Lecture Scope

- The concept of recursion
- Recursion vs. Iteration
- Recursive methods
- Common errors in writing recursive methods
- Using recursion to solve the *Towers of Hanoi* problem
- Analyzing recursive algorithms
Recursion in Math

- Mathematical formulas are often expressed recursively
- $N!$, for any positive integer $N$, is defined to be the product of all integers between 1 and $N$ inclusive
- This definition can be expressed recursively:
  \[ 1! = 1 \]
  \[ N! = N \times (N-1)! \]
- A factorial is defined in terms of another factorial until the base case of $1!$ is reached

Recursion in Programming

- *Recursion* is a programming technique in which a method can call itself to fulfill its purpose.
- A method in Java can invoke itself; if set up that way, it is called a recursive method
- The code of a recursive method must handle both the base case and the recursive case
- Each call sets up a new execution environment, with new parameters and new local variables
- As always, when the method completes, control returns to the method that invoked it (which may be another instance of itself).
Recursive Method

General format for many recursive functions:

```java
if (some condition for which answer is known) // base case
    solution statement
else // general case (recursive case)
    recursive function call
```

Recursive Method (cont…)

- Example 1: A recursive method for computing x!

```java
long factorial (int x) {
    if (x == 0)
        return 1; // base case
    else
        return x * factorial (x - 1); // recursive case
}
```

- This method illustrates all the important ideas of recursion:
  - A base (or stopping) case
    - Code first tests for stopping condition (is $x == 0$?)
    - Provides a direct (non-recursive) solution for the base case ($0! = 1$).
  - The recursive case
    - Expresses solution to problem in 2 (or more) smaller parts
    - Invokes itself to compute the smaller parts, eventually reaching the base case
Why Recursion?

- Usually recursive algorithms have less code, therefore algorithms can be easier to write and understand - e.g. Towers of Hanoi. However, avoid using excessively recursive algorithms even if the code is simple.

- Sometimes recursion provides a much simpler solution. Obtaining the same result using iteration requires complicated coding - e.g. Quicksort, Towers of Hanoi, etc.

- Recursive methods provide a very natural mechanism for processing recursive data structures. A recursive data structure is a data structure that is defined recursively – e.g. Tree.

- Functional programming languages such as Clean, FP, Haskell, Miranda, and SML do not have explicit loop constructs. In these languages looping is achieved by recursion.

Recursive Programming

- Consider the problem of computing the sum of all the integers between 1 and N, inclusive:

  If N is 5, the sum =

  \[ 1 + 2 + 3 + 4 + 5 \]

  This problem can be expressed recursively as:

  The sum of 1 to N =

  \[ N \text{ plus the sum of 1 to N-1} \]
Recursive Programming (cont...)

- The sum of the integers between 1 and N:

\[
\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + (N-1) + \sum_{i=1}^{N-2} i
\]

\[
= N + (N-1) + (N-2) + \sum_{i=1}^{N-3} i
\]

\[
= N + (N-1) + (N-2) + \cdots + 2 + 1
\]

Recursive Programming (cont...)

- A recursive method that computes the sum of 1 to N:

```java
public int sum(int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum(num-1);
    return result;
}
```
Recursive Programming (cont…)

- Tracing the recursive calls of the `sum` method

```
main
    result = 4 + sum(3)
sum(4)
    sum
        result = 3 + sum(2)
sum(3)
    sum
        result = 2 + sum(1)
sum(2)
    sum
        result = 1
sum(1)
    sum
```

Another example where recursion comes naturally

- From mathematics, we know that
  
  \[ 2^0 = 1 \quad \text{and} \quad 2^6 = 2 \times 2^4 \]

- In general,
  
  \[ x^0 = 1 \quad \text{and} \quad x^n = x \times x^{n-1} \]

  for integer \( x \), and integer \( n > 0 \).

- Here we are defining \( x^n \) recursively, in terms of \( x^{n-1} \)

```java
// Recursive definition of power function
public static int Power ( int x, int n )
{
    if ( n == 0 )
        return 1;  // base case
    else
        return ( x * Power ( x, n-1 ) );  // general case
}
```

Of course, an alternative would have been to use looping instead of a recursive call in the function body.
Another example of recursive method

```java
// Another recursive function
public static int Func(int a, int b)
// Pre: a and b have been assigned values
// Post: Function value = ??
{
    int result;
    if (b == 0) // base case
        result = 0;
    else if (b > 0) // first general case
        result = a + Func(a, b - 1); // instruction 50
    else // second general case
        result = Func(-a, -b); // instruction 70
    return result;
}
```

Recursion vs. Iteration

- Just because we can use recursion to solve a problem, doesn't mean we should!
- You must be able to determine when recursion is the correct technique to use.
- Every recursive solution has a corresponding iterative solution
- A recursive solution may simply be less efficient
- Furthermore, recursion has the overhead of multiple method invocations
- However, for some problems recursive solutions are often more simple and elegant to express.
Types of Recursive Methods

- A recursive method is characterized based on:
  - Whether the method calls itself or not (direct or indirect recursion).
  - Whether the recursion is nested or not.
  - Whether there are pending operations at each recursive call (tail-recursive or not).
  - The shape of the calling pattern -- whether pending operations are also recursive (linear or tree-recursive).
  - Whether the method is excessively recursive or not.

Direct vs. Indirect Recursion

- A method invoking itself is considered to be \textit{direct recursion}.
- A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again.
- For example, method m1 could invoke m2, which invokes m3, which invokes m1 again.
- This is called \textit{indirect recursion}.
- It is often more difficult to trace and debug.
Direct vs. Indirect Recursion (cont…)

- A method is **directly** recursive if it contains an explicit call to itself.

  ```java
  public static long factorial(int x) {
    if (x == 0) {
      return 1;
    } else {
      return x * factorial(x - 1);
    }
  }
  ```

- A method x is **indirectly** recursive if it contains a call to another method which in turn calls x. They are also known as **mutually recursive** methods:

  ```java
  public static boolean isEven(int n) {
    if (n==0) {
      return true;
    } else {
      return isOdd(n-1);
    }
  }

  public static boolean isOdd(int n) {
    return !isEven(n);
  }
  ```
Direct vs. Indirect Recursion (cont...)

Example:

```java
public static double sin(double x){
    if (x < 0.0000001)
        return x - (x*x*x)/6;
    else {
        double y = tan(x/3);
        return sin(x/3) * ((3 - y*y)/(1 + y*y));
    }
}
```

```java
public static double tan(double x){
    return sin(x)/cos(x);
}
public static double cos(double x){
    double y = sin(x);
    return Math.sqrt(1 - y*y);
}
```

Nested and Non-Nested Recursive Methods

- Nested recursion occurs when a method is not only defined in terms of itself; but it is also used as one of the parameters:

  Example: The Ackerman function

  \[
  A(n, m) = \begin{cases} 
  m + 1 & \text{if } n = 0 \\
  A(n - 1, 1) & \text{if } n > 0, m = 0 \\
  A(n - 1, A(n, m - 1)) & \text{otherwise} 
  \end{cases}
  \]

  ```java
  public static long Ackmn(long n, long m){
      if (n == 0)
          return m + 1;
      else if (n > 0 && m == 0)
          return Ackmn(n - 1, 1);
      else
          return Ackmn(n - 1, Ackmn(n, m - 1));
  }
  ```

- The Ackermann function grows faster than a multiple exponential function.
Tail Recursion

- The case in which a function contains only a single recursive call and it is the last statement to be executed in the function.
- Tail recursion can be replaced by iteration to remove recursion from the solution as in the next example.
- Tail recursion is important in languages like Prolog and Functional languages like Clean, Haskell, Miranda, and SML that do not have explicit loop constructs (loops are simulated by recursion).

Tail and Non-Tail Recursive Methods

- A method is tail recursive if in each of its recursive cases it executes one recursive call and if there are no pending operations after that call.
- Example 1:

```java
public static void f1(int n){
    System.out.print(n + " ");
    if(n > 0)
        f1(n - 1);
}
```
- Example 2:

```java
public static void f3(int n){
    if(n > 6){
        System.out.print(2*n + " ");
        f3(n - 2);
    } else if(n > 0){
        System.out.print(n + " ");
        f3(n - 1);
    }
}
```
Tail and Non-Tail Recursive Methods (cont…)

Example of non-tail recursive methods:

Example 1:

```java
public static void f4(int n) {
    if (n > 0)
        f4(n - 1);
    System.out.print(n + " ");
}
```

- After each recursive call there is a pending `System.out.print(n + " ")` operation.

Example 2:

```java
long factorial(int x) {
    if (x == 0)
        return 1;
    else
        return x * factorial(x - 1);
}
```

- After each recursive call there is a pending * operation.

Converting tail-recursive method to iterative

It is easy to convert a tail recursive method into an iterative one:

<table>
<thead>
<tr>
<th>Tail recursive method</th>
<th>Corresponding iterative method</th>
</tr>
</thead>
</table>
| public static void f1(int n) { System.out.print(n + " "); if (n > 0) f1(n - 1); } | public static void f1(int n) { for( int k = n; k >= 0; k--)
System.out.print(k + " "); } |
| public static void f3(int n) { if (n > 6) { System.out.print(2*n + " "); f3(n - 2); } else if (n > 0) { System.out.print(n + " "); f3(n - 1); } } | public static void f3(int n) { while (n > 0) {
if (n > 6) {
    System.out.print(2*n + " ");
    n = n - 2;
} else if (n > 0) {
    System.out.print(n + " ");
    n = n - 1;
} else if (n > 0) { System.out.print(n + " ");
    n = n - 1;
} } |

23 24
Converting tail-recursive method to iterative (cont...)

Example 2 (part 1): Tail recursive method

```java
public boolean ValueInList (int list[ ], int value, int startIndex )
{
    if  ( list[startIndex] == value ) // one base case
        return  true ;
    else  if  (startIndex == list.length -1) // another base case
        return  false ;
    else
        return ValueInList( list, value, startIndex +1) ;
}
```

Example 2 (part 2): Corresponding iterative method

```java
// ITERATIVE SOLUTION
public boolean ValueInList (int list[ ], int value, int startIndex )
{
    boolean  found;
    found =  false ;
    while  ( !found && startIndex < list.length )
    {
        if ( value == list [ startIndex ] )
            found = true ;
        else
            startIndex++ ;
    }
    return  found ;
}
```
Linear and Tree Recursive Methods

- Another way to characterize recursive methods is by the way in which the recursion grows. The two basic ways are "linear" and "tree."

- If a recursive method is designed so that each invocation of the body makes at most one new recursive call, this is know as linear recursion.

- The factorial method is a clear example of linear recursion.

```java
long factorial (int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial (n - 1);
}
```

Linear and Tree Recursive Methods (cont…)

- A recursive method is said to be tree recursive when the pending operation involves another recursive call.

- When a method makes two recursive calls, we say that it uses binary recursion.

- The Fibonacci method fib provides a classic example of tree recursion.

```java
int fib(int n){
    if (n == 0 || n == 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```
Excessive Recursion

- A recursive method is excessively recursive if it repeats computations for some parameter values.

- Example: The call $\text{fib}(6)$ results in two repetitions of $\text{fib}(4)$. This in turn results in repetitions of $\text{fib}(3)$, $\text{fib}(2)$, $\text{fib}(1)$ and $\text{fib}(0)$:


Common Errors in Writing Recursive Methods

- The method does not call itself directly or indirectly.

- Non-terminating Recursive Methods (infinite recursion):
  a) No base case.

```java
int badFactorial(int x) {
    return x * badFactorial(x-1);
}
```

b) The base case is never reached for some parameter values.

```java
int anotherBadFactorial(int x) {
    if(x == 0)
        return 1;
    else
        return x * (x-1) * anotherBadFactorial(x-2);
    // When x is odd, we never reach the base case!!
}
```

Be sure that each recursive call progresses toward a base case. Otherwise, the recursion is incorrect.
Common Errors in Writing Recursive Methods (cont…)

- Post increment and decrement operators must not be used since the update will not occur until AFTER the method call - infinite recursion.

```java
public static int sumArray (int[] x, int index) {
    if (index == x.length)
        return 0;
    else
        return x[index] + sumArray (x, index++);
}
```

- Local variables must not be used to accumulate the result of a recursive method. Each recursive call has its own copy of local variables.

```java
public static int sumArray (int[] x, int index) {
    int sum = 0;
    if (index == x.length)
        return sum;
    else {
        sum += x[index];
        return sumArray(x, index + 1);
    }
}
```

Common Errors in Writing Recursive Methods (cont…)

- Wrong placement of `return` statement.
- Consider the following method that is supposed to calculate the sum of the first \( n \) integers:

```java
public static int sum (int n, int result) {
    if (n >= 0)
        return result;
    return sum(n - 1, n + result);
}
```

- When `result` is initialized to 0, the method returns 0 for whatever value of the parameter \( n \). The result returned is that of the final `return` statement to be executed. Example: A trace of the call `sum(3, 0)` is:

```
sum(3, 0) -> sum(2, 3) -> sum(1, 6) -> sum(0, 6) -> sum(-1, 6)
      |           |       |           |
      0          3       5       6
      return 0    return 3    return 5    return 6
```
Common Errors in Writing Recursive Methods (cont…)

- A correct version of the method is:

```java
public static int sum(int n, int result){
    if (n == 0)
        return result;
    else
        return sum(n-1, n + result);
}
```

- Example: A trace of the call `sum(3, 0)` is:

```
sum(3, 0) → return sum(2, 3) → return sum(1, 5) → return sum(0, 6) → return 6
```

Common Errors in Writing Recursive Methods (cont…)

- The use of instance or static variables in recursive methods should be avoided.

- Although it is not an error, it is bad programming practice. These variables may be modified by code outside the method and cause the recursive method to return wrong result.

```java
public class Sum{
    private int sum;
    public int sumArray(int[] x, int index){
        if(index == x.length)
            return sum;
        else {
            sum += x[index];
            return sumArray(x,index + 1);
        }
    }
}
```
Common Errors in Writing Recursive Methods (cont…)

- Overlapping recursive calls must be avoided because they tend to yield exponential algorithms.
- Using recursion in place of a simple loop is bad style.
- Recursive algorithms are analyzed by using a recursive formula. Do not assume that a recursive call takes linear time.

The Towers of Hanoi

- The Towers of Hanoi is a puzzle made up of three vertical pegs and several disks that slide onto the pegs.
- The disks are of varying size, initially placed on one peg with the largest disk on the bottom and increasingly smaller disks on top.
- The goal is to move all of the disks from one peg to another following these rules:
  - Only one disk can be moved at a time.
  - A disk cannot be placed on top of a smaller disk.
  - All disks must be on some peg (except for the one in transit).
Towers of Hanoi (cont…)

- A solution to the three-disk Towers of Hanoi puzzle:

![Image of Towers of Hanoi puzzle moves]

- A solution to ToH can be expressed recursively

- To move N disks from the original peg to the destination peg:
  - Move the topmost N-1 disks from the original peg to the extra peg
  - Move the largest disk from the original peg to the destination peg
  - Move the N-1 disks from the extra peg to the destination peg

- The base case occurs when a peg contains only one disk
Towers of Hanoi (cont…)

- The number of moves increases exponentially as the number of disks increases
- The recursive solution is simple and elegant to express and program, but is very inefficient
- However, an iterative solution to this problem is much more complex to define and program
Towers of Hanoi (cont...)

/**
 * SolveTowers uses recursion to solve the Towers of Hanoi puzzle.
 * @author Lewis and Chase
 * @version 4.0
 */
public class SolveTowers
{
    /**
     * Creates a TowersOfHanoi puzzle and solves it.
     */
    public static void main(String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}

/**
 * TowersOfHanoi represents the classic Towers of Hanoi puzzle.
 * @author Lewis and Chase
 * @version 4.0
 */
public class TowersOfHanoi
{
    private int totalDisks;

    /**
     * Sets up the puzzle with the specified number of disks.
     * @param disks the number of disks
     */
    public TowersOfHanoi(int disks)
    {
        totalDisks = disks;
    }

    /**
     * Performs the initial call to moveTower to solve the puzzle.
     * Moves the disks from tower 1 to tower 3 using tower 2.
     */
    public void solve()
    {
        moveTower(totalDisks, 1, 3, 2);
    }
}
/**
 * Moves the specified number of disks from one tower to another
 * by moving a subtower of n-1 disks out of the way, moving one
 * disk, then moving the subtower back. Base case of 1 disk.
 * @param numDisks the number of disks to move
 * @param start the starting tower
 * @param end the ending tower
 * @param temp the temporary tower
 */
private void moveTower(int numDisks, int start, int end, int temp)
{
    if (numDisks == 1)
        moveOneDisk(start, end);
    else
    {
        moveTower(numDisks-1, start, temp, end);
        moveOneDisk(start, end);
        moveTower(numDisks-1, temp, end, start);
    }
}

/**
 * Prints instructions to move one disk from the specified start
 * tower to the specified end tower.
 * @param start the starting tower
 * @param end the ending tower
 */
private void moveOneDisk(int start, int end)
{
    System.out.println("Move one disk from " + start + " to " + end);
}

Towers of Hanoi (cont...)
Analyzing Recursive Algorithms

- To determine the order of a loop, we determined the order of the body of the loop multiplied by the number of loop executions.
- Similarly, to determine the order of a recursive method, we determine the order of the body of the method multiplied by the number of times the recursive method is called.
- In our recursive solution to compute the sum of integers from 1 to \( N \), the method is invoked \( N \) times and the method itself is \( O(1) \).
- So the order of the overall solution is \( O(n) \).

Analyzing Recursive Algorithms (cont…)

- For the Towers of Hanoi puzzle, the step of moving one disk is \( O(1) \).
- But each call results in calling itself twice more, so for \( N > 1 \), the growth function is \( f(n) = 2^n - 1 \).
- This is exponential efficiency: \( O(2^n) \).
- As the number of disks increases, the number of required moves increases exponentially.
Summary

- Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.
- Any recursive definition must have a nonrecursive part, called the base case, which permits the recursion to eventually end.
- Mathematical problems and formulas are often expressed recursively.
- Each recursive call to a method creates new local variables and parameters.
- A careful trace of recursive processing can provide insight into the way it is used to solve a problem.
- Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.
- The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.
- The Towers of Hanoi solution has exponential complexity, which is very inefficient. Yet the implementation of the solution is incredibly short and elegant.

Any Question

???