1. **being able to program using OOP in Java including using generics - chapter 1 and 2**
   1. Generics - Type parameter
      1. Convention: short (single character) uppercase
      2. Can be used whenever a Type is needed
      3. Will be replaced with actual Type
      4. public class Stack<E>{...}
      5. public void push(E element){...}
      6. public E pop(){...E result = xxx.get(yyy};...}
      7. private ArrayList<E> xxx = new ArrayList<E>();
      8. List<String> parsed = new ArrayList<String>();
   2. Own generic class: Stack<Student> students = new Stack<Student>();
2. **simple Big Oh analysis - chapter 4.3 asymptotic analysis**
   1. Time complexity
      1. For each operation how many instructions must be executed
         1. Machine and compiler independent
   2. Space complexity
   3. Big O defines the upper bound of the running time of an operation or algorithm without regard to constants or machine/compiler factors
      1. O(1): constant
      2. O(log n) logarithmic
      3. O(n) linear
      4. O(n log n) linearithmic
      5. O(n2) quadratic
      6. O(n3) cubic
      7. O(2n) exponential
      8. Big O for linear search is linear O(n)
      9. Big O for binary search logarithmic O(log2 n) = O(log n)
3. **Recursion - chapter 5**
   1. A recursive method calls itself
   2. Program stack
      1. Upon invoke, the method is placed on top of the stack
         1. Save the position in the calling method
         2. Push the called method’s activation record or stack frame (usually growing downward) on the stack
         3. Begin execution of the called method
      2. Upon return/exit, the method is popped off the stack
         1. Pop the current method off the stack
         2. Continue execution of the method which called it
   3. **Base case** and **making progress** toward the base case
      1. Not working towards the base case causes infinite recursion and crash
      2. Java will throw StackOverflowError
   4. Classic example: factorials
      1. if (num<=1) return 1; else return (num\*Factorial(num-1));
   5. Pros and cons
      1. All recursive solutions can be implemented without recursion
      2. Expensive (multiple activation frames)
      3. But the expense is outweighed by a simpler algorithm
   6. Tail recursion: last line of a method makes a recursive call
      1. Multiple active stack frames unnecessary
      2. To get rid of this type of recursion:
         1. Enclose the body of the method in a while loop
         2. Replace the recursive call with an assignment statement for each method argument
   7. Circular recursion occurs when we stop making progress towards the base case
      1. E.g. a(x) calls a(x+1) and a(x+1) calls a(x)
4. **linked lists (single and double) - chapter 3.1, 3.2 and 3.4**
   1. List: a collection of items accessible one after another beginning at the head and ending at the tail.
      1. Head - first item
      2. Tail - last item
   2. Need to estimate size for the array from the beginning
   3. Expensive (in Java) or impossible to expand the array
   4. Insertions and deletions involve moving half on average or worst case all
   5. Linked list create a node for each element that contain a pointer to the next node
   6. Insertion and deletion only involve moving pointers
5. **Stacks**
6. **Queues**
7. **trees (short answer only)**

Practice problems:

5: 1, 7, 12

3: 6, 9, 25, 29

6: 1-5, 7, 17

11: 1, 2, 3, 4, 6