MIDTERM REVIEW

CS 200

This review material is not intended as a complete study guide, as I don't know exactly which questions will be on the exam a the time of writing.

Grammars

Formal and rigorous technique for specifying a language. Concise way to describe a language to another person as long as they also understand grammars. There are even software packages which can generate a parser program given a properly specified grammar.

Example:

Consider the following grammar:

< S > = @ | < W > | @ < S >
< W > = xxy | xx< W >y

Write all the strings in the language which are <= 7 Characters:

@, @@, @@@, @@@@, @@@@@, @@@@@@, @@@@@@@,
xxy, @xxy, @@xxy, @@@xxy, @@@@xxy,
xxxxyy, @xxxxyy

- 'in the language' means strings conforming to the rules of < S >. The first rule listed is almost always the one that matters (unless the problems specifically states otherwise).
- < w > means any string in < w > when it is used in < S >
- order matters: @ < S > is not < S > @

Recursion

Any method which calls itself is recursive. Any time you are designing a recursive method, there are two important questions to ask:

- What is the base case (when should it stop) ?
- How will it make progress ?

Example

```
public static int func(int x){
        if(x == 1 || x == 0){
                if(x == 0){
```

```
                return 0;
            } else {
                return 1;
            }
    } else {
            return func(x-1) + func(x-2);
        }
}
```

- what does this compute?
- Note that questions which ask for a recursive function, require a recursive answer. Correct, but non-recursive, answers will loose many points.

Exercise:

Implement a recursive parser for the above grammar (in the grammars section). Test it with the provided strings.

Induction

Prove that some long series of addition / multiplication works out to what we think it does. Lets look again at a previous example of induction:
The following is the formula, called p(n), for the summation of natural numbers:

$$0 + 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Basis step: Show that it holds for n = 0.

$$0 = \frac{0 \cdot (0+1)}{2}$$
$$0 = \frac{0}{2} = 0$$

- Start with only 0 on the left side, and n = 0 on the right side of the previous equation. Do the arithmetic, and if the left side equals the right side, you have proved the basis. Or shown that p(0) holds, depending on your terminology.

Inductive Hypothesis: If p(k) holds, then p(k+1) also holds. The following inductive step should prove this hypothesis.

Inductive Step: We now need to show that if p(k) holds, then p(k+1) also holds. See later note about n and k. *Assume that p(k) holds, for some unspecified value of k.* What we want to show is that p(k+1) holds:

$$(0 + 1 + 2 + \cdots + k) + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

- Notice what was done here. The left side started out the same as the left side of the very first equation, and then had a '(k+1)' added to it. The right side started out the same as the right side of the very first equation. Then, every k was robotically replaced with a '(k+1)'.

We already have a formula for p(k), from the original equation. We can substitute that for the '(0+1+2+...+k) term above:

$$\frac{k(k+1)}{2} + (k+1) = \frac{(k+1)((k+1)+1)}{2}$$

What we must now do is make the left side match the right side.

$$\frac{k(k+1)+2(k+1)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

$$\frac{k^2+3k+2}{2} = \frac{(k+1)((k+1)+1)}{2}$$

$$\frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

$$\frac{(k+1)((k+1)+1)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

Left now equals right, we are done. We have shown the p(k+1) holds by showing that the left and right sides of the original equation are still equivalent in the (k+1) case just like they were for k.

About k and n, the notation changes depending on which part of the problem is being done. n is usually used for describing the original equation. k is used during the algebra for the inductive step, but it is the same variable renamed. I think this is to create visual distinction between the original equation and the later algebra work. For our purposes n == k.

Notes:

- Don't forget the basis step!
- Don't forget the basis step!
- Inductive hypothesis is (almost always) the same on every problem, write it down anyway.
- The algebra done in the inductive step above is the mechanics of proving the hypothesis, don't leave it out, and show your work.

Exercise:

Rosen 6th ed. pp 280 # 13:
Prove that

$$1^2 - 2^2 + 3^2 - \ldots + (-1)^{n-1}n^2 = \frac{(-1)^{n-1}n(n+1)}{2}$$

Whenever n is a positive integer

Interfaces

Is a:

- Contract between implementing classes and outside world
- Implementing classes MUST provide functionality specified in the interface
- Buttons on electronic devices are an example of a physically implemented interface

A Java Interface:

- ONLY CONTAINS: constants, method signatures, and nested types.
- CANNOT BE INSTANTIATED: it must be implemented by a class or extended by another interface
- Note that interfaces in java cannot specify constructor methods.
  - constructors are specific to a particular class, an interface must be applicable to multiple classes.
  - this limitation can be worked around by having a non-constructor method return class instances. Since it's not a constructor, it can be in the interface. This is the purpose of create() on PA1's skeleton files.
  - see the *factory method* and *singleton* design patterns, which both use this technique.

Stacks and Queues

Both describe a structure for storing data. These are Abstract Data Types (ADT), since they specify how to store any data, not what type it is.

- stacks are Last In, First Out (LIFO) structures. This means that if you put a list of things into a stack, and then get them all back out, they will be in reversed order.
- queues are First In, First Out (FIFO) structures. If you put a list of things into a queue and then get them all back out again, they will be in the same order.
- Stack common methods:
  - push(): add something to the top of the stack
  - pop(): remove top item and return it
  - peek(): reveal top item without removing it
- Queue common methods:
  - enqueue(): add something to back of queue
  - dequeue(): remove and return front-most item
  - peek(): reveal front-most item without removing it

Exercise:

Design an interface which can be implemented by BOTH stacks and queues, give this interface control over how implementing classes are constructed. Then design a stack object and a queue object which implement that interface.

Big O Notation

This is pretty recent, at least in recitation, so I won't redo last week's recit.

Estimation:

The "look at this code, how often does _____ run?' type problems.

- Find the work. Look for a loop / recursion / something which depends on the input size.
- try to find a relation in n which describes how many times the work is done for a given n
- express that in Big O.
- Many of these problems are asking you to count number of runs for a very specific sub-section of code. Be sure to read the problem carefully!

Proofs:

The 'prove that f(x) is O(g(x))' type problems.

- Remember that these problems are about Big O in the mathematical sense, not in the programming sense. As such, you cannot use the rules of Big O that we use when describing software. You can't say that O(something) is the same as O(something else), which would make most proofs trivial anyway.
- Remember the technique from last week:
  - Write down f(x) <= f(x)
  - We want the right side to eventually look like g(x), using:
    * Algebra
    * Term replacement, where we replace terms in the right side with terms which are closer to our goal of g(x). For example, if g(x) is cubic, and f(x) is only quadratic (x^2), we would want to upgrade the power of terms in f(x). When replacing terms, we must make an observation detailing the conditions under which that replacement is valid. The observations tell us the conditions under which the replacement can be made without breaking the inequality.
    * If you are having trouble making observations for term replacement, try using algebra to reduce either side to a simpler form.
    * Note also that with term replacement, you do not have to replace terms with g(x). You can replace them with anything that gets closer to g(x). If replacing terms with g(x) is leading to difficult observations, try replacing with something simpler and then using algebra to get to g(x).

Rehash of last week's example:
Show that $6x^2 + 2x + 3$ is $O(x^3)$
$6x^2 + 2x + 3 \leq 6x^2 + 2x + 3$ **f(x) <= f(x)**
$6x^2 \leq x^3$, for $x > 6$. **Observations**
$2x \leq x^3$, for $x > 2$.
$3 \leq x^3$, for $x > 2$.
$6x^2 + 2x + 3 \leq 6x^3 + 2x^3 + 3x^3$ **Term replacement**
$6x^2 + 2x + 3 \leq 3x^3$is $O(x^3)$ with witness: c=3 and k=6.
Finally, remember the rules of Big O reduction, and that these only apply when Big O is being used in a programming sense. You can use them when referring to code, but not when doing proofs:

- You can ignore low order terms: $O(n^3 + 4n^2 + 3n)$ is also $O(n^3)$
- You can ignore coefficients $O(15n^3)$ is also $O(n^3)$
- You can combine growth rates $O(n^2) + O(n)$ is also $O(n^2 + n)$ is also $O(n^2)$

Exercises, there are two:

1) How many times is the variable ncount incremented in the following code:

```
public static void test(int x){
        int ncount = 0;
        for(int i = 0; i < x; i++){
                for(int j = 0; j < x; j++){
                        for(int k = 0; k < x; k++){
                                //irrelevant code here
                                // ...
                                if(i == j && i == k){
                                        ncount++;
                                }
                        }
                }
        }
        System.out.println("test "+ncount);
}
```

2) Rosen, 6th ed, pp 191 # 3:
Show that $x^4 + 9x^3 + 4x + 7$is $0(x^4)$

Grading this week

Attendance only, come sign the sheet. The mentioned exercises are good practice, but not required because they might take too long for one recit.