Chapter 18 Recursion
Motivations

A directory is a set of files, some of which are directories.

This is an example of a recursive definition.
Motivations

An H-tree is an H shaped circuit with H shaped circuits at its end points. Another recursive definition.
Motivations

An H-tree is an H shaped circuit with H shaped circuits at its end points. Another recursive definition.

H-trees, depicted below, are used in chip design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays.
Fractals

A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole.

*Example:* the Sierpinski triangle, named after a famous Polish mathematician.
Sierpinski Triangle

1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0.
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1.
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski order 2.
4. You can repeat the same process recursively to create a Sierpinski triangle order 3, 4, ..., and so on.
Fractals – the Koch curve
Recursion

- A recursive definition
  \[
  \text{left-hand-side} = \text{right-hand-side}
  \]
  that uses the left-hand-side in the right-hand-side

- e.g.,
  \[
  \text{a list} = \text{either empty}
  \]
  \[
  \text{or an element followed by a list}
  \]

This definition has a non recursive base case and a recursive general case.
Factorial

//recursive method
public int factorial(int n) {
    if (n == 0) // Base case
        return 1;
    else
        return n * factorial(n - 1); // Recursive call
}

//recursive definition
n! = n * (n-1)!
0! = 1
Trace Recursive factorial

Executes factorial(4)

factorial(4)

Space Required for factorial(4)
Main method

Stack

Step 0: executes factorial(4)
Step 1: executes factorial(3)
Step 2: executes factorial(2)
Step 3: executes factorial(1)
Step 4: executes factorial(0)
Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6
Step 9: return 24
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Executing factorial(2)
factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6

Executes factorial(1)
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Executes factorial(0)

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.
Trace Recursive factorial

Step 0: executes factorial(4)
Step 1: executes factorial(3)
Step 2: executes factorial(2)
Step 3: executes factorial(1)
Step 4: executes factorial(0)

return 1

Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method
Factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

returns factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Space Required for factorial(0)

Main method

Stack
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

returns factorial(1)
Trace Recursive factorial

Step 0: executes factorial(4)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

return 24
Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24

returns factorial(4)

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.
factorial(4) Stack Trace
Other Recursive definitions

\[ f(0) = 0; \]
\[ f(n) = n + f(n-1); \]

\[ g(0)=1; \]
\[ g(n)=g(n-1)+2 \]

\[ h(0)=1; \]
\[ h(n)=3*h(n-1); \]
Characteristics of Recursion

All recursive methods have the following characteristics:

– One or more non-recursive base cases are used to stop the recursion.

– Recursive calls that reduce the original problem, bringing it increasingly closer to a base case until it becomes a base case.

To solve a problem using recursion, you break it into smaller subproblems, similar to the original problem.

```
DoSomething(list){
    Do(head); DoSomething(head.next);
}
```
Reaching the base case

✧ You must convince yourself that the non-recursive base case is eventually reached. What about:

```java
public void doIt(int n){
    if(n != 0){
        bla;
        doIt(n-2);
    }
}
```
Think Recursively

Many problems can be solved using recursion. For example, the palindrome problem:

```java
public boolean isPalindrome(String s) {
    if (s.length() <= 1) // Base case
        return true;
    else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
        return false;
    else // Recursive general case
        return isPalindrome(s.substring(1, s.length() - 1));
}
```
Recursive Helper Methods

The preceding recursive isPalindrome method creates a new string for every recursive call. To avoid creating new strings, we write explicit string indices, using a helper method:

```java
public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
}

public static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
        return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
        return false;
    else
        return isPalindrome(s, low + 1, high - 1);
}
```
Recursive Binary Search

1. Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
2. Case 2: If the key is equal to the middle element, the search ends with a match.
3. Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.
Fibonacci’s Rabbits

- Suppose a newly-born pair of rabbits, one male, one female, are put on an island.
  - A pair of rabbits doesn’t breed until 2 months old.
  - Thereafter each pair produces another pair each month
  - Rabbits never die.

- How many pairs will there be after \( n \) months?

image from: http://www.jimloy.com/algebra/fibo.htm
Fibonacci Numbers

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89...

indices: 0 1 2 3 4 5 6 7 8 9 10 11

fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index -1) + fib(index -2); index >=2

fib(3) = fib(2) + fib(1)

= (fib(1) + fib(0)) + fib(1)
= (1 + 0) + 1 + 1 = 2
Fibonacci Numbers, cont.

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```

```
return fib(4)
```

```
return fib(3) + fib(2)
```

```
return fib(2) + fib(1)
```

```
return fib(1) + fib(0)
```

```
return 1
```

```
return 0
```

```
return 1
```

```
return 1
```

```
return 0
```
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.

Break a problem into subproblems.
If a subproblem is the same as the original problem, but with a smaller size, solve the subproblem recursively
Exercise

Let’s write a method `reverseLines(Scanner scan)` that reads lines using the scanner and prints them in reverse order.

- Use recursion without using loops.

- Example input:

<table>
<thead>
<tr>
<th>this</th>
<th>no?</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>fun</td>
</tr>
<tr>
<td>fun</td>
<td>is</td>
</tr>
<tr>
<td>no?</td>
<td>this</td>
</tr>
</tbody>
</table>

- Expected output:

<table>
<thead>
<tr>
<th>no?</th>
<th>fun</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
<td>this</td>
</tr>
</tbody>
</table>

- What are the cases to consider?
  - How can we solve a small part of the problem at a time?
  - What is a file that is very easy to reverse?
Reversal pseudocode

- Reversing the lines of a file:
  - Read a line L from the file.
  - Print the rest of the lines in reverse order.
  - Print the line L.

- If only we had a way to reverse the rest of the lines of the file....
Reversal solution

public void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case
        String line = input.nextLine();
        reverseLines(input);
        System.out.println(line);
    }
}

– Where is the base case?
Spock’s dilemma

✧ Entering a star system for the first time, Spock has a limited time before he has to go pick up Kirk.
  – There are $n$ planets
  – Spock has time to visit $k \leq n$ planets

✧ How many different combinations of planets can Spock visit?
Spock pseudo code

Spock can only visit k out of n planets, so he must choose k out of n (0 ≤ k ≤ n)

if (n==k || k==0) there is only one way
else // n>k and k>0
    take planet n. Spock can either visit n and then he must choose k-1 more out of n-1 or not,
then he must choose k out of n-1
Spock’s dilemma

public long combRec(long n, long k){
    if (n==k || k==0)       // only one way
        return 1;
    else
        return combRec(n-1, k-1)   // take n
        +        combRec(n-1, k);   // or don’t

parkingLot (int n)

- parkingLot computes in how many different ways a parking lot of size n can be filled with two kinds of vehicles:
  - Civics, size 1
  - Explorers, size 2
- Here are some examples:
  - A parking lot of size 1 can have 1 Civic (C), so the answer is 1.
  - A parking lot of size 2 can have 1 Explorer (E) or two Civics (CC), so the answer is 2.
parkingLot (int n)

public static long parkingLot (int n) {
    if (n == 1) return 1;  // a Civic
    else if (n == 2) return 2;  // an Explorer or two Civics
    else return
        parkingLot(n-2)  // Explorer in last position
        +               // or
        parkingLot(n-1); // Civic in last position
}
Memoization

✦ Problems like Fibonacci and parkingLot create “bushy” trees.
✦ These trees are full of repeated calls
✦ We can achieve tremendous speedup by saving intermediate results.

Look back at the Fibonacci call tree:

- fib(n) calls fib(n-1) and fib(n-2)
- fib(n) calls fib(n-2) and fib(n-3)
- So fib(n) calls fib(n-2) twice (1 direct, 1 indirect)
Fast Fib

private long[] memo = new long[100];
public long fastFib(int n) {
    if (n < 2) return n;
    if (memo[n] == 0) { // not computed yet
        // so compute and memoize it
        memo[n] = fastFib(n-1) + fastFib(n-2);
    }
    return memo[n];
}
Fast Spock

```java
public static long spock(int n, int k, long [][] A)

    if (A[n][k] == 0)
    {
        if (k == 0 || n == k)  // pick nobody or pick everybody
            A[n][k] = 1;
        else
            A[n][k] = spock(n-1, k, A)  // pick a subset without n
                + spock(n-1, k-1, A);  // pick a subset with n
    }

    return A[n][k];

```
Exercise

- Write fast parkingLot
  (see parkingLot on slide 39)
Towers of Hanoi

- There are $n$ different sized disks labeled 1, 2, 3, \ldots, $n$, and three towers labeled A, B, and C.
- All the disks are initially placed on tower A. The goal is to move all disks to tower B.
- No disk can be on top of a smaller disk at any time.
- Only one disk can be moved at a time, and it must be the top disk on the source (and destination) tower.
Towers of Hanoi, cont.

Now try it for 4 disks
Solution to Towers of Hanoi

The Tower of Hanoi problem can be decomposed into three subproblems.
Solution to Towers of Hanoi

- Move the top $n-1$ disks from A to C using tower B
- Move disk $n$ from A to B
- Move $n-1$ disks from C to B using tower A
Exercise 18.3 GCD

gcd(2, 3) = 1

gcd(2, 10) = 2

gcd(25, 35) = 5

gcd(205, 301) = 5

gcd(m, n)

Approach 1: Brute-force, start from min(n, m) down to 1, to check if the number is common divisor for both m and n, if so, it is the greatest common divisor.

Approach 2: Euclid’s algorithm

Approach 3: Recursive Euclid
Euclid’s algorithm

E.g., \( \text{gcd}(287, 91) \)

- \( 287 = (287/91) \times 91 + 287 \mod 91 = 91 \times 3 + 14 \)

  any divisor of 287 and 91 is a divisor of 14:
  \( 287 - 91 \times 3 = 14 \)

  also

  any divisor of 91 and 14 must be a divisor of 287:
  \( 287 = 91 \times 3 + 14 \)

- Hence \( \text{gcd}(287, 91) = \text{gcd}(91, 14) \)

Now compute \( \text{gcd}(287, 91) \) using this method.
Euclid’s algorithm

// Get absolute value of m and n;
\texttt{t1 = Math.abs(m); t2 = Math.abs(n);}
// r is the remainder of t1 divided by t2;
\texttt{r = t1 \% t2;}
\texttt{while (r != 0) {}
    \texttt{t1 = t2;}
    \texttt{t2 = r;}
    \texttt{r = t1 \% t2;}
\texttt{}}
// When r is 0, t2 is the greatest common
// divisor between t1 and t2
\texttt{return t2;}
Recursive Euclid

\[
gcd(m, n) = n \text{ if } m \% n = 0;
gcd(m, n) = gcd(n, m \% n); \text{ otherwise;}
\]

Exercise: write this as a java method.
Using Recursion

Recursion is good for solving problems that are inherently recursive, and not easily solved iteratively

   Spock, Parkinglot, Hanoi

This usually means: more than linear recursive

   Multiple recursive calls
   All the above have two recursive calls

Linear recursion can be easily replaced by iteration

   palindrome, reverse, factorial, binary search, gcd