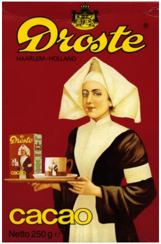
Recursion

CS284



Structure of this week's classes

What is Recursion?

More Examples

Lists

Problem Solving with Recursion

Recursion (in Programming)

- The self-referring condition of some datatypes whereby a data element can be decomposed into “smaller” ones of a “similar” nature
- The self-referring condition of some algorithms whereby a programming problem can be decomposed into “smaller” ones of a “similar” nature

Recursive Datatypes

The self-referring condition of some datatypes whereby an element can be decomposed into “smaller” ones of a “similar” nature

- Natural Numbers N:
  - $0 \in N$
  - $1 + n \in N$ if $n \in N$
- Lists over set A: $List_A$
  - $[] \in List_A$
  - $a :: l \in List_A$ if $a \in A$ and $l \in List_A$
- Trees: We’ll study them later

Recursive Programs

The self-referring condition of some algorithms whereby a problem can be decomposed into "smaller" ones of a "similar" nature

- Computing the size of a list l
  - If it is empty, return 0
  - If not, compute the size of l without the head element and add 1
- Computing the factorial of a number n
  - If it is zero, return 1
  - If not, compute the factorial of $n - 1$ and multiply by n

Lets take a closer look at the second example

Factorial – Mathematically

$$
\begin{aligned}
0! &\stackrel{def}{=} 1 \\
n! &\stackrel{def}{=} n*!(n-1),\ \ n > 0
\end{aligned}
$$

- The first clause is the base case
- The second clause is the recursive case

$$
\begin{aligned}
5! &= 5*4! \\
&= 5*4*3! \\
&= 5*4*3*2! \\
&= 5*4*3*2*1! \\
&= 5*4*3*2*1*0! \\
&= 5*4*3*2*1*1 \\
&= 120
\end{aligned}
$$

Factorial – Java

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

- Consider `factorial(4)`
- We follow its execution by tracing each recursive call

Stacks and Calls

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

- On the board: `factorial(4)`

Infinite Recursion and Stack Overflow

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

- What happens if we execute `factorial(-2)`?

Infinite Recursion and Stack Overflow

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

- What happens if we execute `factorial(-2)`?
- `Exception in thread "main"java.lang.StackOverflowError`

Some Questions

What's wrong with this program?

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

What about this one?

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

Tail Recursion

- Only one recursive call
- It is the last instruction performed

```java
public static int factorialAux(int n, int a) {
  if (n == 0)
     return a;
  else
     return factorial(n-1, n*a);
}

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

Computing Factorial Iteratively (i.e. without recursion)

```
public static int factorial_it(int n) {
  int r = 1;
  for (int i=1; i<n+1; i++) {
      r = r * i;
  }
  return r;
}
```

The above code can be obtained automatically from the tail recursive version:

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

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

Iteration vs Recursion

- Recursive methods often have slower execution times relative to their iterative counterparts
  - Modern optimizing compilers make this difference often imperceptible
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency does not outweigh the advantage of readable code that is easy to debug

Fibonacci - In Maths

The Fibonacci numbers are a sequence defined as follows

$$
\begin{aligned}
fib(0) &\stackrel{def}{=} 1 \\
fib(1) &\stackrel{def}{=} 1 \\
fib(n) &\stackrel{def}{=} fib(n-1) + fib(n-2), n > 1
\end{aligned}
$$

$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Fibonacci - Implemented as a Recursive Program

```java
public static int fibonacci(int n)
{
  if (n<=1)
      return 1;
  else
      return fibonacci(n-1) + fibonacci(n-2);
}
```

Efficiency of `fibonacci`

What is the complexity of `fibonacci(n)`?

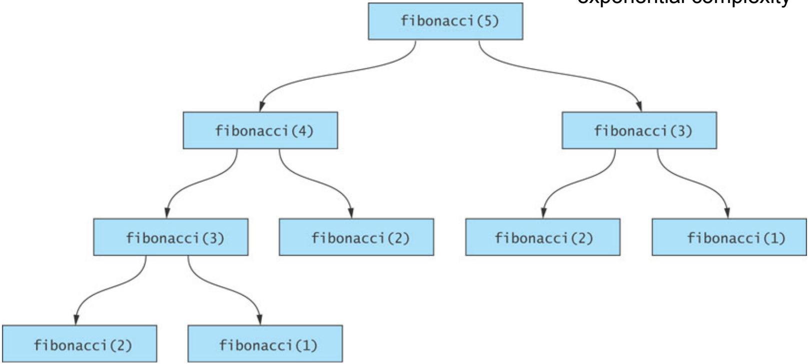
- Let's draw a picture of the trace of execution of `fibonacci(5)`

Efficiency of `fibonacci`

What is the complexity of `fibonacci(n)`?

- Let's draw a picture of the trace of execution of `fibonacci(5)`

Inefficient:
exponential complexity



- How can we do better?

Efficient fibonacci

```
private static int ffib(int prevFibo, int currentFibo, int n)
{
  if (n==0)
    return currentFibo;
   else
    return ffib(currentFibo, prevFibo+currentFibo, n-1);
}

public static int ffibonacciStart(int n) {
     return ffib(0, 1, n);
}
```

What is the complexity of `ffibonacciStart(n)`?

- Let's draw a picture of the trace of execution of `ffibonacciStart(5)`

Efficient fibonacci

- Method fibo is an example of tail recursion or last-line recursion
- When recursive call is the last line of the method, arguments and local variables do not need to be saved in the activation frame
- They can be easily implemented using iteration

Lists

- Lists are examples of recursive data structures, as already mentioned
  - A list is either empty or a head node followed by the rest of the list (its tail)
- Let's see some examples of methods for Linked Lists implemented using recursion
- We'll define a class `LinkedListRec<E>`

LinkedListRec<E>

```java
public class LinkedListRec<E> {
  private Node<E> head;

  // inner class Node<E> here
  // (from lecture on linked lists)

 private int size(Node<E> head) {
        if (head == null) {
            return 0;
        } else {
            return 1 + size(head.next);
        }
    }
}

 public int size() {
        return size(head);
    }
```

```java
private String toString(Node<E> head) {
     if (head == null) {
         return "";
     } else {
         return head.data + "\n" + toString(head.next);
     }
  }

@Override
public String toString() {
    return toString(head);
}
```

```
private void replace(Node<E> head, E oldObj, E newObj) {
        if (head != null) {
            if (oldObj.equals(head.data)) {
                head.data = newObj;
            }
            replace(head.next, oldObj, newObj);
        }
    }

public void replace(E oldObj, E newObj) {
        replace(head, oldObj, newObj);
}
```

LinkedListRec<E>

```java
private void add(Node<E> head, E data) {
      // If the list has just one element, add to it.
      if (head.next == null) {
          head.next = new Node<E>(data);
      } else {
          add(head.next, data); // Add to rest of list.
      }
  }

public void add(E data) {
      if (head == null) {
          head = new Node<E>(data); // List has 1 node.
      } else {
          add(head, data);
      }
}
```

```java
private boolean remove(Node<E> head, Node<E> pred, E outData) {
        if (head == null) // Base case --  empty list.
        { return false; }
        else if (head.data.equals(outData)) { // 2nd base case.
            pred.next = head.next; // Remove head.
            return true;
        } else {
            return remove(head.next, head, outData);
        }
}
public boolean remove(E outData) {
        if (head == null) {
            return false;
        } else if (head.data.equals(outData)) {
            head = head.next;
            return true;
        } else {
            return remove(head.next, head, outData);
        }
}
```

Recursive Search in an Ordered List

- Searching an array can be accomplished using recursion
- Simplest way to search is a linear search
  - Examine one element at a time starting with the first element and ending with the last
  - On average, $(n+1)/2$ elements are examined to find the target in a linear search If the target is not in the list, n elements are examined
- A linear search is $\mathcal{O}(n)$

Recursive Search in an Ordered List

- Base cases for recursive search:
  - Empty array, target can not be found; result is -1
  - First element of the array being searched = target; result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Algorithm for Recursive Linear Array Search

- if the array is empty the result is -1
- else if the first element matches the target the result is the subscript of the first element
- else search the array excluding the first element and return the result

Algorithm for Recursive Linear Array Search

```java
private static int linearSearch(E[] items, E target, int posFirst) {
    if (posFirst == items.length) {
        return -1;
    } else if (target.equals(items[posFirst])) {
        return posFirst;
    } else {
        return linearSearch(items, target, posFirst+1);
    }
}

public static int linearSearch(E[] items, E target) {
    return linearSearch(items, target, 0);
}
```

Design of a Binary Search Algorithm

- A binary search can be performed only on an array that has been sorted
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- A binary search excludes the half of the array within which the target cannot lie
- Base cases?

Design of a Binary Search Algorithm

- A binary search can be performed only on an array that has been sorted
- Rather than looking at the first element, a binary search compares the middle element for a match with the target
- A binary search excludes the half of the array within which the target cannot lie
- Base cases?
  - The array is empty
  - Element being examined matches the target

Design of a Binary Search Algorithm

- if the array is empty
  - return -1 as the search result
- else if the middle element matches the target
  - return the subscript of the middle element as the result
- else if the target is less than the middle element
  - recursively search the array elements before the middle element and return the result
- else
  - recursively search the array elements after the middle element and return the result

Binary Search in an Ordered List – An Example

- Target: Dustin

Caryn	Debbie	Dustin	Elliot	Jacquie	Jonathan	Rich
0	1	2	3	4	5	6

- Initial boundaries of "subarray" to search:
  - The "interval" [first=0,last=6]
  - That is, the entire array

Efficiency of Binary Search

- At each recursive call we eliminate half the array elements from consideration, making a binary search $\mathcal{O}(\log n)$
- An array of 16 would search arrays of length 16, 8, 4, 2, and 1; 5 probes in the worst case
  - $16 = 24$
  - $5 = \log_2 16 + 1$
- A doubled array size would only require 6 probes in the worst case
  - $32 = 25$
  - $6 = \log_2 32 + 1$
- An array with 32,768 elements requires only 16 probes! $(\log_2 32768 = 15)$

Implementation of a Binary Search Algorithm

- Classes that implement the `Comparable` interface must define a `compareTo` method
- Method `obj1.compareTo(obj2)` returns an integer with the following values
  - negative: obj1 $<$ obj2
  - zero: obj1 $==$ obj2
  - positive: obj1 $>$ obj2
- Implementing the `Comparable` interface is an efficient way to compare objects during a search

Implementation of a Binary Search Algorithm

```
private static int binSearch(E[] items, Comparable<E> target, int firs
 if (first > last) {
    return -1; // Base case for unsuccessful search.
 } else {
    int middle = (first+last)/2; // Next probe index
    int compResult = target.compareTo(items[middle]);
    if (compResult == 0) {
        return middle; // Base case for succ. search
     } else if (compResult < 0) {
        return binSearch(items, target, first,middle-1);
    } else {
        return binSearch(items, target, middle+1,last);
    }
 }}}

public static int binSearch(E[] items, Comparable<E> target) {
  return binSearch(items, target, 0, items.length - 1); }
```

Towers of Hanoi

- Move the three disks to a different peg, maintaining their order (largest disk on bottom, smallest on top, etc.)
- Only the top disk on a peg can be moved to another peg
- A larger disk cannot be placed on top of a smaller disk

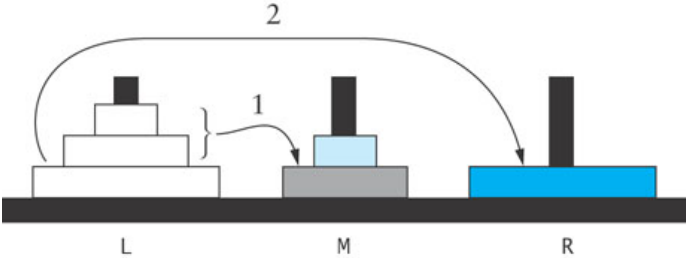
Towers of Hanoi

- Problem input:
  - Number of disks
  - Starting peg
  - Destination peg
  - Temporary peg
- Problem output:
- List of moves

Algorithm for Towers of Hanoi

Solution to Three-Disk Problem: Move Three Disks from Peg L to Peg R

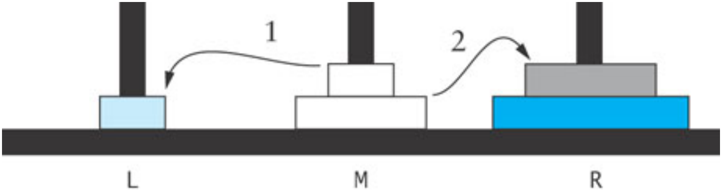
1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.



Algorithm for Towers of Hanoi

Solution to Two-Disk Problem: Move Top Two Disks from Peg M to Peg R

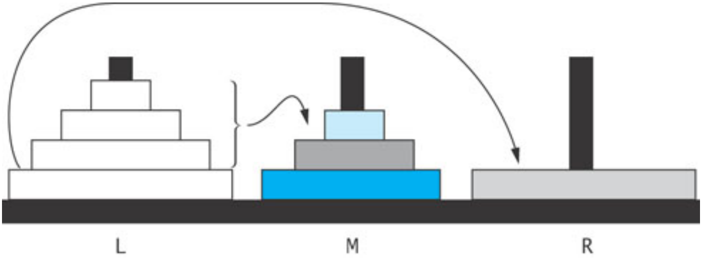
1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.



Algorithm for Towers of Hanoi

Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.



Recursive Algorithm for Towers of Hanoi – n-Disk Problem

Move n Disks from the Starting Peg to the Destination Peg

- if n is 1
  1. move disk 1 (the smallest disk) from the starting peg to the destination peg
- else
  1. move the top $n-1$ disks from the starting peg to the temporary peg (neither starting nor destination peg)
  2. move disk n (the disk at the bottom) from the starting peg to the destination peg
  3. move the top $n-1$ disks from the temporary peg to the destination peg

Java Code

```
public class TowersOfHanoi {
   public static String showMoves(int n, char startPeg, char destPeg,
   tempPeg) {

     if (n==1) { // Base case
         return "Move disk 1 from peg " + startPeg
              + " to peg " + destPeg + "\n";
    } else {        // Recursive case
         return showMoves(n-1,startPeg,tempPeg, destPeg)
           + "Move peg " + n + " from peg " + startPeg
           + " to peg " + destPeg + "\n "
           + showMoves(n-1, tempPeg, destPeg, startPeg);
 }
}
}
```

4 disks, (S)ource, (D)estination, (T)emporary

```
Move disk 1 from peg S to peg T
Move peg 2 from peg S to peg D
 Move disk 1 from peg T to peg D
Move peg 3 from peg S to peg T
 Move disk 1 from peg D to peg S
Move peg 2 from peg D to peg T
 Move disk 1 from peg S to peg T
Move peg 4 from peg S to peg D
 Move disk 1 from peg T to peg D
Move peg 2 from peg T to peg S
 Move disk 1 from peg D to peg S
Move peg 3 from peg T to peg D
 Move disk 1 from peg S to peg T
Move peg 2 from peg S to peg D
 Move disk 1 from peg T to peg D
```

Counting Cells in a Blob

- Consider how we might process an image that is presented as a two-dimensional array of color values
- Information in the image may come from
  - an X-ray
  - an MRI
  - satellite imagery
  - etc.
- The goal is to determine the size of any area in the image that is considered abnormal because of its color values

Counting Cells in a Blob

- Given a two-dimensional grid of cells, each cell contains
  - either a normal background color (say white) or
  - a second color (say red), which indicates the presence of an abnormality
- A blob is a collection of contiguous abnormal cells
- A user will enter the x, y coordinates of a cell in the blob, and the program will determine the count of all cells in that blob

Problem Inputs and Outputs

- Problem Inputs
  - the two-dimensional grid of cells
  - the coordinates of a cell in a blob
- Problem Outputs
  - the count of cells in the blob

Algorithm for `countCells(x, y)`

- if the cell at (x, y) is outside the grid
  - the result is 0
- else if the color of the cell at (x, y) is not the abnormal color
  - the result is 0
- else
  - set the color of the cell at (x, y) to a temporary color
  - the result is 1 plus the number of cells in each piece of the blob that includes a nearest neighbor

Implementation

```java
public class Blob implements GridColors {

    /** The grid */
    private TwoDimGrid grid;

    /** Constructors */
    public Blob(TwoDimGrid grid) {
        this.grid = grid;
    }
```

- `GridColors` is an interface that simply assigns constants to colors
- We'll see it in the next slide

The `GridColors` Interface

```java
import java.awt.Color;

/**
 *  An interface for colors
 *@author Koffman and Wolfgang
 */
public interface GridColors {

    Color PATH = Color.green;
    Color BACKGROUND = Color.white;
    Color NON_BACKGROUND = Color.red;
    Color ABNORMAL = NON_BACKGROUND;
    Color TEMPORARY = Color.black;
}
```

- The `PATH` constant is not used in this example; it is for the maze

Implementation

```java
public int countCells(int x, int y) {
  int result;

  if (x < 0 || x >= grid.getNCols()
          || y < 0 || y >= grid.getNRows()) {
      return 0;
  } else if (!grid.getColor(x, y).equals(ABNORMAL)) {
      return 0;
  } else {
      grid.recolor(x, y, TEMPORARY);
      return 1
        + countCells(x - 1, y + 1) + countCells(x, y + 1)
        + countCells(x + 1, y + 1) + countCells(x - 1, y)
        + countCells(x + 1, y) + countCells(x - 1, y - 1)
        + countCells(x, y - 1) + countCells(x + 1, y - 1);
  }
 }
```