RECURSION

Lecture 6

CS2110 – Fall 2013

Overview references to sections in text

- Note: We've covered everything in JavaSummary.pptx!
- What is recursion? 7.1-7.39 slide 1-7
- Base case 7.1-7.10 slide 13
- How Java stack frames work 7.8-7.10 slide 28-32

Homework. Copy our "sum the digits" method but comment out the base case. Now run it: what happens in Eclipse?

Now restore the base case. Use Eclipse in debug mode and put a break statement on the "return" of the base case. Examine the stack and look at arguments to each level of the recursive call.

Recursion

- Arises in three forms in computer science
  - Recursion as a *mathematical* tool for defining a function in terms of its own value in a simpler case
  - Recursion as a *programming* tool. You've seen this previously but we'll take it to mind-bending extremes (by the end of the class it will seem easy!)
  - Recursion used to prove properties about algorithms. We use the term *induction* for this and will discuss it later.

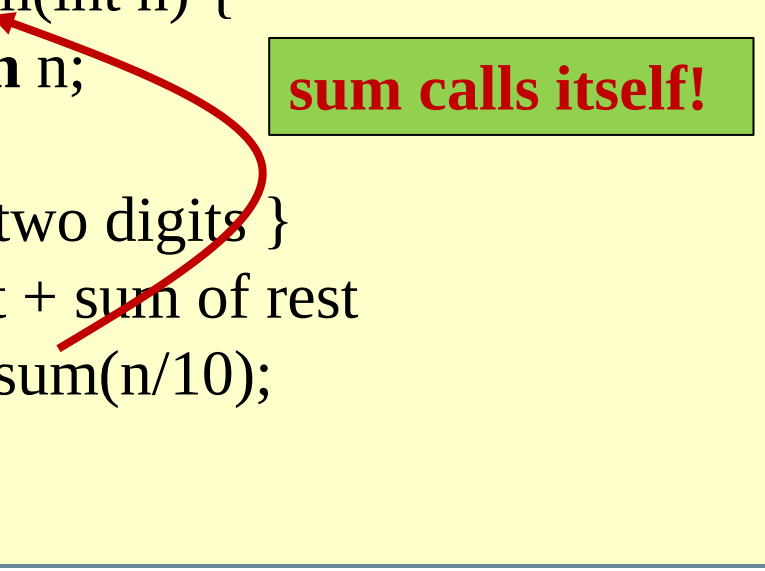
Recursion as a math technique

- Broadly, recursion is a powerful technique for specifying functions, sets, and programs
- A few recursively-defined functions and programs
  - factorial
  - combinations
  - exponentiation (raising to an integer power)
- Some recursively-defined sets
  - grammars
  - expressions
  - data structures (lists, trees, ...)

Example: Sum the digits in a number


```
/** return sum of digits in n, given n >= 0 */
 public static int sum(int n) {
     if (n < 10) return n;

    // { n has at least two digits }
    // return first digit + sum of rest
    return n%10  +  sum(n/10);
 }
```



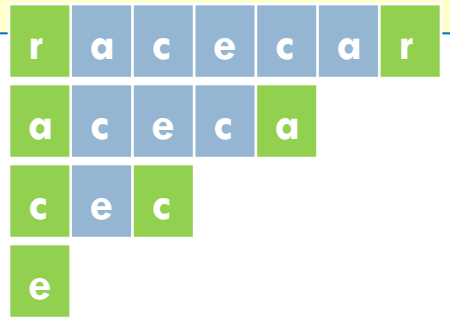
- E.g. sum(87012) = 2+(1+(0+(7+8))) = 18

Example: Is a string a palindrome?


```
/** = "s is a palindrome" */
public static boolean isPal(String s) {
    if (s.length() <= 1)
        return true;

    // { s has at least 2 chars }
    int n= s.length()-1;
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1, n));
}
```

Substring from s[1] to s[n-1]



- isPal("racecar") = true
- isPal("pumpkin") = false

Count the e's in a string


```
/** = "number of times c occurs in s */
public static int countEm(char c, String s) {
   if (s.length() == 0) return 0;

   // { s has at least 1 character }
   if (s.charAt(0) != c)
        return countEm(c, s.substring(1));

   // { first character of s is c}
   return 1 + countEm (c, s.substring(1));
}
```

Substring s[1..], i.e. s[1], …, s(s.length()-1)

- countEm('e', "it is easy to see that this has many e's") = 4
- countEm('e', "Mississippi") = 0

The Factorial Function (n!)

- Define n! = n·(n–1)·(n–2)···3·2·1
  read: "n factorial"
  E.g. 3! = 3·2·1 = 6

- Looking at definition, can see that n! = n * (n-1)!

- By convention, 0! = 1

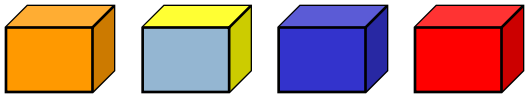
- The function int → int that gives n! on input n is called the factorial function

The Factorial Function (n!)

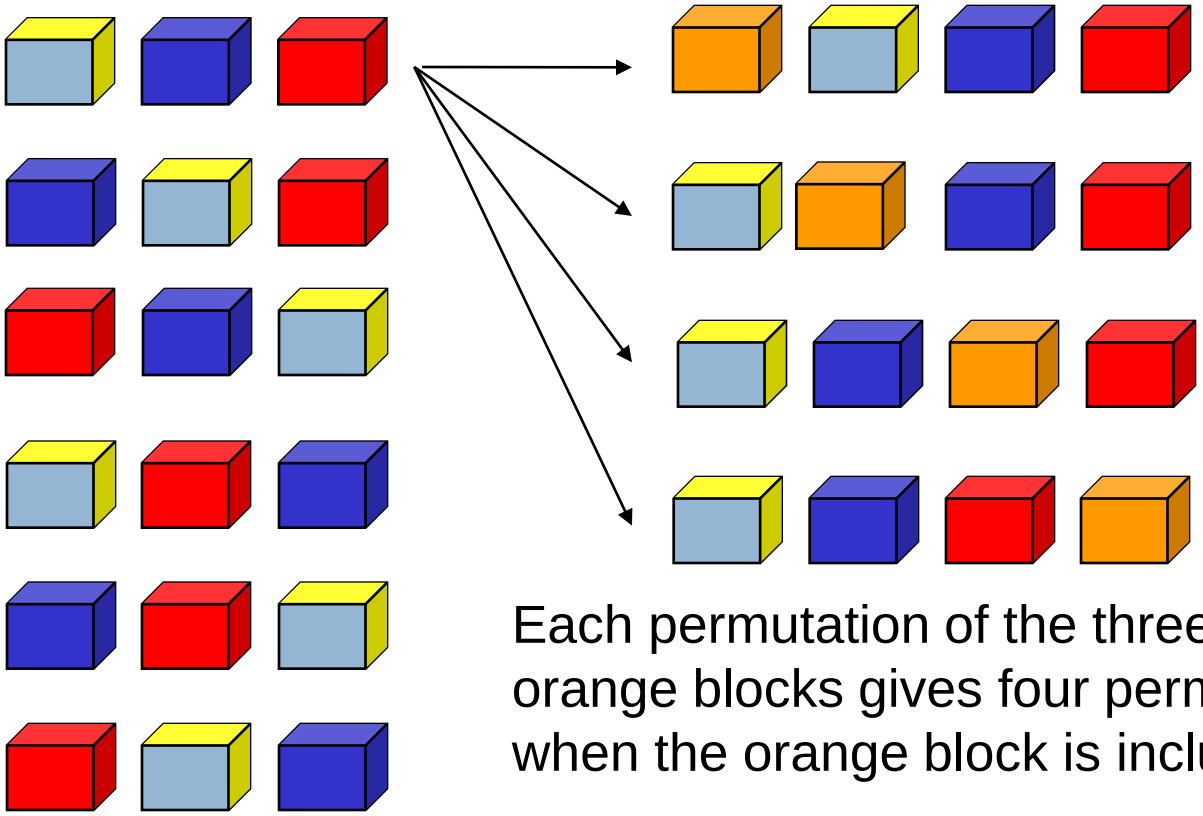
- n! is the number of permutations of n distinct objects
  - There is just one permutation of one object. $1! = 1$
  - There are two permutations of two objects: $2! = 2$

    1 2 2 1
  - 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 $n > 0$, $n! = n \cdot (n - 1)!$

Permutations of 

Permutations of
non-orange blocks



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

- Total number = 4·3! = 4·6 = 24: 4!

Observation

- One way to think about the task of permuting the four colored blocks was to start by computing all permutations of three blocks, then finding all ways to add a fourth block
  - And this "explains" why the number of permutations turns out to be 4!
  - Can generalize to prove that the number of permutations of n blocks is n!

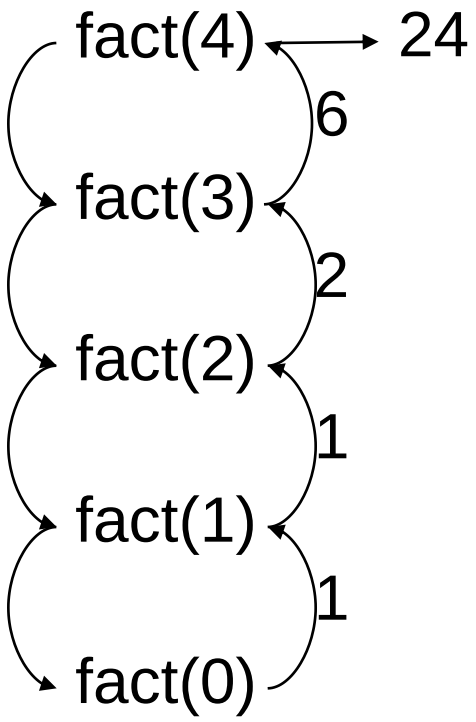
A Recursive Program

0! = 1

n! = n·(n–1)!, n > 0

```
/** = n!, for n >= 0 */
static int fact(int n) {
   if (n = = 0)
        return 1;
   // { n > 0 }
   return n*fact(n-1);
}
```

Execution of fact(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 using smaller values of n* (e.g. (n-1) in our factorial example)

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

A cautionary note

- Keep in mind that each instance of the recursive function has its own local variables
- Also, remember that "higher" instances are waiting while "lower" instances run

- Do not touch global variables from within recursive functions
  - Legal… but a common source of errors
  - Must have a really clear mental picture of how recursion is performed to get this right!

The Fibonacci Function

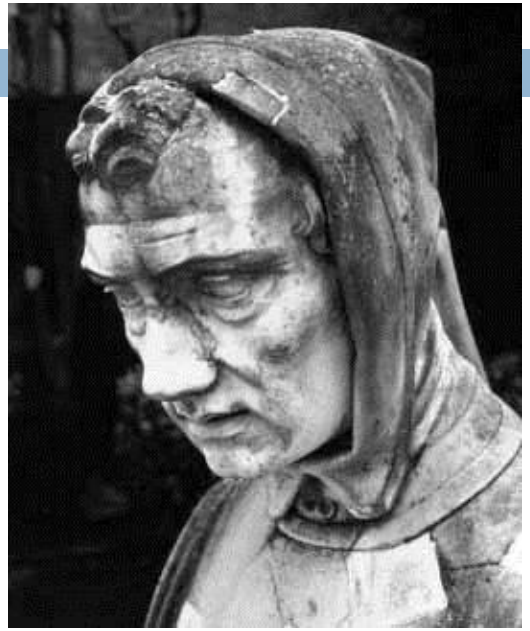
- **Mathematical definition:**

  fib(0) = 0
  fib(1) = 1
  fib(n) = fib(n – 1) + fib(n – 2), n ≥ 2

  two base cases!

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

```
/** = fibonacci(n), for n >= 0 */
static int fib(int n) {
   if (n <= 1) return n;
   // { 1 < n }
   return fib(n-2) + fib(n-1);
}
```



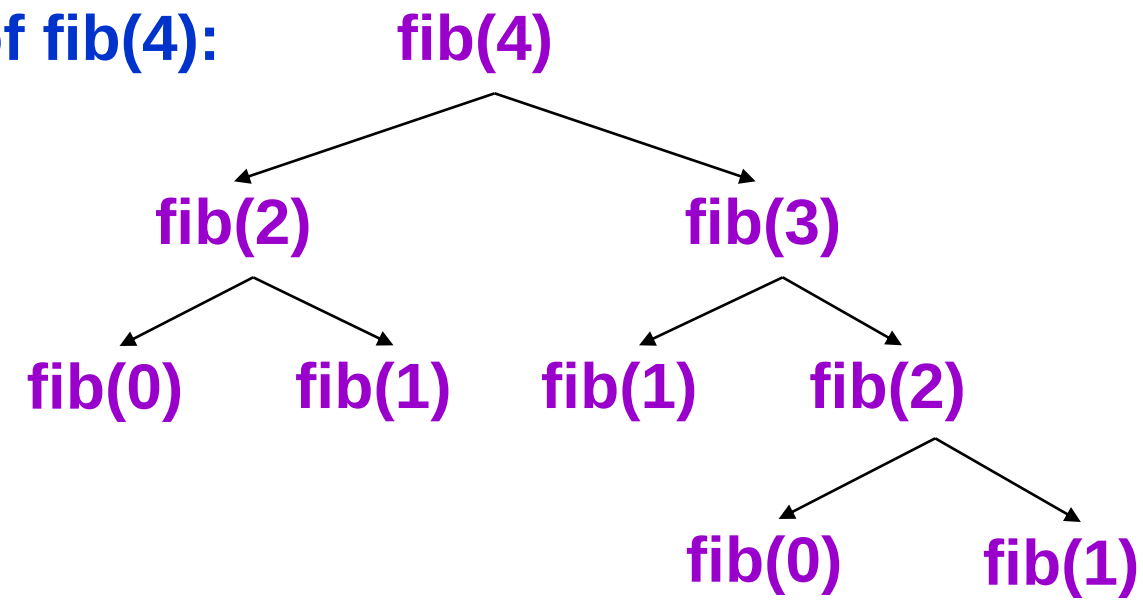
Fibonacci (Leonardo Pisano) 1170–1240?

Statue in Pisa, Italy
Giovanni Paganucci
1863

Recursive Execution

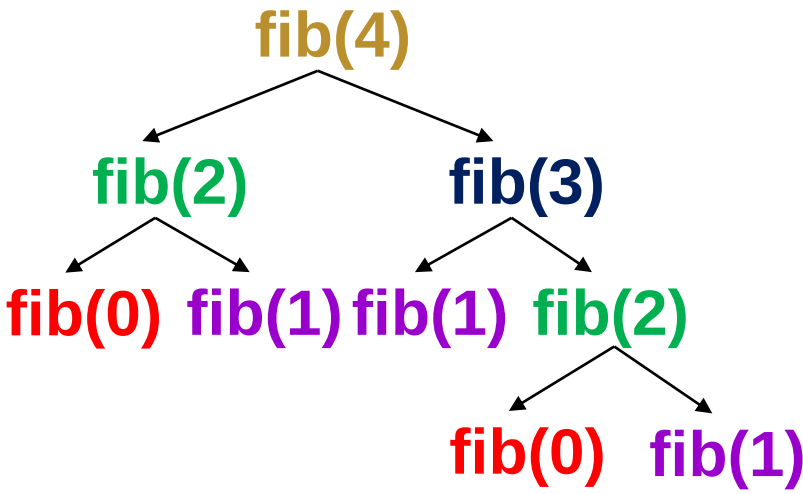

```
/** = fibonacci(n), for n >= 0 */
static int fib(int n) {
   if (n <= 1) return n;
   // { 1 < n }
   return fib(n-2) + fib(n-1);
}
```

Execution of fib(4):



One thing to notice

- This way of computing the Fibonacci function is elegant but inefficient
- It "recomputes" answers again and again!
- To improve speed, need to save known answers in a table!
  - One entry per answer
  - Such a table is called a *cache*



Memoization (fancy term for "caching")

- Memoization is an optimization technique used to speed up computer programs by having function calls avoid repeating the calculation of results for previously processed inputs.
  - The first time the function is called, save result
  - The next time, look the result up
    - Assumes a "side effect free" function: The function just computes the result, it doesn't change things
    - If the function depends on anything that changes, must "empty" the saved results list

Adding Memoization to our solution

- Before memoization:

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

The list used to memoize

```
/** For  0 <= k < cached.size(),  cached[k] = fib(k) */
static ArrayList<Integer> cached=  new ArrayList<Integer>();
```

After Memoization


```
/** For  0 <= k < cached.size(),  cached[k] = fib(k) */
static ArrayList<Integer> cached=  new ArrayList<Integer>();

static int fib(int n) {
   if (n < cached.size()) return cached.get(n);
   int v;
   if (n <= 1)
        v=  n;
   else v=  fib(n-2) + fib(n-1);
   if (n == cached.size())
       cached.add(v);
   return v;
}
```

This works because of definition of cached

This appends v to cached, keeping cached's definition true

Notice the development process

- We started with the idea of recursion
- Created a very simple recursive procedure
- Noticed it will be slow, because it wastefully recomputes the same thing again and again
- So made it a bit more complex but gained a lot of speed in doing so

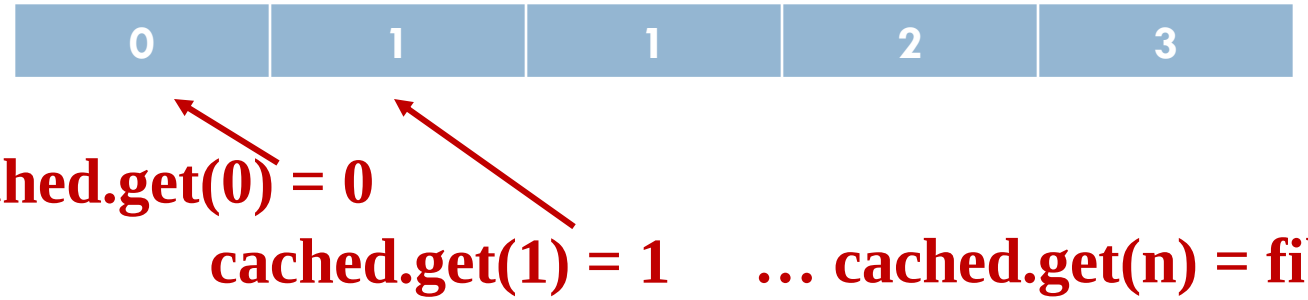
- This is a common software engineering pattern

Why did it work?

- This cached list "works" because for each value of n, either cached.get(n) is still undefined, or has fib(n)

- Takes advantage of the fact that an ArrayList adds elements at the end, and indexes from 0

cached@BA8900, size=5



Property of our code: cached.get(n) accessed <u>after</u> fib(n) computed

Positive Integer Powers

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

- Alternative description:
  - $a^0 = 1$
  - $a^{n+1} = a \cdot a^n$

```
/** = a^n, given n >= 0 */
static int power(int a, int n) {
   if (n == 0) return 1;
   else return a*power(a,n-1);
}
```

A Smarter Version

- Power computation:
  - $a^0 = 1$
  - If n is nonzero and even, $a^n = (a*a)^{n/2}$
  - If n is nonzero, $a^n = a * a^{n-1}$
    - Java note: For ints x and y, "x/y" is the integer part of the quotient
- We'll see that judicious use of the second property makes this a logarithmic algorithm, as we will see

Example: $3^8 = (3*3) * (3*3) * (3*3) * (3*3) = (3*3)^4$

Smarter Version in Java

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

```
/** = a**n, for n >= 0 */
static int power(int a, int n) {
   if (n == 0) return 1;
   if (n%2 == 0) return power(a*a, n/2);
   return a * power(a, n-1);
}
```

Build table of multiplications

n	n	mults
0		0
1	2**0	1
2	2**1	2
3		3
4	2**2	3
5		4
6		4
7		4
8	2**3	4
9		5
…		
16	2**4	5

Start with n = 0, then n = 1, etc., for each, calculate number of mults based on method body and recursion.

See from the table: For n a power of 2, n = 2**k, only k+1 = (log n) + 1 mults

For n = 2*15 = 32768, only 16 mults!

```
static int power(int a, int n) {
    if (n == 0) return 1;
    if (n%2 == 0) return power(a*a, n/2);
    return a * power(a, n-1);
}
```

Another way to look at function

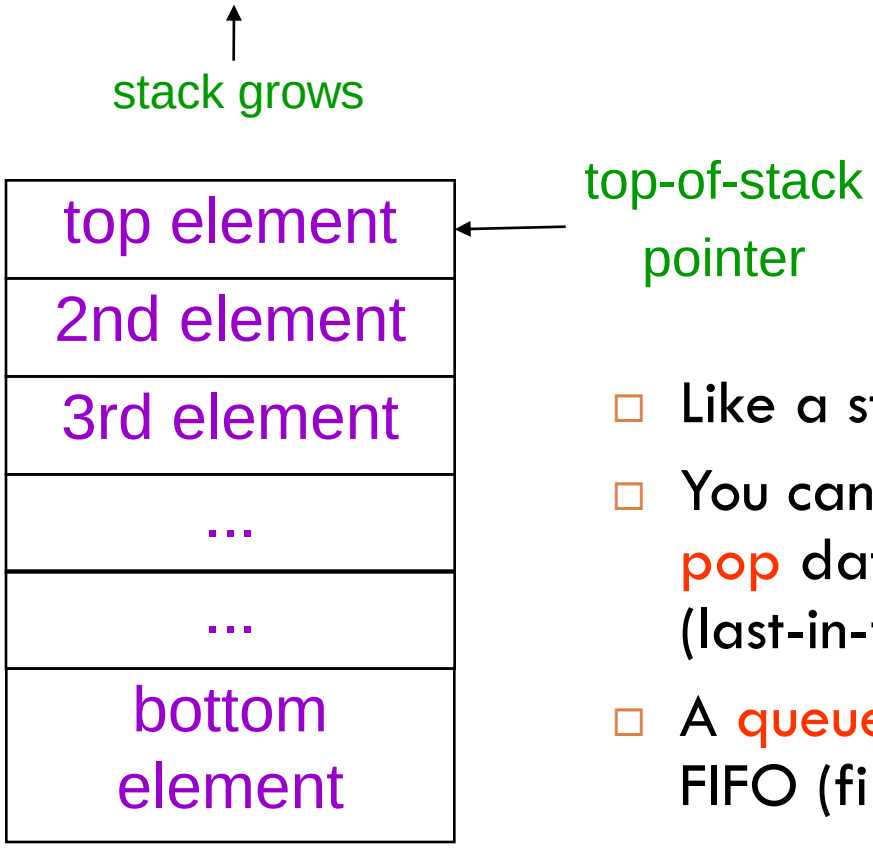
- Recursive function processes binary representation of exponent n.
- Suppose n = 10, which in binary is 1010
  - Test if n is even (n%2 == 0): is last bit 0?
  - Operation n/2: delete last bit (1010) becomes 101)
  - Operation n-1 (when n is odd): change last 1 to 0
- Each bit is processed at most twice —once to change a 1 into a 0
- Length of binary rep is log of number.

How Java "compiles" recursive code

- Key idea:
  - Java uses 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

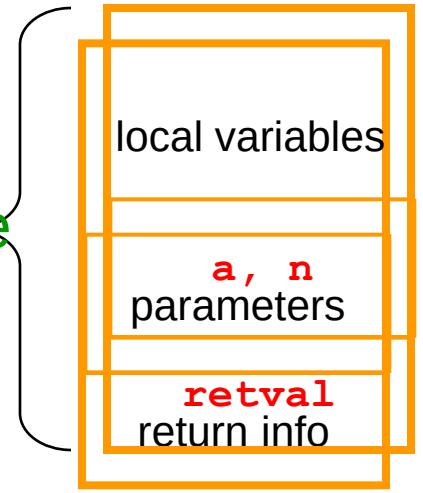
Stacks



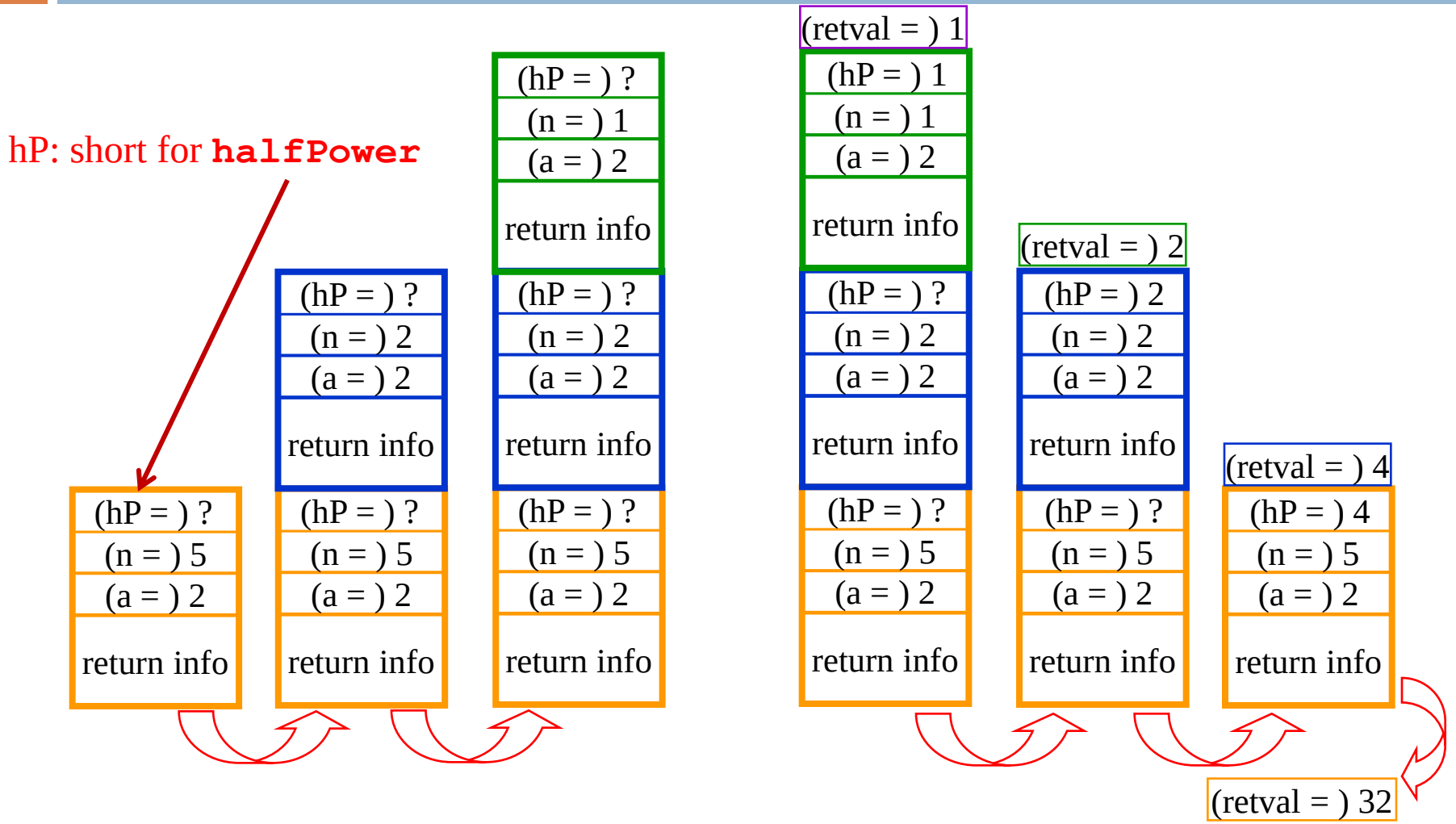
- Like a stack of dinner plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

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



Example: power(2, 5)



How Do We Keep Track?

- Many frames may exist, but computation occurs only in the top frame
  - The ones below it are waiting for results

- The hardware has nice support for this way of implementing function calls, and recursion is just a kind of function call

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

Extra slides

- For use if we have time for one more example of recursion
- This builds on the ideas in the Fibonacci example

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"**

  $\binom{5}{2}$ = number of 2-element subsets of {A,B,C,D,E}

  2-element subsets containing A: $\binom{4}{1}$
  {A,B}, {A,C}, {A,D}, {A,E}

  2-element subsets not containing A: {B,C},{B,D},{B,E},{C,D},{C,E},{D,E} $\binom{4}{2}$

- Therefore, $\binom{5}{2} = \binom{4}{1} + \binom{4}{2}$
- … in perfect form to write a recursive function!

Combinations

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

$$\binom{n}{n} = 1$$

$$\binom{n}{0} = 1$$

Can also show that $\binom{n}{r} = \frac{n!}{r!(n-r)!}$

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

Pascal's triangle =

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Binomial Coefficients

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

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

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

Combinations Have Two Base Cases

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

$$\binom{n}{n} = 1$$

$$\binom{n}{0} = 1$$

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

Recursive Program for Combinations

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

$$\binom{n}{n} = 1$$

$$\binom{n}{0} = 1$$

```
/** = no. combinations of n things taking r at a time.
    Precondition: 0 <= r <= n */
static int combs(int n, int r) {
   if (r == 0 || r == n) return 1; //base cases
   else return combs(n-1,r) + combs(n-1,r-1);
}
```

Exercise for the reader (you!)

- Modify our recursive program so that it caches results
- Same idea as for our caching version of the fibonacci series

- Question to ponder: When is it worthwhile to adding caching to a recursive function?
  - *Certainly not always…*
  - *Must think about tradeoffs: space to maintain the cached results vs speedup obtained by having them*

Something to think about

- With fib(), it was kind of a trick to arrange that:

  cached[n] = fib(n)

- Caching combinatorial values will force you to store more than just the answer:
  - Create a class called `Triple`
  - Design it to have integer fields `n, r, v`
  - Store Triple objects into `ArrayList<Triple> cached;`
  - Search `cached` for a saved value matching `n` and `r`
    - Hint: use a foreach loop