Recursion

Let us now study recursion in its own right. Recursion is a powerful technique for specifying functions, sets, and programs. Recursively-defined functions and programs:
- factorial
- counting combinations
- differentiation of polynomials
Recursively-defined sets:
- grammars
- language of expressions

Factorial function

- How many ways can you arrange \( n \) distinct objects? This function is called \( \text{fact}(n) \).
  - If \( n = 1 \), then there is just one way.
  - If \( n > 1 \), number of ways = \( n \)\* number of ways to arrange \((n-1)\) objects
    (see next slide for example)

\[
\begin{align*}
\text{fact}(1) &= 1 \\
\text{fact}(n) &= n \text{fact}(n-1) \quad (n > 1)
\end{align*}
\]
- Another description of \( \text{fact}(n) \):
  \( \text{fact}(n) = 1 \times 2 \times \ldots \times n = n! \)
- Convention: \( \text{fact}(0) = 1 \)

Permutations of non-green blocks

From each permutation of non-green blocks, we can generate 4 permutations of the four blocks.

Total number = 4\*6 = 24 = 4!
Recursive program: factorial

\[ \text{fact}(0) = 1 \]
\[ \text{fact}(n) = n \times \text{fact}(n-1) \quad (n > 0) \]

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

Execution of factorial(4)

General approach to writing recursive functions

1. Try to find a parameter of problem (say \( n \)) such that solution to problem can be obtained by combining solutions to same problem with smaller values of \( n \). (e.g.) chess-board tiling problem, factorial
2. Figure out base case or base cases by determining small enough values of \( n \) for which you can write down the solution to problem.
3. Verify that for any value of \( n \) of interest, applying the reduction step of step 1 repeatedly will ultimately hit one of the base cases.
4. Write the code.

Fibonacci function

- Mathematical definition:
  \[
  \text{fib}(0) = 1 \quad \text{fib}(1) = 1 \quad \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \quad | n > 1
  \]
  fibonacci sequence: 1,1,2,3,5,8,13,….

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

Statue of Fibonacci in Pisa, Italy
Recursively-defined functions: Counting Combinations

How many ways can you choose \( r \) items from a set \( S \) of \( n \) distinct elements? \( ^nC_r \)

**Example:**

\( S = \{A,B,C,D,E\} \)

Consider subsets of 2 elements.

Subsets containing \( A \): \( ^4C_1 \)

\( \{A,B\}, \{A,C\}, \{A,D\}, \{A,E\} \)

Subsets not containing \( A \): \( ^4C_2 \)

\( \{B,C\}, \{B,D\}, \{C,D\}, \{B,E\}, \{C,E\}, \{D,E\} \)

Therefore, \( ^5C_2 = ^4C_1 + ^4C_2 \)

Counting Combinations has two base cases

\( ^nC_r = n^{-1}C_{r-1} + n^{-1}C_{r-1} \) \( \quad | \text{ for } n > r > 0 \)

\( ^nC_n = 1 \)

\( ^nC_0 = 1 \)

Two base cases

- Coming up with right base cases can be tricky!
- General idea:
  - Figure out argument values for which recursive case cannot be applied.
  - 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 of function definition, you may need \( r \) base cases.
Recursive program: counting combinations

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

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

Polynomial differentiation

Recursive cases:
\[ \frac{d(uv)}{dx} = udv/dx + v du/dx \]
\[ \frac{d(u+v)}{dx} = du/dx + dv/dx \]
Base cases:
\[ \frac{dx}{dx} = 1 \]
\[ \frac{dc}{dx} = 0 \]

Example:
\[ \frac{d(3x)}{dx} = 3dx/dx + x \frac{d(3)}{dx} = 3*1 + x*0 = 3 \]

Positive integer powers

\[ a^n = a * a * ... * a \quad (n \text{ times}) \]
Alternative description:
\[ a^0 = 1 \]
\[ a^n = a * a^{n-1} \]

• Let us write this using standard function notation:
  power(a,n) = a*power(a,n-1) \quad | \quad n > 0
  power(a,0) = 1

Recursive program for power

static int power(int a, int n) {
    if (n == 0) return 1;
    else return a*power(a,n-1);
}
Smarter power program

- Power computation:
  - If \( n = 0 \), \( a^n = 1 \)
  - If \( n \) is non-zero and even, \( a^n = (a^{n/2})^2 \)
  - If \( n \) is odd, \( a^n = (a^{n/2})^2 \cdot a \)

- Java note: If \( x \) and \( y \) are integers, expression "\( x/y \)" returns the integer part of the quotient.

Example:
\[
a^n = (a^{n/2})^2 \cdot a = (a^{n/2})^2 \cdot a = ((a^{n/2})^2 \cdot a
\]

Note: this requires 3 multiplications rather than 5.

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

- We will see later that recursive power is “much faster” than straight-forward computation.
  - Straight-forward computation: \( n \) multiplications
  - Smarter computation: \( \log(n) \) multiplications

Smarter power program in Java

- If \( n \) is non-zero and even, \( a^n = (a^{n/2})^2 \)
- If \( n \) is odd, \( a^n = (a^{n/2})^2 \cdot a \)

```java
static int coolPower(int a, int n)
{
  if (n == 0) return 1;
  else
  {
    int halfPower = coolPower(a, n/2);
    if ((n/2)*2 == n)   //n is even
      return halfPower*halfPower;
    else //n is odd
      return halfPower*halfPower*a;
  }
}
```

Implementing recursive methods

- Ur-Java implementation model already supports recursive methods.

- Key idea:
  - each method invocation gets its own frame
  - frame for method invocation \( f \): bottom to top order
    - return value: where function return value is to be saved before returning to caller
      - lowest location of frame
    - on return, this location becomes part of frame of caller
  - method parameters
  - method variables

Suppose method \( f \) invokes method \( g(p1,p2,p3) \).
When \( g \) returns, it leaves its return value on top of stack.
Analogy: arithmetic expression evaluation
\((2 + 3) \) is implemented as
\[
PUSHIMM 2
\]
\[
PUSHIMM 3
\]
\[
ADD
\]

\[
Frame for
invocation of f
\]
\[
Frame for
invocation of g
\]
\[
Frame for
return value
\]
\[
Frame for
invocation of f
\]
Let us look at how stack frames are pushed and popped for execution of the invocation `power(5, 3)`.

At conceptual level, here is the sequence of method invocations:

```
power(5, 3)  Æ  power(5, 2)  Æ  power(5, 1)  Æ  power(5, 0)
```

```
public static int power(int b, int p){
    if (p == 0) return 1;
    else return power(b, p-1) * b;
}
```

**Exercise**

- Draw similar picture for execution of `fib(5)`.

**Something to think about**

- At any point in execution, many invocations of `power` may be in existence, so many stack frames for power invocations may be in stack area.
- This means that variables `p` and `b` in text of program may correspond to several memory locations at any time.
- How does processor know which location is relevant at any point in computation?
  - another example of association between name and “thing” (in this case, stack location)
Answer:
- Computational activity takes place only in the topmost (most recently pushed) frame.
- Special register called Frame Base Register (FBR) keeps track of where the topmost frame is.
  - 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?
    - See later

- In low-level machine code, addresses of parameters and local variables are never absolute memory addresses (like 102 or 5099), but are always relative to the FBR (like –2 from FBR or +5 from FBR).

Editorial comments

- Recursion is a very powerful way of defining functions.
- Problems that seem insurmountable can often be solved in a 'divide-and-conquer' way
  - Split big problem into smaller problems of the same kind, and solve smaller problems
  - Put solution to smaller problems together to form solution for big problem
- Recursion is often useful for expressing divide-and-conquer algorithms in a simple way.
- We will use parsing of languages to demonstrate this in the next lecture.

static int power(int b, int p)
    if (p == 0) return 1;
    else return b*power(b,p-1);

    p : 0
    b : 5
    rv : 1
    p' : 1
    b' : 5
    rv' : 1
    p'' : 2
    b'' : 5
    rv'' : 1
    i : 1

p : 1
b : 5
rv : 2