Structural Programming and Data Structures

Winter 2000

CMPUT 102: Recursion

Dr. Osmar R. Zaïane

University of Alberta

Course Content

• Introduction	• Vectors
• Objects	• Testing/Debugging
• Methods	• Arrays
• Tracing Programs	• Searching
• Object State	• Files I/O
• Sharing resources	• Sorting
• Selection	• Inheritance
• Repetition	• **Recursion**

Objectives of Lecture 25

Recursion

- Introduce the concept of recursion;
- Understand how recursion works;
- Learn how recursion can be used instead of repetition;
- See some examples that use recursion.

Outline of Lecture 25

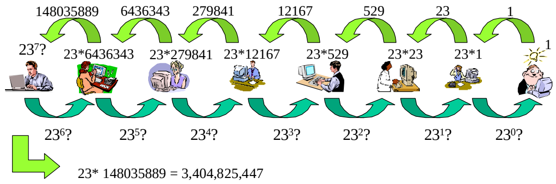
- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

Recursion

- **Recursion** occurs when a method calls itself, either directly or indirectly.
- If a problem can be resolved by solving a simple part of it a resolving the rest of the big problem the same way, we can write a method that solves the simple part of the problem then calls itself to resolve the rest of the problem.
- This is called a **recursive method**.

Recursive Method Example

- Suppose we want to calculate $\mathbf{23^7}$. We know that $\mathbf{23^7}$ is $\mathbf{23*23^6}$. If we know the solution for $\mathbf{23^6}$ we would know the solution for $\mathbf{23^7}$.



$23^7 = 23 * 23^6 =$
$23 * (23 * 23^5) =$
$23 * (23 * (23 * 23^4)) =$
$23 * (23*(23*(23 * 23^3))) =$
$23 * (23*(23*(23*(23*23^2))))=$
$23 * (23*(23*(23*(23*(23*23^1)))))=$
$23 * (23 *(23*(23*(23*(23*(23*23^0))))))=$
23 * (23 *(23*(23*(23*(23*(23*1))))))=
23 * (23 *(23*(23*(23*(23*(23))))))=
23 * (23 *(23*(23*(23*(529)))))=
23 * (23 *(23*(23*(12,167))))=
23 * (23 *(23*(279,841)))=
23 * (23 *(6,436,343))=
23 * (148,035,889)=
3,404,825,447

Outline of Lecture 25

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

Recursive Methods

- For recursion to **terminate**, two conditions must be met:
  - the recursive call must somehow be simpler than the original call.
  - there must be one or more simple cases that do not make recursive calls.

Outline of Lecture 25

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

Factorial

- For example, we would like to write a recursive method that computes the factorial of an Integer:

  0! = 1
  1! = 1
  2! = 2*1 = 2 → 2! = 2*1!
  3! = 3*2*1 = 6 → 3! = 3*2!
  n! = n*(n-1) * … *3*2* 1 → n! = n*(n-1)!
- The last observation, together with the simple cases is the basis for a recursive method.

Integer Factorial Method

- In the class Integer we want to add:

```
public int factorial() {
// Return the factorial of me.
   int   answer;
   Integer selfMinus1;

   if ((this.intValue() == 0)||(this.intValue() == 1))
       answer = 1;
   else {
       selfMinus1 = new Integer(this.intValue() - 1);
       answer = this.intValue()*selfMinus1.factorial();
   }
   return answer;
}
```

No Factorial in Integer

- Unfortunately, we cannot add methods to class Integer or create a subclass and add the method there (since class Integer is a "final" class).
- Therefore, we will build a new class called IntegerPlus and add the factorial method.

Recursive Factorial Method

```
public class IntegerPlus {
/* Each instance of this class represents an Integer.
   The class was created as a repository for Integer
   methods, since the Integer class is final. */

// Private Instance Variables
   private int value;

   public IntegerPlus(int anInt) {
   /*    Initialize me to have the given value. */

         this.value = anInt;
   }
```


Recursive Factorial Method (con't)

```
   public int factorial() {
   // Return the factorial of me.
        int              answer;
        IntegerPlus      selfMinus1;
        if ((this.value == 0) || (this.value == 1))
          answer = 1;
        else {
          selfMinus1 = new IntegerPlus(this.value - 1);
          answer = this.value * selfMinus1.factorial();
        }
        return answer;
   }
}
```


Loop Example

```
// Find the largest element in an array of ints

int  markArray[] = {50, 37, 71, 99, 63};
int  index;
int  max;
index = 0;
max = markArray[index];
for (index = 1; index < markArray.length; index++)
      if (markArray[index] > max)
              max = markArray[index];
System.out.println(max);
```

markArray

50	0
37	1
71	2
99	3
63	4

index=5

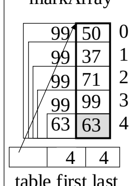
max

99

Recursion Example

```
// Find the largest element in an array of ints
int  markArray[] = {50, 37, 71, 99, 63};
int max=largest(markArray,0,markArray.length-1);
System.out.println(max);
…
public static int largest(int table[], int first, int last){
        if (first >= last) return table[last];
        else {
                int myMax=largest(table,first+1,last);
                if (myMax > table[first])
                        return myMax;
                else    return table[first];
        }
}
```

markArray



max

99

Outline of Lecture 25

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

Direct References in Methods

- When a method is executing it can access some objects and some values.
- The receiver object can be referenced directly using the pseudo-variable **this**.
- Other objects and values can be referenced directly using method parameters and local variables.
- Still other objects and values can only be accessed indirectly by sending messages that return references to them.

Method Activations and Frames

- A method can only access objects while it is executing or **active**.
- The collection of all direct references in a method is called the **frame** or **stack frame** of a method.
- The frame is created when the method is invoked, and destroyed when the method finishes.
- If a method is invoked again, a new frame is created for it.

Multiple Activations of a Method

- When we invoke a recursive method on an object, the method becomes active.
- Before it is finished, it makes a recursive call to the same method.
- This means that when recursion is used, there is more than one copy of the same method active at once.
- Therefore, each active method has its own frame which contains independent copies of its direct references.

Method Frames for Factorial

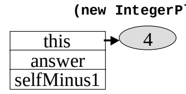
- Each frame has its own pseudo-variable, **this**, bound to a different receiver object.
- Each frame has its local variable, answer, bound to a different value.
- Each frame has its local variable, selfMinus1 bound to a different IntegerPlus object.
- These frames all exist at the same time.

Recursive Factorial Method (again)

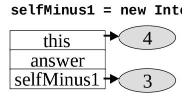
```
public int factorial() {
// Return the factorial of me.
      int             answer;
      IntegerPlus     selfMinus1;
      if ((this.value == 0) || (this.value == 1))
        answer = 1;
      else {
        selfMinus1 = new IntegerPlus(this.value - 1);
        answer = this.value * selfMinus1.factorial();
      }
      return answer;
   }
}
```


Calling (4).factorial()

```
(new IntegerPlus(4)).factorial()
```

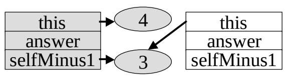



```
selfMinus1 = new IntegerPlus(this.value - 1);
```

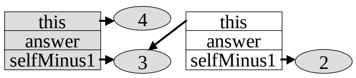



```
answer = this.value * selfMinus1.factorial();
```

Calling (3).factorial()

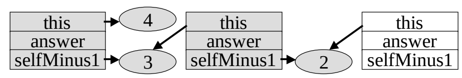


`selfMinus1 = new IntegerPlus(this.value - 1);`

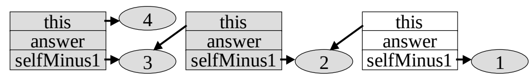


`answer = this.value * selfMinus1.factorial();`

Calling (2).factorial()

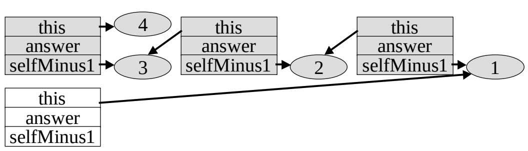


`selfMinus1 = new IntegerPlus(this.value - 1);`

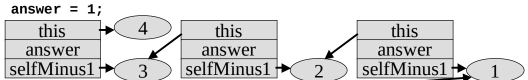


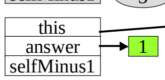
`answer = this.value * selfMinus1.factorial();`

Calling & Exiting (1).factorial()



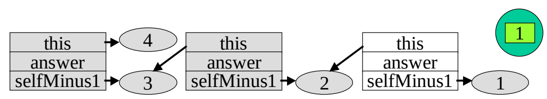
`answer = 1;`



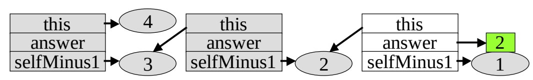


```
return answer;
==> 1
```


Exiting (2)factorial()

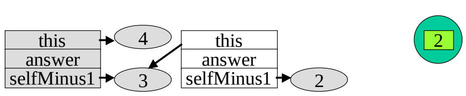


```
answer = this.value * selfMinus1.factorial();
         ----------   ----------------------
             2                  1
```

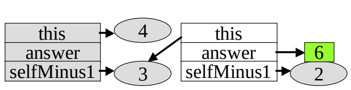


```
return answer;
==> 2
```


Exiting (3).factorial()

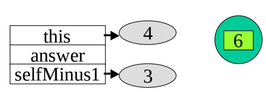


```
answer = this.value * selfMinus1.factorial();
         ----------   ----------------------
             3                  2
```

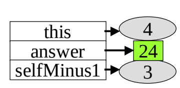


```
return answer;
==> 6
```


Exiting (4).factorial()



```
answer = this.value * selfMinus1.factorial();
         ----------   ----------------------
             4                  6
```



```
return answer;
==> 24
```

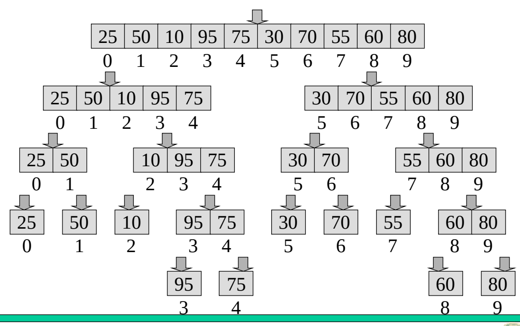
Outline of Lecture 25

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

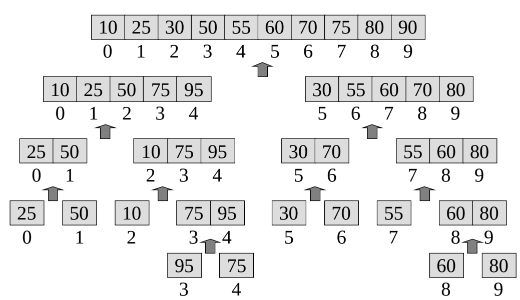
Recursive MergeSort Concept

- We can build a recursive sort, called mergeSort:
  - split the list into two equal sub-lists
  - sort each sub-list using a recursive call
  - merge the two sorted sub-lists

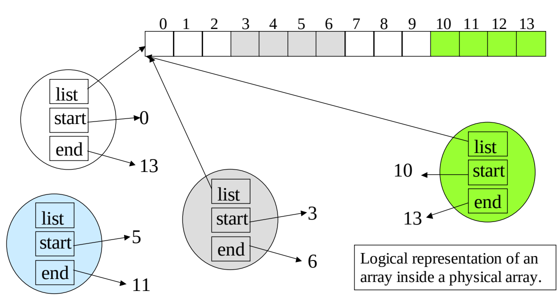
MergeSort Example - split



MergeSort Example - join



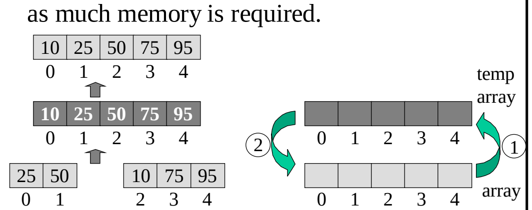
SubArray Object



Logical representation of an array inside a physical array.

MergeSort Needs Extra Storage

- Unlike selection sort, merge sort does not work "in place".
- A temporary collection is needed so twice as much memory is required.



Class SubArray

```
public class SubArray {
// An instance of this class represents a sub-array
// of an Array of ints.

// Constructor

   public SubArray(int anArray[], int start, int end) {
   // Initialize me to represent the given range of
   // the given Array.

        this.list = anArray;
        this.start = start;
        this.end = end;
   }
```


Instance Variables

```
// Private Instance Variables
   private int start;
   private int end;
   private int list[];

   private int size() {
   // Answer my size.
        if (this.end < this.start)   return 0;
        else                         return this.end - this.start + 1;
   }
```


Code for sort

```
public void sort() {
// Sort myself.
     SubArray temp;

     temp = new SubArray(new int[this.list.length],
             this.start, this.start-1);
     // the new subArray has the physical size of list but is empty
     //that is why the end is start-1
     this.mergeSort(temp);
}
```


Code for mergeSort

```
public void mergeSort(SubArray temp) {
// Sort myself using a merge sort.

   int          middle;
   SubArray     lowArray;
   SubArray     highArray;

   if (this.start < this.end) {
        middle = (this.start + this.end) / 2;
        lowArray = new SubArray(this.list, this.start, middle);
        lowArray.mergeSort(temp);
        highArray = new SubArray(this.list, middle+1, this.end);
        highArray.mergeSort(temp);
        this.merge(lowArray, highArray, temp);
   }
}
```


Code for merge

```
private void merge(SubArray low, SubArray high,
   SubArray temp) {
// Assume that both SubArrays are sorted.
// Merge them into me using the given temp.

   temp.start = 0;
   temp.end = -1;
   while ((low.size() > 0)&&(high.size() > 0))
        temp.moveSmallest(low, high);
   temp.moveFrom(low, low.size());
   temp.moveFrom(high, high.size());
   this.end = this.start - 1;
   this.moveFrom(temp, temp.size());
}
```


Code for moveSmallest

```
private void moveSmallest(SubArray low, SubArray high) {
// Move the first element of one of the two SubArrays to
// me. Pick the element which is smallest.

   if (low.list[low.start] < high.list[high.start])
        this.moveFrom(low, 1);
   else
        this.moveFrom(high, 1);
}
```

Code for moveFrom

```
private void moveFrom(SubArray source, int count) {
// Move the given count of ints from the source to me.

   int   index;

   for (index = 0; index < count; index++) {
         this.end = this.end + 1;
         this.list[this.end] = source.list[source.start];
         source.start = source.start + 1;
   }
}
```


Complexity of MergeSort

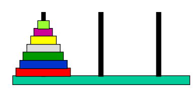
- The complexity of the MergeSort algorithm is beyond the scope of this course.
- However, the comparisons occur only in moveSmallest, which for an initially random collection, on average gets called about *n* log(n)* times for an array of size n.
- Sample times for our Java program:

	n = 20,000	n = 100,000
merge sort	< 1 second	1 second
selection sort	16 seconds	400 seconds

Outline of Lecture 25

- What is recursion?
- Conditions for termination
- Factorial
- Stack frames
- MergeSort
- Towers of Hanoi

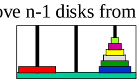
Towers of Hanoi



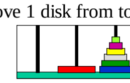
- No disk can be on top of a smaller disk;
- Only one disk is moved at a time;
- A disk must be placed on a tower;
- Only the top most disk can be moved.

To move n disks from tower 1 to 2:

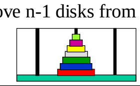
- Move n-1 disks from tower 1 to 3;



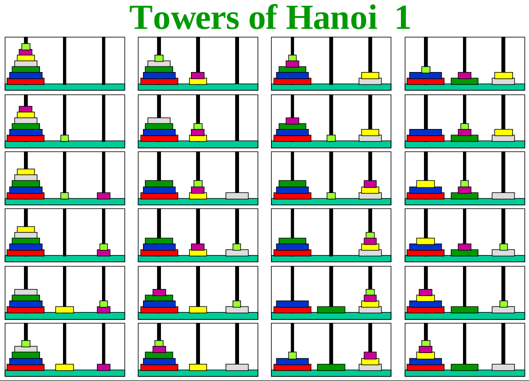
- Move 1 disk from tower 1 to 2;



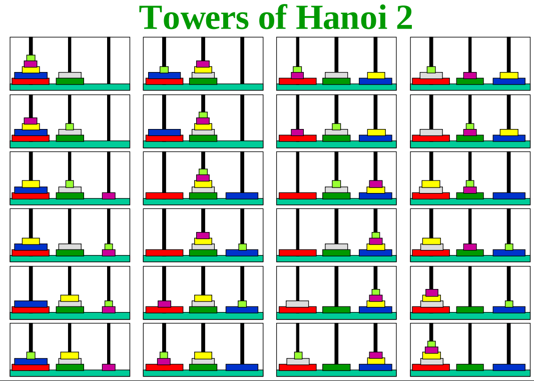
- Move n-1 disks from tower 3 to 2.



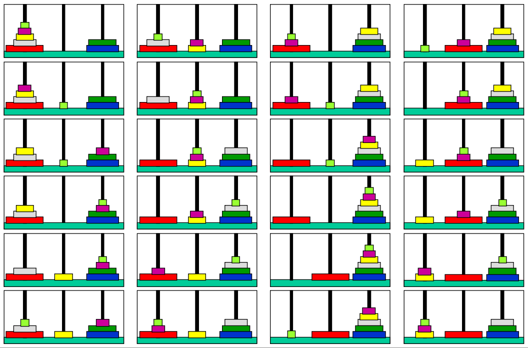
Towers of Hanoi 1



Towers of Hanoi 2



Towers of Hanoi 3



Towers of Hanoi 4