Daily Coding Problem #253

Problem

This problem was asked by PayPal.

Given a string and a number of lines k, print the string in zigzag form. In zigzag, characters are printed out diagonally from top left to bottom right until reaching the k^{th} line, then back up to top right, and so on.

For example, given the sentence `"thisisazigzag"` and `k = 4`, you should print:

```
t     a     g
 h   s z   a
  i i   i z
   s     g
```

Solution

One way to solve this would be to go one line at a time, figuring out what that line should be, and printing it. An advantage of this method would be that we would only need `O(N)` space at any given time. Let's see how this would work.

For the zigzag pattern above, we can see that the letters in the top and bottom lines are each separated by 5 spaces. What about the middle lines? Here it is trickier, it depends on

each separated by 5 spaces. What about the middle lines? Here it is trickier- it depends on whether the pattern is descending or ascending. When we are ascending, we should put 3 spaces after the second line and 1 space after the third line, but if we are ascending the reverse is true.

Let's try to clear up this confusion by looking at what happens with 5 lines:

```
t       o       z
 h     n t     g a
  i   a   h   i   g
   s s     e z
    i       r
```

Here as we move from top to bottom there are 7, 5, 3, and 1 spaces added after each letter, and the same is true when we go from bottom to top.

So if `row` is the current row we're on, `desc` represents whether or not we are descending, and `k` is the number of lines, we can predict the number of tacked-on spaces using the following formula:

```
def get_spaces(row, desc, k):
    max_spaces = (k - 1) * 2 - 1
    if desc:

        spaces = max_spaces - row * 2
    else:
        spaces = max_spaces - (k - 1 - row) * 2
    return spaces
```

This presents us with another challenge: how do we know whether or not the pattern is descending? Note that if we have five rows, we will be descending for the first four, ascending for the next four, and so on. This can be representing mathematically like so:

```
def is_descending(index, k):
    if index % (2 * (k - 1)) < k - 1:
        return True
    else:
        return False
```

Putting these together, our algorithm will create a list of empty strings for the first row. After putting the first character in this list at the appropriate index, it will check whether the

pattern is ascending or descending, find out how many spaces are needed, and move to the next index. When we get to the end of the row, we print it out, and repeat this process for subsequent rows.

```
def zigzag(sentence, k):
    n = len(sentence)

    for row in range(k):
        i = row
        line = [" " for _ in range(n)]

        while i < n:
            line[i] = sentence[i]
            desc = is_descending(i, k)
            spaces = get_spaces(row, desc, k)
            i += spaces + 1

        print("".join(line))
```

Even though `is_descending` and `get_spaces` are constant-time operations, we still need to join and print each line of the string, which will take `O(N)` time, so the whole algorithm will be `O(k * N)`.

We can make this significantly less complicated, however, by loosening our space requirements and storing each row to be printed in a `k x N` matrix. As we move letter by letter through the sentence, we update the appropriate row and column with the current letter. The column will be simply the current index, and the row can be found by keeping track of whether we are ascending or descending and moving one step forward in the right direction.

```
def zigzag(sentence, k):
    n = len(sentence)
    line_matrix = [[' ' for _ in range(n)] for _ in range(k)]

    row = 0
    for col, letter in enumerate(sentence):
        line_matrix[row][col] = letter

        if row == 0:
            descending = True
        elif row == k - 1:
```

```
            descending = False

        if descending:
            row += 1
        else:
            row -= 1

    for line in line_matrix:
        print("".join(line))
```

This algorithm is `O(k * N)` in both time and space.