Tutorial Course 2

This course was thought to be done in java, but you are free to do it in any language you find suitable, or are more comfortable with.

Unit tests

I. The Game of Life

Wikipedia's article

Rules

The universe of the *Game of Life* is an infinite, two-dimensional grid of square *cells*, each of which is in one of two possible states, ***alive*** (1) or ***dead*** (0). Every cell interacts with its eight *neighbors*, which are the cells that are horizontally, vertically, or diagonally adjacent.
At each step in time, the following transitions occur:

- Any living cell with 2 or 3 neighbors survives.
- Any dead cell with 3 live neighbors becomes a live cell.
- All other live cells die in the next generation. Similarly, all other dead cells stay dead.

Instructions

You are given the source code that implements the previous game of life algorithm, which is not unit tested.

1. Download the source code in moodle under the section "TP2", TheGameOfLife.zip
2. Import it and into your IDE and launch the program (understand the algorithm and change the inputs in order to see some different outcomes)
3. Write all the unit tests that you find necessary and modify the source code in order to ensure the following specifications:
    a. The algorithm must be consistent with the base rules of the game of life (cf "Rules") You should mainly focus on (next state computation, the count of the living neighbors, the data structures, the initialization)
    b. The boundaries (n, m) of the grid has to be $n > 1$ & $m > 1$
        i. You may modify the code if you judge it insufficient
4. BONUS: Implement a better error handling for this algorithm and add the unit tests that go along.

II. Merge Sort

Implement and unit test the Merge sort algorithm.

Merge sort is more complicated. The simplest implementation of merge sort uses recursion, but we'll use a different implementation that does not use recursion. The main idea in merge sort is that given two sorted arrays, we can merge them relatively easily to form one (larger) sorted array. Start by writing a function merge:

```
function t = merge(v,w)
% Merge sorted vectors v and w to form vector t, which is also sorted.
% v,w: sorted numeric vectors not necessarily of the same length
% t: sorted vector whose length is length(v)+length(w)
```

Your code should take advantage of the fact that v and w are already sorted. We begin by considering a vector x with length n where n is a power of 2. Here's the general idea:

1. Set the subvector length m to be 1. (Why? It's the shortest vector that is "sorted.")
2. Divide vector x into subvectors of length m.
3. Merge every two adjacent subvectors. I.e., merge subvectors 1 and 2, 3 and 4, 5 and 6, . . . , so that the merged subvector (each of length 2m) is sorted.
4. Double the value of m. (That's the length of each merged—sorted—subvector.)
5. Repeat steps 2 to 4 until < ? >. You should figure this out.

Let's look at an example. The vector below is of length 8.

3	4	5	2	7	8	9	3

With a subvector length of 1, we have 8 subvectors. Merging subvectors l and 2, 3 and 4, 5 and 6, and 7 and 8 gives us the following:

3	4	2	5	7	8	3	9

Now we have four sorted subvectors of length 2. Then we merge subvectors 1 and 2, 3 and 4, giving us two sorted subvectors of length 4:

2	3	4	5	3	7	8	9

Merging these two subvectors gives us the sorted vector of length 8, completing the sorting problem:

2	3	3	4	5	7	8	9

You will generalize this non-recursive algorithm for any positive n value. The key is that the merge step should allow for subvectors of different lengths. Consider a vector of length 11:

4	3	1	9	8	7	2	0	6	5	1

When the subvector length m is 1, there are 11 subvectors and five merges. There is not a sixth merge because there isn't a "right-side" subvector to carry out a merge. Therefore the 11th subvector is left alone.

3	4	1	9	7	8	0	2	5	6	1

Now we have five subvectors of length $m = 2$ and one leftover component, which we'll call the deficient subvector. We will do three merges but note that the last merge has a deficient (short) right-side subvector. The result is

1	3	4	9	0	2	7	8	1	5	6

Now we have two subvectors of length $m = 4$ and one deficient subvector. We can only do one merge since the last subvector doesn't have a right-side partner. So the last subvector is left alone:

0	1	2	3	4	7	8	9	1	5	6

Now $m = 8$. We have one subvector of length 8 and one deficient subvector. Merge them and we're done.

You must implement this non-recursive algorithm in the function `mergeSort`. `mergeSort` must call `merge`, specified above.

```
function sortedArray = mergeSort(A)
% Sort vector A using the non-recursive merge sort algorithm as discussed
% A: the vector of numbers to be sorted, the length of A is > 0
% sortedArray: the sorted vector
```

II. Radix sort

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. Its input is a list of unsorted numbers and it outputs a sorted list.

The radix (or base) of a number is the number of unique digits that can be used for its representation, e.g.:

- Radix of 2: binary numeral system (00001, 1010111,...)
- Radix of 10: Decimal system (7, 83, 9246,...)
- Radix of 16: Hexadecimal system (4, ce7, 6ba4,...)

Radix sort works as follows:

Input list (base 10):

[170, 45, 75, 90, 2, 802, 2, 66]

Starting from the rightmost (last) digit, sort the numbers based on that digit:

[{170, 90}, {02, 802, 02}, {45, 75}, {66}]

Notice that a 0 is prepended for the two 2s so that 802 maintains its relative order as in the previous list (i.e. placed before the second 2) based on the merit of the second digit.

Sorting by the next left digit:

[{02, 802, 02}, {45}, {66}, {170, 75}, {90}]

And finally by the leftmost digit:

[{002, 002, 045, 066, 075, 090}, {170}, {802}]

Deliverable:

- A working implementation of radix sort for decimal number
- Some unit test for ensure the correct functionality
- Bonus: extend your implementation so it also works for an arbitrary radix (e.g. 3, 8, 16,...)

III. TDD : Bowling score computing

Rules

Create a program, which, given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game

We can briefly summarize the scoring for this form of bowling:

- Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
- In each frame, the bowler gets up to two tries to knock down all the pins.
- If in two tries, he fails to knock them all down, his score for that frame is the total number of pins knocked down in his two tries.
- If in two tries he knocks them all down, this is called a "spare" and his score for the frame is ten plus the number of pins knocked down on his next throw (in his next turn).
- If on his first try in the frame he knocks down all the pins, this is called a "strike". His turn is over, and his score for the frame is ten plus the simple total of the pins knocked down in his next two rolls.
- If he gets a spare or strike in the last (tenth) frame, the bowler gets to throw one or two more bonus balls, respectively. These bonus throws are taken as part of the same turn. If the bonus throws knock down all the pins, the process does not repeat: the bonus throws are only used to calculate the score of the final frame.
- The game score is the total of all frame scores.

TDD (french article) / TDD (english article)

Questions

- Using a TDD (Test Driven Development) approach, create a program that solves this bowling score computing problem.

Requirements

Write a **Game** class, that has two methods: (you can/should of course have as many classes as you want e.g. :Frame, the following constraint applies only for the Game class)

- **void roll(int)** is called each time the player rolls a ball. The argument is the number of pins knocked down
- **int score()** returns the total score of that game.