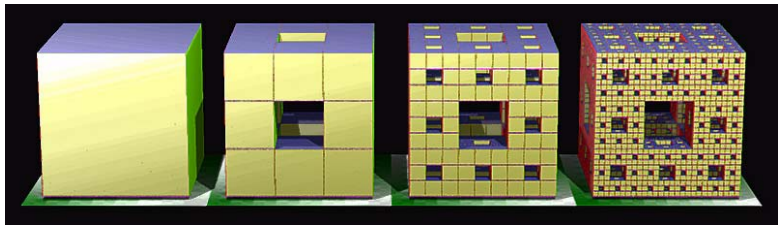
COMPSCI 105

Principles of Computer Science



Recursion

Base case

- *Does not contain a recursive call*

Recursive case

- *Method that calls itself*
- *Makes progress towards the base case*

Thinking recursively

- Can you define the problem in terms of a smaller problem of the same type?
- Does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

Contents

Recursive Curve

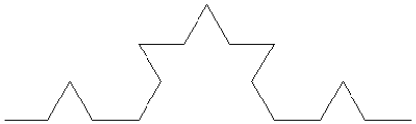
Order 0

Recursive Curve

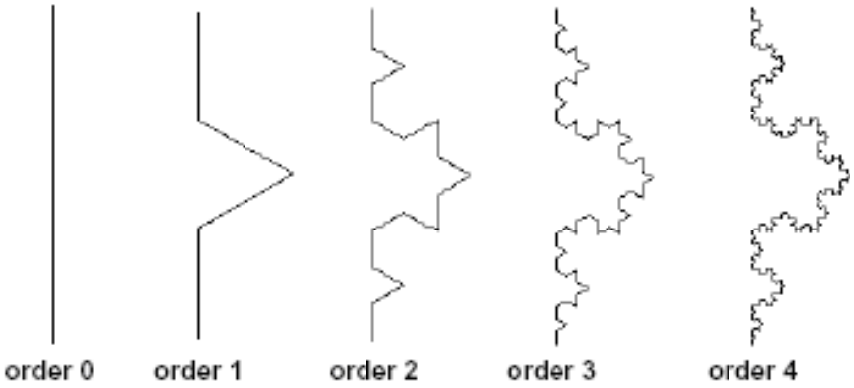
Order 1

Recursive Curve

Order 2



Koch curve



Drawing recursive curves

Use a Turtle class

- Logo

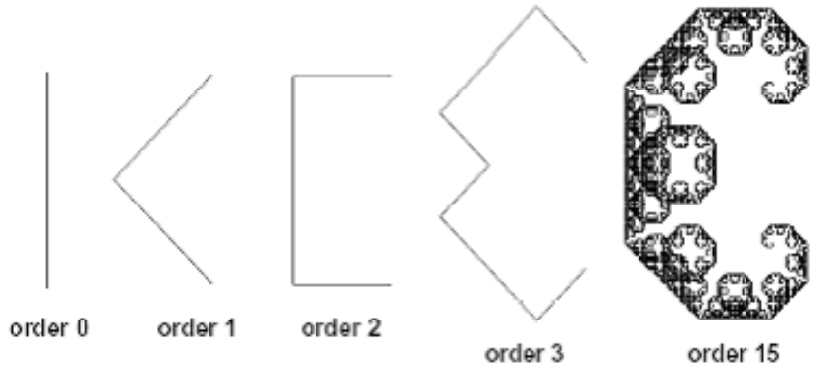
Methods

- penUp
- penDown
- move(distance)
- turn(angle)

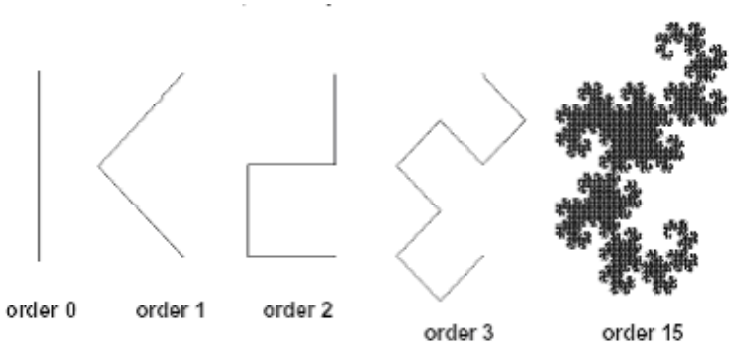
Algorithm for Koch

```
koch(order, size)
      if order is 0
            move size units
      else
            draw koch(order-1) at size/3 units
            turn 60 degrees
            draw koch(order-1) at size/3 units
            turn -120 degrees
            draw koch(order-1) at size/3 units
            turn 60 degrees
            draw koch(order-1) at size/3 units
```


C Curve

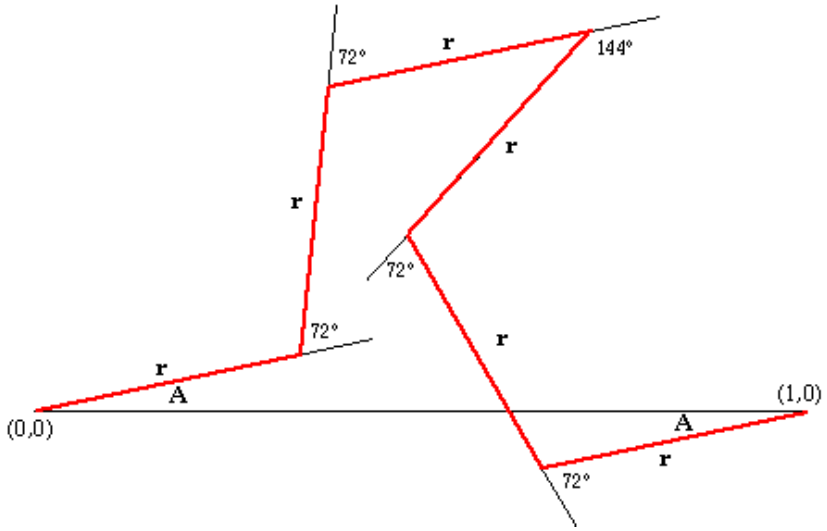


Dragon Curve

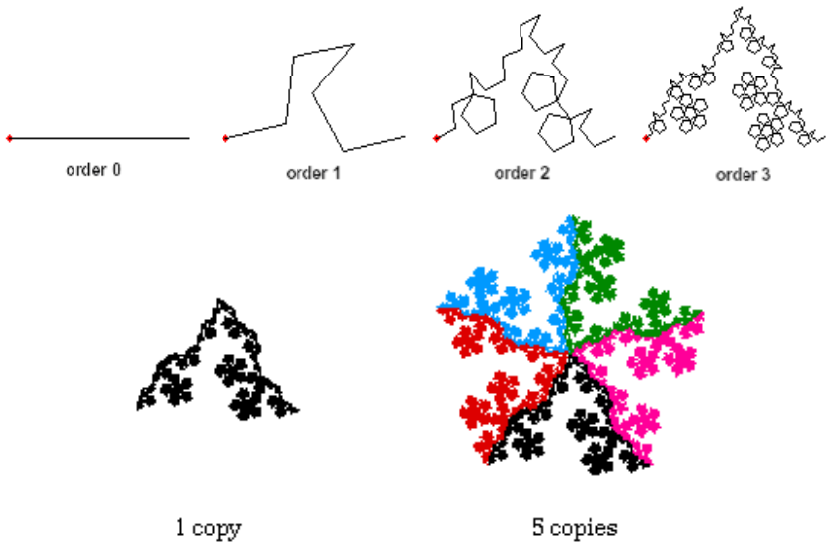


Sierpinski Triangle

Pentadentrite



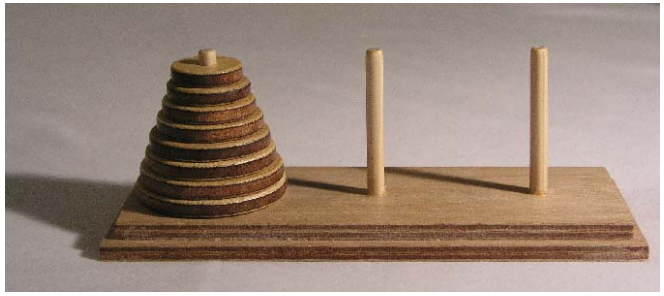
Pentadentrite



The Towers of Hanoi

Move all the disks from one peg to another

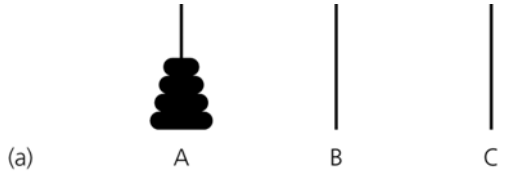
- One at a time
- Larger disk cannot be placed on a smaller disk



http://www.hanoitower.info/

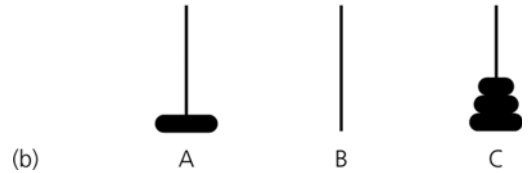
Think recursively

Solve the problem - move all disks from A to B



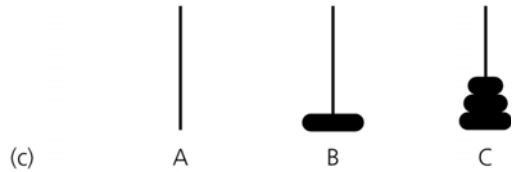
Think recursively

First, move (n-1) pegs from A to C



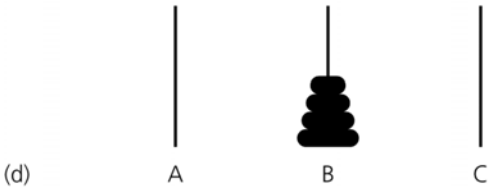
Think recursively

Then, move the single disk from A to B



Think recursively

Finally, move (n-1) disks from C to B



Recursive Solution

```
solve (n, source, destination, spare)
   if (n is 1) {
        Move a disk directly from source to destination
   } else {
        solve(n-1, source, spare, destination)
        solve(1, source, destination, spare)
        solve (n-1, spare, destination, source)
   } //end if
```

Java Code

Prints out the moves required to solve the problem

```
public static void main(String[] args) {
       int disks = Integer.parseInt(args[0]);
       solve(disks, 'A', 'B', 'C');
}

private static void solve(int n,
                    char src, char dest, char spare) {
       if (n == 1)
              System.out.println("move disk from " +
                                  src + " to " + dest);
       else {
              solve(n - 1, src, spare, dest);
              solve(1, src, dest, spare);
              solve(n - 1, spare, dest, src);
       }
}
```


Binary Search - algorithm

```
if (anArray is of size 1) {
       Determine if anArray's item is equal to value
} else {
       Find the midpoint of anArray
       Determine which half of anArray contains value
       if (value is in the first half of anArray) {
              binarySearch (first half of anArray, value)
       } else {
              binarySearch(second half of anArray, value)
       }
}
```


Binary search

Implementation issues:

- How will you pass "half of `anArray`" to the recursive calls to `binarySearch`?
- How do you determine which half of the array contains value?
- What should the base case(s) be?
- How will `binarySearch` indicate the result of the search?

Exercise

Write the code for a recursive binary search.

```
//Write your code here
```

Permuations

Imagine we want to print out all the possible permutations of a String of letters

Example: Permutations of "ABC"

- "ABC"
- "ACB"
- "BAC"
- "BCA"
- "CAB"
- "CBA"

Write a program that prints out all the permutations of a given set of letters.

Thinking recursively

Define the problem in terms of a smaller problem of the same type?

Take the String given

- Print out the first letter
- Print out all combinations of the remainder
- Repeat for the other letters

Given "ABC"

- A BC, CB
- …

Given "ABCD"

- A BCD, BDC, CBD, CDB, DCB, DBC
- …

Thinking recursively

Keep track of the first part, only print when all the letters are combined

Given a prefix and a String

- Move the first letter of the String to the prefix
- Generate all the permutations of the remaining letters
- Only print when there are no more letters remaining

- When the first letter is done, repeat the process for the next letter.

Pseudocode

```
permutation(String prefix, String remainder) {
    if the length of the remainder is 0 then {
        print out the prefix
    } else {
        for each letter in the remainder do {
            remove the letter from the remainder
            add the letter to the prefix
            permutation (prefix, remainder)
        }
    }
}
```

Code

```
public static void main(String[] args){
   permutation("", "ABCD");
}

private static void permutation(String pre, String rem) {
   if (rem.length() == 0) {
         System.out.println(pre);
      } else {
         for (int i = 0; i < rem.length(); i++) {
            char letter = rem.charAt(i);
            String newRem = rem.substring(0, i) +
                            rem.substring(i+1);
            String newPre = pre + letter;
            permutation (newPre, newRem);

         }
      }
   }
}
```