Algorithm

Finite description of steps for solving problem

Problem types

- Satisfying ⇒ find any legal solution
- Optimization ⇒ find best solution (vs. cost metric)

Approaches

- Iterative ⇒ execute action in loop
- Recursive ⇒ reapply action to subproblem(s)
Recursive Algorithm

Definition
- An algorithm that calls itself

Approach
1. Solve small problem directly
2. Simplify large problem into 1 or more smaller subproblem(s) & solve recursively
3. Calculate solution from solution(s) for subproblem
Algorithm Format

1. Base case
   - Solve small problem directly

2. Recursive step
   - Simplify problem into smaller subproblem(s)
   - Recursively apply algorithm to subproblem(s)
   - Calculate overall solution
To find an element in an array

- **Base case**
  - If array is empty, return false

- **Recursive step**
  - If 1st element of array is given value, return true
  - Skip 1st element and recur on remainder of array

Example (See ArrayExamples.java)
Example – Count

To count # of elements in an array

Base case
- If array is empty, return 0

Recursive step
- Skip 1\textsuperscript{st} element and recur on remainder of array
- Add 1 to result
Example – Factorial

- **Factorial definition**
  - \( n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1 \)
  - \( 0! = 1 \)

- **To calculate factorial of \( n \)**
  - **Base case**
    - If \( n = 0 \), return 1
  - **Recursive step**
    - Calculate the factorial of \( n-1 \)
    - Return \( n \times (\text{the factorial of } n-1) \)
Example – Factorial

Code

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

Must have

- Small version of problem solvable without recursion
- Strategy to simplify problem into 1 or more smaller subproblems
- Ability to calculate overall solution from solution(s) to subproblem(s)
Making Recursion Work

- Designing a correct recursive algorithm
- Verify
  1. Base case is
     - Recognized correctly
     - Solved correctly
  2. Recursive case
     - Solves 1 or more simpler subproblems
     - Can calculate solution from solution(s) to subproblems
- Uses principle of proof by induction
Proof By Induction

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

- Problem may usually be solved either way
  - Both have advantages

- Iterative algorithms
  - May be more efficient
    - No additional function calls
    - Run faster, use less memory
Recursion vs. Iteration

- **Recursive algorithms**
  - Higher overhead
    - Time to perform function call
    - Memory for activation records (call stack)
  - May be simpler algorithm
    - Easier to understand, debug, maintain
  - Natural for backtracking searches
  - Suited for recursive data structures
    - Trees, graphs...
Example – Factorial

Recursion algorithm

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

Iterative algorithm

```c
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
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**
  1. Move top $n-1$ disks from X to 3rd peg
  2. Move bottom disk from X to Y
  3. Move top $n-1$ disks from 3rd peg to Y

Recursive algorithm is simpler than iterative solution
Possible Problems – Infinite Loop

- Infinite recursion
  - If recursion not applied to simpler problem

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

- Will infinite loop
- Eventually halt when runs out of (stack) memory
  - Stack overflow
Possible Problems – Inefficiency

- May perform excessive computation
  - If recomputing solutions for subproblems

Example

- Fibonacci numbers
  - \( \text{fibonacci}(0) = 1 \)
  - \( \text{fibonacci}(1) = 1 \)
  - \( \text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \)
Possible Problems – Inefficiency

- 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
  - Implementation (See Fibonacci.java)

- Simple algorithm $\Rightarrow$ exponential time $O(2^n)$
  - Computes fibonacci(1) $2^n$ times

- $O(n)$ Recursive Fibonacci implementation
  - See FibonacciOhN