CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Recursive Algorithms

Department of Computer Science
University of Maryland, College Park

Recursion

- Recursion is a strategy for solving problems
  - A procedure that calls itself
- 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

Example – Factorial

- Factorial definition
  - $n! = n \times n\text{-}1 \times n\text{-}2 \times n\text{-}3 \times \ldots \times 3 \times 2 \times 1$
  - $0! = 1$
- 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)
- Code

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

Properties

- Recursion relies on the call stack
  - State of current procedure is saved when procedure is recursively invoked
  - Every procedure invocation gets own stack space
  - Let’s draw a diagram for factorial(4)
- Any problem solvable with recursion may be solved with iteration (and vice versa)
  - Use iteration with explicit stack to store state
  - Algorithm may be simpler for one approach

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

Examples

- Find → To find an element in an array
  - Base case
    - If array is empty, return false
  - Recursive step
    - If 1^{st} element of array is given value, return true
    - Skip 1^{st} element and recur on remainder of array
- Count Instances → To count # of elements in an array
  - Base case
    - If array is empty, return 0
  - Recursive step
    - Skip 1^{st} element and recur on remainder of array
    - Add 1 to result
- Some recursive problems require an auxiliary function
  - Auxiliary function →the one that actually is recursive
- **Example:** ArrayExamples.java

Examples

- Let’s look at recursive solutions for operations on a linked list
  - Find
  - Count
  - Print list
  - Print list in reverse
- Notice we can use the ?: operator for the implementation of some of these methods

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
- 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
- Uses principle of proof by induction

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 $n+1$

Types of Recursion

- Tail recursion
  - Has a recursive call as final action
  - 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

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

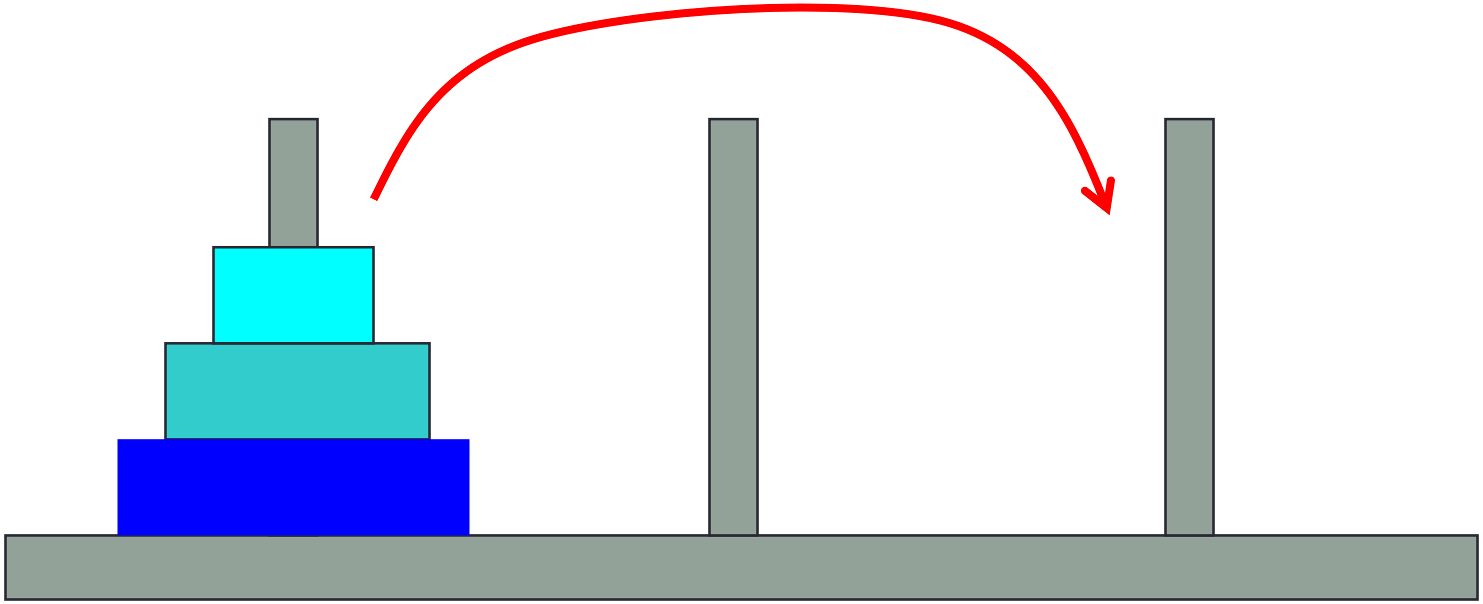
- Recursive algorithm to calculate fibonacci(n)
  - If n is 0 or 1, return 1
  - Else compute fibonacci(n-1) and fibonacci(n-2)
  - Return their sum
- Simple algorithm $\Rightarrow$ exponential time $O(2^n)$
  - Computes fibonacci(1) 2^n times
- Can solve efficiently using
  - Iteration
  - Dynamic programming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

Examples of Recursive Algorithms

- Towers of Hanoi
- Binary search
- Quicksort
- N-queens
- Fractals

Example – Towers of Hanoi

- Problem
  - Move stack of disks between pegs
  - Can only move top disk in stack
  - Only allowed to place disk on top of larger disk



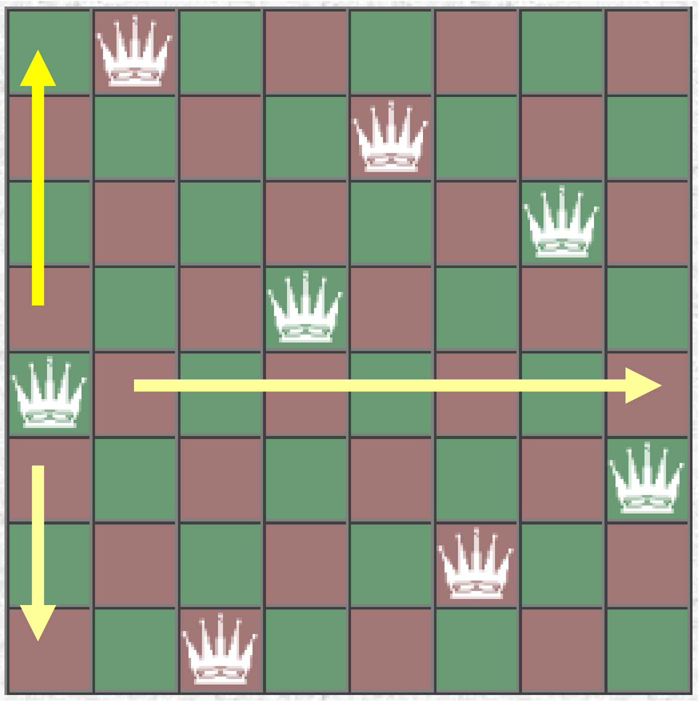
Example – Towers of Hanoi

- To move a stack of n disks from peg X to Y
  - Base case
    - If $n = 1$, move disk from X to Y
  - Recursive step
    - Move top n-1 disks from X to 3rd peg
    - Move bottom disk from X to Y
    - Move top n-1 disks from 3rd peg to Y

Iterative algorithm would take much longer to describe!

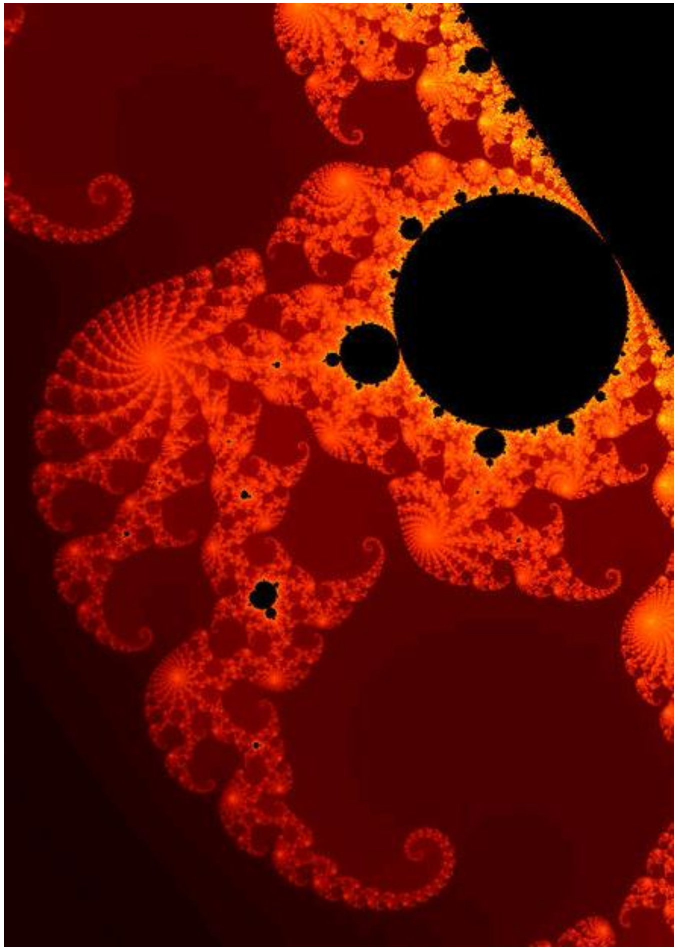
N-Queens

- Goal
  - Place queens on a board such that every row and column contains one queen, but no queen can attack another queen
- Recursive approach
  - To place queens on NxN board
  - Assume you've already placed K queens



Fractals

- Goal
  - Construct shapes using a simple recursive definition with a natural appearance
- Properties
  - Appears similar at all scales of magnification
    - Therefore “infinitely complex”
  - Not easily described in Euclidean geometry



Mandelbrot Set