Recursion Overview

- Recursion is a powerful technique for specifying functions, sets, and programs
- 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(n-1)(n-2) ... 2 1
- By convention, 0! = 1
- The function int -> 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
  1 2
- There are six permutations of three objects: 3! = 6
  1 2 3
- If n > 0, n! = n(n-1)

A Recursive Program

```java
static int fact(int n) {
    if (n == 0) return 1;
    else return n*fact(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 = 4·6 = 24 = 4!

Announcements

- For extra Java help
  - Lots of consulting/office-hours are available
  - General Java-help is more easily available in week before assignment is due
  - Can set up individual meetings with TAs via email
- Check (soon!) that you appear correctly within CMS
  - Report problems to Kelly Patwell (Course Administrator)
- ACSU Event: Helen Newman lanes; 7-9pm tonight (?)
  - Free bowling; free pizza!
- Would you make use of consulting if it were
  - on North Campus (RPU)?
  - on West Campus?
- Academic Integrity Note
  - We treat AI violations seriously
  - The AI Hearing process is unpleasant
  - Please help us avoid this process by maintaining Academic Integrity
  - We test all pairs of submitted programming assignments for similarity
    - Similarities are caught even if variables are renamed
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. Figure out the 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

The Fibonacci Function

Mathematical definition:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2), \quad n \geq 2
\end{align*}
\]

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);
}

two base cases!

Fibonacci (Leonardo Pisano) 1170 − 1240?
Statue in Pisa, Italy Giovanni Paganucci 1863

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);
}

fib(4)
fib(3) fib(2)
fib(2) fib(1) fib(1) fib(0)
fib(1) fib(0)

Execution of fib(4):

Combinations (a.k.a. Binomial Coefficients)

How many ways can you choose r items from a set S of n distinct elements? \( \binom{n}{r} \) "n choose r"

\[ \binom{n}{r} = \text{number of 2-element subsets of S} = \{A,B,C,D,E\} \]

- 2-element subsets containing A: \( \binom{4}{1} \)
- 2-element subsets not containing A: \( \binom{5}{2}, \binom{4}{2}, \binom{3}{2}, \binom{2}{2}, \binom{1}{2} \)

Therefore, \( \binom{5}{2} = \binom{4}{1} + \binom{4}{2} \)

Binomial Coefficients

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

\[
(x+y)^n = \binom{n}{0} x^n + \binom{n}{1} x^{n-1}y + \binom{n}{2} x^{n-2}y^2 + \cdots + \binom{n}{n} y^n
\]

\[ = \sum_{i=0}^{n} \binom{n}{i} x^{n-i}y^i \]
Combinations Have Two Base Cases

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

Two base cases

- Coming up with right base cases can be tricky!
- General idea:
  - Determine argument values for which recursive case does not apply
  - Introduce a base case for each one of these
- Rule of thumb (not always valid): If you have \( r \) recursive calls on right hand side, you may need \( r \) base cases.

Positive Integer Powers

- \( a^n = a \cdot a \cdot a \cdots a \) (n times)

- Alternate description:
  - \( a^0 = 1 \)
  - \( a^{n+1} = a \cdot a^n \)

\[
\begin{align*}
\text{static int power(int a, int n) } & \quad \{ \\
& \quad \text{ if (n == 0) return 1; } \\
& \quad \text{ else return a*power(a,n-1); } \\
& \quad \}
\end{align*}
\]

A Smarter Version

- Power computation:
  - \( a^0 = 1 \)
  - If \( n \) is nonzero and even, \( a^n = (a^{n/2})^2 \)
  - If \( n \) is odd, \( a^n = a \cdot (a^{n/2})^2 \)
    - Java note: If \( x \) and \( y \) are integers, \( x/y \) returns the integer part of the quotient
- Example:
  \[ a^5 = a \cdot (a^{5/2})^2 = a \cdot (a^2)^2 = a \cdot ((a^2/2)^2)^2 \]
  Note: this requires 3 multiplications rather than 5!
- What if \( n \) were larger?
  - Savings would be more significant
  - This is much faster than the straightforward computation
    - Straightforward computation: \( n \) multiplications
    - Smarter computation: \( \log(n) \) multiplications

Smarter Version in Java

- \( n = 0: \ a^0 = 1 \)
- \( n \) nonzero and even: \( a^n = (a^{n/2})^2 \)
- \( n \) nonzero and odd: \( a^n = a \cdot (a^{n/2})^2 \)

\[
\begin{align*}
\text{static int power(int a, int n) } & \quad \{ \\
& \quad \text{ if (n == 0) return 1; } \\
& \quad \text{ int halfPower = power(a,n/2); } \\
& \quad \text{ if (n%2 == 0) return halfPower*halfPower; } \\
& \quad \text{ return halfPower*halfPower*a; } \\
& \quad \}
\end{align*}
\]

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
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)

Example: power(2, 5)

```
return info
(a = ) 2
(n = ) 5
(hP = ) 3

return info
(a = ) 2
(n = ) 5
(hP = ) 1

return info
(a = ) 2
(n = ) 5
(hP = ) 2

return info
(a = ) 2
(n = ) 5
(hP = ) 4

return info
(a = ) 2
(n = ) 5
(hP = ) 5
```

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

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