Learning Outcomes

This assignment achieves the Learning Outcomes of:

- Analyse general problem-solving strategies and algorithmic paradigms, and apply them to solving new problems
- Prove correctness of programs, analyse their space and time complexities
- Develop and implement algorithms to solve computational problems.

In addition, you will develop the following employability skills:

- Text comprehension
- Designing test cases
- Ability to follow specifications precisely

Warning

For all assignments in this unit, you may **not** use python dictionaries or sets. For all assignments in this unit, please ensure that you carefully check the complexity of each python function that you use. Common examples which cause students to lose marks are list slicing, inserting or deleting elements in a list, using the `in` keyword to check for membership of an iterable, or building a string using repeated concatenation of characters. These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!

Assignment timeline

In order to be successful in this assessment, the following steps are provided as a **suggestion**. This is an approach which will be useful to you both in future units, and in industry.

Planning

1. Read the assignment specification as soon as possible and write out a list of questions you have about it.
2. Clarify these questions; You can go to a consultation, talk to your tutor, discuss the tasks with friends or ask in the forums.
3. As soon as possible, start thinking about the problems in the assignment.
   - It is strongly recommended that you **do not** write code until you have a solid feeling for how the problem works and how you will solve it.
4. Writing down small examples and solving them by hand is an excellent tool for coming to a better understanding of the problem.
   - As you are doing this, you will also get a feel for the kinds of edge cases your code will have to deal with.

5. Write down a high level description of the algorithm you will use.
6. Determine the complexity of your algorithm idea, ensuring it meets the requirements.
7. Write your `description.pdf`.
    - You should be able to start working on this before you write your code.
    - If you cannot, perhaps your it is worth thinking a little more about how exactly your code will work.

Implementing

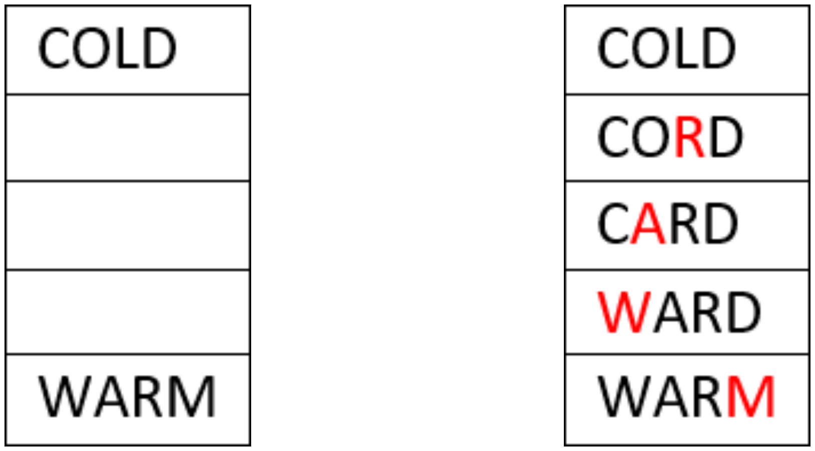
1. Think of test cases that you can use to check if your algorithm works.
    - Use the edge cases you found during the previous phase to inspire your test cases.
    - It is also a good idea to generate large random test cases.
    - Sharing test cases **is** allowed, as it is not helping solve the assignment.
2. Code up your algorithm, (remember decomposition and comments) and test it on the tests you have thought of.
3. Try to break your code. Think of what kinds of inputs you could be presented with which your code might not be able to handle.
    - Large inputs
    - Small inputs
    - Inputs with strange properties
    - What if everything is the same?
    - What if everything is different?
    - etc...

Before submission

- Make sure that the input/output format of your code matches the specification.
- Make sure your filenames match the specification.
- Make sure your functions are named correctly and take the correct inputs.
- Make sure you zip your files correctly

Background

A word ladder is a series of words, all the same length, where each word differs with the previous word by exactly one character. In this assignment we will refer to the first word as the **start_word** and the last word as the **target_word**. A word ladder puzzle is normally presented by giving the **start_word**, then some number of empty spaces, then the **target_word**. To solve the puzzle, the player must find the sequence of words that completes the chain. These words are called the intermediate words. In general the words must be valid English words. In general, the solution to a word ladder will not involve repeated words, since this loop could be removed without invalidating the solution. An example of such a puzzle and its solution are:



To make our own word ladders, we need to use a data structure which allows us to easily check if two words differ by a single letter. In this assignment, we will represent this information using a graph, where each vertex corresponds to a word, and there is an edge between two vertices when those two words differ by a single letter. A solution to a word ladder would be a path in this graph.

Algorithm Descriptions (2 mark)

For each of the three tasks, you must also write a **brief** description of your algorithm. The total length should be no more than 800 words.
These two descriptions will be submitted in the pdf file `description.pdf` mentioned in the "Submission Requirement" section. These description should explain the steps your algorithm takes to solve the problem, and the complexity of each step. Please try to keep these descriptions at a fairly high level, talk about your data structures and algorithms, not individual variables or lines of code.

0 Building the graph (6 marks)

For all the tasks in this assignment, we will be working with a graph where each vertex represents a word, and there is an edge between two vertices if and only if those two words differ by exactly one character. You do not need to check that this is true, you may assume the input files always correctly encode such a graph.

Before we start, we need to construct the graph. You will be provided with two files. You will need to read these files into an appropriate graph data structure. To do this, you will create a Python class, `Graph`. The other tasks in this assignment will each require you to write a method for your `Graph` class.

For this assignment, all the graphs will be simple, undirected graphs.

Important: If you do not create a class, or you implement the later tasks as independent functions rather than methods of the `Graph` class, you may receive 0 marks for the assignment.

0.1 Input

The graph is given to you as two files, the first containing information about the vertices, and the second containing information about the edges. For this task, you need to write the `__init__(self, vertices_filename, edges_filename)` function for your `Graph` class. This function takes two filenames as inputs, and reads the data from them into an appropriate data structure.

The first line of `vertices_filename` has a number, n, which is the number of vertices in the graph. The next n lines each start with a unique integer between 0 and $n-1$ inclusive (the vertex ID) and then a space, and then a word (the word which that vertex represents). The words will contain only lowercase English alphabet characters. **For this assignment, you may assume that all words are constant size**.

Each line of `edges_filename` consists of two numbers, separated by a a space. Each of these pairs of numbers represents an undirected edge connecting the two vertices with those IDs. You are guaranteed that all the numbers in `edges_filename` are valid vertex IDs (i.e. are integers between 0 and $n-1$ inclusive). The edges are in no particular order.

Example:
Calling this line of code should correctly populate an instance of your `Graph` class.
`my_graph = Graph("vertices.txt", "edges.txt")`
Note that here the two files are called `vertices.txt` and `edges.txt`, but they could be called anything. Similarly, the name `my_graph` is just used as an example, it could be any name:

0.2 Output

No output is required. The marks for this task are awarded for having the correct complexity, and having no errors in your graph construction. That said, the choices you make about how your graph is implemented will affect the complexity of later tasks, so make sure you have read the whole assignment.

0.3 Complexity

`__init__` should run in $O(V+E)$, where

- V is the number of lines in `vertices_filename`
- E is the number of lines in `edges_filename`

1 Solving a ladder (10 marks)

Once we have built the graph, we want to be able to solve word ladders! Given a word ladder puzzle (i.e. a **start_vertex** and **target_vertex**, which each have a word), you need to write a function which finds the shortest list of intermediate words which solves it (or reports that no such list exists)

You will write a method for the `Graph` class, called `solve_ladder(self, start_vertex, target_vertex)`, which will return a list of the intermediate words.

For a given pair of **start_vertex** and **target_vertex**, there may be several equally short lists of intermediate words. Any of them is acceptable.

1.1 Input

`start_vertex` and `target_vertex`. These are integers between 0 and $V - 1$ inclusive, where V is the number of vertices in the graph.

1.2 Output

The function should return a list of strings, which correspond to the words in the word ladder from **start_vertex** to **target_vertex**. In some cases, it may not be possible to find a chain from the **start_vertex** to the **target_vertex**. In these cases, the function should return `False`.

Example: If `vertices_filename` contains

```
7
0 aaa
1 aab
2 abb
3 bbb
4 caa
5 cba
6 bba
```

Then `edges_filename` would contain

```
0 1
0 4
1 2
2 3
3 6
4 5
5 6
```

Suppose that we had created an instance of `Graph`, called `my_graph` to store the above graph. `my_graph.solve_ladder(0, 6)` returns `["aaa", "caa", "cba", "bba"]`

1.3 Complexity

`solve_ladder` must run in $O(V + E)$ time, where

- V is the number of vertices in the graph
- E is the number of edges in the graph

2 Restricted word ladder (12 marks)

Now, rather than the shortest word ladder, you want to find the cheapest word ladder. The **cost** of a word ladder is found by summing the the costs of the changes made to characters during that ladder. The cost of changing a character from given by squaring the difference in alphabet position of the two characters. For example, the cost of transforming "a" into "c" would be $(3-1)^2 = 4$. Transforming "a" into "z" would cost $(26-1)^2 = 625$. The cost of the word ladder in the background section, which starts with "cold" and ends with "warm" would be

$$\begin{aligned}&(L \to R) + (O \to A) + (C \to W) + (D \to M)\\ =&(12-18)^2 + (15-1)^2 + (3-23)^2 + (4-13)^2\\ =&36 + 196 + 400 + 81\\ =&713\end{aligned}$$

However, there is an additional requirement. In this task, the word ladder must also contain at least one word which starts with a given letter. This letter will be one of our input parameters.

You will write a method for the `Graph` class, called `cheapest_ladder(self, start_vertex, target_vertex, req_char)` to solve this problem.

2.1 Input

`start_vertex`, `target_vertex` and `req_char`. `start_vertex` and `target_vertex` are integers between 0 and $V-1$ inclusive, where V is the number of vertices in the graph. `req_char` is a lowercase english alphabet character.

2.2 Output

`cheapest_ladder` should return two things as a tuple (or `False`). The first is the cost of the cheapest word ladder. The second is a list of strings, which are the intermediate words in the cheapest word ladder starting at `start_vertex` and ending at `target_vertex`, such that at least one word in the ladder has `req_char` as its first character (This can be the the word associated with `start_vertex` or `target_vertex`).

In some cases, it may not be possible to find a chain from the **start_vertex** to the **target_vertex** with at least one word in the ladder has `req_char` as its first character . In these cases, the function should return `False`.

Example:

```
7
0 aaa
1 aab
2 abb
3 bbb
4 caa
5 cba
```