Recursion

COMP 2210 – Dr. Hendrix

AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

GNU

GNU's Not UNIX

recursive acronym - A hackish (and especially MIT) tradition is to choose acronyms and abbreviations that refer humorously to themselves or to other acronyms or abbreviations. The classic examples were two MIT editors called EINE ("EINE Is Not Emacs") and ZWEI ("ZWEI Was EINE Initially"). More recently, there is a Scheme compiler called LIAR (Liar Imitates Apply Recursively), and GNU stands for "GNU's Not Unix!" - and a company with the name CYGNUS, which expands to "Cygnus, Your GNU Support".

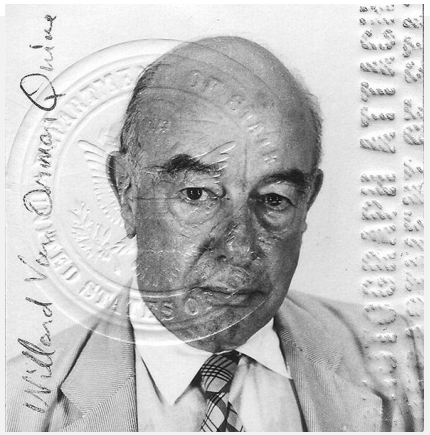
http://encyclopedia2.thefreedictionary.com/Self-referential+acronym

What is the output of this program? **Itself!**

```
public class Quine {
public static void main(String[] args) {
String[] str = {
"public class Quine {",
" public static void main(String[] args) {",
"  String[] str = {",
"  };",
"  for(int i=0;i<3;i++)System.out.println(str[i]);",
"  for(int i=0;i<9;i++)System.out.println((char)34+str[i]+(char)34+',');",
"  for(int i=3;i<9;i++)System.out.println(str[i]);",
" }",
"}",
};
for(int i=0;i<3;i++)System.out.println(str[i]);
for(int i=0;i<9;i++)System.out.println((char)34+str[i]+(char)34+',');
for(int i=3;i<9;i++)System.out.println(str[i]);
}
}
```

Quine

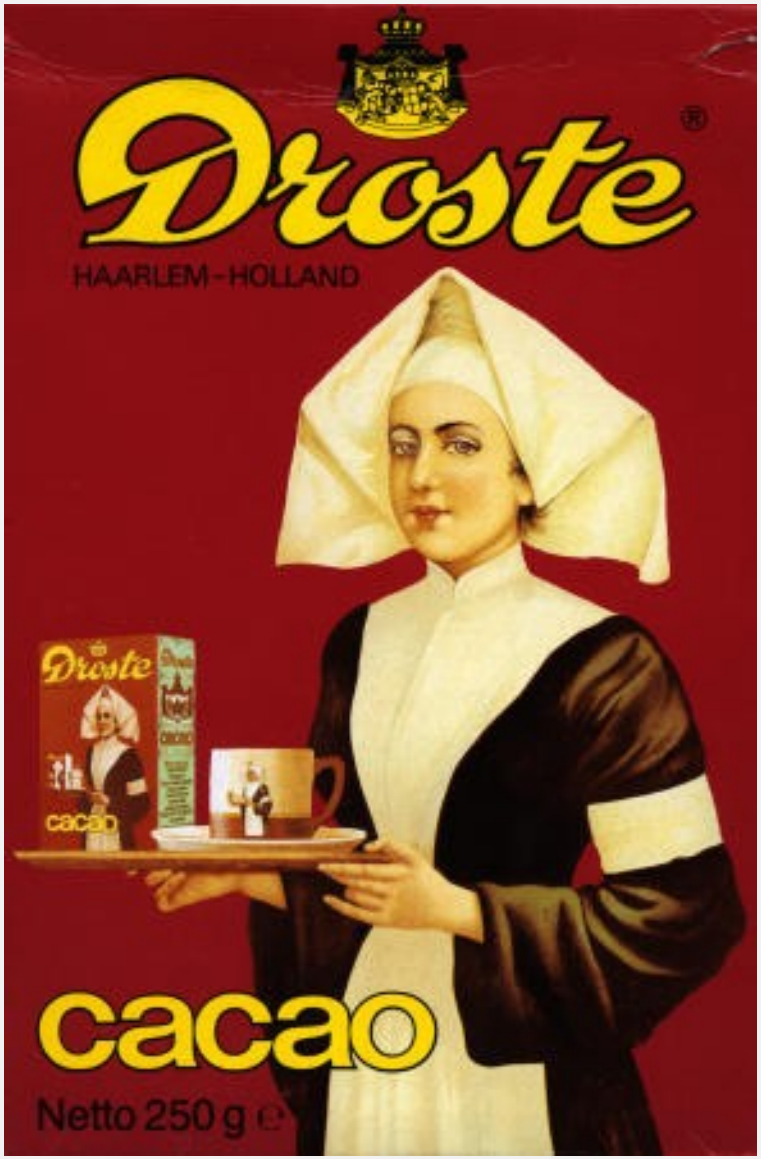
Indirect self-reference



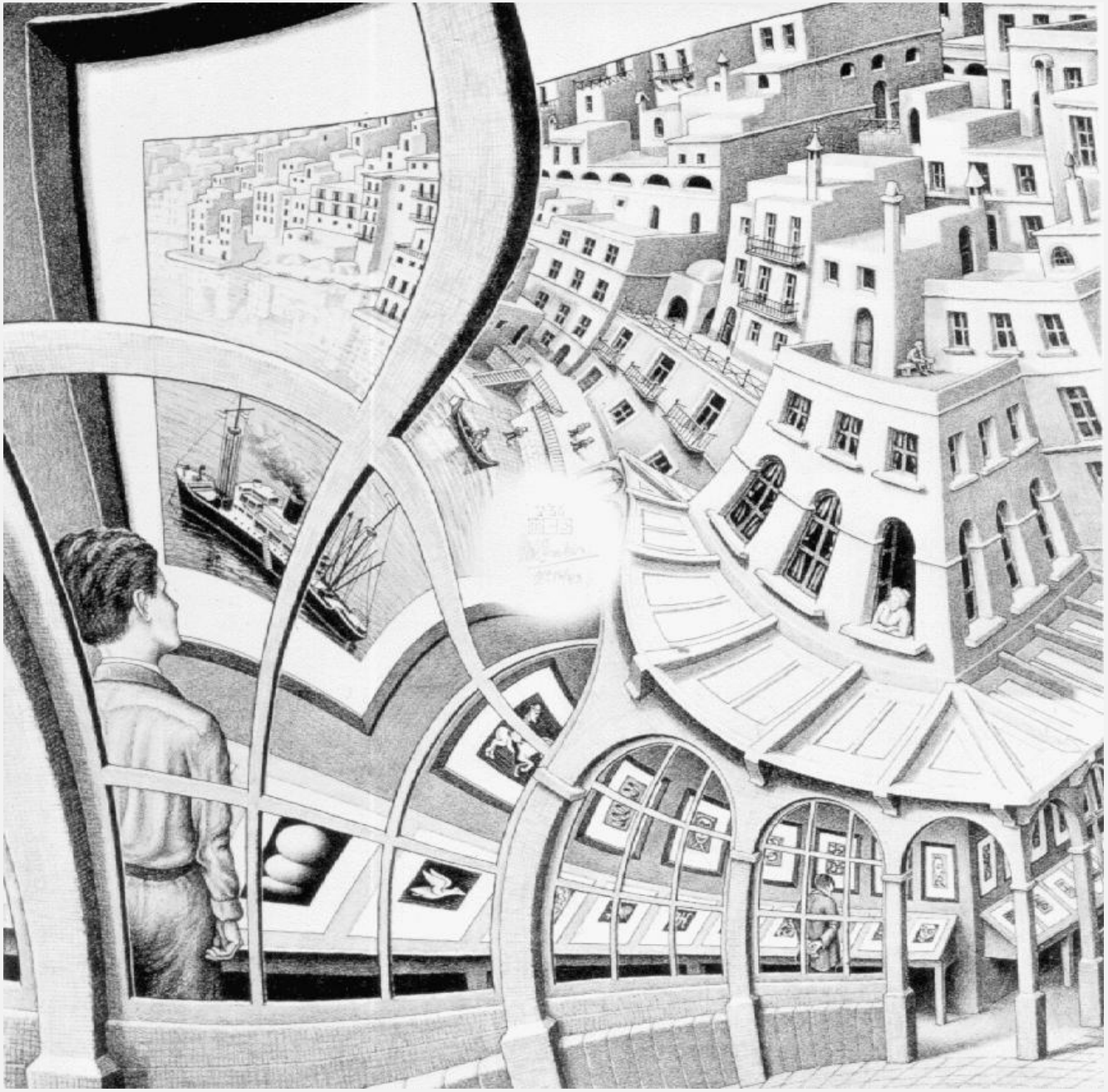
"quoted and followed by itself is a quine." quoted and followed by itself is a quine.

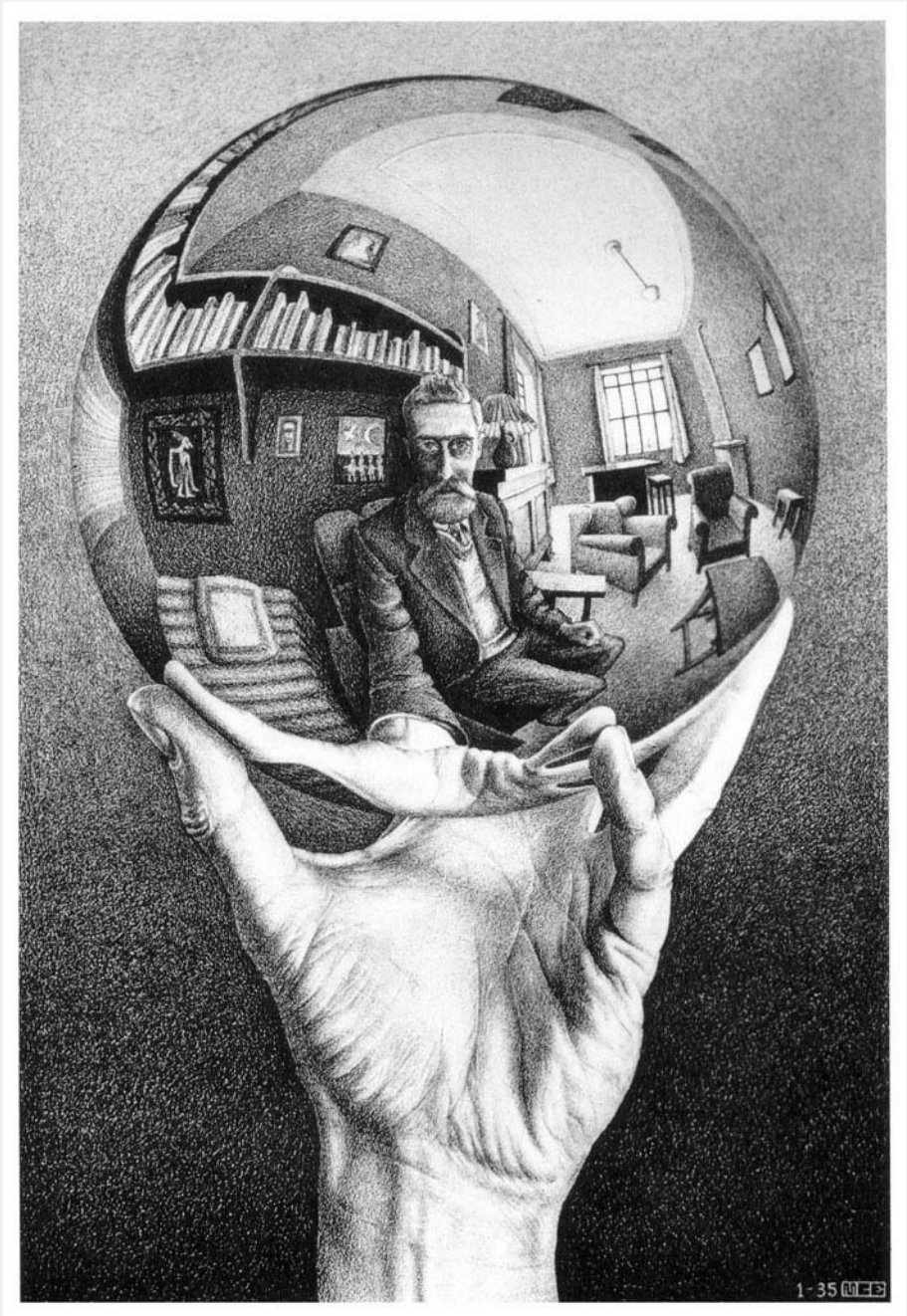

PINK
FLOYD
GiGi
EMI
HARVEST
PINK
FLOYD


metro
Woman spotted yesterday reading today's paper

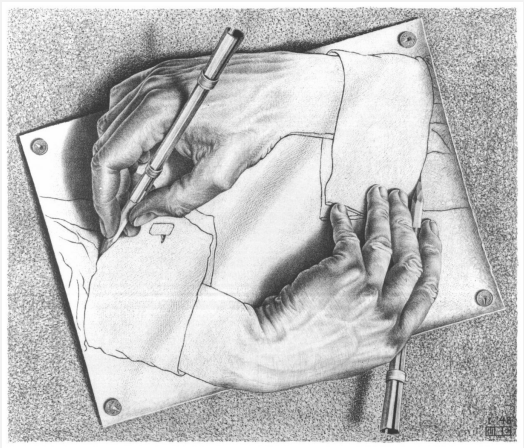


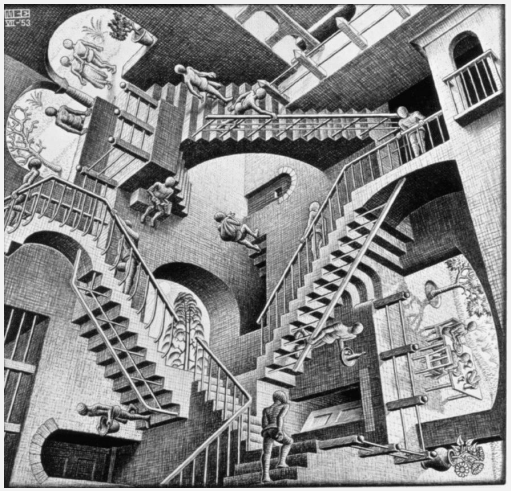
Droste Effect

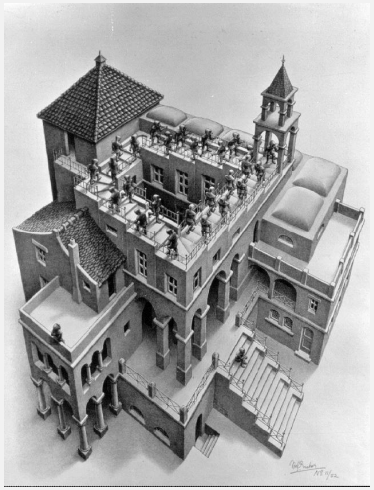


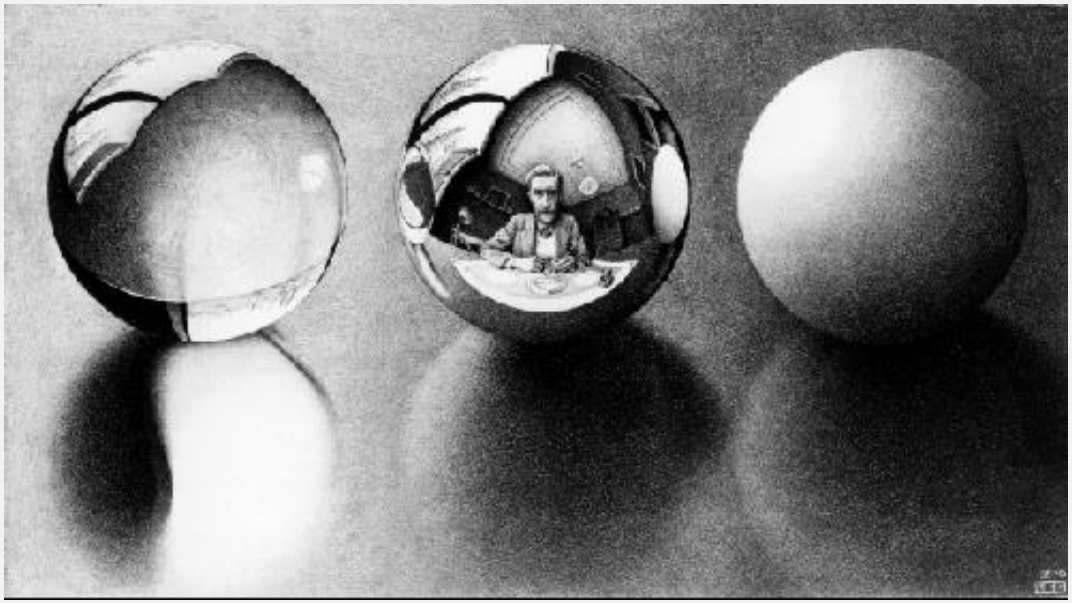


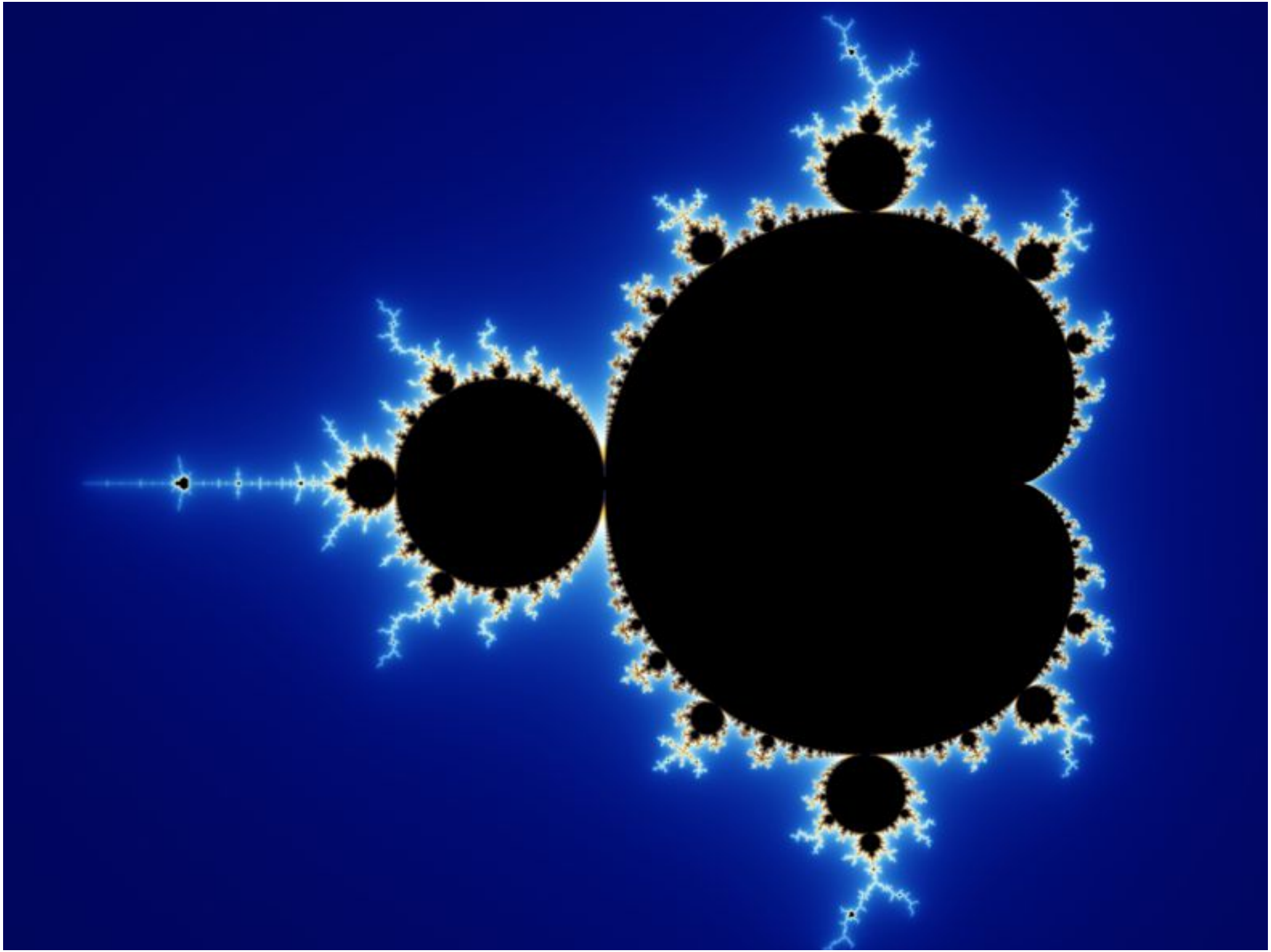
M. C. Escher





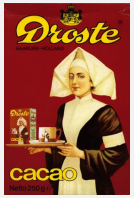


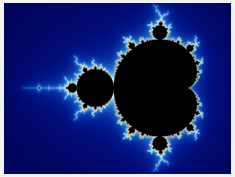




Necessary properties of recursion

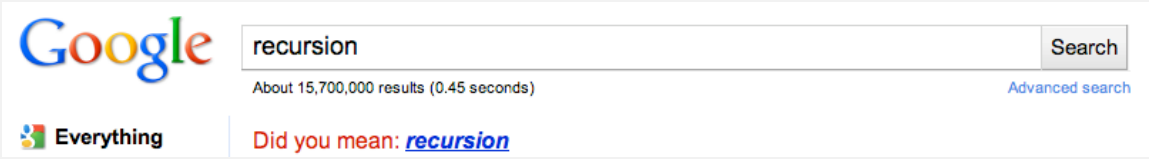
Recursive things are self-referential





Recursive things must not be circular

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



Recursive things must have meaning

This sentence is false.

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

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

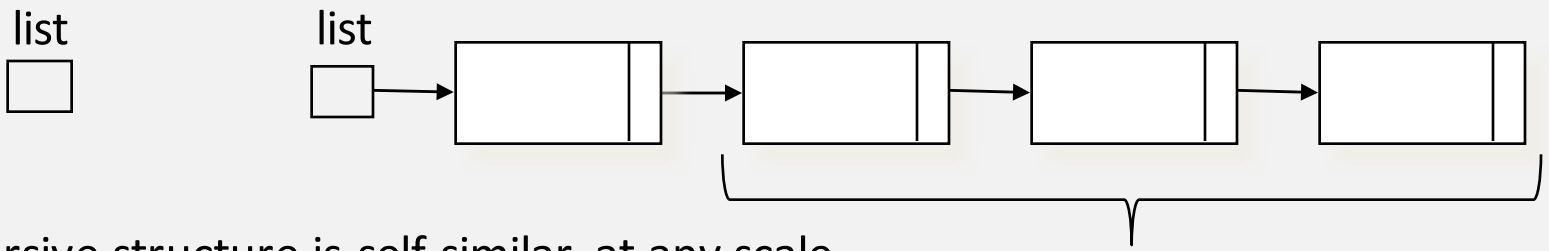
What is a Node?

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

Why is this not circular?

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

Example: The factorial function.

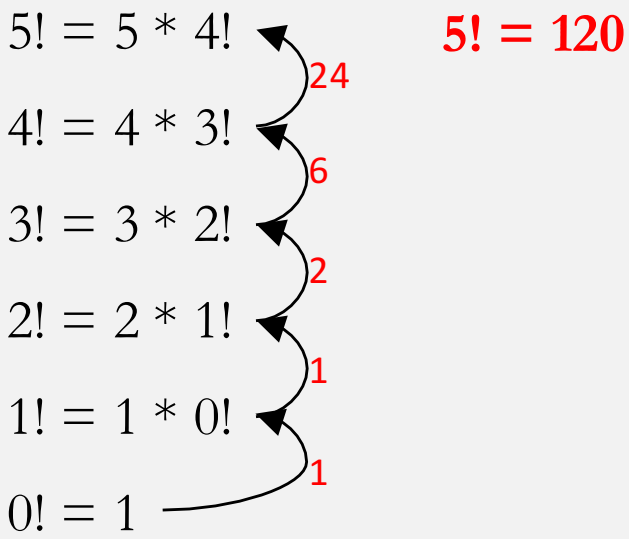
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

How is 5! computed?

5! is computed by multiplying 5 by 4!

Why is this not circular?

Evaluating the function:



5! = 120

Computational Recursion

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

Example: The factorial function.

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

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
```

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

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

Example: The factorial function.

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

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.

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

An implementation in Java:

```
public 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

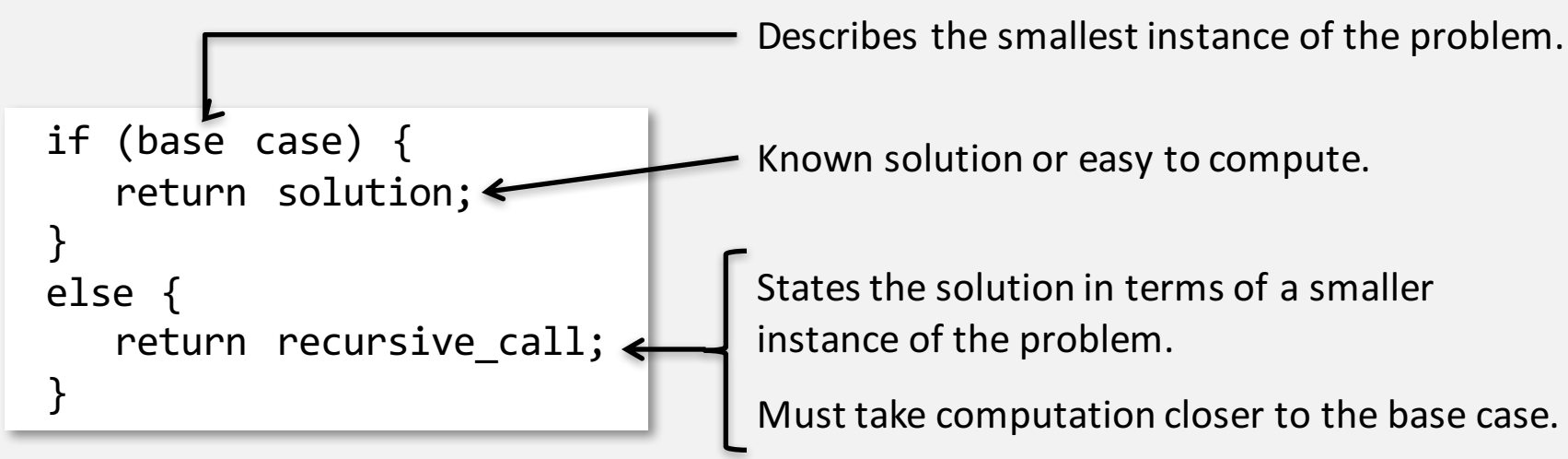
Examples:

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

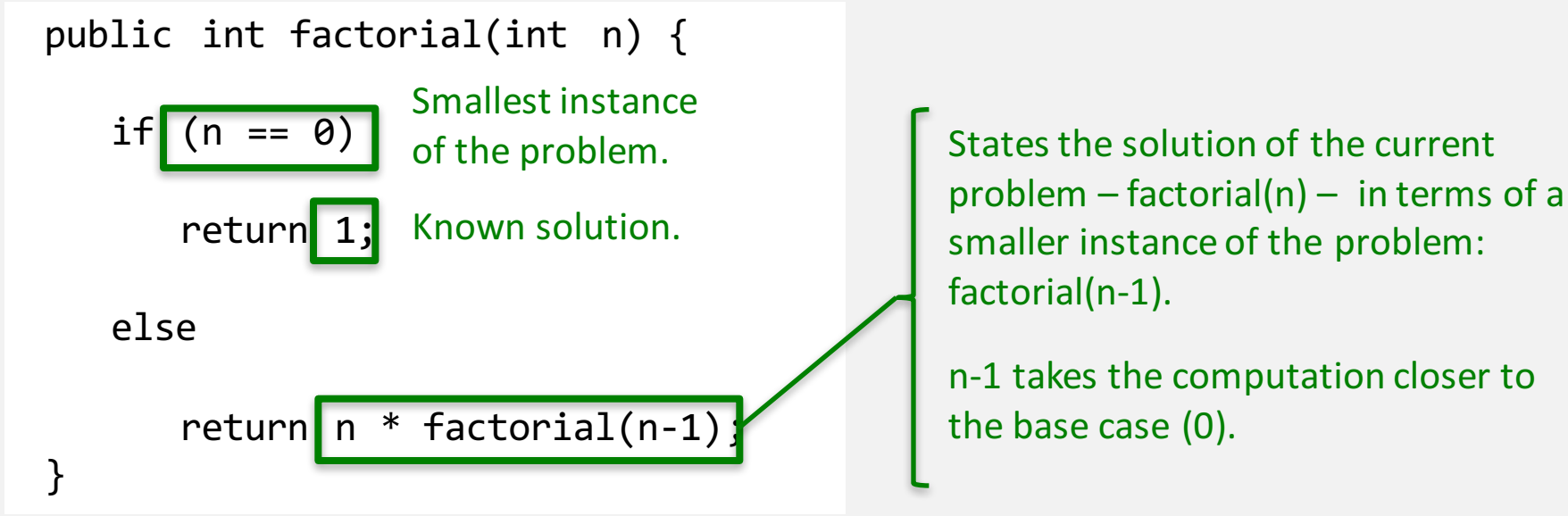
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n - 1)! & \text{if } n > 0 \end{cases}$$

Writing recursive methods

The basic recursive code pattern is this:



Factorial is a good example of this pattern.



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).

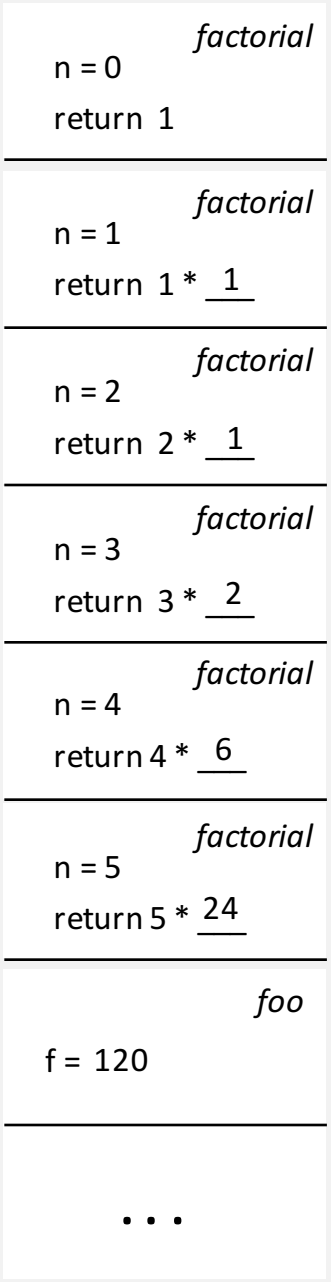
When a method returns – recursive or not – its activation is popped from the call stack

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

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

Time complexity? **O(N)**

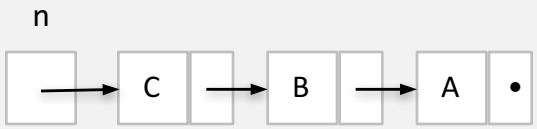
Fun for 3270.



Writing recursive methods

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

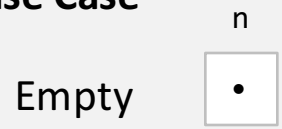
(That is, the number of nodes accessible from the first node in a pointer chain.)



`length(n)` would return 3

Thinking recursively …

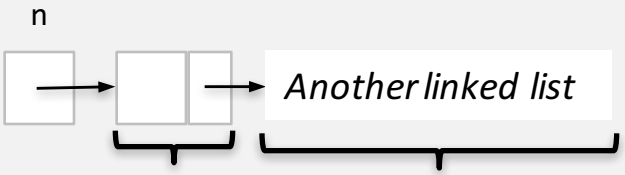
Base Case



Empty Length = 0

Reduction Step (Recursive Case)

A single node that points to a linked list.



Length = 1 + length of this list

Remember our iterative version:

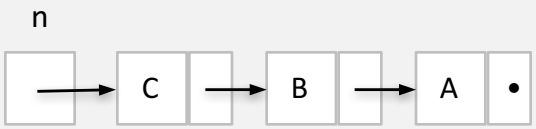
```
public int length(Node  n) {
    Node p = n;
    int i = 0;
    while (p != null) {
       i++;
       p = p.getNext();
    }
    return i;
}
```

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

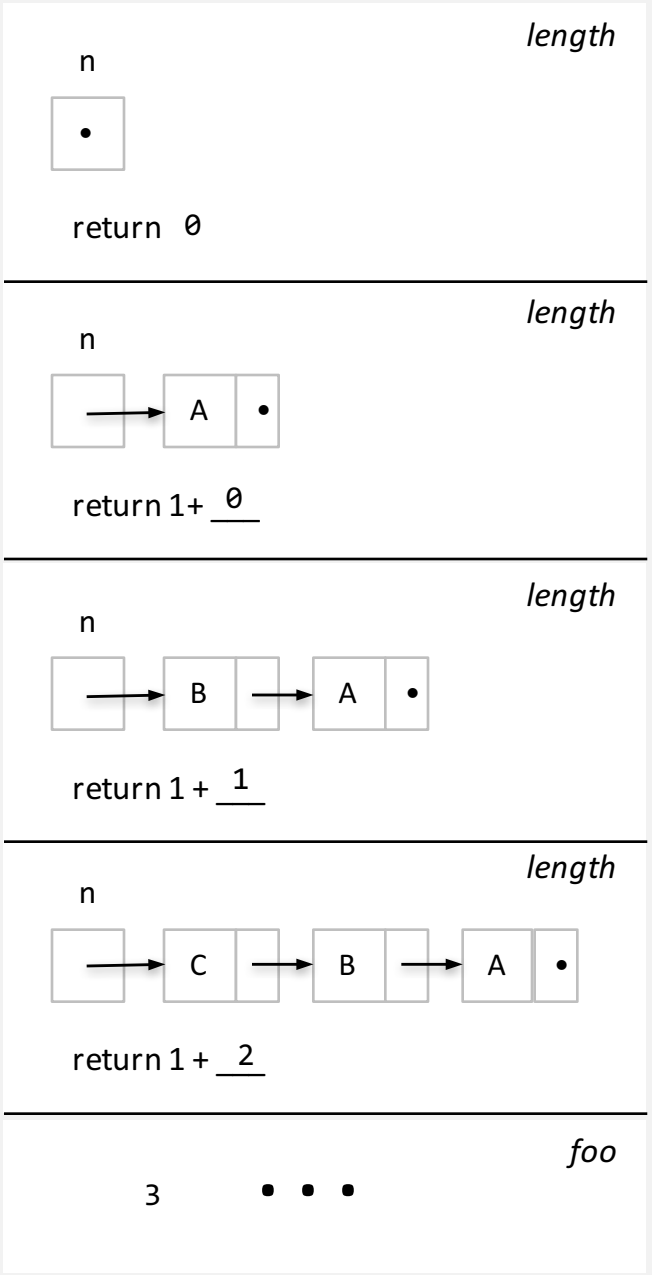
Calling recursive methods

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

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



Time complexity? O(N)



Code structure based on recursive structure

One of the benefits of recursion is that it focuses our thinking very clearly on specific cases of the problem.

Our thinking, and the resulting code structure, is based directly on the recursive structure of the object we're dealing with.

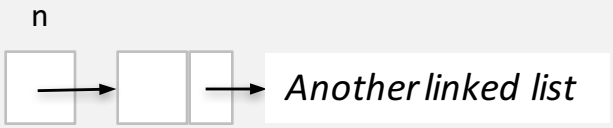
Example: Linear search on a linked list. **Time complexity? O(N)**

Base Case

Empty

Reduction Step (Recursive Case)

A single node that points to a linked list.




```
public boolean contains(Node  n, Object target)
{
          n == null
   if (               ) {
      return false;


   }
   else {
      if (n.getElement().equals(target))
         return true;
      else
         return contains(n.getNext(),  target);



   }
}
```

Variation on the pattern

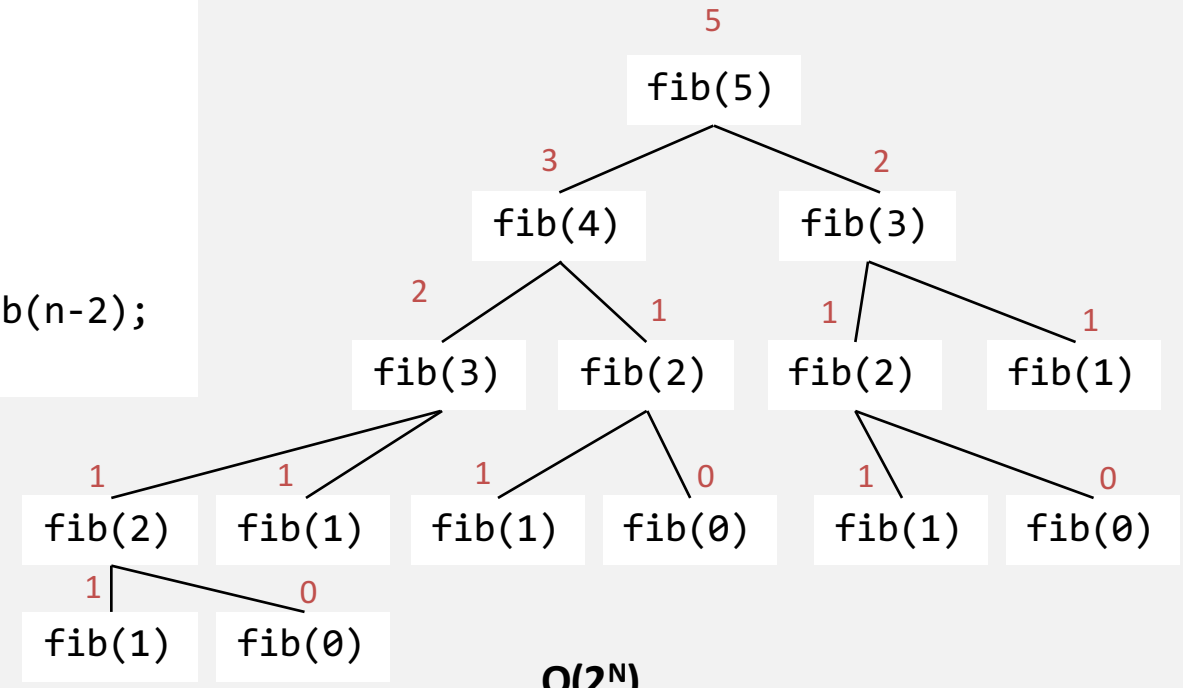
The Fibonacci numbers:

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:

```
public 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.

Time complexity?

Interesting, insightful, and fun for 3270.

$O(2^N)$

Technically correct but you'll do better in 3270.

The power of recursive thinking

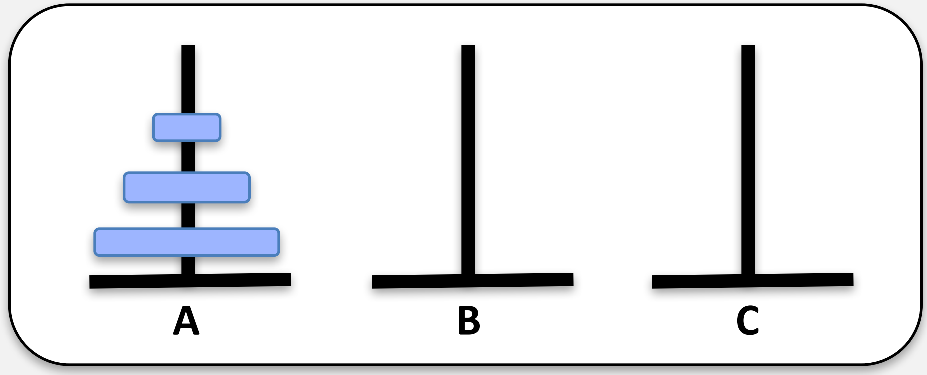
You've been hired to develop control software for a robotic lift in a warehouse. The robotic lift must move stacks of widgets from the incoming shipping bay to a long-term storage bay. The widgets are stacked one on top of another, from largest on the bottom to smallest on the top. The robotic lift can 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.

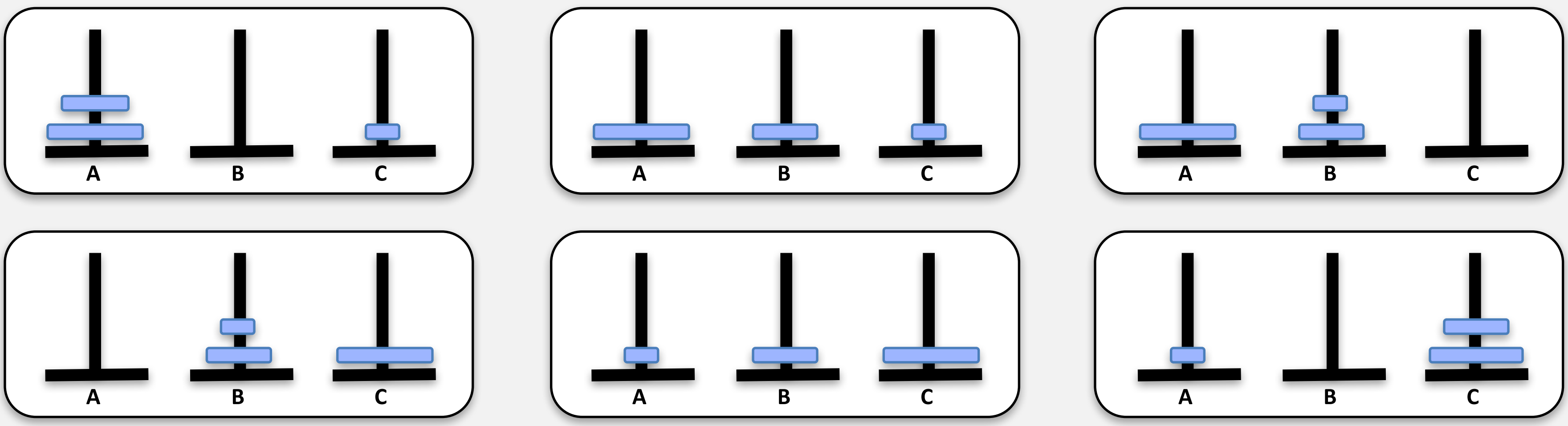
The power of recursive thinking

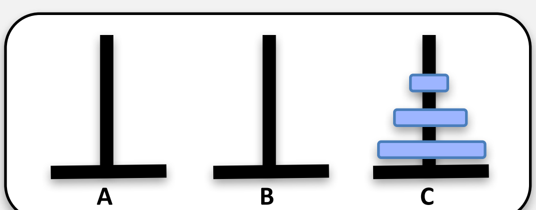
Tower of Hanoi (or Tower of Brahma)

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.



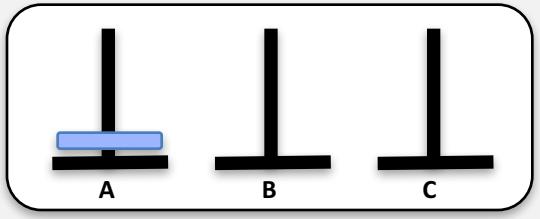




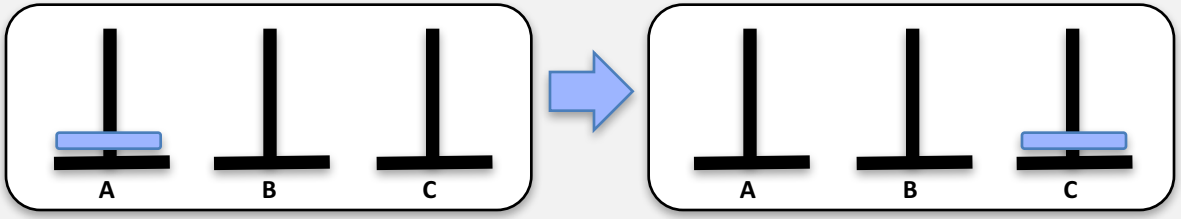
An optimal solution is easy to discover by trial and error for small N, but the trick is to write an algorithm that makes the optimal sequence of moves for arbitrary values of N.

Tower of Hanoi

Base case: One disk.

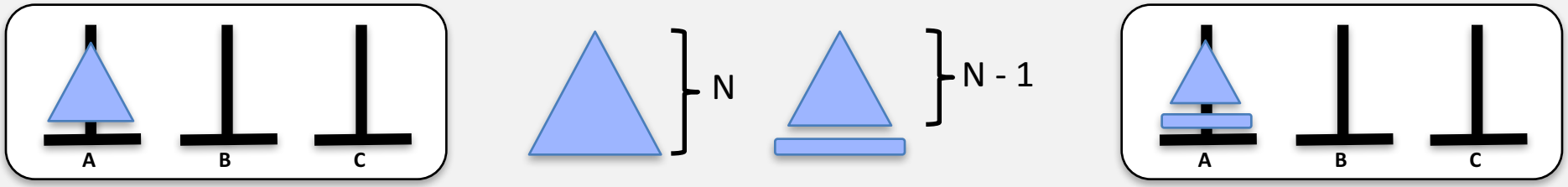


Solution: Move the disk from A to C.



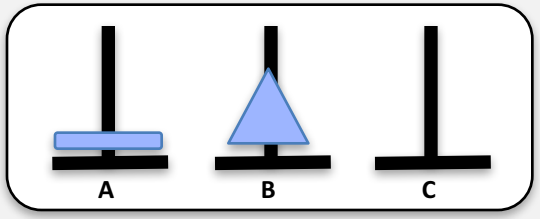
Reduction step: N disks.

The key insight is in how we view these N disks.

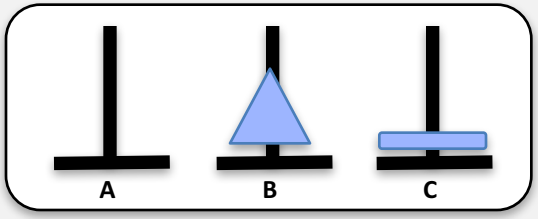


Solution:

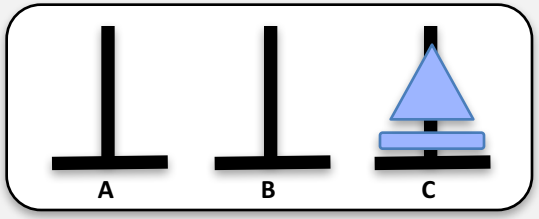
(1) Move N-1 disks from A to B.



(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

```
public 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);
   }
}
```

```
public void moveOneDisk(String startPeg, String endPeg) {
   System.out.println("Move one disk from " + startPeg + " to " + endPeg);
}
```

```
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

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

Dealing with the cost

Rewrite iteratively – new algorithm

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

```
public int factorial(int n) {
   int fact = 1;
   for (int i = 1; i <= n; i++) {
      fact *= i;
   }
   return fact;
}
```

Rewrite iteratively – imitate recursion with a stack

```
search(maze, position) {
   Mark current position as visited
   if (!finished) {
      for each open, not-visited neighbor {
         search(maze, neighbor)
      }
   }
}
```

```
search(maze, start) {
 push the start position onto the stack
 while ((current position != finish) && (stack not empty)) {
    if (∃ adj position open and not yet visited)
        push such a position onto the stack
        mark it visited
    else
        pop the stack
 }
}
```

Tail recursion

Tail recursion is a special case of recursion where the last operation of the method is the recursive call.

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

This is **not** tail recursive since the last operation performed is the multiplication, not the recursive call.

To write a method in tail recursive form, we will add parameters as necessary so that the computation is performed on the "down trip" via the parameters instead of performing it on the "up trip."

```
public int factTR(int n, int fact) {
   if (n == 0)
      return fact;
   else
      return factTR(n-1, n*fact);
}
```

For signature compatibility:

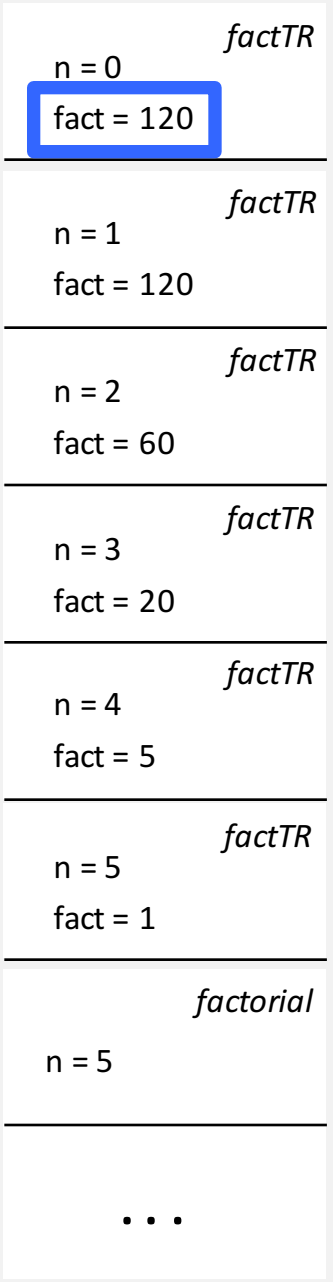
```
public int factorial(int n) {
   return factTR(n, 1);
}
```

Tail recursion – factorial in Java

```
public int factTR(int n, int fact) {
   if (n == 0)
      return fact;
   else
      return factTR(n-1, n*fact);
}
```

```
public int factorial(int n) {
   return factTR(n, 1);
}
```

No need to return!



When factTR is called on the base case, the parameter fact will hold n! – thus eliminating the need for returns, and thus the need for different activation records.

Compilers can eliminate tail recursion automatically, and this is a common optimization.

Note: I'm not aware of any Java compiler that eliminates tail recursion.

Tail recursion – factorial in C with gcc

GCC eliminates tail recursion in C programs with the –O2 optimization flag.
http://gcc.gnu.org/

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

First, let's look at the assembly code that would result from this function.

```
% gcc –S trfactorial.c
```

Stop after compilation, do not link.

Output is in assembler form.

```
_factTR:
Leh_func_begin1:
   pushq %rbp
Ltmp0:
   movq  %rsp, %rbp
Ltmp1:
   subq  $16, %rsp
Ltmp2:
   movl  %edi, -4(%rbp)
   movl  %esi, -8(%rbp)
   movl  -4(%rbp), %eax
   cmpl  $0, %eax
   jne   LBB1_2
   movl  -8(%rbp), %eax
   movl  %eax, -16(%rbp)
   jmp   LBB1_3
LBB1_2:
   movl  -4(%rbp), %eax
   movl  -8(%rbp), %ecx
   imull %ecx, %eax
   movl  -4(%rbp), %ecx
   subl  $1, %ecx
   movl  %ecx, %edi
   movl  %eax, %esi
  callq _factTR
   movl  %eax, %ecx
   movl  %ecx, -16(%rbp)
LBB1_3:
   movl  -16(%rbp), %eax
   movl  %eax, -12(%rbp)
   movl  -12(%rbp), %eax
   addq  $16, %rsp
   popq  %rbp
   ret
```

Tail recursion – factorial in C with gcc

GCC eliminates tail recursion in C programs with the –O2 optimization flag.
http://gcc.gnu.org/

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

Now, let's tell the compiler to optimize and look at the resulting assembly.

```
% gcc –S –O2 trfactorial.c
```

Perform second level optimization suite.

Includes tail recursive calls.

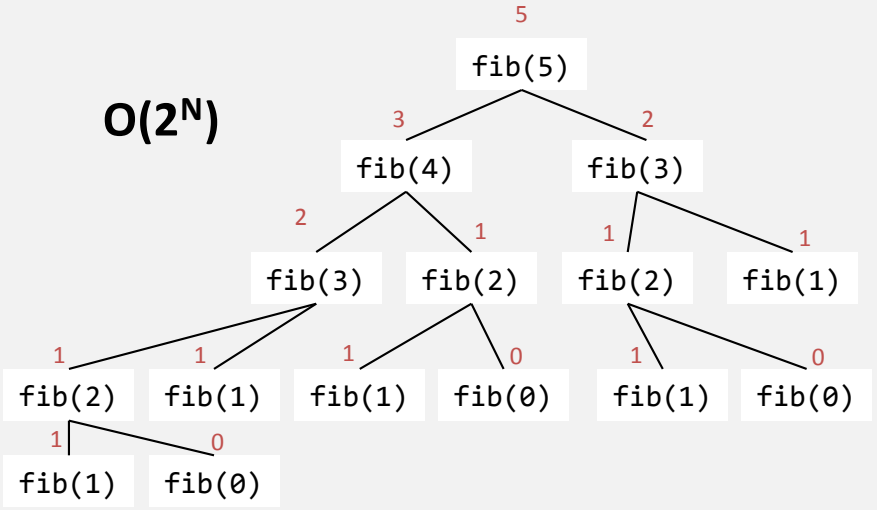
```
_factTR:
Leh_func_begin1:
   pushq %rbp
Ltmp0:
   movq  %rsp, %rbp
Ltmp1:
   jmp   LBB1_1
   .align  4, 0x90
LBB1_3:
   leal  -1(%rdi), %eax
   imull %eax, %esi
   addl  $-2, %edi
LBB1_1:
   testl %edi, %edi
   je LBB1_4
   imull %edi, %esi
   cmpl  $1, %edi
   jne   LBB1_3
LBB1_4:
   movl  %esi, %eax
   popq  %rbp
   ret
```

Tail recursion – fibonacci in Java

Sometimes writing a method in tail recursive form can speed things up independently of what the compiler does or does not do.

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

O(2^N)



The number in the sequence that we want.

Our progress – the number in the sequence that we are now calculating. [1..n]

The k^{th} fibonacci number.

The $(k-1)^{st}$ fibonacci number.

```
public int fibTR(int n, int k, int fibk, int fibk1) {
   if (n == k)
      return fibk;
   else
      return fibTR(n, k+1, (fibk + fibk1), fibk);
}
```

```
public int fib(int n) {
   if (n == 0) return 0;
   else return fibTR(n, 1, 1, 0);
}
```

Tail recursion – fibonacci in Java

Sometimes writing a method in tail recursive form can speed things up independently of what the compiler does or does not do.

```
public int fibTR(int n, int k, int fibk, int fibk1) {
   if (n == k)
      return fibk;
   else
      return fibTR(n, k+1, (fibk + fibk1), fibk);
}
```

```
public int fib(int n) {
   if (n == 0) return 0;
   else return fibTR(n, 1, 1, 0);
}
```

```
fib(7) ⟶ 13
```

This implementation of fibonacci is O(N).

We were able to improve from $O(2^N)$ to O(N) just by rewriting in tail recursive form.

Structure of recursive methods

The intuitive "if-else" template:

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

The else is often omitted since it's unnecessary:

```
public 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:

```
public 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

  }
```

Structure of recursive methods

Note when different parts of the method are executed:

```
public static void foo(int n) {
     // executed once per down trip step
     printBlanks(n);
     System.out.println("Down trip A: n = " + n);

     // base case
     if (n == 0) {
        printBlanks(n);
        System.out.println("Base case");
        return;
     }

     // executed once per down trip step
     printBlanks(n);
     System.out.println("Down trip B: n = " + n);

     // reduction step
     foo(n-1);

     // executed once per up trip step
     printBlanks(n);
     System.out.println("Up trip: n = " + n);
  }
```

```
Down trip A: n = 5
Down trip B: n = 5
 Down trip A: n = 4
 Down trip B: n = 4
  Down trip A: n = 3
  Down trip B: n = 3
   Down trip A: n = 2
   Down trip B: n = 2
    Down trip A: n = 1
    Down trip B: n = 1
     Down trip A: n = 0
     Base case
    Up trip: n = 1
   Up trip: n = 2
  Up trip: n = 3
 Up trip: n = 4
Up trip: n = 5
```

Structure of recursive methods

Note when different parts of the method are executed:

```
public static void bar(int n, int k) {
     printBlanks(n - k);
     System.out.println("Down trip: k = " + k + " n = " + n);

     if (k == n) {
        printBlanks(n - k);
        System.out.println("Base case: k = " + k + " n = " + n);
        return;
     }
     else {
        bar(n, k + 1);
        printBlanks(n - k);
        System.out.println("Up trip / Down trip: k = " + k + " n = " + n);
        bar(n, k + 1);
     }
     printBlanks(n - k);
     System.out.println("Up trip: k = " + k + " n = " + n);
  }
```

Structure of recursive methods

Note when different parts of the method are executed:

```
Down trip: k = 1 n = 5
 Down trip: k = 2 n = 5
  Down trip: k = 3 n = 5
   Down trip: k = 4 n = 5
    Down trip: k = 5 n = 5
    Base case: k = 5 n = 5
   Up trip / Down trip: k = 4 n = 5
    Down trip: k = 5 n = 5
    Base case: k = 5 n = 5
   Up trip: k = 4 n = 5
  Up trip / Down trip: k = 3 n = 5
   Down trip: k = 4 n = 5
    Down trip: k = 5 n = 5
    Base case: k = 5 n = 5
   Up trip / Down trip: k = 4 n = 5
    Down trip: k = 5 n = 5
    Base case: k = 5 n = 5
   Up trip: k = 4 n = 5
  Up trip: k = 3 n = 5
 Up trip / Down trip: k = 2 n = 5

 . . .  // lots omitted here

  Up trip: k = 3 n = 5
 Up trip: k = 2 n = 5
Up trip: k = 1 n = 5
```

Quicksort and Merge sort

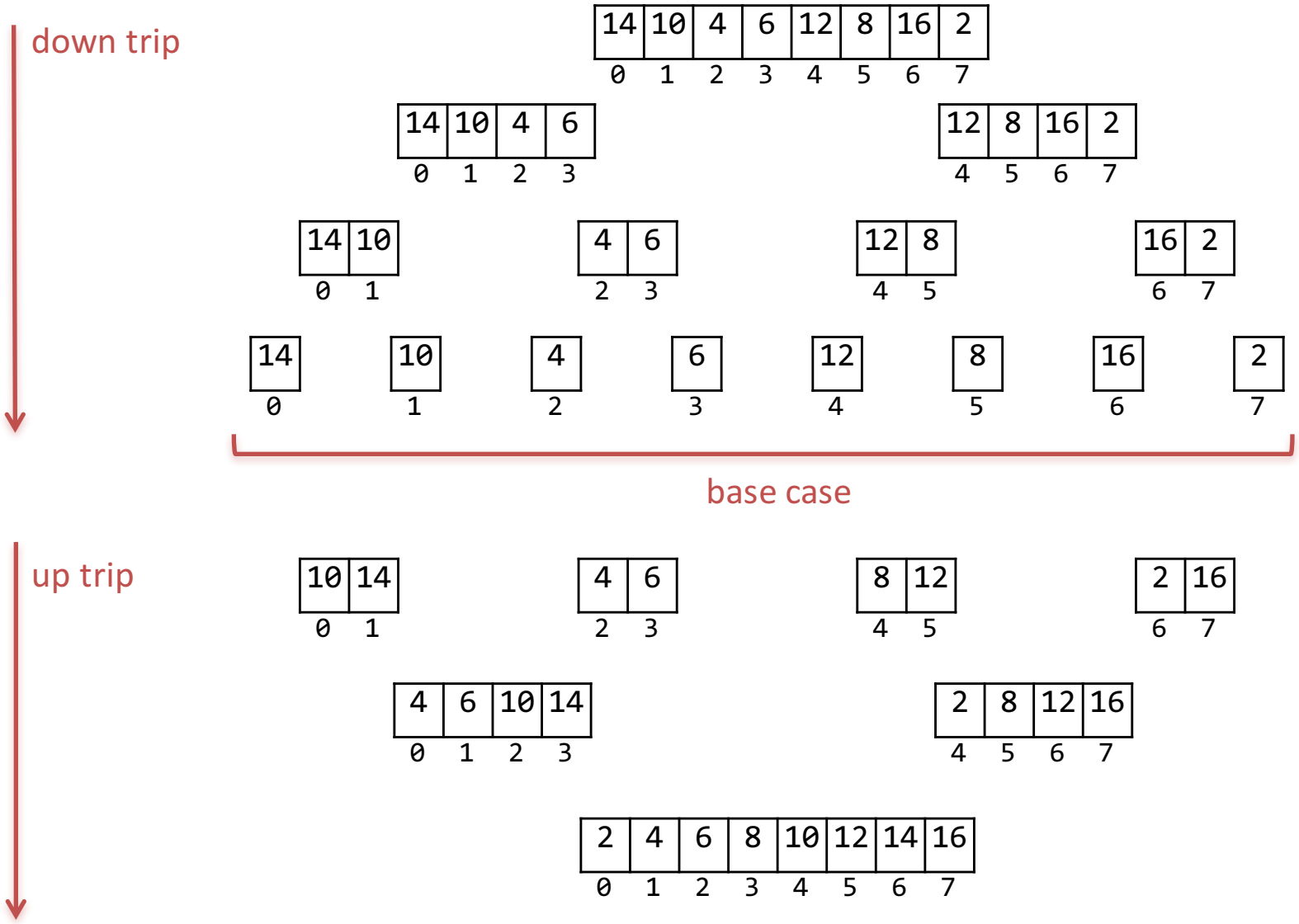
Quicksort does its heavy lifting during the **recursive down trip**.

```
public void qsort(Comparable[] a, int left, int right)
{
    if (right <= left) return;          } base case

    int j = partition(a, left, right);  } work done on down trip

    qsort(a, left, j-1);                }
    qsort(a, j+1, right);               } two reduction steps
 }
```

Merge sort does its heavy lifting during the **recursive up trip**.

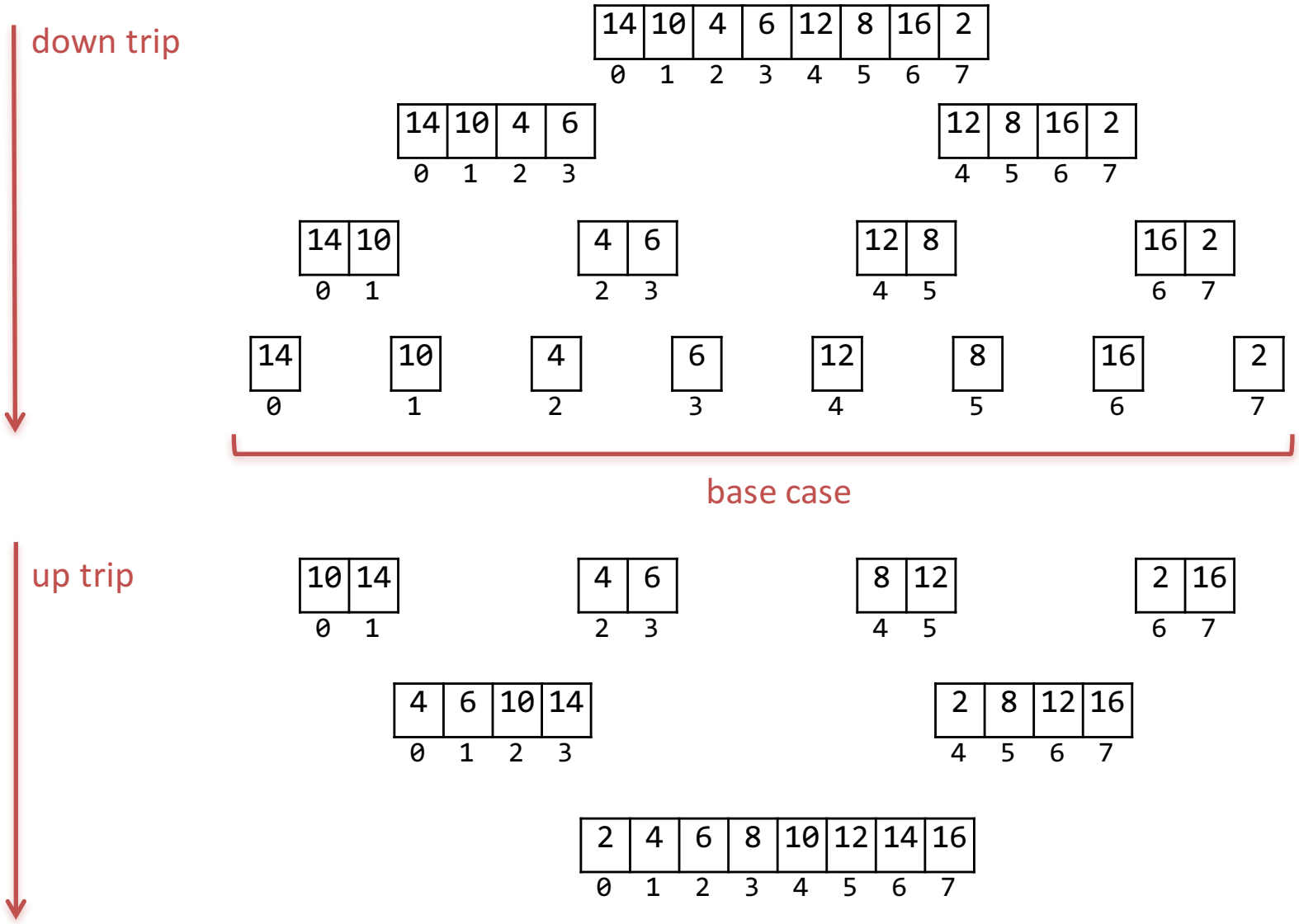
```
public void mergeSort(Comparable[] a, int left, int right) {

   if (right <= left) return;           } base case

   int mid = left + (right - left) / 2; } work done on down trip
   mergeSort(a, left, mid);             }
   mergeSort(a, mid + 1, right);        } two reduction steps

   merge(a, left, mid, right);          } work done on up trip
}
```

Merge sort


down trip
14 10 4 6 12 8 16 2
0 1 2 3 4 5 6 7
14 10 4 6
0 1 2 3
12 8 16 2
4 5 6 7
14 10
0 1
4 6
2 3
12 8
4 5
16 2
6 7
14
0
10
1
4
2
6
3
12
4
8
5
16
6
2
7
base case
up trip
10 14
0 1
4 6
2 3
8 12
4 5
2 16
6 7
4 6 10 14
0 1 2 3
2 8 12 16
4 5 6 7
2 4 6 8 10 12 14 16
0 1 2 3 4 5 6 7

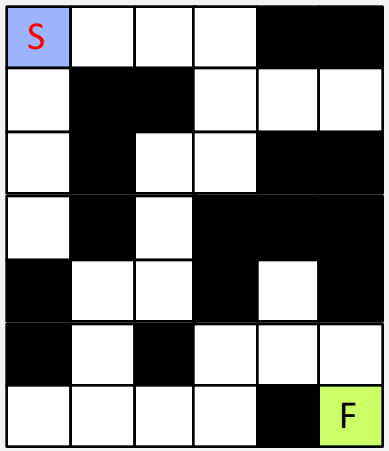
Merge sort

Order of execution



[redacted]

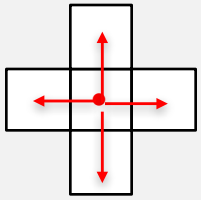
Depth-first search

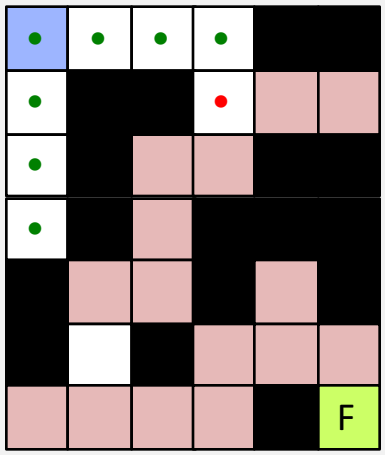


Base case:

F

Reduction step:






```
search(maze, position) {
    Mark current position as visited
    if (!finished) {
        for each open, not-visited neighbor
{
            search(maze, neighbor)
        }
    }
}
```