Assignment Guidelines.

- This assignment covers material in Module 05.
- Submission details:
  - Solutions to these questions must be placed in files a05q1.py, a05q2.py, a05q3.py, and a05q4.py, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
- Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include import check in each solution file.
  - When a function produces a floating point value, you must use check.within for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than math and check.
  - Do not use Python constructs from later modules (e.g. loops). Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
  - While you may use global constants in your solutions, do not use global variables for anything other than testing.
  - Read each question carefully for additional restrictions.
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1. **Vowel runs.** Consider the string "delicious". We say it has a *vowel run* of 3 because the maximum consecutive vowel substring is "iou" which has length 3. Similarly "door" has a *vowel run* of 2 while "xyz" has a *vowel run* of 0.

More formally, given a string *s*, the *vowel run* of *s* is the length of the largest substring containing only the characters 'a', 'e', 'i', 'o', 'u'. The character 'y' is not a vowel for the purposes of this question.

Write the function `max_vowel_run` that consumes a list of lowercase strings *L* and returns a sublist of *L* of all the strings that have the biggest *vowel run* in the list.

For example:

```python
max_vowel_run(['boots','cat','beautiful','hairy','cookeeez'])
```

=> ['beautiful','cookeeez']

because both "beautiful" and "cookeeez" have a *vowel run* of 3 which is the maximum of this list.

```python
max_vowel_run(['xyz','bbb','cvtrrt'])
```

=> ['xyz','bbb','cvtrrt']

because all strings have a *vowel run* of 0.

**You must use accumulative recursion for this question.**

2. **Differences.** For this question assume we are only dealing with lists of integers.

Given a list *L* with at least 2 numbers define the *list difference* of *L*, `diff(L)`, to be the list that is generated by subtracting adjacent elements of *L* from left to right.

For example if *L* = [5, 2, -4, 3, 2] then `diff(L) = [3, 6, -7, 1]`.

Further, define the *difference number* of *L* to be the amount of repeated list differences needed to generate a constant list from *L*. A constant list is a list of numbers that are all the same.

For example:

```python
M = [17, 2, 5, 26, 65, 122]
diff(M) = [15, -3, -21, -39, -57]
diff(diff(M)) = [18, 18, 18, 18]
```

and the *difference number* of *M* is 2. Note that every list of length 1 will have a *difference number* of 0.

Write the function `diff_num` that consumes a non-empty list of numbers, *lon* and returns the *difference number* of *lon*.

**Note that your function must not mutate the original list**

For example:

```python
diff_num([17, 2, 5, 26, 65, 122]) => 2
diff_num([5, 2, -4, 3, 2]) => 4
```
Background: Given a list of numbers that represent $y$ coordinates, if the $x$ coordinates are assumed to be evenly spaced then the difference number will tell you what degree polynomial can be used to intersect all of the points.

In the example above with $M = [17, 2, 5, 26, 65, 122]$ a second degree polynomial $ax^2 + bx + c$ can be made to intersect these points (assuming evenly spaced $x$ coordinates).

3. Hills. A string $s$ with lowercase alphabetic letters is called a hill if either:

- it has length 3 such that $s[0] < s[1]$ and $s[1] > s[2]$
- $s[0] < s[1]$ and $s[-2] > s[-1]$ and $s[1:-1]$ is a hill

For example, "apxgb" is a hill but "apxzb" is not. Note that this definition requires a hill to have odd length of at least 3. Essentially a hill will have an equal number of consecutive alphabetic increases as decreases (with the increases coming first from left to right).

Given a string $s$, and $0 <= i < j <= len(s)$ the substring $s[i:j]$ is called a maximal hill if $s[i-1:j+1]$ is not a hill but $s[i:j]$ is a hill. Note that if the whole of $s$ is a hill then it is maximal.

For example in the string $ex = "abcdcaptb"$, the string "cdc" is a hill but it is not maximal because "bcdca" is also a hill. In fact "bcdca" is a maximal hill in this case because "abcdcap" is not a hill. Furthermore the substring "ptb" is also a maximal hill of $ex$. Thus, $ex = "abcdcaptb"$ has 2 maximal hills: ['bcdca', 'ptb'].

Write the function `find_hills` that will consume a non-empty string $s$ of lowercase alphabetic characters and return a list of all maximal hills of $s$ in the order they appear in $s$.

For example:

```python
find_hills("caba") => ['aba']
find_hills("abacaba") => ['aba', 'aca', 'aba']
find_hills("abba") => []
find_hills("abcde") => []
find_hills("delicious") => ['delic', 'ous']
find_hills("ab") => []
```

4. Finding cheese. In this question you will help a mouse find its way to cheese (if possible). Note that despite the long description, there is a rather short solution to this problem. Consider, as an example, the 4x6 grid as shown below (4 cells down, 6 cells across).

```
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
+---+---+-++-+-+
|   |   |   |   |
```

C
For the purposes of clarity, when discussing directions, you should assume we are referring to how YOUR eyes would describe them based on this top down view. So for example, if we say LEFT we mean left on the screen and not to the left of the mouse. Similar for RIGHT, DOWN and UP, these are all in relation to how the grid is displayed to YOU.

The entrance to the grid is always somewhere at the top and the cheese (labeled with a C) is always somewhere in the bottom row of cells.

Each time the mouse lands on a cell it hasn’t visited it will mark that cell with an X. Also, every time the mouse visits a new cell it will determine its next position by recursively checking:

DOWN, LEFT, RIGHT, UP

in that order. It will only proceed as long as there is no wall in the way and it hasn’t already visited that cell. If there are no places left then it will backtrack to its most recent visited cell (even though there will be an X there) and check the remaining directions recursively.

For example, for the grid above the mouse would find its way to the cheese as seen in the animation ex1.gif and the final state of the grid would be:

+-+ +-+-+-+-+
| X X |
+-+-+ +-+-+-+
| X |
+-+-+ +-+-+-+
| X X |
+-+-+ +-+-+-+
| X | X | X | X | C |
+-+-+-+ +-+-+-+

As long as the mouse follows these steps AND there is a path to the cheese then it will find it. Your function will return True in this case. Note that the cell with the C in it will not be changed.

If there is not a path to the cheese then it will eventually make its way back to the entrance and stop. Your function will return False in this case.

For example, in the grid below, the mouse cannot get to the cheese.

+-+ +-+-+-+
| |
+-+-+ +-+-+
| |
+-+-+-+ +-+-+
| |
+-+-+-+ +-+-+
| |
+-+-+-+ +-+-+
| |
+-+ +-+-+-+
| C |
+-+-+-+ +-+-+-+
In this case the mouse would proceed as in \texttt{ex2.gif} and the final state of the grid would be:
\begin{verbatim}
+-+ ++++++++
 | X X X X X |
+ ++ ++ + ++
 | X X X X X |
+-+ ++++++++
 | | X X X |
+ | | X | |
+| | X X X |
+-+ ++++++++
 | | |X| |
+-+ ++++++++
 | | X X X X |
+-+ ++++++++
 | | C |
+-+ ++++++++
\end{verbatim}

Notes:

- The grid will be made up of a list of lists. Each sublist will contain strings each string having only one character. At the start the only characters in the grid will be \texttt{{' ' ' ' '+' '+' '|'|}} with the exception of a single \texttt{'C'} that appears once in the grid. As the program runs the grid should also be mutated as needed to contain the required number of \texttt{'X'} characters.
- All “cells” of the grid (i.e. where the mouse could potentially go) will have odd indices. That is, if \texttt{grid[y][x] == 'X'} then \texttt{x} and \texttt{y} must be odd. This is also true of the cheese \texttt{'C'}.
- All horizontal walls and horizontal open doorways will have an even \texttt{y} value (i.e. row) and an odd \texttt{x} value (i.e. column).
- All vertical walls and vertical open doorways will have an odd \texttt{y} value (i.e. row) and an even \texttt{x} value (i.e. column).
- Though there are \texttt{'+'} characters present they are more of a visual aid and shouldn’t really come into play.
- No exterior wall should be removed except for the entrance which is always at the top.
- The cheese \texttt{'C'} is always in the bottom row of cells.
- The smallest allowable grid is a 1 by 1 grid that only contains cheese. Specifically \texttt{+ +}
\begin{verbatim}
| C |
+--+
\end{verbatim}

Your job is to write the function \texttt{find_cheese} which will consume a \texttt{Grid} \texttt{grd} (see interface file) and return \texttt{True} or \texttt{False} if the mouse can find its way to the cheese. Furthermore your function should print the final state of \texttt{grd}.

For this question only, we will not mark you on your choice of tests. We will also not check whether you mutated the grid or not (though you should be). Our correctness tests will however mark you for printing the correct grid as well as returning the correct result. The other design recipe components will also be marked.
For example, assume we do the following:

```python
ex1 = '''
+-+ +-+-+-+-+
| | |
+-+ +-+ +-+
| | |
+ + + + +-+
| | |
+ + + + + +
| | C|
+-+-+-+-+-+-+
'''
temp_grid = ex1.split('
')
grd = list(map(list, temp_grid[1:-1]))
```

At this point `grd` will look like:

```
[['+', '-', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '|', '|', '|', '|', '|', '|', '|', '|'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '|', '|', '|', '|', '|', '|', '|', '|'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['+', '|', '+', '+', '+', '+', '+', '+', '+', '+'],
 ['|', '|', '+', '+', '+', '+', '+', '+', '+', '+']]
```

From here if we now call the `find_cheese` function we get:

```python
find_cheese(grd) => True
```

and it will print

```
+-+ +-+-+-+-+-+
| X X |
+ + + + + + + +
| X | |
+ + + + + + + +
| X X | |
+ + + + + + + +
| X X X X X C|
+-+-+-+-+-+-+
```

The interface file has a function called `build_grid` which you can use to help test out your code. Read over the purpose statement to understand how it works.
Appendix
(a different way of looking at things)

Below is essentially a step-by-step view of each example.

1. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X | \\
+ + + + + + + \\
| | \\
+ + + + + + + \\
| C| \\
+-+ -+-+-+-+
\end{array}
\]

2. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| | \\
+ + + + + + + \\
| C| \\
+-+ -+-+-+-+
\end{array}
\]

3. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| | \\
+ + + + + + + \\
| C| \\
+-+ -+-+-+-+
\end{array}
\]

4. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| | \\
+ + + + + + + \\
| C| \\
+-+ -+-+-+-+
\end{array}
\]

5. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

6. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

7. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

8. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

9. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

10. 

\[
\begin{array}{c}
+-+ -+-+-+-+ \\
| X X | \\
+ + + + + + + \\
| X | \\
+ + + + + + + \\
| X X | \\
+-+ -+-+-+-+
\end{array}
\]

\[7\]