Recursion

2301260 Programming Techniques

ผศ. ศศิภา พันธุวดีธร ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

Chapter outline

- Introduction
- Recursion concepts
- Example Using Recursion: Factorials
- Example Using Recursion: Fibonacci Series
- Recursion vs. Iteration
- Recursive helper method

Introduction

- Recursion is a programming technique in which a method calls itself.
- Known as a recursive method
- Some problems can be solved only by recursion, and some problems that can be solved by other techniques are better solved by recursion.

Recursion concepts

- To solve a problem using recursion, break it into subproblems.
- Each subproblem is the same as the original problem but smaller in size.
- Apply the same approach to each subproblem to solve it recursively.

Recursion concepts

- The method is implemented using an **if-else** or a **switch** statement that leads to different cases.
- 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.

Problem Solving Using Recursion

```java
public static void drinkCoffee(Cup cup) {
	if (!cup.isEmpty()) {
		cup.takeOneSip(); // Take one sip
		drinkCoffee(cup);
	}
}
public static void nPrintln(String message, int times) {
	if (times >= 1) {
		System.out.println(message);
		nPrintln(message, times - 1);
	} // The base case is times == 0
}
```

Recursion concepts

- For recursion to eventually terminate, each time the method calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a base case.
- When the method recognizes the base case, it returns a result to the previous copy of the method.
- A sequence of returns ensues until the original method call returns the final result to the caller.

Computing factorial

- Factorial of a positive integer n, written n!

  $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ eg. $4! = 4 \times 3 \times 2 \times 1$
- with 1! equal to 1 and 0! defined to be 1.
- The factorial of integer number (where number >= 0) can be calculated iteratively (nonrecursively) using a for statement as follows:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

- Recursive declaration of the factorial method is arrived at by observing the following relationship:
  - n! = n · (n – 1)!

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6
= 24

Computing factorial (use recursion)

```
import java.util.Scanner;
public class ComputeFactorial {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a non-negative integer: ");
    int n = input.nextInt();
    System.out.println("Factorial of "+n+" is "+factorial(n));
  }
  public static long factorial(int n) {
    if (n == 0) // Base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call
  }
}
```

Computing Fibonacci series

```
import java.util.Scanner;
public class Fibonacci {
    public static void main(String[] args) {
        int fib1, fib2, fibn;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();
        fib1 = 1;
        System.out.println("fib(1) = " + fib1);
        fib2 = 1;
        System.out.println("fib(2) = " + fib2);
        for (int i = 3; i <= n; i++) {
            fibn = fib1 + fib2;
            System.out.println("fib(" + i + ") = " + fibn);
            fib1 = fib2;
            fib2 = fibn;
        }
    }
}
```

```
run:
Enter n: 5
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
```

Computing Fibonacci series (use recursion)

- The Fibonacci series, begins with 1 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two.

  1, 1, 2, 3, 5, 8, 13, 21, …

- This series occurs in nature and describes a form of spiral.
- The Fibonacci series may be defined recursively as follows:

  fibonacci(1) = 1

  fibonacci(2) = 1

  fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

```java
import java.util.Scanner;
public class RecursiveFib {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter n: ");
        int n = in.nextInt();
        for (int i = 1; i <= n; i++){
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
        }
    }

    public static long fib(int n){
        if (n <= 2)
            return 1;
        else
            return fib(n - 1) + fib(n - 2);
    }
}
```

The Efficiency of Recursion

- Recursive implementation of fib is straightforward
- Watch the output closely as you run the test program
- First few calls to fib are quite fast
- For larger values, the program pauses an amazingly long time between outputs
- Method takes so long because it computes the same values over and over
- The computation of fib(6) calls fib(3) three times

BigJava textbook

Call Tree for Computing fib(6)

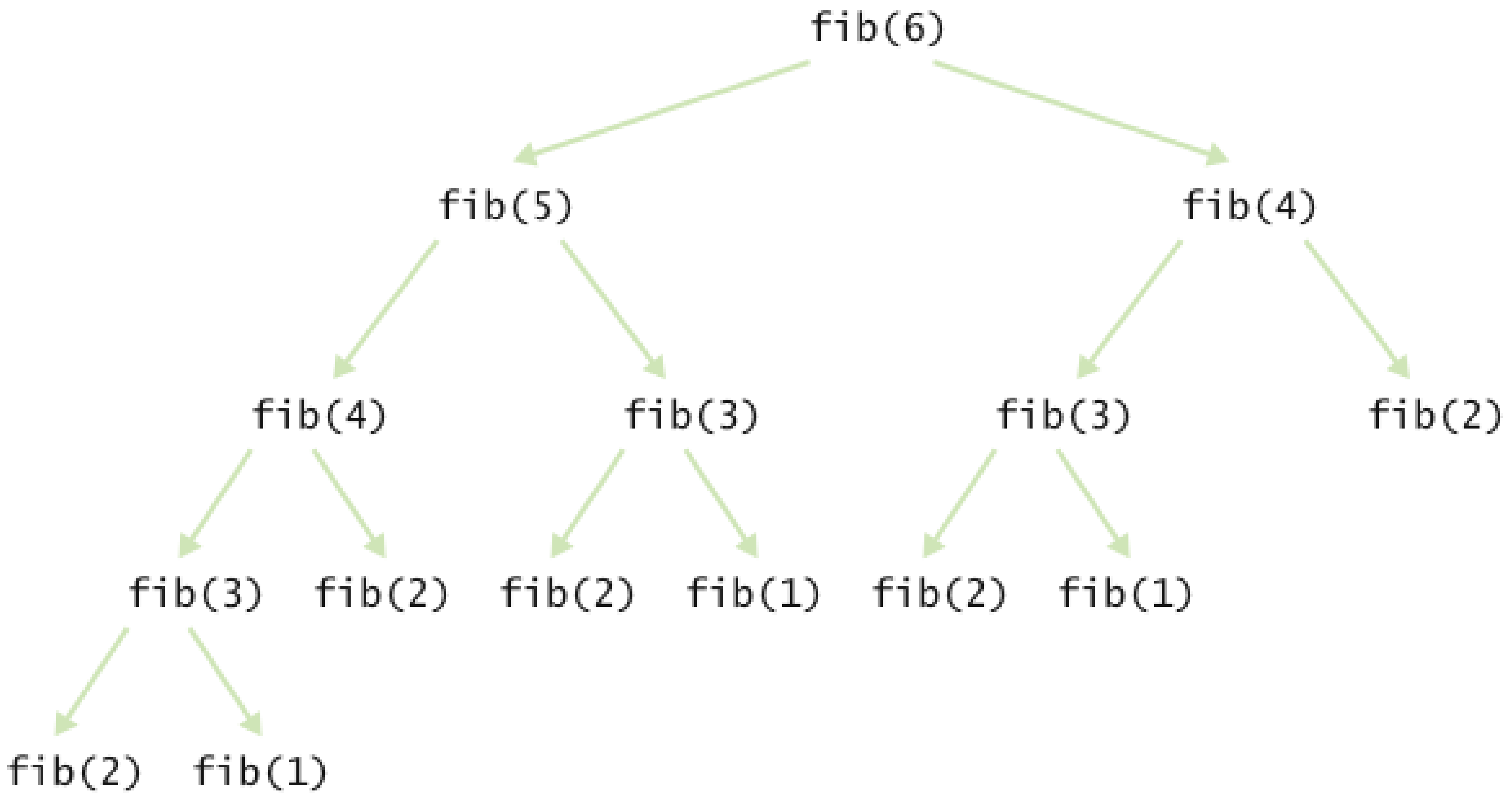

Figure 2 Call Pattern of the Recursive fib Method

BigJava textbook

Recursion vs. Iteration

- Both iteration and recursion are based on a control statement:
  - Iteration uses a repetition statement (e.g., for, while or do…while)
  - Recursion uses a selection statement (e.g., if, if…else or switch)
- Both iteration and recursion involve repetition:
  - Iteration explicitly uses a repetition statement
  - Recursion achieves repetition through repeated method calls
- Iteration and recursion each involve a termination test:
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is reached.

Recursion vs. Iteration (cont.)

- Both iteration and recursion can occur infinitely:
  - An infinite loop occurs with iteration if the loop-continuation test never becomes false
  - Infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

Recursion vs. Iteration (cont.)

- Recursion repeatedly invokes the mechanism, and consequently the overhead, of method calls.
  - Can be expensive in terms of both processor time and memory space.
- Each recursive call causes another copy of the method (actually, only the method’s variables, stored in the activation record) to be created
  - this set of copies can consume considerable memory space.
- Since iteration occurs within a method, repeated method calls and extra memory assignment are avoided.

แบบฝึกหัด

ข้อ 1

f(0) = 0;

f(n) = n + f(n-1);

ข้อ 2 Power

- Ex calculate integer powers of a variable
- evaluate x^n, or x*x...*x where x is multiplied by itself n times.

- You can use the fact that you can obtain xn by multiplying x^{n-1} by x.

- To put this in terms of a specific example, you can calculate 2^4 as 2^3 multiplied by 2, and you can get 2^3 by multiplying 2^2 by 2, and 2^2 is produced by multiplying 2^1, which is 2, of course, by 2.

Palindrome

- a word is a palindrome if
  - The first and last letters match, and Word obtained by removing the first and last letters is a palindrome (recursively check)
  - Strings with a single character
    - They are palindromes
  - The empty string
    - It is a palindrome

```java
public class RecursivePalindromeUsingSubstring {
 public static 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
   return isPalindrome(s.substring(1, s.length() - 1));
 }
```

```
public static void main(String[] args) {
    System.out.println("Is moon a palindrome? " + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? “ + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

The substring method in recursive call creates a new string that is the same as the original string except without the first and last characters.

It is a bit inefficient to construct new Sentence objects in every step

Recursive helper methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome
- Then, simply call the helper method with positions that test the entire string

Recursive Helper Methods

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

    private 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);
} // main method เหมือนเดิม
```

Recursive Selection Sort

1. Find the smallest number in the list and swaps it with the first number.
2. Ignore the first number and sort the remaining smaller list recursively.

Recursive Helper Methods

```
public class RecursiveSelectionSort {
    static double[] aList = {3, 5, 8, 1 ,2};
    public static void main(String[] args) {
        sort(aList);
        for (int i = 0; i<aList.length; i++)
            System.out.println(aList[i]);
    }
    public static void sort(double[] list) {
        sort(list, 0, list.length - 1); // Sort the entire list
    }
}
```

```java
private static void sort(double[] list, int low, int high) {
      if (low < high) {
   // Find the smallest number and its index in list[low .. high]
         int indexOfMin = low;
         double min = list[low];
         for (int i = low + 1; i <= high; i++)
            if (list[i] < min) {
               min = list[i];
               indexOfMin = i;
            }
   // Swap the smallest in list[low .. high] with list[low]
         list[indexOfMin] = list[low];
         list[low] = min;
   // Sort the remaining list[low+1 .. high]
         sort(list, low + 1, high);
      }
   }
```

```
run:
1.0
2.0
3.0
5.0
8.0
```

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.

* Input list must be sorted.

```
public class RecursiveBinarySearch {
    static int[] aList = {5, 8, 10, 12, 20};
    public static void main(String[] args) {
        System.out.println("Found at position : " + recursiveBinarySearch(aList, 0));
    }
    public static int recursiveBinarySearch(int[] list, int key) {
        int low = 0;
        int high = list.length - 1;
        return recursiveBinarySearch(list, key, low, high);
    }
```

```
private static int recursiveBinarySearch(int[] list, int key, int low, int high) {
    if (low > high)  // The list has been exhausted without a match
        return -low - 1;
    int mid = (low + high) / 2;
    if (key < list[mid])
        return recursiveBinarySearch(list, key, low, mid - 1);
    else if (key == list[mid])
        return mid;
    else
        return recursiveBinarySearch(list, key, mid + 1, high);
  }
}
```

```
Search for 0
run:
Found at position : -1
```

```
Search for 20
run:
Found at position : 4
```

References

- Deitel, H.M., and Deitel, P.J., *Java How to Program*, nineth edition, Prentice Hall, 2012.
- Horstmann, C., *Big Java*, John Wiley & Sons, 2009.
- Liang, Y. D., Introduction to Java Programming, tenth edition, Pearson Education Inc, 2015.

แบบฝึกหัด

ข้อ 1

$$f(0) = 0;$$

$$f(n) = n + f(n-1);$$

```java
import java.util.Scanner;
public class SumRecursive {
    public static void main(String[] args) {
/*        Scanner in = new Scanner(System.in);
        System.out.print("Please enter a number to find sum : ");
        int num = in.nextInt();
        System.out.println("sum of 1-" + num + " is " + sum(num));
*/    System.out.println("sum of 1-" + 5 + " is " + sum(5));
    }
    public static int sum(int x) {
        if (x == 0)
            return 0;
        else
            return x + sum (x-1);
    }
}
```

Power

<u>Ex</u> calculate integer powers of a variable

evaluate x^n, or x*x...*x where x is multiplied by itself n times.

You can use the fact that you can obtain x^n by multiplying x^{n-1} by x.

To put this in terms of a specific example, you can calculate 2^4 as 2^3 multiplied by 2, and you can get 2^3 by multiplying 2^2 by 2, and 2^2 is produced by multiplying 2^1, which is 2, of course, by 2.

How it works

- $n > 1$

  A recursive call to power() is made with n reduced by 1, and the value that is returned is multiplied by x. This is effectively calculating x^n as x times x^{n-1}.

- $n < 0$

  x^{-n} is equivalent to $1/x^n$ so this is the expression for the return value. This involves a recursive call to power() with the sign of *n* reversed.

- $n = 0$

  x^0 is defined as 1, so this is the value returned.

- $n = 1$

  x^1 is x, so x is returned.

```
public class PowerCalc {
        public static void main(String[] args) {
                double x = 5.0;
                System.out.println(x + " to the power 4 is " + power(x,4));
                System.out.println("7.5 to the power 5 is " + power(7.5,5));
                System.out.println("7.5 to the power 0 is " + power(7.5,0));
                System.out.println("10 to the power -2 is " + power(10,-2));
        }
        // Raise x to the power n
        static double power(double x, int n) {
                if(n > 1)
                        return x*power(x, n-1); // Recursive call
                else if(n == 0)
                        return 1.0; // When n is 0 return 1
                else if(n == 1)
                        return x;   // When n is 1 return x
                else
                        return 1.0/power(x, -n); // Negative power of x
        }
}
```

Output

5.0 to the power 4 is 625.0

7.5 to the power 5 is 23730.46875

7.5 to the power 0 is 1.0

10 to the power -2 is 0.01

การหาห.ร.ม.โดยวิธียุคลิด(Euclidean Algorithm) หรือตั้งหารสองแถว

- การหา ห.ร.ม.โดยวิธีหารสองแถว เหมาะสำหรับจำนวนที่มีค่ามาก และถ้ามีจำนวนเกิน 2 จำนวน ก็ ให้นำจำนวนน้อยมาจับคู่ก่อนแล้ว นำ ห.ร.ม.ที่ได้ไปจับคู่กับจำนวนต่อไป (เอาจำนวนน้อยหารจำนวนมากเสมอ)
- การหาห.ร.ม.โดยวิธียุคลิดมีขั้นตอน ดังนี้
  1) นำจำนวนนับที่มีค่าน้อยไปหารจำนวนนับที่มีค่ามาก
  2) จากข้อ 1 ถ้ามีเศษ ให้นำเศษไปหาจำนวนนับที่เป็นตัวหารในข้อ 1
  3) ปฏิบัติเช่นนี้ไปเรื่อย ๆ จนกระทั่งพบว่าจำนวนนับใดที่เหลือจากการหารแล้วหารลงตัว จำนวนนั้นคือ ห.ร.ม.
- ตัวอย่าง หา ห.ร.ม. ของ 366 และ 60

366 % 60 = 6

60 % 6 = 0

```java
public class GCD {
    public static void main(String[] args) {
        int n1 = 366, n2 = 60;
        int hcf = hcf(n1, n2);
        System.out.printf("G.C.D of %d and %d is %d.", n1, n2, hcf);
    }
    public static int hcf(int n1, int n2)    {
        if (n2 != 0)
            return hcf(n2, n1 % n2);
       else
           return n1;


    }
}
```

```java
if (n1%n2 != 0)
        return hcf(n2, n1 % n2);
else
        return n2;
```