Recursion Overview

- Recursion is a powerful technique for specifying functions, sets, and programs
  - factorial
  - combinations
  - exponentiation (raising to an integer power)

- Example recursively-defined functions and programs
  - factorial
  - combinations
  - exponentiation (raising to an integer power)

- Example recursively-defined sets
  - grammars
  - expressions
  - data structures (lists, trees, ...)

The Factorial Function \((n!\))

- Define \(n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1\) read: “\(n\) factorial”
  - E.g., \(3! = 3 \cdot 2 \cdot 1 = 6\)
- By convention, \(0! = 1\)
- The function int \(\rightarrow\) int that gives \(n!\) on input \(n\) is called the factorial function
- \(n!\) is the number of permutations of \(n\) distinct objects
  - There is just one permutation of one object. \(1! = 1\)
  - There are two permutations of two objects: \(2! = 2\)
  - There are six permutations of three objects: \(3! = 6\)
- If \(n > 0\), \(n! = n \cdot (n-1)!\)

Permutations of non-orange blocks

Each permutation of the three non-orange blocks gives four permutations when the orange block is included

Total number = \(4 \cdot 6 = 24 = 4!\)
A Recursive Program

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

0! = 1
n! = n*(n-1)!, n > 0

Execution of fact(4)
fact(4) → 24
  fact(3) → 6
    fact(2) → 2
      fact(1) → 1
      fact(0) → 1

The Fibonacci Function

• Mathematical definition:
  fib(0) = 0
  fib(1) = 1
  fib(n) = fib(n-1) + fib(n-2), n > 2

• Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, …

static int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}

Recursive Execution

static int fib(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}

General Approach to Writing Recursive Functions

1. Try to find a parameter, say n, such that the solution for n can be obtained by combining solutions to the same problem using smaller values of n (e.g., (n-1)!)  
2. Find base case(s) – small values of n for which you can just write down the solution (e.g., 0! = 1)
3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

Combinations (a.k.a. Binomial Coefficients)

• How many ways can you choose r items from a set of n distinct elements? \( \binom{n}{r} \) “n choose r”
• \( \binom{5}{2} = \binom{4}{1} + \binom{4}{2} \)

\( \binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}, n > r > 0 \)
\( \binom{n}{0} = 1 \)
\( \binom{n}{n} = 1 \)
Can also show that \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \)

\( \binom{0}{0} \) Pascal’s triangle
\( \binom{0}{1} \)
\( \binom{1}{1} \)
\( \binom{2}{0} \) \( \binom{2}{1} \) \( \binom{2}{2} \)
\( \binom{3}{0} \) \( \binom{3}{1} \) \( \binom{3}{2} \) \( \binom{3}{3} \)
\( \binom{4}{0} \) \( \binom{4}{1} \) \( \binom{4}{2} \) \( \binom{4}{3} \) \( \binom{4}{4} \)

\( \begin{array}{cccc}
  1 & 1 & 1 & 1 \\
  1 & 2 & 1 & 1 \\
  1 & 3 & 3 & 1 \\
  1 & 4 & 6 & 4 & 1 \\
\end{array} \)
Binomial Coefficients

- Combinations are also called binomial coefficients because they appear as coefficients in the expansion of the binomial power \((x+y)^n\):

\[
(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i
\]

Recursive Program for Combinations

\[
\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}, \quad n > r > 0
\]

Positive Integer Powers

- If \(a^n = a \cdot a \cdot \ldots \cdot a\) (\(n\) times)

Smarter Version in Java

```
static int power(int a, int n) {
    if (n == 0) return 1;
    int halfPower = power(a, n/2);
    if (n%2 == 0) return halfPower*halfPower;
    return halfPower*halfPower*a;
}
```
Implementation of Recursive Methods

- **Key idea:**
  - Use a stack to remember parameters and local variables across recursive calls
  - Each method invocation gets its own stack frame

- **A stack frame contains storage for**
  - Local variables of method
  - Parameters of method
  - Return info (return address and return value)
  - Perhaps other bookkeeping info

Example: power(2, 5)

- A new stack frame is pushed with each recursive call
- The stack frame is popped when the method returns
  - Leaving a return value (if there is one) on top of the stack

How Do We Keep Track?

- At any point in execution, many invocations of power may be in existence
  - Many stack frames (all for power) may be in Stack
  - Thus there may be several different versions of the variables a and n

- How does processor know which location is relevant at a given point in the computation?

**Answer:** Frame Base Register

- When a method is invoked, a frame is created for that method invocation, and FBR is set to point to that frame
- When the invocation returns, FBR is restored to what it was before the invocation
- How does machine know what value to restore in the FBR?
  - This is part of the return info in the stack frame

**Stacks**

- Like a stack of plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

**Stack Frame**

- A new stack frame is pushed with each recursive call
- The stack frame is popped when the method returns
  - Leaving a return value (if there is one) on top of the stack

**FBR**

- Computational activity takes place only in the topmost (most recently pushed) stack frame

Example: power(2, 5)
Conclusion

• Recursion is a convenient and powerful way to define functions

• Problems that seem insurmountable can often be solved in a "divide-and-conquer" fashion:
  ▪ Reduce a big problem to smaller problems of the same kind, solve the smaller problems
  ▪ Recombine the solutions to smaller problems to form solution for big problem

• Important application (next lecture): parsing