CSC 300 – Data Structures I
Dr. Tony Lowe

Practice Coding Exercises

Learning a second language is partially about the syntax and semantics, but mostly about solving problems. Each problem below focuses on different aspects of Java (see the Hints to find out more) and a type of problem to solve. You can choose any (or all) of these to complete. Each provides a range of points to earn but you do not need to earn all the points. Mix and match. Tackle problems that are interesting, challenging, or to earn additional points.

#1 Dynamic Story (10-25 points)

Write a 'choose-your-own' adventure story. Your program should start a story then give the users a choice of how the story continues by prompting them for a choice of the next action their character takes or such. The choice entry can be as simple picking an option (e.g., 1, 2, or 3) or you can use more complicated commands if you want to make it slightly more like a text adventure game. Adding a text adventure element (**10 bonus points**) lets a user interact more richly with a scene by commanding actions (e.g., open book, drink water). You can create your own or borrow an existing story and add a twist. Your story should continue include several potential twists. Earn bonus points (**5**) by making your story even more interactive by allowing the user to customize their journey (e.g., set a character name, choose their vehicle, dinner, or such!). Think of it like a mash-up of a Mad-libs and choose-your-own adventure!

#2 Tip Calculator (15 points)

Tipping is a very common occurrence in the United States so having a helper is very useful. The most basic version of the tip calculator lets the user enter the purchase amount and the percentage they wish to tip. An add-on feature is to allow the user to add an extra tip such that the final total is an even dollar value. For instance, on a bill of $10.00 a tip of 15% would be $1.50. If the user selects the round-up option, the tip would be an additional $0.50 for a total tip of $2 and a total bill of $12 instead of $11.50.

The final option is to enable a 'command line' call to your functionality. Command line applications run in terminals, and often use parameters to customize their function, much like parameters do for subprograms (i.e., procedures, functions, methods). The last feature asks you to look for and parse command line parameters for the follow protocol.

```
<calculator app name>  -price 10.0 -percent 15 roundup
```

Most of the framework for the command line program exists in the starter code, so you will simply process the array containing the command line parameters (e.g., -price) above.

#3 String Looping (35 points)

Programmers also use loops to process and query text. This exercise includes various examples of working with text and includes a starter video to get you started.

- Swap the case of the text - every upper case letter (A) is lower (a) and vice versa - **(5 points)**
    - `Tony Lowe` -> `tONY lOWE`
- Count the number of times a character appears in a block of text - **(5 points)**
    - `Tony Lowe` contains 2 'o's
- Strip any characters from the text that are not of the core alphabet (a-z, A-Z) - **(5 points)**

    - `Coder 4 ever!` -> `Coderever`
- Reverse the order of the text - **(5 points)**
    - `Tony Lowe` -> `ewoL ynoT`
- Determine if a word is a palindrome - **(5 points)**
    - `Tony` is not a palindrome
    - `racecar` is a palindrome
- Mark any doubles within the text by placing a 2 between them - **(10 points)**
    - `Mississippi` -> `Mis2sis2sip2pi`

#4 Array Manipulations (40 points)

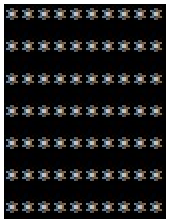
Working with lists and loops is a critical part of programming. Lists hold data that we often want to query for information or calculate statistics for (e.g., averages, medians). You can complete some or all of the following functions around lists.

- Complete the total of a list of numbers (i.e., add them up!) - **(5 points)**
- Complete the mean (i.e., average) - **(5 points)**
- Count the number of times a value appears in the list - **(5 points)**
- Find the smallest value *in the list* - **(5 points)**
- Find the largest value in the list - **(5 points)**
- Calculate the median of the list (i.e., the middle number if the list contains an odd number of values or the average or the two middle numbers if the list is even) - **(5 points)**
- See if any value within the list is ten times the value of any other value in the list - **(10 points)**

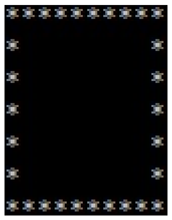
#5 Box Drawing (40 points)

Sometimes a problem's logic requires more than one loop working together to achieve the desired results. Using loops to draw shapes using text provides a very visual demonstration of nested loops. Your code must produce a string to pass the built-in test cases here, not just draw the pictures using print. This exercise includes a video starter to show you how to use the file provided and the general idea of how to nest loops.

- `drawSolidBox` – a box entirely full o the given characters - **(5 points)**
- `drawEmptyBox` – the characters only outline the box; the empty is blank space - **(5 points)**
- `drawRightTriangle` – the characters fill a right triangle - **(5 points)**
- `drawFlippedRightTriangle` – the characters fill a right triangle vertically flipped from the prior- **(5 points)**
- `drawMirroredRightTriangle` – the characters fill a right triangle horizontally flipped from the first - **(10 points)**
- `drawIsosoleseTriangle` – the characters fill an isosceles triangle - **(10 points)**



```
**********
**********
**********
**********
**********
**********
**********
```



```
**********
*        *
*        *
*        *
*        *
*        *
**********
```

#6 Clock Problem (20 points)

Your task is to complete the logic to manage a twenty-four-hour clock (no dates, just time) that tracks the hours, minutes, and sections, and various operations to adjust the time. The framework for your clock is in the `Time` class with the four methods you must complete **(5 points each)**.

1.) `advanceOneSecond`– A user calls this method to advance the clock by one second.
   a. When the seconds value reaches 60, it rolls over to zero.
   b. When the seconds roll over to zero, the minutes advance.
      So 00:00:59 rolls over to 00:01:00.
   c. When the minutes reach 60, they roll over and the hours advance.
      So 00:59:59 rolls over to 01:00:00.
   d. When the hours reach 24, they roll over to zero.
      So 23:59:59 rolls over to 00:00:00.
2.) `compareTo` – The user wants to know if a second time is greater than or less than the time object called, assuming both are on the same date.
   a. Returns -1 if this Time is before `otherTime`.
   b. Returns 0 if this Time is the same as `otherTime`.
   c. Returns 1 if this Time is after `otherTime`.
3.) `add` – Adds the hours, minutes, and seconds of another time (the `offset`) to the current time object. The time should 'rollover' if it exceeds the end of a 24 hour period.
4.) `subtract` – Subtracts the hours, minutes, and seconds of another time (the `offset`) from the current time object. The time should 'roll back' if it precedes the beginning of a 24 hour period.

#7 Impersonation Script Generator (35 points)

Impersonators often take advantage of simple rules to transform normal speech into that of a celebrity or a group of people. We can often codify such rules in a program. Your program transforms a provided quote into one of the modified version as shown in the following table. Your task is first to reverse engineer the manipulation that takes place into a rule, then to code that rule. Each translation is worth **5 points**, with the tricky Yoda option worth **10 points**.

Impersonation	Output
Original	**Four score and seven years ago**
Canadian	**Four score and seven years ago**, eh?
Valley Girl	like **Four** like **score** like **and** like **seven** like **years** like **ago**
Shatner	**Four**... **score**... **and**... **seven**... **years**... **ago**...
Pirate	**Fou**RRR**r sco**RRR**re and seven yea**RRR**rs ago**
Zatanna	ruoF erocs dna neves sraey oga *{words in order, but spelled backwards}*
Yoda	score Four seven and ago years

#8 Validation (30 points)

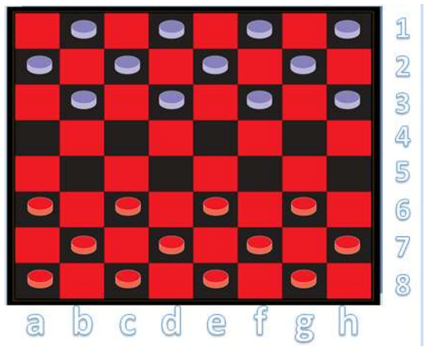
Input validation is a critical behavior that a programmer must master. Without checking the user inputs, most programs would be tough to use at best, and at worst, they do terrible things. This exercise is a good chance to practice `while` loops using the video starter.

https://www.youtube.com/playlist?list=PLgvXUCQxGQryei1WqUApmAzqJhg8nj1CX

- Complete `getRequiredString` – The user must provide text - **(5 points)**
- Complete `getIntegerInput` – The user must provide a number with no decimal points included - **(5 points)**
- Complete `getPositiveInteger` – The user must provide a positive integer - **(5 points)**
- Complete `getNumber` – The user must provide any number - **(5 points)**
- Complete `getMinimumInteger`- The user must provide an integer at least some value or greater - **(5 points)**
- Complete `getSeveralIntegers` – Prompt the user for some number of integers, each of which is required - **(5 points)**
    - *Hint, you already have code to get one integer; how can you do that multiple times?*

#9 Checkers (40 points)

Creating a game is often a goal of programmers, but much of the game is not about pretty graphics but about the underlying engine. Checkers is a very simple game that aligns well with multidimensional arrays. The checkerboard is an 8x8 array, after all! Once again, I am good at starting code but need your help in finishing. This problem is tricky, so read through this entire description before you start.



- initializeBoard() – Setup the checkerboard with the player's pieces in the starting position. Checkers has two players, which you can model as 1 and 2. You should use a 0 to model empty spaces on the checkerboard. The initial setup of the board is as shown in the picture. The 'red' spaces are always black. The red pieces along the bottom are player 1's, and the white pieces are player 2's. Chess players use a numbering system similar to the one shown in this image[1]. Use something similar to align with the numbering system within the multidimensional list.
- makeMove() – Update the board with the provided move. The caller will provide the player and their move. The move is a tuple with 4 values, the column number and row number the piece is moving from, and the column number and row number the piece is moving to. The column values are 0-7, corresponding with a-h and the row values are also 0-7, corresponding with 1-8. Your function can assume the move is valid (that is the next function's responsibility). Make sure that you
    - Place the appropriate player's token at the new location
    - Remove their token from the original location

[1] Chess players use the number 1-8 in the opposite direction, but it is always relative. It seems easier to visualize using lists the way it is drawn!

  - If the move is a jump, remove the opposing player's piece that was jumped
- validGameMove() – This is the tricky one, determining if a move is indeed valid. The caller gives you the player and their move, and you should evaluate the board to see if that move is legal. The move is the same tuple as the makeMove() function. Our version of checkers only deals with the basic pieces (no kings), so you only need to worry about diagonal motion legal to that player. If a player attempts an illegal move, you should use one of the constants defined to report why the move was illegal. The expected return from this function include:
  - `OK_MOVE         = "OK"` ***(The move is valid)***
  - `WRONG_PIECE     = "This space does not contain your piece"`
  - `ONLY_2          = "You can only move a piece one space diagonally, or two when legally jumping"`
  - `PLAYER_1_UP     = "Player 1 must move up"`
  - `PLAYER_2_DOWN   = "Player 2 must move down"`
  - `SPACE_TAKEN     = "The destination space is already taken"`
  - `JUMP_SELF       = "You cannot jump your own piece"`
  - `JUMP_EMPTY      = "You cannot jump an empty space"`
  - `UNKNOWN_INVALID = "Your move is not valid for an unknown reason"`

  The last error is a placeholder and should not occur if you validate the move correctly. Take a few minutes to think about checkers and how you might determine (using a multidimensional list) if a move is valid. You can even try to code some of these ideas before looking at the list below or watching the hint video.
  - First, reject the move if the starting point is not the player's piece.
  - Player 1 must move 'up' (from higher-numbered rows to lower-numbered rows)
  - Player 2 must move 'down' (from lower-numbered rows to higher-numbered rows)
  - Players cannot move onto a space already taken by another piece
  - Players cannot make any move that is more than two spaces vertically or horizontally (rows or columns)
  - If the player is moving just one horizontally and just one vertically, that move is OK!
  - A player can move two spaces horizontally and vertically only if they are jumping the other player, not if they are jumping their own piece or an empty space in the process.

To help you debug your program, the top of the file includes a series of flags that turn on/off additional printing. The debugMove flag prints extra lines when True in the makeMove() function. The debugValidate does the same for validGameMove(). You can control whether the validGameMove() tests run with the runValidationTest flag. If you want to test your code using the UI make playAfterTest True. This flag does not enable a full game of checkers but allows moves up to the moveLimit.

#10 Spelling Test (34 points)

A teacher started an application to help kids practice for the spelling bee, but needs your help to finish the details. They wrote a bunch of structure but need you to complete the following methods in the SpellingBeeHelper class.

- `loadWords` – Load the spelling words and the names of the associated sound files for each word from a text file. A sample of the file is shown in the table below. Return a list of SpellingWord objects.
- `promptTestTaker` – Given a list of words and the input/output streams used to communicate with the user, create the user interface for spelling. A list of constants at the top of the files captures any text shown to the user. The interaction with the user should look as follow
    - Play the audio track for the user (as shown)
    - Prompt the user with the PROMPT text
    - Ensure the user typed in some answer (not blank text) and pass any valid entries to the `recordAttempt` method. If their answer is blank show the user the NO_INPUT_RESPONSE prompt.
    - If the user's answer was correct, move on to the next spelling word
    - If not, show the INCORRECT prompt to the user and get an input to see if they wish to continue. If they enter SKIP_RESPONSE, then skip to the next word. Otherwise start the prompt sequence again.
    - When the test is completed (all words are shown) show the user the COMPLETED prompt
- `recordAttempt` – Write the attempt by the user to a file (ATTEMPT_FILE). The file format includes the following format.
  ```
  <correct word>,<attempted user word>, <time take to respond in millisecond>
  ```
  A sample output from the automated test case is shown below.
- `gradeTest` – Load the data from the provided filename and grade the student's test. The incoming file is the same format as written in `recordAttempt`. When reading the file, capture the number of correctly spelled words, the number of incorrect attempts, and the total number of unique spelling words in the test. Store and return the statistics in a TestResult object.

Sample Word File	Sample basic output

`cat, sounds/Cat.wav`
`inheritance, sounds/Inheritance.wav`
`pedagogy, sounds/Pedagogy.wav`
`keratotomy, sounds/Keratotomy.wav`
`prospection, sounds/Prospection.wav`
`curmudgeon, sounds/Curmudgeon.wav`
	`Word0,wrong0,2774`
	`Word0,Word0,1481`
	`Word1,wrong0,2657`
	`Word1,wrong1,2197`
	`Word1,Word1,2831`
	`Word2,wrong0,1094`
	`Word2,Word2,1954`

Tips: Work through method in the order above and its associated JUnit test. The user input test is expected exactly the behavior described above, so meticulously pay attention to the order of the prompts and user the constants provided!