0 Recursion & Fuction Practice: Sum

Write a recursive function $my\_sum(lst)$ that sums up a list. Check that it works just like python's $sum(lst)$.

hint 1: Assume $my\_sum(shorter\_list)$ somehow gives you the correct value magically. Then what?

hint 2: $assert(boolean)$ would raise AssertionError if the condition is False, and nothing happens if the condition is True.

```
def my_sum(lst):
    # NOTE: sum([]) == 0
    if len(lst) == 0:
      return 0
    else:
      # FILL ME IN

l = # some random list
assert(my_sum(l) == sum(l))
```

1 Recursion & Fuction Practice: Max

Write a recursive function $my\_max(lst)$ that returns maximum value of a list. Check that it works just like python's $max(lst)$.

```
def my_max(lst):
    # NOTE: max([]) is not defined.
    if len(lst) == 1:
      return lst[0]
    else:
      # FILL ME IN

l = # some random list
assert(my_max(l) == max(l))
```

2 Recursion & Fuction Practice: Binary to Decimal

Write a recursive function $bin\_to\_dec(binary)$ that returns value of a binary number(represented as a string) evaluted to a decimal number. Check that it works just like python's $int(binary,\ 2)$.(surprised?)

hint 1: $int(binary,\ 2)$, $int(ternary,\ 3)$ $int(hexadecimal,\ 16)\ldots$ you name it. The default is $int(n,\ 10)$.

```
def bin_to_dec(binary):
    if binary == '0':
      return 0
    elif binary == '1':
      return 1
    else:
      # FILL ME IN

binary_string = input()
print(bin_to_dec(binary_string) == int(binary_string, 2))
```

3 Recursion & Fuction Practice: Reverse

Write a recursive $reverse(lst)$ that returns the reversed list.

```
def reverse(lst):
    #FILL_ME_IN

l = # random list
assert(reverse(l) == l[::-1])
```

4 Recursion & Fuction Practice: Is Palindrome?

Write a recursive $is\_palindrome(s)$ that returns True if a string is palindrome, and False otherwise.

hint: The following recursive definition of a palindrome would help.

$$\begin{aligned} \text{P} &\rightarrow \text{"" (empty)} \\ &\mid \text{c + P + c} \end{aligned}$$

```
def is_palindrome(s):
    #FILL_ME_IN

s = input()
print(is_palindrome(s))
```

5 Partitioning

Calculate the number of cases of partitioning number N with 1,2,3.

$$\begin{aligned} 4 &= 1+1+1+1 \\ &= 1+1+2 \\ &= 1+2+1 \\ &= 1+3 \\ &= 2+1+1 \\ &= 2+2 \\ &= 3+1 \end{aligned}$$

```
4
7
```

Optional: Can it calculate numbers like 100? Try memoization.

6 Prefix Calculator

We usually write mathematical expressions this way: $3+4*2$. This is called an infix notation, and although easy to understand, it has some drawbacks.

1. We have to know the precedence of operators to correctly compute the expression. Without knowing that '*' comes before '+', $3+4*2$ can be miscalculated as 14.
2. We have to use parentheses to express the intended order of operations. $(3+4)*2$ if we want the result to be 14.

Therefore, there are other notations like prefix notations(operators precede their operands), or postfix notations(operators follow their operands). For example, the expression above would be $(+\ 3\ (*\ 4\ 2))$ in prefix notations.

To evaluate this expression, we start from the left, and when we see an operator('+'), we expect two sub-expressions. We see 3 and (* 4 2). 3 is 3. For (* 4 2), we repeat the step above, with an operator('*') and two sub-expressions 4 and 2. Since there are no more sub-expressions to be evaluated, we calculate $(*\ 4\ 2) = 8$, and $(+\ 3\ 8) = 11$.

TLDR; we are going to implement a mathematical expression calculator. For your convenience, I have already implemented the $infix\_to\_prefix(exp)$ (There is no need to understand the code). It's your job to fill in $calculate(exp)$ that evaluates the prefix notation, *recursively*.

INPUT: A single line of a correct mathematical expression, with or without parentheses.

OUTPUT: The result of calculating the mathematical expression.

CONDITION: For simplicity, the numbers used in the expression are single digit, ranging from 0 to 9. The operators used are $+(addition)$, $-(substraction)$, $*(multiplication)$, and $/(division)$.

EXAMPLE:

```
(3+2)*2
10

3+2*2
7
```

hint: Before implementing $calculate(exp)$, examine the prefix notation of expressions. We can see that the prefix expression is formed as following(we call this structure the tree structure):

$$\begin{array}{rcl} exp \rightarrow & (operand,\ exp,\ exp) & \text{: tuple} \\ | & n & \text{: integer} \end{array}$$

When the type of the exp is integer, (in Python: $isinstance(prefix\_exp,\ int)$), there is nothing to do; just return n.

When it's a tuple, we have to do more. Again, assuming that $calculate(smaller\_exp)$ will magically calculate the results would help.

7 Challenge: Compressing DNAs

DISCLAIMER: This problem can be challenging. Also, I am not a biology major so if there is anything wrong here, please forgive me.

DNAs are composed of a, g, c, t(am I right?). They are very long(am I right?). Therefore, there is ongoing research on compressing DNA sequences for data efficieny. One method of compressing DNAs utilizes the fact that multiple DNAs share common patterns(am I right?).

For example, a**gctga**ccgt and cc**gctga**tt share **gctga**. If we can find the *longest common substring* of two DNAs, we can simply replace it(= gctga) with X. This way, agctgaccgt becomes aXccgt, and ccgctgatt becomes ccXtt.

Write a program that finds the longest common substring of two DNA sequences.

```
agctgaccg
tccgctgatt

gctga
```

8 Challenge: Decompressing DNAs

DISCLAIMER: I am not a biology major so if there is anything wrong here, please forgive/tell me.

Another compression method focuses on a single strain of DNA. We can compress a DNA sequence by writing out the repeated pattern in the following manner: $K(S)$. K is a $single$ digit integer indicating the number of repetition, and S is the repeated sequence, which can be compressed as well.

For example, $A3(CGT)3(A)2(3(C)GT)$ could be decompressed as
$A-CGT-CGT-CGT-AAA-CCCGT-CCCGT$ (- added for readability).

Write a code that prints the DNA sequence, given the compressed version.
hint: You might want to locate the index of '(' by s.find(), and find the matching ')' by counting the number of '('s and ')'s inside. Using recursion would help.

```
A3(CGT)3(A)2(3(C)GT)
ACGTCGTCGTAAACCCGTCCCGT

3(3(A)G)
AAAGAAAGAAAG
```