Algorithms and Data Structures II (1DL231)
Uppsala University — Autumn 2020

Assignment 1

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

— Deadline: 13:00 on Friday 13 November 2020 —

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.

Problem 1: The Weightlifting Problem

Given a set P of n weights of weightlifting plates at a gym, as well as a preferred total weight w of a weightlifter, the weightlifting problem is to determine whether there exists a subset $P' \subseteq P$ 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 subset $P' = \{8, 7\}$ is a solution, but there is no solution for $w = 14$.

Perform the following tasks:

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). 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. Motivate your choice between bottom-up iteration (for which you must argue for the chosen nesting and iteration order of the loops) and top-down recursion, and implement an efficient dynamic programming algorithm for the weightlifting problem as a Python function $weightlifting(P, w)$ that returns *True* if and only if there exists a subset P' with integer sum $w \geq 0$ of the set P, which is given as an array of n non-negative integers.

C. Extend your algorithm from Task B to return such a subset P' as a set, empty if none exists. Implement the extended algorithm as a Python function $weightlifting\_subset(P, w)$.

D. Argue that the time complexity of your extended algorithm is $\mathcal{O}(n \cdot w)$.

Problem 2: Ring Detection in Graphs

In an undirected graph, $G = (E, V)$, a *walk* is a sequence of edges $\langle (u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k) \rangle$ where each edge on the walk is in the graph, $\forall i \in 1 \ldots k : (u_i, v_i) \in E$, and where all edges are connected:

$$\forall i \in 1 \ldots k - 1 : u_i \in \{u_{i+1}, v_{i+1}\} \vee v_i \in \{u_{i+1}, v_{i+1}\}$$

A walk is a *ring* if:

- it is closed:
$$u_k \in \{u_1, v_1\} \vee v_k \in \{u_1, v_1\}$$
- and all edges are distinct (note that it is ***not*** the vertices on the walk that must be distinct):
$$\forall i, j \in 0 \ldots k : i \neq j \Rightarrow u_i \notin \{u_j, v_j\} \vee v_i \notin \{u_j, v_j\}$$

Perform the following tasks:

A. Design and implement an efficient algorithm as a Python function *ring*(G) that returns *True* if and only if there exists a ring in the undirected graph $G = (V, E)$. 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. You can ***not*** assume that G is connected. Also recall (see Appendix B.4 of CLRS3) that in an undirected graph the edges (u, v) and (v, u) are considered to be the same edge and that self-loops (edges from a vertex to itself) are forbidden.

B. Extend your algorithm from Task A in order to return also a ring, if one exists. Implement your extended algorithm as the Python function *ring_extended*(G).

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

Submission Instructions

- Identify the team members and state the team number 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 your team, except where explicitly stated otherwise and clearly referenced, (b) that each teammate can individually 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 (by only *one* of the teammates) 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 the requested source code exists in a file with *exactly* the name of the corresponding skeleton code, ***and*** it imports *only* the libraries imported by the skeleton code, and it runs *without* runtime errors under version 3.6.9 of Python, and it produces correct outputs for *all* the provided unit tests and *some* of our grading tests in *reasonable* time on the Linux computers of the IT department, and it has the comments prescribed by the AD2 coding convention for *all* the functions, and it features a *serious* attempt at algorithm analysis, then you get at least 1 point (read on), otherwise your final score is 0 points. Furthermore:

- If the code has a *reasonable* algorithm, and it *passes most* of our grading tests, and the report addresses *all* the tasks and subtasks, then, your final score is 3 or 4 or 5 points, depending *also* on the quality of the Python source-code comments and the report part for this problem; you are *not* invited to the grading session for this problem.
- If the code has an *unreasonable* algorithm, or it *fails many* of our grading tests, ***or*** the report does *not* address all the tasks and subtasks, then your initial score is 1 or 2 points, depending *also* on the quality of the Python source-code comments and the report part for this problem; you *might be* invited to the grading session for this problem, where you can try and increase your initial score by 1 point into your final score.

However, if the coding convention is insufficiently followed or the assistants figure out a minor fix that is needed to make your code run as per our instructions, then, instead of giving 0 points up front, the assistants may at their discretion deduct 1 point from the score thus earned.

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.