Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminder

- Project 4 (Postfix) due this Friday
- Midterm 1 will be given back to you on Monday discussion sections.

More Topics on Recursion

- Fractals
- Recursion on Linked Structures
- Printing a Linked List Backwards
- How Recursion Works Behind the Scenes
- Efficiency of Recursion

Fractals

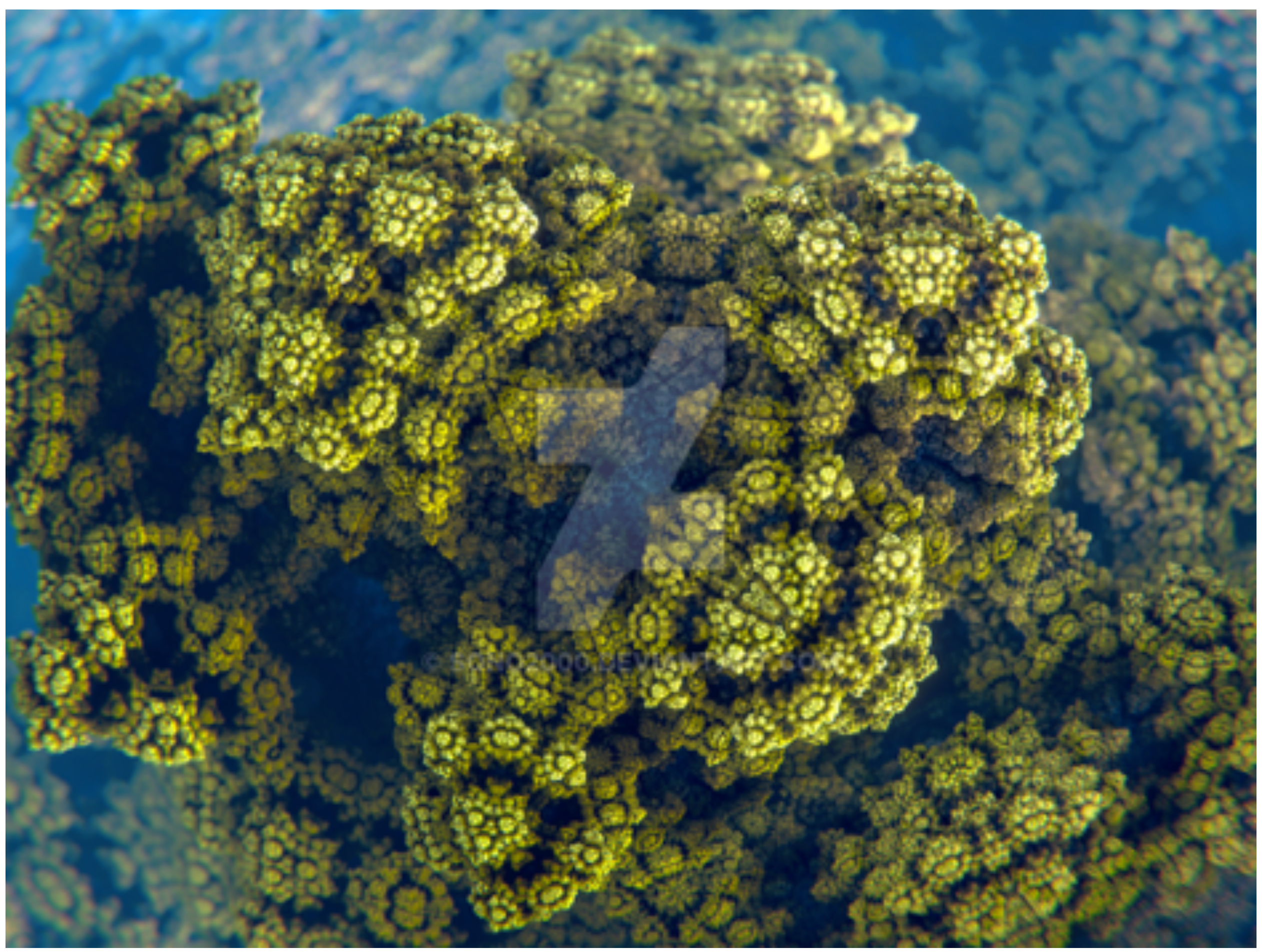
- Recursion exists abundantly in nature.
- What is a Fractal?

Fractals

- What is a Fractal?
  - Natural structure or phenomenon that exhibits repeating patterns at every scale (self-similarity).

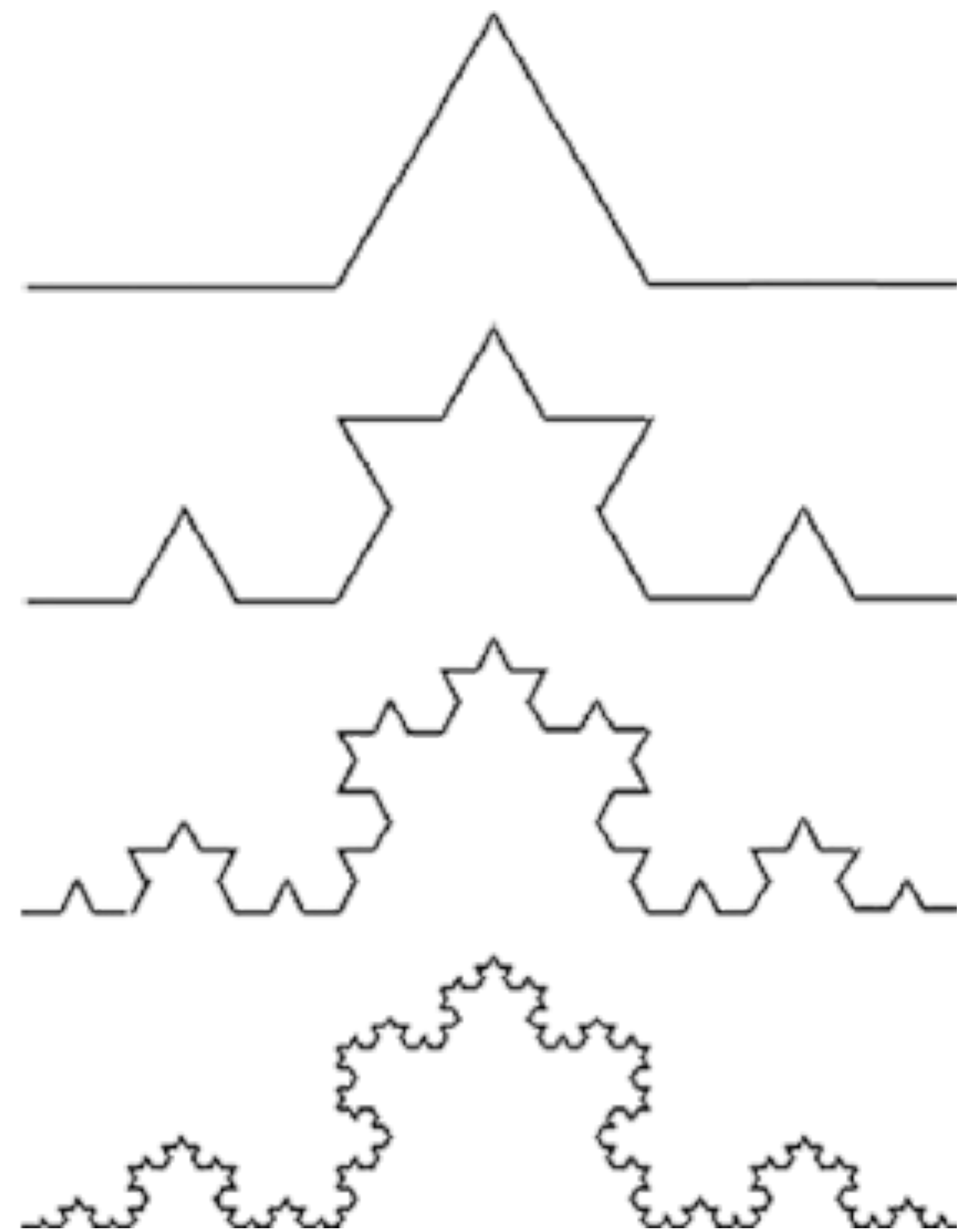
Fractals

- What is a Fractal?
  - Natural structure or phenomenon that exhibits repeating patterns at every scale (self-similarity).



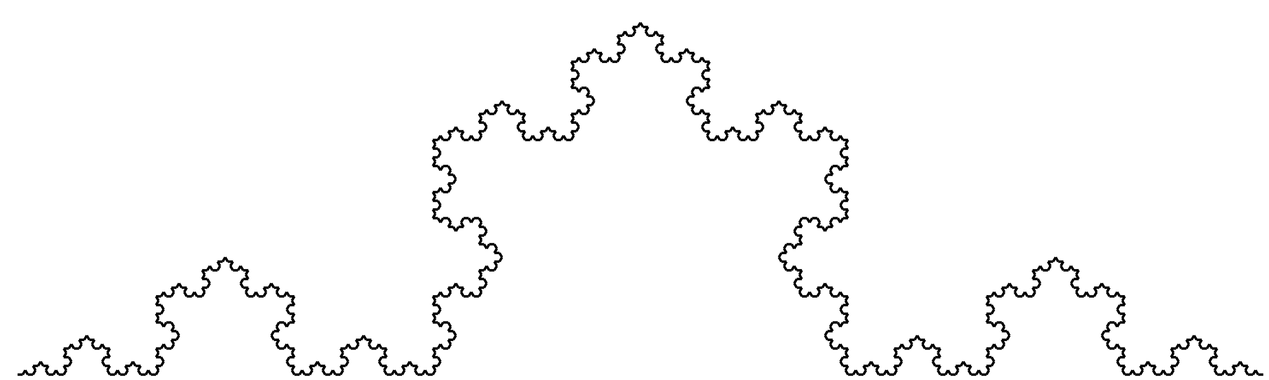
Fractals

- The simplest fractal shape is the **Koch** curve.
  - Start from a straight line
  - Divide it into 3 equal segments.
  - Make an equilateral triangle with the middle segment as the base. Then remove the middle segment.
  - Repeat the same procedure for each segment (this is the recursion part).



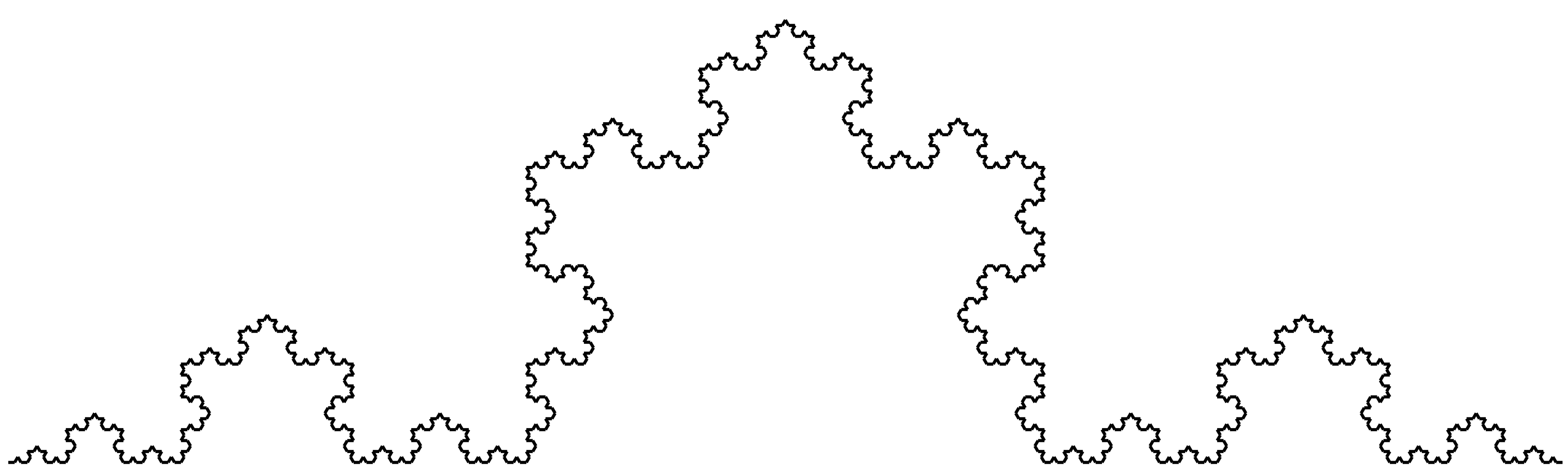
Fractals

- As you can see, the shape of the curve will quickly become very complex.
- **Mathematically, this is an infinite recursion**. When zooming in, you will see infinite details.
    - Wiki page (the zooming animation)
    - Cool Youtube Video
- The length of the Koch curve is…. **infinity! Why?**



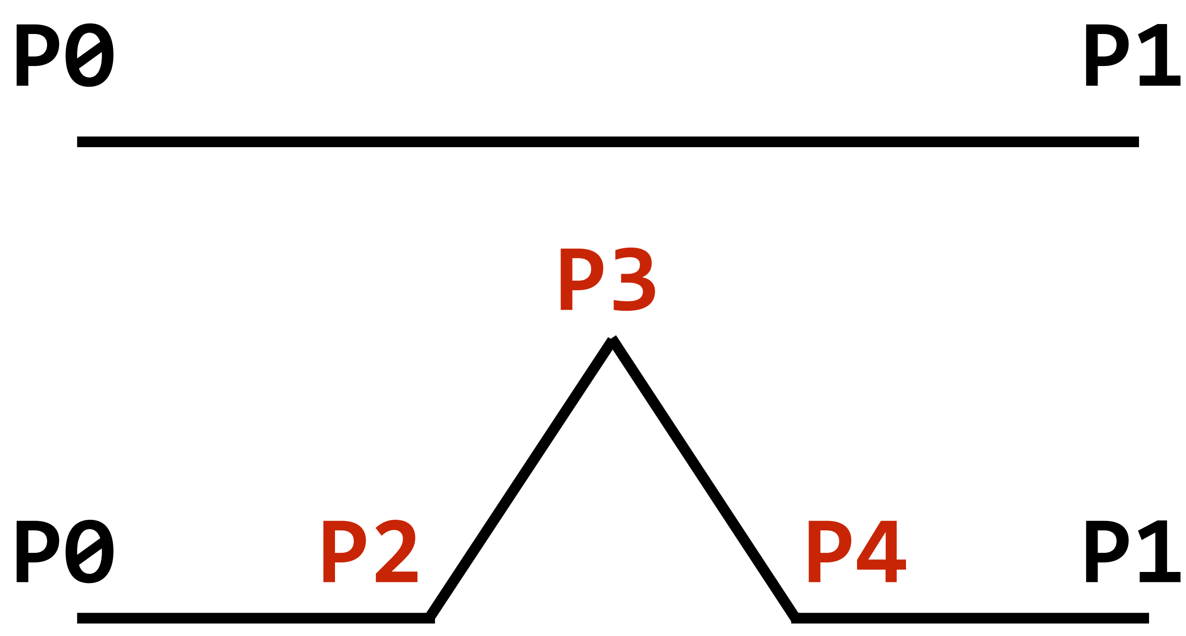
Fractals

- **Computationally, we can set a limit (base case)**. For example, when the line segment falls below the size of a pixel on the screen, we can stop.
- Real-time Demo
- How do you write the code for fractals?



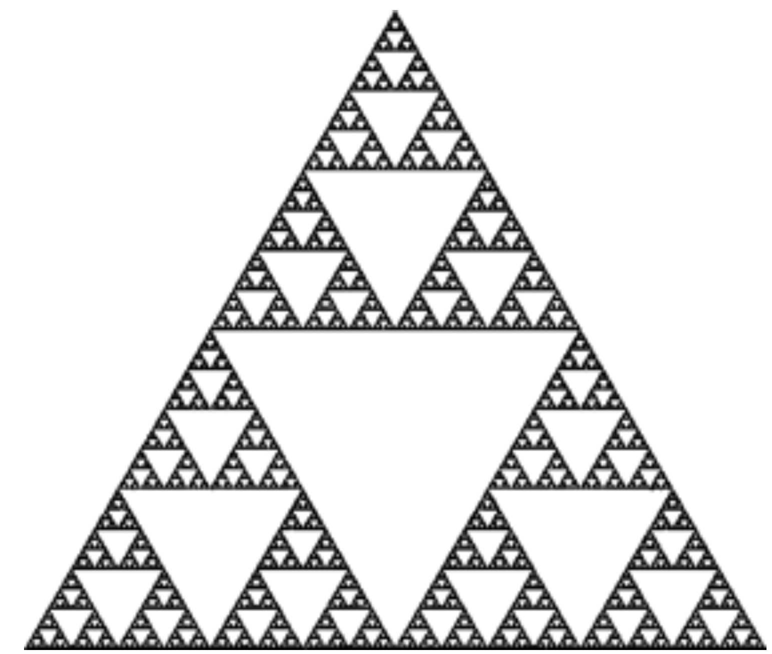
Koch Curve Pseudo-Code

```
void drawKoch(Point P0, Point P1) {
  if(distance(P0,P1) < pixel_size) { // base case
    draw_pixel(P0); return;
  }
  Compute the coordinates of P2, P3, P4; // some math
  drawKoch(P0, P2);
  drawKoch(P2, P3);
  drawKoch(P3, P4);
  drawKoch(P4, P1);
}
```



Fractals

- **Serpenski Triangle:**
  - Start with an equilateral triangle.
  - Split into 4 equal sub-triangles.
  - Remove the middle sub-triangle.
  - Repeat on each sub-triangle.



Clicker Question #1

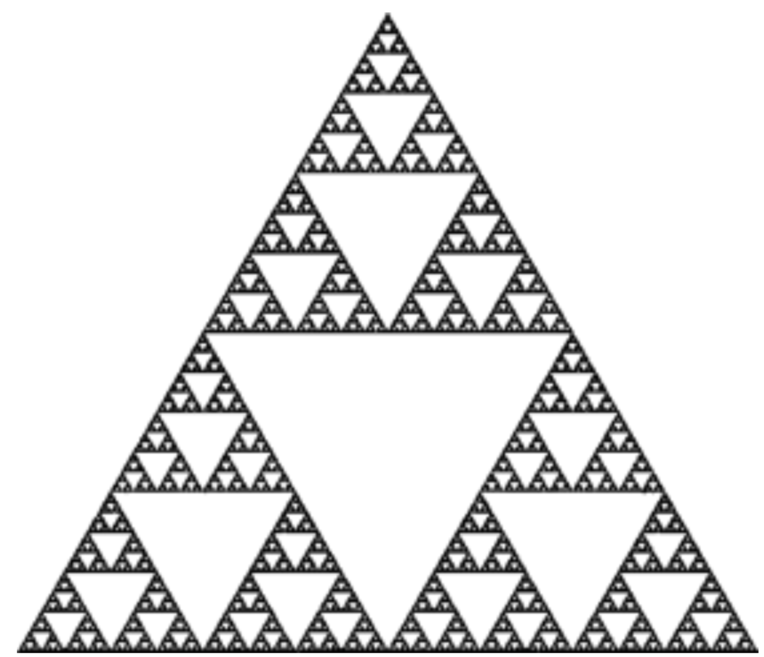
- Assuming the starting equilateral triangle has an area of 1. What's the total area (excluding empty/white space) covered by the Serpenski Triangle?

(a) 0

(b) 1/2

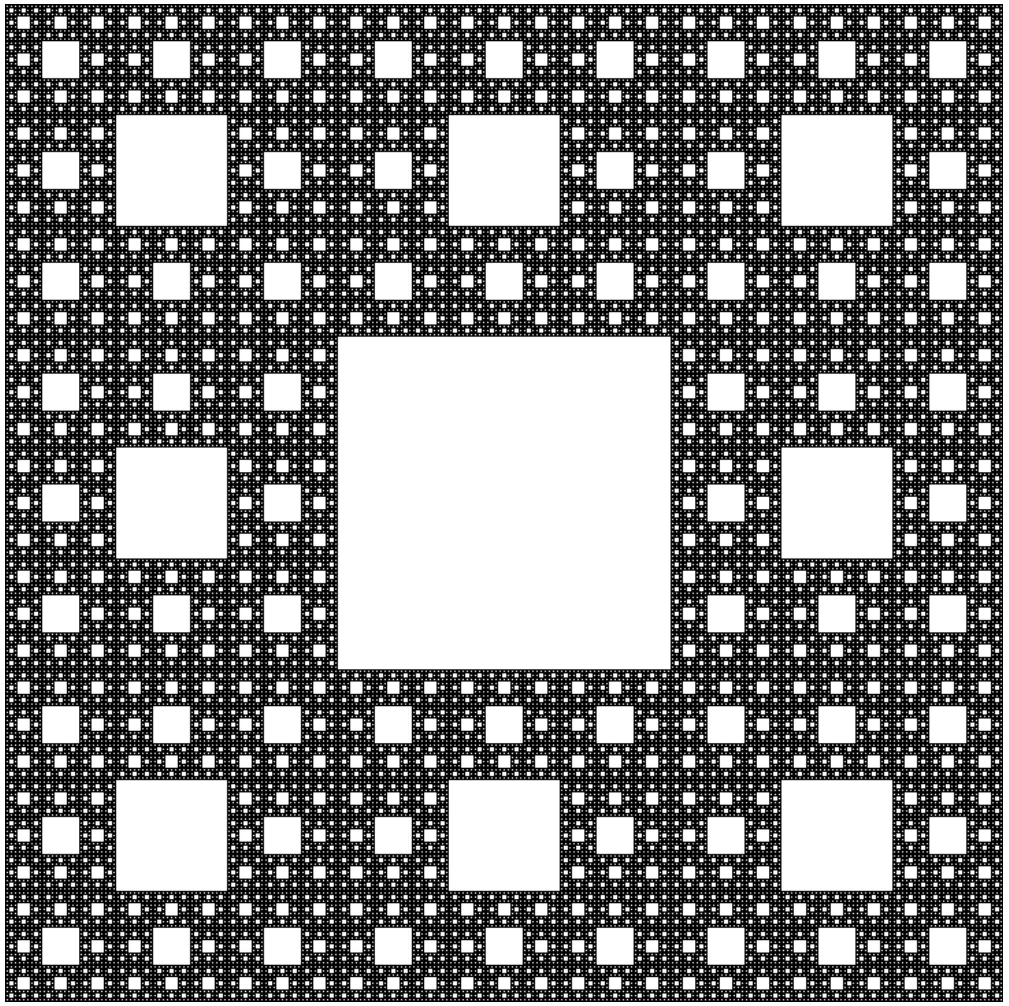
(c) 3/4

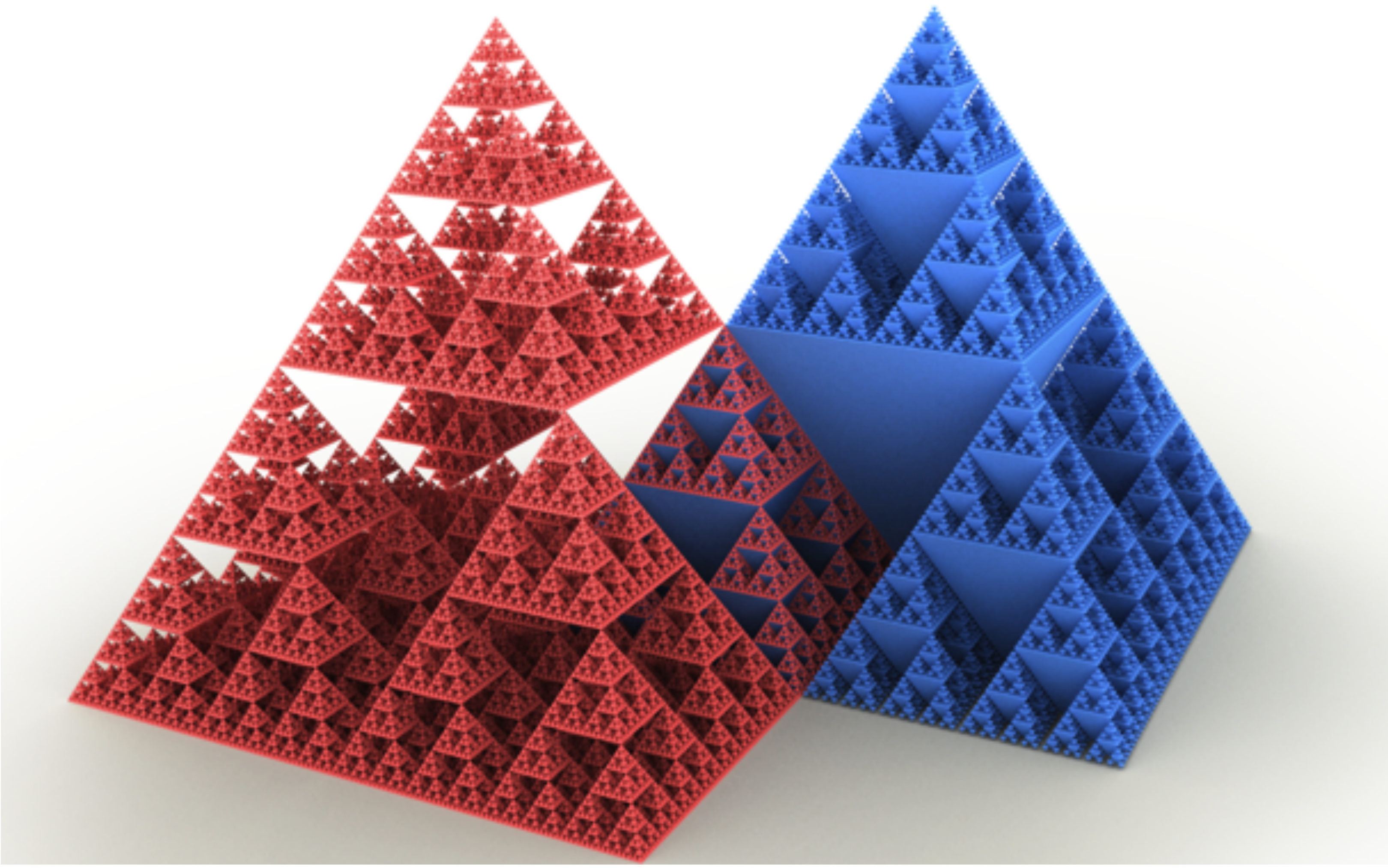
(d) infinity



Fractals

- **Serpenski Carpet and Pyramid:**
  - When you feel bored, create your own fractal rules and see what novel shapes you get!





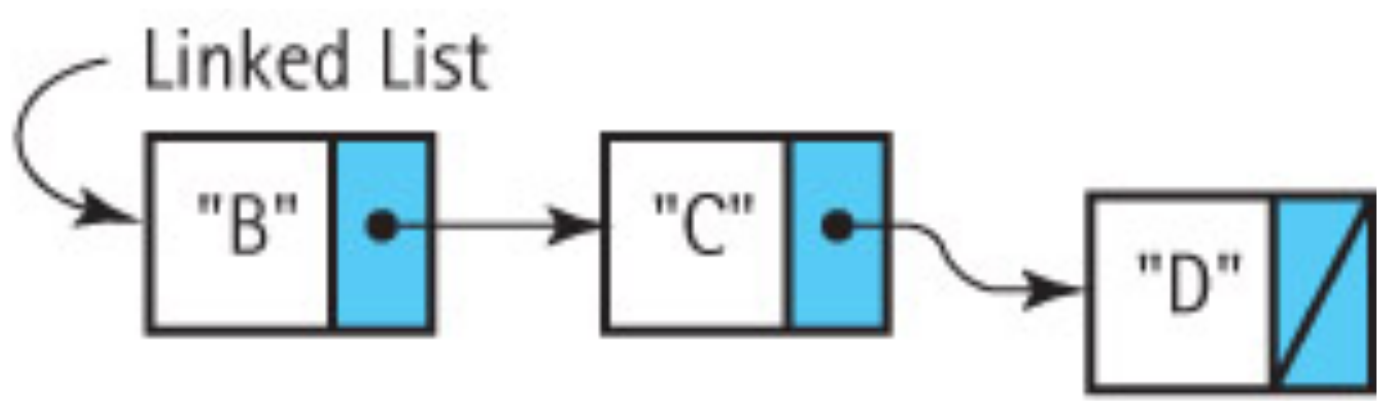
Recursion on Linked Lists

- Done with the visually stunning examples. Now come back to our favorite friend: Linked List.
- Recall that **`LLNode<T>`** is a self-referential structure and its definition bears similarity to the idea of 'recursion' (i.e. something that refers to itself).
- Turns out we can use recursion to easily solve a number of challenging problems involving linked lists.

Printing a List Backwards

- Goal: print out the elements in a linked list in **reverse order**, with the tail element printed first and the head element printed last.

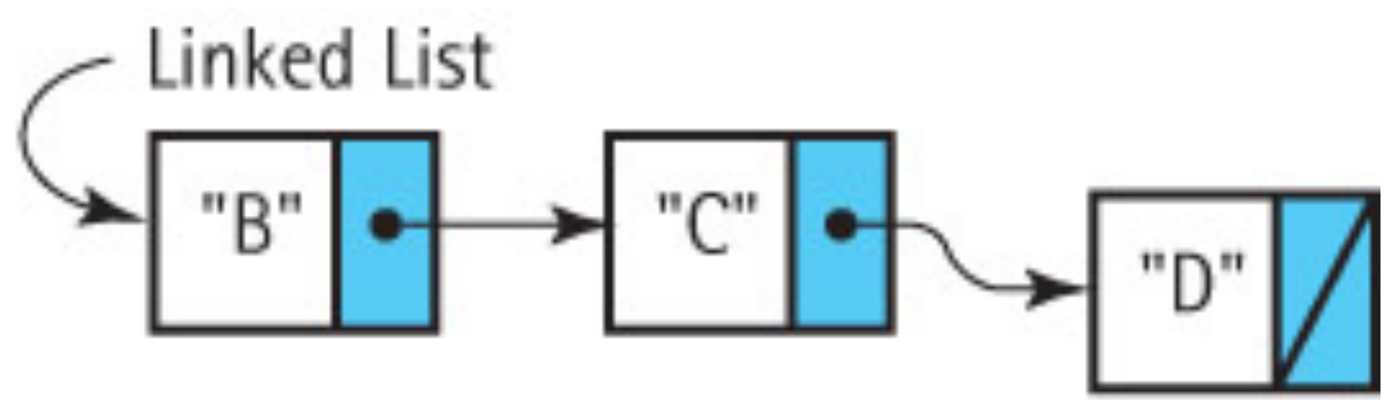
- Pause for a moment to think about how you would solve it.



The goal is to print out: D, C, B

Printing a List Backwards

- We know how to traverse the linked list to find any element on the chain. So we could use a nested loop: the outer loop goes from **`i=n-1`** to **`0`**, and the inner loop basically performs **`elementAt(i)`** which is what occurred in the midterm.
- Note that the traversal is needed as you can't directly jump to the i-th element on the chain.
- But this is $O(n^2)$, which is quite expensive!



Printing a List Backwards

- Now let's think about the problem recursively:
  - If the list (referenced by the current node) is empty we have nothing to do.
  - If it's not empty, we **print out the second through last elements in reverse order**, then print the content of the current node.
  - The bold-font part above is the recursive step.

Printing a List Backwards

```
private void revPrint (LLNode<T> listRef) {
    if (listRef != null)
    {
        revPrint(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

- Show a demo on the board
- To reverse print the entire list, we call the method with the first node (i.e. head node) as the parameter:

  `revPrint(head);`

Printing a List Backwards

```
private void revPrint (LLNode<T> listRef) {
    if (listRef != null)
    {
        revPrint(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

It's a common practice to wrap the initial call (i.e. with the head node as parameter) into a separate, public method, so the **head** variable is not exposed to the outside.

```
public void revPrint ()  {revPrint(head);}
```

Clicker Question #2

```
private void revPrint (LLNode<T> listRef) {
    if (listRef != null)
    {
        revPrint(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

- What's the cost of the `revPrint` method, if we run it a liked list with n elements? (Hint: how many times does it visit each node?)

    (a) O(1)

    (b) O(log n)

    (c) O(n)

    (d) O(n^2)

Clicker Question #3

```
private void revPrint (LLNode<T> listRef) {
   if (listRef != null)
   {
      revPrint(listRef.getLink());
      System.out.println(listRef.getInfo());
   }
}
```

- What happens if we swap the two lines of code inside the **`if`** statement?

  (a) It will run into `StackOverflowException`.

  (b) It will throw a `NullPointerException`.

  (c) It will run ok but print out elements in forward order.

  (d) Nothing will change

Printing a List Backwards

```java
private void revPrint (LLNode<T> listRef) {
    if (listRef != null)
    {
        revPrint(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

The running time is **O(n)**, a lot better than the naive $O(n^2)$ solution (which involves a nested loop).

Fundamentally, how is it able to reduce the cost to O(n)?

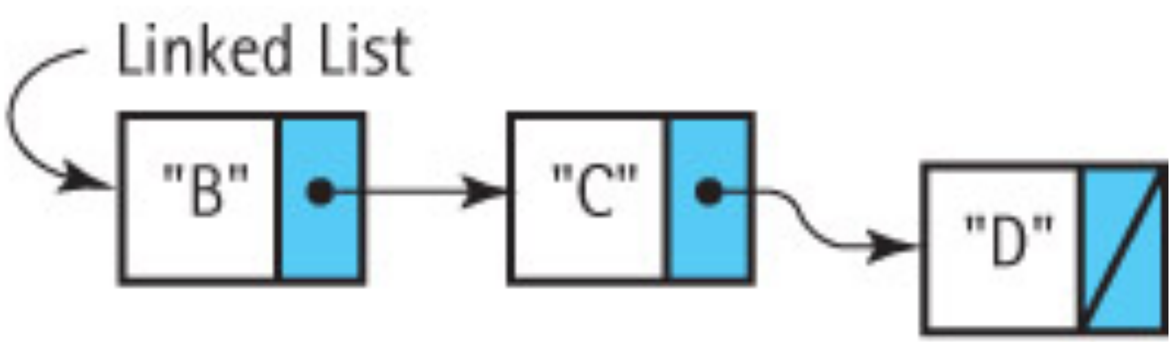
Behind the Scenes

```
private void revPrint (LLNode<T> listRef) {
   if (listRef != null) {
      revPrint(listRef.getLink());
      System.out.println(listRef.getInfo()); }
}
```

- At the last lecture, we explained how computer systems use stacks to manage method calls.
- Each stack frame preserves the **local variables** (in this case the **`listRef`** variable) as well as the return link (i.e. which line of code to resume to, once the method returns).

Behind the Scenes

```
private void revPrint (LLNode<T> listRef) {
  if (listRef != null) {
    revPrint(listRef.getLink());
    System.out.println(listRef.getInfo()); }  // ret link
}
```

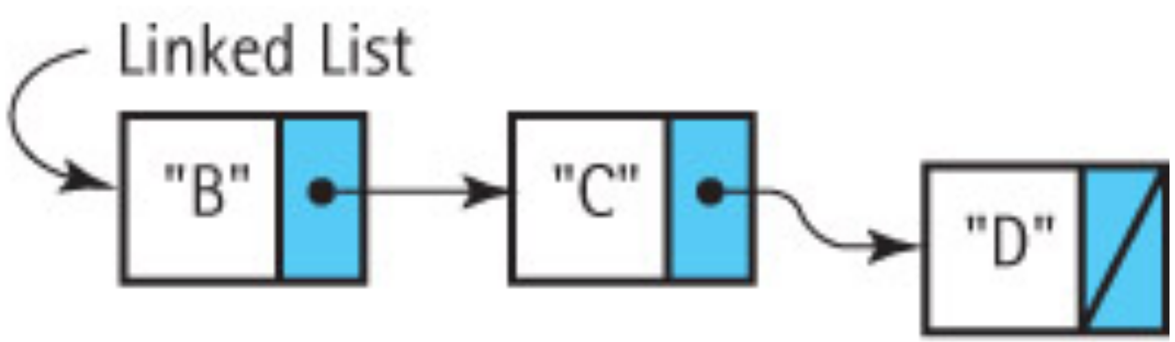


- Calling the method on this linked list eventually produces a system stack with 3 stack frames.

`listRef`—> (null)
`listRef`—> “D” node
`listRef`—> “C” node
`listRef`—> “B” node

Behind the Scenes

```
private void revPrint (LLNode<T> listRef) {
  if (listRef != null) {
    revPrint(listRef.getLink());
    System.out.println(listRef.getInfo()); }  // ret link
}
```



- Once the recursion reaches the base case, it returns, and the program execution continues to the next line: **`System.out.println`**.

`listRef`—> (null)
`listRef`—> “D” node
`listRef`—> “C” node
`listRef`—> “B” node

Behind the Scenes

```
private void revPrint (LLNode<T> listRef) {
  if (listRef != null) {
    revPrint(listRef.getLink());
    System.out.println(listRef.getInfo()); }  // ret link
}
```

- So the recursion implicitly leverages the system stack, which you can think of as an auxiliary data structure.
- If you are allowed to use a stack to implement ‘reverse print’, you can certainly achieve it in O(n) too, by visiting every node in order, while pushing it to the stack; then pop the stack one-by-one and print. This is essentially how the recursive solution works behind the scenes in this particular example.

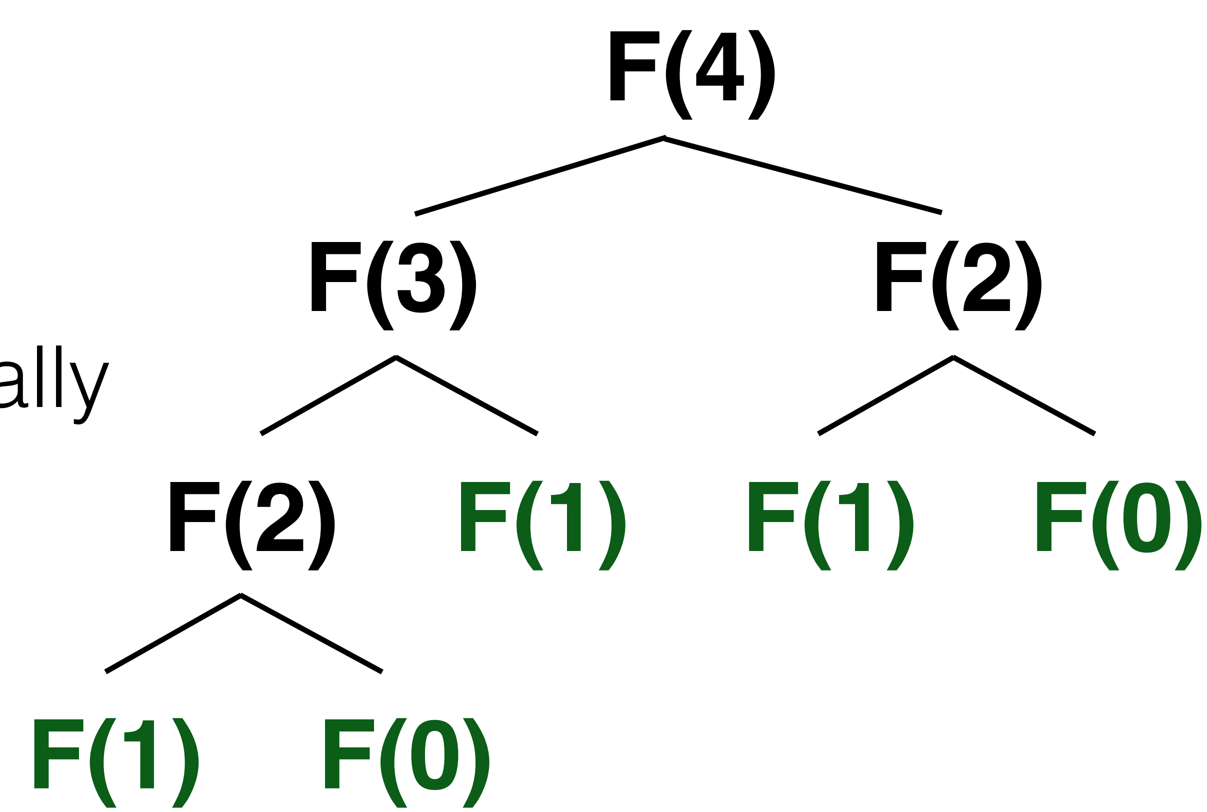
Efficiency of Recursion

- Recursion is a double-edged sword: it’s conceptually simple to solve many problems, but it’s not really computationally efficient.
- Pushing, popping the system stack, and managing method calls / returns incur a lot of overhead.
- In addition to resource consumption, it also doesn’t **cache** the intermediate results, so yo can end up computing the same thing over and over again.
  - What does this mean? Let’s take a look at the Fibonacci method again.

Efficiency of Recursion

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

- The graph shows the complete list of recursive calls to compute F(4). Note that F(2) appears twice. Since recursion doesn't automatically remembers (caches) the result of prior computations, you end up wasting a lot of computations.



Clicker Question #4

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

- When calling `Fibonacci(6)`, How many times will you encounter `F(2)` during the recursion?

  (a) 2

  (b) 3

  (c) 5

  (d) 8

Efficiency of Recursion

- Obviously, a more efficient way is to **cache** the intermediate results as you go, so you don't have to re-compute the same number over and over again.
    - This is similar to the **dynamic programming** idea which you will learn in upper-level classes.
- For **Fibonacci**, a more straightforward solution is to just forward compute the sequence, starting from F(0) and F(1). With a O(n) loop you can compute all Fibonacci numbers from from F(0) to F(n).

Efficiency of Recursion

- There are many cases where a non-recursive solution (e.g. using loops) is more efficient (resource-wise and/or computation cost-wise). So it makes sense to use loops (instead of recursion) as you can.

- Other problems, like the Towers of Hanoi, is conceptually much easier to implement using a recursive solution, this saves the programmer's time.

- So whether to use recursion or not recursion depends on the specific problem, running cost, and how much time you are willing to spend thinking and coding it.

Reverse a Linked List

```
public void reverse(LLNode<T> curr) {
    if (curr == null) {
        return;
    }
    if (curr.getLink() == null) {
        head = curr;
        return;
    }



}
```

Reversing a Linked List

```
public void reverse(LLNode<T> curr)
{
  if (curr == null) {
    return;
  }
  if (curr.getLink() == null) {
    head = curr;
    return;
  }
  reverse(curr.getLink());
  curr.getLink().setLink(curr);
  curr.setLink(null);
}
```

- The base case is a list of size 0 or 1.
- We make progress because each recursive call is to a list that is smaller by one.
- If the recursive call works, we reverse the rest of the list, and put cure at the tail.