

CSC 222: Computer Programming II

Spring 2005

recursion

- recursive algorithms, recursive methods
- base case, recursive case
- avoiding infinite recursion
- recursive classes
- recursion vs. iteration
- recursion & efficiency

1

Recursion

a *recursive algorithm* is one that refers to itself when solving a problem

- to solve a problem, break into smaller instances of problem, solve & combine
- recursion can be a powerful design & problem-solving technique
future examples: binary search, merge sort, hierarchical data structures, ...

classic (but silly) examples:

Fibonacci numbers:

1st Fibonacci number = 1

2nd Fibonacci number = 1

Nth Fibonacci number = (N-1)th Fibonacci number + (N-2)th Fibonacci number

Euclid's algorithm to find the Greatest Common Divisor (GCD) of a and b ($a \geq b$)

- if $a \% b == 0$, the $\text{GCD}(a, b) = b$
- otherwise, $\text{GCD}(a, b) = \text{GCD}(b, a \% b)$

IN-CLASS EXERCISE: sum from 1 to N

2

Recursive methods

```
/**
 * Computes Nth Fibonacci number.
 * @param N sequence index
 * @returns Nth Fibonacci number
 */
public int fibonacci(int N)
{
    if (N <= 2) {
        return 1;
    }
    else {
        return fibonacci(N-1) +
            fibonacci(N-2);
    }
}
```

```
/**
 * Computes Greatest Common Denominator.
 * @param a a positive integer
 * @param b positive integer (a >= b)
 * @returns GCD of a and b
 */
public int GCD(int a, int b)
{
    if (a % b == 0) {
        return b;
    }
    else {
        return GCD(b, a%b);
    }
}
```

these are classic examples, but pretty STUPID

- both can be easily implemented using iteration (i.e., loops)
- recursive approach to Fibonacci has huge redundancy

we will look at better examples later, but first analyze these simple ones

3

Understanding recursion

every recursive definition has 2 parts:

BASE CASE(S): case(s) so simple that they can be solved directly

RECURSIVE CASE(S): more complex – make use of recursion to solve *smaller* subproblems & combine into a solution to the larger problem

```
int fibonacci(int N)
{
    if (N <= 2) { // BASE CASE
        return 1;
    }
    else { // RECURSIVE CASE
        return fibonacci(N-1) +
            fibonacci(N-2);
    }
}
```

```
int GCD(int a, int b)
{
    if (a % b == 0) { // BASE CASE
        return b;
    }
    else { // RECURSIVE
        return GCD(b, a%b);
    }
}
```

to verify that a recursive definition works:

- convince yourself that the base case(s) are handled correctly
- ASSUME RECURSIVE CALLS WORK ON SMALLER PROBLEMS, then convince yourself that the results from the recursive calls are combined to solve the whole

4

Avoiding infinite(?) recursion

to avoid infinite recursion:

- must have at least 1 base case (to terminate the recursive sequence)
- each recursive call must get *closer* to a base case

```
int fibonacci(int N)
{
  if (N <= 2) { // BASE CASE
    return 1;
  }
  else { // RECURSIVE CASE
    return fibonacci(N-1) +
           fibonacci(N-2);
  }
}
```

with each recursive call, the number is getting smaller → closer to base case (≤ 2)

```
int GCD(int a, int b)
{
  if (a % b == 0) { // BASE CASE
    return b;
  }
  else { // RECURSIVE
    return GCD(b, a%b);
  }
}
```

with each recursive call, a & b are getting smaller → closer to base case ($a \% b == 0$)

5

Recursive classes/objects

recursion can apply to class definitions as well

- that is, can have an object which contains "smaller" instances of itself

example: consider a triangle consisting of N rows of symbols

```
[]
>[] []
>[] [] []
>[] [] [] []
```

- if you ignore the bottom row, you also have a triangle

```
[]
>[] []
>[] [] []
>[] [] [] []
```

6

Recursive triangles

can define a triangle of size N (N rows) to be:

- BASE CASE: if N = 1, then the triangle consists of a single []
- RECURSIVE CASE: otherwise, it is a triangle of size N-1 + a row of N []'s

here, Triangle class has a field that is an instance of the class

although circular, it is well-defined

- there is a BASE CASE (if N = 1, no internal Triangle is created)
- the internal Triangle has a smaller size, so the recursion is getting closer to the BASE CASE

```
public class Triangle
{
    private int size;
    private Triangle smallerTriangle;

    public Triangle(int triangleSize)
    {
        size = triangleSize;
        if (size > 1) {
            smallerTriangle = new Triangle(size-1);
        }
        . . .
    }
}
```

7

Triangle class

can utilize recursion to define simple methods

to display a Triangle:

- display the smaller Triangle
- display the bottom line (size []'s)

to compute its area:

- compute area of smaller Triangle
- add to area of bottom line (size)

```
public class Triangle
{
    private int size;
    private Triangle smallerTriangle;

    public Triangle(int triangleSize)
    {
        size = triangleSize;
        if (size > 1) {
            smallerTriangle = new Triangle(size-1);
        }
    }

    private void display()
    {
        if (size == 1) {
            System.out.println("[]");
        }
        else {
            smallerTriangle.display();
            for (int i = 0; i < size; i++) {
                System.out.print("[]");
            }
            System.out.println();
        }
    }

    public int area()
    {
        if (size == 1) {
            return 1;
        }
        else {
            return size + smallerTriangle.area();
        }
    }
}
```

8

Centered triangles

suppose we wanted to alter the display method so that triangle are centered

```
  []
 [] []
[] [] []
[] [] [] []
```

note: a centered triangle of size N =
a centered triangle of size N-1 (offset by one space) + line of N []'s

when calling the display method, need to be able to specify an offset amount

- to display a size N triangle with an offset of S spaces:
 - display a size N-1 triangle with an offset of S+1 spaces,
 - then, display a line of N []'s with an offset of S spaces

9

Recursive helper methods

the new recursive display method
must have an offset parameter

- this requires the user to specify an initial offset of 0 in order to display a centered triangle

```
Triangle tri = new Triangle(4);
tri.display(0);
```

- UGLY!** why would the user ever want to specify any offset other than 0?
- better solution:* provide a public method with no parameter and have it call a private "helper method" to do the real work
- since the helper method is private, the user never even knows it exists

```
tri.display();
```

```
...
public void display()
{
    display(0);
}

private void display(int indent)
{
    if (size > 1) {
        smallerTriangle.display(indent+1);
    }

    for (int i = 0; i < indent; i++) {
        System.out.print(" ");
    }
    for (int i = 0; i < size; i++) {
        System.out.print("[]");
    }
    System.out.println();
}
...
```

10

Better example: generating permutations

a recursive Triangle object is interesting, but still not all that compelling

- could easily implement the area and display methods without recursion **HOW?**

consider a better example:

- numerous applications require systematically generating permutations (orderings) e.g., word games, debugging concurrent systems, tournament pairings, . . .
- want to be able to take some sequence of items (say a String of characters) and generate every possible arrangement without duplicates

"123" → "123", "132", "213", "231", "312", "321"

"tape" → "tape", "taep", "tpae", "tpea", "teap", "tepa",
"atpe", "atep", "apte", "apet", "aetp", "aept",
"ptae", "ptea", "pate", "paet", "peta", "peat",
"etap", "etpa", "eatp", "eapt", "epta", "epat"

11

PermutationGenerator

in particular, we want to be able to generate permutations one-at-a-time

```
PermutationGenerator perms = new PermutationGenerator("tape");  
while (perms.hasMorePermutations()) {  
    System.out.println(perms.nextPermutation());  
}
```

```
tape  
taep  
tpae  
tpea  
teap  
tepa  
atpe  
atep  
apte  
apet  
aetp  
aept  
ptae  
ptea  
pate  
paet  
peta  
peat  
etap  
etpa  
eatp  
eapt  
epta  
epat
```

how do we generate permutations systematically?

- must maintain initial ordering
- must get all permutations, with no duplicates

12

Recursive generation

can think about permutation generation recursively

- for each letter in the word
 1. remove that letter
 2. get the next permutation of the remaining letters
 3. add the letter back at the front

example: "123"

1st permutation: "1" + (1st permutation of "23") = "1" + "23" = "123"
2nd permutation: "1" + (2nd permutation of "23") = "1" + "32" = "132"
3rd permutation: "2" + (1st permutation of "13") = "2" + "13" = "213"
4th permutation: "2" + (2nd permutation of "13") = "2" + "31" = "231"
5th permutation: "3" + (1st permutation of "12") = "3" + "12" = "312"
6th permutation: "3" + (2nd permutation of "12") = "3" + "21" = "321"

13

Java code

similar to Triangle,
can define the class
recursively

- only create nested generator if word length > 1
- current field keeps track of letter currently removed
- nextPermutation recursively gets the next permutation of the remaining letters, adds current at front

```
public class PermutationGenerator
{
    private String word;
    private int current;
    private PermutationGenerator tailGenerator;

    public PermutationGenerator(String aWord)
    {
        word = aWord;
        current = 0;
        if (word.length() > 1) {
            tailGenerator = new PermutationGenerator(word.substring(1));
        }
    }

    public String nextPermutation()
    {
        if (word.length() == 1) {
            current++;
            return word;
        }

        String r = word.charAt(current) + tailGenerator.nextPermutation();

        if (!tailGenerator.hasMorePermutations()) {
            current++;
            if (current < word.length()) {
                String tailString = word.substring(0, current)
                    + word.substring(current+1);
                tailGenerator = new PermutationGenerator(tailString);
            }
        }
        return r;
    }

    public boolean hasMorePermutations()
    {
        return current < word.length();
    }
}
```

14

Testing the `PermutationGenerator`

if using BlueJ, can simply call the `nextPermutation` method directly

or, could create another class with a main method

- prompt the user and read the string using a `Scanner` object

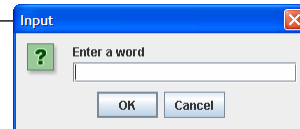
or, could make use of another Swing GUI component

- `JOptionPane` contains a static method, `showInputDialog`, that opens a frame with a prompt (label), text field, and submit buttons

```
import javax.swing.JOptionPane;

public class PermutationTester
{
    public static void main(String[] args)
    {
        String input = JOptionPane.showInputDialog("Enter a word");

        PermutationGenerator perms = new PermutationGenerator(input);
        while (perms.hasMorePermutations()) {
            System.out.println(perms.nextPermutation());
        }
    }
}
```



15

Recursion vs. iteration

it wouldn't be difficult to code `fibonacci` and `GCD` without recursion

```
public int fibonacci(int N)
{
    int previous = 1;
    int current = 1;
    for (int i = 3; i <= N; i++) {
        int newCurrent = current + previous;
        previous = current;
        current = newCurrent;
    }
    return current;
}

public int GCD(int a, int b)
{
    while (a % b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return b;
}
```

similarly, we could define the `Triangle` class without recursion

any recursive method/class can be rewritten iteratively (i.e., using a loop)

- but sometimes, a recursive definition is MUCH clearer
e.g., `PermutationGenerator` would be very difficult to conceptualize & implement without recursion

16

Recursion & efficiency

there is some overhead cost associated with recursion

```
public int GCD(int a, int b)
{
    if (a % b == 0) {
        return b;
    }
    else {
        return GCD(b, a%b);
    }
}

public int GCD(int a, int b)
{
    while (a % b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return b;
}
```

- with recursive version: each refinement requires a method call
involves saving current execution state, allocating memory for the method instance, allocating and initializing parameters, returning value, ...
- with iterative version: each refinement involves a loop iteration + assignments

the cost of recursion is relatively small, so usually no noticeable difference

- in practical terms, there is a limit to how deep recursion can go
e.g., can't represent a 10 million line triangle – would require 10 million objects
- in the rare case that recursive depth can be large (> 1,000), consider iteration

17

Recursion & redundancy

in the case of GCD, there is only a minor efficiency difference

- number of recursive calls = number of loop iterations

this is not always the case → efficiency can be significantly different

(due to different underlying algorithms)

consider the recursive fibonacci method:

```
fibonacci(5)
      fibonacci(4) + fibonacci(3)
      fibonacci(3) + fibonacci(2) + fibonacci(2) + fibonacci(1)
      fibonacci(2) + fibonacci(1) + fibonacci(2) + fibonacci(1)
```

- there is a SIGNIFICANT amount of redundancy in the recursive version
number of recursive calls > number of loop iterations (by an exponential amount!)
- recursive version is MUCH slower than the iterative one
in fact, it bogs down on relatively small values of N

18

When recursion?

- when it is the most natural way of thinking about & implementing a solution
can solve problem by breaking into smaller instances, solve, combine solutions
- when it is roughly equivalent in efficiency to an iterative solution
OR
- when the problems to be solved are so small that efficiency doesn't matter

- *think only one level deep*
make sure the recursion handles the base case(s) correctly
assume recursive calls work correctly on smaller problems
make sure solutions to the recursive problems are combined correctly

- *avoid infinite recursion*
make sure there is at least one base case & each recursive call gets closer

19

Tuesday: TEST 1

will contain a mixture of question types, to assess different kinds of knowledge

- quick-and-dirty, factual knowledge
e.g., TRUE/FALSE, multiple choice *similar to questions on quizzes*
- conceptual understanding
e.g., short answer, explain code *similar to quizzes, possibly deeper*
- practical knowledge & programming skills
trace/analyze/modify/augment code *either similar to homework exercises
or somewhat simpler*

the test will contain several "extra" points

e.g., 52 or 53 points available, but graded on a scale of 50 (hey, mistakes happen ☺)

study advice:

- review lecture notes (if not *mentioned* in notes, will not be on test)
- read text to augment conceptual understanding, see more examples & exercises
- review quizzes and homeworks

- feel free to review other sources (lots of Java tutorials online)

20