Lab 7 : Recursion

วัตถุประสงค์ เข้าใจเรื่อง recursion ทดลองเขียนโปรแกรม recursion knapsack problem

ทฤษฎี

1. Iteration :

   หากเราต้องทำงาน n ชิ้น : เราทำงานชิ้นที่ 1 เสร็จ แล้ว ทำงานชิ้นที่ 2 แล้ว ทำงานชิ้นที่ 3 ... ทำไปอย่างนี้เรื่อยๆ จน ทำชิ้นที่ n ชิ้นสุดท้าย เป็นอันเสร็จงาน การทำแบบนี้เรียกว่า ทำแบบ **iteration** iterate คือทำซ้ำ (repeat) ทำทีละอันไปเรื่อยๆ จนหมด

2. Recursion :

   เราสามารถมองปัญหาได้อีกแบบหนึ่ง คือมองปัญหาให้เป็น ปัญหาแบบเดิมที่เล็กลง หากเราแก้ปัญหาที่ต้องการแต่เล็กกว่าปัญหาตั้งต้นได้ เราก็สามารถหาทางแก้ปัญหาทั้งหมดได้ นั่นคือแนวคิดของ recursion

   ดังนั้น การมองปัญหา ทำงาน n ชิ้น แบบ recursion คือ ต้อง ทำงานน้อยกว่า n ชิ้น ให้ได้ก่อน *หาก น้อยกว่า n คือ n-1* เราต้องทำอีก 1 ชิ้น จึงจะ 'ทำงาน n ชิ้น' เสร็จ การแก้ปัญหาแบบนี้เรียกว่า ทำแบบ **recursion** เป็นการมองปัญหาในรูปแบบของตัวปัญหาเดิมที่เล็กลง ซึ่งทำได้หลายแบบ เช่น

```
work(n) = work(n-1) + do 1 work
work(n) = do 1 work + work(n-1)
work(n) = work(front-half) + work(back-half)
```

   แต่การทำแบบสุดท้าย จะสิ้นเปลืองกว่า 2 แบบแรก ซึ่งเราจะได้เรียนในเรื่อง complexity

   หากดูในรายละเอียด เมื่อ n = 4 :

```
work(4) = do 1 work + work(3)
          work(3) = work(2) + do 1 work
          work(2) = work(1) + do 1 work
                    . . .
```

   ในการเขียนฟังก์ชั่นแบบ recursion เป็นการเรียกตัวเองซ้ำๆกัน แต่หากเรียก recursion ไปเรื่อยๆ จะเกิด infinite loop ดังนั้น ต้องมีเงื่อนไข (condition) ที่เราจะไม่เรียกฟังก์ชั่น recursion กรณีที่ไม่เรียก recursion เรียกว่า base case หรือ simple case ในส่วนที่เรียก recursion เรียก recursion case สำหรับตัวอย่างข้างต้น base case คือ work(1) เราทำ do 1 work ได้โดยไม่ต้อง call recursion ดังแสดงข้างล่าง

```python
def work(n):
    if n==1:
        print('do work ', n)
    else:
        print('do work ', n)    # line 1
        work(n-1)               # line 2
work(5)
```

<u>สรุป</u> ฟังก์ชั่นที่เป็น recursion :

1. ต้องมี parameter
2. recursive call โดยเปลี่ยน parameter
3. การหยุด recursion (base case) ส่วนมาก โดย condition ที่เกี่ยวกับ parameter

จากตัวอย่างที่เรียนในชั่วโมงทฤษฎีในเรื่อง The Eight Queen Problem ทุกครั้งที่เรียกฟังก์ชั่น จะมี stack ของ local variables ของการเรียกครั้งนั้น เมื่อเกิดการย้อนกลับ backtrack มาที่เดิม stack จะจดจำสิ่งแวดล้อมเดิมไว้ นี่ทำให้ recursion เขียน code ได้เร็วกว่า iteration ในกรณีมี branch แตกออกไปหลายๆ กิ่ง ในขณะที่ iteration เขียน code ยากกว่ามาก ทำให้ debug ยากกว่ามากด้วย ดังนั้นในกรณีเช่นนี้ recursion มีประโยชน์มาก

<u>การทดลอง 1</u> ลองทำก่อน ถ้าทำไม่ได้จริงๆ ค่อยดูเฉลยด้านหลัง ดูทีละข้อ

```
def work(n):
    if n==1:
        print('do work ', n)
    else:
        print('do work ', n)    # line 1
        work(n-1)               # line 2
work(5)
```

1. ลองเขียนโปรแกรมตัวอย่าง ลองคิดว่า output จะเป็นอย่างไร และ รันดูผลลัพธ์ output : _____[1]
2. โปรแกรมข้างต้น ลองสลับที่ บรรทัด line 1 และ line 2 คิดว่า output จะเป็นอย่างไร และ รันดูผลลัพธ์ output : _____ นักศึกษาคิดว่าทำไมจึงเป็นเช่นนั้น[2]
3. เขียน recursive `def printNdown(n)` เพื่อพิมพ์เลข n ถึง 1
4. เขียน recursive `def printToN (n)` เพื่อพิมพ์เลข 1 ถึง n
5. เขียน recursive `def sumToN(n)` เพื่อ return ค่า sum ตั้งแต่ 1 ถึง n
6. เขียน recursive `def printForw(...)` และ `def printBack(...)`เพื่อพิมพ์ python list forward และ backward โดยคิด parameter เอง
7. เขียน recursive `def App(...)` เพื่อ append 1 ถึง n เข้าไป python list ตามลำดับ คิด parameter เอง
8. เขียน recursive `def AppB(...)` เพื่อ append n ถึง 1 เข้าไป python list ตามลำดับ คิด parameter เอง

---

[1] ในครั้งแรก n = 5 line 1 พิมพ์ 5 จึง call recursion ซึ่งพิมพ์ 4 และ การเรียกครั้งต่อๆมา พิมพ์ 3 2 และครั้งสุดท้าย base case พิมพ์ 1

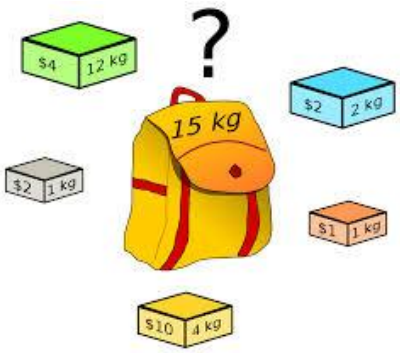
[2] พิมพ์ 1 2 3 4 5 เนื่องจากจะยังไม่พิมพ์จนกว่าจะทำ recursion เสร็จ ดังนั้นจึง recursion ลงไปจนถึง n = 1 พิมพ์ 1 backtrack มาที่ n = 2 พิมพ์ 2 เนื่องจากเป็น stack จึง backtrack ไปที่การ call ครั้งก่อนหน้ามันคือครั้งสุดท้ายบน top ของ stack backtrack ครั้งต่อๆ มาจึงพิมพ์ 5 4 3 ตามลำดับ

9. สร้าง linked list h ขึ้นมาแบบ iterative เขียน recursive `def printList(h)` เพื่อพิมพ์ data ใน node ของ linked list h โดยกำหนด class node ดังนี้

```
class node():
    def __init__(self, d):
        self.data = d
        self.next = None
```

10. เขียน recursive `def createL (...)` เพื่อสร้าง linked list จาก 1 ถึง n คิด parameter เอง ทดสอบโดย `printList()` ในข้อที่แล้ว
11. เขียน recursive `def createLfromlist(...)` เพื่อสร้าง linked list จาก python list คิด parameter เอง

[การทดลอง 2](#) เขียน recursive : knapsack (แนปเสค) problem



มีเงิน k บาท มีของ n ชิ้นราคา b1, b2, ..., bn จะซื้อของให้เงินหมดพอดี (ไม่เหลือและไม่ขาด) ได้หรือไม่ ถ้าได้ ได้ราคาเท่าใดบ้าง (หาทุกค่าที่ซื้อได้) **นั่นคือ B = { b1, b2, ..., bn } เป็น set ของ integers ที่มีค่าซ้ำได้ มี subset ของ B หรือไม่ที่ sum ของสมาชิกทั้งหมด = k พอดี** ของราคาเท่ากันถือเป็นคนละอันกัน ดังนั้น

k = 20, B = {20, 10, 5, 5, 3, 2, 20, 10} outputs จะได้ 9 แบบ ดังแสดง

outputs:

20

10 5 5

10 5 3 2

10 5 3 2

10 10

5 5 10

5 3 2 10

5 3 2 10

20

1. จะใช้ data structure อะไรเก็บ **1.** ของ(ราคา)ทั้งหมดในตลาด **2.** ของ(ราคา)ใน sack
2. recursive part คืออะไร ==> โจทย์คืออะไร อะไรที่ทำซ้ำๆ กัน เพียงเปลี่ยนพารามิเตอร์ไป
3. simple part (base part) คืออะไร ==> เราจะหยุดซื้อของเมื่อไรบ้าง ?

เฉลยการทดลอง 1

```python
def printNdown(n):
    if n > 0:
        print(n, end = ' ')
        printNdown(n-1)
def printToN(n):
    if n > 0:
        printToN(n-1)
        print(n, end = ' ')
def sumToN(n):
    if n == 1:
        return 1
    else:
       return n + sumToN(n-1)

n = 10
printNdown(n)
printToN(n)
print('\nsumToN(', n, '):',sumToN(n))
#--------------------
def printForw( L, i ):
    if i < len(L):
        print(L[i], end = ' ' )
        printForw( L, i+1 )
    else:
        print()
def printBack( L, i ):
    if i < len(L):
        printBack( L, i+1 )
        print(L[i], end = ' ' )
    else:
        print()
L = [ 2, 3, 5, 7, 11 ]
print(L)
rprintForw(L,0)
rprintBack(L,0)
print()
#------------Append--------
def rApp(l, n):
    if n == 1:
        l.append(1)
    else:
        rAdd(l, n-1)
        l.append(n)
l = []
rApp(l, 5)
print(l)
```

```python
#--------------create linked list -------------
class node():
    def __init__(self, d, nxt = None):
        self.data = d
        if nxt is None:
            self.next = None
        else:
            self.next = nxt

def printList(h):
    if h is not None:
        print(h.data, end = ' ')
        printList(h.next)

def createL(h, n):
    if n:
        p = node(n, h)
        p = createL(p, n-1)
        return p
    else:
        return h

h = None
h = createL(h, 5)
printList(h)

#---create linked list from python list ----
def createLfromlist(h, i):
    global fromList
    if i >= 0:
        p = node(fromList[i], h)
        p = createLfromlist(p, i-1)
        return p
    else:
        return h


fromList = [2,5,4,8,6,7,3,1]
i = len(fromList)-1
print(fromList, 'len =', i+1)
h = None
h = createLfromlist(h, i)
printList(h)
```

<u>ตัวอย่าง</u> <u>การทดลองที่ 2</u> แนปแสค knapsack problem

```
def printSack(sack, maxi):
    global good
    global name
    for i in range (maxi+1):
        print(good[sack[i]], end = ' ')
        # print(name[sack[i]],good[sack[i]], end = ' ')
    print()

def pick(sack, i, mLeft, ig):
    global N
    global good
    if ig < N:    # have something left to pick
        price = good[ig]    # good-price
        if mLeft < price:   # cannot afford that ig
            pick(sack, i, mLeft, ig+1) # try to pick next good
        else:               # can buy
            mLeft -= price  # pay
            sack[i] = ig    # pick that ig to the sack at i
            if mLeft == 0:  # done
                printSack(sack, i)
            else:           # still have moneyLeft
                pick(sack, i+1, mLeft, ig+1)
            pick(sack, i, mLeft+price, ig+1)  # take the item off the sack for other solutions

good = [20,10,5,5,3,2,20,10]
name = ['soap', 'potato chips', 'loly pop', 'toffy', 'pencil', 'rubber', 'milk','cookie']
N = len(good)   # numbers of good

sack = N*[-1]   # empty sack
mLeft = 20      # money left
i = 0           # sack index
ig = 0          # good index
pick(sack, i, mLeft,ig)
```