Central Washington University
College of the Sciences
Department of Computer Science

CS-301 Data Structures   Fall 2016

Lab Practice 06

In this lab we are going to explore the usefuleness of the `Comparable<>` *interface. We also finish the exploration of recursion, in particular the idea of* backtracking.

Normally you, will find the source and data files in `/home/cs-301/Labs/Lab06`.

```
lb06.pdf
boxes.pdf
```

plus some programs in `WeeklyPrograms`

1. The program `MazeUser.java` finds a path from start to finish in a maze, by using backtracking. The maze is specified in a text file with the following format.

   – the size of the grid (two int)
   – start position (two int)
   – finish position (two int)
   – the full grid (1=Corridor, 0=Wall), a row per line

   Create two simple input files, say of 5×5, 10×8 with given start and finish. Run the programs with your inputs.

2. The elegant `BacktrackDice.java` and its backtracking mechanism, shows all n-dice configurations adding to a given value. Create another another backtracking method, (but much simpler in terms of parameters and actions) that shows *all* the possible outcomes of the n dice. Be careful with n: exponential running time. *Hint:* you do not need to check for the conditions for prunning, nor for the value of `soFar` just before printing `chosen`.

3. One important issue (which the textbook emphazises) but which we have put aside is the issue of *testing.* This simple example shows how by using a script, you can test (run) many inputs with a single command.

   A script is simply a file of executable commands. You can enter your keyboard input in the script in what is called a *here document.* On the server, using nano or vim, enter

   ```
   #!/bin/bash
   echo "MAZE in1"
   java MazeUser <<END
   ```

```
in1
END
echo "------------"
echo "MAZE in2"
java MazeUser <<END
in2
END
echo "-----------"
```

where `in1, in2` are the names of your input files. Save it as `tests` or any other suitable name. `echo` just prints, `END` serves as a sentinel, but any string can be used. To run the script, just type `./tests`

4. File `boxes.txt` contains lengths, widths and heights of a number of boxes. Create an appropriate class

    ```
    public  class Box implements Comparable<Box>
    ```

    that implements the interface. The `compareTo()` must return $1, -1$, depending whether you can *fit* a box inside the other (to make it simple, consider only the corresponding dimensions). It must return 0, if the boxes have the same dimensions (override `equals( Object x)`) if this is the case. Notice that not all pair of boxes will return any of the above, if that is the case return another integer of your choice.

5. Write a small client to test your interface: read the boxes into an `ArrayList<Box>`, and then prompt for two indices that compare the boxes. For example box 2 in the file, can be placed inside box 3.

6. The usefulness of the interface can be applied when transporting the (empty) boxes. For example you may pack three boxes into one, if they fit into each other. Think of a simple algorithm that provides a list of three boxes that can be transported as one. Implement and run the algorithm.