CSCI 3005 – SP 2018 – PROGRAMMING ASSIGNMENT 1: *Slide or Jump?*

As you wait impatiently for the next Algorithms lecture, a kid approaches you and challenges you to play a game called *Slide or Jump?* The game is as follows: you stand at one end of a board consisting of several cells, with each cell containing a non-negative integer that represents the cost of visiting that cell (as in the example below):

☺	5	75	7	43	11

From any cell, you have one of two possible moves: either *slide* to the adjacent cell or *jump* over the adjacent cell. The objective of the game is to reach the last cell while minimizing the sum of the costs of the visited cells. For the sample board above, the cheapest strategy is to slide, jump, and jump, with overall cost of 23 (5 + 7 + 11).

Despite your best efforts, it seems as if your opponent is consistently beating you. So you decide to take matters into your own hands by writing a program that determines the optimal sequence of moves. After some thought, you realize that the overall cost reaching the goal can be defined recursively. First, consider the base cases:

- If you are in the last cell, the overall cost is equal the cost of that cell (already at the end!!)
- If you are in the cell adjacent to the last cell, the overall cost is the sum of the costs of the last two cells (since your only possible move is to slide)
- If you are in the third cell from the end, the overall cost is the sum of the cost of that cell and the cost of the last cell (since jumping allows you to reach the end with a lower overall cost)

In the general case, let *totalCost(n)* be the minimum cost of reaching the goal from cell n, then:

totalCost(n) = cost of cell n + either the cost of sliding or the cost of jumping (whichever is smaller), or in other words

totalCost(n) = cost of cell n + min(cost(n + 1), cost(n + 2))

After coding the recursive routine, you realize that it takes too long for long boards, mainly because of the many overlapping recursive calls required. Therefore, you decide to try a dynamic programming approach. When using this approach, your solution should store partial solutions in an n-element array, so that the array element at location k contains the value of totalCost(k). The algorithm can then compute and store the values of totalCost(k) without the need to recalculate partial results (starting at the end of the board and working towards the first cell).

Your assignment is to write a Java class named **SlideOrJump** which provides public methods with the following signatures and functionality:

```
SlideOrJump(int[] board)      // board will have at least two elements, the first one always being zero
long recSolution()            // computes overall cost recursively  (required for A-B-C credit)
long dpSolution()             // computes overall cost using dynamic programming (required for A-B credit)
String getMoves()             // returns sequence of moves required (required for A credit, see format below)
```

You should initially test your methods using the `SlideJumpTest` class provided. Here is a sample sequence of method calls and return values:

```
SlideOrJump game = new SlideOrJump(array);  // array = {0, 5, 75, 7, 43, 11}
game.recSolution();                         // returns 23
game.dpSolution();                          // returns 23
game.getMoves();                            // returns "SJJ"  representing slide, jump, jump
```

For full credit, include code to count and display the number of operations performed by each of the solution methods. In the recursive solution, count method calls. In the dynamic programming solution, count loop iterations. Collect enough data to produce a report that includes a summary of the data presented in both tabular and graphical formats. Analyze the data and form a conclusion regarding the relative efficiency of the methods. Your report should be submitted via Moodle in a file named **sj.pdf.** The source code for **SlideOrJump.java** should be submitted to Mimir.