EECS 2011 M: Fundamentals of Data Structures

Suprakash Datta
Office: LAS 3043

Course page: `http://www.eecs.yorku.ca/course/2011M`
Also on Moodle

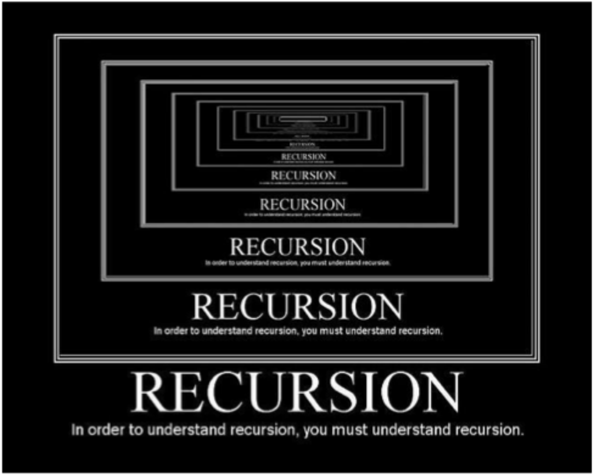
Recursion

Ch. 5

- Design Pattern in Software Engineering
- Algorithm design strategy
- Used for definitions and specifications

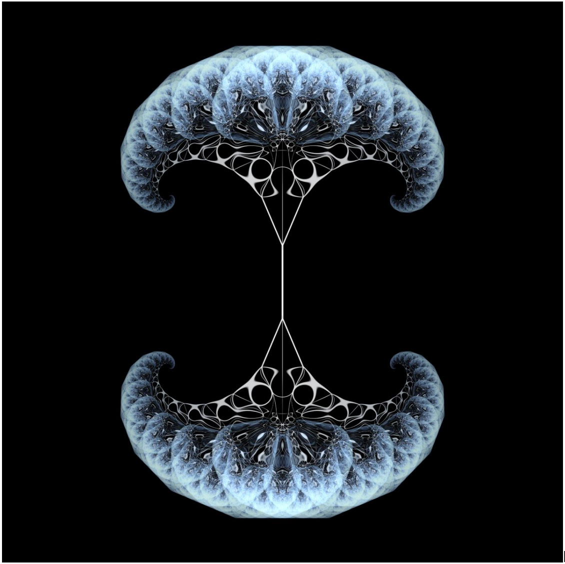
Note: Some slides in this presentation have been adapted from the authors' slides.

Recursion



From https://www.cse.buffalo.edu/ hartloff/CSE250/images/recursion.png

Recursion - 2



From Twitter @luizandregama, Jan 13, 2018

Recursive methods

- Base case(s): place for the execution to terminate
- Recursive calls
  - executes the current method
  - should be defined so that it makes progress towards a base case

Types:

- Linear recursion
- Binary recursion

Recursive Array Reversal

```
Algorithm reverseArray(A, i, j):
Input: An array A and nonnegative integer indices i, j
Output: The reversal of the subarray A[i..j]

if i < j then
    Swap (A[i], A[j])
    reverseArray(A, i + 1, j − 1)
return
```

Q: Where is the base case?

```java
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                          // if at least two elements in subarray
    int temp = data[low];                    // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1);   // recur on the rest
  }
}
```

Recursive GCD

```java
public class GCD {

    public static void main(String[] args) {
        int n1 = 366, n2 = 60;
        int gcd1 = gcd(n1, n2);

        System.out.printf("GCD(%d,%d)=%d", n1, n2, gcd1);
    }
    public static int gcd(int n1, int n2)
    {   // Assumes that the first arg is larger
        if (n2 != 0)
            return gcd(n2, n1 % n2);
        else
            return n1;
    }
}
```

Exercises

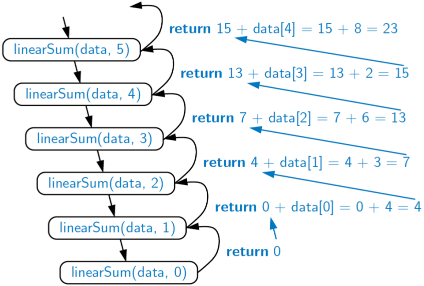
- Analyze the worst-case run-time of the gcd algorithm if terms of $n = n1 + n2$.
- Write a non-recursive (i.e., iterative) version of the gcd method

Tracing Recursion

```
Algorithm linearSum(A, n):
Input:  Array, A, of integers, Integer n,  0 <= n <= |A|
Output: Sum of the first n integers in A

if n = 0 then return 0
else return linearSum(A, n − 1) + A[n − 1]
```

`linearSum(A, 5)` called on array $A = [4, 3, 6, 2, 8]$



Recursive Exponentiation

```
RecPower(y, z)
// return y^z where y ∈ ℝ, z ∈ ℤ, z ≥ 0
if z == 0
   then return 1
   else return y * RecPower(y, z − 1);
```

- This function runs in $\theta(n)$ time (since we make n recursive calls)
- Can we do better than this?

Faster Recursive Exponentiation

```
FastRecPower(y, z)
  // return y^z where y ∈ ℝ, z ∈ ℤ, z ≥ 0
  if z == 0
     then return 1
     else  if Odd(z)
              then x ← FastRecPower(y, (z − 1)/2);
                   return y * x * x
     else  x ← FastRecPower(y, z/2);
           return x * x
```

What is the running time of this function, i.e., how many recursive calls?

Overheads of Recursion

- Each recursive call requires that the current process state (variables, program counter) be pushed onto the system stack, and popped once the recursion unwinds.
- This adds significant overhead
- Thus, it typically affects the running time constants, but not the asymptotic time complexity
- Thus recursive solutions may still be preferred unless there are very strict time/memory constraints.
- We will see instances of problems where recursion affects asymptotic time complexity (e.g., Fibonacci Series)

Common Errors

- Missing/incorrect base case:

```
// recursive factorial function
public static int recursiveFactorial(int n) {
    return n * recursiveFactorial(n- 1);
}
```

- The base condition involve (more) recursion

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

Binary Recursion

- Two recursive calls
- The number of calls grows exponentially with the value of inputs
- Common example: Fibonacci numbers

Fibonacci numbers

- Long history in Mathematics
- Defined as $F_1 = F_2 = 1$, $(\forall n > 2) F_n = F_{n-1} + F_{n-2}$

```
BinaryFib(k)
  // return F_k where k ∈ ℕ
  if k < 3
     then return k
     else  return BinaryFib(k − 1)
                 +BinaryFib(k − 2)
```

Fibonacci numbers - 2

How many recursive calls? Let n_k denote number of recursive calls made by `BinaryFib(k)`

- Each call adds 1, so to compute F_k you need F_k calls!
- Quick calculation: $F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2}$
  So
  $F_n \geq 2F_{n-2} \geq 2^2 F_{n-2*2} \geq 2^3 F_{n-2*3} \geq 2^4 F_{n-2*4} \ldots,$
  or
- $F_n = \Omega(2^{n/2})$, or exponentially many calls.

Fibonacci numbers - faster

$\text{FASTFIB}(k)$

// return (F_k, F_{k-1}) where $k \in \mathbb{N}$
if $k == 1$
  then return $(1, 0)$
  else $(i, j) \leftarrow \text{FASTFIB}(k-1)$
    return $(i + j, i)$

What is the running time?