimes not contain your own location. Make sure you add that yourself. That is, in every turn, `OccupiedLocations [i] = OccupiedLocations[i]` U { own location }. The list starts at (end of) turn 1.

Output

The output should be a list of locations describing your path from the start location to Peking (=location 88).

For example, the output

```
[1,2,2,3,88]
```

describes the path:

- started in startLocation 1

- turn 1: moved to location 2
- turn 2: stayed in location 2
- turn 3: moved to location 3
- turn 4: moved to location 88

Strategy

Your algorithm will look like this:

```
- read from the input file: map, startLocation, OccupiedLocations
initMap (map)
setStartLocation (startLocation)
currentTurn = 1
MyPath = []
while (current location <> 88)
      MyPath = MyPath + nextMove()
      execute nextMove()
      updateOccupiedLocations(OccupiedLocations[currentTurn])
print MyPath
```

--- end EDIT

Python installation and useful links

There is no restriction on what programming language you use to solve the assignments. However, if you choose Python, here are some recommendations. An easy way to get started with Python and Jupyter notebooks on your computer is to install Anaconda. Also check the Getting Started page.

Good online alternatives are:

- Kaggle – https://www.kaggle.com/kernels ("New Notebook" to start your own)
- repl.it: https://repl.it/languages/python3

Recommended FREE resources

- Jupyter Notebook for beginners = a clear introduction to using **Jupyter Notebook**, python and pandas together
- Introduction to Python and Pandas = a very good practical introduction **for absolute beginners**!
- Python 3 in Jupyter Notebook = a handy **Python** overview in notebook form
- 10 minutes to Pandas = an useful overview of the **Pandas** library, part of the official Pandas documentation
- How to use APIs to collect data - easy Python tutorial for collecting data from external sources

- Seaborn Example Gallery = A collection of beautiful Seaborn visualization examples, including the code to make them.

References

1. Knuth 😊