About prelim, about specs, about analysis

Submit requests for regrades on the CMS. Please do not email Gries. Gries does all the regrading.

Prelim tonight: Uris Auditorium, 7:30 -- 9:00/

We hope to have grades posted on the CMS by 1AM tonight, hopefully earlier

About specs

Almost all students were able to generate the 92 solutions.
Most got points deducted somewhere, often for unsatisfactory specifications. Example complaint:

My partner and I have several issues with the way our queens lab was graded and find them unfair. 5 points were deducted because our specification for the method was considered incorrect. Neither of us have taken CS100, and never in class was a list of commenting procedures explictly talked about. How are we supposed to guess the commenting standards we are expected to adhere to?

We address this issue.

From slides for lecture 03-04

Understanding a recursive method

MEMORIZE THE FOLLOWING

Step 0: HAVE A PRECISE SPECIFICATION.

Step 1: Check correctness of the base case.

Step 2: Check that recursive-call arguments are in some way smaller than the parameters, so that recursive calls make progress toward termination (the base case).

Step 3: Check correctness of the recursive case. When analyzing recursive calls, use the specification of the method to understand them.

From lecture 03-04

Step 1: HAVE A PRECISE SPECIFICATION

```
// = !n  (for n>=0)
public static int fact(int n) {
```

We discussed two things:

(1) The specification has to give constraints that the parameters must satisfy —a precondition **(n ≥ 0)**

(2) For a function, say what is produces, in terms of the parameters. **(= !n)**
For a procedure, say what it does, in terms of the parameters. (**e.g. print the squares of integers in 0..n**)

Over and over, I have said that the spec must mention the parameters, saying how they are used or what they are for.

From lecture 03-04

Understanding a recursive function

Step 4: Check correctness of recursive case; use the method spec to understand recursive calls.

In the recursive case, the value returned is

n * fact(n -1).

Using the specification for method fact, we see this is equivalent to

n * !(n -1).

That's definition of !n, so the recursive case is correct.

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

From lecture 03-04

Understanding a recursive function

What we said, and did, was: to understand what a call does: copy the specification, but replace the pars by the args.

fact(n-1) **What is its value?**

Spec is: = !n (for n>=0).

We know from the context of the call that n-1>= 0. So the call is equivalent to

!(n-1) **This is its value**

```
// = !n  (for n>=0)
public static int fact(int n) {  …}
```

From lecture 03-04
Understanding a recursive function

What we said, and did, was: to understand what a call does: copy the specification, but replace the pars by the args.

This statement should make it clear that if the spec does not mention the parameters, and in the correct way, then the spec is unsatisfactory.

Every single time I have written a method in this class, I have FIRST written the specification, in terms of the parameters, giving the precondition and what the method does.

Creating recursive methods from lecture 03-04

Use the same steps that were involved in understanding a recursive method.

- **Be sure to**
  SPECIFY THE METHOD PRECISELY.
- **Handle the base case first.**
- **In dealing with the non-base cases, think about how you can express the task in terms of a similar but smaller task.**

If the spec does not express the task clearly and precisely, in terms of the parameters, how can you express the task in the non-bases cases in terms of a similar but smaller task?

Creating a recursive method from lecture 03-04

Task: Write a method that removes blanks from a String.

0. Specification: **precise specification!**

```
// = s but with its blanks removed
public static String deblank(String s)
```

1. Base case: the smallest String is “”.

```
if (s.length == 0)
    return s;
```

2. Other cases: String s has at least 1 character.
If it’s blank, return s[1..] but with its blanks removed. If it’s not blank, return

s[0] + (s[1..] but with its blanks removed)

s[0] + deblank(s[1..])

Vague specifications

```
/** binary search */
public static int bsearch(int[] b, int v)
```

Principle 1: Can you write the method body based on that spec? If you can’t, then the spec is no good.

Principle 2: Do you know how to write a call on the method based on the spec. If you can’t, then the spec is no good.

Can I call the method if v is not in b? If so, what value does it return? What value does it return if v is in b? A 1 saying it is in and 0 saying it is not? The index of the value?

Take a look at all the specs given in the API package.
They say WHAT the method does, not HOW.

Each such spec is a contract between the programmer and you, the user.
The programmer agrees to write the method to satisfy the specification.
You, the user, agree to call it only when the preconditions of the method are met.

A spec by one student

```
/*the recursive routine: b stores the board, and n stores the
current column. It works by looping through every space in a
given column, trying a queen in each. If that square is valid,
it calls itself again, only with the column shifted right one.
When it gets to the end, the board has passed all tests, and is
therefore a valid solution. If at any time the queen can not go
in the test space, the function returns.*/
public static void generate(… b,  … n) {…}
```

There was no mention anywhere of what b contained.

Can ANYONE in the class write function generate based on this description? Or write a call on it?

Method solutions()

/** = an array of all possible solutions to the eight queens problem. Each solution is given by a String of 8 integers that give the positions of the queens in the solution. … */
public static String[] solutions() { ... }

So, if we execute:

```
    q1= Queens.solutions();
    q2= Queens.solutions();
```

q1 and q2 should contain equal arrays.

generate(…) appends solutions to a variable sol. Therefore, solutions has to make sure it is empty before calling generate(…).

Method solutions()

If we execute:

```
    q1= Queens.solutions();
    q2= Queens.solutions();
```

q1 and q2 should contain equal arrays.

If we execute

```
    b1= Math.abs(-25);
    b2= Math.abs(-25);
```

b1 and b2 should contain equal integers.