Chapter 5

Recursion

Software Construction– Xuechao Li

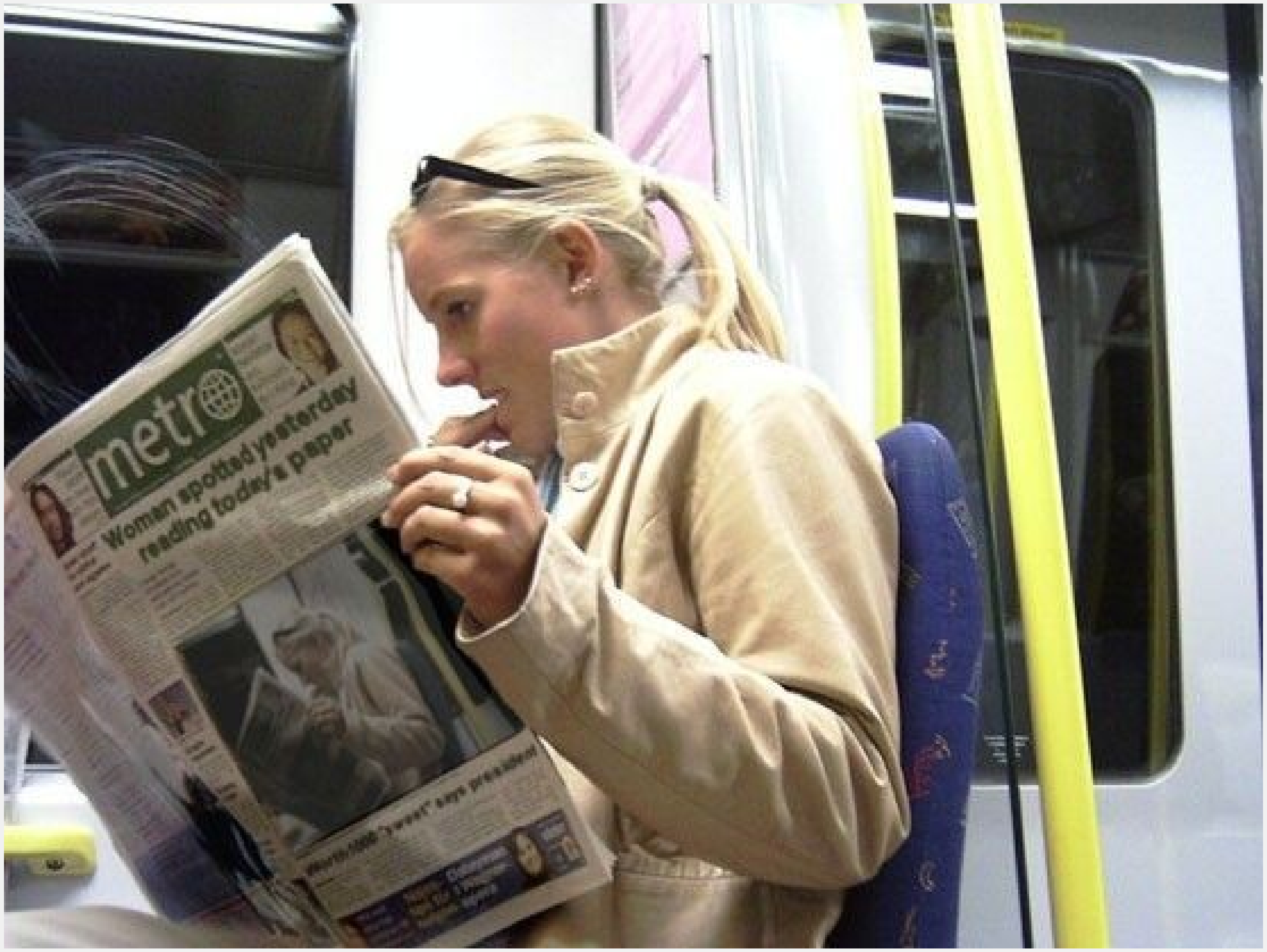
AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Recursive Pictures

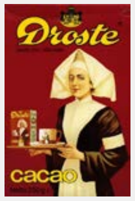
Recursive Pictures

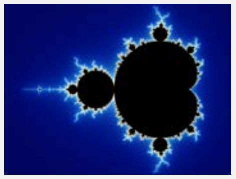


Recursive Pictures

Necessary properties of recursion

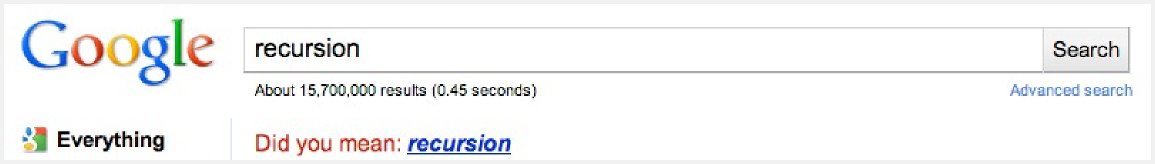
Recursive things are self--referential





Recursive things must not be circular

recursion |riˈkər zh ən|
noun Mathematics & Linguistics
ORIGIN 1930s: from late Latin recursio(n--), from recurrere
(see recursion).



recursion

About 15,700,000 results (0.45 seconds)

Did you mean: *recursion*

Recursive things must have meaning

This sentence is false.

Structural Recursion

Structural recursion: The structure being defined appears as part of the definition.

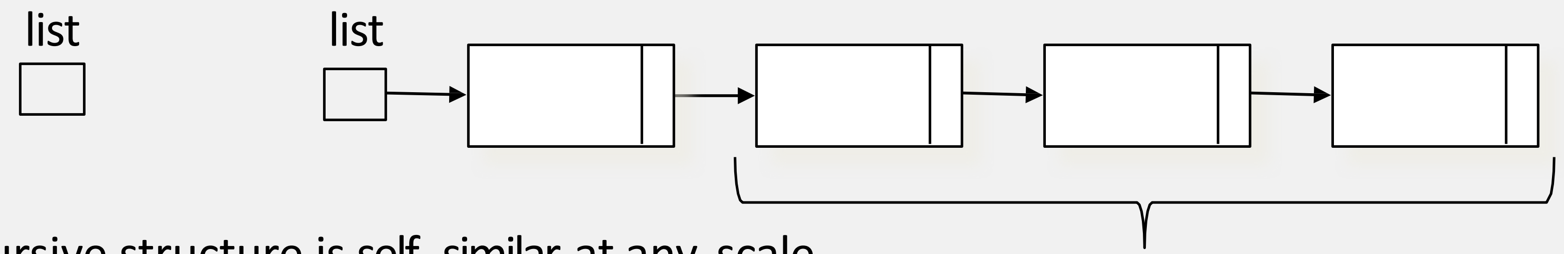
```
public class Node
{
   private   T element;
   private   Node next;
   …
}
```

What is a Node?

A node is a structure that contains an element and a node.

A higher level example:

A linked list is either empty or it is a single node that points to a linked list.



A recursive structure is self-similar at any scale.

Computational Recursion

Computational recursion: The computation being defined is applied within its own definition.

Example: The factorial function.

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

How is 5! computed?

5! is computed by multiplying 5 by 4!

Why is this not circular?

Evaluating the function:

$$\begin{tikzcd}
5! = 5 * 4! \\
4! = 4 * 3! \arrow[u, bend right, "24"'] \\
3! = 3 * 2! \arrow[u, bend right, "6"'] \\
2! = 2 * 1! \arrow[u, bend right, "2"'] \\
1! = 1 * 0! \arrow[u, bend right, "1"'] \\
0! = 1 \arrow[u, bend right, "1"']
\end{tikzcd}$$

5! = 120

Computational Recursion

Computational recursion: The computation being defined is applied within its own definition.

Example: The factorial function.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \, X(n - 1)! & \text{if } n > 0 \end{cases}$$

Evaluating the function:

```
5! = 5 * 4!                                  5! = 5 * 4!
   = 5 * (4 * 3!)                                     4 * 3!
   = 5 * (4 * (3 * 2!))                                   3 * 2!
   = 5 * (4 * (3 * (2 * 1!)))                                 2 * 1!
   = 5 * (4 * (3 * (2 * (1 * 0!))))                               1 * 0!
   = 5 * (4 * (3 * (2 * (1 * 1))))                                    1
   = 5 * (4 * (3 * (2 * 1)))                                      1
   = 5 * (4 * (3 * 2))                                        2
   = 5 * (4 * 6)                                          6
   = 5 * 24                                           24
   = 120                                          120
```

Computational Recursion

Computational recursion: The computation being defined is applied within its own definition.

Example: The factorial function.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \; X \; (n - 1)! & \text{if } n > 0 \end{cases}$$

Evaluating the function:

```
5! = 5 * 4!
   = 5 * (4 * 3!)
   = 5 * (4 * (3 * 2!))
   = 5  * (4 * (3 * (2 * 1!)))
   = 5  * (4 * (3 * (2 * (1 * 0!))))
   = 5  * (4 * (3 * (2 * (1 * 1))))
   = 5  * (4 * (3 * (2 * 1)))
   = 5  * (4 * (3 * 2))
   = 5  * (4 * 6)
   = 5 * 24
   = 120
```

Recursive
“down trip”

Recursive
“up trip”

```
5! = 5 * 4!
         4 * 3!
             3 * 2!
                 2 * 1!
                     1 * 0!
                         1
                     1
                 2
             6
         24
     120
```

Recursive
“down trip”

Recursive
“up trip”

Computational Recursion

Computational recursion: The computation being defined is applied within its own definition.

Example: The factorial function.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \, X (n - 1)! & \text{if } n > 0 \end{cases}$$

An implementation in C++:

```
int factorial(int n)
{
    if(n == 0)
         return 1;
    else
          return n * factorial(n - 1);

}
```

Recursive Definitions

A recursive definition, whether structural or computational, consists of two parts:

(1) Base Case

A simple statement or definition that does not involve recursion

(2) Reduction Step

A set of rules that reduce all other cases toward the base case

Examples:

A linked list is either **empty** or it is **a single node that points to a linked list**.

$$n! = \begin{cases} \mathbf{1} & \textbf{if } \mathbf{n = 0} \\ \mathbf{n * (n - 1)!} & \textbf{if } \mathbf{n > 0} \end{cases}$$

Writing recursive methods

The basic recursive code pattern is this:

```
if (base case) {
   return solution;
}
else {
   return recursive_call;
}
```

Describes the smallest instance of the problem.

Known solution or easy to compute.

States the solution in terms of a smaller instance of the problem.

Must take computation closer to the base case.

Factorial is a good example of this pattern.

```
 int factorial(int n) {

    if (n == 0)

       return 1;

    else

       return n * factorial(n-1);
}
```

Smallest instance of the problem.

Known solution.

States the solution of the current problem – factorial(n) – in terms of a smaller instance of the problem: factorial(n--1).

n–1takes the computation closer to the base case (0).

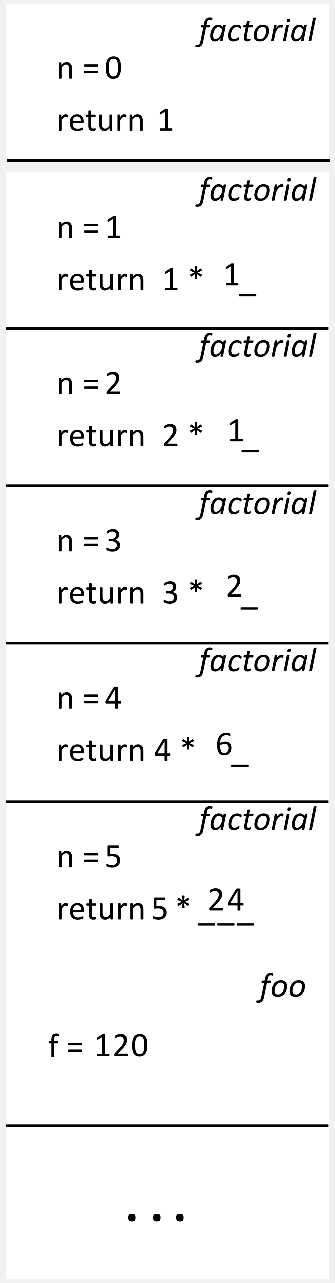
Calling recursive methods

When a method is called – recursive or not – an activation record (or stack frame) for that method is pushed onto the runtime stack (call stack).

When a method returns – recursive or not – its activation is popped from the call stack

```
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

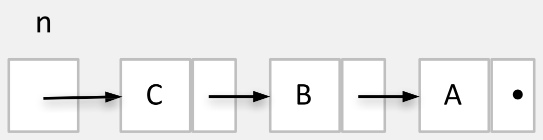
```
 void foo()  {
   int f = factorial(5);
}
```



Writing recursive methods

Write a recursive method that calculates the length of a linked list.

(That is, the number of nodes accessible from the first node in a pointer chain.)



`length(n)` would return 3

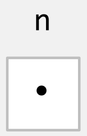
Remember our iterative version:

```
 int length(Node n) {
    Node p = n;
    int i = 0;
    while (p != null) {
       i++;
       p = p.getNext();
    }
    return i;
}
```

Thinking recursively…

Base Case

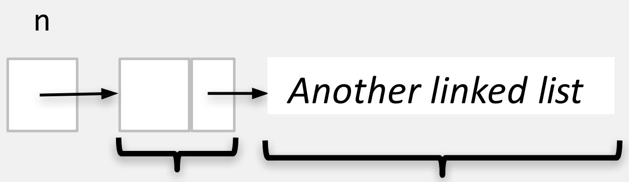
Empty



Length = 0

Reduction Step (Recursive Case)

A single node that points to a linked list.



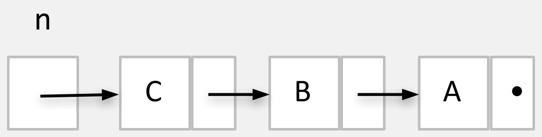
Length = 1 + length of this list

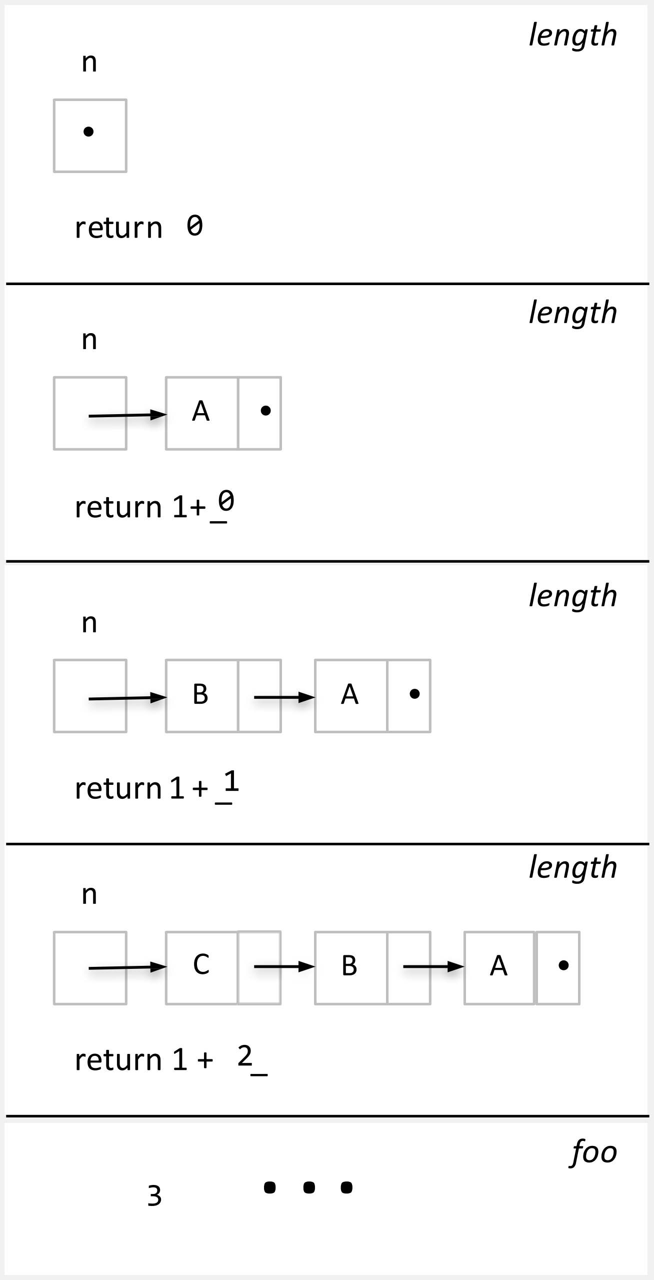
```
int length(Node n) {
    if ( n == null ) {
      return 0;
   }
   else {
      return 1 + length(n.getNext());
   }
}
```

Calling recursive methods

```
  int length(Node n) {
   if (  n == null ) {
      return 0;
   }
   else {
      return 1 + length(n.getNext());
   }
}
```

```
 void foo()  {
   Node n = new Node("A");
   n = new Node("B",  n);
   n = new Node("C", n);
   int len = length(n);
 }
```





Variation on the pattern

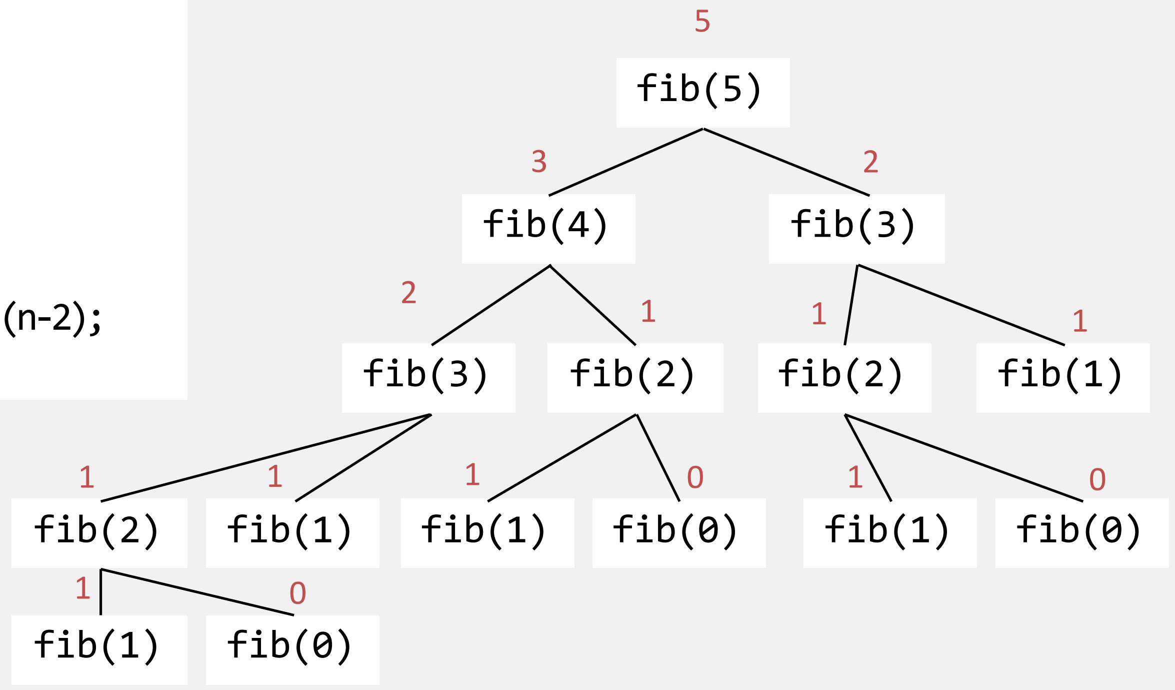
The Fibonacci numbers:

An integer sequence in which the first two numbers are 0 and 1, and each subsequent number is the sum of the previous two.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Finding the nth Fibonacci number:

```
int fib(int n) {
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fib (n-1)  + fib (n-2);
}
```



More than one base case.

More than one recursive call in the reduction step.

The power of recursive thinking

You've been hired to develop control software for a robotic lift in a warehouse. The robotic lift must move stacks of widgets from the incoming shipping bay to a long--term storage bay. The widgets are stacked one on top of another, from largest on the bottom to smallest on the top. The robotic lift can only move one widget at a time due to weight, and can never stack a larger widget on top of a smaller widget. There is one temporary storage bay that the robotic lift can use while moving the widgets from the shipping bay to the long--term storage bay. At all times each storage bay can hold at most one stack of widgets, and the stacks must be arranged from largest on bottom to smallest on top. The transfer of a single widget from the stack in one bay to the stack in another bay (shipping, temporary, or long--term) is considered a "move." You must write the control software so that the robotic lift makes the minimum number of widget moves possible.

Game: https://www.coolmathgames.com/0-tower-of-hanoi

Https://www.mindgames.com/game/Tower+of+Hanoi

Video: https://www.youtube.com/watch?v=5QuiCcZKyYU

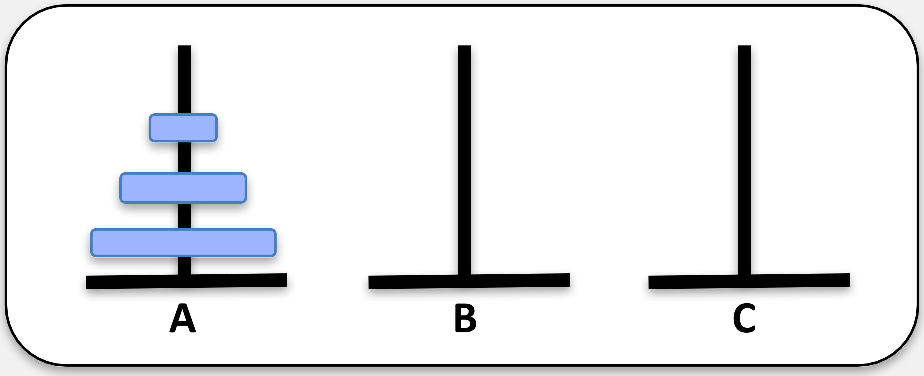
Robot: https://www.youtube.com/watch?v=h8JPtWpctl4

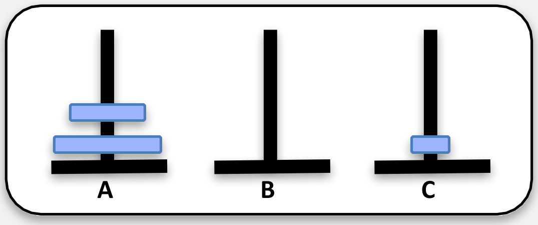
The power of recursive thinking

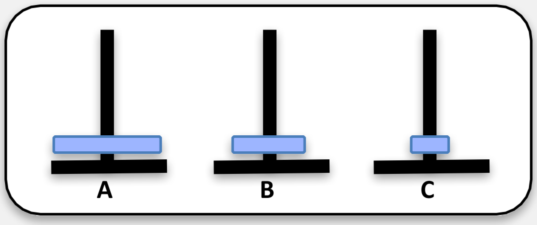
Tower of Hanoi (or Tower of Brahma)

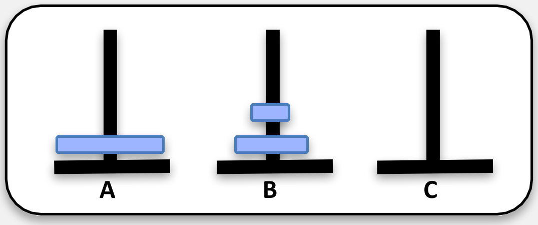
Given three pegs (A, B, and C) and N disks, with the initial configuration of all N disks stacked from largest to smallest on peg A, move all N disks from peg A to peg C while obeying the following rules.

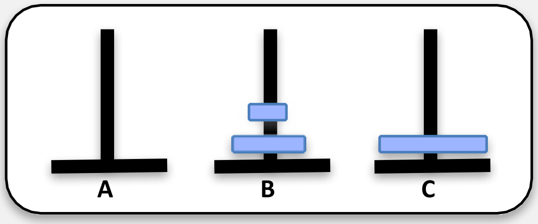
1. Only the top disk on a stack can be moved.
2. Only one disk can be moved at a time.
3. No disk can be placed on top of a smaller disk.

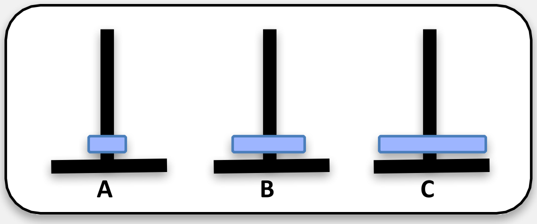


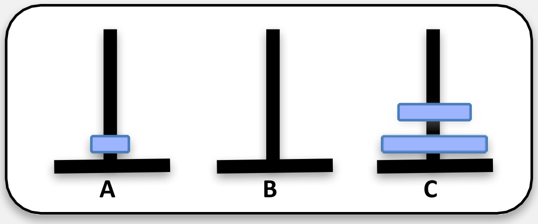


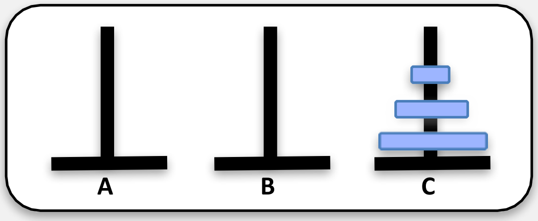








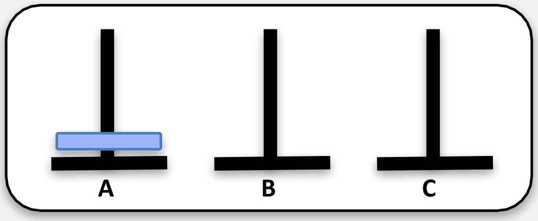




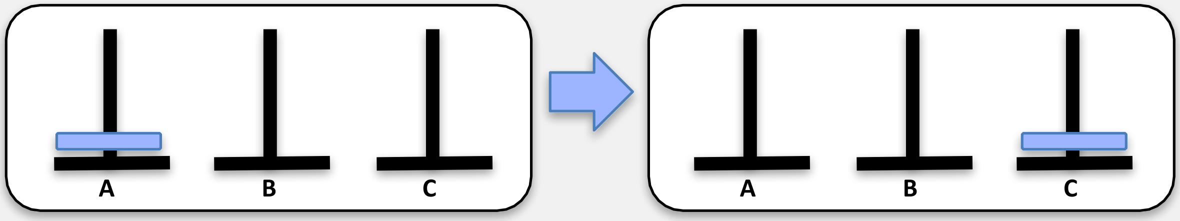
An optimal solution is easy to discover by trial and error for small N, but the trick is to write an algorithm that makes the optimal sequence of moves for arbitrary values of N.

Tower of Hanoi

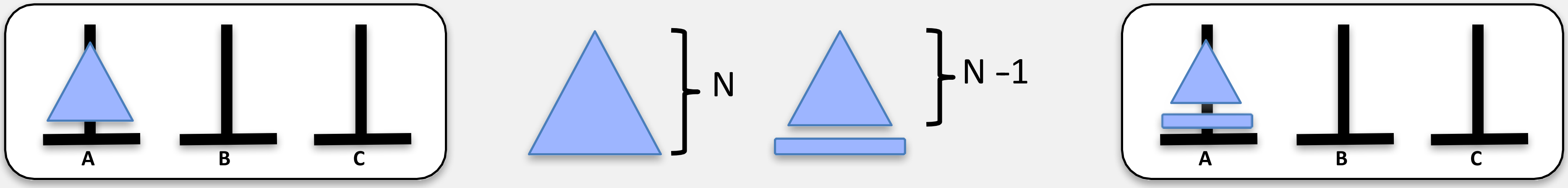
Base case: One disk.



Solution: Move the disk from A to C.



Reduction step: N disks. *The key insight is in how we view these N disks.*

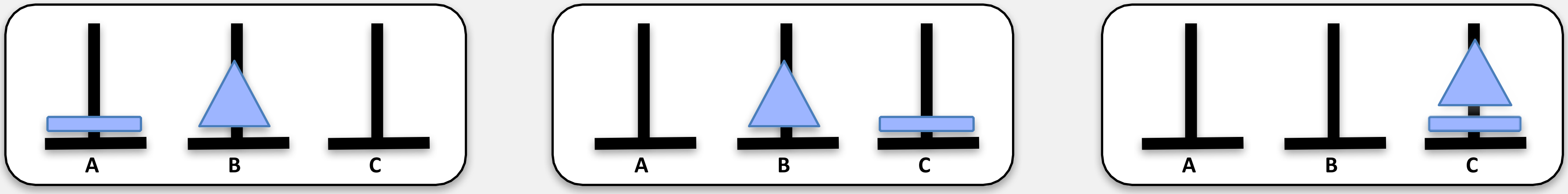


Solution:

(1) Move N–1 disks from A to B.

(2) Move 1 disk from A to C.

(3) Move N–1 disks from B to C.



The code is just this simple.

Tower of Hanoi

```
void moveTower(int numDisks, String startPeg, String endPeg, String tempPeg)
{
   if (numDisks == 1)
      moveOneDisk(startPeg,  endPeg);
   else {
      moveTower (numDisks-1, startPeg, tempPeg, endPeg);
      moveOneDisk(startPeg,  endPeg);
      moveTower (numDisks-1,  tempPeg, endPeg,  startPeg);
   }
}
```

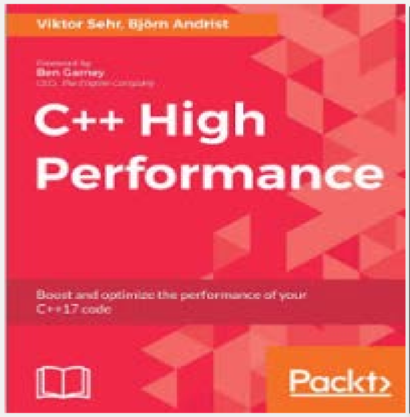
```
void moveOneDisk(String startPeg, String  endPeg) {
  cout <<“Move one disk from " << startPeg << “ to “ <<  endPeg << endl;
}
```

```
Move one disk from A to C
Move one disk from A to B
Move one disk from C to B
Move one disk from A to C
Move one disk from B to A
Move one disk from B to C
Move one disk from A to C
```

Performance issues

Recursive code can be less efficient that an equivalent iterative version because of overhead costs.

Primary costs associated with recursion: (1) Method call/return overhead (2) Call stack space


Viktor Sehr, Björn Andrist
C++ High Performance
Boost and optimize the performance of your C++17 code
Packt

Generally, the advice for dealing with methods that are naturally recursive (because that is the natural way to code them for clarity) is to go ahead with the recursive solution. You need to spend time counting the cost (if any) only when your profiling shows that this particular method is a bottleneck in the application. At that stage, it is worth pursuing alternative implementations or avoiding the method call completely with a different structure. – Jack Shirazi

Mechanics

1. Determine the base case of the Recursion
2. Implement a loop that will iterate until the base case is reached
3. Make a progress towards the base case

See samples in Practice

Structure of recursive methods

The intuitive “if–else” template:

```
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

The else is often omitted since it’s unnecessary:

```
int factorial(int n) {
   if (n == 0)
      return 1;
   return n * factorial(n-1);
}
```

Structure of recursive methods

Note when different parts of the method are executed:

```
  static void foo(int n)  {

      // recursive down trip actions go here

      if (n == 0) {
      // base case
         return;
      }
      else {
      // reduction
         foo(n-1);
      }

      // recursive up trip actions go  here

   }
```

Summary

1. Recursion definition
    - base case
    - reduction step

2. A step-by-step analysis
    - the factorial function
    - the length of a singly linked list

3. A real world application
    - Tower of Hanoi

4. Performance/Structure of recursive methods