Algorithms and Data Structures II (1DL231)
Uppsala University — Autumn 2024

Assignment 1

Based on assignments by Pierre Flener,
but revised by Frej Knutar Lewander and Justin Pearson

— Deadline: 13:00 on Friday 22nd November 2024 —

It is strongly recommended to read the *Submission Instructions* and *Grading Rules* at the end of this document even before attempting to solve the following problems. It is also strongly recommended to prepare and attend the help sessions.

The submission must be your own work. You are not use generative AI for coding assistance.

For this assignment you will need the two Python skeleton files `weightlifting.py` and `augmenting.py`

Problem 1: The Weightlifting Problem

Given a list P of n weights of weightlifting plates at a gym, as well as a preferred total weight w for a weightlifter, the weightlifting problem is to determine whether there exists a list P' of elements in P whose sum is exactly w, since we neither want to put on too little weight nor to overburden the weightlifter. For example, if $P = [7, 9, 8]$ and $w = 15$, then the list $P' = [8, 7]$ is a solution, but there is no solution for $w = 14$.

A. Give a recursive equation for a parameterised quantity after stating its meaning in terms of all its parameters (do not rename the problem parameters P and w). Complete the following sub-tasks:

   (a) Give the base case(s) of the recursive equation.

   (b) Give the recursive case(s) of the recursive equation.

   (c) Use the equation to justify that the weightlifting problem has the optimal substructure property and overlapping subproblems, so that dynamic programming is applicable to it.

B. Using the recursive equation as a guide, implement a recursive programming algorithm without memoisation for weightlifting problem as a Python function

$$weightlifting\_recursive(P, w, p),$$

   that returns *True* if and only if there exists a list P' with integer sum $w \geq 0$ containing elements from P. Note that the implemented algorithm does not have to be efficient and should *not* use memoisation to cache results from overlapping subproblems. Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

C. Implement an *efficient* top-down dynamic programming algorithm for the weightlifting problem as a python function

$$weightlifting\_top\_down(P, w, dp\_matrix)$$

   that returns *True* if and only if there exists a list P' with integer sum $w \geq 0$ containing elements from P, where dp_matrix is a matrix of size $n \times w$ where initially each element takes value *None*. Note that the implemented algorithm has to be efficient and must use memoisation by storing results from subproblems in dp_matrix.

   You must use the provided data structure and function signatures when implementing memoisation or caching. Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

D. Consider the algorithms in Tasks B and C. Without having to prove the time complexities of the algorithms, which algorithm would you recommend someone to implement and use? Justify your answer.

E. Using the recursive equation as a guide, implement a bottom-up dynamic (iterative) programming algorithm for the weightlifting problem as a Python function

$$weightlifting\_bottom\_up(P, w, dp\_matrix),$$

   that returns *True* if and only if there exists a list P' with integer sum $w \geq 0$ containing elements from P. Note that the implemented algorithm has to be efficient and must use memoisation by storing results from subproblems in dp_matrix. Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

F. Using the algorithm implemented in Task E, implement an extended algorithm as a Python function

$$weightlifting\_list(P, w, dp\_matrix)$$

that returns such a list P' if one exists, otherwise an empty list. Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

G. We can modify the weightlifting problem such that the greatest total weight $\widehat{w}$ of weightlifting plates that does not exceed the preferred total weight of the weightlifter, $\widehat{w} \leq w$, is to be returned. Given the algorithm in Task E, formulate how the greatest total weight $\widehat{w}$ can be obtained. (You do ***not*** need to provide an implemented algorithm for this task.)

If you pass only Tasks A to D (and their sub-tasks), then your score is up to 3 points for the Problem. If you pass only Tasks A to F (and their sub-tasks), then your score is up to 4 points for the Problem. If you pass Tasks A to G (and their sub-tasks), then your score is up to 5 points for the Problem.

Problem 2: Augmenting Path Detection in Network Graphs

You will meet flow networks later in the course, but to do this assignment all you need to do is understand the definition of a flow network and the definition of an augmented path. All the information is given here in the assignment and you do not need to consult the textbook or other material. At the moment, do not worry about why flow networks and augmented paths are useful, but simply treat the problem as an exercise in designing algorithms for labelled directed graphs.

A flow network is a connected directed graph $G = (V, E)$ with a source s, a sink t, a nonnegative capacity $c(u, v)$ on each edge, and a non-negative flow $f(u, v)$ on each edge. An augmenting path P of G is a duplicate-free list of edges from s to t:

$$P = [(s, n_1), (n_1, n_2), \ldots, (n_{m-1}, n_m), (n_m, t)],$$

such that the flow is less than the capacity for each edge:

$$\forall (u, v) \in P : f(u, v) < c(u, v)$$

Perform the following tasks:

A. Design and implement an *efficient* algorithm, such as depth-first search, as a Python function

$$augmenting(G, s, t)$$

for a flow network $G = (V, E)$ with source s and sink t that returns *True* if and only if there exists an augmenting path from the source s to the sink t in G. Two points will be deducted from your score if your algorithm deletes vertices or edges; in that case, deletions must be made on a ***copy*** of G in order to comply with the style of graph algorithms in CLRS3. Also note that (see 26.1 of CLRS3) that for a network graph $G = (V, E)$ the source, s, has no incoming edges; the sink, t, has no outgoing edges; if the edge (u, v) exists in E, then no edge (v, u) exists in E; and that self-loops (edges from a vertex to itself) are forbidden. Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

B. Extend your algorithm from Task A in order to return also an augmenting path, if one exists. Implement your extended algorithm as the Python function

$$augmenting\_extended(G, s, t).$$

Your implemented function must pass ***all*** corresponding unit tests in the skeleton code file.

C. Argue that the time complexity of your extended algorithm is $\mathcal{O}(|E|)$.

If you pass only Task A, then your score is up to 3 points for the Problem. If you pass only Tasks A and B, then your score is up to 4 points for the Problem. If you pass Tasks A to C, then your score is up to 5 points for the Problem.

Submission Instructions

- Identify yourself inside the report and ***all*** code.
- State the problem number and task identifier for each answer in the report.

- Take Part 1 of the demo report at `http://user.it.uu.se/~justin/Hugo/courses/ad2/demorep` as a *strict* guideline for document structure and as an indication of its expected quality of content.
- Comment *each* function according to the AD2 coding convention at `http://user.it.uu.se/~justin/Hugo/courses/ad2/codeconv`.
- Test *each* function against *all* the provided unit tests.
- Write *clear* task answers, source code, and comments: write with the precision that you would expect from a textbook.
- Justify *all* task answers, except where explicitly not required.
- State in the report *all* assumptions you make that are not in this document. Every legally re-used help function of Python can be assumed to have the complexity given in the textbook, even if an analysis of its source code would reveal that it has a worse complexity.
- *Thoroughly* proofread, spellcheck, and grammar-check the report.
- Match *exactly* the uppercase, lowercase, and layout conventions of any filenames and I/O texts imposed by the tasks, as we will process submitted source code automatically.
- Do *not* rename any of the provided skeleton codes, for the same reason.
- Import the commented Python source-code files *also* into the report: for brevity, it is allowed to import only the lines between the copyright notice and the unit tests.
- Produce the report as a *single* file in PDF format; all other formats will be rejected.
- Remember that when submitting you implicitly certify (a) that your report and all its uploaded attachments were produced solely by you, except where explicitly stated otherwise and clearly referenced, (b) that you can explain any part starting from the moment of submitting your report, and (c) that your report and attachments are not freely accessible on a public repository.
- Submit the solution files (one report and up to two Python source-code files) without folder structure and without compression via *Studium*

Grading Rules

For each problem: if (for all of the following statements)

- there are no runtime errors when running the code under Python 3 on a Linux computer of the IT department, and
- the code passes *all* of the provided unit tests and *all* of our grading tests,

then your report will be graded for the problem (continue reading). Otherwise, the final score is automatically 0 points for the problem, and the report will not be graded for the problem. Furthermore, if (for all of the following statements)

- the requested source code exists in a file with *exactly* the name of the corresponding skeleton code (except for any characters introduced by studium);

- the code imports *only* the libraries imported by the skeleton code;
- the code has the comments prescribed by the AD2 coding convention for *all* the functions; and
- the source code features a *serious* attempt at algorithm analysis,

Then no points will be deducted from the final score for the problem. Otherwise, points will be deducted from the final score of the problem, at the discretion of the assistant.

Considering there are three help sessions for each assignment, you must earn at least 3 points (of 10) on each assignment until the end of its grading session, including at least 1 point (of 5) on each problem and at least 15 points (of 30) over all three assignments, in order to pass the *Assignments* part (2 credits) of the course.