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 Binary Search

Remember the Up&Down games during the parties? A participant would guess a number in $range(1, 51)$, and the other player would tell you if the answer is lower, or higher than the guess. The optimal strategy would be to call the median (25→12 if down, 37 if up), that halves the search space everytime.

Binary Search is an algorithm that does exactly this. Write a recursive $bin\_search(lst, n)$, finds n from an already sorted list. It would return True if n is in our list, and False otherwise.

hint: If the median of the list is the value we are looking for, return True. Else, we will have to look at the right half, or the left half.

```
def bin_search(lst, n):
    # FILL_ME_IN


l = sorted(#some random list)
print(bin_search(l, n))
```

7 The idea behind dynamic programming

Given a list of integers, finding its partial sum is easy: $sum(lst[start : end])$. However, if we were to calculate the partial sum 100,000,000 times, it would approximately take $(end - start) * 100,000,000$ times to calculate it.

Write a function $partial\_sum(start, end)$ that calculates $sum(lst[start : end])$ fast, by storing some data in advance.

hint: $sum(lst[start : end])$ can be expressed as the difference of certain two values.

8 Prefix Calculator

Download "calculate.py" located in the same directory as this pdf file. 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):

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

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.

9 Challenge: DP and optimization

DP and optimization go along well, because an optimized solution of smaller problem often lead to an optimized solution of the whole.

Consider the following situation; We want to travel to Europe(*or any country you like*) N days from now, and we are going to work part-time for the tickets. It would be great if we can work every single day, but we have to take a rest - we can't work 3 consecutive days.

Given a list of N integers, each representing the money we can earn, calculate the maximum amount of money we can earn.

```
10 20 30 40 50 60
160
```

hint: We can work 20,30,50,60 to earn 160. This is larger than any other combination($e.g.$ 10,20,40,50). One approach would be a recursive one. Let $solution(d)$ = maximum money earned while working on dth day. Then we can calculate $solution(d)$ by comparing two diffrent cases. One would be adding dth and d-1th payroll with the maximum money we earned 3 days ago, the other would be adding dth payroll with the maximum money we earned 2 days ago.

10 Challenge: Decompressing DNAs

DISCLAIMER: I am not a biology major so if there is anything wrong here, please forgive/tell 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 common patterns occur in a DNA strain (am I right?).

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
```