Lecture 9

Recursion

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Example: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes $h(n)$?

Example: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes $h(n)$?

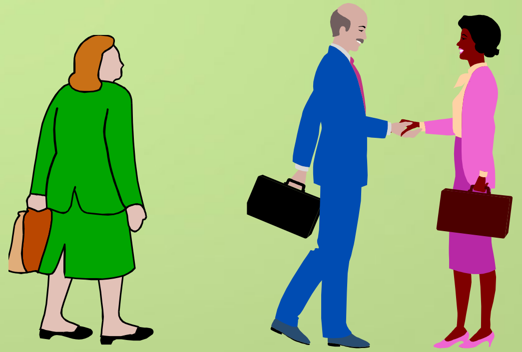
$h(2) = 1$

Example: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes h(n)?

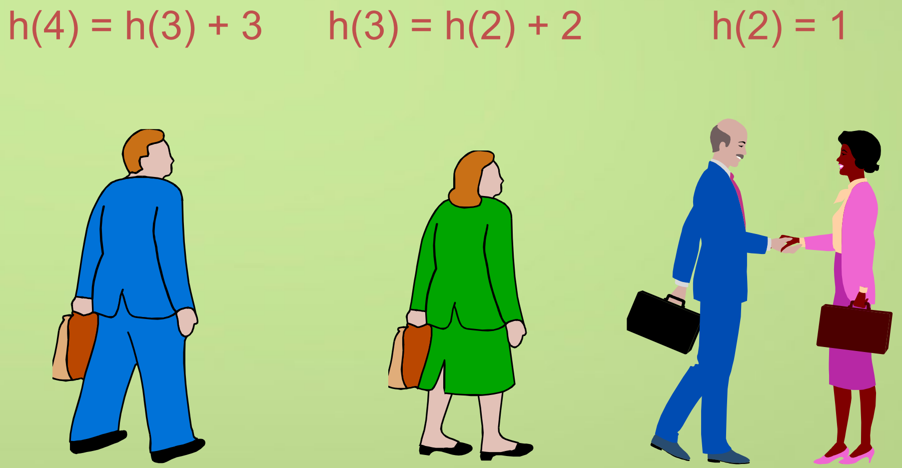
h(3) = h(2) + 2

h(2) = 1



Example: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes $h(n)$?



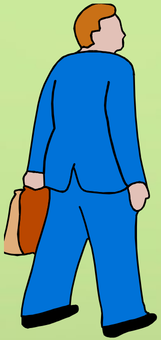
香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Example: The Handshake Problem

There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes $h(n)$?

$h(n) = h(n-1) + n-1$

$h(4) = h(3) + 3$



$h(3) = h(2) + 2$

$h(2) = 1$

Recursion

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Consider for example the factorial function:

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could also write:

```
6! = 6 * 5!
```

Recursion

In general, we can express the factorial function as follows:

```
n! = n * (n-1)!   // Are we done? Well… almost.
```

The factorial function is only defined for *non-negative* integers. So we should be a little bit more precise:

```
n! = 1              // if n is equal to 1
n! = n * (n-1)!     // if n is larger than 1
```

Recursion

- When a function calls itself, we speak of *recursion.*
- Implement n! using a recursive function:

```
public static int fact(int n){
    if(n<=1)
        return 1;
    else
        return n * fact(n-1);
}
```

Recursive method calls

- Assume the number typed is 3, that is, n=3.

```
fact(3)  :
   3 <= 1 ?                 No.
   fact3 = 3 * fact(2)
       fact(2)  :
            2 <= 1 ?
              No.
            fact2 = 2 * fact(1)
              fact(1)  :
                   1 <= 1 ?
                     Yes.
                   return 1
           fact2 = 2 * 1 = 2
           return fact2

    fact3 = 3 * 2 = 6
    return fact3

 fact(3)  has the value 6
```

```
public static int fact(int n){
        if(n<=1)
           return 1;
        else
           return n * fact(n-1);
    }
```

Recursion

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Here is a comparison of the recursive solution with the iterative solution:

```
public static int fact(int n){
    int t = 1;
    int counter = 1;

    while (counter <= n) {
        t = t * counter;
        counter = counter + 1;
    }
    return t;
}
```

```
public static int fact(int n){
        if(n<=1)
            return 1;
        else
            return n * fact(n-1);
    }
```

Recursion: Handshake problem

- Total number of handshakes for n persons:

  $$h(n) = h(n-1) + (n-1)$$

- Implement **h(n)** using a recursive method:

```
public static int handShake(int n){
    if(n <= 2)
        return n - 1;
    else
        return handShake(n-1) + (n-1);
}
```

- Alternative implementation:

  Sum of integers from 1 to n-1 = n(n-1) / 2

Recursion

- When we use recursion we must be careful not to create an infinite chain of recursive method calls:

```
public int fac(int n){
      return n * fac(n-1);
}                        // Oops! no termination condition
```

or:

```
public int fact(int n){
 if (n<=1)
      return 1;
 else
      return n * fact(n+1);     // Oops!
}
```

Example: Fibonacci Sequence

How many pairs of rabbits can be produced from a single pair in a year's time?

- Assumptions:
  - Each pair of rabbits produce a new pair of offspring every month;
  - each new pair becomes fertile at the age of one month;
  - none of the rabbits dies in that year.

- Example:
  - After 1 month there will be 2 pairs of rabbits;
  - after 2 months, there will be 3;
  - after 3 months, there will be 5 (since the following month the original pair and the pair born during the first month will both produce in a new pair and there will be 5 in all).

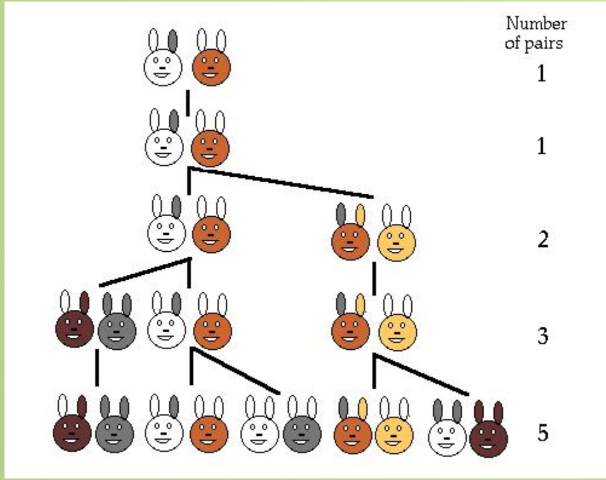
Computation Methods

- Fibonacci numbers:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

  where each number is the sum of the preceding two.

- Recursive definition:
  - F(0) = 0;
  - F(1) = 1;
  - F(n) = F(n-1)+ F(n-2);



Computing Fibonacci numbers

```
//Calculate Fibonacci numbers using recursive method
public class Fibonacci
{
    static int fib(int n){
        if (n == 0) return 0;
        if (n == 1) return 1;
        return (fib(n-1) + fib(n-2));
    }

    public static void main(String[] args) {
        IO.output("Enter the value n: ");
        int n = IO.inputInteger();
        int fibN = fib(n);
        IO.outputln("Fib(" + n + ") = " + fibN) ;
    }
}
```

Computation Methods

- Fibonacci numbers:

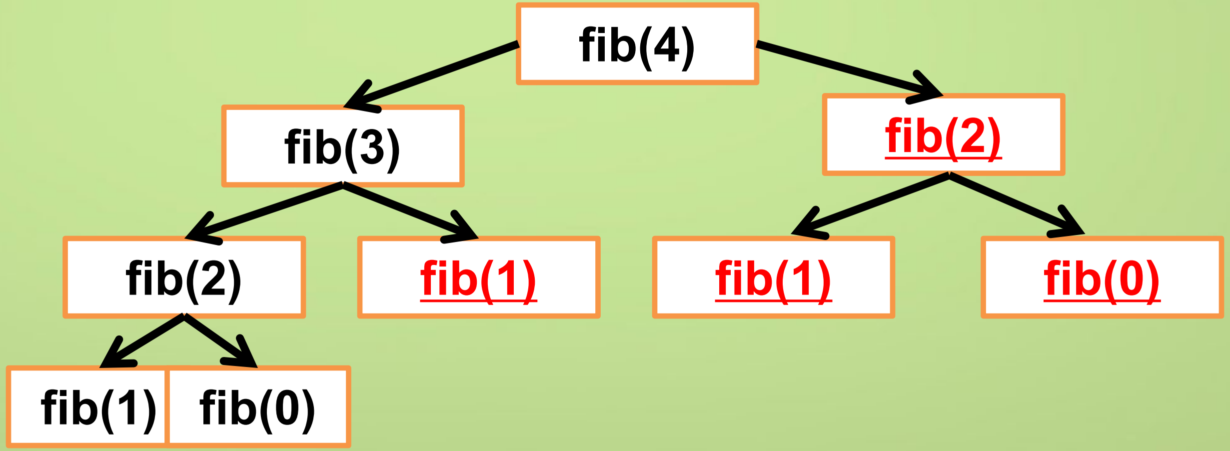
  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(n) = F(n-1)+ F(n-2);`

Computing Fibonacci numbers

- Calculating the 4th Fibonacci number fib(4) using recursion:
  - Many intermediate steps are re-calculated (underlined items)



Fibonacci Numbers

- Fibonacci numbers can also be represented by the following formula.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Other Recursive Applications

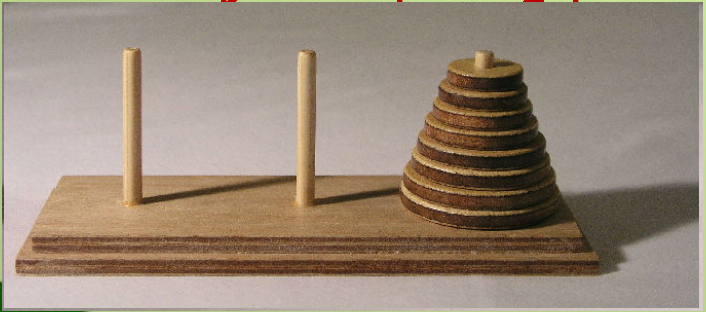
- Binary search:
  - Given a sorted array, the binary search find an element in the array efficiently.
    - Compare search element with middle element of the array
    - If not equal, then apply binary search to half of the array (if not empty) where the search element could be found

Binary Search with Recursion

```
/**
 * @param data    input array
 * @param lower   lower bound index
 * @param upper   upper bound index
 * @param value   value to search for
 * @return        index if found, otherwise return -1
 */
public int binSearch(int[] data, int lower, int upper, int value)
{
    int middle = (lower + upper) / 2;
    if (data[middle] == value)
        return middle;
    else if (lower >= upper)
        return -1;
    else if (value < data[middle])
        return binSearch(data, lower, middle-1, value);
    else
        return binSearch(data, middle+1, upper, value);
}
```

The Towers of Hanoi

- According to legend, monks in a remote monastery could predict when the world would end.
  - They had a set of 3 diamond needles.
  - Stacked on the first diamond needle were 64 disks of decreasing size.
  - Their task is to move all the disks from one needle to another by following certain rules.



From http://commons.wikimedia.org/wiki/File:Tower_of_Hanoi.jpeg

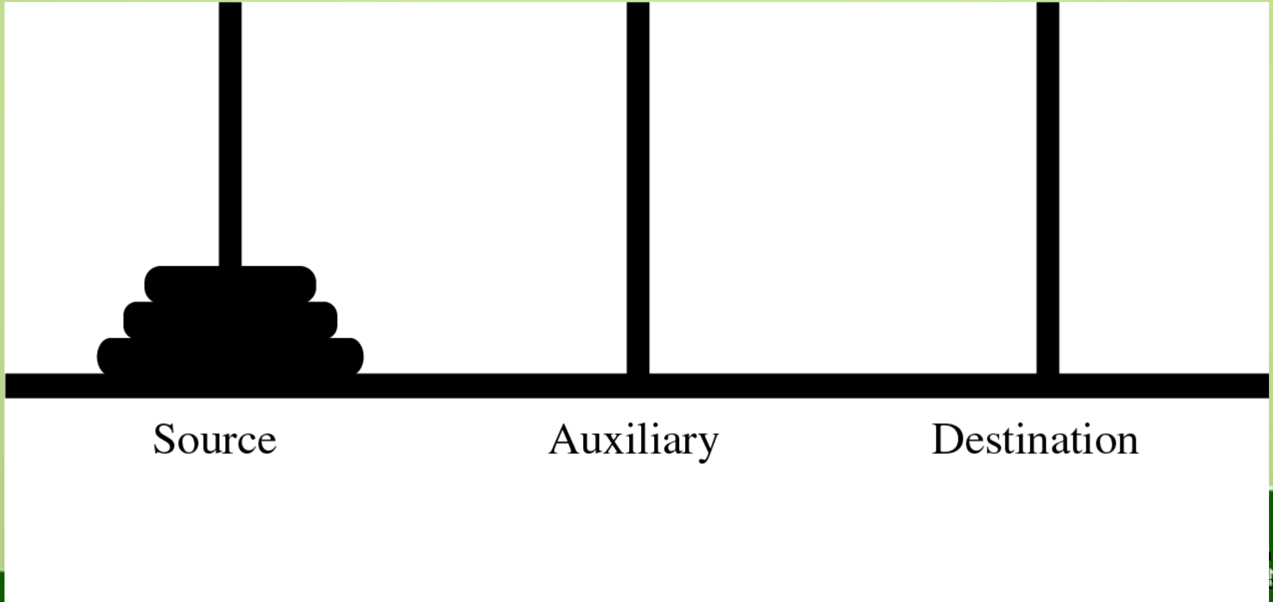
The world would end when they finished the task!

The Towers of Hanoi

- The monks moved one disk to another needle each day, subject to the following rules:
  - Only one disk could be moved at a time
  - A larger disk must never be stacked above a smaller one
  - One and only one extra needle could be used for intermediate placement of disks
- This task requires $2^{64}-1$ moves!
  - It will take 580 billion years to complete the task if it takes 1 sec. to moved each disk.
  - For n disks, $2^{n}-1$ moves are required

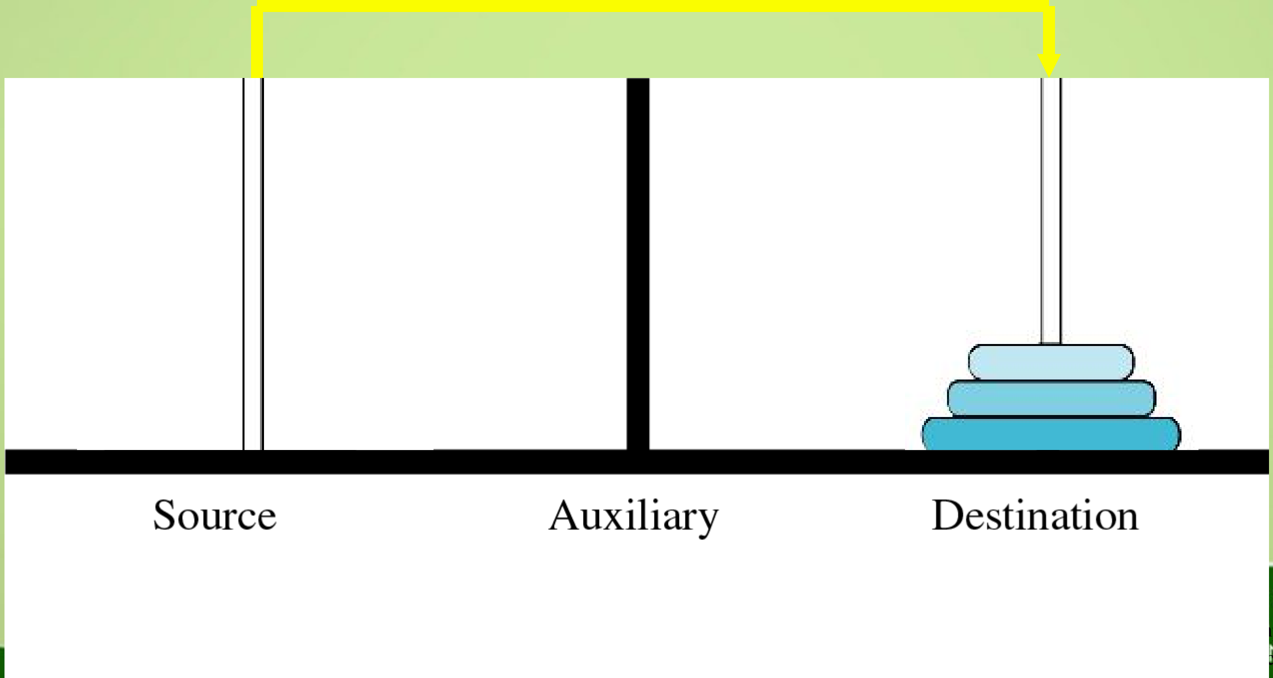
The Towers of Hanoi

Let's try some simple examples:



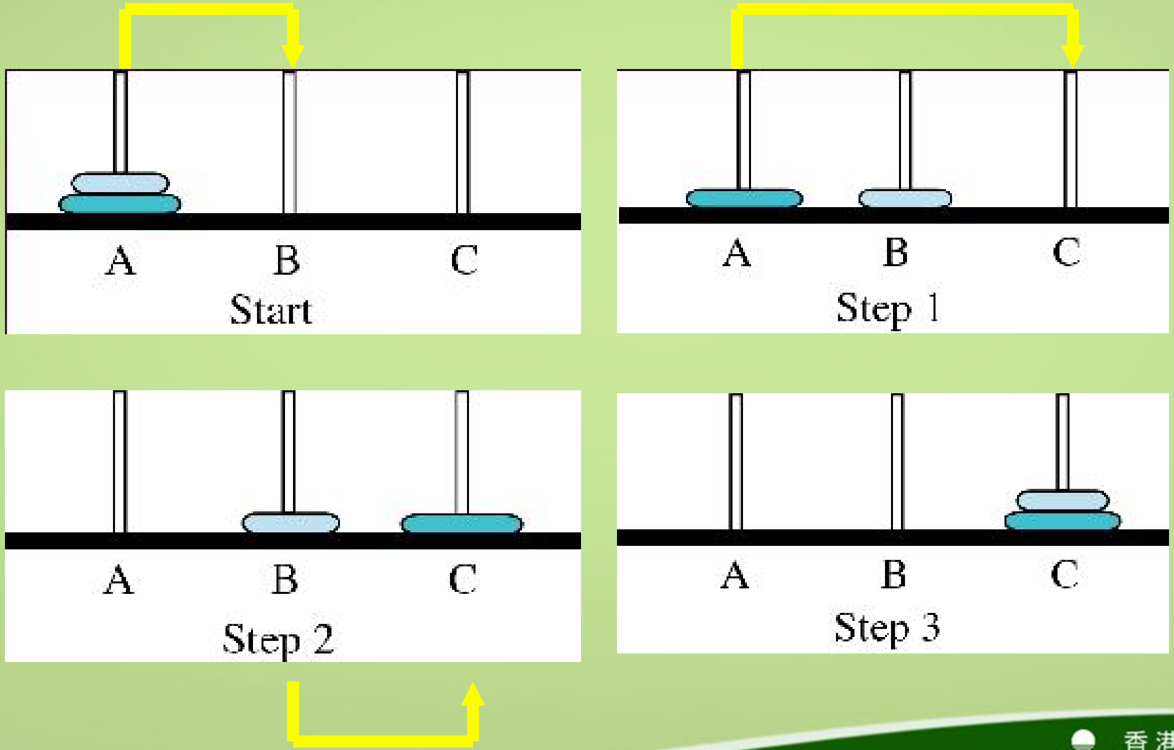
The Towers of Hanoi

Move all three disks from source to destination



The Towers of Hanoi

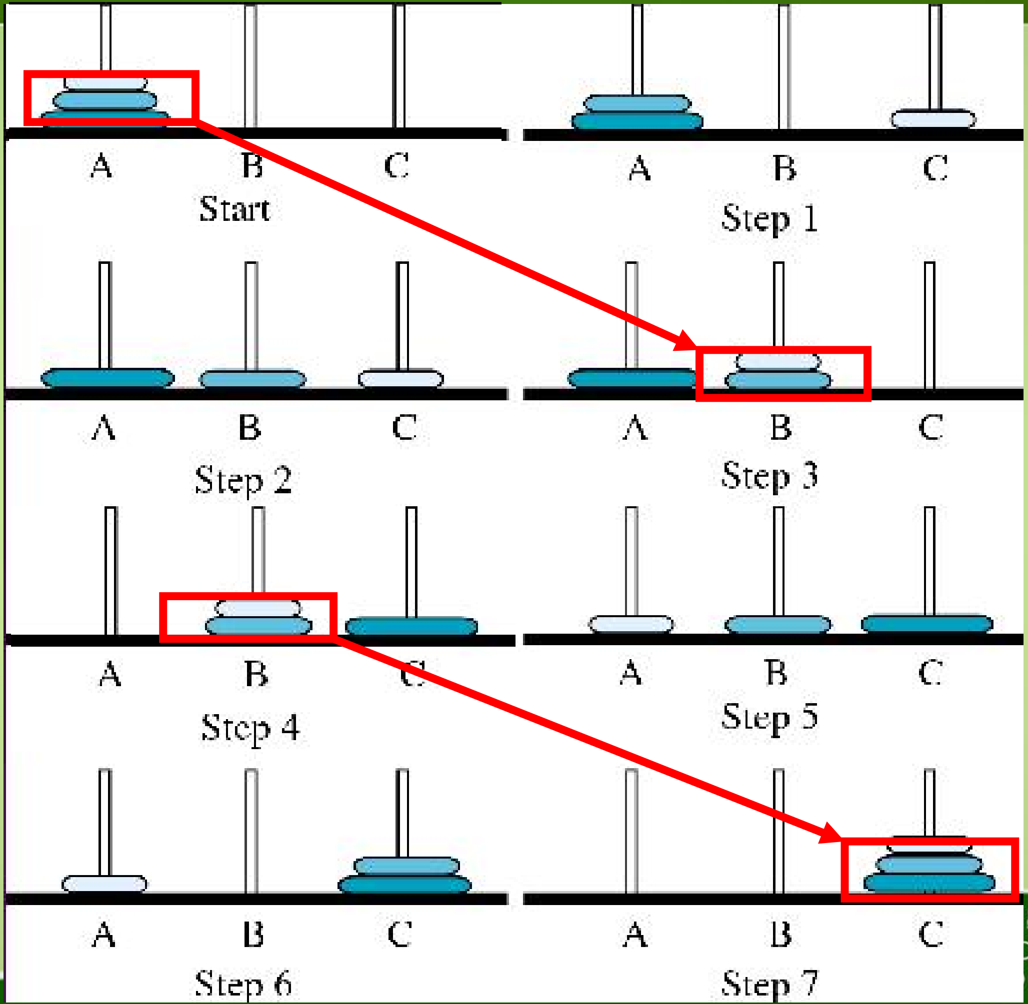
Moving 2 disks from A to C



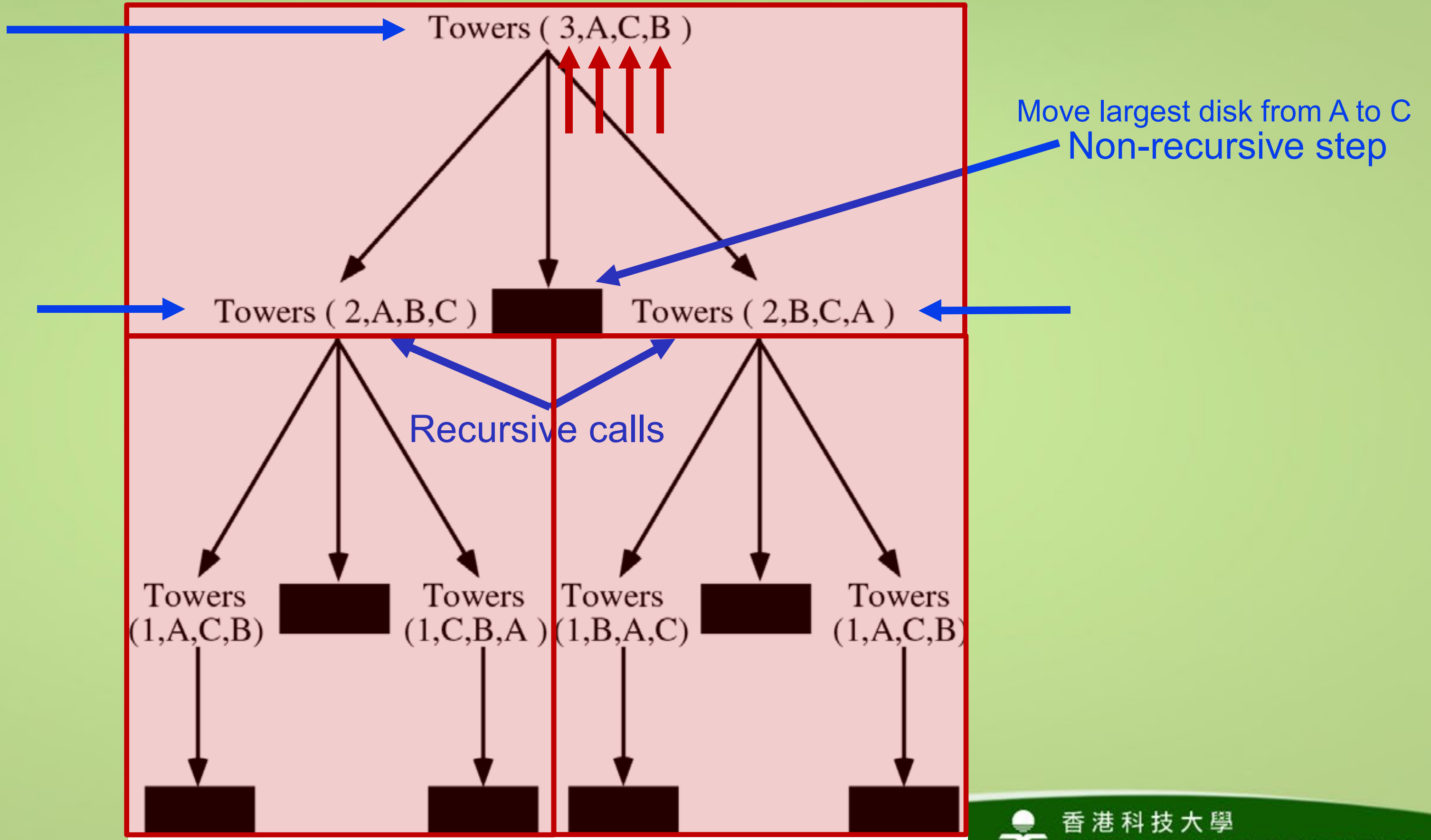
Moving first two disk from A to B

Moving third disk from A to C

Moving first two disk from B to C



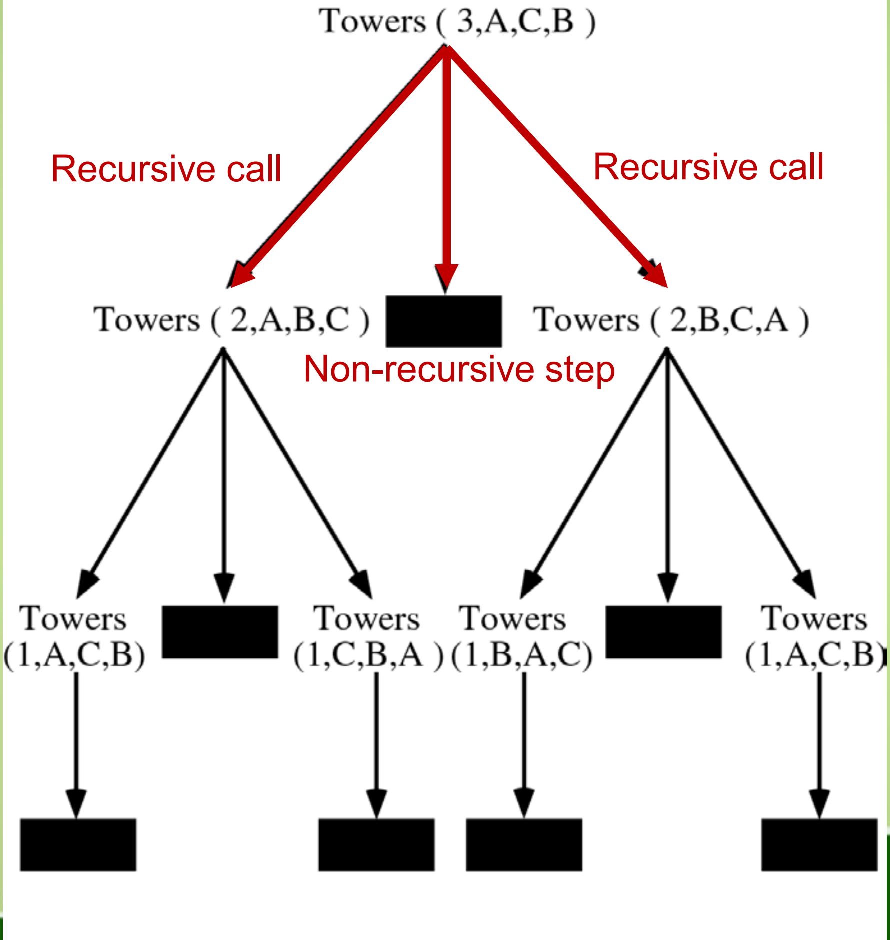
The Towers of Hanoi



The Towers of Hanoi

```
public void towers(int num, int from, int to) {
        int temp = 6 - from - to;
        if (num == 1){
            IO.outputln("Move disk 1 from " + from + " to " + to);
        } else {
            towers(num-1, from, temp);
            IO.outputln("Move disk "+ num +" from "+ from +" to " + to);
            towers(num-1, temp, to);
        }
    }
```

The Towers of Hanoi



The Towers of Hanoi

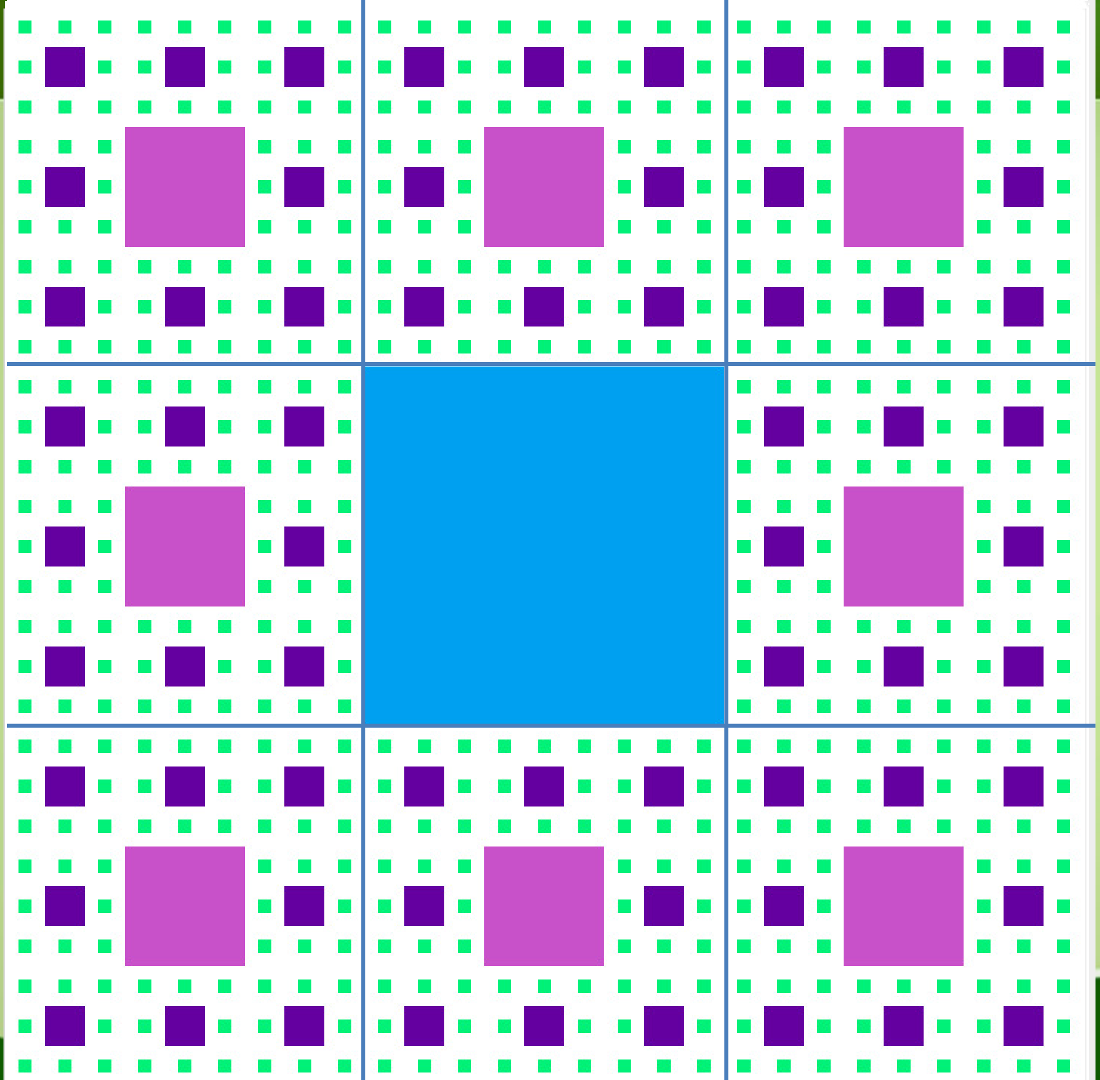
```
public void towers(int num, int from, int to) {
    int temp = 6 - from - to;
    if (num == 1){
        IO.outputln(“Move disk 1 from " + from + " to " + to);
    } else {
        towers(num-1, from, temp);
        IO.outputln(“Move disk "+ num +" from "+ from +" to " + to);
        towers(num-1, temp, to);
    }
}
```

Fractal

- A ***fractal*** is a mathematical set that displays self-similar patterns.
- Fractals appear the same or nearly the same at different scales.
- The term “fractal” was first used by Mandelbrot in 1975.

Sierpinski Carpet

- A Sierpinski carpet is created by
  - creating a square
  - dividing it into nine smaller squares
  - removing the central square
  - repeating the process for the eight other squares
- Each of the smaller squares is a mini-version of the whole Sierpinski carpet.
- A recursive definition!



Sierpinski Carpet

```
private void drawSierpinskiCarpet(ColorImage image, int left, int top,
                                   int width, int height, int iterations) {

    if (image == null || width != height || width < 3 || iterations < 1)
             return;

    int size = width /= 3;
    image.drawRectangle(left + 1 * size, top + 1 * size, size, size);
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++) {
            if (i == 1 && j == 1) continue;
            drawSierpinskiCarpet(image, left + j * size, top + i * size,
                                  size, size, iterations - 1);
     }
}
```