CSC 1052 – Algorithms & Data Structures II:

Avoiding Recursion

Professor Henry Carter
Spring 2017

Project 1 graded

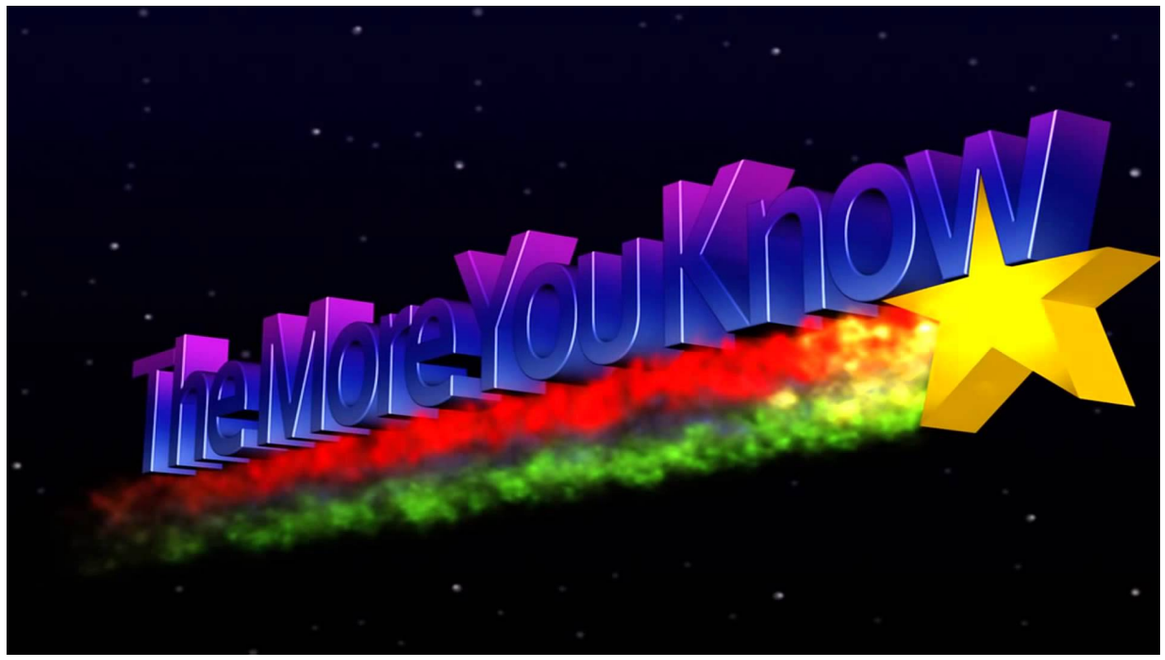
- Grades are posted on Blackboard
- Breakdown:
  - Submitted code and write up: 60
  - Submitted compiling code: 70
  - Submitted compiling code that returns the correct song order and plays: 80
  - Passed additional method testing: 100

Lessons learned

- "Simple concepts are not simple to implement"

- "Writing out functionality was helpful

- "Testing was difficult"



Recap

- Recursion is an obvious solution for many mathematical problems
- Recursion may be able to solve some problems in more subtle ways
  - Towers of Hanoi
- Recursion can be used as a tool on its own
  - Generating recursive structures in fractals

Is recursion right for me?

- Recall: the hidden cost of recursion is the added space required to store each call
- Some languages do not support recursion at all
- Some applications are simplified but needlessly inefficient when using recursion

How memory is allocated

- Statically vs Dynamically
- Static allocation:
  - Program is compiled into machine code
  - Additional space is reserved along with machine instructions for variables

Static Code

```
public class Kids
{
  private static int countKids(int girlCount, int boyCount)
  {
    int totalKids;
    . . .
  }

  public static void main(String[] args)
  {
    int numGirls; int numBoys; int numChildren;
    . . .
  }
}
```

Static Variable Allocation

- Space for the main() method
  - Space for variables
  - Code to initialize variables
  - Jump statement
  - Print statement code
  - Exit
- Space for the countKids() method
  - Space for variables
  - Method code
  - Return statememt

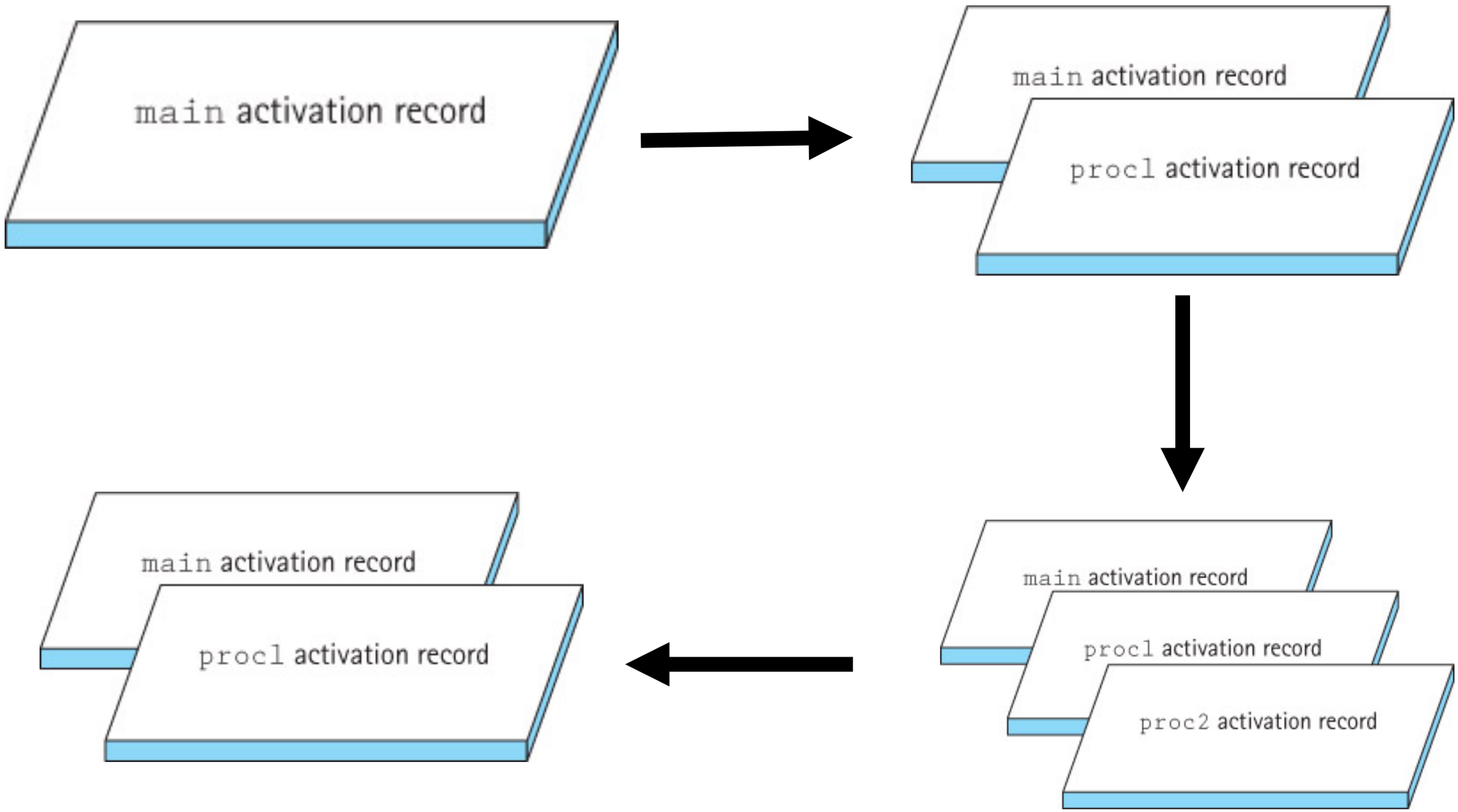
What's wrong?

- Space allocated for each method is fixed in size
- Assumption: method will be called once and return before being called again
- Recursion breaks this assumption, so *purely* statically allocated languages don't allow recursion

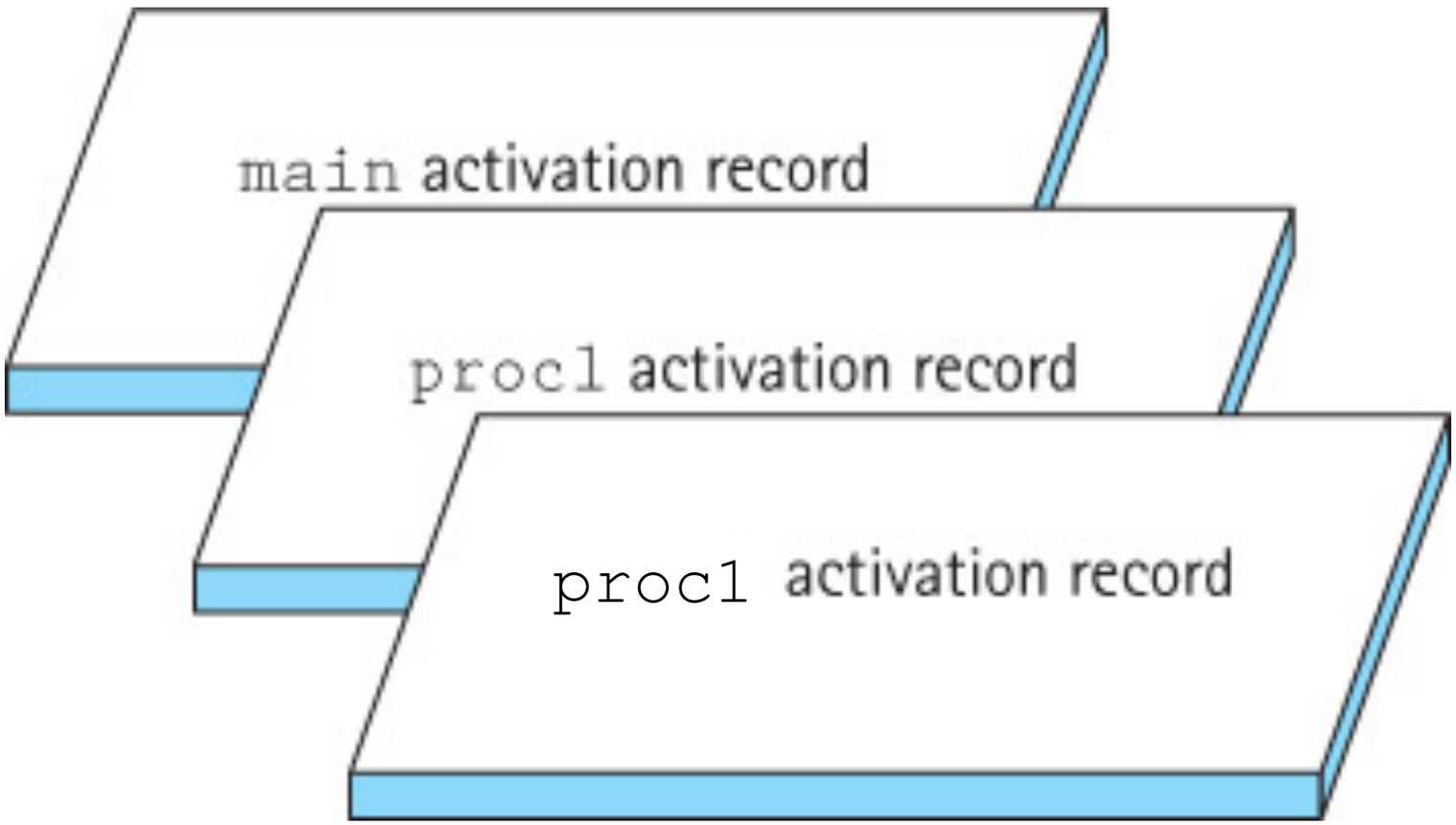
Dynamic allocation

- Method storage is allocated when the method is called
- Parameters, variables, and return address are nested in an activation record
- Activation records are stored on a stack to ensure they are handled and return in the correct order

Example Call Stack



Example Recursive Stack



Call Stack

- Stack ensures that all required return statements are handled in the correct order
- Depth of recursion refers to the depth of the call stack required
- Creating, storing, and handling activation records is the major reduction to efficiency in recursion

When to avoid recursion

- Some recursive functions have easily replicable functionality in a loop
- Some recursive functions can be emulated by explicitly using a stack
- Some recursive functions unnecessarily repeat computation

Recursion Overhead

- Traditional recursion calls the recursive function, performs an operation, and returns the result
- Note the nested structure requires all of the activation records to be maintained
- Requires stack space O(n) for n recursive calls

Example Traditional Call

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

Tail Recursion

- Makes the recursive call last, passing a running tally into the call parameters
- After the base case is reached, call frames simply return in sequence
- Smart compilers account for this and deallocate activation records that aren't needed
- Requires stack space O(1) if optimized

Example Tail Recursive Call

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

Converting to a loop

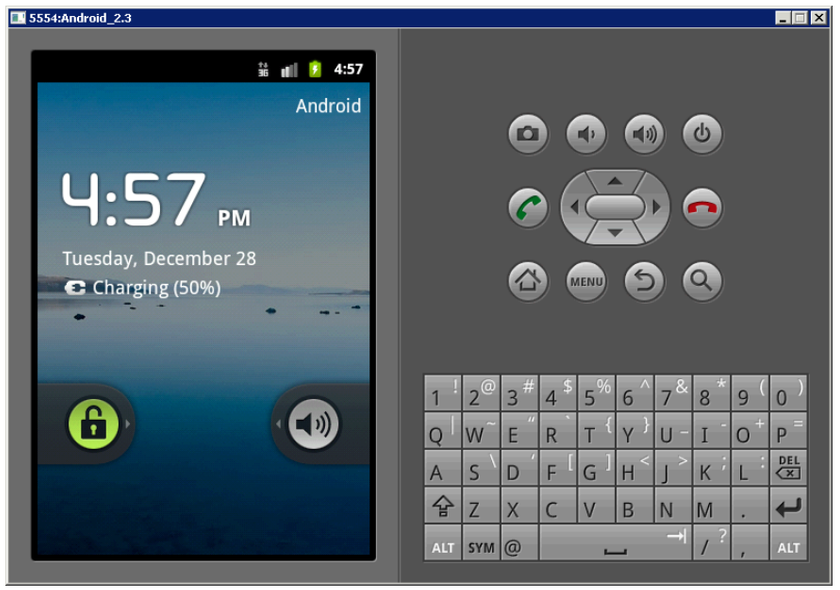
- Tail recursion can often be converted into a loop
- Instead of passing the running total into a recursive call, maintain an explicit variable
- Loop over the input value until the base case is reached

Example Loop Call

```
private static int factorial(int n)
{
   int retValue = 1;    // return value
   while (n != 0)
   {
      retValue = retValue * n;
      n = n - 1;
   }
   return(retValue);
}
```

Stack Emulation

- Recursion works because of the implicit use of a call stack
- Explicit use of a stack in a loop can emulate this behavior
- Requires additional complexity in the implementation code and data structure code
  - We're doing what the OS was doing for us in recursion



Recall recursive reverse print

```
void recPrintList(LLNode<String> listRef)
{
    if (listRef != null)
    {
        recPrintList(listRef.getLink());
        System.out.println(listRef.getInfo());
    }
}
```

Loop and stack version

```
static void iterRevPrintList(LLNode<String> listRef)
// Prints the contents of the listRef linked list to standard output
// in reverse order
{
   StackInterface<String> stack = new LinkedStack<String>();
   while (listRef != null) // put info onto the stack
   {
      stack.push(listRef.getInfo());
      listRef = listRef.getLink();
   }

   // Retrieve references in reverse order and print elements
   while (!stack.isEmpty())
   {
      System.out.println(stack.top());
      stack.pop();
   }
}
```

Efficiency issues: repeated work

- Recursive solutions are often clearer but may repeat calls
- Example: Combinations
- Recall, choosing k different elements from an n element pool

Combinations

```
public static int combo(int n,int k)
{
   if (k == 1)
      return n;
   else if (k == n)
      return 1;
   else
      return combo(n-1,k-1) + combo(n - 1,k);
}
```

Example: 4 choose 3

Example: 6 choose 4

Solutions

- Remember previously calculated steps
  - Dynamic Programming
- Closed form solution
  - N choose k = N!/(N-k)!k!

- Note that the closed form solution (factorial) can be easily implemented with a loop

Practice

- Convert traditional recursion to tail recursion:
  - Exponentiation
  - Summation
  - Multiplication
- Convert the same functions to a loop

Recap

- Variable space can be allocated dynamically or statically
- Dynamic allocation is critical to allow recursion but represents a hidden cost
- Recursion can be eliminated from several inefficient scenarios:
  - Tail recursion or looping
  - Explicit use of a stack
  - Closed form solutions or dynamic programming

Next Time...

- Dale, Joyce, Weems Chapter 4.1-4.2
  - Remember, you need to read it BEFORE you come to class!
- Check the course webpage for practice problems
- Peer Tutors
  - http://www.csc.villanova.edu/help/