Story

Nathan showed up to last week's games night expecting a glorious victory, and sweet revenge on all his friends. Alas, even with the help of your algorithm, he lost. Someone else had an even faster algorithm! But that is a problem for another time.

A new week, a new game. This time, the group will be playing the game **Snake-words**. Nathan again has requested your help, and has actually invited you to come along in case the algorithm needs modifications on the night.

The two of you arrive early, and the host this week invites you to play a quick game, called the **coin taking game** before the other guests arrive. Nathan agrees, but you decline politely so you can work on your **Snake-words** algorithm.

After Nathan has lost several times, you realise that the **coin taking game** is a perfect candidate for a technique you have just learnt, and decide to code up a solution for it in addition to your **Snake-words** algorithm.

Background

Coin taking game

This game is played between 2 players, player 1 and player 2. There are two piles of coins. The values of a coin can be any integer. Both players know the values of all coins in both piles. Player 1 makes the first move, and play alternates between the players. A move consists of taking a coin from the top of either of the piles (either player can take from either pile). The game ends when both piles are empty. A player's score is the sum of the values of all the coins they took during the game. The player with the higher score wins.

Snake-words

This game is played on $n \times n$ grid of lowercase English letters, and involves finding words in this grid. We say this grid **contains** a word if the word exists as a **chain** in the grid. A **chain** is a sequence of letters $a_1, a_2, ...a_n$ in the grid which are sequentially adjacent. In other words, the $i{+}1^{th}$ letter is vertically, horizontally or diagonally adjacent to the i^{th} letter.

You may not use the same grid cell for letter i and letter $i + 1$, however, you **are** allowed to re-use a letter in a **chain**.

Algorithm Descriptions (2 mark)

For each of the two tasks, you must also write a **brief** description of your algorithm. The total length should be no more than 500 words.
These two descriptions will be submitted in the pdf file `descriptions.pdf` mentioned in the "Submission Requirement" section. These descriptions 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.

1 Winning the coin taking game

In this task you will write a function `best_score(pile1, pile2)` to determine optimal play for the coin taking game. It will determine the highest score which can be achieved by player 1, assuming that both players play optimally.

1.1 Input

Two lists, `pile1` and `pile2`. These lists contain integers which represent the values of the coins in the game. The numbers in `pile1` are the values of coins in the first pile, where `pile1[i]` is the value of the coin with `i` coins beneath it in the pile. `pile2` represents the coins in the second pile in the same way. The piles may be empty.

1.2 Output

A tuple with two elements. The first is a single number which represents the highest score which player 1 can achieve. The second represents the choices made by both players during the game (assuming optimal play).

The choices are represented by a list containing only the numbers `1` and `2`, where a `1` indicates that the player took the top coin from `pile1` on their turn, and `2` indicates that the player took the top coin from `pile2` on their turn.

If multiple sequences of choices are optimal, any of these sequences is acceptable.

Example:

- Calling `best_score([20,5], [1])` would return `(21, [2,1,1])`. It is optimal for player 1 to take the coin of value 1, forcing player 2 to take the coin of value 5, which allows player 1 to take the coin of value 20.
- Calling `best_score([1,2,3], [1])` would return `(4, [1,1,1,2])`
- Calling `best_score([5,8,2,4,1,10,2], [6,2,4,5,6,9,8])` would return `(42, [2, 2, 2, 2, 2, 1, 1, 1, 2, 2, 1, 1, 1, 1])`

1.3 Complexity

`best_score` must run in $O(NM)$, where

- N is the number of elements in `pile1`
- M is the number of elements in `pile2`

`best_score` should use no more than $O(NM)$ auxiliary space

2 Finding Snakey words

You need to be able to find a word in a given Snake-words grid. To do this, you will write a function `is_in(grid, word)`

2.1 Input

`grid` is a list of length N, each element of which is a list of length N. The elements of the internal lists are all single lowercase english alphabet characters (a-z). `grid[i][j]` represents the letter in the i^{th} row and j^{th} column of `grid`.

In other words, `grid` is a $N \times N$ table of letters, represented as a list of lists.

`word` is a sequence of lowercase English characters (a-z). A word does not contain any kind of punctuation or other non-alphabet symbols.

2.2 Output

If `grid` does not **contain** `word` (as described in the Background section) then `is_in` should return `False`. If `grid` does **contain** `word`, `is_in` should return a **list of tuples**, which represent the grid cells that correspond to the word. The i^{th} tuple corresponds to `(row, column)` of the i^{th} letter of `word` in `grid`.

If a word appears in multiple different ways in the grid, any of the ways it can appear are acceptable answers

Example:
If the following variables were defined:

```
grid = [[‘a’,‘b’,‘c’,‘d’],
        [‘e’,‘a’,‘p’,‘f’],
        [‘e’,‘p’,‘g’,‘h’],
        [‘l’,‘i’,‘j’,‘k’]]
word1 = ‘apple’
word2 = ‘xylophone’
```

`is_in(grid, word1)` should return `[(1,1), (1,2), (2,1), (3,0), (2,0)]`

For your reference, the word’s position in the grid is shown below

```
[[‘a’,‘b’,‘c’,‘d’],
 [‘e’,‘a’,‘p’,‘f’],
 [‘e’,‘p’,‘g’,‘h’],
 [‘l’,‘i’,‘j’,‘k’]]
```

`is_in(grid, word2)` should return `False`

2.3 Complexity

`is_in` must run in $O(KN^2)$, where

- K is the number of characters in `word`
- N is the length of `grid` (this means that there are N^2 elements in `grid`)

`is_in` should use no more than $O(KN^2)$ auxiliary space

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!