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 two forms in computer science

- Recursion as a *mathematical* tool for defining a function in terms of itself 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!)

Mathematical induction is used to prove that a recursive function works correctly. This requires a good, precise function specification.

See this in a later lecture.

Recursion as a math technique

Broadly, recursion is a powerful technique for defining 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.
  * Precondition:  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?

```
/** = "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

r a c e c a r
a c e c a
c e c
e

Example: 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

Example: 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

A Recursive Program

0! = 1

n! = n·(n−1)!, n > 0

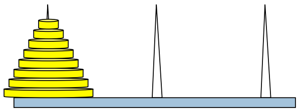
```
/** = n!. Precondition: n >= 0 */
static int fact(int n) {
   if (n = = 0)
       return 1;
   // { n > 0 }
   return n*fact(n-1);
}
```

General Approach to Writing Recursive Functions

1. Find *base case(s)* – small values of n for which you can just write down the solution (e.g. 0! = 1)
2. 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)
3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

Example: Tower of Hanoi

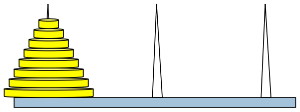
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



Stacked upon the leftmost needle were 64 golden disks, each a different size, stacked in concentric order:

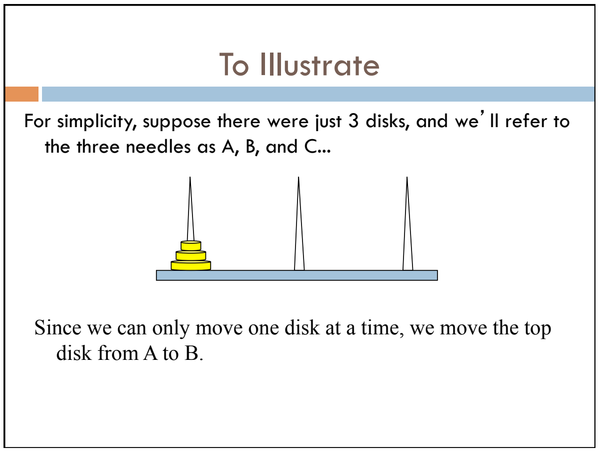
A Legend

The priests were to transfer the disks from the first needle to the second needle, using the third as necessary.



But they could *only move one disk at a time*, and could *never put a larger disk on top of a smaller one*.

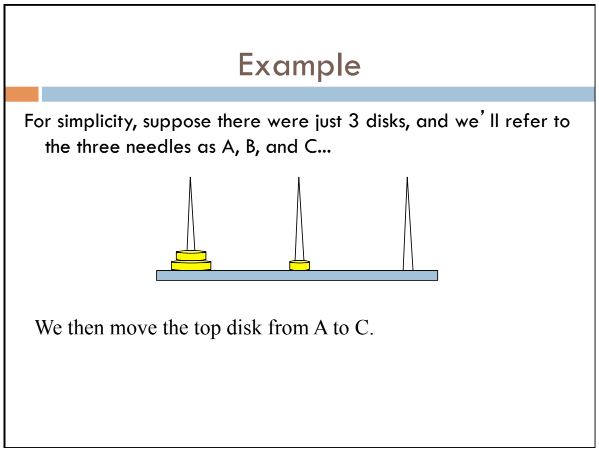
When they completed this task, the world would end!



To Illustrate

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

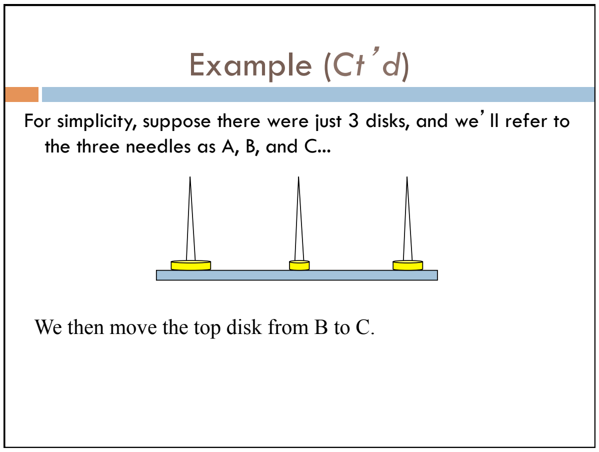
Since we can only move one disk at a time, we move the top disk from A to B.



Example

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

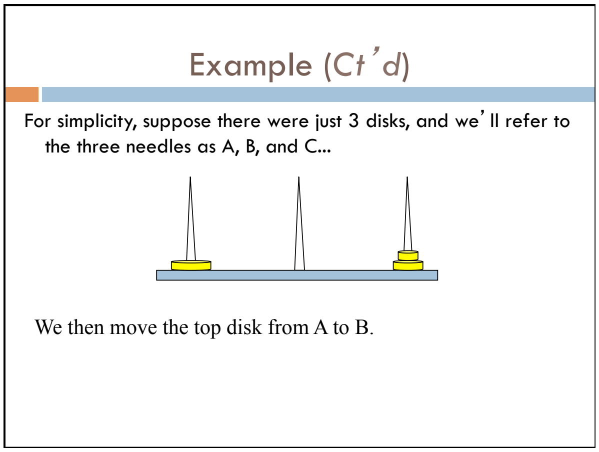
We then move the top disk from A to C.



Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

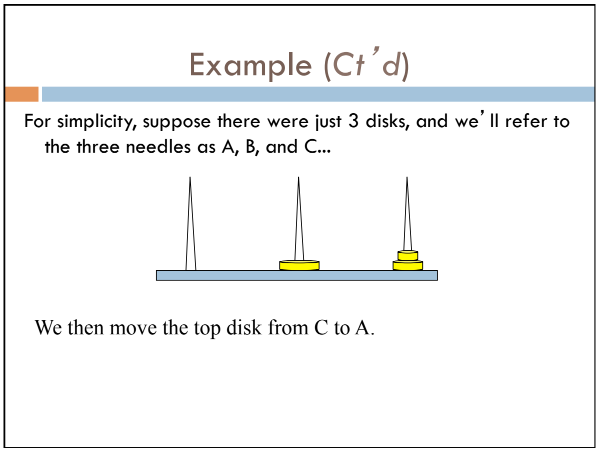
We then move the top disk from B to C.



Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

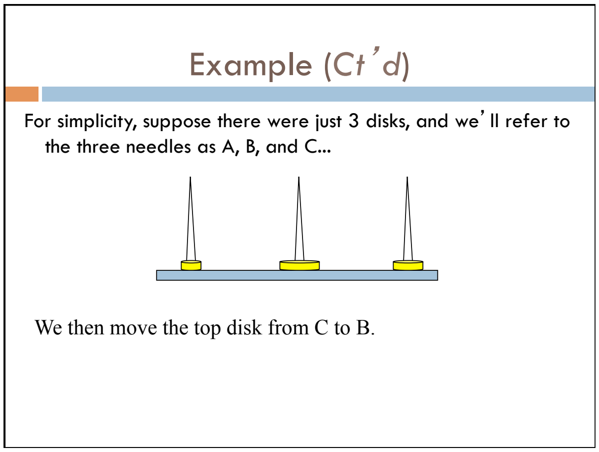
We then move the top disk from A to B.



Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

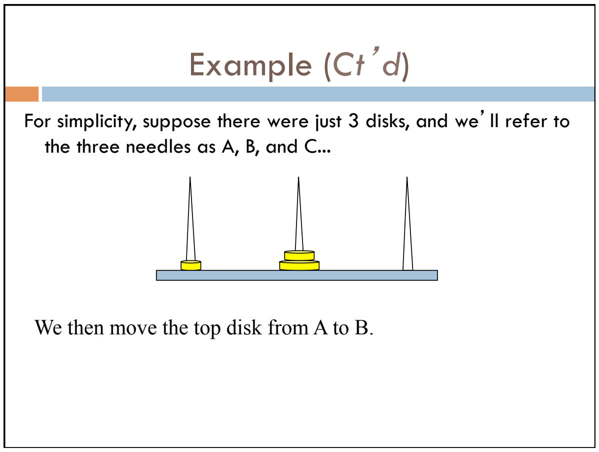
We then move the top disk from C to A.



Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

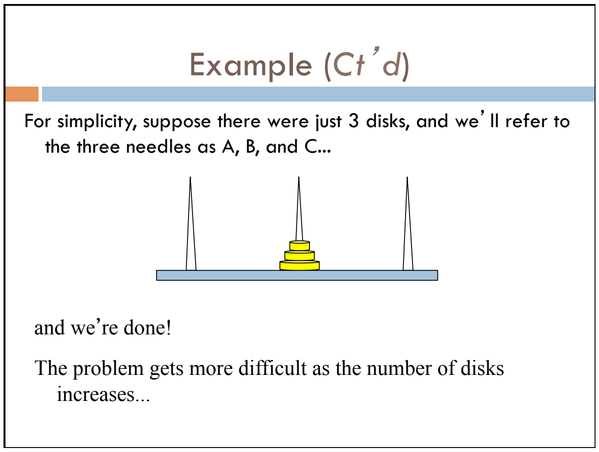
We then move the top disk from C to B.



Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

We then move the top disk from A to B.

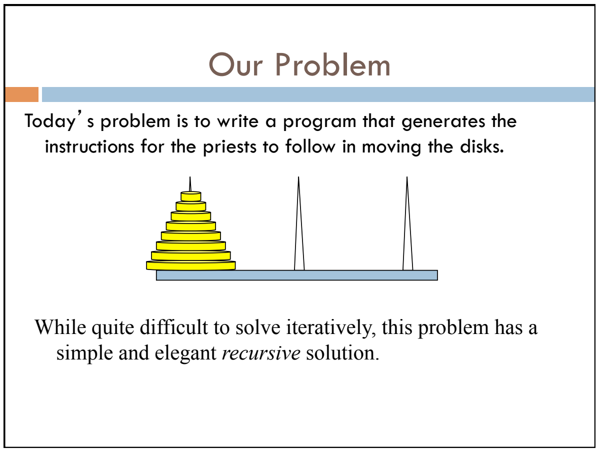


Example (Ct’d)

For simplicity, suppose there were just 3 disks, and we’ll refer to the three needles as A, B, and C...

and we’re done!

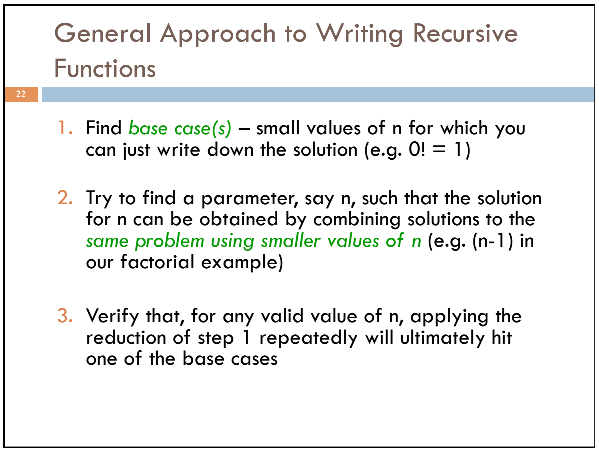
The problem gets more difficult as the number of disks increases...



Our Problem

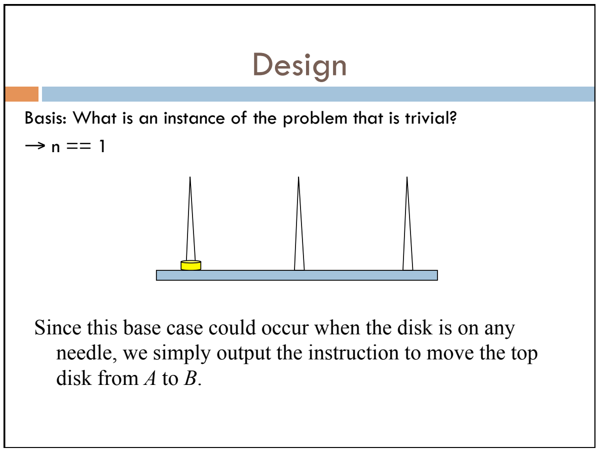
Today’s problem is to write a program that generates the instructions for the priests to follow in moving the disks.

While quite difficult to solve iteratively, this problem has a simple and elegant *recursive* solution.



General Approach to Writing Recursive Functions

1. Find *base case(s)* – small values of n for which you can just write down the solution (e.g. 0! = 1)
2. 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)
3. Verify that, for any valid value of n, applying the reduction of step 1 repeatedly will ultimately hit one of the base cases

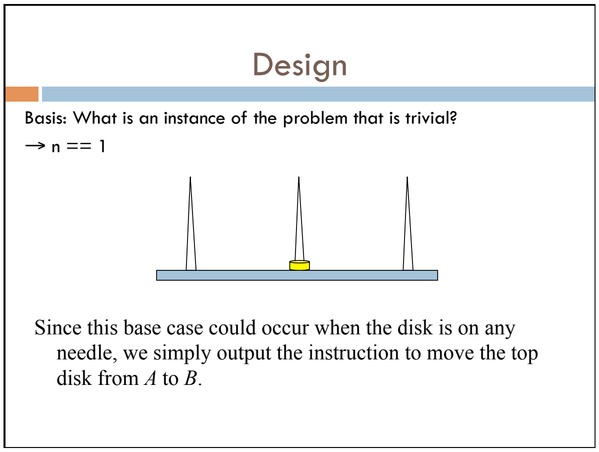


Design

Basis: What is an instance of the problem that is trivial?

→ n == 1

Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *A* to *B*.



Design

Basis: What is an instance of the problem that is trivial?

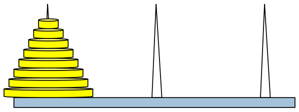
→ n == 1

Since this base case could occur when the disk is on any needle, we simply output the instruction to move the top disk from *A* to *B*.

Design (Ct'd)

Induction Step: n > 1

→ How can recursion help us out?

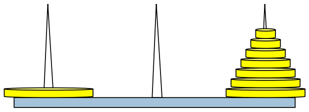


a. Recursively move n-1 disks from *A* to *C*.

Design (Ct'd)

Induction Step: n > 1

→ How can recursion help us out?

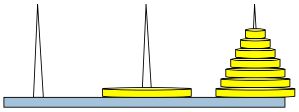


b. Move the one remaining disk from *A* to *B*.

Design (Ct'd)

Induction Step: n > 1

→ How can recursion help us out?

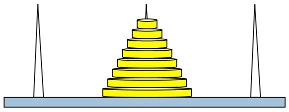


c. *Recursively* move n-1 disks from *C* to *B*...

Design (Ct'd)

Induction Step: n > 1

→ How can recursion help us out?

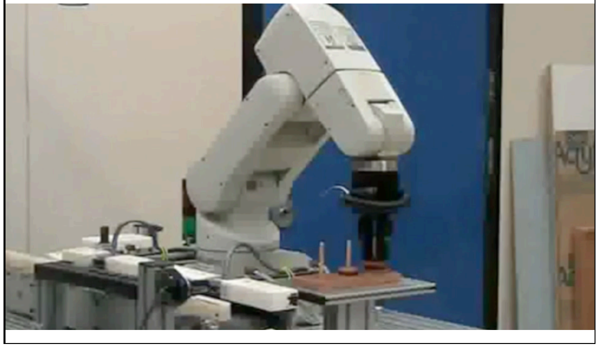


d. We're done!

Tower of Hanoi: Code

```
void Hanoi(int n, string a, string b, string c)
  {
     if (n == 1)  /* base case */
        Move(a,b);
     else { /* recursion */
        Hanoi(n-1,a,c,b);
        Move(a,b);
        Hanoi(n-1,c,b,a);
     }
  }
```

Tower of Hanoi on Robot!



The Fibonacci Function

Mathematical definition:

$fib(0) = 0$
$fib(1) = 1$ two base cases!
$fib(n) = fib(n - 1) + fib(n - 2), \; n \ge 2$

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

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

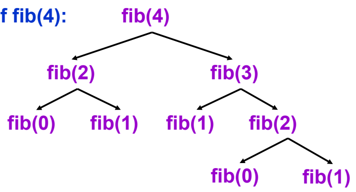
Fibonacci (Leonardo Pisano) 1170-1240?

Statue in Pisa, Italy
Giovanni Paganucci
1863

Recursive Execution

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

Execution of fib(4):



Non-Negative 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. Pre: n >= 0 */
static int power(int a, int n) {
  if (n == 0) return 1;
  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

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. Precondition: 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);
}
```

How Java "compiles" recursive code

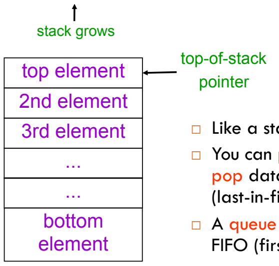
Key idea:

- 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



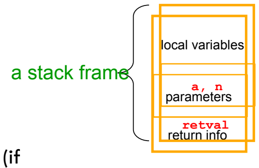
- Like a stack of dinner plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

Stack Frame

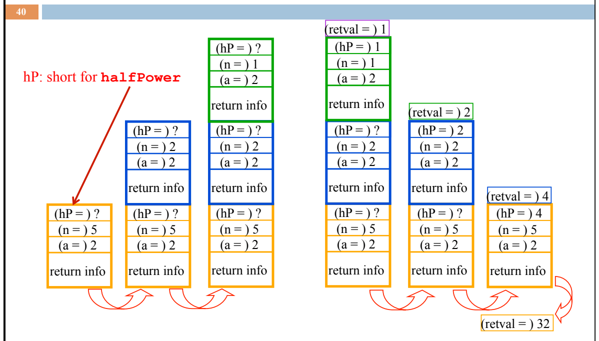
A new stack frame is pushed with each recursive call

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)


hP: short for **halfPower**

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

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!

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.

- First time the function is called, save result
- Next times, look up the result
  - 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

---

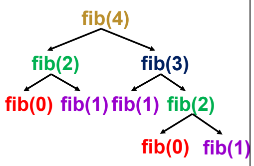
One thing to notice: Fibonacci

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*



Adding Memoization to our solution

Before memoization:

```
static int fib(int n) {
   if (n <= 1) return n;
   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
- We made it a bit more complex but gained a lot of speed in doing so

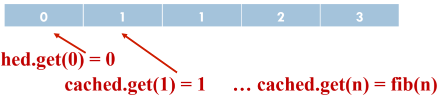
- This is a common software engineering pattern

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 after fib(n) computed