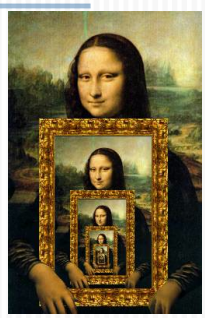
Computer Science 105 S2 C Lecture 15 – and a bit of 16 - Contents

Recursion 3

Textbook: Chapter 10



Sorting recursively

Accept a pile of cards.

If you were given a single card, return it,

else

1. Divide pile in half
2. Give first half to neighbour and wait for it to be returned
3. Give second half to neighbour and wait for it to be returned
4. Give both halves to the Merge person and wait for it to be returned sorted
5. Return the sorted pile to the person that gave it to you

Mergesort

Mergesort is a recursive sorting algorithm.

Gives the same performance, regardless of the initial order of the array items.

Mergesort Strategy
- Divide an array into halves,
- Sort each half,
- Merge the sorted halves into one sorted array

A divide-and-conquer sorting algorithms

Mergesort - pseudocode

```
private void mergeSort(int[] a, int start, int end) {

   If number of elements to sort is one or fewer, return

   Find the middle position

   Sort left half (start, middle - 1)  //i.e., not
                                       //including the
                                       //middle element

   Sort right half (middle, end)

   Merge both halves
}
```


Mergesort - code

```
1 private void mergeSort(int[] anArray,
                         int start, int end) {
2   if (first < end - 1) {
3     int mid = (first + end) / 2; //find the midpoint
4     mergesort(anArray, first, mid);
5     mergesort(anArray, mid, end);
6     merge(anArray, first, mid, end);
7   }
8 }
```

Note: the `end` value is the position **one past the position of the last value** to be sorted.

Merging - pseudocode

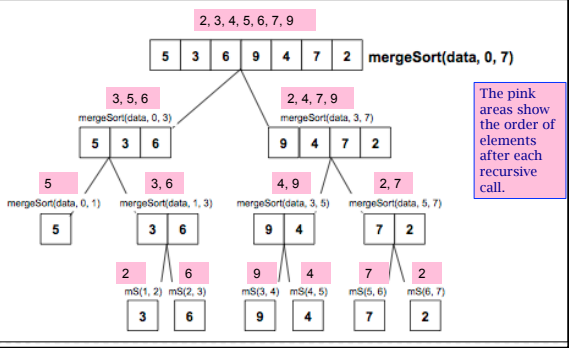
Create a temporary array of the same size as the sum of the two arrays you are combining,

for each element of the temporary array,
compare elements from each sorted half
and
assign smallest element to the temporary array

Mergesort - Merging

```
private void merge(int[] nums, int start, int mid, int end) {
   int temp[] = new int[end - start];
   int fromLeft = start;
   int fromMid = mid;
   for(int k = 0; k < temp.length; k++) {
       if (fromMid >= end || (fromLeft < mid &&
                      nums[fromLeft] <= nums[fromMid])) {
            temp[k] = nums[fromLeft];
            fromLeft++;
       } else {
            temp[k] = nums[fromMid];
            fromMid++;
       }
   }
   for(int k = start; k < end; k++) {
        nums[k] = temp[k-start]; //copy array back into the array, nums
   }
}
```


Mergesort - Call Tree



The pink areas show the order of elements after each recursive call.

Cost of Mergesort

If you mergesort 2^{100} items,

What is the height of the recursive tree?

How many calls are there along the lowest level of the recursive tree?

What is the cost of the mergesort algorithm?

Recursion and Efficiency

Some recursive solutions are so inefficient that they should not be used.

Factors that contribute to the inefficiency of some recursive solutions:

- overheads associated with method calls,
- inherent inefficiency of some recursive algorithms

If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.

Recursion – The Costs

Recursive solutions are often harder to debug

Recursion requires numerous method calling:
(this can lead to much poorer run-time performance)

Takes up more memory space (on the run-time stack)

In some cases the solution may be much less efficient than an iterative solution

Recursion – The Benefits

Often your code is clearer and shorter

It is usually easier to define the solution recursively (since writing the code is often just a matter of implementing the definition)

More elegant solution

You may not need local variables to implement your solution

Some solutions are only expressible recursively (or at least are only easily expressible recursively) e.g. binary search

Summary

Recursion solves a big problem by solving smaller versions of the same problem.

Four questions to keep in mind when constructing a recursive solution:

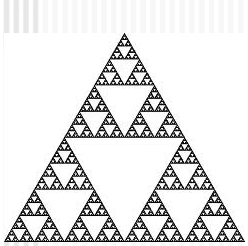
- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

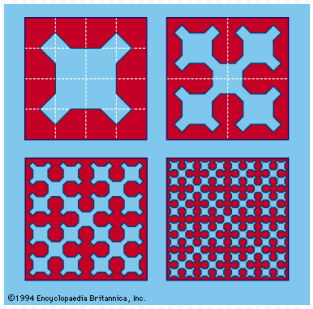
Summary

A box trace or a recursive call tree can be used to trace the execution of a recursive method.

Recursion can be used to solve problems whose iterative solutions are difficult to conceptualize.

Recursion using a Turtle





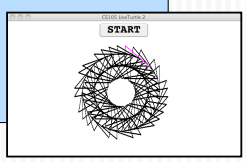
The Turtle class

Imagine a robotic turtle starting somewhere in the x-y plane and initially facing east. Give it the command turtle.move(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.turn(25), and it rotates in-place 25 degrees clockwise.

The turtle also has the pen up/pen down functionality so it can move without drawing.

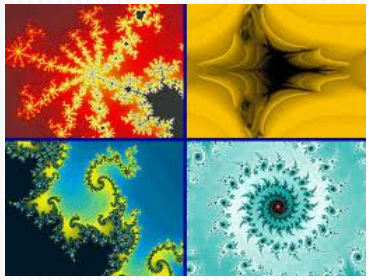
The Turtle class - Example

```
Turtle turtle = new Turtle(260, 230);
…
turtle.move(40);
turtle.turn(-60);
turtle.move(70);
turtle.turn(120);
turtle.move(100);
turtle.turn(-60);
turtle.move(50);
…
```



Fractals

A fractal is a shape that is self-similar: any small piece of it has the same general shape as the whole. This property is called self-similarity.



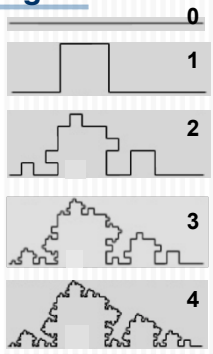
Some fractals can be drawn using the Turtle class.

Recursive Drawings using a Turtle

Draw a fractal with N recursion levels (order).

Base case: if the order is 0 draw a straight line.

Recursive case: draw a straight line, replace part with an angled line segments (forming a "bump"). Call this drawing method recursively with order (n-1) for each line segment of the shape.

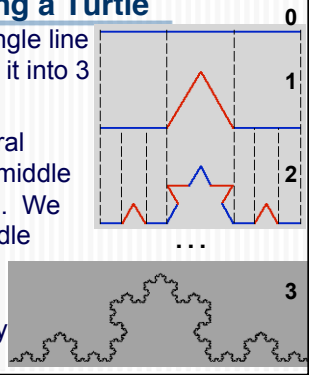


Koch Curve – using a Turtle

We start off with a single line segment, then divide it into 3 equal segments.

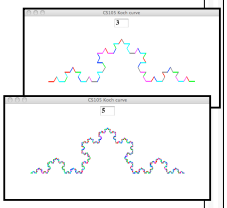
We draw an equilateral triangle that has the middle segment as the base. We then remove the middle segment.

We apply this procedure recursively on all line segments.



Koch Curve

```
private void drawKoch(Graphics g, int level,
                                        int steps) {
  if (level == 0) {
    turtle.move(steps);
    turtle.draw(g);
  } else {
    drawKoch(g, level-1, steps/3);
    turtle.turn(60);
    drawKoch(g, level-1, steps/3);
    turtle.turn(-120);
    drawKoch(g, level-1, steps/3);
    turtle.turn(60);
    drawKoch(g, level-1, steps/3);
  }
}
```



C Curves

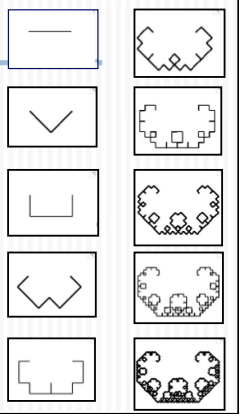
0: horizontal line.

1: bisect level 0 line and join two halves at right angles.

2: joining two level 1 C-curves at right angles

...

N: joining two level *N* – 1 C-curves at right angles



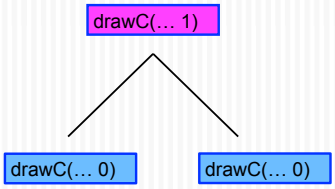
C Curve

```
1  private void drawC(Graphics g, int level, int steps) {
2    if (level == 0) {
3      turtle.move(steps);
4      turtle.draw(g);
5    } else {
6        turtle.turn(-45);
7        drawC(g, level-1, steps/Math.sqrt(2));
8        turtle.turn(90);
9        drawC(g, level-1, steps/Math.sqrt(2));
10       turtle.turn(-45);
11   }
12 }
```

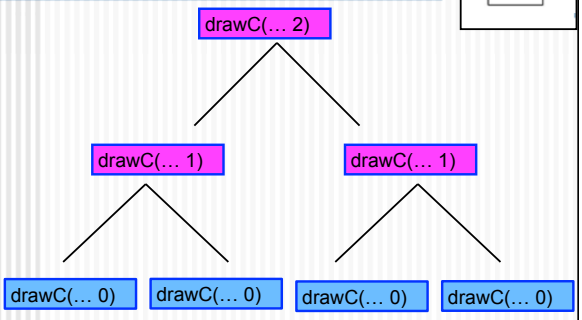

drawC(1) – Recursive call tree

drawC(... 0)

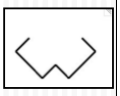
drawC(1) – Recursive call tree

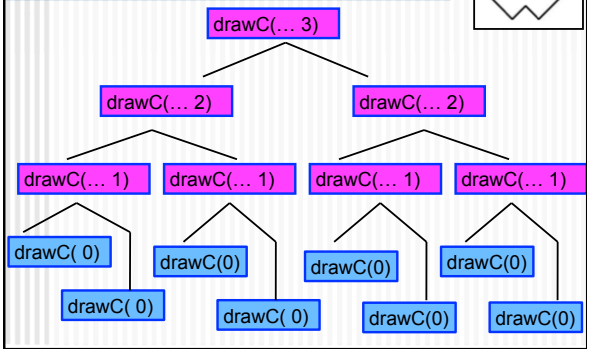


drawC(2) – Recursive call tree

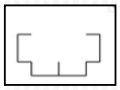


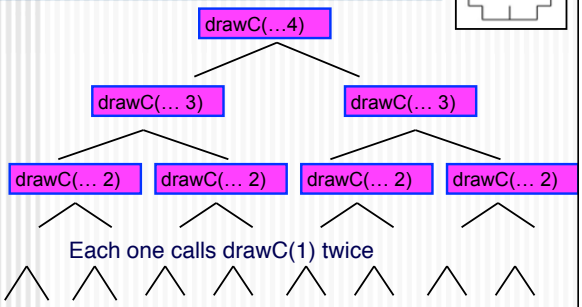
drawC(3) – Recursive call tree



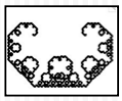


drawC(4) – Recursive call tree





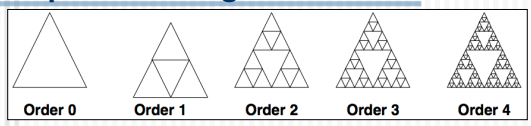
drawC(n) – Recursive call tree



etc/.

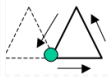
How many drawC(0) calls are made?

Sierpinski Triangle

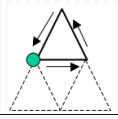


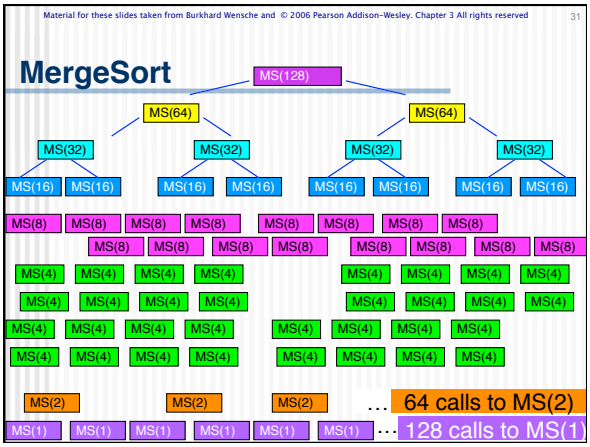
Draw a triangle starting at bottom left corner,

Move to the bottom right corner and repeat,



Move to top corner and repeat.




Material for these slides taken from Burkhard Wensche and © 2006 Pearson Addison-Wesley. Chapter 3 All rights reserved
31
MergeSort
MS(128)
MS(64)
MS(64)
MS(32)
MS(32)
MS(32)
MS(32)
MS(16)
MS(16)
MS(16)
MS(16)
MS(16)
MS(16)
MS(16)
MS(16)
MS(8)
MS(4)
MS(2)
MS(1)
64 calls to MS(2)
128 calls to MS(1)