Dartmouth College
Computer Science 10, Winter 2012
Solution to the Final Exam

Professor Drysdale

Print your name: _____

- If you need more space to answer a question than we give you, you may use the backs of pages or you may use additional sheets of paper and attach them to the exam. Make sure that we know where to look for your answer!
- Read each question carefully and make sure that you answer everything asked for. Write legibly so that we can read your solutions. Please do not write anything in red.
- We suggest that you include comments for solutions that require you to write Java code. They will help your grader understand what you intend, which can help you get partial credit.
- Hand in only what you want graded. We do not want your scrap paper unless it contains solutions you want us to grade.
- You may use an 8.5 x 11 inch sheet of paper with notes. These notes should be handwritten or typed in with a 10 point font or larger. You may use both sides of the paper.
- We will supply Javadoc-style documentation for any class or interface that you need to answer a question.

Question	Value	Score
1	30	
2	15	
3	20	
4	15	
5	20	
Total	100	

Question 1 30 points

Short answer

(a) (4 points)
Suppose that you are implementing a GUI for a pizza-order web page. What GUI component would you use to allow the user to pick the size of the pizza? What GUI component would you use to allow the user to select the set of toppings on the pizza? Why?

Solution: Radio buttons or combo box for size (only one size for a given pizza). Check boxes for toppings (any subset of toppings is valid).

(b) (4 points)
What is the horizon effect in game tree search? How can it lead to seemingly irrational play?

Solution: In game tree search the program looks ahead a fixed number of moves and then does static evaluation. If something very good or very bad happens further on than this "horizon" it is invisible to the program. Irrational play can result from something bad happening just at the horizon. In that case, a move that does not have any effect on the bad event but that requires a response can push the bad event over the horizon. Thus it looks as if playing a normal move results in something bad, and an inferior move that pushes the bad event over the horizon looks better. Thus the program will choose the inferior move and will be worse off after the bad event eventually happens than if had it played a normal move.

(c) (4 points)
What is an admissible heuristic in A* search, and how can using an non-admissible heuristic lead to finding a suboptimal solution?

Solution: An admissible heuristic is one that never overestimates the remaining distance to a solution. If the heuristic can over-estimate the distance then a sub-optimal solution node can have a priority queue key that is smaller than the key of a node on an optimal path, because the node on the optimal path adds in an overestimate of the remaining work. Thus the sub-optimal solution can come out of the priority queue first, and A* will declare it to be the optimal solution.

(d) (4 points)
A student failed to override `hashCode` in PS-6, and ended up evaluating so many nodes that the program ran out of memory on the 8-puzzle given as a test case. Explain why it is important to override `hashCode` when you override `equals` and how failure to do so could lead to the behavior described.

Solution: If you don't override `hashCode` two puzzle states that are equal according to `equals` will have different hashCodes and will almost certainly end up in different buckets in the hash table. When a move is made and a new puzzle object is created the first question is whether this puzzle position is in the closed set (in which case it should be ignored) or the open set (in which case the earlier puzzle either remains or gets effectively replaced by the new one). But if the new puzzle hashes into a different bucket than the earlier one that it is equal to the earlier puzzle will not be found. The program will think the new position has never been seen before. Therefore huge numbers of multiple copies of a given puzzle position will end up in the queue and/or in the closed set. (Think of moving a single piece back and forth many times, creating many copies of the two puzzle states.)

(e) (4 points)
How does Norvig's spell corrector, which we saw in class, avoid computing the edit distance between the word to be corrected and every word in the dictionary?

Solution: It inverts the process. It computes a list everything of edit distance 1 from the word to be corrected. If any are in the dictionary it takes the one that appears most frequently in big.txt (or in our version bigger.txt). Otherwise it computes everything of edit distance 2 by computing everything of edit distance 1 from each word in the edit distance 1 list. It looks them up in the dictionary, and picks the one that appears most frequently in big.txt (or bigger.txt). If none of the words of edit distance 1 or 2 appear in the dictionary it gives up.

(f) (6 points)
What is the Decorator design pattern? How is it implemented? Explain how this pattern allowed us to implement a graph with directed edges.

Solution: The Decorator design pattern allows the addition of additional information called decorations to existing objects. A decoration consists of a key identifying the decoration and a value associated with the key. It is implemented by including a small map as part of the object, so that any desired decoration can be created by choosing an appropriate key and set of values. We implemented directed graphs by adding an EDGE_TYPE decoration to edge objects along with a DIRECTED value.

(g) (4 points)
Consider the following proposed solution to the Dining Philosophers problem. Each philosopher has an `eat` method given in pseudocode below. Assume that the forks are implemented with methods tryToPickUpFork and putDownFork. The tryToPickUpFork method does not wait for the fork to be available but returns immediately. It returns true if it succeeds in picking up the fork and false if it fails because the fork is in use.

```
Method eat:
  while(true) {
    Think

    while(this philosopher does not hold two forks) {
      while(! tryToPickUpFork(left fork))
        ;
      if (! tryToPickUpFork(right fork)) {
        putDownFork(left fork)
        Think
      }
    }

    Eat
    putDownFork(left fork)
    putDownFork(right fork)
  }
```

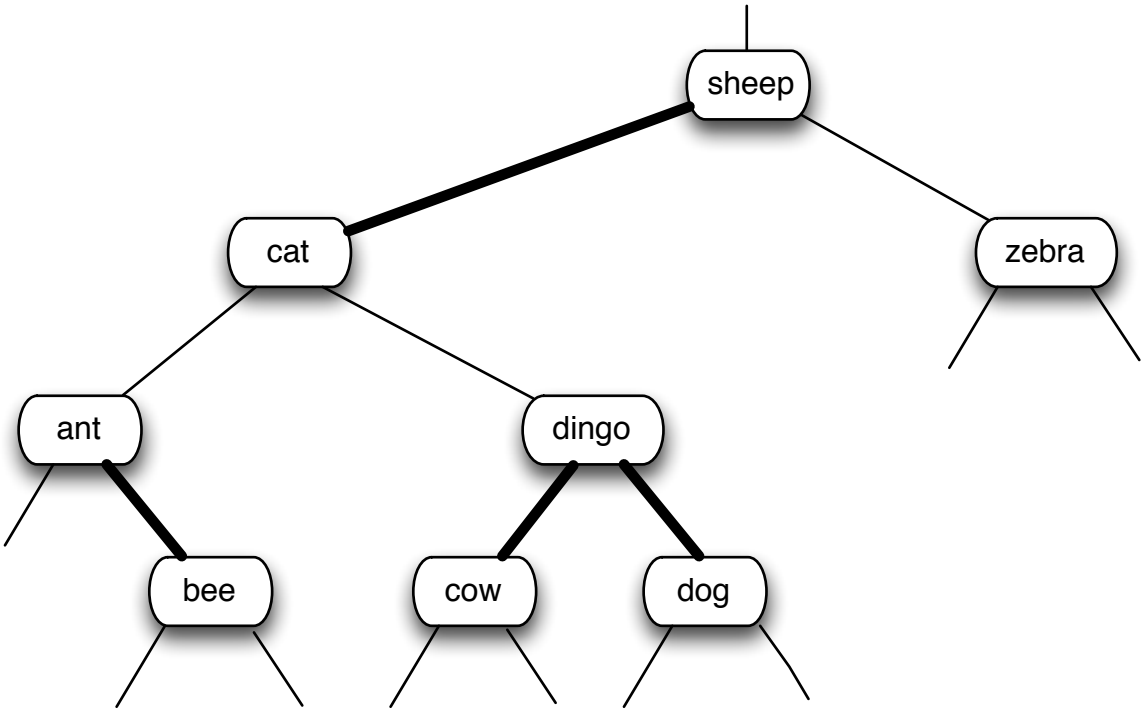
That is, the philosopher waits for the left fork to become available. He or she then tries to pick up the right fork. If this succeeds the philosopher eats. If not the philosopher puts down the left fork, thinks a while, and tries again.

Can this deadlock? If so, explain how that can happen. If not, explain why not. Can it lead to starvation? If so, explain how that can happen. If not, explain why not.

Solution: This cannot deadlock. It violates the second condition for deadlock - holding some resources while waiting for others to be available. A philosopher only waits while holding no forks. It can lead to starvation, though. Every philosopher could pick up the left fork, fail to pick up the right fork, and put down the left fork over and over in perfect synchronization forever, so it is possible that no philosopher ever eats.

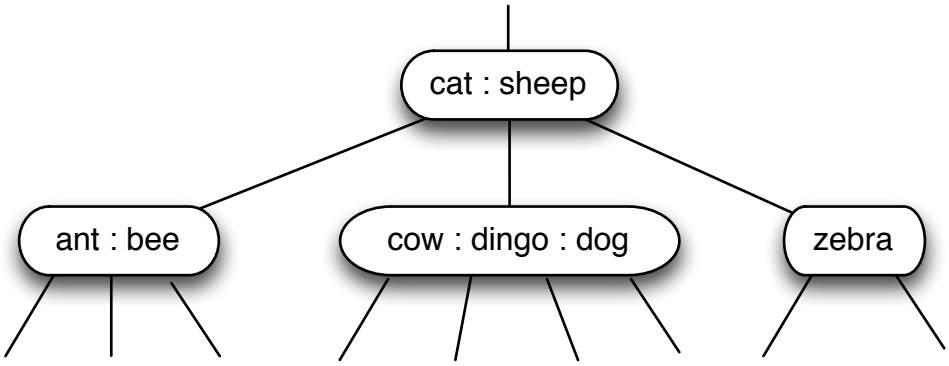
Question 2 15 points

Consider the following red-black binary search tree. Dark edges indicate that the node at the bottom is red.



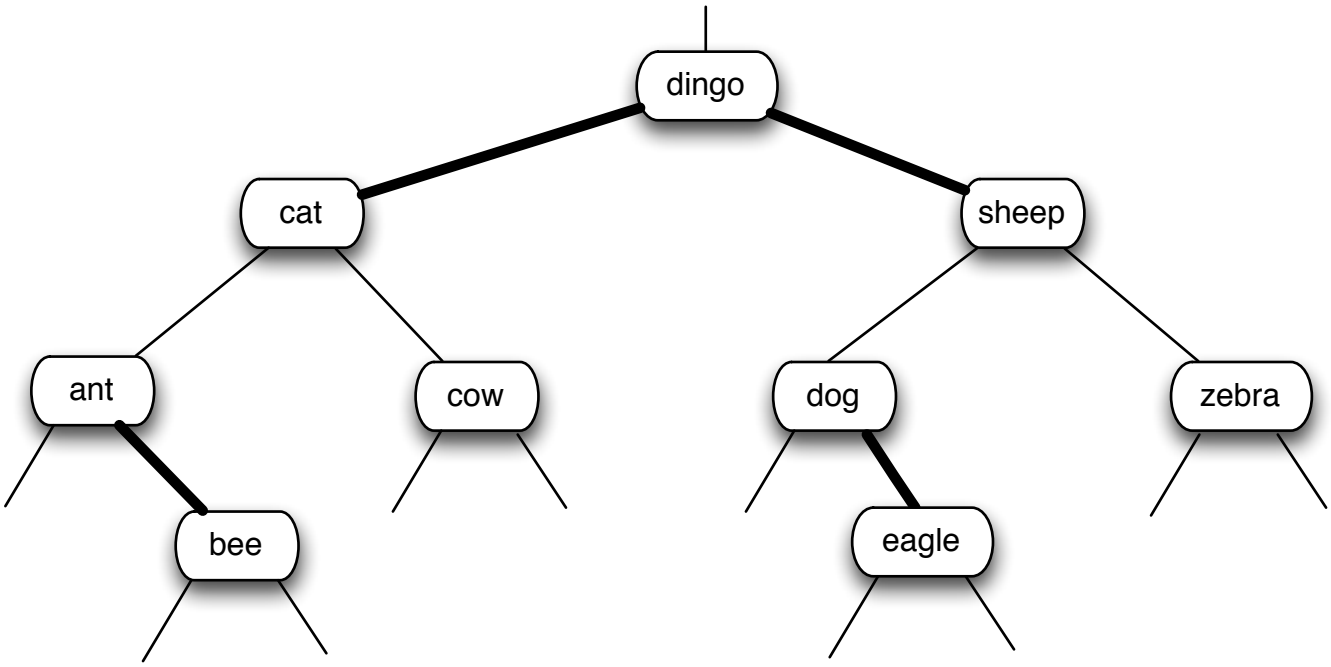
1. (5 points) Draw the 2-3-4 tree that this red-black tree encodes.

   Solution:



2. (10 points) Insert "eagle" into the original tree and re-draw the tree below. You should update the tree according to the rules of red-black tree insertion.

   Solution:



Question 3 20 points

You are to implement a method `getPath` that takes a list of vertices that form a path as a parameter and returns a list of the edges that comprise that path. Suppose that the graph contained vertices v1, v2, and v10. Then calling `getPath(vertices)`, where `vertices` is a list containing v1, v2, and v10 (in that order), should return a list containing an edge incident to v1 and v2 and an edge incident to v2 and v10. If some edge in the supposed path does not exist in the graph the method should throw an IllegalStateException.

Complete the method started below, assuming that it is in a class that implements the `Graph` interface. The graph where the vertices and edges are stored is `this`.

```
/**
 * Finds the edges in a path given the vertices.
 * Precondition - verts is not an empty list.
 * @param verts a list containing the vertices in the path
 * @return the list of edges in the path
 * @throws IllegalStateException if not a valid path
 */
public List<Edge<E>> getPath(List<Vertex<V>> verts) {
```

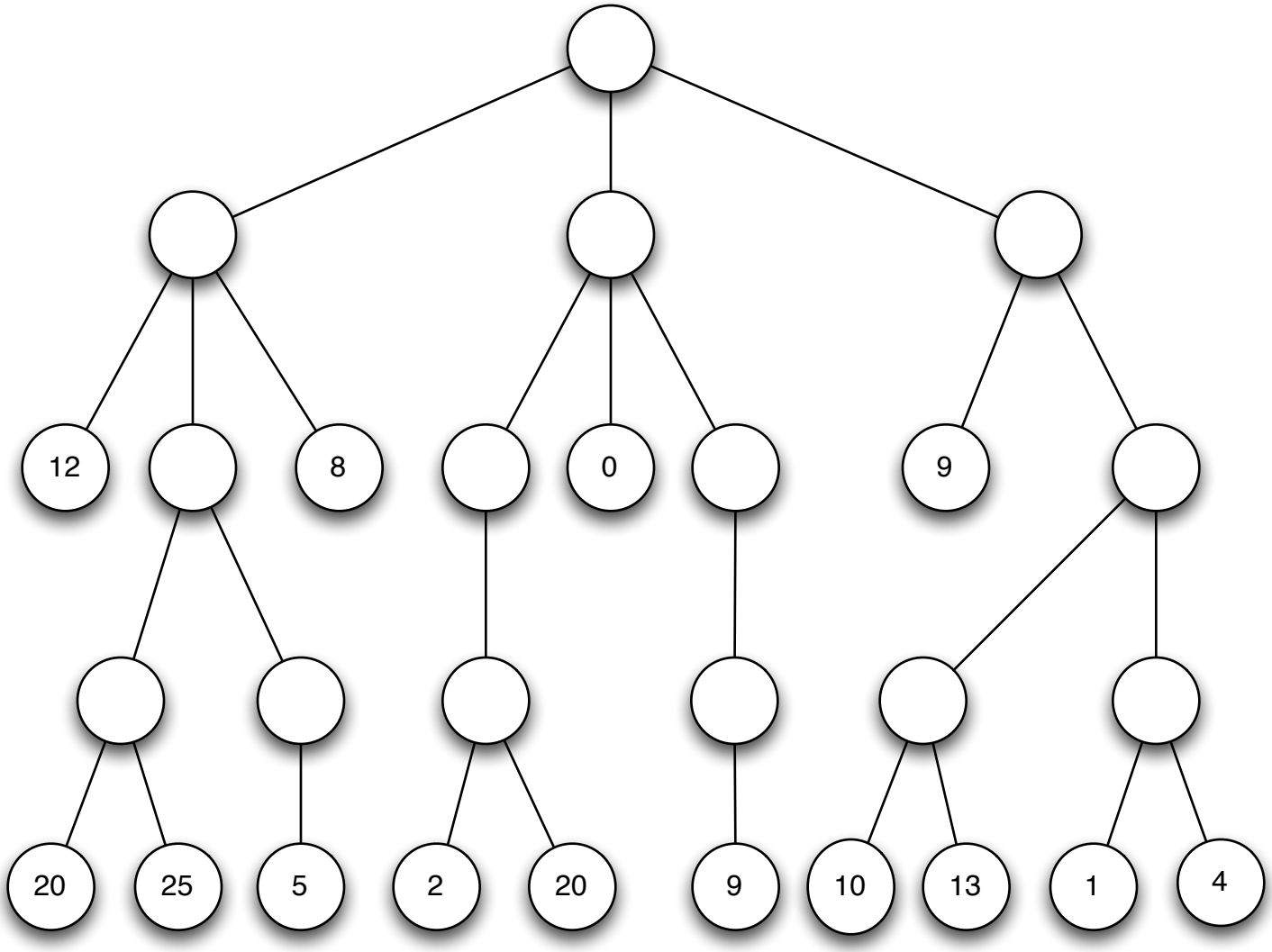
Solution:

```
  List<Edge<E>> returnList = new ArrayList();
  Iterator<Vertex<V>> iter = verts.iterator();
  Vertex<V> first = iter.next();

  while(iter.hasNext()) {
    Vertex<V> second = iter.next();

    if(!areAdjacent(first, second))
      throw new IllegalStateException("Not a valid path");

    Iterator<Edge<E>> edges = this.incidentEdges(first).iterator();
    Edge<E> e = edges.next();  // Must be at least one edge
    while(this.opposite(first, e) != second)  // Must succeed because areAdjacent
      e = edges.next();

    returnList.add(e);
    first = second;
  }
  return returnList;
}
```

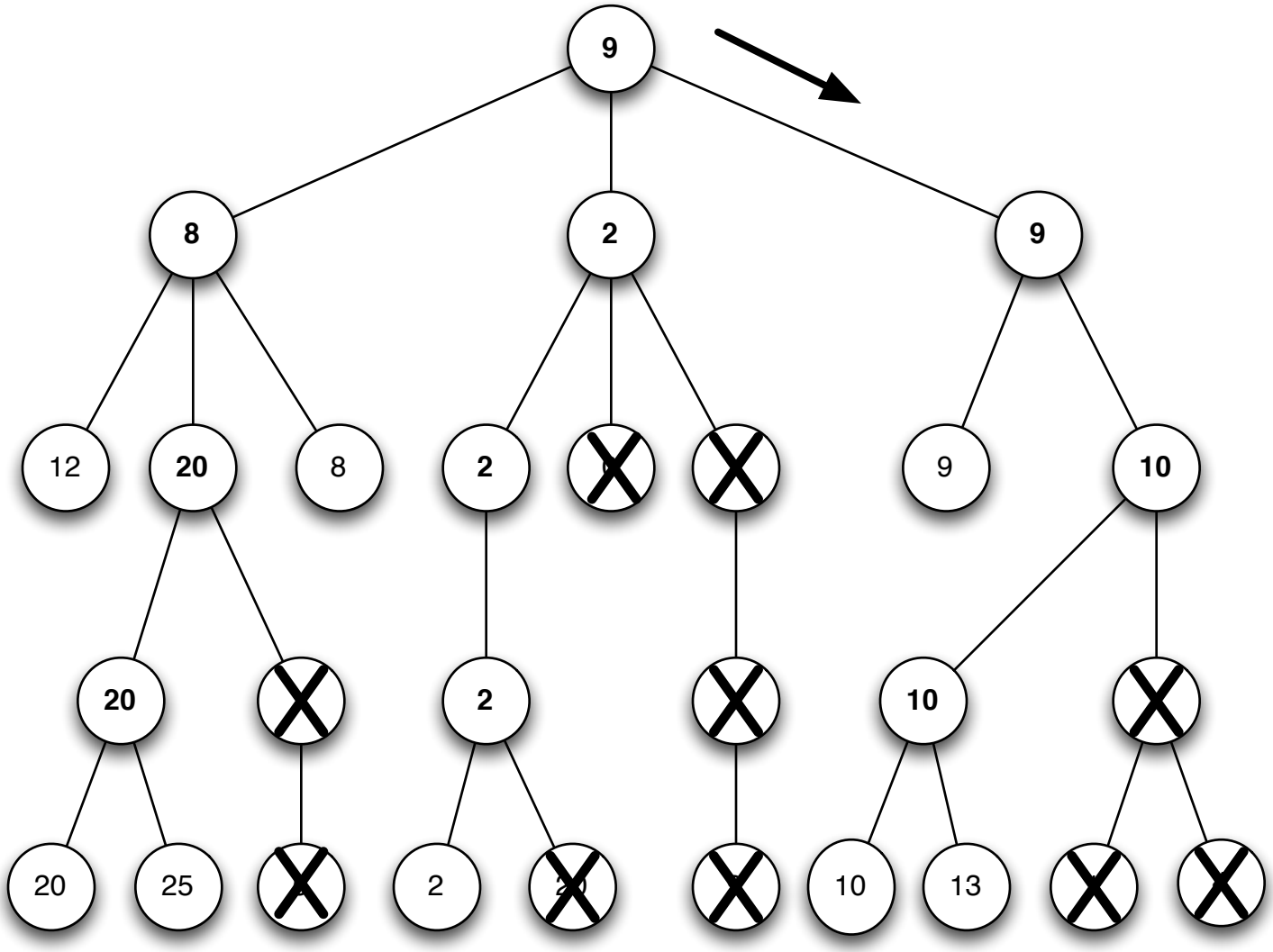
Question 4 15 points

A game tree is shown below. The player on move alternates on each level. Evaluate the tree using game tree search with alpha-beta pruning. Fill in the values of the nodes that get evaluated and draw x's through the nodes that get pruned. Give the value of the game for the first player and show the best first move for that player.

You may evaluate the tree by either having maximizing and minimizing levels or by maximizing at every level but negating the values from the level below, as Kalah does.



Solution:



Question 5 20 points

Write a backtracking method that generates all sequences of numbers taken from a set of positive integers that add up to a given target. Thus if the set of numbers is {1, 6, 3, 2, 9} and your target is 9, your method should print the following (where all of these sequences should appear, but the order in which they are printed does not matter):

```
[1, 2, 6]
[1, 6, 2]
[2, 1, 6]
[2, 6, 1]
[3, 6]
[6, 1, 2]
[6, 2, 1]
[6, 3]
[9]
```

The method that generated the output above is:

```
/**
  * Uses backtracking to find all sequences of numbers from a given set of
  * positive integers that add up to target.  No repetitions allowed.
  * @param numbers the set of numbers that sequences are chosen from
  * @param target the desired sum of the sequence
  */
public static void sequenceSum(Set<Integer> numbers, int target) {
  sequenceSum(numbers, target, new ArrayList<Integer>(), 0);
}
```

You are to complete the helper method begun on the next page. You should do appropriate pruning as you generate partial solutions.

```
/**
 * Helper method.
 * Uses backtracking to find all sequences of numbers from a given set of
 * positive integers that add up to target.  No repetitions allowed.
 * @param numbers the set of numbers that sequences are chosen from
 * @param target the desired sum of the sequence
 * @param sequence the current sequence of integers
 * @param sum the sum of the integers in sequence
 */
public static void sequenceSum(Set<Integer> numbers, int target,
            ArrayList<Integer> sequence, int sum) {
```

Solution:

```
  if(sum == target)         // Found a good sequence?
    System.out.println(sequence);
  else if (sum < target)    // Already too big?  If so, quit
    for(Integer num : numbers)
      if(!sequence.contains(num)) {
        sequence.add(num);
        sequenceSum(numbers, target, sequence, sum + num);
        sequence.remove(sequence.size() - 1);
      }
}
```