SOFT1x02 Recursion 2

School of Information Technologies

Today's Lecture

- More on recursion

Towers of Hanoi

At a remote temple somewhere in Asia, a group of monks is working to move 64 disks from the leftmost peg (peg A) to the rightmost peg (peg C). After all 64 disks have been moved, the universe will dissolve! When will this happen if each disk takes 1 second to move?

- Rules of the puzzle:
  - **Only one topmost disk may be moved at a time.**
  - **Larger disks may not be placed on top of smaller disks.**

Towers of Hanoi

- For n=64, number of moves = 2^{64}-1 ≅ 1.844× 10^{19}.
- Astronomers estimate the present age of the universe to be 20 billion (0.2× 10^{11}) years.
- Recall that one move takes one second.
- **Time left in universe = 1.844× 10^{19} seconds = 5.85× 10^{11} years** (585 billion years)**.**

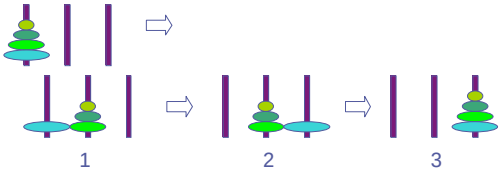
Tower of Hanoi (Non-Recursive)

- At the beginning of the subroutine, code is inserted which declares *stacks* associated with each formal parameter, each local variable, and the return address for each recursive call. Initially all stacks are empty.
- The label 1 is attached to the first executable program statement.
- If the subroutine is a *function*, i.e., returns some value, then we must replace all return statements with *assignment* statements, i.e., we introduce a fresh variable, say z, which has the same type as that of the function, and replace each return e statement with a z:=e statement. Now, each recursive call is replaced by a set of instructions which do the following:
- Store the values of all parameters and local variables in their respective stacks. The *stack pointer* is the same for all stacks.
- Create the i-the label, i, and store **i** in the address stack. The value **i** of this label will be used as the return address. This label is placed in the subroutine as described in rule 3.
- Evaluate the arguments of this call (they may be expressions) and assign these values to the appropriate formal parameters.
- Insert an unconditional branch to the beginning of the subroutine.
- If this is a procedure, add the label created above to the statement immediately following the unconditional branch. If this is a function then follow the unconditional branch by code to use the value of the variable z in the same way a return statement was handled earlier. The first statement of this code is given the label that was created above. These steps are sufficient to remove all recursive calls from a subroutine. Finally, we need to append code just after the last executable statement to do the following:
- If the recursion stacks are empty, then return the value of z, i.e., return z, in case this is a function, or else simply return.
- If the stacks are not empty, then restore the value of all parameters and of all local variables. These are at the top of the each stack. Use the return label from the corresponding stack and execute a branch to this label. This can be done using a switch statement.

Towers of Hanoi recursively

N disks to be moved from source to dest using aux for temporary moves

1. Move N-1 top disks from source to aux using dest
2. Move (bottom) disk from source to dest
3. Move the N-1 disks from aux to dest using source



Recursive Algorithm for Hanoi Towers

- Hanoi(disks, source, dest, aux): move top disks from source to dest using aux
- Hanoi(disks, source, dest, aux)
  - If disks is zero
    - All done, return
  - Else
    - Hanoi(disks - 1, source, aux, dest)
    - Move the single disk from source to dest
    - Hanoi(disks - 1, aux, dest, source)

Direct conversion of recursive sequence definition

```
int fact(int n) {
  if (n==1)
    return 1;
  else
    return n*fact(n-1);
}
```

FACTORIAL

$fact_n = \begin{cases} 1 \text{ if } n=1; \\ n \cdot fact_{n-1} \text{ if } n>1. \end{cases}$

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

FIBONACCI

$fib_n = \begin{cases} 1 \text{ if } n=1,2; \\ fib_{n-1}+fib_{n-2} \text{ if } n>2. \end{cases}$

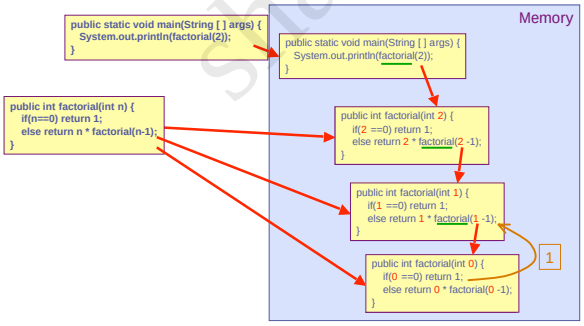
Mechanics of Method Calls

- When *any* method is called:
  - Fresh space (memory) is set aside for the method's parameters and local variables
    - Parameters are assigned the values given in the call
    - Local variables are created
  - The method body is executed
  - After body completes, execution resumes in the caller
    - return value of called method can be used in calling expression
    - local storage for called method is no longer accessible
- Recursive methods are no different.

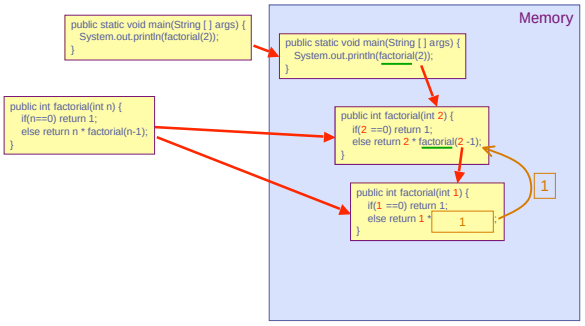
The program stack

- Each time a recursive method calls itself, current information is pushed onto the program stack.
- The *maximum depth* of the recurrence affects how much memory we need.
- The *number of calls* affects how long it will take!

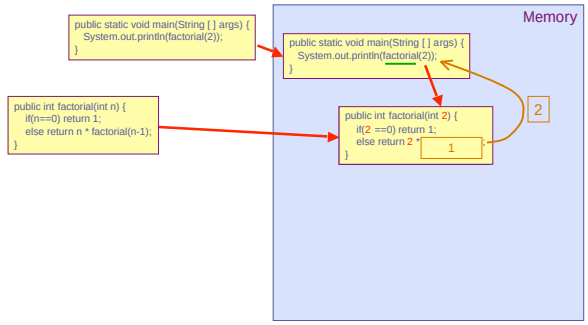
Executing factorial(2)



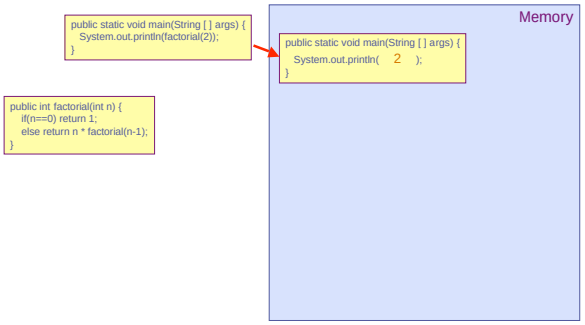
Executing factorial(2)



Executing factorial(2)



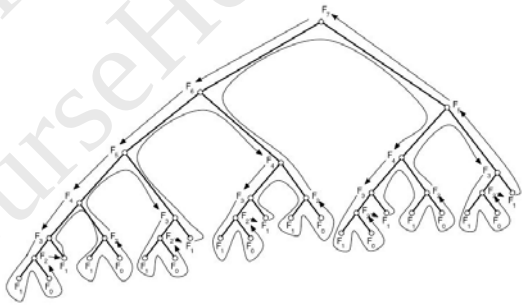
Executing factorial(2)



Tracing the stack

- We now trace the method calls for a Fibonacci number…

The recursion tree for F_7



Stack analysis

- For Fibonacci sequence, what is the maximum *depth* of our code?
  - F_n calls F_{n-1}, F_{n-2}, right down to F_1: depth n
- How *long* will it take?
  - Suppose F_n calls the Fibonacci method g(n) times.
  - Then g(n) = g(n–1) + g(n–2), and g(1) = g(0) = 1 so
  - g(n) = F_n – yuck!
- This recursive method for Fibonacci takes AGES (there is a faster way, using iteration).

Conquering the stack

- The stack analysis shows that if a recursive function for f(n) has f(n–1) in it, the depth will be O(n).
- Recursive methods that reduce a problem's size by a constant amount will always suffer this way.
- Methods that *divide* a problem into parts are faster and can require less space.

Recursive algorithms

- "Divide and conquer" is a widely-used way to solve algorithmic problems
- To solve the problem in one case X
  - find some smaller cases (X_1, X_2, etc) of the same problem
  - solve them
  - combine the solutions somehow to get a solution of the problem for X

Recursive binary search

- To search for a value in a segment of a sorted array,
- look at the middle element, and then search only half the segment.
- If array[middle] < x, we know that x can't occur in part of array before middle, so search in half-segment above middle;
- if array[middle] > x, search in half-segment below middle.

Recursive Binary Search code

```
// return an index between left and right inclusive where
// the sortedarray has value x
// return -1 if there is no such index
int binsearch (int x, int[] sortedArray, int left, int right) {
   if (left > right) return -1; // empty segment of the array
   if (left == right) {         // array segment with one element
      if (sortedarray[left] == x) return left;
      else return -1;
   }
   else { // segment of the array has 2 or more elements
     int middle = (left + right) / 2;
     if (x == sortedArray[middle])
        return  middle;
     else if (x < sortedArray[middle])
        return  binsearch (x, sortedArray, left, middle-1);
     else if (x > sortedArray[middle])
        return binsearch (x, sortedArray, middle+1, right);
   }
}
```

Warning: this algorithm is easy to code wrongly! Always think about boundary cases carefully.

Recursion vs Iteration

```
/**
 * Recursive Solution
 */
public int factorial(int n) {
   if(n==0) return 1;
   else return n * factorial(n-1);
}
```

```
/**
 * Iterative Solution
 */
public int factorial(int n) {
   int result = 1;
   for(int i=n; i>0; i - -)
      result = result * i;
   return result;
}
```

- As a general rule, use recursion over iteration when recursion provides a significantly more elegant solution.

Recursion and Scalability

- A recursive solution is ***not*** any better than the iterative solution in terms of run time cost.
- Bad recursion may lead to exponential running times (example: fibonacci)
- Recursive solutions may have an overhead compared to iterative solutions.
- But they may offer a ***neat*** answer.

Recursion for general programming

- Some "functional" programming languages don't use iteration at all, but instead use recursion for these purposes
  - eg Haskell, ML, Scheme, LISP
- They treat a loop as "do one turn, then do rest of loop". Weird but true.

Exercise

- Write a program that checks whether a word is a palindrome
- Examples:
  - tattarrattat – the longest palindrome in the Oxford English Dictionary, coined by James Joyce in Ulysses for a knock on the door
  - aibohphobia – a joke word meaning "fear of palindromes", deliberately constructed so as to be one

What have we done?

- Recursion
  - mechanism
  - code for recursively defined sequences
  - algorithms
- For this week: read Kingston ch 19 on Trees