香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

AIE 1001 Introduction to AI Programming

Lecture 6
Recursion

Prof. Junjie Hu
School of Artificial Intelligence

Part of the slides are adapted from NJU SICP

Outline

- Recursion
  - Concept of recursion
  - Base cases and recursive cases
  - Recursion in Python
  - Binary search
  - Linear Recursion
  - Multiple recursion

Suppose you're waiting in line for a concert.

You can't see the front of the line, but you want to know what your place in line is. Only the first 100 people get free t-shirts!

You can't step out of line because you'd lose your spot.

What should you do?

An **iterative algorithm** might say:

1. Ask my friend to go to the front of the line.
2. Count each person in line one-by-one.
3. Then, tell me the answer.

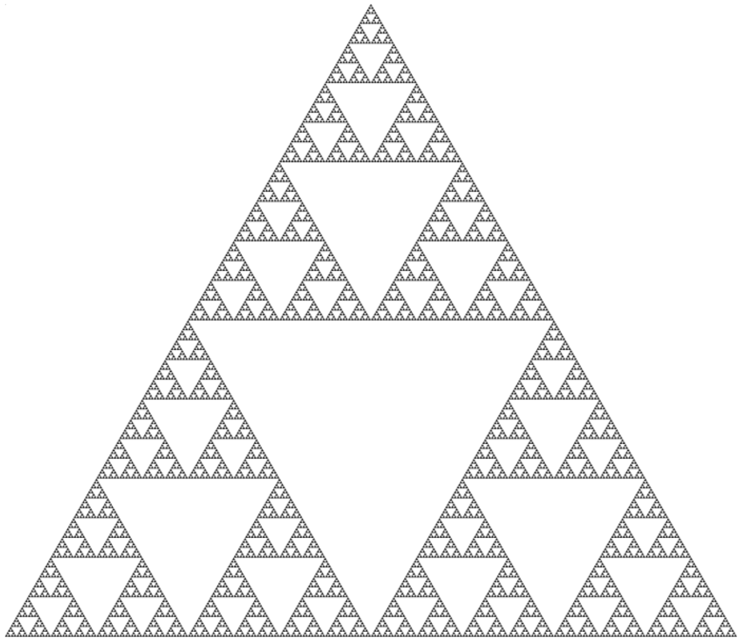
A **recursive algorithm** might say:

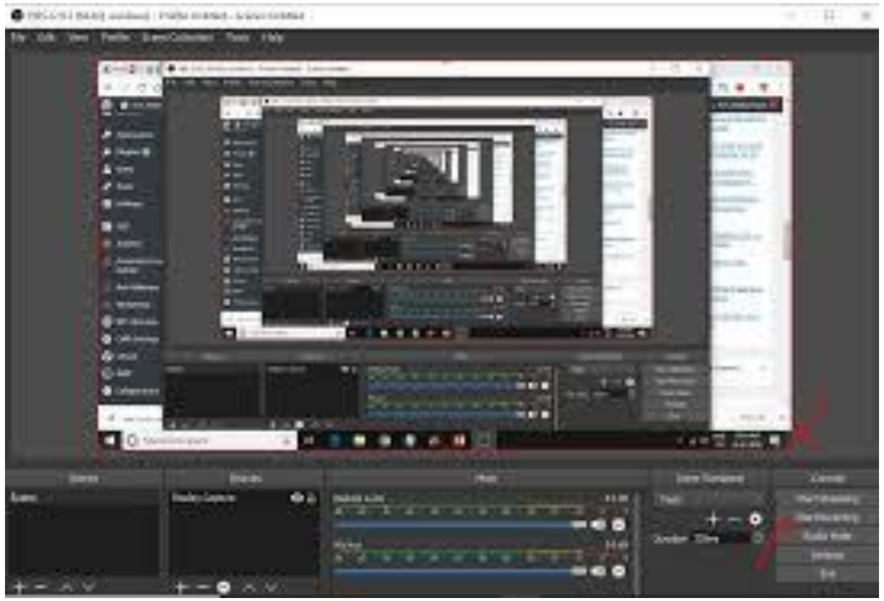
- If you're at the front, you know you're first.
- Otherwise, ask the person in front of you, **"What number in line are you?"**
- The person in front of you figures it out by asking the person in front of them who asks the person in front of them etc…
- Once they get an answer, they tell you and you add one to that answer.

Recursion

Recursion is useful for solving problems with a naturally repeating structure – they are defined in terms of themselves

It requires you to find patterns of smaller problems, and to define the smallest problem possible





Recursion

- Recursion is a technique by which a function makes one or more calls to itself during execution

- Recursion provides an elegant and powerful alternative for performing repetitive tasks

Example: The factorial function

- The factorial of a positive integer n, denoted n!, is defined as follows:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of permutations of n items

The recursive definition

- First, a recursive definition contains one or more base cases, a simple situation with a known answer (no more function calls). Without it, the function would keep calling itself forever.

- Second, it also contains one or more recursive cases, a way to reduce the problem to a smaller version of itself, and then use *the same function* to solve that smaller problem and move toward the base case.

The recursive definition of factorial function

- The factorial function can be naturally defined in a recursive way, for example, $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$

- More generally, for a positive integer n, we can define n! to be $n \cdot (n-1)!$

- Therefore, the recursive definition of factorial function is:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

Base case

Recursive case

Solution

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)
```

How Python implements recursion

- In Python, each time a function (recursive or otherwise) is called, a structure known as an activation record or frame is created in main memory to keep track of what's going on in that call.

- This activation record stores the function call's parameters and local variables

- If a function calls another function: Python pauses the current function. Its activation record stores where in the code to resume once the called function finishes (last called, first to finish).

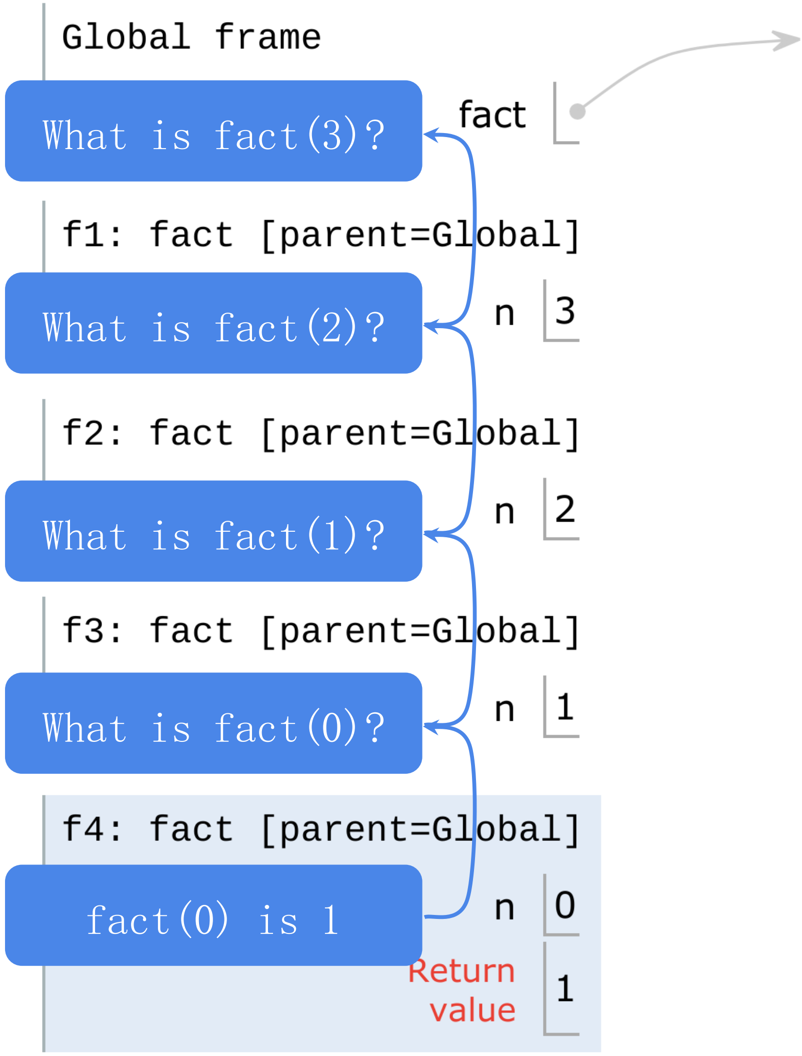
Recursion in environment diagram

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)

fact(3)
```

The same function fact is called multiple times, each time solving a simpler problem

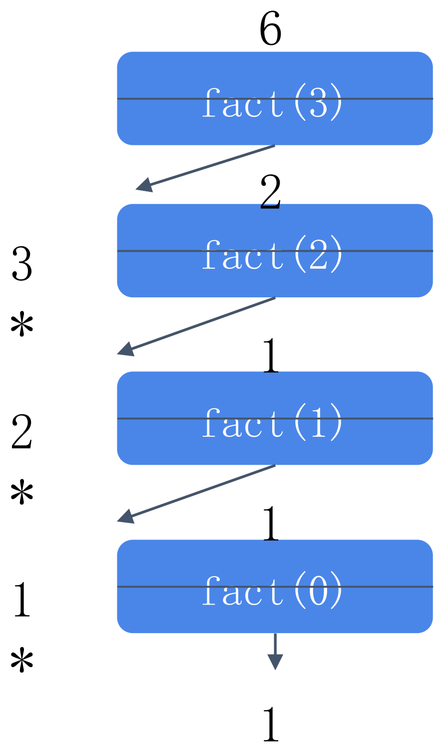
All the frames share the same parent - only difference is the argument

What n evaluates to depends upon the **current environment**



Recursive tree - another way to visualize recursion

```python
def fact(n):
	"""Calculates n!"""
	if n == 0:
		return 1
	else:
		return n * fact(n-1)
```



Example: Binary search

- A classic and very useful recursive algorithm, binary search, can be used to efficiently locate a target value within a sorted sequence of n elements

- When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set; This is known as the sequential search algorithm

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Binary search and sequential search

Feature	Binary Search	Sequential Search
Needs sorted data?	Yes	No
Speed	Very fast (halving each step)	Slower (checks each element)
Recursion friendly?	Yes!	Usually loops instead

Binary search

- When the sequence is sorted and indexable, binary search is a much more efficient algorithm

- For any index j, we know that all the values stored at indices $0, \ldots, j-1$ are less than or equal to the value at index j, and all the values stored at indices $j+1, \ldots, n-1$ are greater than or equal to that at index j

Guess a number x, where $x \in [0,100]$

The strategy of binary search

- We call an element of the sequence a candidate if, at the current stage of the search, we cannot rule out that this item matches the target

- The algorithm maintains two parameters, low and high, such that all the candidate entries have index at least low and at most high

- Initially, $low = 0$ and $high = n-1$. We then compare the target value to the median candidate, that is, the item data[mid] with index

$$mid = \lfloor (low+high)/2 \rfloor$$

The strategy of binary search

- If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully

- If target < data[mid], then we recur on the first half of the sequence, that is, on the interval of indices from low to mid−1

- If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high

Solution

```
def binarySearch(data, target, low, high):
    if low>high:
        print('Cannot find the target number!')
        return False
    else:
        mid = (low+high)//2
        if target==data[mid]:
            print('The target number is at position',mid)
            return True
        elif target<data[mid]:
            return binarySearch(data, target, low, mid-1)
        else:
            return binarySearch(data, target, mid+1, high)

def main():
    data = [1, 3, 5, 6, 16, 78, 100, 135, 900]
    target = 16
    binarySearch(data, target, 0, len(data)-1)
```

Case 1: Target = 3

data = [1, 3, 5, 6, 16, 78, 100, 135, 900]

```
Call: binarySearch(data, 3, 0, 8)
mid = (0+8)//2 = 4 → data[4] = 16
target < 16 → search left half

    binarySearch(data, 3, 0, 3)
    mid = (0+3)//2 = 1 → data[1] = 3
    target == 3 → found at position 1
```

```
binarySearch(0,8)
    ↓
    binarySearch(0,3)
        ↓
        FOUND target at index 1
```

Case 2: Target = 900

data = [1, 3, 5, 6, 16, 78, 100, 135, 900]

```
Call: binarySearch(data, 900, 0, 8)
mid = (0+8)//2 = 4 → data[4] = 16
target > 16 → search right half

    binarySearch(data, 900, 5, 8)
    mid = (5+8)//2 = 6 → data[6] = 100
    target > 100 → search right half

        binarySearch(data, 900, 7, 8)
        mid = (7+8)//2 = 7 → data[7] = 135
        target > 135 → search right half

            binarySearch(data, 900, 8, 8)
            mid = 8 → data[8] = 900
            target == 900 → found at position 8
```

```
binarySearch(0,8)
     ↓
     binarySearch(5,8)
          ↓
          binarySearch(7,8)
               ↓
               binarySearch(8,8)
                    ↓
                    FOUND target at index 8
```

Case 3: Target = 0

data = [1, 3, 5, 6, 16, 78, 100, 135, 900]

```
Call: binarySearch(data, 0, 0, 8)
mid = (0+8)//2 = 4 → data[4] = 16
target < 16 → search left half

    binarySearch(data, 0, 0, 3)
    mid = (0+3)//2 = 1 → data[1] = 3
    target < 3 → search left half

        binarySearch(data, 0, 0, 0)
        mid = 0 → data[0] = 1
        target < 1 → search left half

            binarySearch(data, 0, 0, -1)  # low > high → not found
```

```
binarySearch(0,8)
    ↓
    binarySearch(0,3)
        ↓
        binarySearch(0,0)
            ↓
            binarySearch(0,-1)
                ↓
                target NOT found
```

Practice: Power function

- Write a program to calculate the power function $f(x, n) = x^n$ using Recursion.

x*x*x* … *x

Solution:

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot x^{n-1}, & \text{if } n > 0 \end{cases}$$

Base case: If $n = 0$, return `1`.

Recursive case: Multiply `x` by `f(x, n-1)`.

```python
def mypower(x, n):
    # Base case
    if n == 0:
        return 1
    # Recursive case
    return x * mypower(x, n - 1)
```

A better recursive definition of power function

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

Fewer Function Calls

Instead of reducing n by 1 each time, we reduce it to $\lfloor n/2 \rfloor$ in each step.

That means:

- For $n = 1024$, basic recursion has 1024 calls.
- This method has only about $\log_2(1024) = 10$ calls.

Solution:

```python
def myPower(x,n):
    if n==0:
        return 1
    else:
        partial = myPower(x,n//2)
        result = partial * partial
        if n%2==1:
            result = result * x
        return result
```

Linear Recursion

- If a recursive function is designed so that each invocation of the body makes at most one new recursive call, this is known as linear recursion

- Calculating the factorial and performing binary search are both linear recursive algorithms, as there is only one recursive call in the function body

Linear Recursion

```python
def binarySearch(data, target, low, high):
    if low > high:
        print ('Cannot find the target number!')
    else:
        mid = (low+high) // 2
        if target==data[mid]:
            print('The target number is at position', mid)
            return True
        elif target<data[mid]:
            # print ('We are searching position:', low, 'to', mid-1)
            return binarySearch(data, target, low, mid-1)
        else:
            # print ('We are searching position:', mid+1, 'to', high)
            return binarySearch(data, target, mid+1, high)
```

Practice: Sum of a list

- Given a list of numbers, write a program to calculate the sum of this list using recursion

Solution:

```
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]

def main():
    L = [1,2,3,4,5,9,100,46,7]
    print('The sum is:',linearSum(L,len(L)))
```

`linearSum([1,100,7], 3)`

```
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]
```

n = 3

`linearSum([1,100,7], 2)`

```
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]
```

n = 2

`linearSum([1,100,7], 1)`

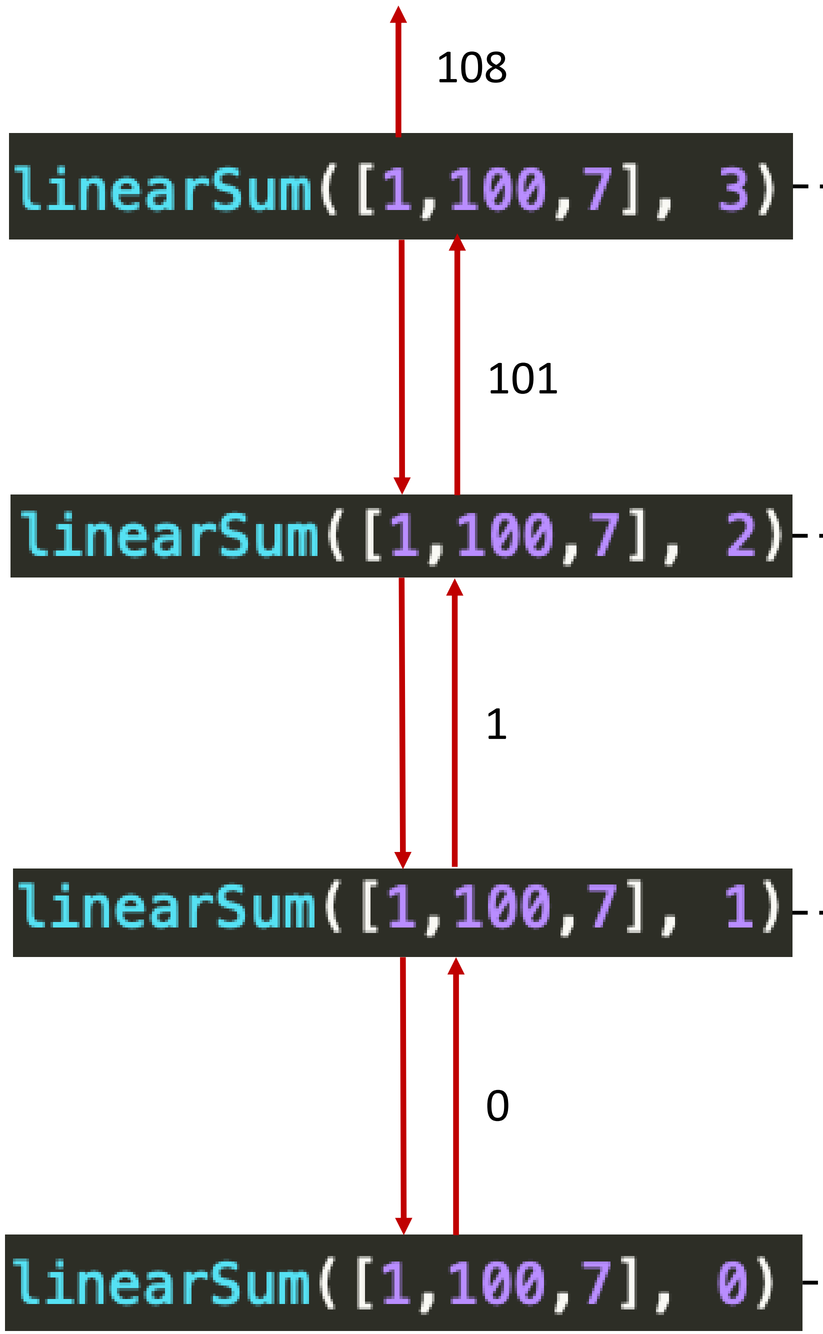
```
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]
```

n = 1

`linearSum([1,100,7], 0)`

```
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]
```

n = 0




```python
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return 101+L[n-1]
```

n = 3

```python
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return 1+L[n-1]
```

n = 2

```python
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return 0+L[n-1]
```

n = 1

```python
def linearSum(L,n):
    if n==0:
        return 0
    else:
        return linearSum(L,n-1)+L[n-1]
```

n = 0

Multiple recursion

- When a function makes two or more recursive calls, we say that it uses multiple recursion
- Drawing the English ruler is a multiple recursion program

Example: Drawing an English ruler

- We denote the length of the tick designating a whole inch as the ***major tick length***.

- Between the marks for whole inches, the ruler contains a series of ***minor ticks***, placed at intervals of 1/2 inch, 1/4 inch, and so on.

- As the size of the interval decreases by half, the tick length decreases by one

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

(a)

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

(c)

Recursive implementation of English ruler

- An interval with a central tick length $n \geq 1$ is composed of:
  - ✓ An interval with a central tick length $n-1$
  - ✓ A single tick of length n
  - ✓ An interval with a central tick length $n-1$

$$draw\_interval(n) = \begin{cases} \emptyset & \text{if } n = 0 \\ draw\_interval(n-1)\ ,\ draw\_line(n)\ ,\ draw\_interval(n-1) & \text{if } n > 0 \end{cases}$$

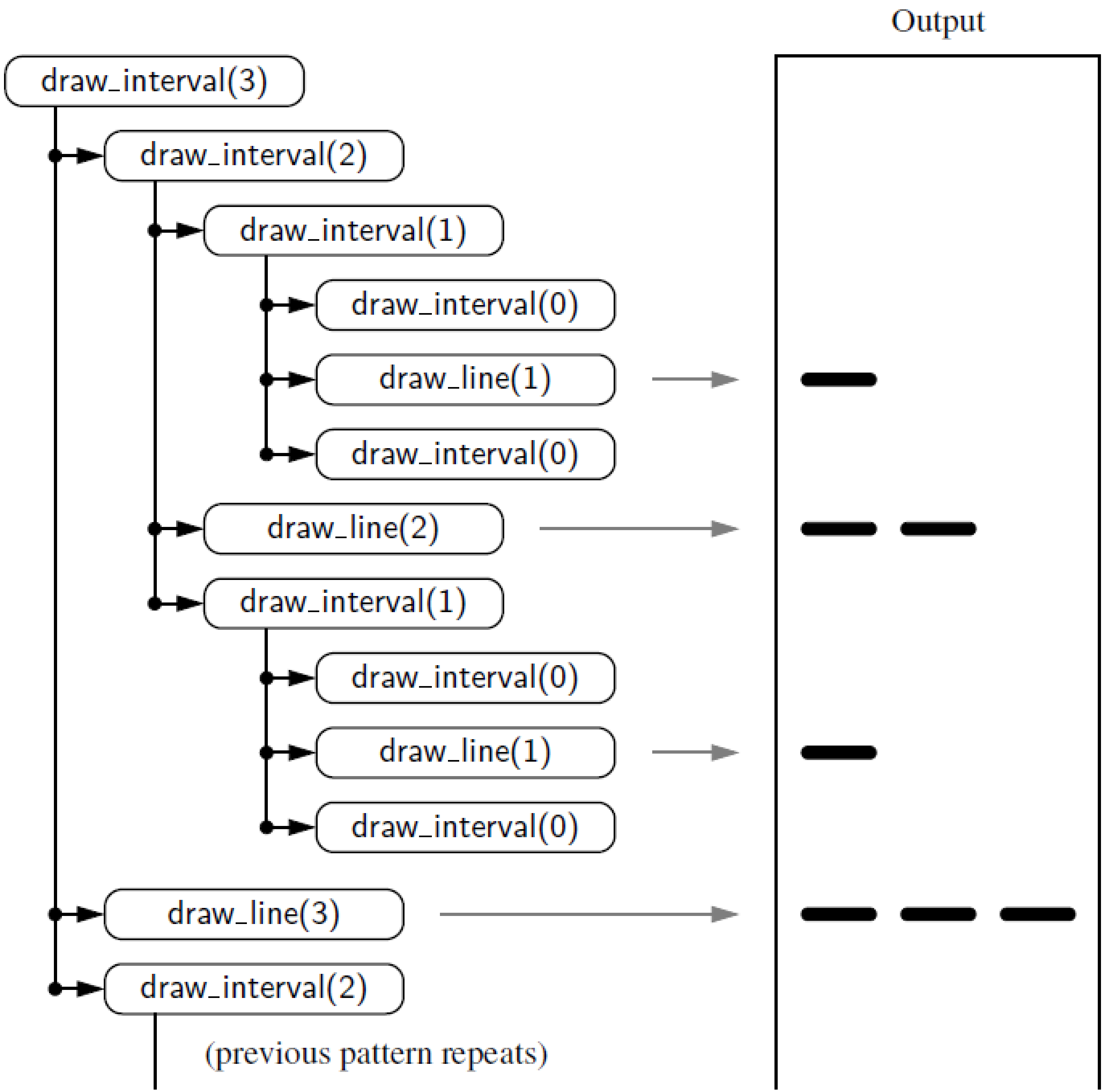
Solution

```
def draw_line(tickLen, tickLabel=''):
    line = '-'*tickLen
    if tickLabel:
        line+=' '+tickLabel
    print(line)

def draw_interval(centerLen):
    if centerLen>0:
        draw_interval(centerLen-1)
        draw_line(centerLen)
        draw_interval(centerLen-1)

def draw_ruler(numInch, majorLen):
    draw_line(majorLen,'0')

    for j in range(1, 1+numInch):
        draw_interval(majorLen-1)
        draw_line(majorLen, str(j))
```

```python
def draw_line(tickLen, tickLabel=''):
    line = '-'*tickLen
    if tickLabel:
        line+=' '+tickLabel
    print(line)

def draw_interval(centerLen):
    if centerLen>0:
        draw_interval(centerLen-1)
        draw_line(centerLen)
        draw_interval(centerLen-1)

def draw_ruler(numInch, majorLen):
    draw_line(majorLen, '0')

    for j in range(1, 1+numInch):
        draw_interval(majorLen-1)
        draw_line(majorLen, str(j))
```

```
draw_ruler(2, 3)
|
├── draw_line(3, '0')          --- 0
|
├── draw_interval(2)
|   ├── draw_interval(1)
|   |     ├── draw_interval(0)     [no output]
|   |     ├── draw_line(1)         -
|   |     └── draw_interval(0)     [no output]
|   ├── draw_line(2)               --
|   └── draw_interval(1)
|         ├── draw_interval(0)     [no output]
|         ├── draw_line(1)         -
|         └── draw_interval(0)     [no output]
|
├── draw_line(3, '1')          --- 1
|
├── draw_interval(2)
|   ├── draw_interval(1)
|   |     ├── draw_interval(0)     [no output]
|   |     ├── draw_line(1)         -
|   |     └── draw_interval(0)     [no output]
|   ├── draw_line(2)               --
|   └── draw_interval(1)
|         ├── draw_interval(0)     [no output]
|         ├── draw_line(1)         -
|         └── draw_interval(0)     [no output]
|
└── draw_line(3, '2')          --- 2
```

The recursive trace for English ruler



Practice: Binary sum

- Write a function binarySum() to calculate the sum of a list of numbers. Inside binarySum() two recursive calls should be made

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Practice: Binary sum

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

```
binarySum(L,start,mid)+binarySum(L,mid,stop)
```

Practice: Binary sum

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

```
elif start==stop-1:
    return L[start]
```

Solution:

```
def binarySum(L, start, stop):
    if start>=stop:
        return 0
    elif start==stop - 1:
        return L[start]
    else:
        mid = (start+stop)//2
        return binarySum(L,start,mid)+binarySum(L,mid,stop)

def main():
    L = [1,2,3,4,5,6,7]
    print(binarySum(L,0,len(L)))
```

Practice: Fibonacci sequence

- The **Fibonacci Sequence** is a series of numbers starting with **0** and **1**, where each succeeding number is the sum of the two preceding numbers. The sequence goes on infinitely. So, the sequence begins as:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Solution:

The Fibo

```
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage
num = 10
print(f"Fibonacci({num}) = {fibonacci(num)}")
```

- The *f* tells Python to evaluate expressions inside {} and insert them into the string.
- Very useful for dynamic text formatting.