



Recursion

Iteration

Iteration การใช้ loop แก็บัญหา loop ทำซ้ำไปเรื่อยๆ

Iterate = Repeat ทำซ้ำ



Problem : Eat Up กิจกรรม n ชิ้น

iterate (repeat) eat ชิ้นที่ 1 , 2, ..., n

```
def eatUp(n):  
    for i in range(n):  
        print('eat',i+1)  
eatUp(8)
```

Recursion

Recursion การใช้ปัญหาแบบเดิมที่เล็กลงแก้ปัญหา



`eatUp (n)`



`eatUp (n-1)`



`eat 1`



`eatUp (n)`



`eat 1`



`eatUp (n-1)`



`eatUp (n)`

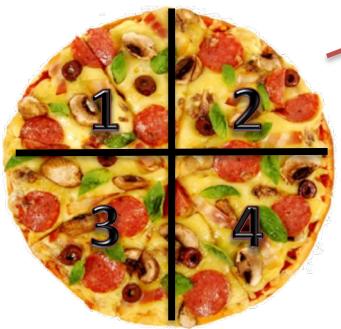


`eatUp (n/2)`



`eatUp (n/2)`

Recursive Case VS Base Case (Simple Case)



$n = 4$
EatUp (n) : EatUp ($n-1$) + eat 1

EatUp (4): EatUp (3) + eat 1

EatUp (3): EatUp (2) + eat 1

EatUp (2): EatUp (1) + eat 1

EatUp (1): EatUp (0) + eat 1

EatUp (0): EatUp (...)

EatUp (-1): EatUp (...)

EatUp (-2): EatUp (...)

EatUp (-3): EatUp (...)

```
def EatUp (n):  
  
    if n>1: #recursive case  
        EatUp (n-1)  
        print('eat1')  
    # n <= 1  
    elif n==1: #base case  
        print('eat1')  
  
    #base case : n <= 0 : do nothing
```

→ เกิด infinite loop จึงต้องมีกรณีที่ไม่ทำ recursion เรียกว่า **Base case/Simple case** เกิดจากเงื่อนไขของค่า **parameter**

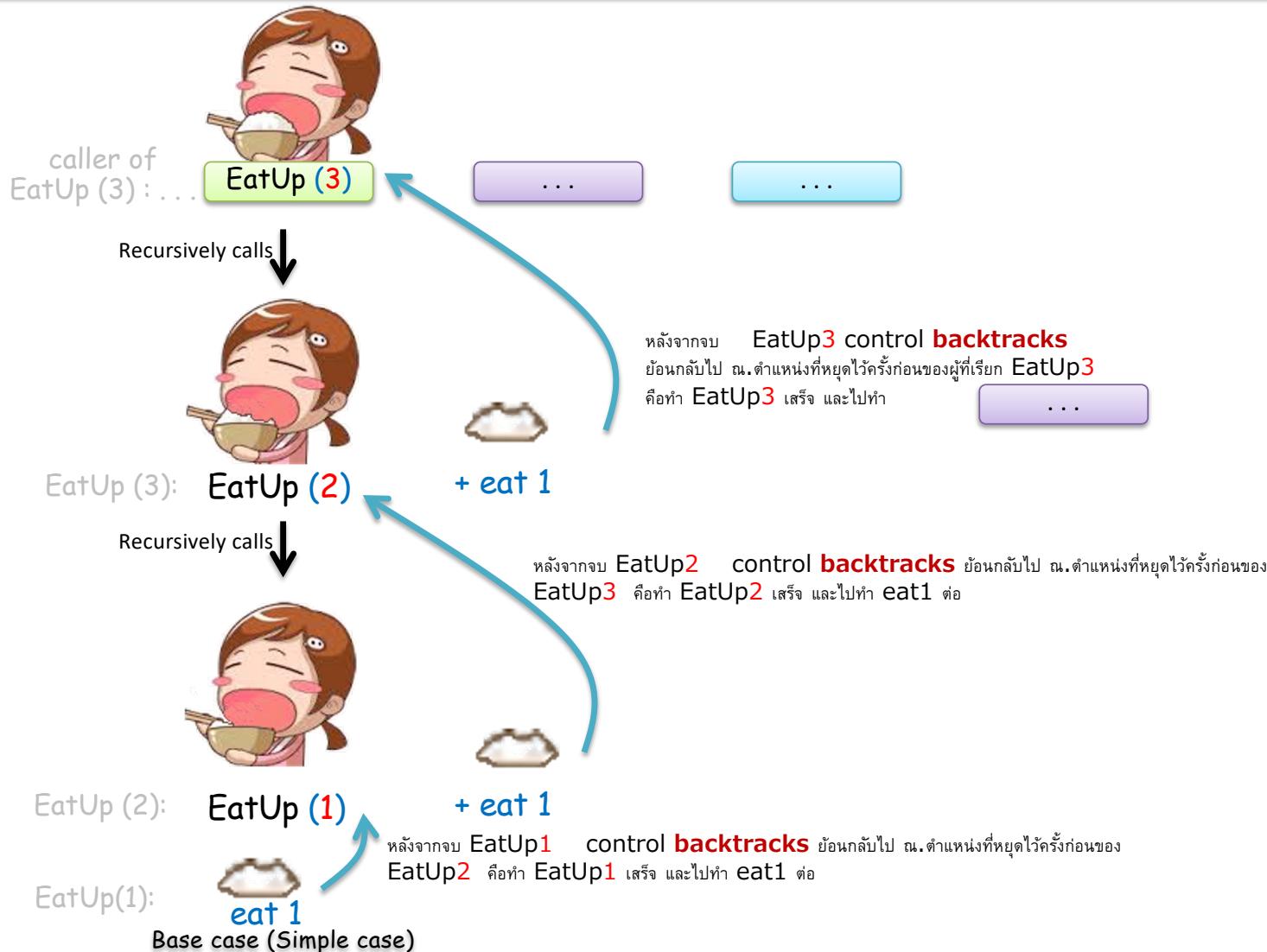
$n == 0$ ต้องทำ EatUp ?

$n == 1$ ต้องทำ EatUp ?

$n == 2$ ต้องทำ EatUp ?

กรณีนี้ base case เกิดเมื่อ $n \leq 1$
recursive case : $n > 1$

Backtracking

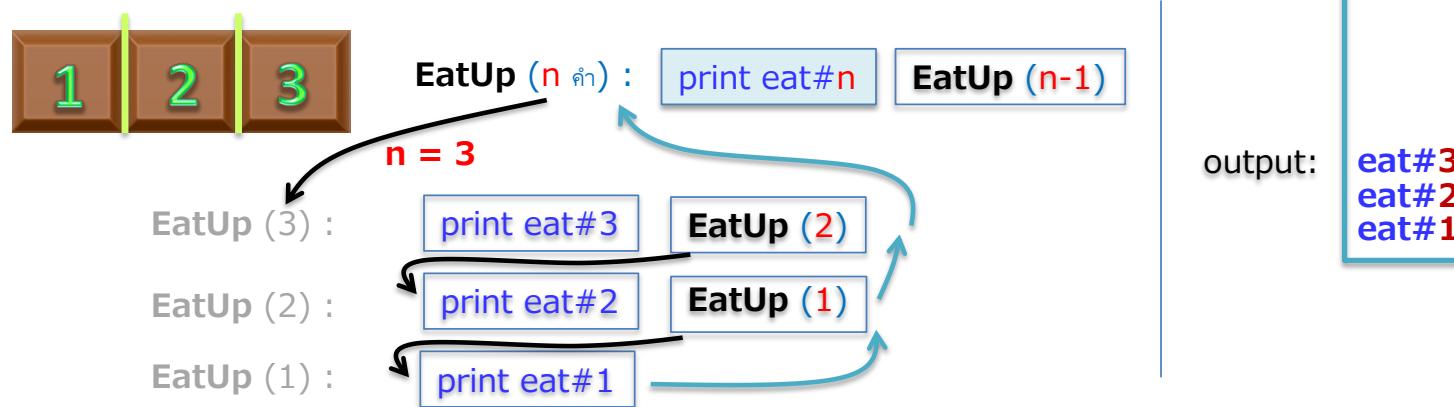
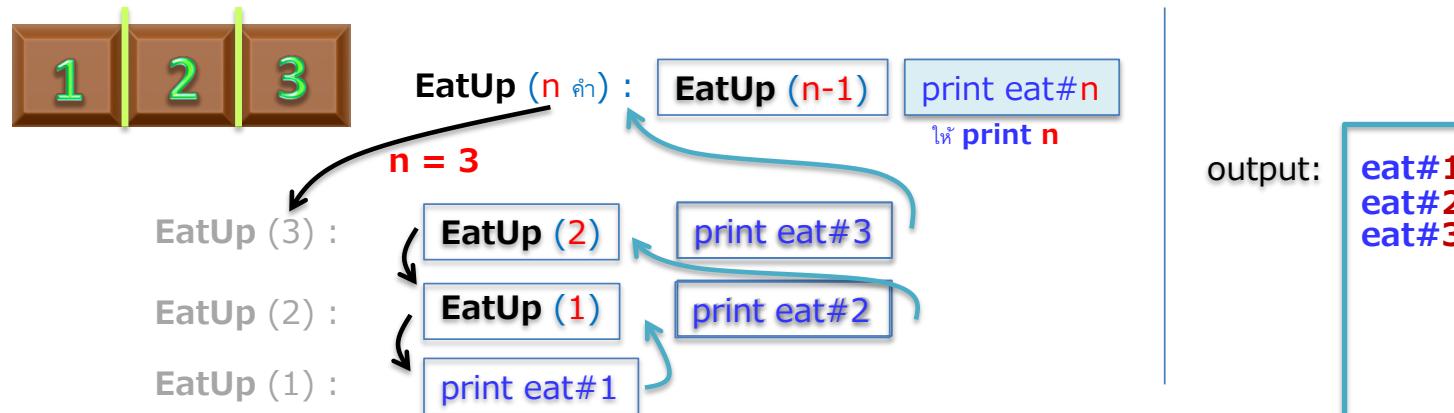


Recursive Algorithms

Recursive algorithms : แก้ปัญหาโดย ใช้ ปัญหาเดิมที่เล็กลง

1. ต้องมี parameter
2. Recursive call โดยเปลี่ยน parameter
3. ต้องมี base case / simple case :
ส่วนมากใช้เงื่อนไขของค่า parameter

ดำเนินการเรียก Recursion



Iterative Factorial

ปัญหา (problem) : หา factorial ของ n

$$\begin{aligned}0! &= 1 \\1! &= 1 \\n! &= 1 \times \dots \times (n-1) \times (n-2) \times n \quad : n \geq 1\end{aligned}$$

// Iterative Algorithm:

```
int Fac(int n){  
    result = 1  
    for (i = 1; i<= n; i++)  
        result = result * i  
    return result  
}  
int fac7 = Fac(7)
```

#Iterative Python function

```
def Fac(n):  
    result = 1  
    for i in range(2, n+1):  
        result *= i  
    return result  
  
print(Fac(7))
```

Recursive Factorial

$$0! = 1 \\ 1! = 1 \\ n! = 1 \times \dots \times (n-1) \times (n-2) \times n : n \geq 1$$

```
n! = 1           if n=0, n=1 //base case
n! = n*(n-1)!   if n>1    //recursive case
```

$$\begin{array}{ll} 0! & = 1 \\ 1! & = 1 \\ 2! & = 1 \times 2 \\ 3! & = 1 \times 2 \times 3 \\ 4! & = 1 \times 2 \times 3 \times 4 \end{array} \quad \begin{array}{ll} 0! & = 1 \\ 1! & = 1 \\ 2! & = 1! \times 2 \\ 3! & = 2! \times 3 \\ 4! & = 3! \times 4 \end{array}$$

$n! = (n-1)! \times n$

หาก
คิด
ไม่
ออก

```
def Fac(n): # n >= 0
    if n == 0 or n == 1: #base case
        return 1
    else: #recursive case n > 1
        return Fac(n-1) * n #recursive case
```

คิด recursion ให้ง่ายขึ้น ไม่ต้องคิดว่างานที่เล็กลงนั้นต้องทำย่างไร คิดว่าหากทำได้ ต้องทำอะไรต่อให้งานใหญ่ๆ จบ

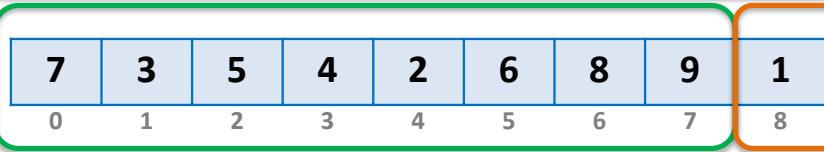
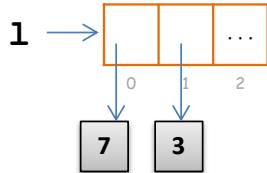
1. นัญหาเดิมที่เล็กลง -> ลอง $n-1$?
2. ให้คิดว่า $\text{Fac}(n-1)$ คือ fac ของ $n-1$ โดยไม่ต้องคิดว่ามันทำได้อย่างไร
ดังนั้น $\text{Fac}(n-1)$ คือ $1 \times \dots \times (n-1)$
3. ต้องการ $\text{Fac}(n)$: $1 \times \dots \times (n-1) \times n$

$\text{Fac}(n-1) \times n$

Recursive : แก้ปัญหาโดยใช้ปัญหาเดิมที่เล็กลง

1. ต้องมี parameter
2. Recursive call โดยเปลี่ยน parameter
3. ต้องมี Base case / Simple case
(เงื่อนไขขึ้นกับค่า parameter)

Sum of Array (Python List) Elements



```
sum(n) : #sum elements n ตัวแรก
    if n is 0 : #None list
        return 0
    elif n is 1 : #base case
        return element ตัวแรก
    else:
        return sum(n - 1)+ element ตัวสุดท้าย
#sum elements n-1 ตัวแรก + ...
```

```
def sum(n, l): #sum elements n ตัวแรกของ list l
    if n is 0: #None list base case
        return 0
    elif n is 1: #base case
        return l[0]
    else:
        return sum(n-1, l)+ l[n-1]

l = [1,2,3,4,5,6,7,8,9]
print(sum(len(l)), l)      → 45
```

ปัญหาคือ ?

sum n ตัวแรก → sum(n) # สมมุติว่า n < len(list)

parameter คือ ?

sum(n)

ปัญหาเล็กลง, recursion โดยเปลี่ยน parameter ?

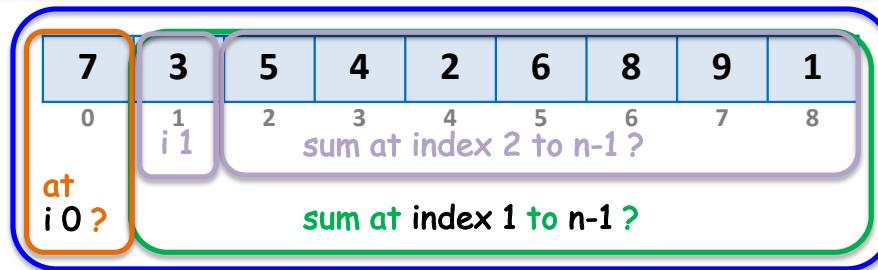
sum(n - 1)

None list ?

Base case ? n = 4, 3, . . . ?

Base case : n = 1 n = 0

Sum of Array (Python List) Elements 2



สมมุติว่า $n < \text{len}(l)$

```
def sum2(l, fromI, toI):    #sum elements ของ list l จาก index fromI ถึง to index toI

    if fromI > toI:      #None list
        return 0

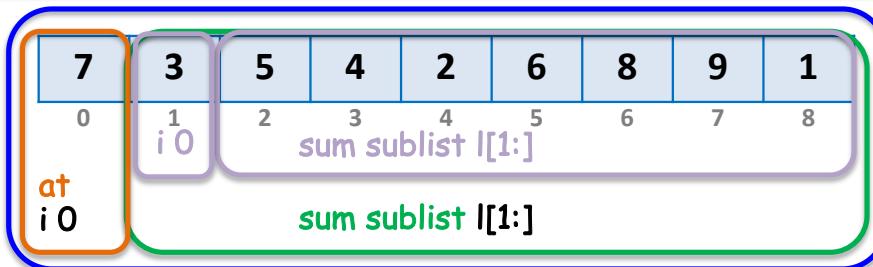
    elif fromI == toI:   #base case
        return l[fromI]

    else:
        return l[fromI] + sum2(l, fromI+1, toI)    #recursive case
```

```
#for easier debugging
x = l[fromI]
y = sum2(l, fromI+1, toI)
return x + y
```

```
l = [1,2,3,4,5,6,7,8,9]
print( sum2(l, 0, len(l)-1) ) ➔ 45
```

Sum of Array (Python List) Elements : sublist



```
>>> l = [7,3,5,4,2,6,8,9,1]
>>> l2 = l[1:]
>>> print(l2)
[3,5,4,2,6,8,9,1]
```

default step= 1

sublist format `l[start index : before last index : step]`

default = list size
-> to last element

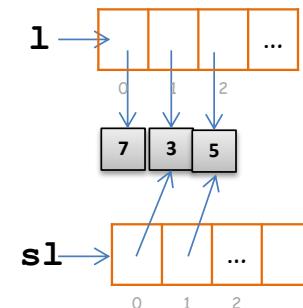
```
def sum3(l): #sum list l using sublist
    n = len(l)

    if n is 0:
        return 0
    elif n is 1:
        return l[0]
    else:
        return l[0] + sum3(l[1:]) #recursive case

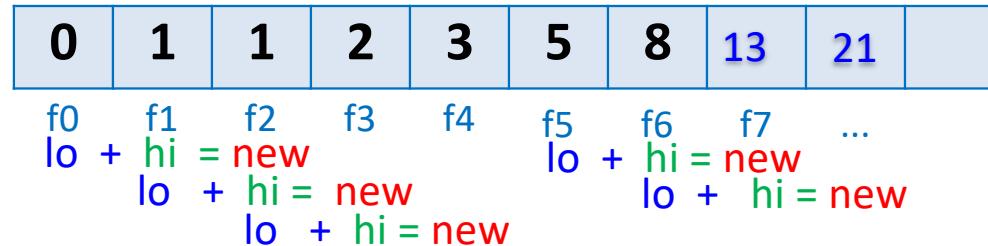
li = [1,2,3,4,5,6,7,8,9]
print('sum list elements 3 ', sum3(li)) ➔ 45
```

Do it yourself :
`def sum3(?) :`

sublist:
Expensive แพงมาก
ไม่ควรใช้



Fibonaci Sequence Iterative



```
def fib(n):      #iterative, n>=0

    if n == 0 or n == 1:

        return n

    else:

        lo, hi = 0, 1

        for i in range(2, n+1):

            new = hi + lo

            lo = hi

            hi = new

    return new
```

Fibonaci Sequence Recursive

0	1	1	2	3	5	8		
f0	f1	f2	f3	f4	f5	f6	f7	...

$$f_7 = ? = 5+8=13$$

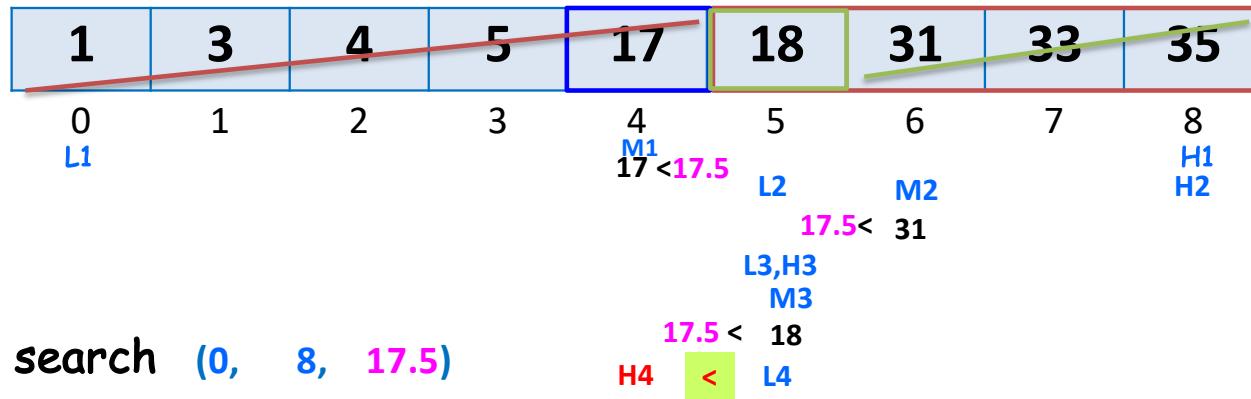
$$f_7 = f_6 + f_5$$

```
fib(n) = n           if n=0, n=1    //base case  
fib(n) = fib(n-1) + fib(n-2)  if n>1   //recursive case
```

```
def fibR(n):      # recursive, n>=0  
    if n <= 1:  
        return n  
    else:  
        return fibR(n-1) + fibR(n-2)
```

Iterative Binary Search

search for
 $x = 17.5$



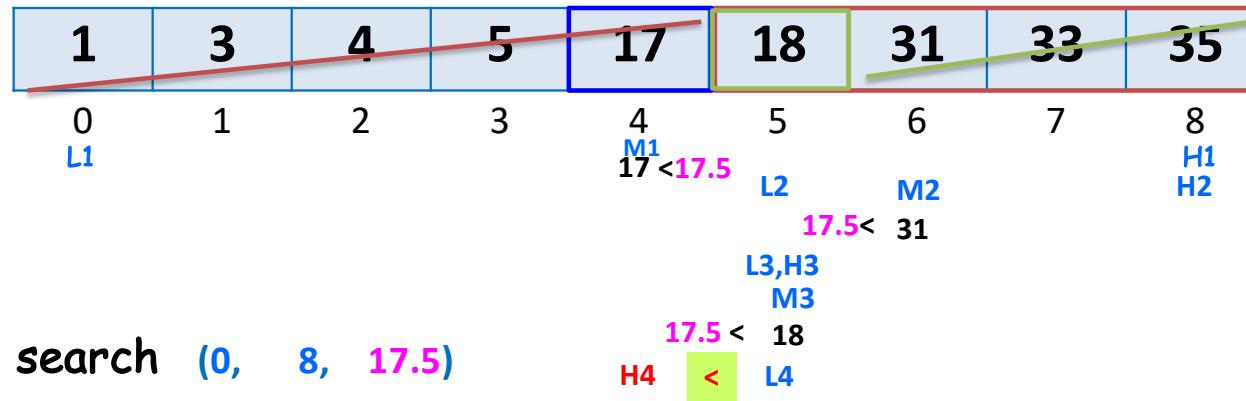
ret_value $\text{search } (\text{low}, \text{high}, x)$

```
while ( low <= high ) {  
    mid = (low+high) div 2;  
    if (x==a[mid])  
        return (mid); //found at mid  
    else if (a[mid] < x)  
        low = mid+1; //search : mid+1 - high  
    else high = mid-1; //search : low - mid-1  
}  
return(-1); //not found
```

ทำการ search เหลืออนเดิม แต่
เปลี่ยน ขอบเขต
-> เปลี่ยน parameter
-> recursive ได้

Recursive Binary Search

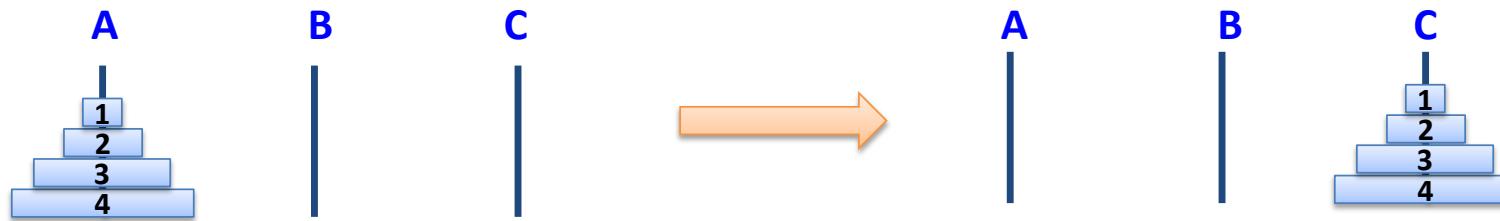
search for
 $x = 17.5$



ret_value **search (low, high, x)**

```
if (high < low)
    return(-1); //simple case : not found
mid = (low+high) div 2;
if (x==a[mid])
    return(mid); //simple case
else if (a[mid] < x)
    return search (mid+1, high, x) //recursive case
else return search (low, mid-1, x) //recursive case
```

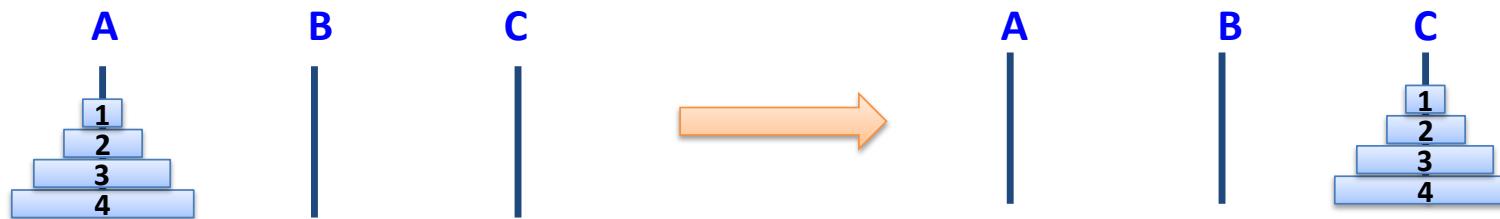
Tower of Hanoi Problem



ต้องการหยົບແຜ່ນດີກສົກ້າໜົມຈາກເສາ A ໄປເສາ C ໂດຍໃຊ້ເສາ B ຂ່າຍ

- ຫຍົບທີ່ລະແຜ່ນ
- ຕ້ອງຫຍົບແຜ່ນທີ່ອຸ່ງດ້ານບນກ່ອນ
- ແຜ່ນໄຫ້ມູ່ຫ້ມວາງກັບແຜ່ນເລັກ

Recursive Tower of Hanoi



ปัญหาคือ ?

move 4 disks จาก A ไป C

parameter คือ ?

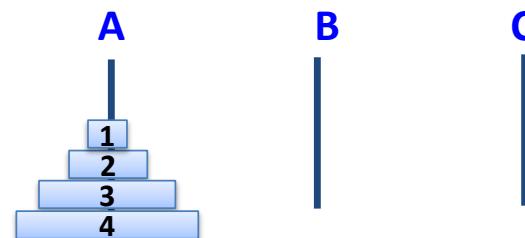
move (n , A , C)

ปัญหาเล็กลง, recursion โดยเปลี่ยน parameter ?

move (n - 1 , , ,)

move (4, A, C)

move (3, , ,)



move (3 , A, C) → NO

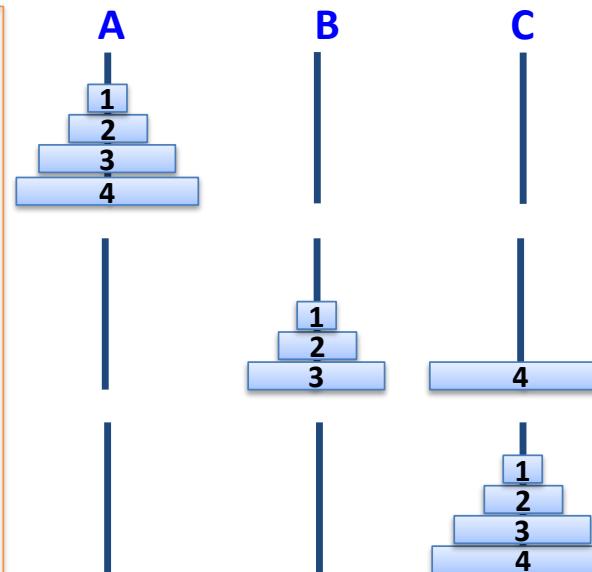
move (3 , A, B)

Recursive Tower of Hanoi



```
move (4, A, C) :  
if n != 1:  
    move (3, A, B)  
  
move disk #4 from A to C  
print move disk #4 from A to C  
  
move (3, B, C)  
else:  
    print move disk #n from A to C
```

ขั้นตอนรู้ว่าทำ move(3,A,B) อย่างไร
รู้ว่าได้ผลลัพธ์หรือไม่ : เลื่อน 3 disks A->B
แล้วคิดว่าต้องอ่านฟังต้องทำอะไรต่อ

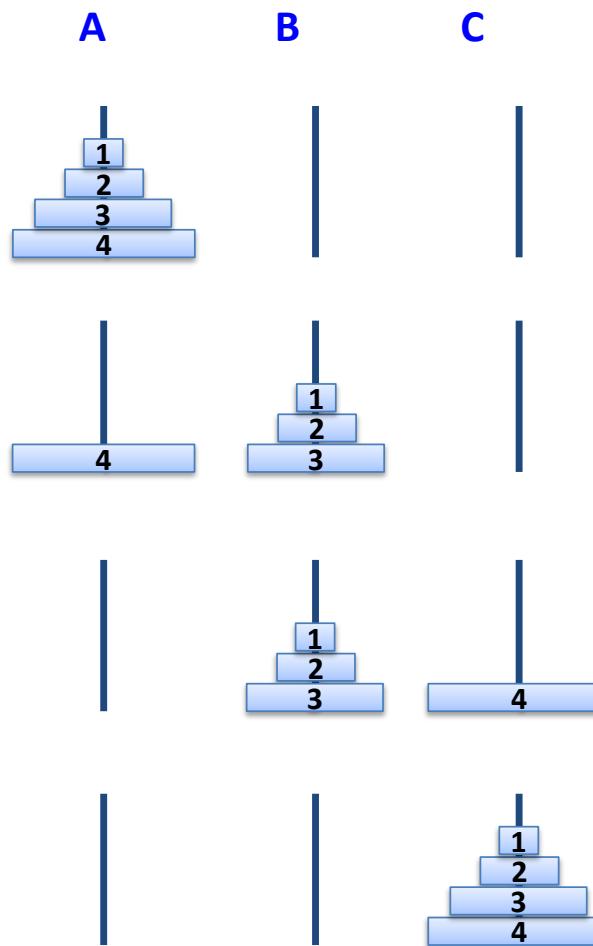


Base case ? n = 4, 3, . . . ? Base case : n = 1

Tower of Hanoi

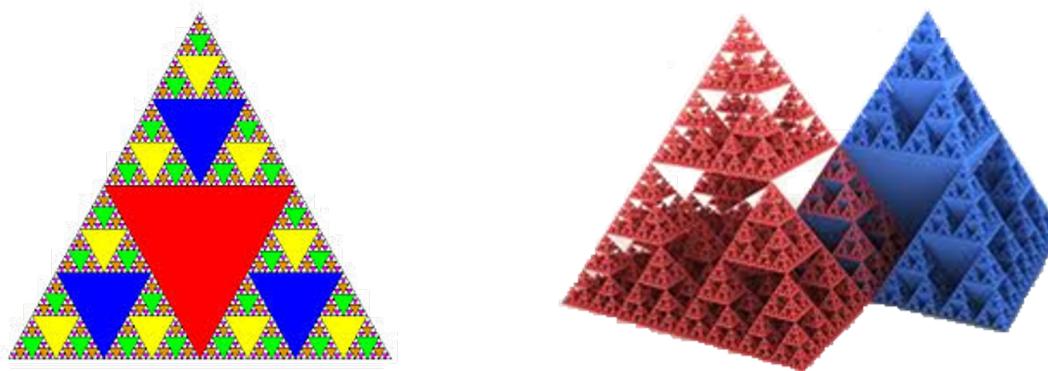
```
def move(n, A, C, B):
    if n == 1:
        print(n, 'from', A, 'to', C)
    else:
        move(n-1, A, B, C)
        print(n, 'from', A, 'to', C)
        move(n-1, B, C, A)
```

การเขียน : trace code ว่าได้ output อะไร :

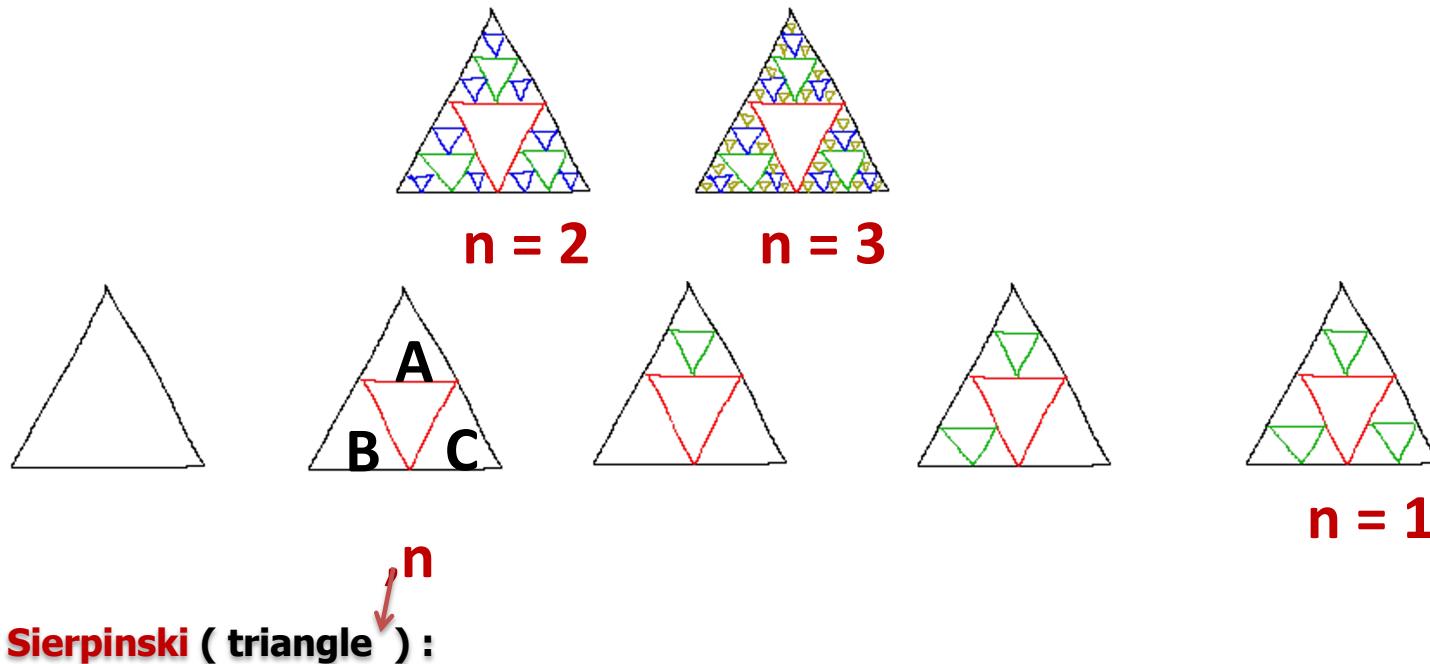


Sierpinski Triangle

The Sierpinski Triangle, also called Sierpinski Gasket and Sierpinski Sieve, is named after Waclaw Sierpinski, a Polish mathematician (1882-1969).



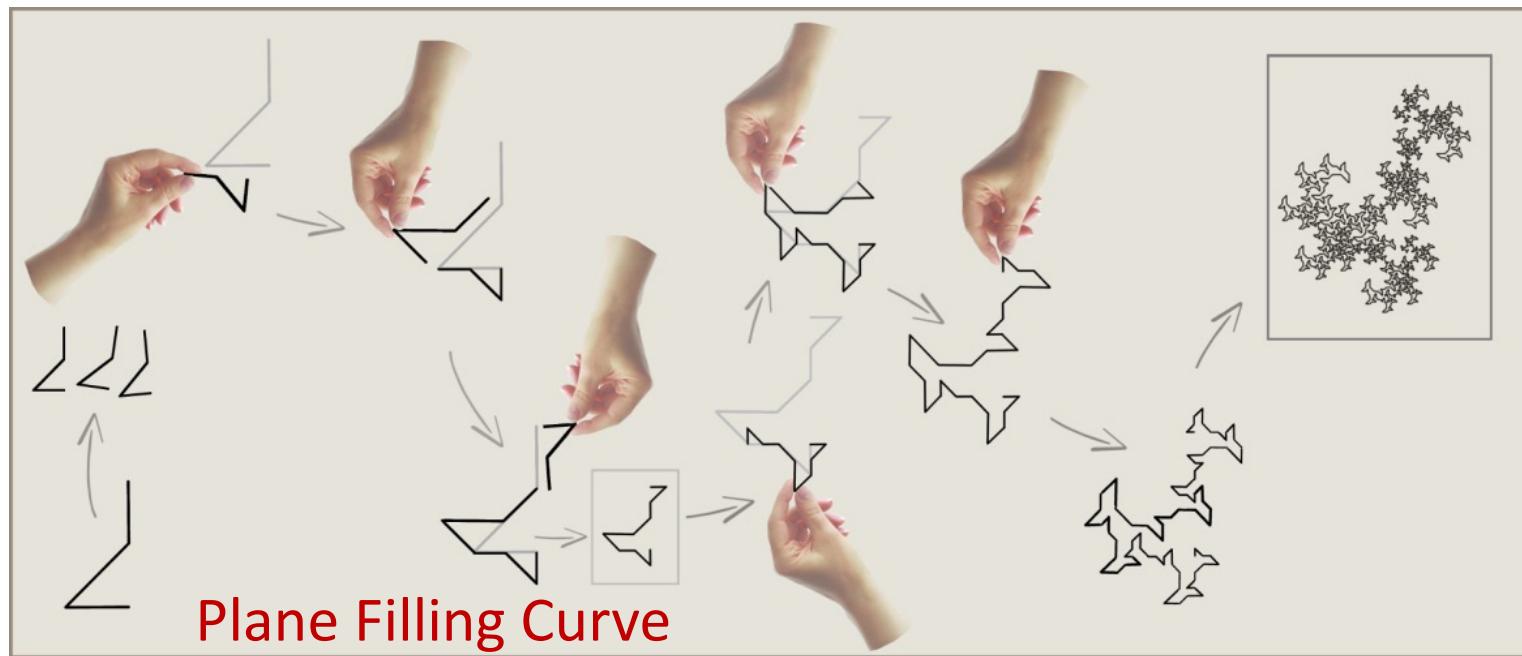
Sierpinski Triangle



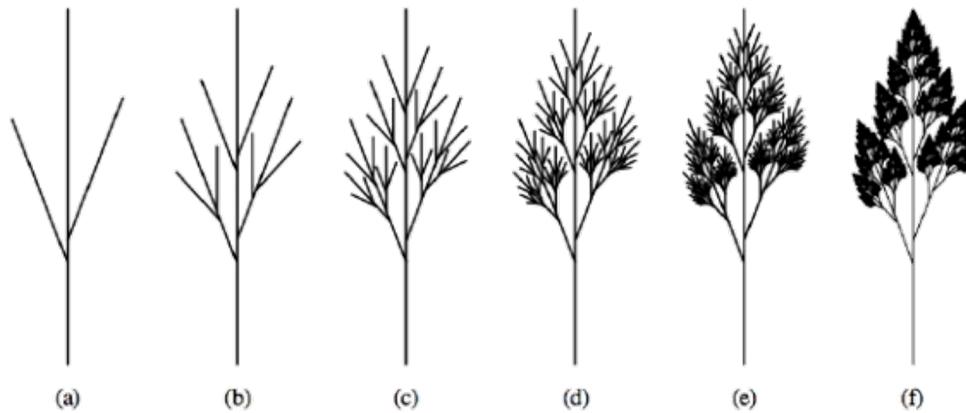
Sierpinski (triangle) :

1. แบ่งสามเหลี่ยมใหญ่ เป็น 4 สามเหลี่ยมเล็ก โดยลากเส้นต่อจุดกึ่งกลางของแต่ละด้าน
2. recursion **Sierpinski (triangle A)**
3. recursion **Sierpinski (triangle B)**
4. recursion **Sierpinski (triangle C)**

Recursion



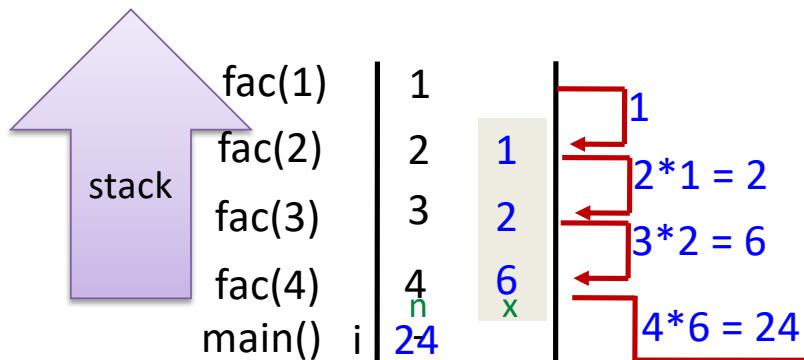
Natural Recursion



Stack of Recursion

Backtracking :

return กลับไปที่การ call ครั้งก่อน



```
def fac (n): # n>=0
    if n == 0 or n == 1:
        return 1
    else:
        x = fac (n-1)
        return n * x
```


i = fac (4);

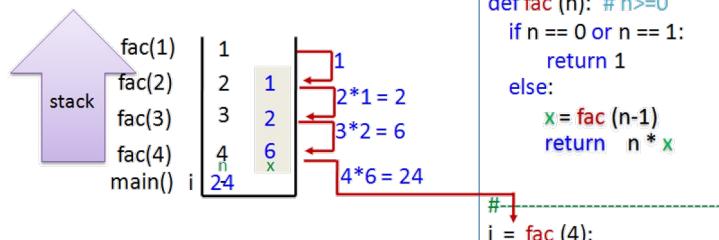
เพื่อให้เห็นกระบวนการซัดเจน เราจะเปลี่ยน
code โดยใช้ **x** เก็บค่าที่ return จาก fac()

Iteration VS Recursion

```
def fac(n):
    result = 1
    for i in range(n, 0, -1):
        result *= i
    return result
```

```
def facR (n): # n>=0
    if n == 0 or n == 1:
        return 1
    else:
        return n * facR(n-1)
```

RunTime ?



Space ?

```
def fac (n): # n>=0
    if n == 0 or n == 1:
        return 1
    else:
        x = fac (n-1)
        return n * x
```


i = fac (4);

ทำไมต้อง recursion?

Coding Time น้อยกว่า

Code อ่านง่าย เข้าใจง่าย กว่า

Debug ง่ายกว่า

Iteration ส่วนมากมีประสิทธิภาพ (efficient) กว่า

Recursion แม้เพราๆ function call ต้อง

- Passing parameters.
- Pushing /Poping stack.

Tail Recursion

```
def facR (n): # n>=0
    if n == 0 or n == 1:
        return 1
    else:
        return n * facR(n-1)
```

```
def fibR(n): # recursive, n>=0
    if n <= 1:
        return n
    else:
        return fibR(n-1) + fibR(n-2)
```

```
def searchR ( L, x, low, high ):
    if low > high:
        return None

    mid = (low + high) // 2
    if x == L[mid]:
        return mid
    elif L[mid] < x:
        return searchR ( L, x, mid+1, high )
    else:
        return searchR ( L, x, low, mid-1 )
```

Tail Recursion

execute recursion เป็นสิ่งสุดท้ายในฟังก์ชันนั้น

Tail recursion
ง่ายที่จะเขียนแบบ iteration

ดังนั้น ควรเขียนแบบ ?

When would I use recursion?

Recursion ใช้ได้ดีเมื่อเราต้องทำ iterative branching ไปเรื่อยๆ ซึ่งเขียน Code ง่ายกว่า iterative มาก -> debug ง่าย

n-queen problem

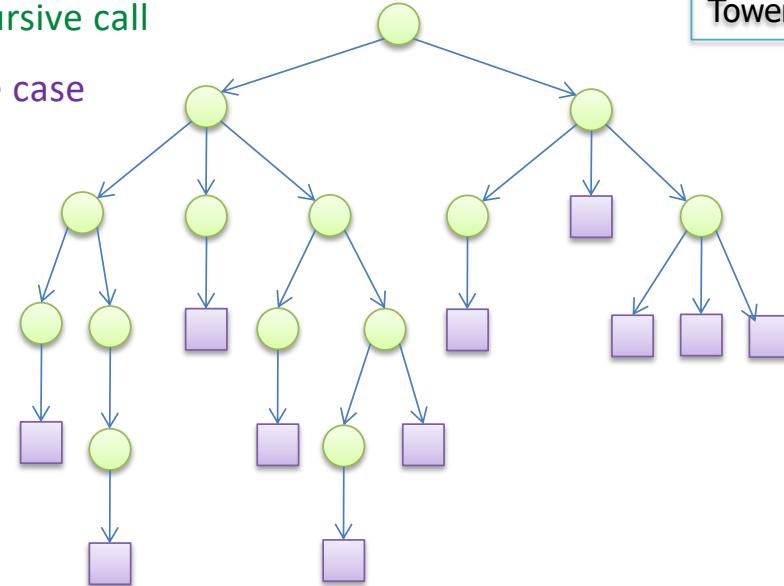
- มี iterative branching จึงควรเขียนแบบ recursion
- ซึ่งจะได้ข้อมูลของ call stack ของ recursion เพื่อจำ condition เอาไว้ เมื่อ backtrack กลับมาจะได้สภาพเดิม

Sierpinski triangle (เซอร์ปินสกี้)
call recursion 3 ครั้งด้วย parameters ที่ต่างกัน

Tower of Hanoi call recursion 2 ครั้ง

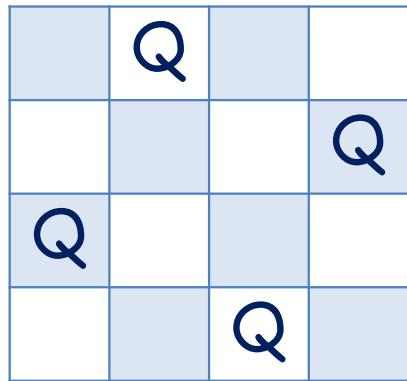
(○) Recursive call

(□) Base case

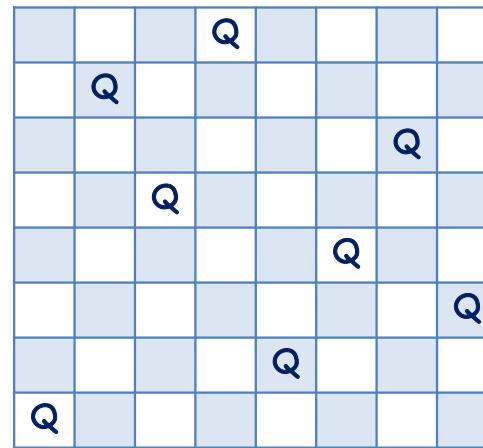


The N Queen Problem

ใส่ N queens บน N x N board โดยไม่ให้กินกันเลย



Solution for 4 Queen problem.



Solution for 8 Queen problem.

8 x 8 board has

- 92 distinct solutions.
- 12 unique solutions (reduce redundancy of symmetry (rotations & reflections)).

Data Structure for Queens – 2D array , Python : list of list

จะเก็บ queens อย่างไร ?

	Q		
			Q
Q			
		Q	

. 1 . .
. . . 1
1 . . .
. . 1 .

list of list (Python)

board = [4*[0],4*[0],4*[0],4*[0]]

[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

[[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]]

```
>>> l = [0]*4
>>> print(l)
[0, 0, 0, 0]
```

Data Structure for Solution Queens – 1D array, Python list

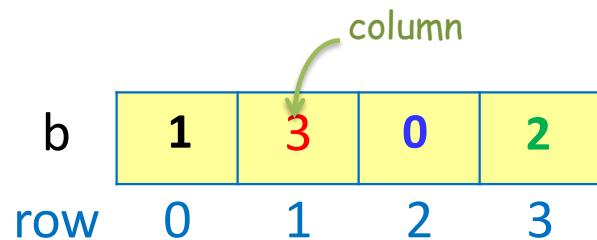
2-D array

```
b = [4*[0],4*[0],4*[0],4*[0]]
```

.	1	.	.	.	1
.
1
.	.	.	1	.	.

1-D array

```
b = N*[-1]  
b[1] = 3
```



	c0	c1	c2	c3
r0		Q		
r1				Q
r2	Q			
r3			Q	

Data Structure & isSafe() Algorithm

$0 \rightarrow \text{false}$. นอกนั้นทั้งหมด $\rightarrow \text{true}$

ตอนแรก Initialize ทั้งหมด 1 free ทั้งหมด

1. col free? เช็คกี่ col ?

$\text{colFree} = N*[1]$

colFree

$c=3$ ไม่ free

1	1	1	0
0	1	2	3

2. up free $\text{upFree}(r+c)$ $r=1, c=3 \rightarrow \text{upFree}(4)$ ไม่ free

$\text{upFree} = (2*N - 1)*[1]$

upFree

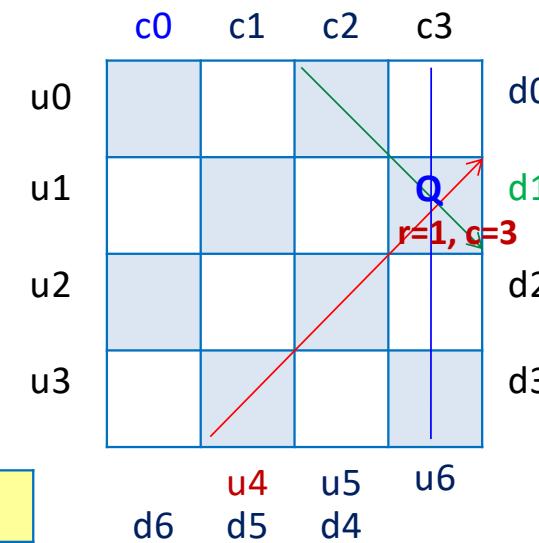
				0		
0	1	2	3	4	5	6

3. down free $\text{downFree}(r-c+(n-1))$
 $\text{downFree}(1-3+4-1) = \text{downFree}(1)$ ไม่ free

$\text{downFree} = (2*N - 1)*[1]$

downFree

	0					
0	1	2	3	4	5	6

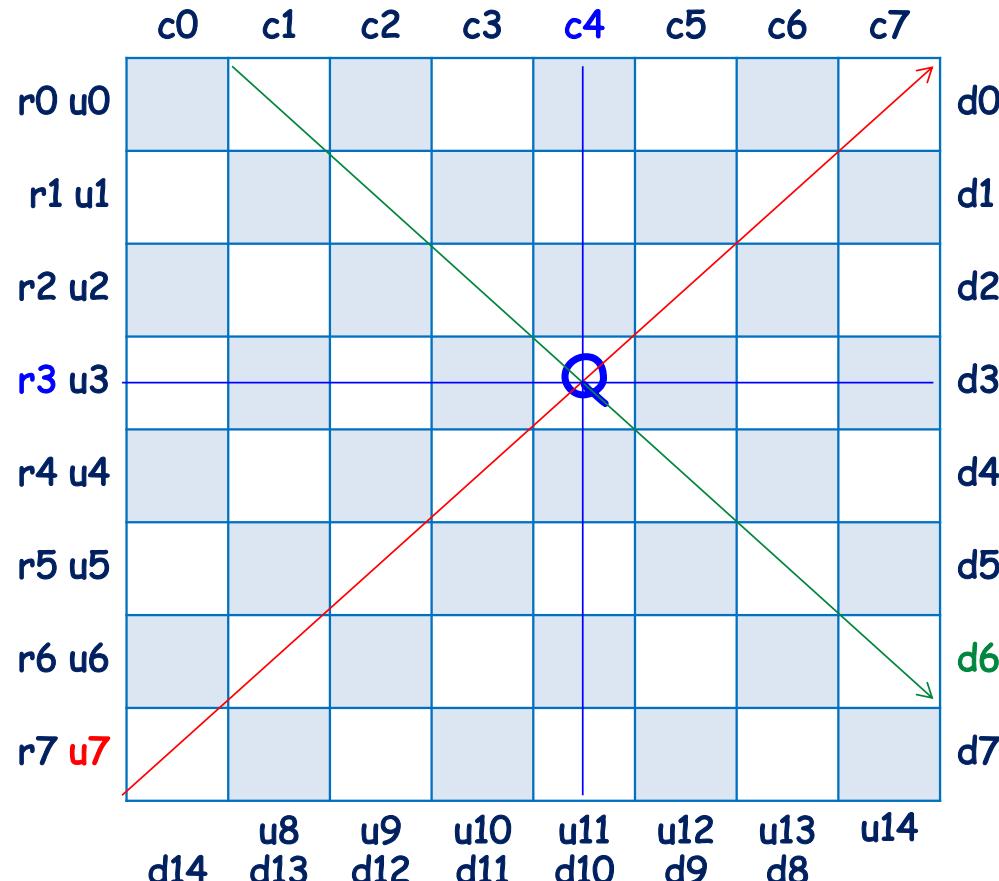


isSafe() () : Algorithm for 8 Queens

Queen กิน 8 ทิศ

จะได้ $Q(r, c)$ ต้องเช็ค

1. col(c) free?
4
 2. up(7) free?
up(r+c)
up(3+4)
 3. down(6) free?
down(r-c+(n-1))
3-4+7
 4. เนื่องจากเราใส่ทีละແຕງ
ใส่แล้วใส่ແກວถัดไป
จึงไม่ต้องเช็คແຕງ



Initializations : Python

```
N = 8                                # N x N Board
numSol = 0                            # number of solutions

def printBoard(b):                      # in next page
    pass
def putQueen(r, b, colFree, upFree, downFree): # in next page
    pass

b = N*[-1]                            # indices = rows, b[index] = column, first init to -1
colFree = N*[1]                          # all N col are free at first
upFree = (2*N - 1)*[1]                  # all up diagonals are free at first
downFree = (2*N - 1)*[1]                 # all down diagonals are free at first

putQueen(0, b, colFree, upFree, downFree) # first add at 1st (ie. row 0)

print('number of solutions = ', numSol)
```

Recursive PutQueen : Python

```
def printBoard(b):    # ใส่ควีนทีละແຄວ เริ่มจากແຄວบนสุด ใส่ແຄວละตัว  จะใส่ได้มีเมื่อถูกตัวที่ใส่ไปแล้วกิน  
    print(b)
```

putQueen (0,...)

```
def putQueen(r, b, colFree, upFree, downFree):
```

```
    global N
```

```
    global numSol
```

```
    for c in range(N): # ใส่ไปทีละ column ทุก col.
```

```
        if colFree[c] and upFree[r+c] and downFree[r-c+N-1]: # ใช่ได?
```

```
            b[r] = c # ใส่ ที่ r, c
```

Q?							
ok?	ok?	ok?					

```
colFree[c] = upFree[r+c] = downFree[r-c+N-1] = 0 # เปลี่ยน data struct ใหม่ให้ใส่แนวนี้
```

```
if r == N-1: # ถ้าใส่ควีนครบแล้ว
```

```
    printBoard(b) #print(b)
```

```
    numSol += 1
```

```
else:
```

```
    putQueen(r+1, b, colFree, upFree, downFree) # ใส่ควีนແຄວถัดไป
```

```
colFree[c] = upFree[r+c] = downFree[r-c+N-1] = 1 # เอา Queen ออกเพื่อให้ได้ solution อื่น
```

```
# หรือ เพราะ queen ตัวนี้แม่ใส่ได้แต่ไม่ทำให้เกิด solution
```

Recursion VS Iteration

1. Iteration ส่วนมากมีประสิทธิภาพ (efficient) กว่า recursion ทั้งด้าน space และ runtime เพราะ recursion มีงานในการทำ function call : passing parameters, pushing-poping stack
2. code ที่มี iterative branching แบบ recursion ใช้เวลาเขียน code น้อยกว่า code ที่มี iterative branching และ debug ง่ายกว่า