Recursion

Recursion คืออะไร

- Recursion คือวิธีการแก้ปัญหาแบบหนึ่งที่เกี่ยวกับการแตกปัญหาเป็นปัญหาที่เล็กลงๆ จนกระทั่งปัญหาเล็กพอที่เราจะแก้มันได้โดยง่าย

- โดยทั่วไปแล้ว recursion เกี่ยวกับฟังก์ชันที่เรียกตัวเอง
- แม้ว่ามันจะมองออกยาก แต่ recursion ทำให้เราสามารถเขียนคำตอบในรูปที่สวยงามของปัญหาได้ แม้ว่าจะเขียนโปรแกรมยากก็ตาม

การคำนวณผลรวมของจำนวนในลิสต์

- เราจะเริ่มต้นด้วยปัญหาที่ง่ายก่อนซึ่งเรารู้วิธีการแก้โดยที่ไม่ใช้ recursion
- สมมติว่าเราต้องการคำนวณหาผลรวมของจำนวนใน list เช่น [1,3,5,7,9] หากเขียนฟังก์ชันแบบวนซ้ำจะเขียนได้ดังนี้

```
def list_sum(num_list):
    the_sum = 0
    for i in num_list:
        the_sum = the_sum + i
    return the_sum
```

- ฟังก์ชันนี้ใช้ตัวแปร the_sum เป็นตัวแปรที่รวมผลบวกของจำนวนทุกจำนวนใน list โดยเริ่มต้นให้มีค่าเป็น 0 หลังจากนั้นจะเพิ่มค่าตามจำนวนใน list ทีละตัว

```
print(list_sum([1,3,5,7,9]))
```

สมมติว่าถ้าเราไม่มี for หรือ while ให้ใช้ เราจะคำนวณผลรวมของจำนวนใน list ได้อย่างไร

- เราอาจจะเริ่มด้วยสิ่งที่เรามี ฟังก์ชัน นั่นเอง โดยสร้างฟังก์ชัน addition ที่เป็นฟังก์ชันที่รับข้อมูลเข้าสองตัว
- ดังนั้นเราอาจจะต้องนิยามปัญหาใหม่จากการบวกกันของ list เป็นการบวกเป็นคู่ๆ โดยใช้ฟังก์ชัน
- เราจะเขียน list ใหม่ด้วยวงเล็บแทนการเรียกใช้ฟังก์ชัน จะได้
- ((((1+3)+5)+7)+9)
- หรืออาจจะเขียนเป็น (1+(3+(5+(7+9))))

- สังเกตว่าวงเล็บในสุด (7+9) เป็นปัญหาที่เราสามารถแก้ได้โดยไม่ต้องใช้ loop หรือคำสั่งพิเศษอื่นๆ
- ทั้งนี้เราสามารถใช้ลำดับที่กำหนดให้ต่อไปนี้เพื่อคำนวณการบวก
- Total = (1+(3+(5+(7+9))))
- Total = (1+(3+(5+16)))
- Total = (1+(3+21))
- Total = (1+24)
- Total = 25

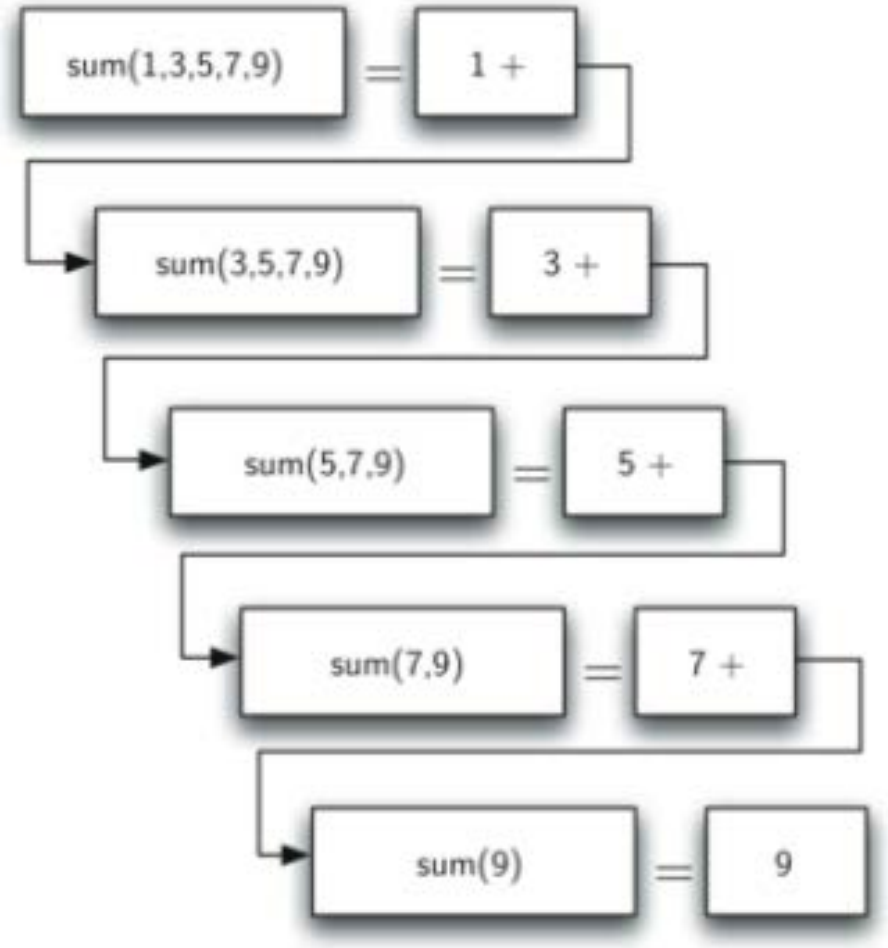
- เราสามารถเอาแนวคิดนี้มาเขียนเป็นภาษา Python ได้หรือไม่
- เริ่มต้นปัญหาการหาผลรวมในรูปของ list
- เราอาจจะบอกว่า ผลรวมของ list (num_list) คือผลรวมของสมาชิกตัวแรกกับตัวที่เหลือ num_list[0] กับ num_list[1:]
- หากประกาศในรูปของฟังก์ชันจะได้ว่า

list_sum(num_list) = first(num_list)+list_sum(rest(num_list))

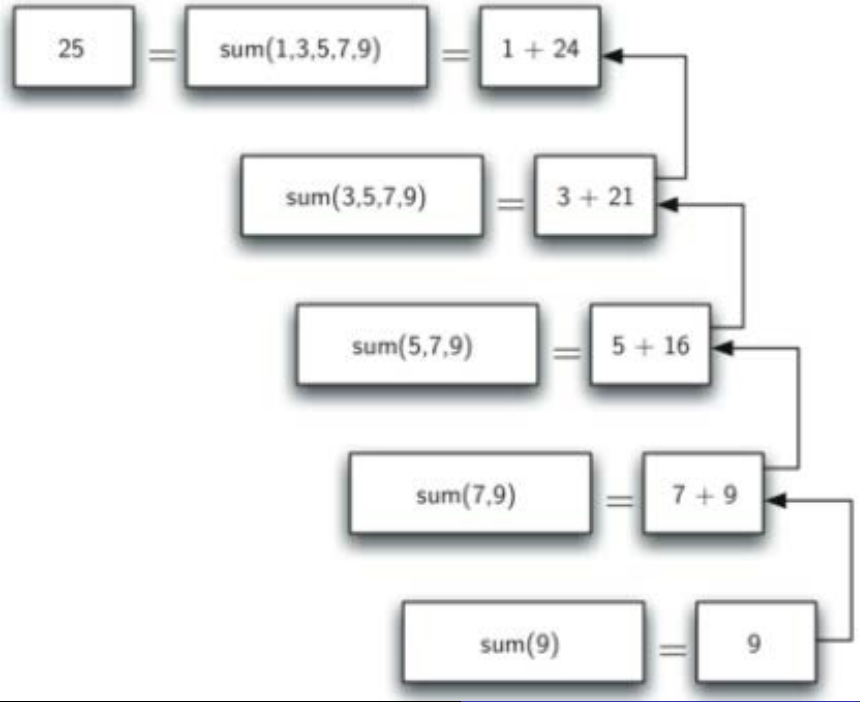
โดยที่ first(num_list)คืนค่าสมาชิกตัวแรกและrest(num_list) คืน list ของทุกตัวยกเว้นตัวแรก

```
def list_sum(num_list):
    if len(num_list) == 1:
        return num_list[0]
    else:
        return num_list[0] + list_sum(num_list[1:])
print(list_sum([1,3,5,7,9]))
```

- จากตรงนี้เราพอจะได้ไอเดีย ไอเดียแรกในบรรทัดที่ 2 เราได้ตรวจสอบว่าถ้า list มี 1 ตัวหรือไม่
- การตรวจสอบนี้สำคัญมาก เพราะว่าจะเป็นจุดที่ทำให้เราออกจากฟังก์ชันได้
- ทั้งนี้ผลรวมของ list ที่ยาว 1 ช่องนั้นหาได้ง่าย คือตัวมันเอง
- ไอเดียที่สองในบรรทัดที่ 5 ฟังก์ชันของเรามี**การเรียกตัวเอง**
- **ฟังก์ชันที่เรียกตัวเองเราเรียกว่า recursive function**


sum(1,3,5,7,9)
=
1 +
sum(3,5,7,9)
=
3 +
sum(5,7,9)
=
5 +
sum(7,9)
=
7 +
sum(9)
=
9

- เมื่อเราทำตามไปจนถึงจุดหนึ่งที่ปัญหาแก้ง่ายแล้ว เราจะนำส่วนเล็กๆ ที่แก้แล้วมารวมกันกลับเป็นคำตอบของปัญหาตั้งต้น
- ในรูปเป็นการแสดงว่า list_sum ทำงานอย่างไร เป็นการเรียกทำงานจากหลังไปหน้า เมื่อ list_sum ไปถึงจุดบนสุด เราจะได้คำตอบ



กฎสามข้อของ Recursion

- ทุก recursive algorithm จะมีกฎ 3 ข้อ

1. Recursive algorithm ต้องมี base case
2. Recursive algorithm ต้องเปลี่ยน state และเปลี่ยนไปเป็น base case
3. Recursive algorithm ต้องเรียกตัวเอง

- มาลองพิจารณาแต่ละข้อ เมื่อเปรียบเทียบกับ list_sum
- ข้อแรก base case คือเงื่อนไขที่ทำให้ algorithm หยุดการเรียกตัวเอง base case โดยทั่วไปแล้วคือปัญหาที่มีขนาดเล็กพอที่เราจะแก้ได้โดยตรง ใน list_num นั้น base case คือ list ที่ความยาว 1
- ข้อสอง เราจะต้องเรียงการเปลี่ยนแปลงของ state ที่เปลี่ยนไปเป็น base case ทั้งนี้การเปลี่ยน state หมายถึงข้อมูลที่ algorithm ใช้ถูกเปลี่ยนแปลง

- โดยทั่วไปแล้วข้อมูลที่แสดงในปัญหาของเรานั้น จะถูกทำให้เล็กลงได้ในบางวิธี
- ใน list_sum โครงสร้างข้อมูลของเราคือ list ซึ่งเราจะสนใจการเปลี่ยน state ของ list
- เนื่องจาก base case คือ list ที่ยาว 1 การเปลี่ยน state เพื่อให้ไปถึง base case ได้คือ การทำให้ list สั้นลง นั่นคือสิ่งที่เกิดขึ้นในบรรทัดที่ 5 ที่เราเรียก list_sum ด้วย list ที่สั้นลง
- กฎข้อสุดท้ายคือ ต้องเรียกตัวเอง นี่เป็นนิยามของ recursion

- หากคำนวณผลรวมของ list [2,4,6,8,10] มีการเรียกตัวเองกี่ครั้ง

- สมมติว่าต้องการเขียน recursive function เพื่อคำนวณค่า factorial f(n) คือค่า n*n-1*n-2*… เมื่อค่า factorial ของ 0 มีค่าเป็น 1 เราจะเลือก base case เป็นเท่าไรดี

- จงเขียนฟังก์ชัน คำนวณค่า n! (factorial) เมื่อให้ค่า n

การเปลี่ยนเลขจำนวนเต็มเป็นข้อความในฐานใดๆ

- สมมติว่าต้องการเปลี่ยนเลขจำนวนเต็มไปเป็นข้อความในบางฐานระหว่างฐานสองกับฐานสิบหก ตัวอย่างเช่นต้องการเปลี่ยนเลขจำนวนเต็ม 10 ไปเป็นข้อความในฐานสิบได้ 10 หรือเปลี่ยนไปเป็นข้อความในฐานสองได้ 1010

- เราเคยเขียนโปรแกรมแก้ไปแล้วโดยใช้ stack
- เราจะมาดูกันหากแก้ด้วย recursive

- มาดูตัวอย่างของการใช้ฐาน 10 และเลข 769

- สมมติว่าเรามีลำดับของอักขระที่สอดคล้องกับฐาน 10 เช่น

conv_string = “0123456789”

มันง่ายในการเปลี่ยนจำนวนที่น้อยกว่า 10 ไปเป็น string ที่เท่ากัน โดยการมองไปในลำดับ เช่น ถ้าจำนวนที่สนใจเป็น 9 แล้ว ข้อความใน conv_string[9] หรือ “9”

ถ้าเราสามารถแบ่งเลข 769 ออกเป็นเลขโดด 3 ตัว(7, 6, 9)แล้ว การเปลี่ยนเป็นเป็นข้อความจะง่าย

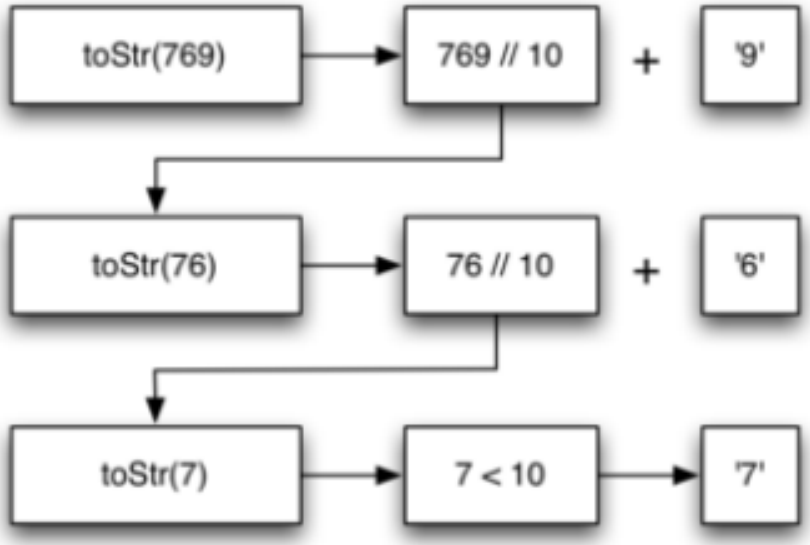
ดังนั้นจำนวนที่น้อยกว่า 10 น่าจะเป็น base case

- เมื่อรู้ว่าอะไรคือ base case แล้ว ภาพรวมของโปรแกรมของเราจะเกี่ยวข้องกับ 3 ส่วน
  - ลดเลขตั้งต้นให้เป็นลำดับของเลขโดด
  - เปลี่ยนเลขโดดให้เป็นข้อความโดยใช้การเทียบ
  - ต่อข้อความที่เป็นเลขโดดเข้าด้วยกันเป็นคำตอบ

- ขั้นต่อไปเป็นการหาว่าจะเปลี่ยน state อย่างไรเพื่อให้มีความก้าวหน้าจนไปถึง base case ได้
- เนื่องจากเราทำงานกับจำนวนเต็ม พิจารณาว่าการดำเนินการทางคณิตศาสตร์อะไรที่ทำให้จำนวนลดลง
- ตัวเลือกที่น่าจะเป็นไปได้คือ การหาร และ การลบ
- การลบน่าจะใช้ได้แต่ว่ามันไม่ชัดเจนว่าเราจะลบอะไร
- การหารกับส่วนที่เหลือให้สิ่งที่เราต้องการ ดังนั้นเราจะมาดูว่าอะไรเกิดขึ้นถ้าเราหารด้วยฐานที่เราต้องการเปลี่ยน

- เมื่อเราหาร 769 ด้วย 10 เราจะได้ 76 และได้เศษ 9
- อย่างแรกเราได้เศษ คือจำนวนที่น้อยกว่าฐานที่เราต้องการเปลี่ยน ซึ่งเราเปลี่ยนเป็น string ได้เลย
- อย่างที่สองเราได้จำนวนที่น้อยกว่าค่าเริ่มต้นและลดลงมุ่งไปสู่ base case
- งานต่อไปก็คือ เราจะเปลี่ยน 76 ให้ไปเป็น string ซึ่งเราจะได้ 7 เศษ 6
- สุดท้ายเราลดรูปปัญหาได้เหลือเพียงการเปลี่ยน 7 จากเงื่อนไขที่ว่า n <= base ซึ่ง base =10

- ลำดับของการทำงานเป็นดังรูปนี้



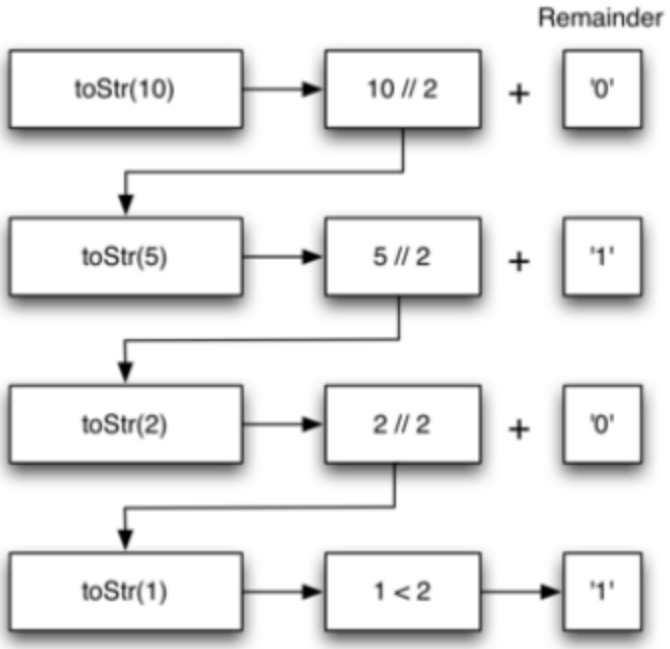
- สังเกตว่าเลขที่เราต้องจำคือ เลขเศษ ที่อยู่ในกล่องด้านขวา

- หากเขียนเป็น Python ระหว่างฐาน 2 ถึงฐาน 16 จะได้

```
def to_str(n , base):
    convert_string = "0123456789ABCDEF"
    if n < base:
        return convert_string[int(n)]
    else:
        return to_str(int(n/base),base)+convert_string[int(n % base)]


print(to_str(1453,16))
```

- สังเกตว่าบรรทัดที่ 3 เราจะตรวจสอบ base case ว่า n น้อยกว่า base ที่เราต้องการเปลี่ยนหรือไม่
- ถ้าตรวจสอบพบ เราจะหยุดเรียกตัวเอง และคืนค่า string จากลำดับใน convert_string
- ในบรรทัดที่ 6 สอดคล้องกับกฎข้อ 2 และ 3 โดยเรียกตัวเองและลดขนาดปัญหาด้วยการหาร

- ให้ทดลองเปลี่ยนเลขฐานดู เช่น 10 เปลี่ยนเป็นฐาน 2


Remainder
toStr(10)
10 // 2
+
'0'
toStr(5)
5 // 2
+
'1'
toStr(2)
2 // 2
+
'0'
toStr(1)
1 < 2
'1'

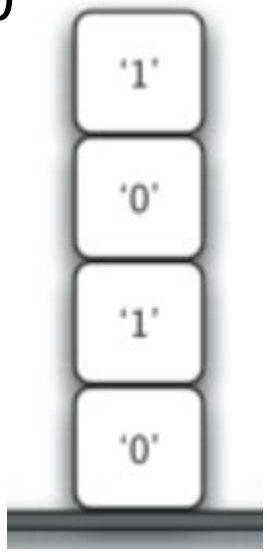
Implement recursion ด้วย stack

- สมมติว่าแทนที่เราจะเชื่อมผลลัพธ์ของการเรียก recursive ของ to_str กับข้อความจาก convertString เราจะปรับอัลกอริทึมของเราให้ push string บน stack แทนการเรียก recursive

- ปรับ code เป็นแบบนี้

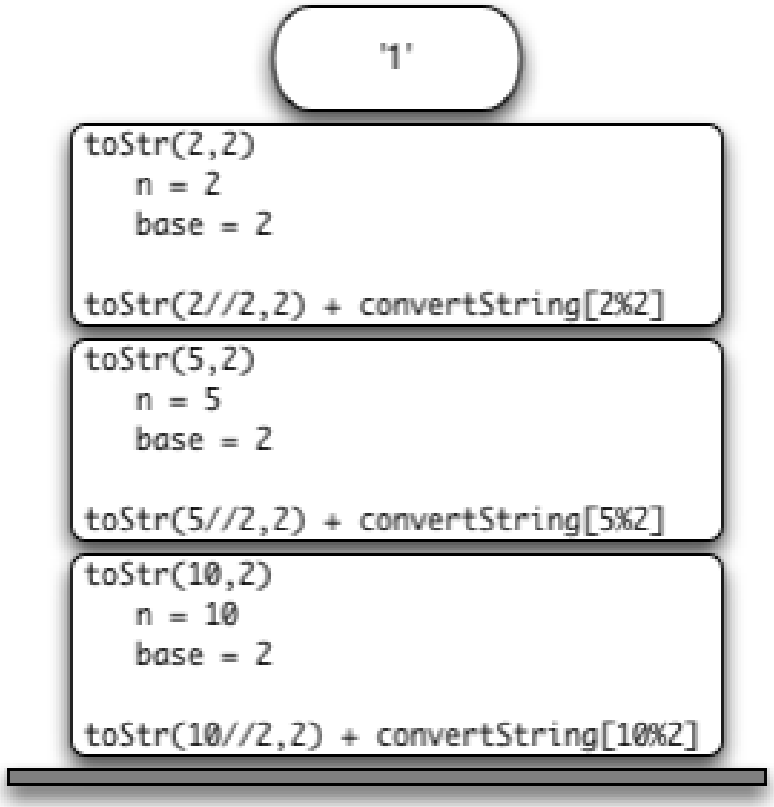
```
from mystack import Stack # As previously defined

r_stack = Stack()

def to_str(n, base):
    convert_string = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            r_stack.push(convert_string[n])
        else:
            r_stack.push(convert_string[n % base])
        n = n // base
    res = ""
    while not r_stack.is_empty():
        res = res + str(r_stack.pop())
    return res


print(to_str(1453, 16))
```

- ในแต่ละครั้งที่มีการเรียก to_str เราจะ push อักขระไว้บน stack จากตัวอย่างก่อนหน้า เราจะเห็นได้ว่าหลังจากมีการเรียก to_str ในครั้งที่ 4 ครั้ง stack จะมีหน้าตาแบบนี้
- สังเกตว่าเราสามารถ pop อักขระออกจาก stack แล้วเชื่อมกันเป็นผลลัพธ์สุดท้าย 1010



- ตัวอย่างก่อนหน้าทำให้เห็นว่า Python นั้นสร้างการเรียก recursive function นั้นทำอย่างไร
- เมื่อฟังก์ชันถูกเรียกใน Python, stack frame จะถูกจองเพื่อจัดการเก็บ local variable ของฟังก์ชัน
- เมื่อฟังก์ชันคืนค่า ค่าที่คืนคือ top ของ stack สำหรับการเรียกฟังก์ชัน

- ตัวอย่างของการ call stack ที่ถูกสร้างจาก to_str(10,2)



- Stack frame ให้ scope สำหรับตัวแปรที่ใช้ได้จากฟังก์ชัน แม้ว่าเราเรียกฟังก์ชันเดียวกันซ้ำแล้วซ้ำอีก การเรียกแต่ละครั้งจะสร้าง scope ใหม่ของตัวแปรที่เป็น local ของฟังก์ชันนั้น

- ถ้าหากเข้าใจเรื่อง stack จะทำให้เข้าใจ recursive function ง่ายขึ้น

แบบฝึกหัด

- จงเขียนฟังก์ชันที่รับ string แล้วตรวจสอบว่า string ที่รับมาเป็น palindrome หรือไม่ ให้คืนค่า True / False

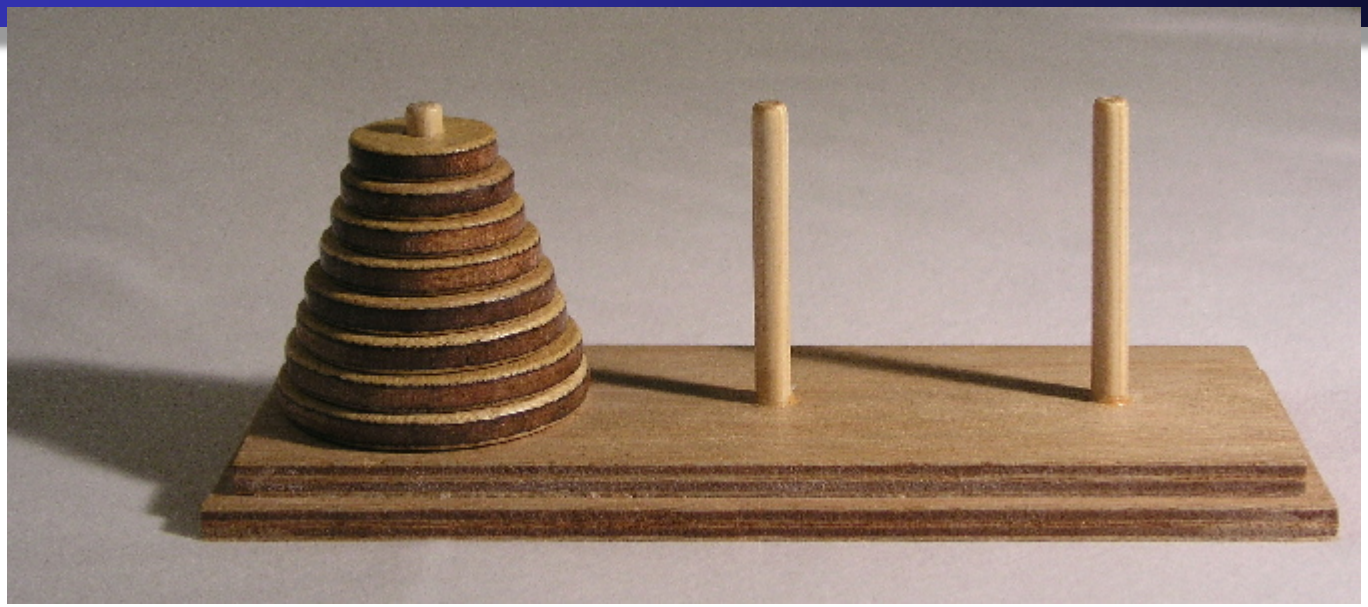
- String เป็น Palindrome ถ้าสะกดจากด้านหน้าไปหลัง และหลังไปหน้าได้ตัวเดียวกัน ตัวอย่างเช่น kayak

แบบฝึกหัด

- จงเขียนฟังก์ชัน คำนวณค่าลำดับ Fibonacci
- ค่า 1 1 2 3 5 8 13
- ตัวที่ 1 2 3 4 5 6 7
- เช่น print(fibo(5)) ได้ค่า 5
- print(fibo(6)) ได้ค่า 8

Tower of Hanoi

- หอคอยแห่งฮานอย เป็นเกมคณิตศาสตร์ ประกอบด้วยหมุด 3 แท่ง และ จานกลมแบนขนาดต่างๆ ซึ่งมีรูตรงกลางสำหรับให้หมุดลอด
- เกมเริ่มจากจานทั้งหมดวางอยู่ที่หมุดเดียวกัน โดยเรียงตามขนาดจากใหญ่ที่สุดอยู่ทางด้านล่าง จนถึงจานขนาดเล็กที่สุดอยู่ด้านบนสุด เป็นลักษณะกรวยคว่ำตามรูป



- เป้าหมายของเกมคือ พยายามย้ายกองจานทั้งหมดไปไว้ที่อีกหมุดหนึ่ง โดยการเคลื่อนย้ายจานจะต้องเป็นไปตามกติกาคือ
- สามารถย้ายจานได้เพียงครั้งละ 1 ใบ
- ไม่สามารถวางจาน ไว้บนจานที่มีขนาดเล็กกว่าได้

Tower of Hanoi

- หอคอยฮานอยเป็นเกมปริศนาที่สร้างขึ้นมาโดย นักคณิตศาสตร์ชื่อ Edouard Lucas ในปี 1883

- มีตำนานเกี่ยวกับวัดฮินดู ซึ่งมีห้องที่ภายใน มีเสา 3 หลัก และจานทองอยู่ 64 ใบ คล้องอยู่กับเสา โดยที่พราหมณ์ในโบสถ์นั้นจะทำการเคลื่อนย้ายจานทองตามคำสั่งที่ระบุไว้ในคำพยากรณ์ โดยการเคลื่อนย้ายนั้นจะต้องเป็นไปตามเงื่อนไขของเกมปัญหา

- คำพยากรณ์ในตำนานได้ทำนายไว้ว่า เมื่อปัญหาถูกแก้ วาระสุดท้ายของโลกจะมาถึง ดังนั้นปัญหานี้จึงมีอีกชื่อหนึ่งว่า ปัญหา "Tower of Brahma" (หอแห่งพรหม) ไม่มีข้อมูลเด่นชัดว่า ลูคาสนั้นเป็นผู้แต่งตำนานนี้ขึ้น หรือ ว่าได้รับแรงบันดาลใจจากตำนานนี้

- หากตำนานนี้เป็นจริง และ พราหมณ์สามารถย้ายจานด้วยความเร็ว 1 ใบต่อวินาทีและใช้จำนวนครั้งการย้ายที่น้อยที่สุด เวลาทั้งหมดที่ใช้ในการแก้ปัญหานี้คือ $2^{64} - 1$ วินาที หรือ ประมาณ 585 พันล้านปี (อายุของจักรวาลในตอนนี้ ประมาณ 13.7 พันล้านปี)

- http://gamecenter.kapook.com/flashgames-48574

- เราจะแก้ปัญหานี้แบบ recursive ได้อย่างไร
- ลองคิดแก้ปัญหานี้จากล่างขึ้นบน
- สมมติว่าเรามีเสาที่มี 5 จาน (เสาแรกหรือเสาซ้ายมือสุด) ถ้าเรารู้วิธีย้ายเสาเมื่อมี 4 จาน เราจะทำอย่างไร
- เราก็ย้าย 4 จานมาไว้เสากลาง ย้ายจานสุดท้ายไปไว้เสาที่สาม (ขวามือสุด) แล้วย้าย 4 จานไปไว้เสาที่สาม

- ถ้าเราไม่รู้วิธีการย้ายเสาเมื่อมี 4 จาน
- สมมติว่าเรารู้วิธีการย้าย เสาที่มี 3 จาน เราก็จะย้ายเสา 3 จาน ไปไว้เสาต้นกลาง แล้วย้ายจานที่ 4 ไปไว้เสาที่ 3 จากนั้นย้ายจานจากเสาต้นกลางมาต้นขวาสุด
- ถ้าไม่รู้ก็ดูว่า 2 จานย้ายอย่างไร
- ... ก็ทำแบบเดิม

- Outline สมมติว่าเราย้ายจาก starting pole ไป goal pole โดยใช้ intermediate pole
  1. ย้ายเสาที่มีจาน height-1 ไป intermediate pole โดยใช้ goal pole
  2. ย้ายจานที่ค้างอยู่ไปยัง final pole
  3. ย้ายเสาที่มีจาน height-1 จาก intermediate pole ไป goal pole โดยใช้ starting pole

ตราบเท่าที่เรายังใช้กฎที่ว่า ไม่สามารถวางจาน ไว้บนจานที่มีขนาดเล็กกว่าได้ เราจะสามารถใช้ 3 ขั้นตอนที่ว่าแบบ recursive ได้ โดยทำเหมือนกับว่าจานขนาดใหญ่ไม่ได้อยู่ตรงนั้น

สิ่งที่ขาดไปจาก outline คือ นิยามของ base case

ปัญหาง่ายสุดขอ tower of Hanoi คือมีจานกี่ใบ

- ใบเดียว คือเราย้ายไปยังเสาปลายทางได้เลย
- เพิ่มเติมขั้นตอนใน outline ที่จะทำให้เราเปลี่ยน state ไปยัง base case คือ การลดความสูงของ tower ในขั้นที่ 1 และ 3

```
def move_tower(height, from_pole, to_pole, with_pole):
    if height >= 1:
        move_tower(height - 1, from_pole, with_pole, to_pole)
        move_disk(from_pole, to_pole)
        move_tower(height - 1, with_pole, to_pole, from_pole)
```

- สังเกตว่า code คล้ายกับคำอธิบายใน outline เลย
- หัวใจของอัลกอริทึมนี้คือการเรียก recursive 2 อันที่ต่างกัน บรรทัดที่ 3 และ 5
- บรรทัดที่ 3 ย้ายทุกจานยกเว้นจานสุดท้ายจากเสาเริ่มต้นไปยังเสาที่พัก
- บรรทัดต่อมาย้ายจานสุดท้ายไปยังเสาปลายทาง
- บรรทัดที่ 5 ย้ายทุกจานจากเสาที่พักไปยังเสาปลายทางวางบนจานสุดท้ายนั่นเอง

- Base case จะตรวจพบเมื่อความสูงเป็น 0 ในกรณีนี้จะไม่ทำอะไร

- ฟังก์ชัน moveDisk จะเป็นการ print ว่าเราย้ายจานจากที่ไหนไปที่ไหน

```
def move_disk(fp,tp):

    print("moving disk from",fp,"to",tp)
```

```
def move_tower(height, from_pole, to_pole, with_pole):
    if height >= 1:
        move_tower(height - 1, from_pole, with_pole, to_pole)
        move_disk(from_pole, to_pole)
        move_tower(height - 1, with_pole, to_pole, from_pole)


def move_disk(fp,tp):
    print("moving disk from",fp,"to",tp)
move_tower(3, "A", "B", "C")
```