Recursion with Mazes, Bunnies, and Towers

Classic recursion examples

Friday, October 27, 2006

CS111 Computer Programming

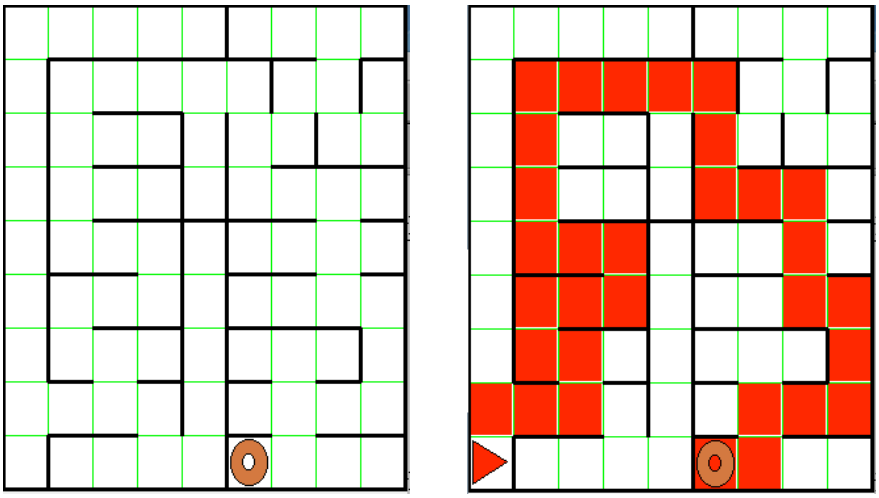
Department of Computer Science
Wellesley College

Today:

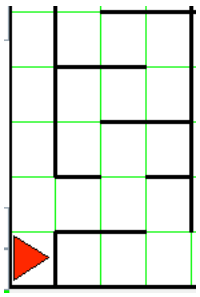
1. An example of mutual recursion in BuggleWorld: PathFinder.
2. Review of class (static) methods.
3. Some classic recursion examples that every computer scientist must know: factorial, fibonacci, and Towers of Hanoi.

PathFinder

Patty buggle wants to find a single bagel in an acyclic maze and draw a path from the bagel back to the origin.



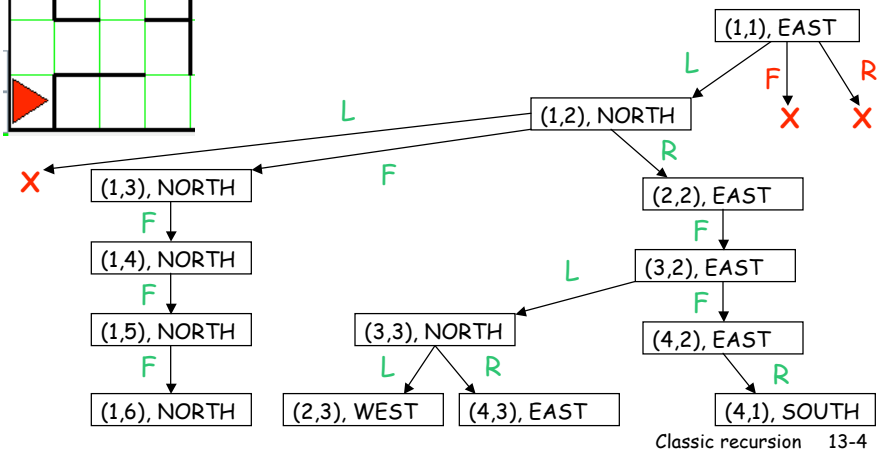
An Acyclic Maze Looks Like a Tree



L = left(); forward();

F = forward();

R = right(); forward();



findBagel() Specification

public boolean findBagel()

Assume the brush is initially up. Search for bagel in the submaze defined by cells reachable to the left, forward, and right.

If no bagel is in the submaze, go back to the current position and heading and return false.

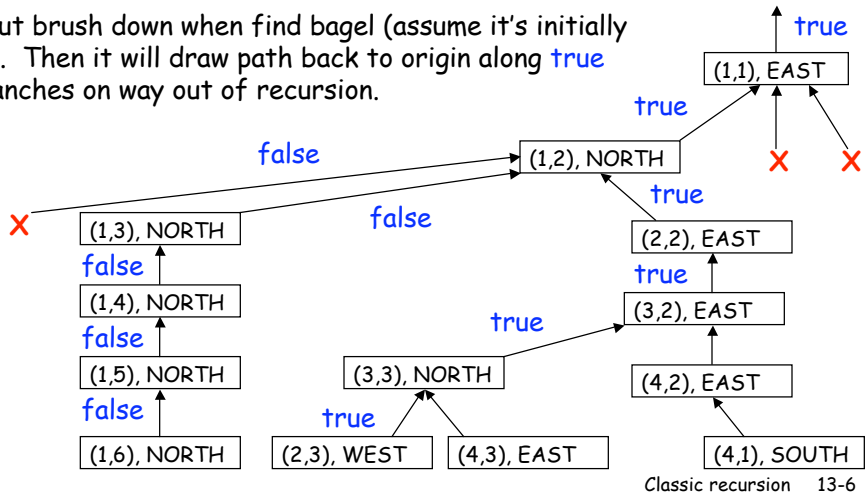
If there is a bagel in the submaze, draw a path from bagel's position when going back to the current position and heading (by putting the brush down at the bagel) and return true.

Sample use:

```
public void pathToBagel() { // Assume buggle is at (1,1)
  brushUp();  // Buggle will put brush down when finds bagel.
  if (findBagel()) {
     System.out.println("You found the bagel!");
   } else {
     System.out.println("Sorry, you didn't find the bagel.");
   }
  }
```


An Idea

- At each node, explore subtrees in order L, F, R.
- Return boolean indicating whether bagel's in subtree.
- Don't explore anymore once the bagel is found.
- Put brush down when find bagel (assume it's initially up). Then it will draw path back to origin along true branches on way out of recursion.



findBagel() implementation

```
public boolean findBagel() {
  // Base case: if there is a bagel, put down brush and return true
  if (isOverBagel()) {
     brushDown();
     return true;
  } else {
     // Recursive case: explore left, front, and right.
     // If a bagel is found in any of these directions,
     // return true immediately without further exploration.
     // Return false if no bagel found in any of the 3 directions.
     return findLeft() || findFront() || findRight();
     // Note: "or" operator (||)  is a "short-ciruit" operator
     // that returns true as soon as one operand is true,
     // without evaluating subsequence operands.
  }
}
```


findFront() and Friends

findFront() is like findBagel() except that it only finds a bagel in the submaze rooted at the cell in front of the buggle.

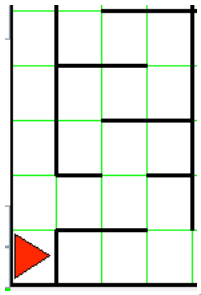
```
public boolean findFront() {
   if (isFacingWall()) {
      return false;
   } else {
      forward();
      boolean result = findBagel();
      backward();
      return result;
   }
}
```

findLeft() is like findBagel() except that it only finds a bagel in the submaze rooted at the cell to the left of the buggle.

(findRight() is similar.)

```
public boolean findLeft() {
   left();
   boolean result = findFront();
   right();
   return result;
}
```

PathFinder Illustrates Mutual Recursion



(Red nodes are never created)

Class (static) vs. Instance vs. Local Variables

o Class (static) variables reside in the class itself. There is exactly one occurrence of a class variable, regardless of the number of instances.

o Instance variables live in each instance of a class. There can be many distinct occurrences of an instance variable, one in each instance.

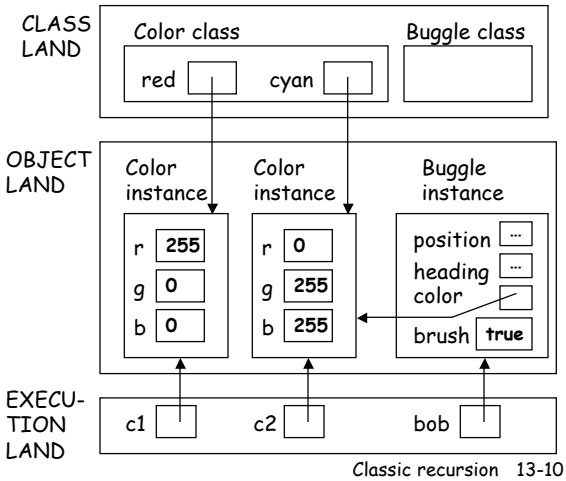
o Local variables reside in execution frames

```
public class Color {
   // class variables
  public static Color
    red, cyan, …;

   // instance variables
   private int r, g, b;

   … rest of Color class …}
```

```
Color c1 = Color.red;
Color c2 = Color.cyan;
Buggle bob = new Buggle();
bob.setColor(c2);
```



Class (static) vs. Instance Methods

o Instance methods are invoked on a receiver object (the contents of the this variable) and the argument values. They can refer to the state of the receiver and invoke other methods on the receiver via an explicit or implicit this variable.

o Class (static) methods do not have a receiver object. They are invoked on the class itself. Execution frames for static methods do *not* have a this variable. They correspond to functions/procedures in non-object-oriented programming languages.

Example

```
public class Thing {
  public static int a = 5; // class variable
  public int b = 0; // instance variable

  public Thing (int initb) { b = initb; } //constructor method

  public static void scale (int s) { a = a*s; } // class method

  public int f (int x) { return a*x + b; } // instance method

  public static void main (String[] args) {
    Thing thing1 = new Thing(1);
    Thing thing2 = new Thing(2);
    System.out.println("thing1.f(10)=" + thing1.f(10)
                       + "; thing2.f(10)=" + thing2.f(10));
    scale(2);
    System.out.println("thing1.f(10)=" + thing1.f(10)
                       + "; thing2.f(10)=" + thing2.f(10));
  }
}
```

Classic Recursion: Factorial

4! = 4 x (3 x 2 x 1)
= 4 x 3!

recursive cases

3! = 3 x (2 x 1)
= 3 x 2!

Likewise

2! = 2 x (1)
= 2 x 1!

1! = 1 X 0!

0! = 1

base case

Factorial in Java

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

So, if n==4, fact(4)

24

4*fact(3)

6

3*fact(2)

2

2*fact(1)

1

1*fact(0)

1

1

Invoking fact()

```
public class Factorial
{
     public static int fact (int n)
     {
          if (n == 0) {
               return 1;
          } else {
               return n * fact (n - 1);
          }
     }

     public static void main(String [] args)
     {
          int n = 4;
          System.out.println("The value of " +
                             n + "! is " + fact(n));
     }
}
```


Can we write fact() using tail recursion?

```
public static int factTail (int num, int ans)
{


   // factorial using tail recursion


}
```

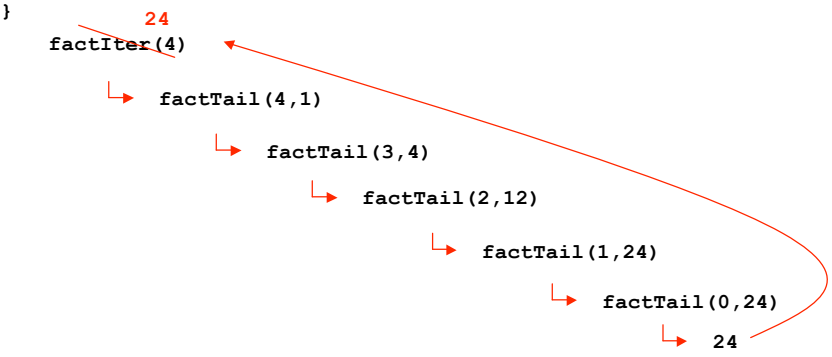
serves as a place to pass answer along

Invoking factTail() using seed for ans

```
public class Factorial
{
     public static int factTail (int num, int ans)
     {
          if (num == 0) return ans;
          else return factTail(num-1, num*ans);
     }
     public static int factIter(int n)
     {
          return factTail(n, 1);
     }
     public static void main(String [] args)
     {
          int n = 4;
          System.out.println("The value of " + n
                             + "! is " +
                             factIter(n));
     }
}
```

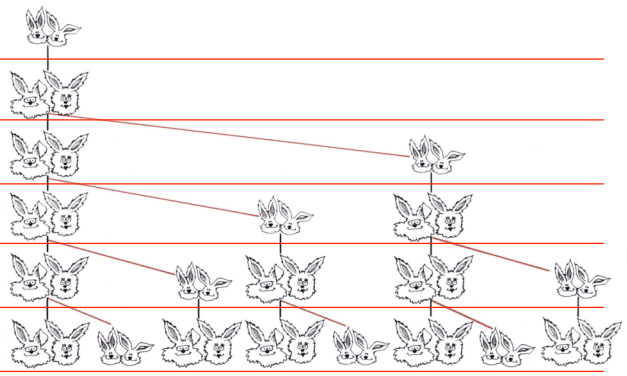

factIter(n) invokes factTail(n,1)

```
public static int factTail (int num, int ans)
{
     if (num == 0) {
          return ans;
     } else {
          return factTail(num-1, num*ans);
     }
}
```



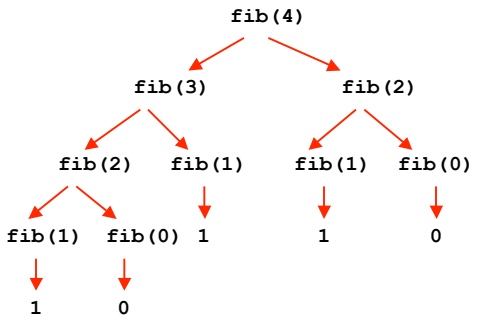
Leonardo Pisano Fibonacci

Time	# Pairs
1	1
2	1
3	2
4	3
5	5
6	8

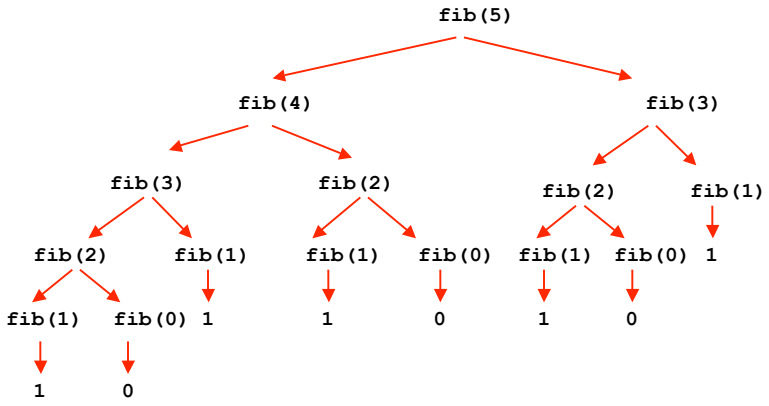


fibonacci() method

```
public int fibonacci(int n)
{
     if (n < 2) // Assume n >= 0
          return n;
     else
          return fibonacci(n-1) + fibonacci(n-2);
}
```



It gets worse



fibonacciFast(n) invokes fibTail(n,1,0,1);

```
public int fibTail(int n, int i, int fibIMinus1, int fibI)
{
     if (i == n) {
          return fibI;
     } else {
          return fibTail(n, i+1, fibI, fibIMinus1 + fibI);
     }
}
```

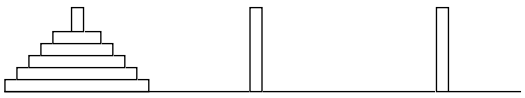
```
fibonacciFast(5)  5
    ↳ fibTail(5,1,0,1)
        ↳ fibTail(5,2,1,1)
            ↳ fibTail(5,3,1,2)
                ↳ fibTail(5,4,2,3)
                    ↳ fibTail(5,5,3,5)
                        ↳ 5
```

Java Fibonacci

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

public int fibTail(int n, int i, int fibIMinus1, int fibI)
{
     if (i == n) {
          return fibI;
     } else {
          return fibTail(n, i+1, fibI, fibIMinus1 + fibI);
     }
}
```


Tower of Hanoi



Java code

```
public void hanoi (int n)
{
     System.out.println("-----------------------------------");
     System.out.println("Instructions for solving Hanoi("
                        + n + ", A, B, C):”);
     Hanoi(n, "A”, "B”, "C”);
 }
public void hanoi (int n,String source,String dest,String spare)
{
     if (n > 0) {
          hanoi(n - 1, source, spare, dest); // n-1 disks->spare
          moveDisk(n, source, dest);         // largest disk->goal
          hanoi(n - 1, spare, dest, source); // n-1 disks->goal
     }
 }
```