Recursion

• Recursion is a powerful technique for specifying functions, sets, and programs
  • factorials
  • combinations
  • differentiation of polynomials
  • Recursively-defined sets
  • grammars
  • expressions
  • data structures (lists, trees, ...)

The Factorial Function (n!)

• Define n! = n(n−1)(n−2)···2·1 read: "n factorial"
  • E.g., 3! = 3·2·1 = 6
  • 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
      1 3 2
      2 1 3
      2 3 1
      3 1 2
      3 2 1
    • 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);
}
```

Execution of fact(4)

0! = 1
n! = n(n−1)!, n > 0

Recursion

• Assignment 2 is online (since Friday)
  • Due date: Wednesday, September 14
  • Recommendation: Start now
• If you would like a partner for A2
  • Sign up sheet
  • Name and netID
• Be sure to “form your group” on CMS!
  • It does not happen automatically
• 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

Permutations of non-pink blocks

Each permutation of the three non-pink blocks gives four permutations of the four blocks.

Total number = 4·6 = 24 = 4!
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 with smaller values of \( n \) (e.g., chess-board tiling, factorial).
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 value of \( n \) of interest, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases.

The Fibonacci Function

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

  \( \text{fib}(n) \) is the \( n \)th Fibonacci number.

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

```java
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

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

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

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”

- \( \binom{5}{2} = \text{number of 2-element subsets of } S = \{A, B, C, D, E\} \)
- \( \binom{2}{2} = \text{number of 2-element subsets containing A: } \{A, B\}, \{A, C\}, \{A, D\}, \{A, E\} \)
- \( \binom{2}{2} = \text{number of 2-element subsets not containing A: } \{B, C\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\}, \{D, E\} \)

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

Combinations

- You can also show that

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

- \( \binom{n}{0} = 1 \)

- \( \binom{n}{n} = 1 \)

- \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \)
Combinations

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

Pascal’s triangle

\[
\begin{array}{cccccccc}
1 & & & & & & & \\
1 & 1 & & & & & & \\
1 & 2 & 1 & & & & & \\
1 & 3 & 3 & 1 & & & & \\
1 & 4 & 6 & 4 & 1 & & & \\
1 & 5 & 10 & 10 & 5 & 1 & & \\
\end{array}
\]

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

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

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

Combinations have two base cases

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

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.

Recursive Program for Combinations

```
static int combs(int n, int r){    //assume n>=r>=0
    if (r == 0 || r == n) return 1; //base cases
    else return combs(n-1,r) + combs(n-1,r-1); //recursive case
}
```

Polynomial Differentiation

Inductive cases:

\[
d(uv)/dx = u dv/dx + v du/dx \\
d(u+v)/dx = du/dx + dv/dx
\]

Base cases:

\[
dx/dx = 1 \\
dc/dx = 0
\]

Example:

\[
d(3x)/dx = 3dx/dx + x d(3)/dx = 3 + x \cdot 0 = 3
\]

```
Example:

\[
d(3x)/dx = 3dx/dx + x d(3)/dx = 3 + x \cdot 0 = 3
\]

Positive Integer Powers

\[
a^0 = a \cdot a \cdot a \cdot \ldots \cdot a \ (n \ times)
\]

Alternative description:

\[
a^0 = 1 \\
a^{n+1} = a \cdot a^n
\]

```
```
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, \( \lfloor x/y \rfloor \) returns the integer part of the quotient

- Example:
  \[ a^5 = a \cdot (a^2)^2 = a \cdot ((a^2/2)^2)^2 = a \cdot (a^2)^2 \]
  Note: this requires 3 multiplications rather than 5!

- What if \( n \) were higher?
  - savings would be higher

- 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: \( a^n = (a^{n/2})^2 \)
- \( n \) odd: \( a^n = a \cdot (a^{n/2})^2 \)

```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

- The method has two parameters and a local variable
- Why aren’t these overwritten on recursive calls?

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)

java.lang.Stack

- Stack() Creates an empty Stack
- boolean empty() Tests if the stack is empty
- E peek() Looks at the object at the top of the stack without removing it from the stack
- E pop() Removes the object at the top of the stack and returns that object as the value of the function
- push(E item) Pushes an item onto the top of the stack
- int size() Returns the position of the given item on the stack
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

Example: power(2, 5)

```
(a = 2) (n = 5) (hP = ?)
(a = 2) (n = 5) (hP = ?)
(a = 2) (n = 2) (hP = ?)
(a = 2) (n = 5) (hP = ?)
(a = 2) (n = 2) (hP = ?)
(a = 2) (n = 1) (hP = ?)
```

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

- Computational activity takes place only in the topmost (most recently pushed) stack frame
  - Special register called Frame Base Register (FBR) keeps track of where the topest frame is
- Using the FBR
  - 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 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 of languages