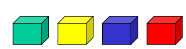
Recursion

Recursion

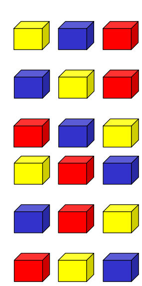
- Let us now study recursion in its own right.
- Recursion is a powerful technique for specifying functions, sets, and programs.
- Recursively-defined functions and programs
  - factorial
  - counting combinations
  - differentiation of polynomials
- Recursively-defined sets
  - grammars
  - language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

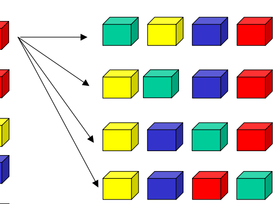
Factorial function

- How many ways can you arrange n distinct objects? This function is called *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)
  - $fact(1) = 1$
    $fact(n) = n*fact(n-1) \mid (n > 1)$
- Another description of *fact(n)*:
  $fact(n) = 1*2*\ldots*n = n!$
- Convention: $fact(0) = 1$

Permutations of 

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

fact(0) = 1

*fact(n) = n*fact(n-1) | (n > 0)*

factorial(0) 1
factorial(1) 1
factorial(2) 2
factorial(3) 6
factorial(4) 24

```
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*.
   (eg.) 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:

  fib(0) = 1
  fib(1) = 1   two base cases
  fib(n) = fib(n-1) + fib(n-2) | n > 1

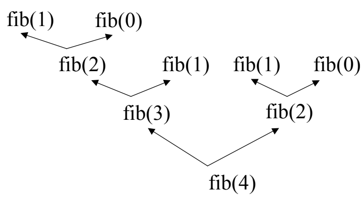
  fibonacci sequence: 1,1,2,3,5,8,13,….

```
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

Execution of fibonacci

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



Execution of fib(4)

Recursively-defined functions: Counting Combinations

How many ways can you choose r items from a set S of n distinct elements? $^{n}C_{r}$

Example:

S = {A,B,C,D,E}

Consider subsets of 2 elements.

Subsets containing A: $^{4}C_{1}$

{A,B}, {A,C}, {A,D},{A,E}

Subsets not containing A: $^{4}C_{2}$

{B,C},{B,D},{C,D},{B,E},{C,E},{D,E}

Therefore, $^{5}C_{2} = {}^{4}C_{1} + {}^{4}C_{2}$

Counting Combinations

$$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \quad | \quad n > r > 0$$
$$^{n}C_{n} = 1$$
$$^{n}C_{0} = 1$$

- How many ways can you choose r items from a set S of n distinct elements?
  - Consider some element A.
  - Any subset of r items from set S either contains A or it does not.
  - Number of subsets of r items that do not contain A = $^{n-1}C_{r}$.
  - Number of subsets of r items that contain A = $^{n-1}C_{r-1}$.
  - Required result follows.
- You can also show that

$$^{n}C_{r} = n!/r!(n-r)!$$

Counting combinations has two base cases

$$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \quad | \quad n > r > 0$$
$$^{n}C_{n} = 1$$
$$^{n}C_{0} = 1$$

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

$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \mid n > r > 1$

$^{n}C_{n} = 1$

$^{n}C_{0} = 1$

```
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:

d(uv)/dx = udv/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*1 + x*0 = 3

Positive integer powers

$a^n = a * a * .... * a$ (n 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) | n > 0

  power(a,0) = 1

Recursive program for power

power(a,n) = a*power(a,n-1) | n > 0

power(a,0) = 1

```
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 is 0, $a^n = 1$
  - If n is non-zero and even, $a^n = (a^{n/2})^2$
  - If n is odd, $a^n = (a^{n/2})^2 * a$
- Java note: If x and y are integers, expression "x/y" returns the integer part of the quotient.
- Example:

  $a^5 = (a^{5/2})^2 * a = (a^2)^2 * a = ((a^{2/2})^2)^2 * a$

  $= ((a)^2)^2 * a$

  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, $a^n = (a^{n/2})^2 * a$

```
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

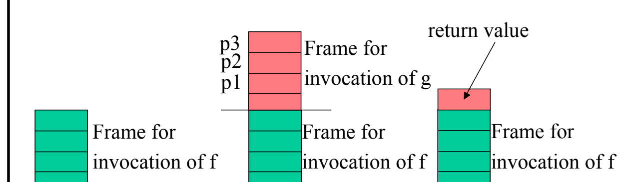
- Ur-Java implementation model already supports recursive methods.
- Key idea:
  - each method invocation gets its own frame
  - frame for method invocation *I:* bottom to top order
    - return value: where function return value is to be saved before returning to caller
      - lowest location of frame
      - 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



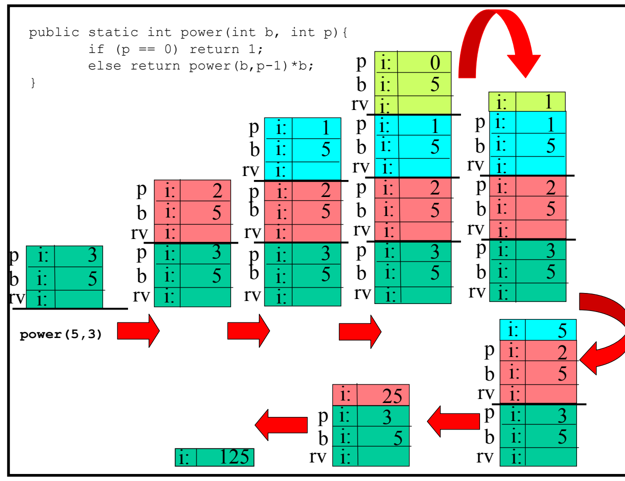
Let us look at how stack frames are pushed and popped for execution of the invocation power(5,3).

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

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

→ power(5,3) → power(5,2) → power(5,1) → power(5,0)

125   25   5   1



Exercise

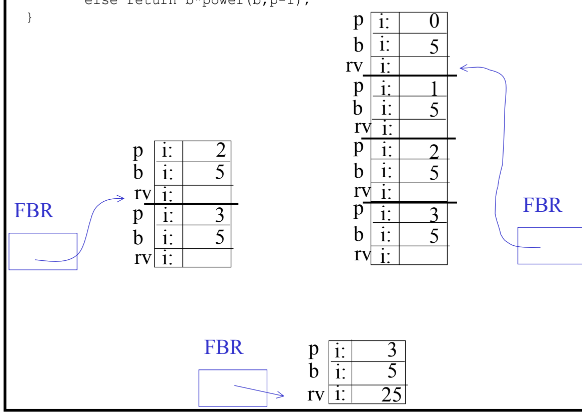
- Draw similar picture for execution of fib(5).

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).

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



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.