Problem 1: Search Term Replacement

A useful feature for a search engine is to suggest a replacement term when a search term given by the user is not known to the engine. In order to suggest and rank replacement terms, the engine must have some measure of the difference between the given search term and a possible replacement term. We can define this difference as the minimum number of changes required to transform the user’s search term into the replacement term, where a change is either (a) altering a letter in the user’s term in order to get a letter in the replacement term, or (b) skipping a letter in one of the two terms.

For example, over the alphabet $A = \{A, \ldots, Z\}$, let the user’s search term be $u = \text{DINAMCK}$ and let the proposed replacement term be $r = \text{DYNAMIC}$. A positioning of $u$ and $r$ is a way of matching up these two strings by writing them in columns; for example:

$$
\begin{array}{ccccccc}
D & I & N & A & M & \rightarrow & C & K \\
D & Y & N & A & M & I & C & \leftarrow
\end{array}
$$

A dash (–) indicates that a letter has been skipped.

The difference of a positioning is the sum of the measures of resemblance of the pairs of letters in each column of the positioning, as given by a resemblance matrix $R$. For an alphabet $A$, we have that $R$ is an $(|A| + 1) \times (|A| + 1)$ matrix, as it must include the dash in addition to the $|A|$ characters of the alphabet. For example, the positioning above has a difference of:


Note that the difference of a positioning is the number of changes (as defined above) if $R[x,y] = 1$ for all $x, y \in A \cup \{-\} \text{ with } x \neq y$ and if $R[x,x] = 0$ for all $x \in A \cup \{-\}$.

Given two strings $u$ and $r$ of possibly different lengths over an alphabet $A$ that does not contain the dash character, and given an $(|A| + 1) \times (|A| + 1)$ resemblance matrix $R$ of numbers, perform the following tasks:

a. Give a recursive equation for the required number. Use it to justify that dynamic programming is applicable to the problem of computing the minimum difference of $u$ and $r$.

b. Design and implement an efficient dynamic programming algorithm for this problem as a Python function $\text{min_difference}(u, r, R)$, assuming that the last row and column of $R$ pertain to the dash character.

c. Extend your function from task[b] to return also a positioning for the minimum difference. Implement the extended algorithm as the Python function $\text{min_difference_align}(u, r, R)$. 
d. Argue that your (extended) algorithm has a time complexity of $O(|u| \cdot |r|)$.

Solo teams may omit task [□]. (We are not implying that search engines actually use such a dynamic programming algorithm for suggesting search term replacements.)

**Problem 2: Ring Detection in Graphs**

In an undirected graph, a path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a ring if the vertices $v_0$ and $v_k$ are equal and all edges on the path are distinct:

\[ \forall i, j \in 0 \ldots k - 1 : i \neq j \Rightarrow (v_i, v_{i+1}) \neq (v_j, v_{j+1}) \]

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 $G = (V, E)$. 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 function from task [□] in order to return also a ring, if one exists. Implement your extended algorithm as the Python function `ring_extended(G)`.

c. Argue that your (extended) algorithm has a time complexity of $O(|V|)$, independently of $|E|$.

Solo teams may omit task [□].

**Problem 3: Recomputing a Minimum Spanning Tree**

Given a connected, weighted, undirected graph $G = (V, E)$ with non-negative number edge weights, as well as a minimum(-weight) spanning tree $T = (V, E')$ of $G$, with $E' \subseteq E$, consider the problem of incrementally updating $T$ if the weight of a particular edge $e \in E$ is updated from $w(e)$ to $\hat{w}(e)$. There are four cases:

1. $e \notin E'$ and $\hat{w}(e) > w(e)$
2. $e \notin E'$ and $\hat{w}(e) < w(e)$
3. $e \in E'$ and $\hat{w}(e) < w(e)$
4. $e \in E'$ and $\hat{w}(e) > w(e)$

Perform the following tasks:

a. For each of the four cases, describe in plain English with mathematical notation an efficient algorithm for updating the minimum spanning tree, and argue that each algorithm has a time complexity of $O(|V|)$ or $O(|E|)$.

b. For at least one case that does not take constant time, say case $i \in 1 \ldots 4$, implement your algorithm as a Python function `update_MST_i(G, T, e, w)` for $w = \hat{w}(e)$.

Solo teams may omit task [□].
Submission Instructions

All program documentation and task answers (other than the programs) must be in a single report in PDF format, called a2-t=AD2-ht13.pdf for team \( t \); all other formats are rejected.

- Take Section 1 of the demo report as a guideline for document structure and as an indication of its expected quality of content.
- Document each function according to the coding convention of the course.
- State the problem number and task number of each answer in the report.
- Write clear answers, programs, and documentation.
- Justify all claims and answers, except where explicitly not required.
- State any assumptions you make. Any 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 proof-read, spell-check, and grammar-check your report.
- Match exactly the uppercase, lowercase, layout, and spacing conventions of any file names and structures imposed by the questions, as we reserve the right to process your program automatically: no grade of 4 or 5 will be awarded if manual intervention is necessary.

Only one of the teammates submits the solution files (one PDF report with the answers to all the questions (including the analyses) and three Python functions (including their specifications) that are also imported into the report), without folder structure and without compression, via the Student Portal (whose clock may differ from yours) by the hard deadline given above.

Grading

For each problem: If the requested function exists in a file with exactly the name of the corresponding skeleton code, depends only on the libraries imported by the skeleton code, runs without error under version 2.7.3 of Python (use python) on the Unix computers of the IT department, produces correct outputs for some of our grading tests in reasonable time on one of those Unix computers, has the prescribed documentation, and features a serious attempt at algorithm analysis, then you score at least 1 point (of 5):

- If your function passes most of our grading tests, then you get an initial score of 4 or 5 points, depending also on the quality of the report; you are not invited to the grading session and your initial score is your final score.
- If your function fails many of our grading tests, then you get an initial score of 1 or 2 points, depending also on the quality of the report; you are invited to the grading session, where you can try and increase your initial score by 1 point into your final score.

Otherwise you get a final score of 0 points (of 5).

You must score minimum 5 points (of 15) on this assignment until the end of its grading session, otherwise you fail the assignment part of the course.