Introduction to Python Programming
Problem Collection

Ali Kashefi
kashefi@stanford.edu

- Each homework problem is accompanied by a gray box that illustrates the exact format for the sample input and output. It is crucial for you to understand that when you submit your homework, only the input and output values should appear in the console. Do not include additional prompts or messages like "Please enter the name" or anything else.
- You must use the `input()` function to read the input and the `print()` function to generate the output.

Problem 1. Write a Python program that does the following:

- Takes a name as input from the user.
- Outputs "Hello, [User's Name]!" to the console.

This is an example of how the program should work:

Sample Input:

```
Alice
```

Sample Output:

```
Hello, Alice!
```

Problem 2. Write a Python code that takes a non-negative integer n as input and calculates the n-th Pell number (P_n). The Pell numbers are a sequence of integers defined as:

$$\begin{aligned} P_0 &= 0 \quad \text{if } n = 0, \\ P_1 &= 1 \quad \text{if } n = 1, \\ P_n &= 2P_{n-1} + P_{n-2} \quad \text{if } n \geq 2. \end{aligned}$$

For example, if the input is 2 (i.e., $n = 2$), the output should be 2, because $P_2 = 2P_1 + P_0 = 2(1) + 0 = 2$.

Sample Input:

```
0
```

Sample Output:

```
0
```

Sample Input:

```
1
```

Sample Output:

```
1
```

Sample Input:

```
3
```

Sample Output:

```
5
```

Sample Input:

```
5
```

Sample Output:

```
29
```

Problem 3. Goldbach's Conjecture says that every even integer greater than 2 can be expressed as the sum of two prime numbers. Mathematically, it can be stated as: For any even integer $a > 2$, there exist prime numbers p_1 and p_2 such that $a = p_1 + p_2$. Write a Python program that inputs an even number greater than 2, finds two prime numbers (p_1 and p_2) whose sum equals the input number according to Goldbach's Conjecture, and outputs the product of these two prime numbers (i.e., $p_1 \times p_2$). The program should ensure that p_1 is the smallest possible prime number that satisfies the conjecture for the given even number. For example, if the input of your code is 10, the output must be 21, because $10 = 3 + 7$, and $3 \times 7 = 21$. Note that although we can write $10 = 5 + 5$, 25 is not an acceptable answer because 5 is not the smallest possible prime number that can be used to sum up to 10. In this case, 3 is the smallest prime number that, when added to another prime number (7), equals 10. Therefore, the correct output is the product of 3 and 7, which is 21.

If you are not familiar with prime numbers, let's go through a few examples to better understand what prime numbers are:

Number: 2

Divisors: 1, 2

Explanation: The number 2 is divisible only by 1 and itself. It has exactly two distinct positive divisors, so it is a prime number.

Number: 7

Divisors: 1, 7

Explanation: The number 7 is divisible only by 1 and itself. It has exactly two distinct positive divisors, so it is a prime number.

Number: 12

Divisors: 1, 2, 3, 4, 6, 12

Explanation: The number 12 has more than two divisors, including 1, 2, 3, 4, 6, and 12. Therefore, it is not a prime number.

Specific policy: You must define at least one function in your code to handle this problem.
Advice: Define a function such as `isPrime`.

```
def isPrime(number):
    // return True or False ...
```

Sample Input:

```
10
```

Sample Output:

```
21
```

Sample Input:

```
1000
```

Sample Output:

```
2991
```

Sample Input:

```
24
```

Sample Output:

```
95
```

Problem 4. A palindrome is a sequence of characters that reads the same forwards as it does backward. When this concept is applied to numbers, a palindrome number is an integer that remains the same when its digits are reversed. In other words, a number is palindromic if it is the same when read from left to right and from right to left. Write a Python program that takes a string of numbers as input and outputs the "minimum number of characters" that should be added to the end of the string to make it a palindrome. For example, for the input string of '123', the output of your code must be 2, because by adding '21' to the end of '123', we will have '12321', which is a palindrome number.

Note: The input to your code must be a string (i.e., `str`) that contains only the digits 0 through 9. The input string can also start with a 0.

Specific policy: You must define at least one function in your code to handle this problem.
Advice: Create a function that checks if a string is a palindrome or not, such as

```
def isPalindrome(number):
    // return True or False ...
```

Sample Input:

```
123
```

Sample Output:

```
2
```

Sample Input:

```
9
```

Sample Output:

```
0
```

Sample Input:

```
01
```

Sample Output:

```
1
```

Sample Input:

```
122
```

Sample Output:

```
1
```

Sample Input:

```
121
```

Sample Output:

```
0
```

Sample Input:

```
999991
```

Sample Output:

```
5
```

Sample Input:

```
112358132523
```

Sample Output:

```
7
```

Sample Input:

```
112344
```

Sample Output:

```
4
```

Problem 5. Multiplicative persistence is the number of times you must multiply the digits of a number together until you reach a single digit. For example, consider the number 39:

step#1: $3 \times 9 = 27$
step#2: $2 \times 7 = 14$
step#3: $1 \times 4 = 4$

The multiplicative persistence of 39 is 3 because it took three steps to reduce it to a single-digit number. Write a Python program that takes a positive integer number and returns its multiplicative persistence.

Specific policy: You must define at least one function in your code to handle this problem.
Advice: Create a function that calculates the product of the digits of an integer, such as:

```
def multiplyDigits(number):
    # ...
```

Sample Input:

```
39
```

Sample Output:

```
3
```

Sample Input:

```
7
```

Sample Output:

```
0
```

Sample Input:

```
11112211
```

Sample Output:

```
1
```

Sample Input:

```
2023
```

Sample Output:

```
1
```

Sample Input:

```
277777788888899
```

Sample Output:

```
11
```

Problem 6. Parentheses are symbols used in programming, mathematics, and various written languages to group elements or information together. For this exercise, we are concerned with round brackets: ( ). The term "valid parentheses" in the context of round brackets means:

- Every opening parenthesis (must have a corresponding closing parenthesis). They should correctly pair up.
- Parentheses must close in the correct order, respecting their nesting. An opening parenthesis cannot be closed by another opening one, and a closing parenthesis cannot appear before its corresponding opening one.

Write a Python code that takes a string as input from the user and checks if the parentheses (i.e., round brackets) in a given string are valid. If it is valid, the output is `yes`, if it is not valid, the output is `no`. For this specific problem, we assume the string does not contain any space.

Sample Input:

```
(HW)(1)((PIC)))(16A)(University))()
```

Sample Output:

```
no
```

Sample Input:

```
(math((()())Programming(()U)C)L)A(()))((Python))
```

Sample Output:

```
yes
```

Sample Input:

```
((((a)bc)d)efg((h))(i)j)klmno((12345))+-((!!!))(?)?(?)
```

Sample Output:

```
yes
```

Sample Input:

```
)123)4)5(67(8(9)10)1112)13(14(1516(17
```

Sample Output:

```
no
```

Problem 7. Write a Python code that takes a single, space-less string (with at least three characters) as input and determines the sub-string with the highest frequency within it, ensuring that the minimum acceptable length for a sub-string is 2 characters. If there are multiple sub-strings with this highest frequency, the code should output the longest one. Further, if these highest frequency sub-strings are of equal length, your program should opt for the one that appears first in the main string, from left to right.

Advice: Using `list` or/and `dict` is useful to solve this problem.

Sample Input:

```
abcdabefabgh
```

Sample Output:

```
ab
```

Sample Input:

```
Apbcd!cDbcd!LApbcd!!!BcD
```

Sample Output:

```
bcd!
```

Sample Input:

```
UCLA
```

Sample Output:

```
UCLA
```

Sample Input:

```
ArchCircleArchCircleArchCircleArch
```

Sample Output:

```
rc
```

Sample Input:

```
#$%ABBA$%ABBA###$%ABBA()()$%ABBA*$%ABBA&$%ABBA
```

Sample Output:

```
$%ABBA
```

Sample Input:

```
WEPointCloudWEPointCloudPointCloudPointCloudWEPointCloudWEWEPointCloud
```

Sample Output:

```
PointCloud
```

Problem 8. Write a Python code that takes two space-less strings (with at least one character) as input and merges these two strings. A merge occurs when a substring from the end of the first string exactly matches a substring from the beginning of the second string. The program should combine these two strings into one, overlapping the matching substring, and output the merged string. The merged string should be one with the minimum possible length. If a merge is not possible, the program should output the first string unaltered. For instance, merging "Caltech" with "technology" would result in "Caltechnology". As another example, merging "Apple" with "orange" would result in "Apple". For example, merging "AB===" with "===BA" would lead in "AB===BA", and **not** "AB=====BA".
To read the inputs, you **must** use the following format.

```
first_string, second_string = input(), input()
```

Sample Input:

```
Sun

Sunset
```

Sample Output:

```
Sunset
```

Sample Input:

```
Sun

sunset
```

Sample Output:

```
Sun
```

Sample Input:

```
cross

crossroad
```

Sample Output:

```
crossroad
```

Sample Input:

```
sunset

set
```

Sample Output:

```
sunset
```

Sample Input:

```
overlap

lapping
```

Sample Output:

```
overlapping
```

Sample Input:

```
Caltech

technology
```

Sample Output:

```
Caltechnology
```

Sample Input:

```
a

b
```

Sample Output:

```
a
```

Sample Input:

```
%%%$$##

##()@#^&
```

Sample Output:

```
%%%$$##()@#^&
```

Sample Input:

```
A*()*A++==

+==A*()*A
```

Sample Output:

```
A*()*A++==A*()*A
```

Sample Input:

```
AB===

===BA
```

Sample Output:

```
AB===BA
```

Sample Input:

```
Python

Python
```

Sample Output:

```
Python
```

Problem 9. Write a Python code that computes the summation of the Armstrong numbers strictly smaller than any positive integer provided by the user. An Armstrong number of a given number of digits is an integer such that the sum of its own digits raised to the power of the number of digits is equal to the number itself. For instance, 153 is an Armstrong number because:

$$1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$$

For example, if the input is 400, then the output must be 939 because Armstrong numbers smaller than 400 are 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, and 371. Therefore, the sum is $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 153 + 370 + 371 = 939$.

Specific policies: You must define the `main` function to handle this problem. Additionally, you must define at least one function, except `main`, in your code to handle this problem.
Advice: A function can assist you by counting the number of digits in the number you are evaluating, for example:

```
def countDigits(number):
    ## return ...
```

Moreover, you might consider the task of identifying whether a number is Armstrong. Instead of repeating the checking logic every time you need it, you could encapsulate this logic within a dedicated function, such as:

```
def isArmstrong(number):
    ## return True or False
```

Sample Input:

```
400
```

Sample Output:

```
939
```

Sample Input:

```
4
```

Sample Output:

```
6
```

Sample Input:

```
60000
```

Sample Output:

```
75410
```

Problem 10. In this problem, you are required to simulate a scenario involving a number of soldiers standing in a row. Some soldiers are facing to the right, while others are facing to the left. When commanded by their leader, every pair of soldiers standing face-to-face will simultaneously turn around, ending up back-to-back. This process will be repeated until no soldiers are standing face-to-face. Write a Python code that reads input as a string and outputs the number of iterations required to reach the state where no soldiers are facing each other. The input string provides the initial direction of each soldier, using the letters 'L' (for left) and 'R' (for right). Your program should calculate and output the number of turns required to reach the stable state. For example, if the initial state is RLLLL, we have

Input RLLLL
move #1: LRLLL
move #2: LLRLL
move #3: LLLRL
move #4: LLLLR

As another example, if the initial state is LRLLLRLL, we have

Input LRLLLRLL
move #1: LLRLLLRL
move #2: LLLRLLLR
move #3: LLLLRLLR
move #4: LLLLLRLR
move #5: LLLLLLRR

Specific policies: You must define the `main` function to handle this problem. Additionally, you must define at least one function, except `main`, in your code to handle this problem.
Advice: Defining a function that modifies the string of soldiers after each move could be helpful, such as:

```
def move(soldiers):
    ## return modified_soldiers
```

Sample Input:

```
RLLLL
```

Sample Output:

```
4
```

Sample Input:

```
LLL
```

Sample Output:

```
0
```

Sample Input:

```
RRRRRRRRRRRRRR
```

Sample Output:

```
0
```

Sample Input:

```
LLLLRR
```

Sample Output:

```
0
```

Sample Input:

```
LLLRL
```

Sample Output:

```
1
```

Sample Input:

```
LRLLLRLL
```

Sample Output:

```
5
```

Sample Input:

```
RRLLRRRRLLLLLLRL
```

Sample Output:

```
12
```

Sample Input:

```
RLRLRLRLRL
```

Sample Output:

```
5
```

Sample Input:

```
RRLLRRLLLL
```

Sample Output:

```
7
```

Sample Input:

```
RRRRRLLLLL
```

Sample Output:

```
9
```

Sample Input:

```
RLLRLLLRLLLLRLLLLL
```

Sample Output:

```
14
```

Problem 11. Consider an operation, denoted as the $\mathcal{D}$ operator, which processes a string containing a sequence of mathematical terms. Each term in the string consists of a numerical coefficient and a variable `y`, which may be repeated to represent its multiplicity. For example, a term traditionally expressed as $2y^3$ is represented in the string as `2*y*y*y`. The $\mathcal{D}$ operator modifies each term in the string according to the following rules:

- Multiplying the number of `y` in the term by the coefficient and then decreasing the count of `y`s by one.
- Eliminating the term if it does not contain `y`.

For example, a term like `2*y*y*y` is transformed to `6*y*y`, and `-4*y*y*y` becomes `-12*y*y`. A term like `-5`, which does not include `y`, is removed. Write a Python code that receives a string input formatted in this manner, and applies the $\mathcal{D}$ operator to it, and outputs the resulting string. Note that the input string format may include several terms; however, we assume that each term has a distinct number of `y`s. For instance, a string like `2*y*y-5*y+y*y` cannot be an input for the problem because it contains two terms with the same number of `y`s, specifically `2*y*y` and `y*y`. Additionally, the string does not contain spaces between terms. For example, let us consider the following string as input:

```
2*y*y*y-3*y*y+y-5
```

When the $\mathcal{D}$ operator is applied to this string, the following transformations occur:

- `2*y*y*y` is transformed to `6*y*y`.
- `-3*y*y` changes to `-6*y`.
- `y` (implicitly `1*y`) becomes `1`.
- `-5`, as it does not contain `y`, is excluded.

Hence, the output of your code must be `6*y*y-6*y+1`.
Specific policies: You must define the `main` function to handle this problem. Additionally, you must define at least one function, except `main`, in your code to handle this problem.
Advice: To handle this problem, one idea is to split each string into sub-terms and define a function such as:

```
def operatorD(term):
    # return ...
```

Then, concatenate the resulting terms. Note that at some point, you will need to convert strings to integers and vice versa, as you have learned in class.

Sample Input:

```
-5*y*y*y
```

Sample Output:

```
-15*y*y
```

Sample Input:

```
2*y*y*y*y*y*y*y*y+4*y-5
```

Sample Output:

```
14*y*y*y*y*y*y*y+4
```

Sample Input:

```
y*y*y*y*y-123*y*y
```

Sample Output:

```
5*y*y*y*y-246*y
```

Sample Input:

```
2024
```

Sample Output:

Problem 12. In chess, a queen can move any number of squares vertically, horizontally, or diagonally. Write a Python program that takes an 8×8 chessboard configuration with 8 queens and checks whether any of the queens can attack each other. If any of the queens can attack another, the program should print `yes`. If none of the queens can attack any other, it should print `no`. The input will be a two-dimensional `list` (8×8) representing the chessboard. Each element of the list will be either `*` or `Q`, where `*` represents an empty square and `Q` represents a square occupied by a queen. We further assume there will always be exactly 8 queens on the board. You **must** use the following format to get the board from the user:

```
chessboard = []
for i in range(8):
    s = input()
    chessboard += [list(s)]
```

Note that with this format, each input the user enters represents a row of the board as a spaceless string, such as `***Q****`. This process creates the chessboard as a two-dimensional array for you. For instance, after reading the input, you can print your board using the following piece of code.

```
for i in range(8):
    for j in range(8):
         print(chassboard[i][j],end='')
    print()
```

There are different ways to write code to handle this problem; however, one important aspect is that after checking each queen on the chessboard, you do not need to track it again when checking the next queen. One possible approach is to define a data structure (like those you learned in class) containing all the queens' information, and after checking each one, you can remove it from your data list. Note that this is just a suggestion, and you are free to implement this program in any other way you prefer.

Sample Input:

```
***Q****
******Q*
****Q***
**Q*****
Q*******
*****Q**
*******Q
*Q******
```

Sample Output:

```
no
```

Sample Input:

```
***Q****
******Q*
****Q***
*Q******
Q*******
*****Q**
*******Q
***Q****
```

Sample Output:

```
yes
```

Sample Input:

```
Q*******
*****Q**
***Q****
*Q******
******Q*
****Q***
**Q*****
*******Q
```

Sample Output:

```
yes
```

Sample Input:

```
Q*******
******Q*
****Q***
*******Q
*Q******
***Q****
*****Q**
**Q*****
```

Sample Output:

```
no
```

Problem 13. In chess, the knight moves in an L-shape: either two squares in a horizontal direction and then one square vertically, or two squares in a vertical direction and then one square horizontally. This unique movement allows the knight to 'jump over' other pieces. Write a Python program that takes the configuration of a chessboard with only knights and checks whether any of the knights can attack each other in a single move. If any knight can attack another, the program should print `yes`. If no knight can attack another, it should print `no`. The input will be a two-dimensional `list` (4×4) representing the chessboard. Each element of the list will be either `*` or `K`, where `*` represents an empty square and `K` represents a square occupied by a knight. We further assume the number of knights on the board can vary from 0 to 16. You **must** use the following format to get the board from the user:

```
chessboard = []
for i in range(4):
    s = input()
    chessboard += [list(s)]
```

Note that with this format, each input the user enters represents a row of the board as a space-less string, such as `**K*`. This process creates the chessboard as a two-dimensional array for you. For instance, after reading the input, you can print your board using the following piece of code.

```
for i in range(4):
    for j in range(4):
         print(chassboard[i][j],end='')
    print()
```

Sample Input:

```
****
**K*
****
K***
```

Sample Output:

```
no
```

Sample Input:

```
****
****
KKKK
****
```

Sample Output:

```
no
```

Sample Input:

```
****
****
****
****
```

Sample Output:

```
no
```

Sample Input:

```
***K
*K**
****
K***
```

Sample Output:

```
yes
```

Sample Input:

```
*K**
*K**
K*K*
*K**
```

Sample Output:

```
yes
```