CS1010S

Tutorial 4: Advanced Recursion

Nicholas Russell Saerang (russellsaerang@u.nus.edu)

Announcement

Week 7 tutorial will be a consultation.

NO attendance for week 7 tutorial.

You may come and discuss midterm question with me.

Table of contents

1 Recap

2 Tutorial 4

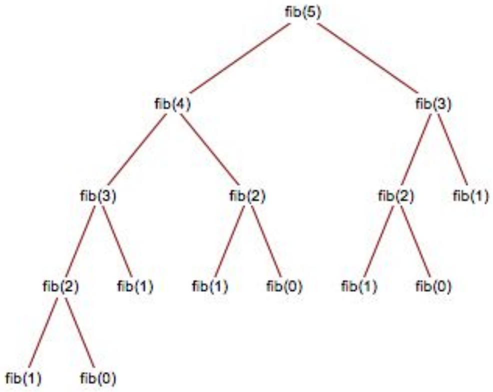
Recap

Recursion model

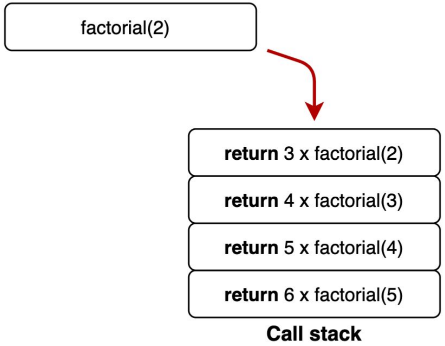
Recap

For iteration, you can model / trace the process using trace table

Similarly, for recursion, we have model to trace the process!!



Recursion tree



Call stack

```
(factorial(6))
(6 * factorial(5))
(6 * (5 * (4 * factorial(3))))
```

Expand expression

Useful to see how many operations need to be done

Useful to model linear and nonlinear recursion

Recursion tree

Useful to see the order of function being called

Call stack

Easy to model for linear recursion

Expand expression

Tutorial 4

Advanced Recursion

Tutorial 2 Q2: Recursion relation

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

(a) Implement a function that computes f(n) by means of a recursive process.

```
def f(n):
    if n < 3:
        return n
    else:
        return f(n-1) + 2*f(n-2) + 3*f(n-3)
```

Tutorial 2 Q2: Recursion relation

$$f(n) = \boxed{f(n-1) + 2f(n-2) + 3f(n-3)}$$

$$\boxed{f(n-1) = f(n-2) + 2f(n-3)} + 3f(n-4)$$

$$f(n-2) = f(n-3) + 2f(n-4) + 3f(n-5)$$

$$f(n-3) = f(n-4) + 2f(n-5) + 3f(n-6)$$

d **a** **b** **c**

Iterative step:

Tutorial 2 Q2: Recursion relation

```
 f(n)    =   f(n-1)   +  2f(n-2)   +  3f(n-3)
f(n-1)   =   f(n-2)   +  2f(n-3)   +  3f(n-4)

f(n-2)   =   f(n-3)   +  2f(n-4)   +  3f(n-5)

f(n-3)   =   f(n-4)   +  2f(n-5)   +  3f(n-6)
```

d **a** **b** **c**

```
Iterative step:   d = a + 2b + 3c
                  a, b, c = d, a, b
```

Tutorial 2 Q2: Recursion relation

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

(b) Implement a function that computes f(n) by means of an iterative process.

```
def f(n):
    "initial values"
    "iterating step"
```

Tutorial 2 Q2: Recursion relation

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

(b) Implement a function that computes f(n) by means of an iterative process.

```
def f(n):
    if n < 3:
        return n
    a, b, c = f(2), f(1), f(0)
    i = 0
    while i < n-2:
        d = a + 2*b + 3*c
        a, b, c = d, a, b
        i += 1
    return d
```

Question 1: Evaluation

What is the value of res?

```
def f(x, y, z):
    if x < y:
        return 0
    if z != 0:
        return -x + f(x-1, y, (x-y)%2)
    else:
        return x + f(x-1, y, (x+y)%2)

res = f(7,1,0)
```

Draw the recursion tree model!

What are the values of (x, y, z) as you go down the tree?

Question 1: Evaluation

What is the value of res?

```
def f(x, y, z):
    if x < y:
        return 0
    if z != 0:
        return -x + f(x-1, y, (x-y)%2)
    else:
        return x + f(x-1, y, (x+y)%2)

res = f(7,1,0)
```

Draw the call stack model!

f(7,1,0) -> return 7 + f(6, 1, 0)

-> return 6 + f(5, 1, 1) -> …

Question 1: Evaluation

```
Write the expression!
(f(7,1,0))
(7 + f(6,1,0))
(7 + (6 + f(5,1,1)))
 . # continue
 . # expand
(7 + (6 + (-5 + (4 + (-3 + (2 + (-1 + f(0,1,0))))))) # Base case
(7 + (6 + (-5 + (4 + (-3 + (2 + (-1 + 0))))))) # Base case
 . # just
 . # simple
 . # math
(10)
```

Question 1: Evaluation

There are different ways to think about recursion

Which one is the best?

It depends on the question ...

Choose the one that makes sense to you!

Question 2: Recursive num_pairs

Implement a recursive version of num_pairs!

num_pairs takes in string and return number of adjacent pair.

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
    return res
```

Question 2: Recursive num_pairs

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
    return res
```

Convert this to recursion!!

Base case: When do you stop?

Recursive Case 1: What you do if is adjacent pair?

Recursive Case 2: What you do if is not adjacent pair?

Question 2: Recursive num_pairs

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
    return res
```

Convert this to recursion!!

Base case: **When only one char left! (for loop condition)**

Recursive Case 1: What you do if is adjacent pair?

Recursive Case 2: What you do if is not adjacent pair?

Question 2: Recursive num_pairs

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
    return res
```

Convert this to recursion!!

Base case: When only one char left! (for loop condition)

Recursive Case 1: **Add 1 and continue check**

Recursive Case 2: What you do if is not adjacent pair?

Question 2: Recursive num_pairs

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
        else:
            continue # Do nothing and continue check
    return res
```

Convert this to recursion!!

Base case: When only one char left! (for loop condition)

Recursive Case 1: Add 1 and continue check

Recursive Case 2: What you do if is not adjacent pair?

Question 2: Recursive num_pairs

```
def num_pairs_iter(s):
    res = 0
    for idx in range(0, len(s)-1, 1):
        if s[idx] == s[idx+1]:
            res += 1
        else:
            continue # Do nothing and continue check
    return res
```

Convert this to recursion!!

Base case: When only one char left! (for loop condition)

Recursive Case 1: Add 1 and continue check

Recursive Case 2: **Do Nothing and continue check**

Question 2: Recursive num_pairs

```
def num_pairs(s):
    if len(s) < 2:
        return 0
    elif s[0] == s[1]:
        return 1 + num_pairs(substring(s, 1, len(s), 1))
    else:
        return num_pairs(substring(s, 1, len(s), 1))
```

Convert this to recursion!!

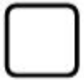
Base case: When only one char left! (for loop condition)

Recursive Case 1: Add 1 and continue check

Recursive Case 2: Add 0 (Nothing) and continue check

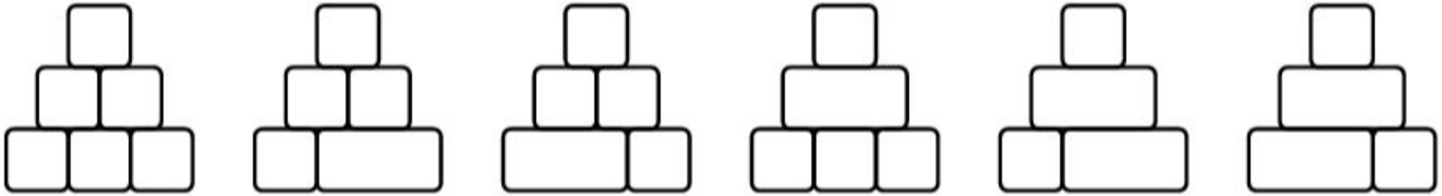
Question 3: Pyramids

For $n = 1$, there is only one way to build the pyramid:



For $n = 2$, there are two possible ways, using either two cubes, or one cuboid at the base:

For $n = 3$, there are 6 possible ways:



Implement the **pyramids** function that return the number of ways to create an n-layer pyramid

Question 3: Pyramids

The number of ways to construct an **N-layer pyramid** depends on the number of ways to construct an **(N-1)-layer pyramid** multiplied by the **number of ways to add the N-th layer**.

Question 3: Pyramids

The number of ways to construct an **N-layer pyramid** depends on the number of ways to construct an **(N-1)-layer pyramid** multiplied by the **number of ways to add the N-th layer**.

```
def bottom(n):
    # Wishful thinking

def pyramids(n):
    if n == 1:
        return 1
    else:
        return bottom(n) * pyramids(n-1)
```

Question 3: Pyramids

The number of ways to construct an **N-layer pyramid** depends on the number of ways to construct an **(N-1)-layer pyramid** multiplied by the **number of ways to add the N-th layer**.

```
def bottom(n):
    # Wishful thinking
```

Notice that bottom is a count change problem!!!

Choose to put 1-unit block

OR

Choose to put 2-unit block

Question 3: Pyramids

The number of ways to construct an **N-layer pyramid** depends on the number of ways to construct an **(N-1)-layer pyramid** multiplied by the **number of ways to add the N-th layer**.

```
def bottom(n):
    if n <= 2:
        return n
    else:
        return bottom(n-1) + bottom(n-2) # Count Change Idea!!
```

Question 3: Pyramids

The number of ways to construct an **N-layer pyramid** depends on the number of ways to construct an **(N-1)-layer pyramid** multiplied by the **number of ways to add the N-th layer**.

```
def bottom(n):
    if n <= 2:
        return n
    else:
        return bottom(n-1) + bottom(n-2) # Count Change Idea!!

def pyramids(n):
    if n == 1:
        return 1
    else:
        return bottom(n) * pyramids(n-1)
```

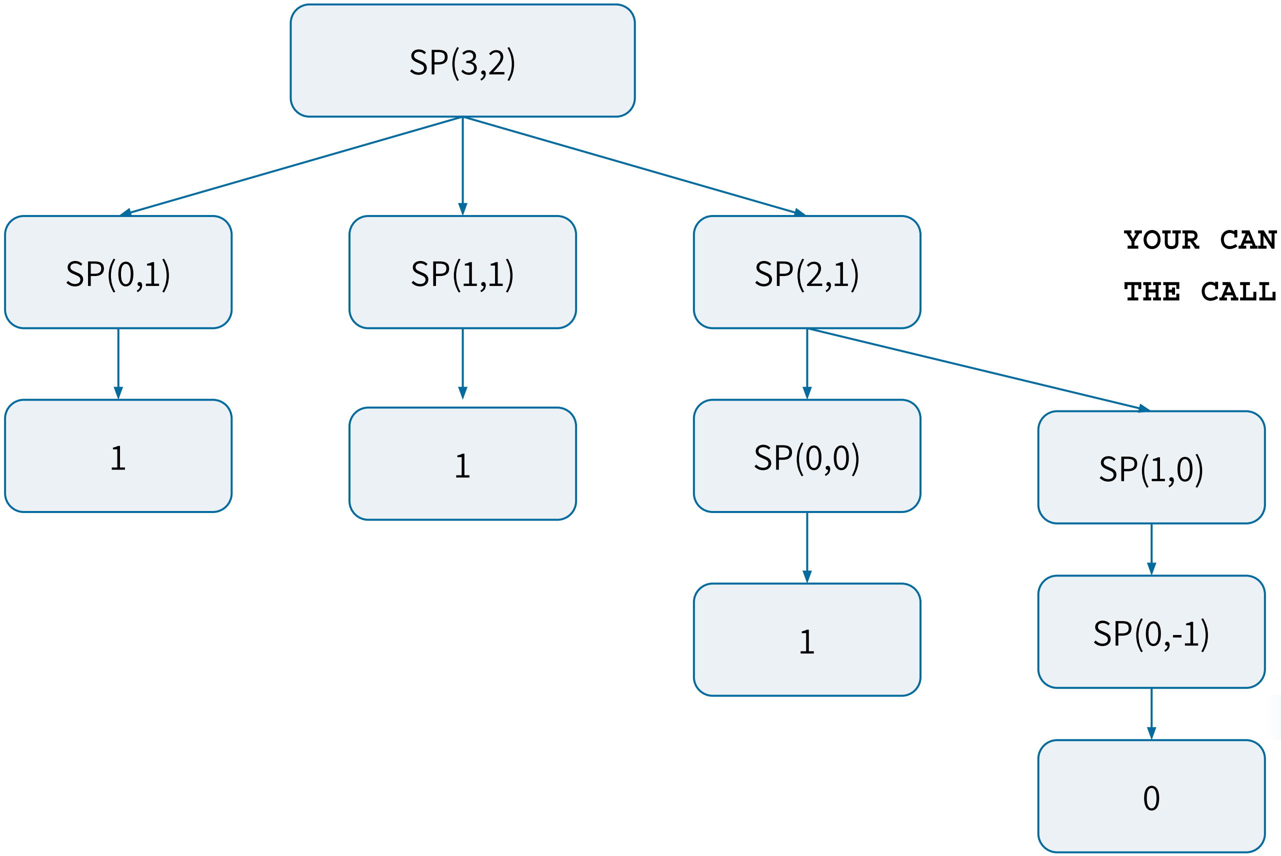
Extra Questions

```
def special_recursion(n, x):
    if n <= x:
        return 1
    res = 0
    for i in range(0, n, 1):
        res += special_recursion(i, x-1)
    return res


print(special_recursion(3, 2))
```

Draw the recursive tree!

Extra Questions



YOUR CAN TRY TO DRAW
THE CALL STACK!

The End