20 RECURSION PRACTICE QUESTIONS

EASY/INTERMEDIATE

SOLUTIONS & EXPLANATIONS INCLUDED

LIU ZUO LIN

1) Factorial

Write a recursive function *factorial(n)* that takes in a positive integer n, and returns the factorial of n. The factorial of n is 1 x 2 x 3 x … x n.

```
factorial(1)   # 1
factorial(2)   # 2
factorial(3)   # 6
factorial(4)   # 24
factorial(5)   # 120
factorial(6)   # 720
factorial(7)   # 5040
factorial(8)   # 40320
```

2) Summation

Write a recursive function *summation(n)* that takes in a positive integer n, and returns the summation of n. The summation of n is 1 + 2 + 3 + … + n

```
summation(1)  # 1
summation(2)  # 3
summation(3)  # 6
summation(4)  # 10
summation(5)  # 15
summation(6)  # 21
summation(7)  # 28
summation(8)  # 36
```

3) Reversing Uppercase And Lowercase

Write a recursive function *reverse(string)* that takes in a string, and returns the same string, but with uppercase and lowercase letters reversed. Originally uppercase letters should become lowercase, and vice versa.

```
reverse('apple')    # APPLE
reverse('Apple')    # aPPLE
reverse('AppLe')    # aPPlE
reverse('APPLE')    # apple
```

4) Sum Of Odd Numbers

Write a recursive function *sum_odd(numbers)* that takes in a list of integers, and returns the sum of ONLY the odd numbers.

```
sum_odd([1,2,3,4])    # 4
sum_odd([1,2,3,4,5])  # 9
sum_odd([1,2,4,6,8])  # 1
sum_odd([])           # 0
sum_odd([2,4,6,8])    # 0
```

5) Add Odd Numbers, Subtract Even Numbers

Write a recursive function *add_sub(numbers)* that takes in a list of integers. It sums up all ODD numbers, but subtracts all EVEN numbers.

```
add_sub([1,2,3,4])    # -2
add_sub([1,2,3,4,5])  # 3
add_sub([1,3,5])      # 9
add_sub([2,4,6])      # -12
add_sub([99,10,10])   # 79
```

6) Counting Vowels

Write a recursive function *count_vowels(string)* that takes in a string, and counts the number of vowels inside the string. Vowels include a, e, i, o, and u.

```
count_vowels('apple')       # 2
count_vowels('orange')      # 3
count_vowels('pear')        # 2
count_vowels('pineapple')   # 4
count_vowels('durian')      # 3
```

7) Removing Vowels

Write a recursive function *remove_vowels(string)* that takes in a string, and returns another string with ALL vowels removed. Vowels include a, e, i, o, and u.

```
remove_vowels('apple')       # ppl
remove_vowels('orange')      # rng
remove_vowels('pear')        # pr
remove_vowels('pineapple')   # pnppl
remove_vowels('durian')      # drn
```

8) Replacing Vowels

Write a recursive function *replace_vowels(string)* that takes in a string, and replaces all vowels with its 'next' vowel.

- replace 'a' with 'e'
- replace 'e' with 'i'
- replace 'i' with 'o'
- replace 'o' with 'u'
- replace 'u' with 'a'

```
replace_vowels('apple')       # eppli
replace_vowels('orange')      # urengi
replace_vowels('pear')        # pier
replace_vowels('pineapple')   # ponieppli
replace_vowels('durian')      # daroen
```

9) Double Letters

Write a recursive function *double_letters(string)* that takes in a string, and doubles each character inside the original string.

```
double_letters('apple')       # aappplle
double_letters('orange')      # oorraannggee
double_letters('pear')        # ppeeaarr
double_letters('pineapple')   # ppiinneeaappplle
double_letters('durian')      # dduurriiaann
```

10) Palindromes

A palindrome is a word that is the same backward as it is forward.

Examples of palindromes:

- aaa
- aba
- ababa
- abcba
- mom
- moom

Examples of NON-palindromes:

- aaab
- abab
- ababab
- abcda
- moms

Write a recursive function *is_palindrome(string)* that takes in a string, and returns True if the string is a palindrome, and False if the string is NOT a palindrome.

```
is_palindrome('apple')     # False
is_palindrome('ababab')    # False
is_palindrome('abcda')     # False
is_palindrome('aaaba')     # False
is_palindrome('aaaab')     # False

is_palindrome('aba')       # True
is_palindrome('abba')      # True
is_palindrome('aaaaaa')    # True
is_palindrome('moooom')    # True
is_palindrome('mooooom')   # True
```

11) Number Pyramid

Write a recursive function *number_pyramid(n)* that takes in a positive integer n, and prints the following number pyramid of height n.

```
# number_pyramid(3)
1
12
123
```

```
# number_pyramid(5)
1
12
123
1234
12345
```

12) Reverse Number Pyramid

Write a recursive function *reversed_pyramid(n)* that takes in a positive integer n, and prints the following reversed number pyramid of height n.

```
# reverse_pyramid(3)
321
21
1
```

```
# reverse_pyramid(5)
54321
4321
321
21
1
```

13) Spaced Number Pyramid

Write a recursive function *spaced_pyramid(n)* that takes in a positive integer n, and prints the following spaced number pyramid of height n.

```
# spaced_pyramid(3)
  1
 12
123
```

```
# spaced_pyramid(5)
    1
   12
  123
 1234
12345
```

14) Fibonacci Numbers

Fibonacci numbers are a sequence of numbers:

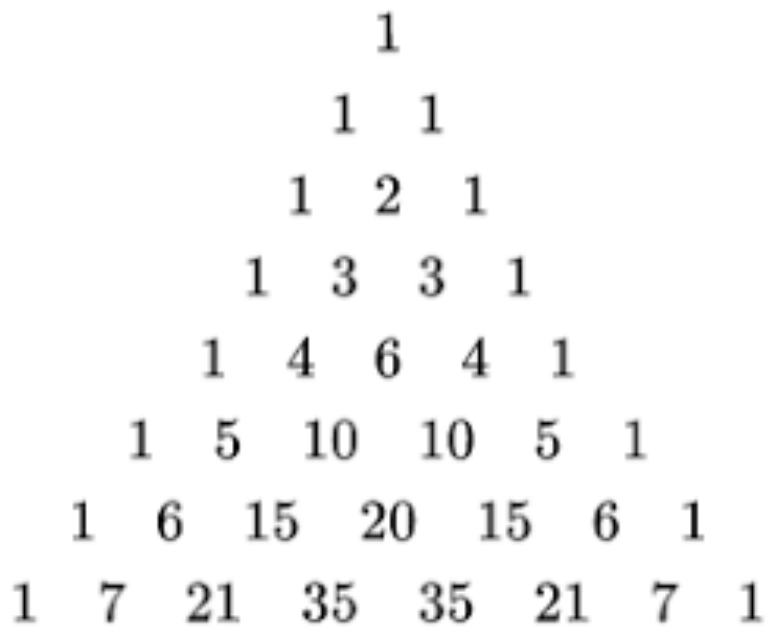
- That begin with 0 and 1
- Where each new number is the sum of the previous 2 numbers.

The first 10 numbers go 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Write a recursive function *fib(n)* that takes in a positive integer n, and returns the *nth* fibonacci number.

```
fib(1)   # 0
fib(2)   # 1
fib(3)   # 1
fib(4)   # 2
fib(5)   # 3
fib(6)   # 5
fib(7)   # 8
fib(8)   # 13
fib(9)   # 21
fib(10)  # 34
```

15) Pascal's Triangle

This is Pascal's triangle:



Except the top 2 rows, each new row can be derived from the previous row:

1. Add each adjacent pair of numbers
2. Insert 2 1's at both ends

Write a recursive function *pascal(n)* that takes in an integer n, and returns the *nth* row of Pascal's triangle as a list of numbers.

```
pascal(1)   # [1]
pascal(2)   # [1, 1]
pascal(3)   # [1, 2, 1]
pascal(4)   # [1, 3, 3, 1]
pascal(5)   # [1, 4, 6, 4 ,1]
pascal(6)   # [1, 5, 10, 10, 5, 1]
pascal(7)   # [1, 6, 15, 20, 15, 6, 1]
pascal(8)   # [1, 7, 21, 35, 35, 21, 7, 1]
```

16) Switching Every 2 Numbers

You are given a list of integers. Write a recursive function *switch2(lis)* that takes in this list of integers, and switches every 2 elements in the list.

Note – If the length of the input list is an odd number, simply ignore the last element.

```
switch2([1, 2, 3])          # [2, 1, 3]
switch2([1, 2, 3, 4])       # [2, 1, 4, 3]
switch2([1, 2, 3, 4, 5])    # [2, 1, 4, 3, 5]
switch2([1, 2, 3, 4, 5, 6]) # [2, 1, 4, 3, 6, 5]
```

17) Plus Signs

Write a recursive function *plus(n)* that takes in a positive integer n, and returns a string containing all numbers from 1 to n, with + characters in between them.

```
plus(1)   # 1
plus(2)   # 1+2
plus(3)   # 1+2+3
plus(4)   # 1+2+3+4
plus(5)   # 1+2+3+4+5
```

18) Alternate Signs

Write a recursive function *alternate(n)* that takes in a positive integer n, and returns a string containing all number from 1 to n, with alternating + and - characters in between them (starting with a + character)

```
alternate(1)     # 1
alternate(2)     # 1+2
alternate(3)     # 1+2-3
alternate(4)     # 1+2-3+4
alternate(5)     # 1+2-3+4-5
```

19) Counting Elements In Nested List

You are given a messy nested list of numbers. Elements are either integers, list of integers, or nested list of integers.

Write a recursive function count(lis) that takes in this nested list, and counts the total number of integers inside the entire nested list.

```
lis = [1, 2, 3, 4, 5]
count(lis)    # 5

lis = [1, 2, [3, 4], 5]
count(lis)    # 5

lis = [[1, 2], [3, 4], 5]
count(lis)    # 5

lis = [1, [2, [3, [4], 5]]]
count(lis)    # 5

lis = [[[1], [2], [3], [4], [5]]]
count(lis)    # 5
```

20) Sum of Nested List

You are given a messy nested list similar to that in question 19. The nested list contains only integers, lists of integers, or nested lists of integers.

Write a recursive function *nested_sum(lis)* that takes in a nested list as stated above, and returns the sum of all integers inside that list.

```
lis = [1, 2, 3, 4, 5]
nested_sum(lis)    # 15

lis = [1, 2, [3, 4], 5]
nested_sum(lis)    # 15

lis = [[1, 2], [3, 4], 5]
nested_sum(lis)    # 15

lis = [1, [2, [3, [4], 5]]]
nested_sum(lis)    # 15

lis = [[[1], [2], [3], [4], [5]]]
nested_sum(lis)    # 15
```

ANSWERS

1) Factorial

```
def factorial(n):
    if n == 1:
        return 1

    return n * factorial(n-1)
```

How this works:

```
# Base case
if n == 1, We simply return 1

# Recursive step
return n * factorial(n-1)

# Using n=5

factorial(1) = 1

factorial(2) = 2 x factorial(1)
             = 2 x 1
             = 2

factorial(3) = 3 x factorial(2)
             = 3 x 2
             = 6

factorial(4) = 4 x factorial(3)
             = 4 x 6
             = 24

factorial(5) = 5 x factorial(4)
             = 5 x 24
             = 120
```

2) Summation

```
def summation(n):
    if n == 1:
        return  1

    return n + summation(n-1)
```

How this works:

```
# Base case
if n == 1, We simply return 1

# Recursive step
return n + summation(n-1)

# Using n=5

summation(1) = 1

summation(2) = 2 + summation(1)
             = 2 + 1
             = 3

summation(3) = 3 + summation(2)
             = 3 + 3
             = 6

summation(4) = 4 + summation(3)
             = 4 + 6
             = 10

summation(5) = 5 + summation(4)
             = 5 + 10
             = 15
```

3) Reversing Uppercase And Lowercase

```
def reverse(string):
   if len(string) == 0:
       return ''

   first = string[0]
   if first.isupper():
       return first.lower() + reverse(string[1:])

   return first.upper() + reverse(string[1:])
```

How this works:

```
# Base case
if string is empty, we return an empty string

# Recursive case
if first is uppercase,
    we return first.lower() + reverse(string[1:])

if first is lowercase,
    we return first.upper() + reverse(string[1:])

# Using string = 'Abc'

reverse('') = ''

reverse('c') = 'C' + reverse('')
             = 'C' + ''
             = 'C'

reverse('bc') = 'B' + reverse('c')
              = 'B' + 'C'
              = 'BC'

reverse('Abc') = 'a' + reverse('bc')
               = 'a' + 'BC'
               = 'aBC'
```

4) Sum Of Odd Numbers

```
def sum_odd(numbers):
   if len(numbers) == 0:
       return 0

   first = numbers[0]
   if first % 2 == 1:
       return first + sum_odd(numbers[1:])

   return sum_odd(numbers[1:])
```

How this works:

```
# Base case
if numbers is empty, return 0

# Recursive Step
if first is odd:
    return first + sum_odd(numbers[1:])

if first is even:
    return sum_odd(numbers[1:])

# eg. numbers = [1, 2, 3, 4, 5]

sum_odd([]) = 0

sum_odd([5]) = 5 + sum_odd([])
             = 5 + 0
             = 5

sum_odd([4, 5]) = sum_odd([5])
                = 5

sum_odd([3, 4, 5]) = 3 + sum_odd([4, 5])
                   = 3 + 5
                   = 8

sum_odd([2, 3, 4, 5]) = sum_odd([3, 4, 5])
                      = 8

sum_odd([1, 2, 3, 4, 5]) = 1 + sum_odd([2, 3, 4, 5])
                         = 1 + 8
                         = 9
```

5) Add Odd Numbers, Subtract Even Numbers

```
def add_sub(numbers):
    if len(numbers) == 0:
        return 0

    first = numbers[0]
    if first % 2 == 1:
        return first + add_sub(numbers[1:])

    return -first + add_sub(numbers[1:])
```

How this works:

```
# Base case
if numbers is empty, return 0

# Recursive case
if first is odd,
    return first + add_sub(numbers[1:])

if first is even,
    return -even + add_sub(numbers[1:])

# numbers = [1, 2, 3, 4]
add_sub([]) = 0

add_sub([4]) = -4 + add_sub([])
             = -4 + 0
             = -4

add_sub([3, 4]) = 3 + add_sub([4])
                = 3 + -4
                = -1

add_sub([2, 3, 4]) = -2 + add_sub([3, 4])
                   = -2 + -1
                   = -3

add_sub([1, 2, 3, 4]) = 1 + add_sub([2, 3, 4])
                      = 1 + -3
                      = -2
```

6) Counting Vowels

```
def count_vowels(string):
    if len(string) == 0:
        return 0

    if string[0] in 'aeiou':
        return 1 + count_vowels(string[1:])
    else:
        return count_vowels(string[1:])
```

How this works:

```
# Base case
if string is empty, we simply return 0

# Recursive case
if first letter is a vowel,
    return 1 + count_vowels(string[1:])

if first letter is NOT a vowel,
    return 0 + count_vowels(string[1:])

# string = 'apple'
count_vowels('') = 0

count_vowels('e') = 1 + count_vowels('')
                  = 1 + 0
                  = 1

count_vowels('le') = count_vowels('e')
                   = 1

count_vowels('ple') = count_vowels('le')
                    = 1

count_vowels('pple') = count_vowels('ple')
                     = 1

count_vowels('apple') = 1 + count_vowels('pple')
                      = 1 + 1
                      = 2
```

7) Removing Vowels

```
def remove_vowels(string):
    if len(string) == 0:
        return ''

    if string[0] in 'aeiou':
        return remove_vowels(string[1:])
    else:
        return string[0] + remove_vowels(string[1:])
```

How this works:

```
# Base case
if string is empty, we simply return an empty string

# Recursive case
if first letter of string is a vowel,
    return remove_vowels(string[1:])

else if first letter of string is NOT a vowel,
    return first_letter + remove_vowels(string[1:])

# string = 'apple'
remove_vowels('') = ''

remove_vowels('e') = '' + remove_vowels('')
                   = '' + ''
                   = ''

remove_vowels('le') = 'l' + remove_vowels('e')
                    = 'l' + ''
                    = 'l'

remove_vowels('ple') = 'p' + remove_vowels('le')
                     = 'p' + 'l'
                     = 'pl'

remove_vowels('pple') = 'p' + remove_vowels('ple')
                      = 'p' + 'pl'
                      = 'ppl'

remove_vowels('apple') = '' + remove_vowels('pple')
                       = '' + 'ppl'
                       = 'ppl'
```

8) Replacing Vowels

```
def replace_vowels(string):
    if len(string) == 0:
        return ''

    d = {'a':'e', 'e':'i', 'i':'o', 'o':'u', 'u':'a'}
    first_letter = string[0]

    if first_letter in d:
        return d[first_letter] + replace_vowels(string[1:])
    else:
        return first_letter + replace_vowels(string[1:])
```

How this works:

```
# Base case
if string is empty, we simply return an empty string

# Recursive case
if first letter is a vowel,
    return corresponding_vowel + replace_vowels(string[1:])

else if first letter is NOT a vowel
    return first_letter + replace_vowels(string[1:])

# string = 'apple'
replace_vowels('') = ''

replace_vowels('e') = 'i' + replace_vowels('')
                    = 'i' + '' = 'i'

replace_vowels('le') = 'l' + replace_vowels('e')
                     = 'l' + 'i' = 'li'

replace_vowels('ple') = 'p' + replace_vowels('le')
                      = 'p' + 'li' = 'pli'

replace_vowels('pple') = 'p' + replace_vowels('ple')
                       = 'p' + 'pli' = 'ppli'

replace_vowels('apple') = 'e' + replace_vowels('pple')
                        = 'e' + 'ppli' = 'eppli'
```

9) Double Letters

```
def double_letters(string):
   if len(string) == 0:
       return ''

   return string[0] + string[0] + double_letters(string[1:])
```

How this works:

```
# Base case
if string is empty, we return an empty string

# Recursive case
return first_letter + first_letter + double_letters(string[1:])

# string = 'apple'
double_letters('') = ''

double_letters('e') = 'e' + 'e' + double_letters('')
                    = 'e' + 'e' + ''
                    = 'ee'

double_letters('le') = 'l' + 'l' + double_letters('e')
                     = 'l' + 'l' + 'ee'
                     = 'llee'

double_letters('ple') = 'p' + 'p' + double_letters('le')
                      = 'p' + 'p' + 'llee'
                      = 'pplee'

double_letters('pple') = 'p' + 'p' + double_letters('ple')
                       = 'p' + 'p' + 'pplee'
                       = 'pppplee'

double_letters('apple') = 'a' + 'a' + double_letters('pple')
                        = 'a' + 'a' + 'pppplee'
                        = 'aapppplee'
```

10) Palindromes

```
def is_palindrome(string):
    if len(string) <= 1:
        return True

    if string[0] == string[-1]:
        return is_palindrome(string[1:-1])
    else:
        return False
```

How this works:

```
# Base case
if string has length of 0 or 1,
    it is definitely a palindrome, so we return True

# Recursive case
We check if the first letter is equal to the last letter

if they are equal:
    return is_palindrome(string[1:-1])
else:
    return False
```

Some examples:

```
# string = 'abcba'
is_palindrome('c') = True

is_palindrome('bcb') = is_palindrome('c') # as 'b'=='b'
                     = True

is_palindrome('abcba') = is_palindrome('bcb') # as 'a'=='a'
                       = True
```

```
# string = 'abca'
is_palindrome('') = True

is_palindrome('bc') = False  # as 'b'!='c'
```

11) Number Pyramid

```
def number_pyramid(n):
    if n == 1:
        print(1)
    else:
        number_pyramid(n-1)
        for i in range(1, n+1):
            print(i, end='')
        print()
```

How this works:

```
# Base case
if n is 1, simply print 1

# Recursive case
if n is larger than 1, we:
    number_pyramid(n-1), then
    print(1, 2, ... n)

# n = 4
number_pyramid(1) # prints:
1                 # base case

number_pyramid(2)
1                 # printed from number_pyramid(1)
12                # printed from number_pyramid(2)

number_pyramid(3)
1                 # printed from number_pyramid(1)
12                # printed from number_pyramid(2)
123               # printed from number_pyramid(3)

number_pyramid(4)
1                 # printed from number_pyramid(1)
12                # printed from number_pyramid(2)
123               # printed from number_pyramid(3)
1234              # printed from number_pyramid(4)
```

12) Reverse Number Pyramid

```
def reverse_pyramid(n):
    if n == 1:
        print(1)
    else:
        for i in range(n, 0, -1):
            print(i, end='')
        print()
        reverse_pyramid(n-1)
```

How this works:

```
# Base case
if n is 1, simply print 1

# Recursive case
if n is larger than 1, we:
    print(n, n-1, ... 1) then
    reverse_pyramid(n-1)

# n = 4
number_pyramid(1) # prints:
1                 # base case

number_pyramid(2)
21                # printed from number_pyramid(2)
1                 # printed from number_pyramid(1)

number_pyramid(3)
321               # printed from number_pyramid(3)
21                # printed from number_pyramid(2)
1                 # printed from number_pyramid(1)

number_pyramid(4)
4321              # printed from number_pyramid(4)
321               # printed from number_pyramid(3)
21                # printed from number_pyramid(2)
1                 # printed from number_pyramid(1)
```

13) Spaced Pyramid

```
def spaced_pyramid(n, length=None):
    if length is None:
        length = n

    if n == 1:
        print(' '*(length-1) + '1')
    else:
        spaced_pyramid(n-1, length=length)
        print(' '*(length-n), end='')
        for i in range(1, n+1):
            print(i, end='')
        print()
```

Note – The *length* variable is shared amongst all recursive calls. How this works:

```
# Base case
if n is 1, simply print 1 with (length-1) spaces in front

# Recursive case
if n is larger than 1, we:
    print(1, 2, ... n) with (length-n) spaces in front, then
    reverse_pyramid(n-1, length=length)

# n = 4                         # length=4 is shared among all calls
spaced_pyramid(1, length=4)
   1                            # base case

spaced_pyramid(2, length=4)
   1                            # from spaced_pyramid(1, length=4)
  12                            # from spaced_pyramid(2, length=4)

spaced_pyramid(3, length=4)
   1                            # from spaced_pyramid(1, length=4)
  12                            # from spaced_pyramid(2, length=4)
 123                            # from spaced_pyramid(3, length=4)

spaced_pyramid(4, length=4)
   1                            # from spaced_pyramid(1, length=4)
  12                            # from spaced_pyramid(2, length=4)
 123                            # from spaced_pyramid(3, length=4)
1234                            # from spaced_pyramid(4, length=4)
```

14) Fibonacci Numbers

```
def fib(n):
   if n == 1:
       return 0

   if n == 2:
       return 1

   return fib(n-1) + fib(n-2)
```

How this works:

```
# Base case 1
if n is 1, we return the first fibonacci number 0

# Base case 2
if n is 2, we return the second fibonacci number 1

# Recursive case
fib(n) = fib(n-1) + fib(n-2)

# n = 8
fib(1) = 0          # base case 1

fib(2) = 1          # base case 2

fib(3) = fib(2) + fib(1) = 1 + 0
                         = 1

fib(4) = fib(3) + fib(2) = 1 + 1
                         = 2

fib(5) = fib(4) + fib(3) = 2 + 1
                         = 3

fib(6) = fib(5) + fib(4) = 3 + 2
                         = 5

fib(7) = fib(6) + fib(5) = 5 + 3
                         = 8

fib(8) = fib(7) + fib(6) = 8 + 5
                         = 13
```

15) Pascal's Triangle

```
def pascal(n):
    if n == 1:
        return [1]

    if n == 2:
        return [1, 1]

    previous = pascal(n-1)
    new = []
    for i in range(len(previous)-1):
        left = previous[i]
        right = previous[i+1]
        new.append(left+right)

    return [1] + new + [1]
```

How this works:

```
# Base case 1
if n is 1, return the first row [1]

# Base case 2
if n is 2, return the second row [1, 1]

# recursive case
1) first generate the previous row
2) Add every 2 adjacent numbers together
3) add 2 1's at both ends of the list

eg. [1,2,1] → [3, 3] -> [1,3,3,1]

# n = 5
pascal(1) = [1]          # base case 1
pascal(2) = [1, 1]       # base case 2

pascal(3) = [1, 2, 1]
    [1, 1] -> [2] -> [1, 2, 1]

pascal(4) = [1, 3, 3, 1]
    [1, 2, 1] -> [3, 3] -> [1, 3, 3, 1]

pascal(5) = [1, 4, 6, 4, 1]
    [1, 3, 3, 1] -> [4, 6, 4] -> [1, 4, 6, 4, 1]
```

16) Switching Every 2 Numbers

```
def switch2(lis):
   if len(lis) <= 1:
       return lis

   return lis[:2][::-1] + switch2(lis[2:])
```

How this works:

```
# base case
if the list has 0 or 1 elements, simply return the list itself.
    (no switching required)

# recursive case
given a list [a, b, c, ...],
    switch the first 2 elements, and
    return [b, a,]

# lis = [1, 2, 3, 4]
switch2([]) = []

switch2([3, 4]) = [4, 3] + switch2([])
                = [4, 3] + [] = [4, 3]

switch2([1, 2, 3, 4]) = [2, 1] + switch2([3, 4])
                      = [2, 1] + [4, 3]
                      = [2, 1, 4, 3]

# lis = [1, 2, 3, 4, 5]
switch2([5]) = [5]

switch2([3, 4, 5]) = [4, 3] + switch([5])
                   = [4, 3] + [5] = [4, 3, 5]

switch2([1, 2, 3, 4, 5]) = [2, 1] + switch2([3, 4, 5])
                         = [2, 1] + [4, 3, 5]
                         = [2, 1, 4, 3, 5]
```

17) Plus Signs

```
def plus(n):
  if n==1:
      return '1'

   return plus(n-1) + f'+{n}'
```

How this works:

```
# Base case
if n is 1, we simply return the string '1'

# Recursive case
plus(n) = plus(n-1) + '+' + str(n)

# n = 5
plus(1) = '1'

plus(2) = plus(1) + '+2'
        = '1' + '+2'
        = '1+2'

plus(3) = plus(2) + '+3'
        = '1+2' + '+3'
        = '1+2+3'

plus(4) = plus(3) + '+4'
        = '1+2+3' + '+4'
        = '1+2+3+4'

plus(5) = plus(4) + '+5'
        = '1+2+3+4' + '+5'
        = '1+2+3+4+5
```

18) Alternate Signs

```python
def alternate(n):
    if n == 1:
        return '1'

    if n%2 == 0:
        return alternate(n-1) + f'+{n}'
    else:
        return alternate(n-1) + f'-{n}'
```

How this works:

```
# Base case
if n is 1, simply return the string '1'

# Recursive case
if n is even, alternate(n) = alternate(n-1) + f'+{n}'
if n is odd, alternate(n) = alternate(n-1) + f'-{n}'

# n = 5
alternate(1) = '1'

alternate(2) = alternate(1) + '+2'
             = '1' + '+2'
             = '1+2'

alternate(3) = alternate(2) + '-3'
             = '1+2' + '-3'
             = '1+2-3'

alternate(4) = alternate(3) + '+4'
             = '1+2-3' + '+4'
             = '1+2-3+4'

alternate(5) = alternate(4) + '-5'
             = '1+2-3+4' + '-5'
             = '1+2-3+4-5'
```

19) Counting Elements In Nested List

```
def count(lis):
    if type(lis) != list:
        return 1

    total = 0
    for element in lis:
        total += count(element)
    return total
```

How this works:

```
# Base case
if lis is not a list (meaning that it is an integer etc), return 1

# recursive case
return the sum of count() of all its elements

# lis = [1, [2], [3, 4], [[5]]]
count([1, [2], [3, 4], [[5]]]) = count(1) + count([2]) +
                                 count([3, 4]) + count([[5]])

# computing the inner stuff
count(1) = 1

count([2]) = count(2)
           = 1

count([3, 4]) = count(3) + count(4)
              = 1 + 1
              = 2

count([[5]]) = count([5])
             = count(5)
             = 1

# hence
count([1, [2], [3, 4], [[5]]]) = count(1) + count([2]) +
                                 count([3, 4]) + count([[5]])
                               = 1 + 1 + 2 + 1
                               = 5
```

20) Sum of Nested List

```
def nested_sum(lis):
   if type(lis) != list:
       return lis

   total = 0
   for element in lis:
       total += nested_sum(element)
   return total
```

How this works (quite similar to Q19):

```
# Base case
if lis is not a list, return itself

# recursive case
return the sum of count() of all its elements

# lis = [1, [2], [3, 4], [[5]]]
nested_sum([1, [2], [3, 4], [[5]]]) = nested_sum(1) + nested_sum([2]) +
                                      nested_sum([3, 4]) +
                                      nested_sum([[5]])

# computing the inner stuff
nested_sum(1) = 1

nested_sum([2]) = nested_sum(2)
                = 2

nested_sum([3, 4]) = nested_sum(3) + nested_sum(4)
                   = 3 + 4
                   = 7

nested_sum([[5]]) = nested_sum([5])
             = nested_sum(5)
             = 5

# hence
nested_sum([1, [2], [3, 4], [[5]]]) = nested_sum(1) + nested_sum([2]) +
                                      nested_sum([3, 4]) +
                                      nested_sum([[5]])
                                    = 1 + 2 + 7 + 5
                                    = 15
```

Conclusion

Congratulations for making it to the end of this PDF file. Hope these questions got your recursive muscle working, and hope that the answers and explanations were clear.