CPSC 320 Assignment #6

March 16, 2015

Due date: Monday, 2015/03/23 at 5PM

You'll submit this assignment electronically (only one submission per team of up to two students, please!) through handin. Hand in to `cs320` with the target `assn6` by the due time above.

In your submission, include **a plain text file named README.txt** with all relevant information from our assignment cover page and any special notes you have; **a plain text file named SOLUTIONS.txt with the textual information requested below**; and **the graph image requested below**. **Clearly indicate where each problem begins and ends. Make code clear, well-indented, and well-commented!** However, you need not include peripheral material (e.g., your `main`, boilerplate like the `#lang` line in a Racket file, setup required for your testing, etc.).

We guarantee that we will mark at least 5 parts the assignment.

For this assignment, you'll work through a thorough solution to the textbook's problem 6.6 (on page 317, the one beginning "In a word processor, the goal...").

1. Give a good set of examples to explore the problem. (A "good" set should explore trivial cases, cases that may be base cases for your recurrence, a couple of small but not trivial cases, and at least one substantial case. For the last, you may use the one given in the text.) There are some implicit constraints on instances of the problem; be sure to give at least one example that violates these and explain why it's an instance that cannot be solved. **Submit:** examples and solutions (sum of squares of slack space and correct line breaks) as text, along with the discussion of constraints.

2. Give a mathematical recurrence for the **cost** (sum of squares of slacks) of the optimal solution. (If you like, you may instead give short, recursive pseudocode. This shouldn't be an efficient solution, however.

It's just a correct recursive solution without unnecessary parameters.) **Submit the recurrence (or pseudocode) as text.**

3. Give an analysis of the memory use and runtime of a memoized version of the recurrence. (You shouldn't need to implement it to analyse it!) **Submit your bounds and the work you used to get them as text.**

4. Tell us your favorite language. Use it for the implementation problems below. **Submit just the name of the language so we know what we're reading!**

5. Implement the recurrence naïvely. (I.e., straightforwardly translate the recurrence to code, which can have (super-)exponential runtime.) **Submit the recursive version as text.**

6. Implement a memoized version of the same code. **Submit the memoized version version as text.**

7. Implement a dynamic programming version of the same code (maintaining the whole table). **Submit the dynamic programming version as text.**

8. Implement a "solution extractor" that takes the original instance and the table of solutions created by the memoizing or dynamic programming solution and outputs the text nicely divided into lines. Adapt your test cases to test this "end-to-end" solution. (You may "feed it" with either your memoized or DP solution.) **Submit the solution extractor as text.**

9. Implement a function that takes in a number n and a list of words—one per line—and produces a random "text", a list of words each chosen uniformly at random from the word list (allowing duplicates). You can find a great list of words on `remote.ugrad.cs.ubc.ca` in `/usr/share/dict/words`. **Submit the function as text.**

10. Graph the original recursive, memoized recursive, and dynamic programming solution on problems of a few interesting sizes using your generator above. **Submit:** the graph as a separate JPEG, PNG, BMP, GIF, or PDF file named "GRAPH" with an appropriate extension.