Report of ESE5023_Assignments_#1

Name: Huang Pizhu

SID:12332298

Date:2023/10/24

1.Flowchart

Write a function `Print_values()` with arguments a, b, and c accoding to the flowchart with conditional statement.

```
def Print_values(a,b,c):
    if a>b:
        if b>c:
            return a,b,c
        else:
            if a>c:
                return a,c,b
            else:
                return c,a,b
    else:
        if b>c:
            return None
        else:
            return c,b,a
```

Report the output with a,b and c that can express code logic.

a	b	c	output
5	3	1	(5,3,1)
5	1	3	(5,3,1)
3	1	5	(5,3,1)
1	5	1	None
1	3	5	(5,3,1)

2.Matrix multiplication

2.1

Use `random.randint()` to generate the random numbers.
loop `for i in range()` to fill out rows and comlums.

```
import random
M1 = [[random.randint(0, 50) for _ in range(10)] for _ in range(5)]
M2 = [[random.randint(0, 50) for _ in range(5)] for _ in range(10)]
```

2.2

Initialize the shape of the result matrix based on the two matrices being mulplied. Use nested loops to teaverse and fill each position with value in the result maxtrix.

```
def Matrix_multip(M1,M2):
    result = [[0 for _ in range(len(M2[0]))] for _ in range(len(M1))]
    for i in range(len(M1)):
        for j in range(len(M2[0])):
            for k in range(len(M2)):
                result[i][j] += M1[i][k] * M2[k][j]
    return result
```

3.Pascal triangle

I got the mathematical version of Pascal triangle by reading "https://www.mathsisfun.com/pascals-triangle.html"
The function Pascal triangle(k), when k>1, for kth row, initialize the triangle shape and use nested loops to teaverse ith-row in per position(jth-comlum). The jth number in the ith row is sun of the (m-1)th and mth numbers in the (n-1)th row.
Pascal triangle(100) and Pascal triangle(200) has output as annotation in PS1_3.py

```
def Pascal_triangle(k):
    if k <= 0:
         return "parameter must be an integer more than zero"
    elif k == 1:
        return [[1]]
    else:
        triangle = [[1]]
        for i in range(2, k + 1):
            row = [1]
            for j in range(1, i - 1):
                row.append(triangle[i-2][j-1] + triangle[i-2][j])
            row.append(1)
            triangle.append(row)
    return triangle[k-1]
```

4.Add or double

Condsider path form number x to 1, double operation is more than add 1 in case of number more than 1. If x is even, one step and recursively call the function with x//2. If x is odd, one step that x-1 and recursively call the function with x-1. The function will recursively calculate the minimum number of steps reducing x to 1.(*I learned about the **recursion** of thought from this passage accessed online at https://blog.csdn.net/dreamispossible/article/details/90552557*)

```
def Least_moves(x):
    if x == 1:
        return 0
    elif x % 2 == 0:
        return 1 + Least_moves(x // 2)
    else:
        return 1 + Least_moves(x - 1)
```

5.Dynamic programming

5.1

In the function `find_expressions(target)`, first define numbers' string and three state("+", "-", and " ") and initialize a list to save valid expressions. For number '12345678', per number has three state behind, so there are total 24 state in 8 positions between '123456789', using `itertools.product()` from itertools package to generate this 24 states. Start with 1, everytimes get a state from 3 states in a position, totally 3**8 expression. IN for loop, `product()` generate the three state in 8 position; in every loop, concatenate numbers '12345678' and states by `zip()` and number 9 is appended in end; use `eval()` to decide the string expression and target. (*I got information of some python function to solve this problem, its java code accessed online at https://blog.csdn.net/tao617/article/details/107547933*)

```
import itertools
def find_expressions(target):
    num, ope, expressions = "123456789", ["+", "-", ""], []
    for i in itertools.product(ope, repeat=len(num) - 1):
        expression = "".join(j + k for j, k in zip(num, i)) + num[-1]
        if eval(expression) == target:
            print(f"{expression} = {target}")
    return expressions
```

5.2

Rewrite the function in 5.1 to return the lenth of different targets, which is number of expression.

```
def Total_solutions():
    def new_find_expressions(target):
        num, ope, expressions= "123456789", ["+", "-", ""], []
        for i in itertools.product(ope, repeat=len(num) - 1):
            expression = "".join(j + k for j, k in zip(num, i)) + num[-1]
            if eval(expression) == target:
                expressions.append(f"{expression} = {target}")
        return len(expressions)
    results = []
    for target in range(1, 101):
        num_of_expressions = new_find_expressions(target)
        results.append(num_of_expressions)
    return results
Total_solutions = Total_solutions()
```

Plot a bar chart of numbers and their valid expressions.

```
import matplotlib.pyplot as plt
import numpy as np
x = range(1, 101)
plt.figure(figsize=(18, 5))
plt.bar(x, Total_solutions, color='forestgreen', alpha=0.7, label='Number of
Expressions')
plt.xticks(np.arange(1, 102,2))
plt.yticks(np.arange(int(min(Total_solutions)), int(max(Total_solutions)) +
1))
plt.xlabel('Target Value',fontsize=16)
plt.ylabel('Number of Expressions',fontsize=16)
plt.title('Number of Expressions for Targets 1-100',fontsize=16)
plt.legend()
plt.show()
```

Find the max and min in the list saving number of expressions:
maximum is 26 for target numbers 1 and 45
minimun is 6 for target number 88

```
max_value = max(Total_solutions)
min_value = min(Total_solutions)

max_indices = [index + 1 for index, value in enumerate(Total_solutions) if
value == max_value]
min_indices = [index + 1 for index, value in enumerate(Total_solutions) if
value == min_value]

print(f"maximum: {max_value}, number: {max_indices}")
print(f"minimum: {min_value}, number: {min_indices}")
```