Assignment 01

YuGuobin 12332284

1. Flowchart

a) Generate and pass three integer parameters a, b, and c using "np.random.randint," then print these values in a specific order based on their size relationship using if-else statements.

b) It's worth noting that the flowchart is not complete; after the first check where a > b is false and b > c is true, there are no further checks. Therefore, the following program may sometimes print nothing.

```
import numpy as np
def Print_values(a,b,c):
    if a>b:
        if b>c:
            print(a,b,c)
        elif a>c:
            print(a,c,b)
        else:
            print(c,a,b)
    else :
       if b>c:
           print()
       else:
           print(c,b,a)
Print_values(np.random.randint(1,100),np.random.randint(1
,100),np.random.randint(1,100))
```

2. Matrix multiplication

a) First, define a function named "Matrix_multip" that takes two matrices, x and y, as parameters.

b) Next, determine the number of rows (m) and columns (p) for matrix x, and the number of rows (n) and columns (q) for matrix y.

c) Create a zero matrix to store the result, which will have m rows and n columns.

d) Then, iterate through the rows and columns of the result matrix, apply the matrix multiplication formula to calculate the result 's,' and assign it to the corresponding position in the result matrix.

```python
import numpy as np
def Matrix_multip(x,y):
    m,p = x.shape
    p,n = y.shape
    result = np.zeros((m,n))
    for i in range(m):
        for j in range(n):
            s = 0
            for a in range(p):
                e = x[i,a]*y[a,j]
                s =s+e
            result[i,j]= s
    print(result)
M1 = np.random.randint(0, 51, size=(5, 10))
M2 = np.random.randint(0, 51, size=(10, 5))
Matrix_multip(M1,M2)
```

3. Pascal triangle

a) According to the properties of Pascal's triangle, each number is equal to the sum of the two numbers directly above it in the previous row.
b) In other words, the (i+1)-th number in the (n+1)-th row is the sum of the (i-1)-th number and the i-th number in the n-th row.
c) Starting from the third row, each row begins and ends with 1, and the middle numbers are the sum of the two numbers directly above them in the previous row.
d) First define each row of the first number and then use this rule to calculate the middle part.
e) Finally, add the last element.

I got inspired by reading:
https://blog.csdn.net/W_chuanqi/article/details/123679167

```python
def Pascal_triangle(k):
    top = [[1],[1,1]]
    for i in range(2,k):
        pre = top[i-1]
        lines = [1]
        for j in range(i-1):
            lines.append(pre[j]+pre[j+1])
        lines.append(1)
        top.append(lines)
    print(lines)
Pascal_triangle(100)
Pascal_triangle(200)
```

4. Add or double

a) If n is less than or equal to 3, return n - 1, as these three numbers, 1, 2, and 3, require 0, 1, and 2 steps to reach, respectively.
b) If n is greater than 3, create a list T with a length of n+1 to store the minimum steps for each number.
c) Initialize the first three elements of T because 1, 2, and 3 require 0, 1, and 2 steps to reach, respectively.
d) Starting from 4, use a loop to fill the T list, calculating the minimum steps for each number.
e) For odd number i, use 1 + T[i - 1] because you can only add one.
f) For even number i, you have a choice to either add one or double the number. Therefore, choose 1 + min(T[i - 1], T[i / 2]), where T[i - 1] represents the addition operation, and T[i /2] represents the doubling operation, and choose the minimum of the two.
g) Finally, return T[n], which represents the minimum steps required to reach the number n.

I got inspired by reading:
https://blog.csdn.net/OldDriver1995/article/details/105529928

```python
def dp(n):
    if n <= 3:
        return n - 1
    else:
        T = [0 for i in range(n + 1)]
        T[1], T[2], T[3] = 0, 1, 2
        for i in range(4, len(T)):
            if i % 2 != 0:
                T[i] = 1 + T[i - 1]
            else:
                T[i] = 1 + min(T[i - 1], T[int(i / 2)])
        return T[n]
print(dp(5))
```

5. Dynamic programming

5.1 .

a) First, define a string containing the digits "123456789" and a list of operators ["", "+", "-"]. These digits will be used to construct expressions.
b) Use the itertools.product function to generate all possible combinations of operators. The repeat=8 parameter specifies that operators should be arranged in combinations of 8 because there are 8 operator positions. This will create an iterator of nested tuples, with each tuple representing a possible arrangement of operators.
c) Iterate through the generated operator combinations. For each combination, create an empty string expression and then add digits and operators one by one to

construct a complete expression.

d) Use the eval function to calculate the result of this expression.

e) If the result is equal to the given number x, output the relationship between the expression and x, and print the expressions that satisfy this condition.

I got inspired by reading:
https://blog.csdn.net/weixin_44093555/article/details/121384424,
https://www.runoob.com/python/python-func-eval.html
https://blog.csdn.net/zhangyunwei_blog/article/details/105335446

```
from itertools import product
def Find_expression(x):
    digits = "123456789"
    operators = ["", "+", "-"]
    operator_comb = product(operators, repeat=8)
    for ops in operator_comb:
        expression = ""
        for i in range(8):
            expression += digits[i] + ops[i]
        expression = expression + digits[-1]
        result = eval(expression)
        if result == x:
            print(expression + " = " + str(x))
Find_expression(50)
```

5.2 .

f) Create a list named Total_solutions containing 100 zeros, to store the count of different expressions for each integer from 1 to 100.

g) If the result falls within the range of 1 to 100, increment the index in the corresponding Total_solutions list, indicating that the integer has an expression meeting the criteria.

h) Return Total_solutions, which contains the count of different expressions for each integer from 1 to 100.

i) Identify the integers in Total_solutions with the maximum and minimum values, along with their counts. Traverse the Total_solutions list to find the integers with the highest and lowest counts.

```
from itertools import product

def Find_expression():
    digits = "123456789"
    operators = ["", "+", "-"]
    operator_comb = product(operators, repeat=8)
    Total_solutions = [0] * 100
    for ops in operator_comb:
        expression = ""
        for i in range(8):
            expression += digits[i] + ops[i]
        expression += digits[-1]
        result = eval(expression)

        if 1 <= result <= 100:
            Total_solutions[result - 1] =
Total_solutions[result - 1]+ 1
    return Total_solutions

Total_solutions = Find_expression()
max_solutions = max(Total_solutions)
min_solutions = min(Total_solutions)
max_indices = [i + 1 for i, count in
enumerate(Total_solutions) if count == max_solutions]
min_indices = [i + 1 for i, count in
enumerate(Total_solutions) if count == min_solutions]
print(f"maximum ({max_solutions}): {max_indices}")
print(f"minimum ({min_solutions}): {min_indices}")
```