Recursion

Professor Frank Kreimendahl

School of Computing and Data Science
Wentworth Institute of Technology

October 5, 2022

Recursion

Introduction

- Recursion can solve programming problems that are tough to solve linearly
- Recursion is a staple in many AI applications:
  - playing strategy games
  - proving math properties
  - pattern recognition
  - considering multiple branches in situations with many decisions
- Recursion can also replace iterative loops and solve linear problems:
  - Compute factorials
  - Process data structures – strings, lists, etc.
  - search through an array for a specific value

Recursive Thinking

- Recursion is a problem-solving approach that can solve some problems with a small amount of code
- Recursion decomposes a problem into one or more simpler/smaller versions of itself

Nesting dolls – each doll has a smaller subdoll except the innermost, smallest doll.

Recursive Thinking

Recursive algorithm for processing nested figures. ‘Processing’ may entail gathering information or modifying dolls in some way.

```
if we are at the innermost doll then
    do whatever work we need to the current doll
else
    do whatever work we need to the current doll
    process the dolls inside the current doll
```

Recursion Example

Consider searching for a target value in a sorted array:

- Compare the target value to the middle value in the array
- If we didn't find the target, we only have to look on one side of the middle
- How do we search one side? Run a search with the same target, on the half our target might appear in

Recursion Example

Recursive algorithm to search a sorted array. Returns an index that the target appears at:

```
if array range is empty then
    return -1 // target not in array
else if the middle element matches the target then
    return the index of the middle element
else if target < middle element then
    Consider an array that is the first half of the original array
    return result of a search on that smaller array
else
    Consider an array that is the second half of the original array
    return result of search on that smaller array
```

Recursive Method Design

- Identify a base case – the simplest problem of that type that can be solved directly
- Identify a way of slightly reducing a problem size, progressing towards a base case
- Problem size reductions often include:
  - Decreasing an array range by 1
  - Decreasing a list length by 1
  - Splitting a range into left and right halves
- Identify what work should be done to the current array/list/data in recursive cases

String Operations

String Length

```
if string is empty then
    the length is 0
else
    the length is 1 + the length of the string starting after first letter
```

String Length

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
  if (str == null || str.equals(""))
    return 0;
  else
    return 1 + length(str.substring(1));
}
```

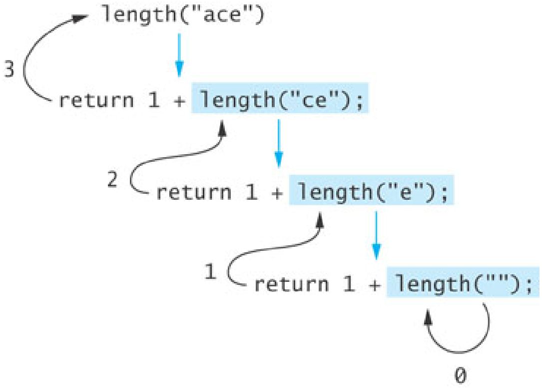
Printing a String

```
/** Recursive method printChars
  post: The argument string is displayed,
    one character per line
  @param str The string
*/
public static void printChars(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    System.out.println(str.charAt(0));
    printChars(str.substring(1));
  }
}
```

Printing a String in Reverse

```
/** Recursive method printCharsReverse
  post: The argument string is displayed in
    reverse, one character per line
  @param str The string
*/
public static void printCharsReverse(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    printCharsReverse(str.substring(1));
    System.out.println(str.charAt(0));
  }
}
```

Tracing Recursion



```
length("ace")
return 1 + length("ce");
return 1 + length("e");
return 1 + length("");
```

Showing initial calls (down the right side) and returned values (up the left side)

Java’s Runtime Stack

- When a Java program runs, it maintains a stack called a *runtime stack*
- The stack stores *activation record* objects – one object for each method that is currently running
- An activation record contains:
  - method arguments
  - local variables
  - the method to return to
- Whenever a method is called, Java pushes a new activation record onto the stack
- Whenever a method returns, Java pops the top activation record from the stack

Recursion in Math

Formulas

- Many math functions can be defined recursively
- Examples include:
    - Factorial
    - Powers
    - Greatest Common Divisor

Factorial

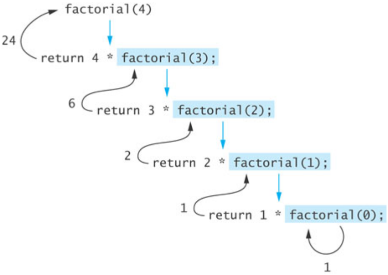
Factorial is defined as:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)!, & \text{n>0} \end{cases}$$

- $n = 0$ is the base case
- $n > 0$ is the recursive case – note the $(n-1)!$ call to the same function in the definition

Factorial

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



Infinite Recursion

- A `factorial` call with a negative argument will never terminate
- `n` never reaches 0
- Each recursive call generates a new activation record
- Infinite calls require infinite memory, which is an issue
- When the runtime stack is full, Java throws a `StackOverflowError`

Calculating x^n for $n \geq 0$

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x * x^{n-1}, & \text{otherwise} \end{cases}$$

```
/** Recursive power method
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```

Greatest Common Divisor

- The greatest common divisor of two numbers is the largest integer that divides both numbers
- gcd(20,15) = 5
- gcd(36,24) = 12
- gcd(18,38) = 2
- gcd(17,97) = 1

Greatest Common Divisor

$$gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

```
/** Recursive gcd method
    @param m First number
    @param n Second number
    @return gcd(m, n)
*/
public static int gcd(int m, int n) {
  if (n == 0)
    return m;
  else
    return gcd(n, m % n);
}
```

Iteration vs. Recursion

- There are similarities between iterative loops and recursion
- In iteration, a condition determines when to terminate the loop
- In recursion, a base case determines when to stop recursive calls
- The loop condition in iteration often corresponds to the base case in recursion
- The loop variable often corresponds to a parameter of the recursive method

Fibonacci Numbers

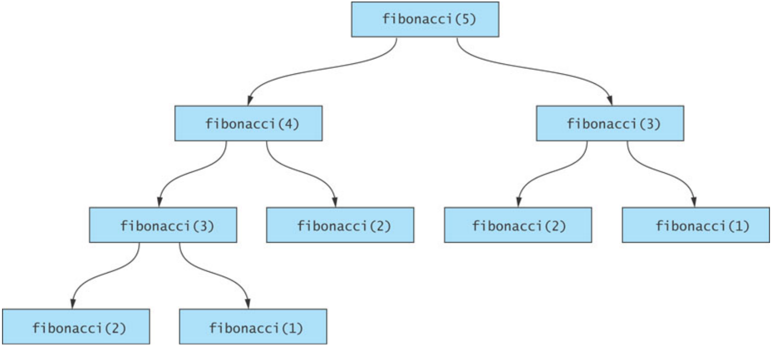
- The Fibonacci sequence was described as an interesting sequence of numbers
- The sequence has a relationship with the Golden ratio, Pascal's triangle, plant growth, etc.

$$\begin{aligned}\mathrm{fib}_0 &= 1\\ \mathrm{fib}_1 &= 1\\ \mathrm{fib}_n &= \mathrm{fib}_{n-1} + \mathrm{fib}_{n-2}\end{aligned}$$

Fibonacci Numbers

```
/** Recursive Fibonacci method
    @param n The index of the sequence
    @return The Fibonacci number
*/
public static int fibonacci(int n) {
  if (n <= 1)
    return 1;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Fibonacci Recursive Call Tree



Visualizing which arguments `fibonacci` is recursively called with

Recursive Search

Linear Search

- Problem: Search an array for a target value
- Solution: Compare each element to the target; stop when a match is found or continue until all elements have been compared
- Rather than a loop, use a recursive approach
- Base cases:
  - Empty array: return -1
  - First element of array range matches target: return index
- Recursive case: search the array except the first element in the current range

Linear Search

```
/** Recursive linear search
    @param items The array being searched
    @param target The item being searched for
    @param pos The position of the current first element
    @return The index of target in the array or -1
*/
public static int linSearch(Object[] items,
                            Object target, int pos) {
  if (pos == items.length)
    return -1;
  else if (target.equals(items[pos]))
    return pos;
  else
    return linSearch(items, target, pos + 1);
}
```

Wrapper Method

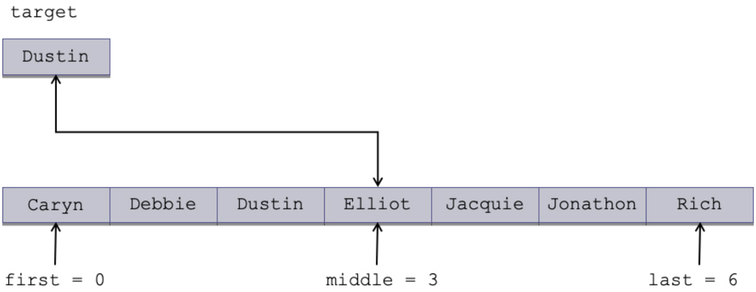
A common companion to recursive methods: a wrapper method with fewer parameters that calls the recursive method with beginning values.

```
/** Wrapper for recursive linear search
    @param items The array being searched
    @param target The item being searched for
    @return The index of target in the array or -1
*/
public static int linSearch(Object[] items,
                            Object target) {
  return linSearch(items, target, 0);
}
```

Binary Search

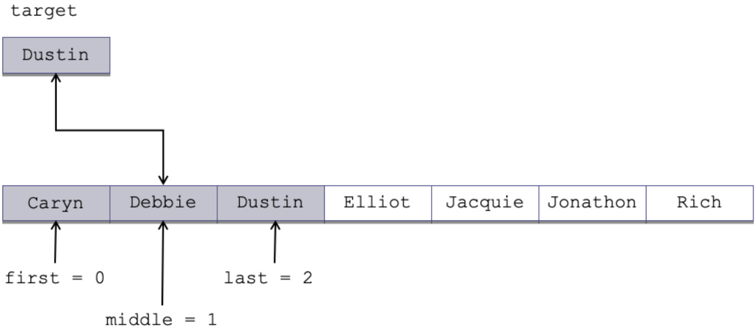
- Problem: Search a **sorted** array for a target value
- Solution: Compare a middle element to the target; stop when a match is found or continue until all elements have been compared
- Rather than a loop, use a recursive approach
- Base cases:
  - Empty array: return -1
  - Middle element of array range matches target: return index
- Recursive case: search one half of the array range, based on a comparison between middle element and target

Binary Search Example



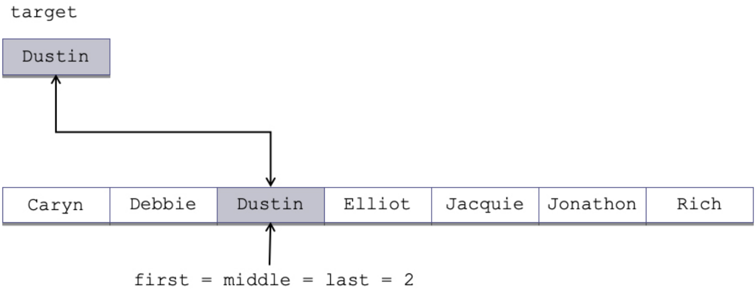
The target might appear anywhere in the array

Binary Search Example



The target can only appear in the left half of the array

Binary Search Example



Base case: the target is found

Binary Search

```
/**
 * Recursive binary search method.
 * @param <T> The type of items being searched
 * @param items The array being searched
 * @param target The object being searched for
 * @param first The subscript of the first element
 * @param last The subscript of the last element
 * @return The subscript of target if found; otherwise -1.
 */
 private static <T> int binSearch(T[] items, Comparable<T> target,
        int first, int last) {
    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 successful search.
        } else if (compResult < 0) {
            return binSearch(items, target, first, middle - 1);
        } else {
            return binSearch(items, target, middle + 1, last);
        }
    }
}
```

Wrapper Method

```
/** Wrapper for recursive binary search
    @param items The array being searched
    @param target The item being searched for
    @return The index of target in the array or -1
*/
public static int binSearch(Object[] items,
                            Object target) {
  return binSearch(items, target, 0,
                   items.length - 1);
}
```

Data Structures

Recursive Data Structures

- Many data structures can be defined recursively
- Linked lists and binary trees have simple recursive definitions
- Recursive methods can perform operations on recursive data structures
- Functional languages like LISP make heavy use of recursion and recursive data structures

Linked List

- A linked list is a collection of nodes with a base case and recursive case
- Base case: the list is empty
- Recursive case: the list has a head node which references a (potentially empty) list of nodes after it
- Every list is either an empty list or a head node (at the front of the list) followed by a linked list

Recursive size

```
/**
 * Finds the size of a list.
 * @param head The head of the current list
 * @return The size of the current list
 */
private int size(Node<E> head) {
    if (head == null) {
        return 0;
    } else {
        return 1 + size(head.next);
    }
}
```

Size Wrapper Method

```
/**
 * Wrapper method for finding the size of a list.
 * @return The size of the list
 */
public int size() {
    return size(head);
}
```

Recursive toString

```
/**
 * Returns the string representation of a list.
 * @param head The head of the current list
 * @return The state of the current list
 */
private String toString(Node<E> head) {
    if (head == null) {
        return "";
    } else {
        return head.data + "\n" + toString(head.next);
    }
}
```

toString Wrapper Method

```
/**
 * Wrapper method to return the string representation.
 * @return The string representation of the list
 */
@Override
public String toString() {
    return toString(head);
}
```

Recursive add

```
/**
 * Adds a new node to the end of a list.
 * @param head The head of the current list
 * @param data The data for the new node
 */
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<>(data);
    } else {
        add(head.next, data); // Add to rest of list.
    }
}
```

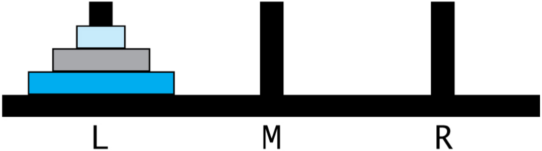
Add Wrapper Method

```
/**
 * Wrapper method for adding a new node to the end
 *   of a list.
 * @param data The data for the new node
 */
public void add(E data) {
    if (head == null) {
        head = new Node<>(data); // List has 1 node.
    } else {
        add(head, data);
    }
}
```

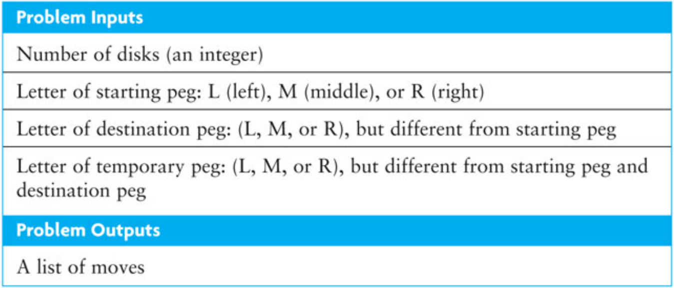
Towers of Hanoi

Problem Description

- Goal: Move all disks from one tower to another
- Rule: Only one disk can move at a time, from one tower to another
- Rule: A disk may only be moved onto a larger disk



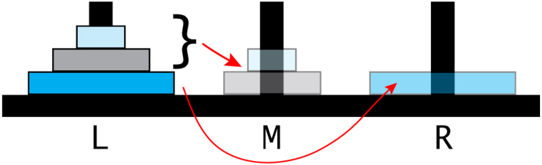
Input/Output



Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg: (L, M, or R), but different from starting peg
Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg

Problem Outputs
A list of moves

3 Disks



The largest disk must be moved from the start to destination, so we will somehow move the other disks to the temporary tower.

General Case

- Base case: solve a tower with one disk – move it from start tower to destination tower
- Recursive case:
    1. Move all disks except the largest from start tower to temporary tower
    2. Move the largest disk from start to destination tower
    3. Move all the smaller disks from the temporary tower to the destination tower
- How can we move all the smaller disks? By solving a smaller Towers of Hanoi problem!

Solution Code

```
/**
 * Recursive method for "moving" disks.
 * @pre startPeg, destPeg, tempPeg are different.
 * @param n is the number of disks
 * @return A string with all the required disk moves
 */
public static String showMoves(int n, char startPeg,
        char destPeg, char tempPeg) {
    if (n == 1) {     // base case
        return "Move disk 1 from peg " + startPeg
                + " to peg " + destPeg + "\n";
    } else {          // recursive case
        String ret = showMoves(n - 1, startPeg, tempPeg, destPeg)
        ret += "Move disk " + n + " from peg " + startPeg
                + " to peg " + destPeg + "\n"
        ret += showMoves(n - 1, tempPeg, destPeg, startPeg);
        return ret;
    }
}
```