COMP122/20 - Data Structures and Algorithms

09 Recursion

Instructor : Ke Wei 〖柯韋〗

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

February 17, 2020

Outline

Textbook §*4.1, 4.3.*

Recursive Functions

A function that calls itself in its body (why?) is a recursive function.

- Directly

```
def is_even(n):
    return True if n == 0 else not is_even(n-1)
```

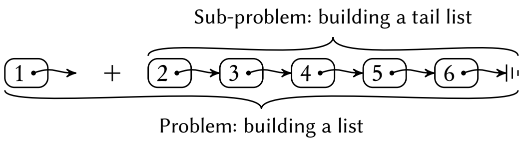
- Indirectly

```
def is_even(n):
    return True if n == 0 else is_odd(n-1)

def is_odd(n):
    return False if n == 0 else is_even(n-1)
```

Problems and Sub-problems

- Why must a function call itself?
- Some problems can be simplified to sub-problems that have similar or identical structures.
- It's simple to solve the main problem based on the solution of its sub-problems.
  - How can we build a list? If we have a smaller list (a sub-problem) as the tail, then we can add a head node in front of it to build a bigger list.



  - We must build the tail list first, and we can build it in the same way.
- There must be at least a *base case* to terminate the recursion. Solving the base case problem does *not* depend on the solution of any sub-problem.

Building a List Recursively

- Suppose we have the *Node* class with a constructor declared below:

```
class Node:
    def __init__(self, elm, nxt):
        self.elm = elm
        self.nxt = nxt
```

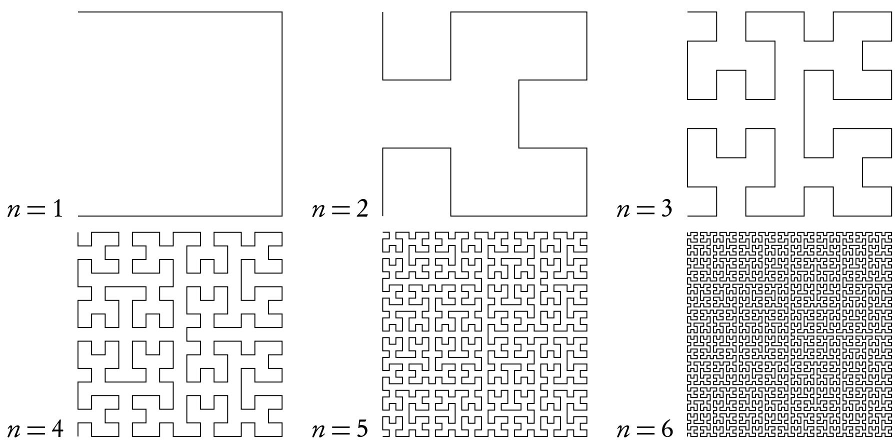
- The *build_list* function defined below builds a list of nodes containing integers from *start* to *stop*, and returns the reference to the head node.

```
def build_list(start, stop):
    return None if start >= stop else Node(start, build_list(start+1, stop))
```

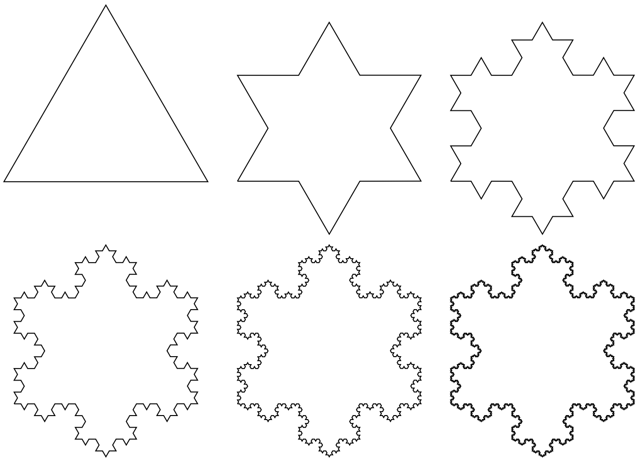
- The list shown in Slide 4 can be built by *build_list*(1, 7).

- Behind the magic, the actual work is just the reduction from the main problem to the sub-problems, and the recursion finds a proper way to *repeat* this reduction.
- The series of reductions builds on the ground of the solution of the base case.

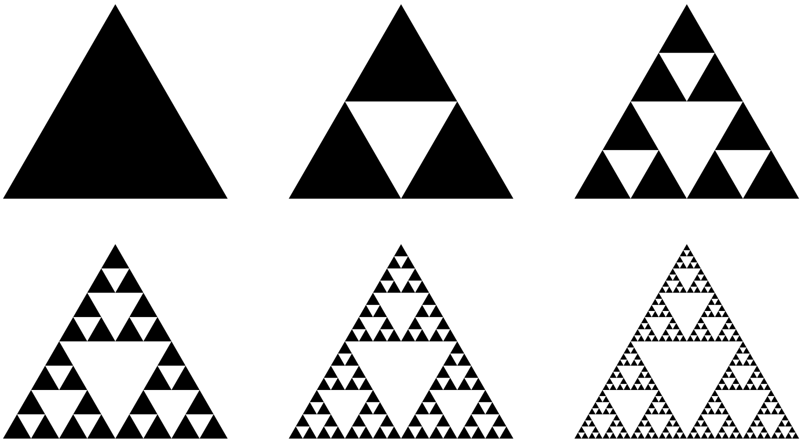
Hilbert Curve



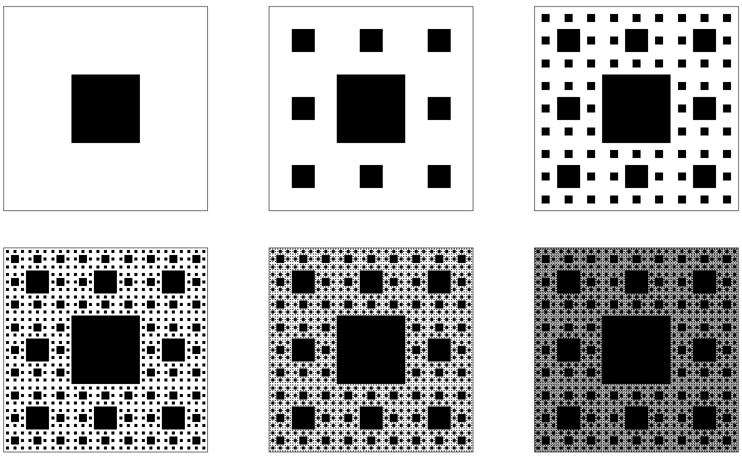
Koch Snowflake



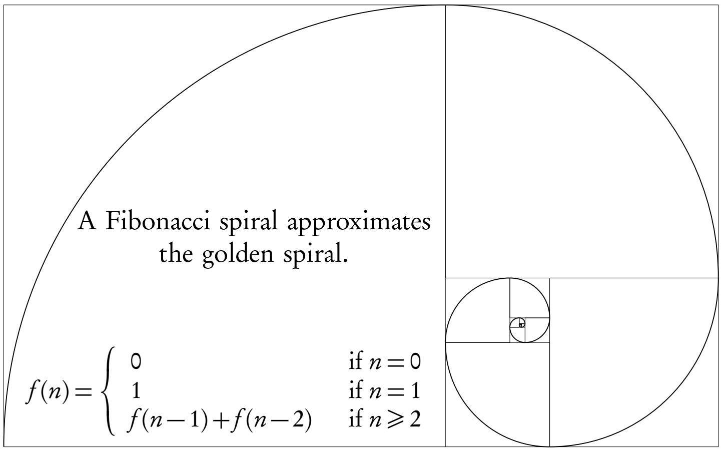
Sierpiński Triangle



Sierpiński Carpet



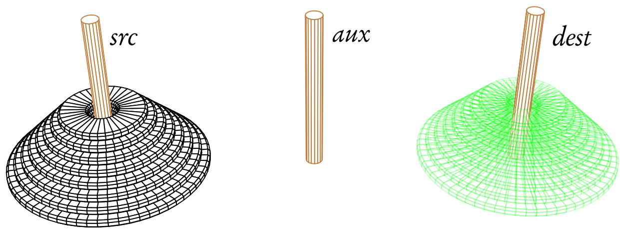
Fibonacci Spiral



$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geqslant 2 \end{cases}$$

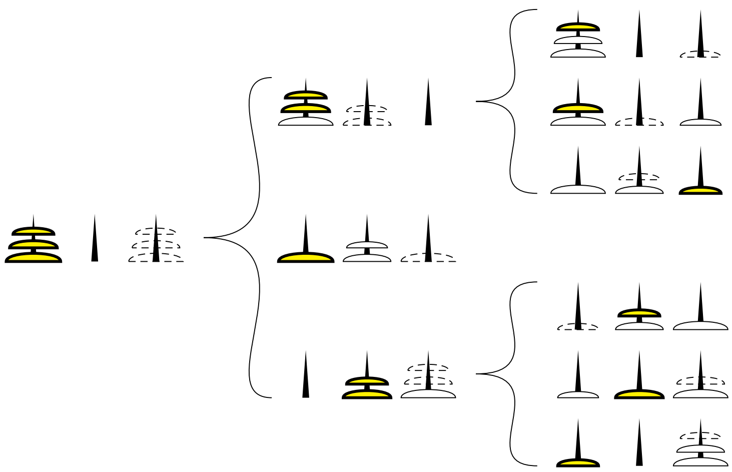
The Tower of Hanoi

- We are given a tower of 8 disks, initially stacked in decreasing size on one of three pegs:



- The objective is to transfer the entire tower to one of the other pegs, moving only one disk at a time and never moving a larger one onto a smaller one. (Édouard Lucas, 1883)
- This puzzle can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a direct solution is reached.

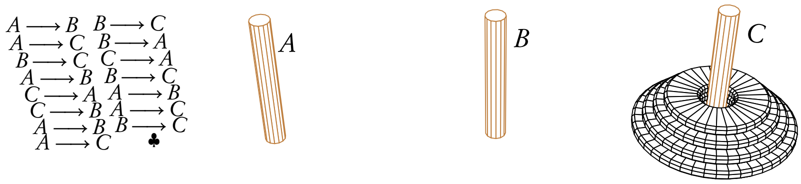
Moving Three Disks



A Recursive Solution

We need to construct a method to move n ($n \geqslant 0$) disks from peg *src* to peg *dest*, using peg *aux* as temporary storage — $moveTower(n, src, aux, dest)$.

- If n is 0, we need to do nothing.
- If n is greater than 0, we can break the task into smaller tasks:
  1. Move the top $n-1$ disks from peg *src* to peg *aux*, using peg *dest* as temporary storage — $moveTower(n-1, src, dest, aux)$.
  2. Move the bottom disk from peg *src* to peg *dest*.
  3. Move the top $n-1$ disks from peg *aux* to peg *dest*, using peg *src* as temporary storage — $moveTower(n-1, aux, src, dest)$.



The Tower of Hanoi — Code

- The state of the three towers is stored as three stacks of numbers ranging from 1 to n, with larger numbers representing larger disks.
- The steps of transferring the disks are generated as a sequence of snapshots of the intermediate states.

```
def hanoi(n):
    ts = [LnLs(range(1, n+1)), LnLs(), LnLs()]
    yield [list(t) for t in ts]
    yield from move_tower(ts, n, *ts)
def move_tower(ts, n, src, aux, dest):
    if n > 0:
        yield from move_tower(ts, n-1, src, dest, aux)
        dest.push(src.pop())
        yield [list(t) for t in ts]
        yield from move_tower(ts, n-1, aux, src, dest)
```


The Number of Steps

We define the number of steps required for transferring an n-disk tower as

$$steps(n).$$

- For transferring 0 disk, there is $0 = 2^0 - 1$ step.
- For transferring 1 disk, there is only $1 = 2^1 - 1$ step.
- For transferring 2 disks, there are $1 + 1 + 1 = 3 = 2^2 - 1$ steps.
- For transferring n disks, there are

$$steps(n-1) + 1 + steps(n-1) = \left(2^{n-1} - 1\right) + 1 + \left(2^{n-1} - 1\right) = 2^n - 1$$

  steps.