Announcements

• Mid-term exam is scheduled on Thurs, Mar 11\textsuperscript{th}. In class, closed-book.
• Bring a pen; no calculator needed.
• Get prepared by reviewing quizzes, assignments, and slides; and referring to textbook.
• There will be a review lecture on Mar 9\textsuperscript{th}.

• My office hour today is moved to 3—4pm.
CSE 187: Programming with Data Structures (Spring 2010)

Lecture 12: Recursion 1

Rui Wang
Topics

- Triangular numbers
- Factorials
- Permutation
- Recursive Binary Search
- The Towers of Hanoi
- Fractals
- Eliminating Recursion
What is Recursion?

• A method (function) that calls itself.
  ```
  void recursiveMethod() {
      ...
      recursiveMethod();
  }
  ```

• A conceptually simple way to solve many practical problems.
  – But may sacrifice some computational efficiency.
What is Recursion?

• Recursion that involves more than one method:
  
  ```java
  void method1() {
    ... ...
    method2();
  }
  
  void method2() {
    ... ...
    ... ...
    method1();
  }
  ```
Triangle Numbers

• A series of numbers:
  1, 3, 6, 10, 15, 21, ...

  what’s the pattern? what’s the next number?
Triangle Numbers

- A series of numbers:
  1, 3, 6, 10, 15, 21, ...
  what’s the pattern? what’s the next number?

- 1, 1+2, 1+2+3, 1+2+3+4, 1+2+...+5, ...
  ① ② ③ ④ ⑤

- triangle(n)
Triangle Numbers

• A series of numbers:
  1, 3, 6, 10, 15, 21, ...
what’s the pattern? what’s the next number?
Triangle Numbers

- We can calculate the triangle number using a loop:

```c
int triangle(int n)
{
    int total = 0;

    while(n > 0) // until n is 1
    {
        total = total + n; // add n (column height) to total
        --n; // decrement column height
    }

    return total;
}
```
Triangle Numbers

• But we can also use recursion:

\[ \text{triangle}(n) = n + \text{triangle}(n-1); \]
Triangle Numbers

• But we can also use recursion:
  
  ```c
  int triangle(int n)
  {
      return (n + triangle(n-1));
  }
  ```

• Think about “passing the buck”
  – Transferring responsibility.
  – But when does buck stop?
Triangle Numbers

• But we can also use recursion:

```c
int triangle(int n)
{
    return (n + triangle(n-1));
}
```

• The buck stops here:
  – When n=1 we know the answer is 1.
  – In this case, we return without making a further recursive call.
  – This is called the **base case**.
Triangle Numbers

```
int triangle(int n)
{
    if (n==1)
        return 1;
    else
        return (n + triangle(n-1));
}
```

- Every recursive method must have a **base case**, otherwise it can go into infinite recursion.
Triangle Numbers

• Note that the base case must exist for all cases:

```c
int sum2(int n)
{
    if (n==1)
        return 1;
    else
        return (n + sum2(n-2));
}
```

What happens if n is an even number?
Triangle Numbers

- Code
- Try to remove the base case, what run-time error will you get?
Triangle Num

• What’s really happening during recursion?
• Imagine there are multiple incarnations of the same method simultaneously.
• Triggers a series of returns.
Triangle Numb

- Remember we talked about **stack frames** of a program.
- Upon **calling** a method, the local variables of the current method (caller) are preserved in the stack; then the stack pointer moves to the new frame (callee).
- Upon **returning**, the stack pointer moves back to the caller’s frame, allowing the computation to continue there.
Triangle Numbers

• This kind of calling mechanism is the same for all method calls, whether recursive or non-recursive.
• Therefore to the compiler there is no real distinction between a recursive or non-recursive method: they are treated in the same way.
Recursion

• **Characteristics:**
  – It calls itself.
  – When it calls itself, it does so to solve a smaller problem.
  – There is a base case, which returns without calling the method itself.
Recursion

• **Is it efficient?**
  
  – Calling a method involves overhead (control transfer, storing arguments, stack space).
  
  – In many cases, recursion can be converted to a loop using an explicit stack.

  – But we use it because it *simplifies a problem conceptually* (thus making the programmer’s job easier), not because of its efficiency.
Factorial

• Similar to the triangle numbers, except using multiplication instead of addition.

\[ n! = 1 \times 2 \times 3 \ldots \times n \]

• Formula:

\[
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
\[
\text{factorial}(1) = 1;
\]

What about \text{factorial}(0)?
int factorial(int n)
{
    if (n==0)
        return 1;
    else
        return (n * factorial(n-1));
}

• Factorial grows extremely rapidly. Where is it in terms of order:

\[ O(2^n) < O(n!) < O(n^n) \]
Fibonacci Numbers

• Recursive formula:

\[ F(n) = F(n-1) + F(n-2) \]
\[ F(0) = 0, \quad F(1) = 1 \]

• 0, 1, 1, 2, 3, 5, 8, 13, .....
Fibonacci Numbers

```c
int F(int n)
{
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return F(n-1)+F(n-2);
}
```
Anagrams (Permutation)

- **Problem**: given an English word (which contains no duplicate character), find all the **anagrams** of the word (whether they make a word or not)

- Example: **cat**
  - cat, cta, atc, act, tca, tac

- Basically just permutate all the characters.

- How many anagrams are there?
Anagrams (Permutation)

• **Problem**: given a string (which contains no duplicate character), find all the anagrams of the string (whether they make a word or not)

• Example: **cat**
  
cat, cta, atc, act, tca, tac

• Basically just permutate all the characters.

• How many anagrams are there?
  – It’s **factorial**!
Anagrams (Permutation)

• How do you write a program to print out all the permutations?
Anagrams (Permutation)

• How do you write a program to print out all the permutations?

• **Idea using recursion:**
  For a string of length n:
  – If n==1, there is only one trivial permutation.
  – Otherwise, pick one character as head, from all n characters; then recursively permutate the remaining n-1 characters.
Anagrams (Permutation)

- The solution in the textbook uses ‘rotation’ to achieve the ‘pick one character as head’ step.
- Demo
Recursive Binary Search

- Remember the **binary search** we learned in the sorted array lecture?
- The original version uses a loop, where each iteration reduces the search range by half.
- We can transfer this loop-based method into a recursive method.
- Code: loop-based vs. recursive
Recursive Binary Search

• What are the differences of the two?
Recursive Binary Search

• What are the differences of the two?
  – **While** loop replaced by a recursive call to `recFind()`.
  – The loop-based version **returns directly** to the caller; the recursive version **returns from only one level** of recursion (and eventually returns to the caller after a series of returns).
Divide and Conquer

• The recursive binary search is an example of **divide and conquer**:  
  – A big problem is divided into two problems with smaller sizes (**sub-problems**).  
  – To solve a sub-problem, you **again** divide it into two even smaller problems.  
  – The process continues until you get to the **base case**, which can be solved trivially.
Divide and Conquer

• The divide-and-conquer approach usually involves a method that contains two recursive calls to itself, one for each subproblem.

• In some cases (i.e. binary search), only one of them is actually executed.

• In other cases, both are executed.