CH. 5
RECURSION

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
OVERVIEW

• **Recursion** is an algorithmic technique where a function calls itself directly or indirectly.
  • Can also involve data ownership

• Why learn recursion?
  • Powerful algorithmic paradigm.

• Many computations are naturally self-referential.
  • A folder containing files and other folders.
  • Mathematical sequences – $f_{n+1} = f_n + 1$ (whole numbers)
  • Exploring mazes – make a step in the maze, then keep exploring the maze
EXAMPLE COUNTING

• Lets take an example of counting from 0

• The next number is the previous number plus 1, or mathematically: 
  \[ f_n = f_{n-1} + 1, \text{ where } f_0 = 0 \]

• So lets compute \( f_5 \)
  • \( f_5 = f_4 + 1 \), this would be great if we knew \( f_4 \), so lets expand it
  • \( f_5 = (f_3 + 1) + 1 \), then
  • \( f_5 = ( (f_2 + 1) + 1) + 1 \), then
  • \( f_5 = ( ( (f_1 + 1) + 1) + 1) + 1 \), then
  • \( f_5 = ( ( ( (f_0 + 1) + 1) + 1) + 1) + 1 \), then finally
  • \( f_5 = ( ( ( (0 + 1) + 1) + 1) + 1) + 1 = 5 \)
EXERCISE
FIBONACCI SEQUENCE

• The Fibonacci Sequence is used in various places in mathematics and computer science
  \[ f_n = f_{n-1} + f_{n-2}, \text{ where } f_0 = 0, f_1 = 1 \]

• Expand and evaluate \( f_6 \), work with a partner and show your work
HOW DO WE DO RECURSION IN CODE?

• Simply call the function within its own body

Pseudocode

**Algorithm** Foo
1. {Possibly do some stuff}
2. Foo()
3. {Possibly do some more stuff}

• Java

1. public static void foo() {
2. //possibly do some stuff
3. foo(); //This example of recursion
4. //possibly do some more stuff
5. }

recursion [ri-kur-zhuhn] 

n. See recursion.
EXAMPLE COUNTING

// This function counts using recursion

Algorithm recursiveCount

Input: Integer n

Output: The nth integer

1. \{f_0 = 0\}
2. if n = 0 then
3. return 0
4. \{f_n = f_{n-1} + 1\}
5. return recursiveCount(n - 1) + 1

• Together lets trace
  recursiveCount(3);
EXERCISE
FIBONACCI SEQUENCE

• Write an algorithm that computes the $n$th Fibonacci number:
  
  $$f_n = f_{n-1} + f_{n-2}$$

• Practice tracing $Fibonacci(4)$
CHARACTERISTICS OF RECURSION

• All recursive methods have the following characteristics:
  • One or more base cases (the simplest case) are used to stop recursion.
  • Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

• In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.
DESIGNING RECURSIVE FUNCTIONS

• Identify the base case
  • The base case is the part of the recursion not defined in terms of itself, e.g., \( f_0 = 0, f_1 = 1 \)
  • This is when the recursion stops! If you forget your base case, then the world will end
    • Really an infinite series of function calls until your computer crashes (if it ever does)

• Identify the recursive process
  • This is the algorithmic process or algorithmic formula

• Write the code
EXAMPLE
GREATEST COMMON DENOMINATOR (GCD)

• GCD
  • For two integers $p$ and $q$, if $q$ divides $p$, then the GCD of $p$ and $q$ is $q$.
  • Otherwise, the GCD of $p$ and $q$ is the same as $q$ and $p$ mod $q$.

• Step 1: Identify the base case
  • $q = 0$ implies that the GCD is $p$.

• Step 2: Identify the recursive step
  • $GCD(q, p \text{ mod } q)$

• Step 3: Write algorithm/code
EXAMPLE
GREATEST COMMON DENOMINATOR (GCD)

Algorithm gcd
Input: Integers $p$ and $q$
Output: Greatest common denominator of $p$ and $q$

1. if $q = 0$ then
2. return $p$;
3. return gcd($q$, $p \mod q$);
4. // Lets trace together gcd(1272, 216)
RECURSIVE HELPER METHODS

• Many problems can be solved with recursion, as it is loosely a looping technique. Example: palindrome

• However this example has an inefficiency, what is it?

**Algorithm** isPalindrome

**Inputs**: String s

1. if s.length() ≤ 1 then
2. return true;
3. else if s.charAt(0) ≠ s.charAt(s.length() - 1) then
4. return false;
5. else
6. return isPalindrome(s.substring(1, s.length() - 1));
RECURSIVE HELPER METHODS

- The preceding recursive `isPalindrome` method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a non-recursive method to jump start the recursive process, often with additional parameters.

**Algorithm** `isPalindrome`

**Input:** String `s`

**Output:** `true` if `s` is a palindrome, `false` otherwise

1. `return ` `isPalindrome(s, 0, s.length() - 1);`

**Algorithm** `isPalindrome`

**Input:** String `s`, Integers `low` and `high`

**Output:** `true` if `s` is a palindrome, `false` otherwise

1. `if ` `high ≤ low` `then`
2. `return ` `true;`
3. `else` `if ` `s.charAt(low) ≠ s.charAt(high)` `then`
4. `return ` `false;`
5. `else`
6. `return ` `isPalindrome(s, low + 1, high - 1);`
ANALYZING RECURSIVE METHODS

• Concerning time
  • Count the total number of operations based on the initial recursive invocation, e.g., the non-recursive method
  • Includes all operations of invocations down to the base case
  • Palindrome() takes $O(n)$ time because each character of the string is read once.

• Concerning memory
  • Yes, big-oh captures memory.
  • The parameters of each invocation need a separate place in memory
  • So, $k$ invocations at the very least takes $O(k)$ extra memory for the space of the method invocation. Tally up all memory of all invocations to see the correct answer
  • The first isPalindrome() took $O(n^2)$ memory because substrings were copied, we needed more memory. So $\sum_{i=0}^{n-1} n - 2i \leq O(n^2)$
  • The second isPalindrome() took $O(n)$ memory because only primitive types were copied to each invocation.
EXERCISE

• With your team, write and analyze a recursive algorithm that determines the number of uppercase letters in a string.
  • Use a non-recursive helper to make an efficient algorithm
  • Analyze time complexity
  • Analyze memory complexity
DOWNSIDE OF RECURSION

• Recursion is not always efficient!
• Take for instance, the Fibonacci sequence

**Algorithm** fib

**Input:** Integer \( n \)

**Output:** \( n \)th Fibonacci number

1. if \( n = 0 \) then
2. return 0
3. if \( n = 1 \) then
4. return 1
5. return \( \text{fib}(n - 1) + \text{fib}(n - 2) \)

- \( \text{fib}(50) \) is called once.
- \( \text{fib}(49) \) is called once.
- \( \text{fib}(48) \) is called 2 times.
- \( \text{fib}(47) \) is called 3 times.
- \( \text{fib}(1) \) is called 12,586,269,025 times.
BEST PRACTICE

• Try to Convert Recursive Algorithms to iterative ones whenever possible

• Example Fibonacci Sequence in Java

```java
1. //This is an example conversion. You can be even more efficient!
2. public static long fib(int n) {
3.     if (n == 0) return 0;
4.     if (n == 1) return 1;
5.     int fn2 = 0;
6.     int fn1 = 1;
7.     int fn = 1;
8.     // Iterative means repetition until failure condition,
9.     // typically done with loops and not recursion
10.    for (int i = 2; i < n; i++) {
11.        fn = fn1 + fn2;
12.        fn2 = fn1;
13.        fn1 = fn;
14.    }
15.    return fn;
16. }
```
OR DO TAIL RECURSION

• Tail recursion is when the last operation of a function is the recursive call
  • The compiler can optimize this to safe memory by reusing the space of the previous method invocation – this can only be done on tail recursion

• Example with factorial:
  \[ f_n = n \times f_{n-1} \]

**Without tail recursion**

1. public static long factorial(int n) {
2.   if (n == 0) return 1;
3.   return n*factorial(n-1);
4. }

**With tail recursion**

1. public static long factorial(int n) {
2.   return factorial(n, 1);
3. }

4. public static long factorial(int n, int result) {
5.   if(n == 0) return result;
6.   return factorial(n - 1, n * result);
7. }
SUMMARY

• How to write recursive processes?
  • Base case and reduction step.

• Why learn recursion?
  • New mode of thinking.
  • Powerful algorithmic tool