Data Structure and Algorithms [CO2003]

Chapter 3 - Recursion

Lecturer: Vuong Ba Thinh
Contact: vbthinh@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Outcomes

- **L.O.8.1** - Describe the basic components of recursive algorithms (functions).
- **L.O.8.2** - Draw trees to illustrate callings and the value of parameters passed to them for recursive algorithms.
- **L.O.8.3** - Give examples for recursive functions written in C/C++.
- **L.O.8.5** - Develop experiment (program) to compare the recursive and the iterative approach.
- **L.O.8.6** - Give examples to relate recursion to backtracking technique.

Recursion and the basic components of recursive algorithms

Definition
Recursion is a repetitive process in which an algorithm calls itself.

- Direct : A $\rightarrow$ A
- Indirect : A $\rightarrow$ B $\rightarrow$ A

Example
Factorial

$$Factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Using recursion:

$$Factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times Factorial(n-1) & \text{if } n > 0 \end{cases}$$

Two main components of a Recursive Algorithm

1. Base case (i.e. stopping case)
2. General case (i.e. recursive case)

Example

Factorial

$$Factorial(n) = \left[\begin{array}{lll} 1 & \text{if } n = 0 & \text{base case} \\ n \times Factorial(n-1) & \text{if } n > 0 & \text{general case} \end{array}\right.$$

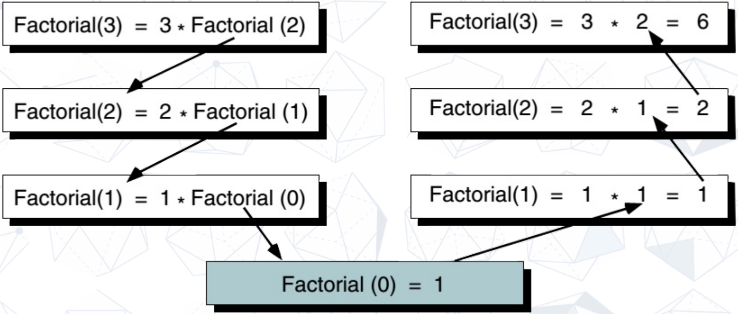


Figure 1: Factorial (3) Recursively (source: Data Structure - A pseudocode Approach with C++)

Factorial: Iterative Solution

Algorithm iterativeFactorial(n)

Calculates the factorial of a number using a loop.

Pre: n is the number to be raised factorially

Post: n! is returned - result in factoN

```
i = 1
factoN = 1
while i <= n do
    factoN = factoN * i
    i = i + 1
end
return factoN
End iterativeFactorial
```

Factorial: Recursive Solution

Algorithm recursiveFactorial(n)

Calculates the factorial of a number using a recursion.

Pre: n is the number to be raised factorially

Post: n! is returned

```
if n = 0 then
    return 1
else
    return n * recursiveFactorial(n-1)
end
End recursiveFactorial
```

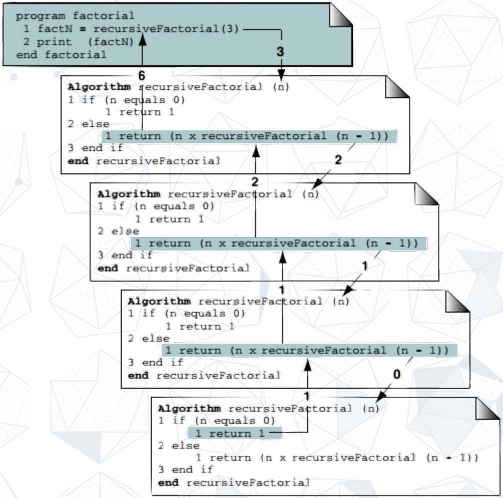


Figure 2: Calling a Recursive Algorithm (source: Data Structure - A

Properties of recursion

Properties of all recursive algorithms

- A recursive algorithm solves the large problem by using its solution to a simpler sub-problem
- Eventually the sub-problem is simple enough that it can be solved without applying the algorithm to it recursively.
  $\rightarrow$ This is called the base case.

Designing recursive algorithms

Every recursive call must either solve a part of the problem or reduce the size of the problem.

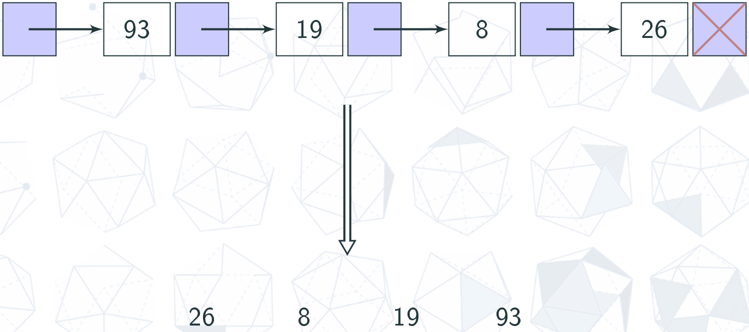
Rules for designing a recursive algorithm

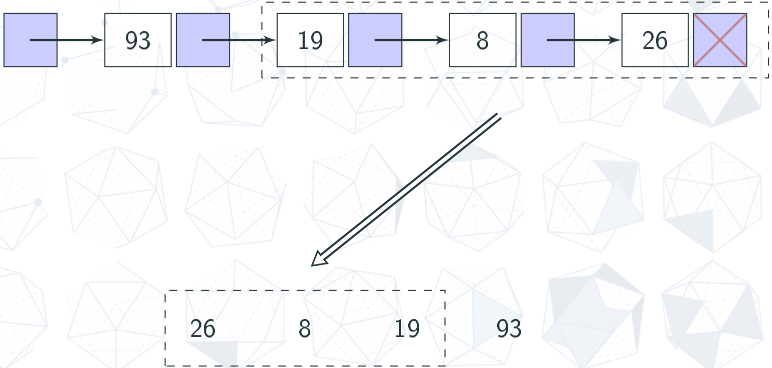
1. Determine the base case (stopping case).
2. Then determine the general case (recursive case).
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion

- A recursive algorithm generally runs more slowly than its nonrecursive implementation.
- BUT, the recursive solution shorter and more understandable.

Print List in Reverse




93
19
8
26
26
8
19
93

Algorithm printReverse(list)
Prints a linked list in reverse.
Pre: list has been built
Post: list printed in reverse

if *list is null* **then**
    return
end
printReverse (list -> next)
print (list -> data)
End printReverse

Definition

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \gcd(b, a \mod b) & \text{otherwise} \end{cases}$$

Example

$\gcd(12, 18) = 6$

$\gcd(5, 20) = 5$

Algorithm gcd(a, b)

Calculates greatest common divisor using the Euclidean algorithm.

Pre: a and b are integers

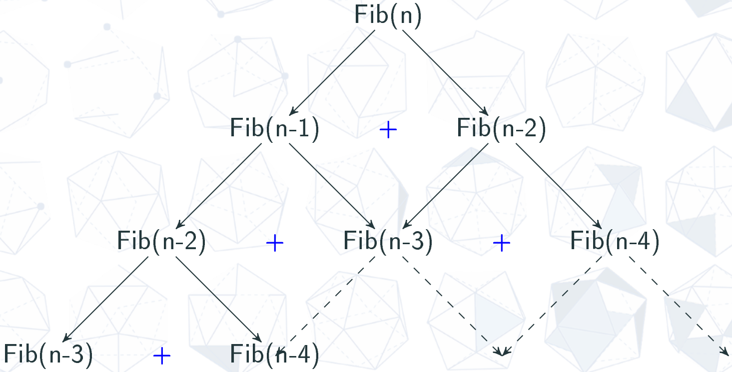
Post: greatest common divisor returned

```
if b = 0 then
    return a
end
if a = 0 then
    return b
end
return gcd(b, a mod b)
End gcd
```

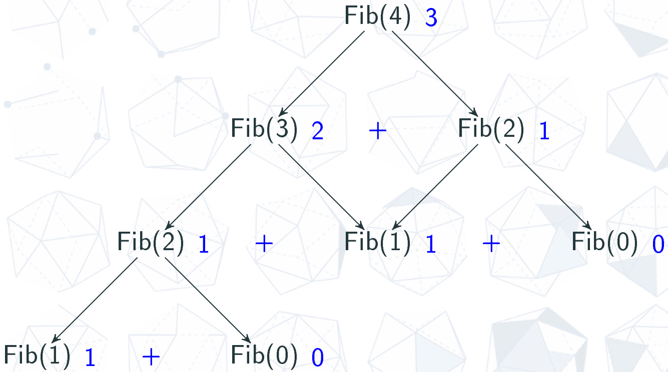
Definition

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

Fibonacci Numbers



Fibonacci Numbers



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

Algorithm fib(n)

Calculates the n^{th} Fibonacci number.

Pre: n is postive integer

Post: the n^{th} Fibonnacci number returned

if $n = 0$ *or* $n = 1$ **then**

  return n

end

return fib(n-1) + fib(n-2)

End fib

No	Calls	Time	No	Calls	Time
1	1	$<$ 1 sec.	11	287	$<$ 1 sec.
2	3	$<$ 1 sec.	12	465	$<$ 1 sec.
3	5	$<$ 1 sec.	13	753	$<$ 1 sec.
4	9	$<$ 1 sec.	14	1,219	$<$ 1 sec.
5	15	$<$ 1 sec.	15	1,973	$<$ 1 sec.
6	25	$<$ 1 sec.	20	21,891	$<$ 1 sec.
7	41	$<$ 1 sec.	25	242,785	1 sec.
8	67	$<$ 1 sec.	30	2,692,573	7 sec.
9	109	$<$ 1 sec.	35	29,860,703	1 min.
10	177	$<$ 1 sec.	40	331,160,281	13 min.

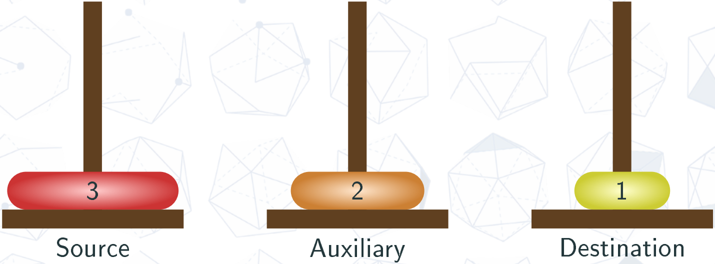
Move disks from Source to Destination using Auxiliary:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. Only one auxiliary needle could be used for the intermediate storage of disks.

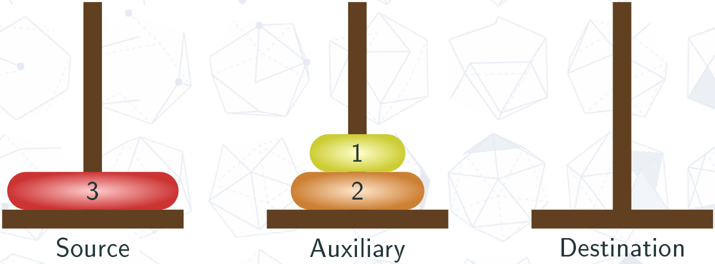
The Towers of Hanoi

Moved disc from pole 1 to pole 3.

The Towers of Hanoi

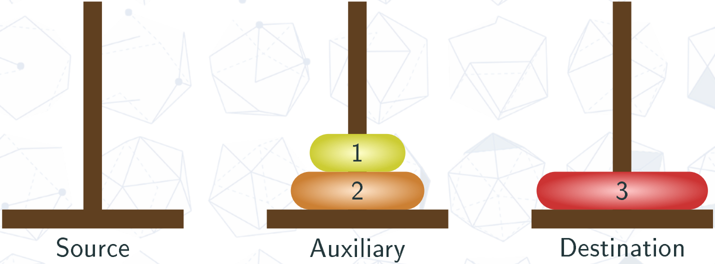


Moved disc from pole 1 to pole 2.

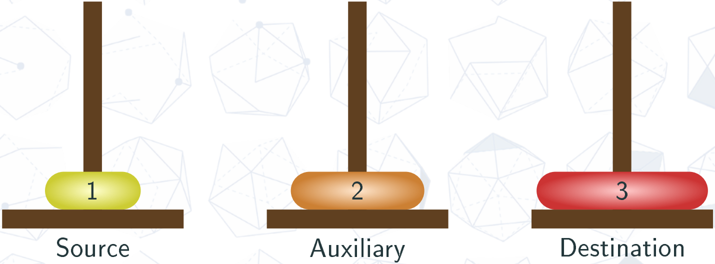


Moved disc from pole 3 to pole 2.

The Towers of Hanoi



Moved disc from pole 1 to pole 3.



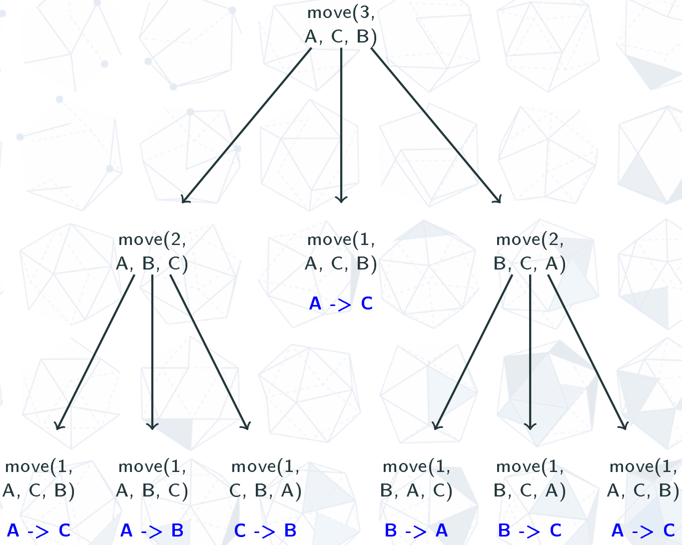
Moved disc from pole 2 to pole 1.

Moved disc from pole 2 to pole 3.

The Towers of Hanoi

Moved disc from pole 1 to pole 3.

The Towers of Hanoi



move(n, A, C, B)

move(n-1, A, B, C) move(1, A, C, B) move(n-1, B, C, A)

Complexity

$$T(n) = 1 + 2T(n-1)$$

Complexity

$T(n) = 1 + 2T(n-1)$

$=> T(n) = 1 + 2 + 2^2 + ... + 2^{n-1}$

$=> T(n) = 2^n - 1$

$=> T(n) = O(2^n)$

- With 64 disks, total number of moves: $2^{64} - 1 \approx 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$
- If one move takes 1s, 2^{64} moves take about 5×10^{11} years (500 billions years).

```
Algorithm move(val disks <integer>, val source <character>, val
  destination <character>, val auxiliary <character>)
Move disks from source to destination.
Pre: disks is the number of disks to be moved
Post: steps for moves printed
print("Towers: ", disks, source, destination, auxiliary)
if disks = 1 then
  | print ("Move from", source, "to", destination)
else
  | move(disks - 1, source, auxiliary, destination)
  | move(1, source, destination, auxiliary)
  | move(disks - 1, auxiliary, destination, source)
end
return
End move
```

Recursion and backtracking

Definition
A process to go back to previous steps to try unexplored alternatives.

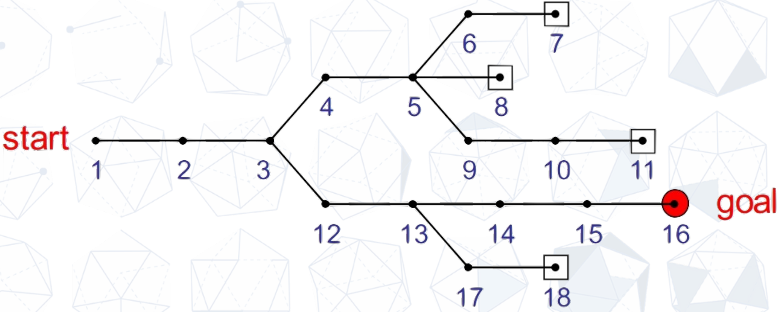
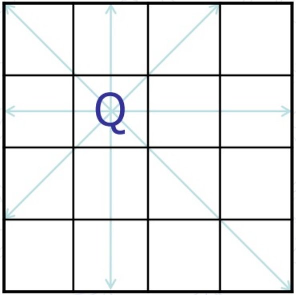
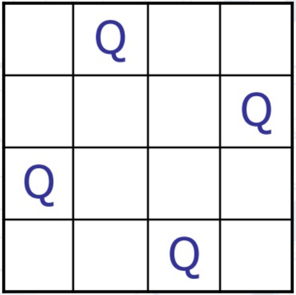


Figure 3: Goal seeking

- Place eight queens on the chess board in such a way that no queen can capture another.





Algorithm putQueen(ref board <array>, val r <integer>)
Place remaining queens safely from a row of a chess board.

Pre: `board` is nxn array representing a chess board
`r` is the row to place queens onwards

Post: all the remaining queens are safely placed on the board; or backtracking to the previous rows is required

```
for every column c on the same row r do
    if cell r,c is safe then
        place the next queen in cell r,c
        if r < n-1 then
            putQueen (board, r + 1)
        else
            output successful placement
        end
        remove the queen from cell r,c
    end
end
return
End putQueen
```