Problem 1.

Without using loops, the `**` operator, or any function in the `math` module, write a function with the signature

```
power(x, n)
```

which returns x raised to the power of n. You can assume n is a non-negative integer.

Hint: $x^n = x^{n-1} \times x$.

Problem 2.

Without using loops, write the function with the signature

```
interleave(L1, L2)
```

which takes `L1` and `L2`, two lists of the same length, and returns a list which consists of `L1` and `L2` interleaved, i.e., `[L1[0], L2[0], L1[1], L2[1], ..., L1[n-1], L2[n-1]]` (here, `n == len(L1) == len(L2)`).

Problem 3.

Without using loops or slicing, write a function that reverses a list in place. That is, the effect of calling

```
reverse_rec(L)
```

should be that `L` is reversed.

Here is how you might do this *with* loops:

```
def reverse_loop(L):
  for i in range(len(L)//2):
    L[i], L[-1-i] = L[-1-i], L[i]
```

You need to write a helper function that calls itself.

Problem 4.

Fun fact: this question is taken from the final exam that I wrote in my first year.

Without using loops and without ever using `print` with a list (as opposed to individual elements of a list), write a function that, given a list `L` of size n (assume n is odd), prints the elements of `L` in the following order:

`L[n//2] L[n//2-1] L[n//2+1] L[n//2-2] L[n//2+2] L[n//2-3] L[n//2+3] ... L[n-1]`

Hint: here is a recursive function that prints the following sequence for a list `L` of size n:

`L[0] L[n-1] L[1] L[n-2] L[2] L[n-3] ... L[n//2]`

```
def zigzag1(L):
  if len(L) == 0:
    print('')
  elif len(L) == 1:
    print(L[0], end = "")
  else:
    print(L[0], L[-1], end = "")
    zigzag1(L[1:-1])
```

Problem 5.

Without using any loops or global variables, write a function the with signature (exactly, without default parameters)

```
is_balanced(s)
```

which returns `True` iff the string `s` is has "balanced" parentheses, i.e., all parentheses `()` in string `s` match exactly. For example, `"((()()))"` is balanced but the following:

`"(well (I think), recursion works like that (as far as I know)"`

is not, since it's missing a closing parenthesis. To simplify matters, you may start by thinking about strings that contain only parentheses, but your final function should work with all strings. Your function should only care about balancing parentheses `()`, not brackets `[]` or braces `{}`.

You may use `str.find` and `str.rfind`, or (recursive) helper functions. (Note: `str.find` is not a recursive function, but of course could be implemented recursively.)

Problem 6.

This is the last ESC180 lab this semester – say goodbye and thank you to your lab TAs!

Problem 7.

Challenge question

Download `nim.py`.

Nim is a generalization of the Race to 21 game we played in class. Write code that uses the Nim code in order to play Race to 21 (i.e., write a function that returns the best move given a current sum)

Now, write similar code to analyze tic-tac-toe positions. Tic-tac-toe is different from Nim and Race to 21 in that a tie is also possible.

Write a function named `score_state` that returns 1 if the player starting at the given state will win with perfect play, 0 if the result with perfect play is a tie, and -1 if the player starting at the given state will eventually lose with perfect play.

In `is_winning_state`, we return True if every possible move by the opponent will result in False. In `score_state`, we should return 1 if every possible move the the opponent will result in a score of 1; return 0 if every possible move by the opponent will result in a score of -1 or 0 (so the opponent would choose a move that results in a score of 0), and 1 if every possible move by the opponent will result in a score of -1.

In other words, we want to return -max(possible opponent scores). (Make sure you understand why.) This strategy is known as Minimax (we are minimizing the maximum score by the opponent).

Unlike with Nim or Race to 21, the possible moves in tic-tac-toe are different depending on whose turn it is. Likewise, `is_win` needs to check how many X's and 0's there are on the board, infer whose turn it is, and then return the result based on that.

Skeleton code that you should use is in `minimax_ttt.py`.