CS242 - Spring 2021 - Assignment #3

Assigned: March 13th, 2021
Due: March 28th, 2021
(Extra credit: May 1st, 2021)
NO LATE SUBMISSIONS.
Non-coding answers may be resubmitted once, after receiving corrections on the first attempt. That is, if you do not submit anything you cannot resubmit.

Collaboration policy: The goal of assignment is to give you practice in mastering the course material. Consequently, you are encouraged to collaborate with others. In fact, students who form study groups generally do better on exams than do students who work alone. If you do work in a study group, however, you owe it to yourself and your group to be prepared for your study-group meeting. Specifically, you should spend at least 30–45 minutes trying to solve each problem beforehand. If your study group is unable to solve a problem, it is your responsibility to get help from the instructor before the assignment is due.

For this assignment, you can form a team of up to three members. Each team must write up each problem solution and/or code any programming assignment without external assistance, even if you collaborate with others outside your team for discussions. You are asked to identify your collaborators outside your team. **If you did not work with anyone outside your team, you must write "Collaborators: none."** If you obtain a solution through research (e.g., on the web), acknowledge your source, but write up the solution in your own words. **It is a violation of this policy to submit a problem solution that any member of the team cannot orally explain to the instructor.** No other student or team may use your solutions; this includes your writing, code, tests, documentation, etc. It is

a violation of this policy to permit anyone other than the instructor and yourself read-access to the location where you keep your solutions.

Submission Guidelines: Your team has to submit your work on Blackboard (no email) by the due date. Only one submission per team is necessary. For each class in the programming assignments you must use the header template provided in Blackboard. Make sure that you identify your team members in the header, and any collaborators outside your team, if none, write "none". Your code must follow the Java formatting standards posted in Blackboard. Format will also be part of your grade. To complete the submission, you have to upload two files to Blackboard: your source file and your class file. Your answers to questions that do not require coding must be included in the remarks section of the header. **The submission will not be accepted in any other format.**

Style and Correctness: Keep in mind that your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Assignment #3 Grading Rubric

Coding:

Program characteristic	Program feature	Credit possible		
Design 30%	Algorithm	30%		
Functionality 30%	Program runs without errors	20%		
	Correct result given	10%		
Input 15%	User friendly, typos, spacing	10%		
	Values read in correctly	5%		
Output 15%	Output provided	10%		
	Proper spelling, spacing, user friendly	5%		
Format 10%	Comments, name	5%		
	Indentation	5%		
	TOTAL	100%		

Non-coding:
Embedded in questions.

1(15)	2(40)	3(15)	4(15)	5(15)	**TOTAL**(100)	EC

Assignment:

As you know at this point, Dynamic Programming is a powerful technique that explores all possible cases to obtain a solution, yet achieves an exponential improvement in running time when compared with a simpler brute-force approach. A well-known example of Dynamic Programming usage is Text Justification, where we are given an array of words and a page width and we have to decide how to split the text so that it looks "nice" after justifying against margins. The purpose of this homework is to compare experimentally the result obtained justifying text with a greedy algorithm (as used by MS Word) against doing the same using LaTeX rules and Dynamic Programming.

Specifically, your assignment is the following.

1. (15 points) Implement a "badness" method that, given a width ω as a number of characters, an array $W[0 \ldots n-1]$ of strings, and two indices i and j such that $0 \leq i \leq n-1$ and $1 \leq j \leq n$, computes the following function used by LaTeX.

$$badness(W, i, j, \omega) = \begin{cases} (\omega - \ell(W, i, j))^3 & \text{if } \omega - \ell(W, i, j) \geq 0, \\ \infty & \text{otherwise.} \end{cases}$$

   Where $\ell(W, i, j)$ is the total length in characters of the words from index i to $j-1$ in the array W, taking into account that one space must be left in between each pair of consecutive words.

2. (40 points) Implement a "split" method that, given as parameters the width ω and the array W, returns a list L of the indices of the array where each line should start to minimize the aggregated badness (i.e. the sum of the badness of all lines in such split). The Dynamic Programming pseudocode to compute the minimum aggregated badness seen in class follows in Algorithms 1 and 2.

3. (15 points) Implement a "justify" method that, given as parameters the width ω, the array of words W, and the list of breakpoints L, creates a text file, call it for instance `just.txt`, justified accordingly. That is, for each line, your code should spread the words evenly in the width of ω characters. Hint: a line starting at index i and ending at index $j-1$ has $j-i$ words. So, the $\omega - \ell(W, i, j)$ space characters have to be distributed roughly evenly among the $j-i-1$ separations of words.

Algorithm 1: `Minimum Badness`

input : an array $W[0 \ldots n-1]$ of n strings and an integer ω.
output: an array with the indices of W where the next line of each suffix of W should start to obtain the minimum aggregated badness.

```
1 memo[0...n] ← new array of integers
2 linebreaks_memo[0...n] ← new array of integers
3 for i = 0 to n do memo[i] ← −1
4 Memoized Minimum Badness(W, 0, memo, linebreaks_memo, ω)
5 return linebreaks_memo
```

4. (15 points) Implement a main method that prompts the user to enter a number of words n and a page width ω as a number of characters. Then, create the array $W[0 \ldots n-1]$ of strings and fill it with random strings. The length of each string should be chosen at random in the range $[1, 15]$. Notice that, to the extent to evaluate the result of justification, the actual characters used are irrelevant. For instance, you could choose all the characters to be the same, as in "aaaaaaaaaaa". (If you prefer to use real text, read the words from an input file or the web.) Then, your main method should call the method "split" to obtain the list of breakpoints L, and subsequently call the method "justify" passing W, ω, and L as parameters. Finally, output the array W to a second text file in a single line, call it for instance `unjust.txt`.

5. (15 points) Evaluation: Compare the result of `just.txt` with `unjust.txt` opening the latter in MS Word and justifying the text. How do they look? Can you evaluate if MS Word is using a greedy approach? Play with the maximum word length of 15 as needed.

Algorithm 2: `Memoized Minimum Badness`

input : an array $W[0 \ldots n-1]$ of n strings, an integer i in the range $[0, n]$, an array $memo[0 \ldots n]$ of integers, an array $linebreaks\_memo[0 \ldots n]$ of integers, and an integer ω.

output: minimum aggregated badness of the suffix [i,n-1] of W.

```
1  if memo[i] ≥ 0 then return memo[i]
2  if i = n then
3      memo[i] ← 0
4      linebreaks_memo[i] ← n
5  else
6      min ← ∞
7      index_of_min ← 0
8      for j = i + 1 to n do
9          temp ← badness(W, i, j, ω)
10         temp ← temp +
               Memoized Minimum Badness(W, j, memo, linebreaks_memo, ω)
11         if temp < min then
12             min ← temp
13             index_of_min ← j
14     memo[i] ← min
15     linebreaks_memo[i] ← index_of_min
16 return memo[i]
```