RECURSION

Lecture 6
CS2110 – Spring 2013

Recursion

2

- 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

3

- 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

4

```
/** 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);
  }
```

sum calls itself!

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

Example: Is a string a palindrome?

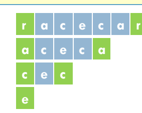
5

```
/** = "s is a palindrome" */
  public static boolean isPalindrome(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) && isPalindrome(s.substring(1, n));
  }
```

Substring from char(1) to char(n-1)

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



Count the e's in a string

6

```
/** = " 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 from char(1) to end

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

The Factorial Function (n!)

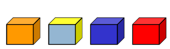
7

- Define n! = n·(n–1)·(n–2)···3·2·1
  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

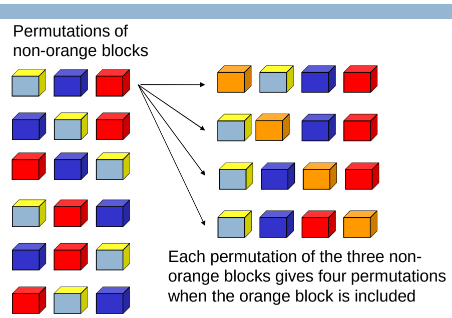
The Factorial Function (n!)

8

- 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·(n – 1)!

Permutations of 

9



- Total number = 4·3! = 4·6 = 24: 4!

Observation

10

- 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

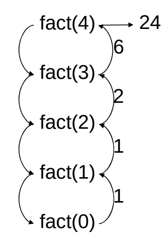
11

0! = 1

n! = n·(n–1)!, n > 0

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

Execution of fact(4)



General Approach to Writing Recursive Functions

12

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

13

- Keep in mind that each instance of your recursive function has its own local variables
- Also, remember that "higher" instances are waiting while "lower" instances run
- Not such a good idea to 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

14

- **Mathematical definition:**
  fib(0) = 0
  fib(1) = 1 ← two base cases!
  fib(n) = fib(n – 1) + fib(n – 2), n ≥ 2
- **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-2) + fib(n-1);
}
```

Fibonacci (Leonardo Pisano) 1170–1240?

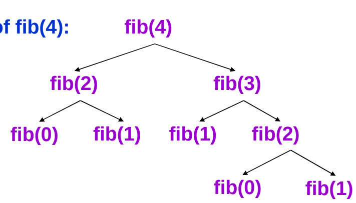
Statue in Pisa, Italy
Giovanni Paganucci
1863

Recursive Execution

15

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

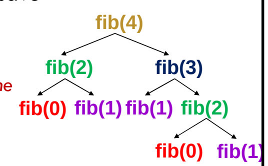
Execution of fib(4):



One thing to notice

16

- 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")

17

- 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, we save result
  - The next time, we can 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

18

- Before:

```
static i
  if (r
      r
  else
      r
  else
      r
}
```

- After

```
static ArrayList<Integer> cached =
                       new ArrayList<Integer>();

static int fib(int n) {
   if(n < cached.size())
       return cached.get(n);
   int v;
   if (n == 0)
       v = 0;
   else if (n == 1)
       v = 1;
   else
       v = fib(n-2) + fib(n-1);
   // cached[n] = fib(n).  This code makes use of the fact
   // that an ArrayList adds elements to the end of the list
   if(n == cached.size())
       cached.add(v);
   return v;
}
```

Notice the development process

19

- 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

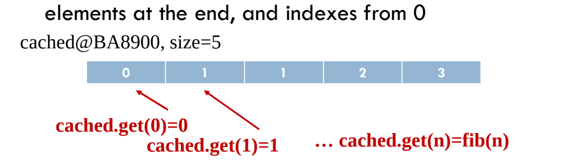
- This is a common software engineering pattern

Why did it work?

20

- 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



Property of our code: cached.get(n) accessed after fib(n) computed

Positive Integer Powers

21

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

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

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

A Smarter Version

22

- Power computation:
  - a0 = 1
  - If n is nonzero and even, an = (an/2)2
  - If n is odd, an = a·(an/2)2
    - Java note: If x and y are integers, "x/y" returns the integer part of the quotient
- Example:
  - a5 = a·(a5/2)2 = a·(a2)2 = a·((a2/2)2)2 = a·(a2)2

    Note: this requires 3 multiplications rather than 5!

A Smarter Version

23

- … Example:
  - a5 = a·(a5/2)2 = a·(a2)2 = a·((a2/2)2)2 = a·(a2)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

24

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

local variable parameters

```
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;
}
```

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

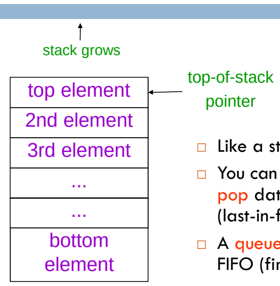
How Java "compiles" recursive code

25

- 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

26

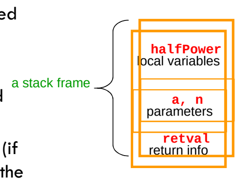


- Like a stack of dinner plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

Stack Frame

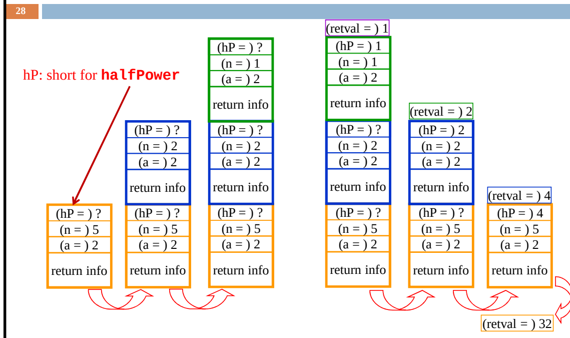
27

- 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)

28



How Do We Keep Track?

29

- Many frames may exist, but computation is only occurring 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

30

- 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

31

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

Binomial Coefficients

34

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

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

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

34

Combinations (a.k.a. Binomial Coefficients)

32

- 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 Have Two Base Cases

35

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

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

Combinations

33

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

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

Pascal's triangle =

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

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Recursive Program for Combinations

36

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

```
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);
}
```

Exercise for the reader (you!)

37

- 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

38

- 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