Recursive Problem Solving

2140101 Computer Programming for International Engineers
Objectives

Students should:

• Be able to explain the concept of recursive definition.
• Be able to use recursion in Java to solve problems.
Recursive Problem Solving

How to solve problem recursively?

• Break a problem into identical but smaller, or simpler problems.

• Solve smaller problems to obtain a solution to the original one.
Example: Summation

Alternative Solution:

Let $S(n)$ be the sum of integers from 0 to $n$. Mathematically, we can write:

$$S(n) = \sum_{i=0}^{n} i$$

Java Implementation:

```java
public static int s(int n){
    int sum = 0;
    for(int i=0;i<=n;i++)
        sum += i;
    return sum;
}
```

Iterative Approach
Example: Summation

Write a method finding the sum of integers from 0 to \( n \)

Let \( S(n) \) be the sum of integers from 0 to \( n \).
Mathematically, we can write:

\[
S(n) = \begin{cases} 
  S(n - 1) + n & n = 1, 2, 3, ... \\
  0 & n = 0 
\end{cases}
\]

Java Implementation:

```java
public static int s(int n){
    if(n==0) return 0;
    return s(n-1)+n;
}
```

*Recursive Approach*
Example: Summation

Solving $S(n)$ is broken into solving $S(n-1)$, which is a simpler (but somewhat identical) problem.

$$S(n) = \begin{cases} S(n-1) + n & n = 1, 2, 3, \ldots \\ 0 & n = 0 \end{cases}$$

```java
public static int s(int n) {
    if (n == 0) return 0;
    return s(n - 1) + n;
}
```
Example: Summation

- In finding $S(2)$, method invocation can be depicted as:

```java
public static int s(int n){
    if(n==0) return 0;
    return s(n-1)+n;
}
```
Example: Factorial

Write a method finding $n!$

Mathematically, we can write:

$$n! = \begin{cases} 
  n \times (n-1) \times \ldots \times 1 & n = 1, 2, 3, \ldots \\
  1 & n = 0 
\end{cases}$$

Java Implementation:

```java
public static int factorial(int n) {
    int s = 1;
    for (int i = 1; i <= n; i++) s *= i;
    return s;
}
```

Iterative Approach
Example: Factorial

Write a method finding $n!$

Alternatively, we can write:

$$n! = \begin{cases} 
 n \times (n-1)! & n = 1, 2, 3, \ldots \\ 
 1 & n = 0 
\end{cases}$$

Java Implementation:

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

Recursive Approach
Example: Factorial

\[
\text{factorial}(4) = (\text{factorial}(3) \times 4)
\]

\[
\text{factorial}(3) = (\text{factorial}(2) \times 3)
\]

\[
\text{factorial}(2) = (\text{factorial}(1) \times 2)
\]

\[
\text{factorial}(1) = (\text{factorial}(0) \times 1)
\]

\[
\text{factorial}(0) = 1
\]
A recursive method must have two parts.

- **Base cases**: determine the case where the recursive method invocation terminates

- **Recursive cases**: recursive calls itself, but with simpler parameters

```java
public static int s(int n) {
    if (n == 0) return 0;
    return s(n - 1) + n;
}
```

```java
public static int factorial(int n) {
    if (n == 0) return 1;
    return factorial(n - 1) * n;
}
```
• Example: Fibonacci Numbers (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …)

  - The Fibonacci numbers form a sequence of integer defined recursively by:

\[
F(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
F(n-1) + F(n-2) & \text{if } n > 1
\end{cases}
\]
public class FiboDemo
{
    public static void main(String[] args)
    {
        final int n = 20;
        for(int i=0;i<20;i++)
            System.out.print(fibo(i)+",");
        System.out.println();
    }
    public static int fibo(int n){
        if(n<=0) return 0;
        if(n==1) return 1;
        return fibo(n-1)+fibo(n-2);
    }
}
Recursive Method Design

```
fibo(4)
```

Diagram showing recursive calls for `fibo(4)`. The diagram breaks down the calculation into smaller recursive calls, with `fibo(1)`, `fibo(2)`, and `fibo(0)` as the base cases.
Costs of Recursion

• A recursive method accomplishes its task by successively calling itself.
• Therefore, there are many invocations of method involved.
• Each time a recursive call is made, a certain amount of memory must be allocated.
• For a recursive method that makes very deep recursions, a large amount of memory is required.

Does this mean we should avoid recursive algorithms?
Example: Fibonacci Numbers Revisited

```java
import java.io.*;
public class FiboDemo
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter n:");
        int n = Integer.parseInt(stdin.readLine());
        System.out.println("---Using fibo()----------");
        System.out.println("F(\+n\+)\=\+fibo(n)\);
        System.out.println("---Using fiboNew()-----");
        System.out.println("F(\+n\+)\=\+fiboNew(n)\);
    }
    public static int fibo(int n){
        System.out.println("fibo("+n\+)\ is called.";
        if(n<=0) return 0;
        if(n==1) return 1;
        return fibo(n-1)+fibo(n-2);
    }
    // continue on the next page
```
Example: Fibonacci Numbers Revisited

```java
// The same fibo() as the previous example
public static int fiboNew(int n){
    int [] remember = new int[n+1];
    for(int i=0;i<=n;i++) remember[i]=-1;
    return fiboNew(n,remember);
}
public static int fiboNew(int n,int [] r){
    System.out.println("fiboNew("+n") is called.");
    if(n<=0){
        r[0]=0;
        return r[0];
    }
    if(n==1)
        r[n]=1;
    else
        r[n]=(r[n-1]==-1?fiboNew(n-1,r):r[n-1])
            + (r[n-2]==-1?fiboNew(n-2,r):r[n-2]);
    return r[n];
}
```
Example: Fibonacci Numbers Revisited

From the picture, we can see that finding the 6th Fibonacci number using \textit{fibo_0} requires more than three times as many method invocations as it is required in the case of using \textit{fiboNew_0}.
Example: The Towers of Hanoi

• Sometimes, the easiest and the least error-prone ways to write programs for solving some problems are recursive methods.
• Sometimes, an iterative approach is much more difficult than the recursive ones.
• See example “Towers of Hanoi”
Towers of Hanoi

Goal: Move all disks on Peg A to Peg B.

Rules:
1. Only one disk can be moved at a time, and this disk must be top disk on a tower.
2. A larger disk cannot be placed on the top of a smaller disk.

Using minimum number of moves
Towers of Hanoi

See “Towers of Hanoi Flash Demo” distributed with this courseware.
Example: The Towers of Hanoi

```java
import java.io.*;
public class TowerOfHanoiDemo {
    public static void main(String[] args) throws IOException{
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter number of disks:");
        int n = Integer.parseInt(stdin.readLine());
        move(n, "A", "B", "C");
    }
    // continue on the next page
```
Example: The Towers of Hanoi

```java
public static void move(int n,
        String orgPole,String destPole,String otherPole){
    String step;
    if(n<=1){
        step = "Move Disk1 from Peg "+orgPole+" to Peg "+destPole;
        System.out.println(step);
    }else{
        move(n-1,orgPole,otherPole,destPole);
        step = "Move Disk"+n+" from Peg "+orgPole+" to Peg "+destPole;
        System.out.println(step);
        move(n-1,otherPole,destPole,orgPole);
    }
}
```
Example: The Towers of Hanoi

Try solving it using an iterative approach.