CMSC 132:
OBJECT-ORIENTED PROGRAMMING II

Recursive Algorithms

Department of Computer Science
University of Maryland, College Park

Java Program Memory Organization

- Four memory areas:
  - **Stack** - makes possible method execution
    - Makes recursion possible
  - **Heap**
    - Where objects are created
  - **Static area**
  - **Code**
  - See http://www.cs.umd.edu/~nelson/classes/resources/MemoryMapsInformation/

Proof By Induction

- Mathematical technique
- A theorem is true for all $n \geq 0$ if
  - Base case
    - Prove theorem is true for $n = 0$, and
  - Inductive step
    - Assume theorem is true for n (inductive hypothesis)
    - Prove theorem must be true for $n+1$

Factorial Computation

- Factorial definition
  - n! = n × n-1 × n-2 × n-3 × … × 3 × 2 × 1
    - n! = n * (n-1)!
  - 0! = 1
- To calculate factorial of n
  - Base case
    - If n = 0, return 1
  - Recursive step
    - Calculate the factorial of n-1
    - Return n × (the factorial of n-1)

Recursion

- Recursion is a strategy for solving problems
  - A procedure that calls itself
    - Can be seen as “a procedure that calls another procedure that performs the same task”
    - **Example:** Factorial.java (factA/factB methods)
- Approach
  - If (problem instance is simple / trivial)
    - Solve it directly
  - Else
    - Simplify problem instance into smaller instance(s) of the original problem
    - Solve smaller instance using same algorithm
    - Combine solution(s) to solve original problem

Classical Example – Factorial

- Code

```
int fact (int n) {
    if (n == 0)
       return 1;                              // base case
    return  n * fact(n-1);                    // recursive step
}
```

- **Example:** Factorial.java
- Let’s see a diagram for factorial(3)
- Factorial Visualization
  - https://www.cs.usfca.edu/~galles/visualization/RecFact.html
- Other Visualizations
  - https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Properties

- **Recursion relies on the call stack**
  - State of current method is saved when method is recursively invoked
  - Every method invocation gets own stack space
- **Any problem solvable with recursion may be solved with iteration (and vice versa)**
  - Use iteration with explicit stack (if needed) to store state

Recursion vs. Iteration

- Recursive algorithm

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

- Iterative algorithm

```
int fact ( int n ) {
    int i, res;

    res = 1;
    for (i = n; i > 0; i--) {
        res = res * i;
    }

    return res;
}
```

Recursive algorithm is closer to factorial definition

Recursion vs. Iteration

- Iterative algorithms
  - **May be more efficient**
    - No additional function calls
    - Run faster, use less memory
- Recursive algorithms
  - **Higher overhead**
    - Time to perform function call
    - Memory for call stack
  - **May be simpler algorithm**
    - Easier to understand, debug, maintain
  - **Natural for backtracking searches**
  - **Suited for recursive data structures**
    - Trees, graphs…

Making Recursion Work

- Designing a correct recursive algorithm
  - Think of the simplest case possible
  - Work with the next simplest case possible
- Verify
  - Base case(s) is
    - Recognized correctly
    - Solved correctly
  - Recursive case
    - Solves 1 or more simpler subproblems
    - Can calculate solution from solution(s) to subproblems
    - Makes progress toward the base case
      - Needs to converge to the base case
- Uses principle of proof by induction

Examples

- We can provide simpler and more efficient solutions to the following problems using loops. We are using this simple problems to illustrate how to use recursion The power of recursion is seen other kind of problems (e.g., fractals, graph processing)
- Print → To print a string
  - **Base case**
    - If string has one character, print the character
  - **Recursive step**
    - Print first character
    - Skip 1st character and recur on remainder of the string
- **Example:** StringRecursiveMethods.java

Examples

- Find → To find an element in a string
  - **Base case**
    - If empty string, return false
  - **Recursive step**
    - If 1st character of string is target value, return true
    - Skip 1st character and recur on remainder of the string
- **Example:** StringRecursiveMethods.java
- Count Character Instances → To count # of a character in a string
  - **Base case**
    - If empty string, return 0
  - **Recursive step**
    - Skip 1st character and recur on remainder of the string
    - Add 1 to result
- **Example:** Additional examples at StringRecursiveMethods.java
- **Example:** TestsStringRecMethods.java

Examples (Auxiliary Method)

- Some recursive problems require an auxiliary method
  - The method supported by the auxiliary method cannot be called itself in order to solve the problem
  - Auxiliary method →the one that is actually recursive
- **Example:** ArrayExamples.java
- **Important:** Unless stated otherwise, all our recursive solutions may not rely on instance nor static variables

Types of Recursion: Tail Recursion

- Has a recursive call as final action
  - You can just pass back the result of the recursive call
    - No extra processing required for the value received and returned
- **Example**

```
int factorial(int n, int partialResult) {
    if (n == 0)
        return partialResult;
    return factorial(n-1, n*partialResult);
}
```

- **Can easily transform to iteration (loop)**
- In functional languages tail call elimination is often guaranteed by the language

Types of Recursion: Non-tail Recursion

- Example

```
int nontail( int n ) {
    …
    x =  nontail(n-1) ;
    y =  nontail(n-2) ;
    z = x + y;

    return z;
}
```

- **Can transform to iteration using explicit stack**

Possible Problems – Infinite Loop

- Infinite recursion
  - If recursion not applied to simpler problem

```
int bad (int n) {
    if (n == 0)
        return 1;
    return bad(n);
}
```

  - Infinite loop?
  - Eventually halt when runs out of (stack) memory
    - Stack overflow
    - Let’s see an example

Possible Problems – Efficiency

- May perform excessive computation
  - If recomputing solutions for subproblems
- **Example**
  - Fibonacci numbers
    - fibonacci(0) = 0
    - fibonacci(1) = 1
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- **Example:** Fibonacci.java

Possible Problems – Efficiency

- Recursive algorithm to calculate fibonacci(n)
  - If n is 0 or 1, return 1
  - Else compute fibonacci(n - 1) and fibonacci(n - 2)
  - Return their sum
- Simple algorithm - exponential time $O(2^n)$
  - Computes fibonacci(1) 2^n times
- Can solve efficiently using
  - Iteration
  - Dynamic programming

Memory Maps

- How does memory maps look like for recursive code?
- See example #7 of
- http://www.cs.umd.edu/~nelson/classes/resources/MemoryMapsInformation/MemoryMapsInformation.pdf

Fractals

- What is a Fractal?
- **Example:** fractals package