Recursion

- Recursion
- Recursion
Recursion

- *Recursion* specifies (or constructs) a class of objects or methods (or an object from a certain class) by defining
  - a few very simple *base cases* or methods (often just one)
  - rules to break down complex cases into simpler cases
- Here is another, perhaps simpler way to understand recursive processes:
  - Are we done yet? If so, return the results. Without such a *termination condition* a recursion would go on forever.
  - If not, *simplify* the problem (move *towards* a base case), solve the simpler problem(s), and assemble the results into a solution for the original problem. Then return that solution.

- "*In order to understand recursion, one must first understand recursion.*"
- Recursion: If you still don’t get it, see Recursion
Classic Recursion: \( n! \)

- **Warning:** This is a simple illustration of the *concept* of recursion. It is easy to visualize, but it is not intended to be an example of a *good* use of recursion.

```java
public int factorial (int n) {
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product = product * i;
    }
    return product;
} // end factorial
```

**Iterative approach**

```java
public int factorial (int n) {
    if (n <= 1) { // base case
        return 1;
    } else { // recursive step
        return (n * factorial (n-1));
    }
} // end factorial
```

**Recursive approach**
Memory usage for iterative n!

- Iterative implementation requires:
  - storage of 3 local/stack variables
  - \( n \) multiplications
- Considering calling factorial(5)
  - One stack frame
- Overall analysis
  - Memory: \( O(3) \)
  - Time: \( O(n) \)

```java
public int factorial (int n) {
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product = product * i;
    }
    return product;
} // end factorial
```

Returns:

- \( n \)
- \( i \)
- \( product \)
Memory usage for recursive n!

- Consider calling factorial(5)
- Recursive implementation requires:
  - storage of 1 local/stack variables per frame
  - 1 multiplication per call
- Overall analysis?
  - Memory: \( O(n) \)
  - Time: \( O(n) \)

```java
public int factorial (int n) {
    if (n <= 1) {  // base case
        return 1;
    } else {       // recursive step
        return (n * factorial (n-1));
    }
} // end factorial
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursion: The basics

- A recursive method is one that calls itself
  - Recursive methods use a divide and conquer decomposition where one or more of the decomposed subcases are a smaller example of the main case
  - If this continues infinitely, then you’ll get a stack overflow
  - Thus recursive methods must also have one or more base cases in which they do not need to call themselves to provide a solution

- Example: Searching for a name in the phone book
  - Start at middle
  - Are you one the right page?
  - Decide direction, ignore the portion of the book in the wrong direction
  - Repeat

- What is the base case? What is the general decomposition?
Iterative paradigm

```java
public class Main {

    static int getIndexOfNumber (int searchNumber, int[] sortedList) {
        for (int i = 0; i < sortedList.length; i++ ){
            if (sortedList[i] == searchNumber ) {
                return i;
            }
        }
        return -1;  // searchNumber not found in list
    } // end method getIndexOfNumber

    public static void main (String[] args) {
        int[] sortedNumberList = { 2, 15, 35, 67, 90, 102, 154, 256, 400, 900, 1024, 1300, 1301, 2000, 2005, 2006};
        System.out.println(getIndexOfNumber(2000,sortedNumberList));
    } // end method main
} // end class Main
```

- Expected number of comparisons
  - Number in list: 8
  - Number not in list: 16

In general:
- Number in list: $O(n/2)$
- Number not in list: $O(n)$
Iterative paradigm, improved

```java
static int getIndexOfNumber (int searchNumber, int[] sortedList) {
    for (int i = 0; i < sortedList.length; i++) {
        if (searchNumber >= sortedList[i]) {
            if (searchNumber == sortedList[i]) {
                return i;
            } else {
                return -1; // searchNumber not found in list
            }
        }
    }
    return -1; // searchNumber not found in list
} // end method getIndexOfNumber

```

- Expected number of comparisons
  - Number in list: 8 + 1 O(n/2)
  - Number not in list: 8 + 1 O(n/2)

In general:
Recursive paradigm

```java
static int getIndexOfNumberR (int searchNumber, int[] sortedList,
    int lowIndex, int highIndex) {
    if (lowIndex == highIndex) {  // base case
        if (searchNumber == sortedList[lowIndex]) {
            return lowIndex;
        } else {
            return -1;
        }
    }
    int midIndex = (lowIndex + highIndex)/2;
    if (searchNumber <= sortedList[midIndex]) {
        answer = getIndexOfNumberR(searchNumber,sortedList,lowIndex,midIndex);
    } else {
        answer = getIndexOfNumberR(searchNumber,sortedList,midIndex+1,highIndex);
    }
    return answer;
} // end method getIndexOfNumber
```

- Expected number of comparisons
  - Number in list: $4 + 1$  \(O(\log_2 n)\) runtime
  - Number not in list: $4 + 1$  \(O(\log_2 n)\) runtime

In general:

---
College selector: a simple data structure

- Decision Diagram can be implemented as a *binary tree*
- Allows generic implementation
- Defined recursively
- Allows runtime change

Like Math?

- True
- False

Like design?

- True
- False

Engineering

Like kids?

- True
- False

Want to be a suit?

- True
- False

Business

Fear needles?

- True
- False

Education

Nursing

Liberal Arts
public class DecisionNode {
    String text;
    DecisionNode trueNode;
    DecisionNode falseNode;
    ...
} // end DecisionNode

public DecisionNode (String text, DecisionNode trueNode, DecisionNode falseNode) {
    ...
} // end constructor

DecisionNode engineering = new DecisionNode("Engineering",null,null);
DecisionNode sciMath = new DecisionNode("Sci/Math",null,null);
DecisionNode nursing = new DecisionNode("Nursing",null,null);
DecisionNode education = new DecisionNode("Education",null,null);
DecisionNode buisness = new DecisionNode("Buisness",null,null);
DecisionNode liberalArts = new DecisionNode("Liberal Arts",null,null);
DecisionNode question5 = new DecisionNode("be suit?", buisness, liberalArts);
DecisionNode question4 = new DecisionNode("fear needles?", education, nursing);
DecisionNode question3 = new DecisionNode("like kids?", question4, question5);
DecisionNode question2 = new DecisionNode("like design?", sciMath, engineering);
DecisionNode question1 = new DecisionNode("like math?", question2, question3);
DecisionNode root = question1;
public String getCollege (decisionNode root) {
    if ( (root.getTrueNode()  == null) ||
        (root.getfalseNode() == null)) {
        return decisionNode.getText();
    }
    boolean answer = askQuestion(decisionNode.getText());
    if (answer) {
        return getCollege (root.getTrueNode())
    } else {
        return getCollege (root.getFalseNode());
    }
} // end getCollege

- To process the whole tree, the method is called with a root node of the entire tree as an initial parameter.
- The procedure calls itself recursively on the correct subtrees (based on the answer to the question)
- until reaching the base case with no children (a "leaf").
Counting the size of a binary tree

```java
public int countNodes (decisionNode root) {
    if ( (root.getTrueNode()  == null) ||
        (root.getFalseNode() == null) ) {
        return 1;
    }
    int trueSideCount = countNodes(root.getTrueNode());
    int falseSideCount = countNodes(root.getFalseNode());
    return 1 + trueSideCount + falseSideCount;
} // end countNodes
```

Shorthand: Each `countNodes` frame listed as root node `<text>`, `trueSideCount`, `falseSideCount`
Stack trace I

Shorthand: Each countNodes frame listed as root node <text>, trueSideCount, falseSideCount
Stack trace II

Shorthand: Each countNodes frame listed as root node <text>, trueSideCount, falseSideCount

returns: 11
Final thoughts on recursion

- We’ve already encountered many “recursive-like” programming features:
  - Constructor chaining
  - Dealing with aggregate objects (deep copies, for example)
- Thinking recursively takes time and practice
  - Recursive is not always better (usually not, in fact)
  - Recursive is sometimes simpler
    - In some instances a LOT simpler that
  - Recursive is sometimes more flexible