FIT2004 S1/2022: Assignment 1

DEADLINE: Friday 25th March 2022 16:30:00 AEDT

LATE SUBMISSION PENALTY: 10% penalty per day. Submissions more than 7 calendar days late will receive 0. The number of days late is rounded up, e.g. 5 hours late means 1 day late, 27 hours late is 2 days late. For special consideration, please visit this page: `https://forms.monash.edu/special-consideration` and fill out the appropriate form.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using the **Python programming language** (version should not be earlier than 3.5). This practical work will be marked on the time complexity, space complexity and functionality of your program, and your documentation.

Your program will be tested using automated test scripts. It is therefore critically important that you name your files and functions as specified in this document. If you do not, it will make your submission difficult to mark, and you will be penalised.

SUBMISSION REQUIREMENT: You will submit a single python file, `assignment1.py`.

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. In previous semesters, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. Helping others to solve the assignment is NOT ACCEPTED. Please do not share your solutions partially or completely to others. If someone asks you for help, ask them to visit a consultation for help.

Learning Outcomes

This assignment achieves the Learning Outcomes of:

- 1) Analyse general problem solving strategies and algorithmic paradigms, and apply them to solving new problems;
- 2) Prove correctness of programs, analyse their space and time complexities;
- 3) Compare and contrast various abstract data types and use them appropriately;
- 4) 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.

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.

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 (if required)

Documentation

For this assignment (and all assignments in this unit) you are required to document and comment your code appropriately. This documentation/commenting must consist of (but is not limited to):

- For each function, high level description of that function. This should be a one or two sentence explanation of what this function does. One good way of presenting this information is by specifying what the input to the function is, and what output the function produces (if appropriate).
- For each function, the Big-O complexity of that function, in terms of the input. Make sure you specify what the variables involved in your complexity refer to. Remember that the complexity of a function includes the complexity of any function calls it makes.
- Within functions, comments where appropriate. Generally speaking, you would comment complicated lines of code (which you should try to minimise) or a large block of code which performs a clear and distinct task (often blocks like this are good candidates to be their own functions!).

1 Partial Wordle Trainer (5 marks + 1 mark for documentation)

You and your friends have been competing daily in a game of Wordle [1]. You often find yourself struggling in the game, due to your limited vocabulary. Thus in order to improve by quickly recognising character combinations, you have decided to build yourself a simple trainer to practice using a list of words.

Due to time constraints, the trainer you are building is only a partial one where each letter in the guessed word could only be marked as either – (1) that letter is correct and in the correct position; or (2) that letter is in the answer but not in the right position. The marking for the guessed word is detailed in Section 1.1 with the variable `marker`.

Your trainer is a simple Python function `trainer(wordlist, word, marker)` which you will use to quickly identify the possible word matches from the a given word list. This word list could contain words of any length; instead of being restricted to the 5-letters Wordle standard.

1.1 Input

- `wordlist` is a list of N words.
  - They are all strings of length M, with each character in the range of lowercase $\{a - z\}$.
  - You can assume that there are no duplicate words in the list.
  - However, you cannot assume that the list is in lexicographical order.
  - Unlike Wordle that only contains words of length 5, the words in `wordlist` can be of any fixed length.
- `word` is a word which you have guessed.
  - It is a string of length M, with each character in the range of lowercase $\{a - z\}$.
- `marker` is an array of integers of length M.
  - Each element is in the range $\{0, 1\}$, and is used to mark the characters in the guessed word.
  - 0 denotes that the character at that index exists in another position.
  - 1 denotes that the character at that index exists in the same position.
  - Recall that your trainer only needs to handle guesses where all the letters are correct, but not necessarily in the right position.

[1] https://en.wikipedia.org/wiki/Wordle

1.2 Output

- Your algorithm returns a list of strings containing the valid words, based on the input provided.
- The words in the returned list are in lexicographical order.
- If there are no valid words possible, then your algorithm returns an empty list.

Several examples of inputs and corresponding outputs are given in Section 1.3.

1.3 Examples

Consider the following examples.

```
# Example 01
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'pares'
# The markers based on the guessed word
marker = [0, 0, 0, 0, 1]

>>>trainer(wordlist, word, marker)
['reaps']
```

In the first example above, you have guessed the word *pares* which is marked with $[0, 0, 0, 0, 1]$. This means that:

- The first 4 letters of the guess, p, a, r, and e, are in the wrong position. The last letter s is in the right position.
- Words like *pears* are not valid because we know that the letter p shouldn't be at the start of a word.
- Words like *parse* are not valid because we know that the word has to end with the letter s.
- Thus, the only remaining possible correct word would be *reaps*.

```
# Example 02
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'pares'
# The markers based on the guessed word
marker = [1, 0, 0, 0, 1]

>>>trainer(wordlist, word, marker)
['pears']
```

In this second example, you have guessed the word *pares* which is now marked with more matches $[1, 0, 0, 0, 1]$. This means that:

- Your guesses of the first letter p and the last letter s are both in the right position.
- Your guesses of the middle 3 letters a, r, and e are in the wrong positions.
- Thus, the only remaining possible correct word would be *pears*.

```
# Example 03
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'pares'
# The markers based on the guessed word
marker = [0, 0, 0, 0, 0]

>>>trainer(wordlist, word, marker)
['spare', 'spear']
```

For the third example above, you have guessed the word *pares* and all letters are in the wrong position. This leaves us with *spare* and *spear* as the only valid words. Do note that these words are in lexicographical order.

```
# Example 04
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'spare'
# The markers based on the guessed word
marker = [1, 1, 0, 0, 1]

>>>trainer(wordlist, word, marker)
[]
```

In the fourth example above, your guess of *spare* returns an empty list [] because there are no valid words. On the other hand, the fifth example below with the query *sprae* would return *spare* in the result.

```
# Example 05
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'sprae'
# The markers based on the guessed word
marker = [1, 1, 0, 0, 1]

>>>trainer(wordlist, word, marker)
['spare']
```

```
# Example 06
# The wordlist input
wordlist = ['limes', 'spare', 'store', 'loser', 'aster', 'pares',
    'taser', 'pears', 'stare', 'spear', 'parse', 'reaps', 'rates',
'tears', 'losts']
# The guessed word
word = 'spare'
# The markers based on the guessed word
marker = [1, 1, 1, 1, 1]

>>>trainer(wordlist, word, marker)
['spare']
```

In the sixth example above, the guess of *spare* with all markers being 1s would return the guessed word itself.

```
# Example 07
# The wordlist input
wordlist = ['costar', 'carets', 'recast', 'traces', 'reacts',
    'caster', 'caters', 'crates', 'actors', 'castor']
# The guessed word
word = 'catrse'
# The markers based on the guessed word
marker = [1, 1, 0, 0, 0, 0]

>>>trainer(wordlist, word, marker)
['carets', 'caster']
```

Lastly, in the seventh example above, we look at words with 6 letters. It retains the same behaviour as the 5 letters example.

1.4 Complexity

`trainer(wordlist, word, marker)` must run in $O(NM + NX + XlogN) = O(NM)$ worst-case time complexity and $O(NM)$ space, given that:

- N is the number of items in `wordlist`.
- M is the length of the words in `wordlist` and `word`.
- X is the number of 0's in `marker`.
- You can assume the character set for the words in the range a to z as a constant.

1.5 Hint(s)

In order to solve this problem, consider utilising algorithms and concepts that you have learnt from FIT2004, namely:

- Non-comparison based sorting.
- Divide and Conquer.

Going through every word in the `wordlist` and comparing them against the guess `word` can be expensive even with the information from `marker`. You would either need a mechanism to track the misplaced letter or to try out every combination of the misplaced letter. How can you search for the candidate words faster?

With that, consider the sorting applications – with a sorted list, what can we use it for? Studio 3 discusses several applications of sorting.

2 Finding a Local Maximum (3 marks + 1 mark for documentation)

You are given as input an n-by-n grid of distinct numbers (represented as a matrix), and want to find a local maximum. For each number, its neighbours are the numbers immediately above it, below it, to its left, and to its right. Note that while most numbers have 4 neighbours, the ones on the edge of the matrix only have 3 neighbours, and the ones in the corners only have 2 neighbours. We will consider a number to be a local maximum if all its neighbours are smaller than it.

2.1 Input

Your local maximum finder is a Python function `local_maximum(M)`, where `M` is an n-by-n grid of distinct integers (i.e., an n-by-n matrix).

2.2 Output

Your algorithm should output a single pair of coordinates x and y such that `M`$[x][y]$ is a local maximum. The output should be in the following format: `[x, y]`. If there are multiple local maxima, your algorithm should output the coordinates of exactly one local maximum, and this can be any of the existing local maxima.
Consider the following matrix where the only local maximum is in red:

$$\begin{bmatrix} 1 & 2 & 27 & 28 & 29 & 30 & 49 \\ 3 & 4 & 25 & 26 & 31 & 32 & 48 \\ 5 & 6 & 23 & 24 & 33 & 34 & 47 \\ 7 & 8 & 21 & 22 & 35 & 36 & 46 \\ 9 & 10 & 19 & 20 & 37 & 38 & 45 \\ 11 & 12 & 17 & 18 & 39 & 40 & 44 \\ 13 & 14 & 15 & 16 & 41 & 42 & 43 \end{bmatrix}$$

For this example, the input/output behaviour of your Python function should be as follows:

```
# Example
# The matrix input
M = [[1, 2, 27, 28, 29, 30, 49],
     [3, 4, 25, 26, 31, 32, 48],
     [5, 6, 23, 24, 33, 34, 47],
     [7, 8, 21, 22, 35, 36, 46],
     [9, 10, 19, 20, 37, 38, 45],
     [11, 12, 17, 18, 39, 40, 44],
     [13, 14, 15, 16, 41, 42, 43]]

>>>local_maximum(M)
[0, 6]
```

In other words, running `local_maximum(M)` returns `[0, 6]` where `M`[0][6] has the local maximum with the value 49.
Consider the following example where the matrix has two local maxima:

$$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 & 28 \\ 2 & 5 & 9 & 14 & 20 & 27 & 34 \\ 4 & 8 & 13 & 19 & 26 & 33 & 39 \\ 7 & 12 & 18 & 25 & 32 & 38 & 90 \\ 11 & 17 & 24 & 31 & 37 & 57 & 91 \\ 99 & 98 & 97 & 60 & 59 & 58 & 56 \\ 22 & 29 & 35 & 40 & 44 & 55 & 49 \end{bmatrix}$$

In this case your algorithm should output either `[5, 0]` or `[4, 6]`.

2.3 Further Examples

We present below some further example of matrices, and their local maxima in red.

$$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 & 28 \\ 2 & 5 & 9 & 14 & 20 & 27 & 34 \\ 4 & 8 & 13 & 19 & 50 & 33 & 39 \\ 7 & 12 & 18 & 25 & 32 & 38 & 51 \\ 52 & 17 & 24 & 31 & 37 & 42 & 46 \\ 16 & 23 & 30 & 36 & 41 & 45 & 48 \\ 22 & 29 & 35 & 40 & 44 & 47 & 49 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 & 21 & 28 & 164 & 201 & 203 & 206 & 210 & 215 & 221 & 228 \\ 2 & 5 & 9 & 14 & 20 & 27 & 34 & 163 & 202 & 205 & 209 & 214 & 220 & 227 & 234 \\ 4 & 8 & 13 & 19 & 26 & 33 & 39 & 162 & 204 & 208 & 213 & 219 & 226 & 233 & 239 \\ 7 & 12 & 18 & 25 & 32 & 38 & 43 & 161 & 207 & 212 & 218 & 225 & 232 & 238 & 290 \\ 11 & 17 & 24 & 31 & 37 & 42 & 46 & 160 & 211 & 217 & 224 & 231 & 909 & 908 & 907 \\ 16 & 23 & 30 & 36 & 41 & 45 & 48 & 159 & 216 & 223 & 230 & 260 & 906 & 904 & 902 \\ 22 & 29 & 35 & 40 & 44 & 47 & 49 & 158 & 222 & 229 & 235 & 340 & 305 & 903 & 901 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 157 & 506 & 505 & 504 & 503 & 502 & 501 & 650 \\ 101 & 102 & 127 & 128 & 129 & 130 & 149 & 156 & 601 & 302 & 327 & 328 & 629 & 630 & 649 \\ 103 & 104 & 125 & 126 & 131 & 132 & 148 & 155 & 603 & 604 & 625 & 626 & 631 & 632 & 648 \\ 105 & 106 & 123 & 124 & 133 & 134 & 147 & 154 & 605 & 606 & 623 & 624 & 633 & 634 & 647 \\ 107 & 108 & 121 & 122 & 135 & 136 & 146 & 153 & 607 & 608 & 621 & 622 & 635 & 636 & 646 \\ 109 & 110 & 119 & 120 & 137 & 138 & 145 & 152 & 609 & 610 & 619 & 620 & 637 & 638 & 645 \\ 111 & 112 & 117 & 118 & 139 & 140 & 144 & 151 & 611 & 612 & 617 & 618 & 639 & 640 & 644 \\ 113 & 114 & 115 & 116 & 141 & 142 & 143 & 150 & 613 & 614 & 615 & 616 & 641 & 642 & 643 \end{bmatrix}$$

2.4 Complexity

For an n-by-n grid of distinct numbers, your algorithm for finding a local maximum should have a worst-case time complexity of $O(n)$.

Warning

For all assignments in this unit, you may **not** use python **dictionaries** or **sets**. This is because the complexity requirements for the assignment are all deterministic worst-case requirements, and dictionaries/sets are based on hash tables, for which it is difficult to determine the deterministic worst-case behaviour.

Please ensure that you carefully check the complexity of each in-built python function and data structure that you use, as many of them make the complexities of your algorithms worse. Common examples which cause students to lose marks are **list slicing**, inserting or deleting elements **in the middle or front of a list** (linear time), using the `in` keyword to **check for membership** of an iterable (linear time), or building a string using **repeated concatenation** of characters. Note that use of these functions/techniques is **not forbidden**, however you should exercise care when using them.

These are just a few examples, so be careful. Remember, you are responsible for the complexity of every line of code you write!