Algorithms and Data Structures 2014/15
Coursework 2

Issue date: Thurs, 23rd October 2014

The deadline for this coursework is 4pm on Wednesday 12th November, 2014 (Wed week 9). You will need the file `LineFormat.java`*, available from the course webpage. Please submit your solutions electronically via* `submit` *as described on the final page of this specification.* **Remember the School's policy is that late coursework will not be accepted without good reason (given in advance). If you have good reason, this goes through your PT.**

This coursework should be your own individual work. You may discuss understanding of the questions with your classmates, but may not share solutions, or give strong hints. If you use any resources apart from the course slides/notes, or the book, you must cite these.

This is worth 50% of the coursework for A&DS.

In this coursework we consider the problem of neatly formatting a paragraph of words on the screen. Suppose that L characters fit into one line, and our paragraph consists of the words $w_1, \ldots, w_n$ (in this order). We assume that the length of each word is at most L, so no individual word needs to use more than one line. In printing the paragraph, we will always put an empty space between two successive words which are being printed on the same line. Our goal is to lay out the words in sequence on a series of *lines* of the fixed length L, to somehow minimize the "trailing whitespace" that remain at the ends of the lines. For the purposes of this assignment we will use $L = 80$.

For any pair of indices $i, i' \in \{1, \ldots, n\}$ with $i < i'$, we define the "leftover space" $e_{|\mathbf{w}|}(i, i')$ at the end of a single line containing the words $w_i, \ldots, w_{i'}$ as follows:

$$e_{|\mathbf{w}|}(i, i') \;=\; L - \Big(\sum_{j=i}^{i'} |w_j| \Big) - (i' - i),$$

where $|\mathbf{w}| = (|w_1|, \ldots, |w_n|)$, and where $|w_j|$ denotes the length of word w_j. The term $i' - i$ accounts for the spaces that must be inserted between the words.

Our goal, for this coursework, is to develop algorithms to print a paragraph in such a way that the *sum of the squares* of the number of empty spaces at the end of each line (except perhaps the last line) is minimised. We will study (and implement) three algorithms as possible solutions for this problem - one algorithm will use a *greedy* strategy, and the two others apply the technique of *dynamic programming.* Part of the work you will do is theoretical (developing the dynamic programming algorithms, and proving a $\Theta(\cdot)$ bound on their running time). The rest of the work will involve implementing the algorithms and studying their "sum of squares" performance on randomly generated lists of word lengths.

We first present a couple of extra definitions. The standard metric used to evaluate the quality of a line formatting of sequence of words is the *sum of squares* of the trailing spaces at the end of the lines. Sometimes a decision is made to ignore *the final line* when summing these squares, because we know that there will have to be trailing whitespace on the final line anyhow. Suppose we decided to split our sequence of words into m lines as follows, of course assuming that each line has enough space for the allocated words and their intermediate spacings:

$$\begin{gathered} w_1 w_2 \ldots w_{i_1} \\ w_{i_1+1} \ldots w_{i_2} \\ \vdots \\ w_{i_{m-1}+1} \ldots w_n \end{gathered}$$

We represent such a formatting in terms of a list of the "breakpoints" - for the format above, this list would be $\mathbf{i} = (i_1, i_2, \ldots, i_m)$, with $i_m = n$ always[1]. Although we will not include i_0 in $\mathbf{i}$, we assume it has value 0.

Assumption: We will insist that a list of "breakpoints" $\mathbf{i} = (i_1, \ldots, i_{m-1}, i_m)$ (with $i_m = n$) for a formatting satisfies the constraint that that $e_{|\mathbf{w}|}(i_{k-1}+1, i_k) \geq 0$ for every $1 \leq k \leq m$. We only consider formattings where the list of breakpoints satisfies this constraint.

Definition 1: For a particular formatting given by the list of breakpoints $\mathbf{i}$ of length m, the "sum-of-squares cost" of formatting the paragraph of words with lengths $\ell = (\ell_1, \ell_2, \ldots \ell_n)$ (where $\ell_j = |w_j|$) this way is then

$$ss(\ell; \mathbf{i}) = ss(\ell;\ i_1, \ldots, i_{m-1}, i_m) \ =_{\text{def}} \sum_{k=1}^{m} e_\ell(i_{k-1}+1, i_k)^2.$$

The "adjusted sum-of-squares cost" (when the final line is ignored) is then

$$ss^*(\ell; \mathbf{i}) = ss^*(\ell;\ i_1, \ldots, i_{m-1}, i_m) \ =_{\text{def}} \sum_{k=1}^{m-1} e_\ell(i_{k-1}+1, i_k)^2.$$

Our goal in this coursework will be to develop algorithms which generate formattings which have low values of ss and ss^*. You will develop and implement algorithms to generate line formats, and will carry out an experimental comparison of the *greedy* algorithm versus the *dynamic programming algorithms*. The *dynamic programming* algorithms will be developed in two stages: you will first develop a dynamic programming algorithm to compute a line formatting $\mathbf{i} = (i_1, i_2 \ldots, n)$ which has the minimum possible value for $ss(\ell;\ \mathbf{i})$; and subsequently you will show how to adapt that algorithm to compute a line formatting which has an optimum value for $ss^*(\ell;\ \mathbf{i})$.

1 $\Theta(n)$ greedy implementation (10 marks)

In this section we discuss a very simple greedy algorithm for constructing line formattings. The greedy algorithm is not guaranteed to output an optimal formatting with respect to the either of the sums-of-squares measures; however we will implement it in order to test its experimental performance against the performance of the dynamic programming algorithms (ie, to see how bad it is in practice).

Unlike the dynamic programming algorithms we will develop later, the greedy algorithm does not need to make use of *any* table. It simply takes in a list of word lengths, and iterates through this list, keeping a running total of the current line size - whenever this line size

[1]Observe that for *any* format, we definitely have $i_1 \leq i_2 \leq \ldots \leq i_m$.

will become greater than L by the addition of the next word w_j, we append $j - 1$ to the list of "breakpoints" and reset the current line size to 0. Note that this ensures that the list of breakpoints will satisfy $e_{|\mathbf{w}|}(i_{k-1} + 1, i_k) \geq 0$ for every adjacent pair of breakpoints i_{k-1}, i_k in the output list. Here is some pseudocode:

Algorithm GREEDY(ℓ)

```
1. $n \leftarrow \ell.length$; $size \leftarrow 0$; $B \leftarrow$ NIL
2. for $i \leftarrow 1$ to $n$ do
3.         if $L - size < \ell_i$ then
4.                 append($B, i - 1$)
5.                 $size \leftarrow \ell_i + 1$
6.         else
7.                 $size \leftarrow size + (\ell_i + 1)$
8. append($B, n$)
9. return B
```

Observe that GREEDY runs in $\Theta(n)$ time. Your first task for this coursework is to implement the GREEDY algorithm in Java, within the template file `LineFormat.java`, as a method of the following type:

```
public static int[] greedyLF(int[] wordLengths)
```

The input `wordLengths` will be a list of integers of length n, containing the lengths of words in the input file in sequential order. The output will be a list of integers of length at most n, containing the "breakpoints" for the greedy formatting.

2 Dynamic programming algorithm for ss (15 marks)

In this section, you are asked to develop a dynamic programming algorithm which takes a list of *word lengths* $\ell_1, \ldots, \ell_n$ and construct the *subsequence* $\mathbf{i} = (i_1, i_2, \ldots, i_m)$ of $1, 2, \ldots, n$ (always with $i_m = n$, and refereed to as a *breakpoint sequence*) such that $\mathsf{ss}(\ell;\ \mathbf{i})$ is the minimum possible, taken over all breakpoint subsequences $\mathbf{i}$ which satisfy the Assumption from the Introduction.

Definition 2: For a given sequence of word lengths $\ell = \ell_1, \ldots, \ell_n$, we define the term $\mathsf{opt}(\ell)$ to denote the breakpoint sequence $\mathbf{i}$ for ℓ for which $\mathsf{ss}(\ell;\ \mathbf{i})$ is the least possible value, and $\mathsf{vopt}(\ell)$ to denote the value itself. We will similarly define $\mathsf{opt}^*(\ell)$ and $\mathsf{vopt}^*(\ell)$ with respect to the $\mathsf{ss}^*(\ell;\ \mathbf{i})$ measure.

For any $k, 1 \leq k \leq n$, define $\ell[k]$ to be the truncated sequence of word lengths $\ell_1, \ldots, \ell_k$ (observe that $\ell[n] = \ell$).

We will now develop a recurrence for $\mathsf{opt}(\ell)$, by considering the options for the various line formatting of the list of input words $w_1, \ldots, w_n$ (with $|w_i| = \ell_i$ for all $i = 1, \ldots, n$). Our goal is to find the formatting $\mathbf{i} = (i_1, \ldots, i_{m-1}, n)$ (for arbitrary m) which has the minimum value of $\mathsf{ss}(\ell;\ \mathbf{i})$, over all formattings satisfying the Assumption. Consider an arbitrary such formatting, as shown below:

$$\begin{array}{c} w_1 w_2 \ldots w_{i_1} \\ w_{i_1+1} \ldots w_{i_2} \\ \vdots \\ w_{i_{m-1}+1} \ldots w_n \end{array}$$

We can make the following key observation about a formatting of $w_1, \ldots, w_n$:

Observation 1: Any overall formatting $\mathbf{i}$ of words $w_1, \ldots, w_n$ is equivalent to a formatting of words $w_1, \ldots, w_k$, followed by one extra line containing the words $w_{k+1}, \ldots, w_n$, for some $k < n$ such that $e_{|\mathbf{w}|}(k+1, n) \geq 0$. If the list of breakpoints $\mathbf{i}$ has length m, observe that $ss(\ell; \mathbf{i})$ is equal to

$$ss(\ell[i_{m-1}];\ i_1, \ldots, i_{m-1}) + e_\ell(i_{m-1}+1, n)^2.$$

For the formatting mentioned above, we have $k = i_{m-1}$; however, when solving the problem of computing $\mathtt{opt}(\ell)$, we do not know where the line breaks occur in advance (in particular, we do not know the value of i_{m-1}, or even the value of the number of lines m). However, we do know that $(n-k) \leq (L+1)/2$, because we need to fit $(n-k)$ words (each of length at least 1), and $(n-k-1)$ spaces into the L characters of the final line. This bounds the number of possible options we need to consider for the layout of the final line. Now observe that in searching for the "best formatting" for the words $w_1, \ldots, w_n$, a natural recursive approach could be to consider each of these possible k-values in turn, and then find the "best formatting for $w_1 \ldots w_k$" and add $e_\ell(k+1, n)^2$ to that. Clearly the best overall formatting for $w_1, \ldots, w_n$ will correspond to at least one of these optimal decompositions into "all lines but the last one" and "the final line" (unless $w_1, \ldots .w_n$ all fit on one line). We can now formulate a recurrence for $\mathtt{opt}$:

Observation 2: For any list of input words $w_1, \ldots, w_n$ with lengths given by $\ell = (\ell_1, \ldots, \ell_n)$,

$$\mathtt{vopt}(\ell) \quad = \quad \begin{cases} e_\ell(1,n)^2 & \text{if } e_\ell(1,n) \geq 0 \\ \min_{\{k:\ e_\ell(k+1,n) \geq 0\}} \{\mathtt{vopt}(\ell[k]) + e_\ell(k+1,n)^2\} & \text{if } e_\ell(1,n) < 0 \end{cases} \qquad (1)$$

Note that the number of different indices k which can potentially satisfy $e_\ell(k+1, n) \geq 0$ is bounded above by $(L+1)/2$.

A similar recurrence can also be defined for $\mathtt{opt}(\ell)$, the optimal list of breakpoints for ℓ.

Your first task for the coursework will be to design a dynamic programming algorithm of complexity $\Theta(n^2)$ or $\Theta(n \cdot L)$ based on (1). Your algorithm should take a list ℓ of word lengths, and should compute the value of $\mathtt{vopt}(\ell)$ (and its associated subproblems) and the breakpoint sequence $\mathtt{opt}(\ell)$ which realises this value. Your algorithm may be described in pseudocode, or alternatively just in paragraphs (in sufficient detail), but you should make sure you cover all the following points (the marks below describe the breakdown of the 15 marks):

- Describe the collection of *subproblems* of $\mathtt{vopt}(\ell)$, $\mathtt{opt}(\ell)$ which you will solve (recall that a common feature of dynamic programming algorithms is the identification of a collection of related subproblems). *[2 marks]*

- Describe the tables used by your dynamic programming algorithm and explain what the entries of the tables represent. *[2 marks]*
- Describe the order (either using pseudocode, or giving a detailed explanation in your own words) in which your algorithm builds/updates your dynamic programming tables, and eventually computes the solution. *[5 marks]*
- Give details of how the actual breakpoint sequence $\mathtt{opt}(\ell)$ itself is computed, in addition to the value $\mathtt{vopt}(\ell)$. *[2 marks]*
- Depending on some details of your algorithm (and the particular tables used by your algorithm), the complexity will either be $\Theta(n \cdot L)$ or $\Theta(n^2)$. Justify the running time of your algorithm in detail. *[4 marks]*

Your solution should be submitted as the first (main) section of a file named `dp.txt`, `dp.tex` or `dp.pdf`.

3 Algorithm for ss^* (5 marks)

In the Introduction to this coursework, and in Definition 2 of Section 2, we gave details of an alternative measure of quality for a proposed line-formatting, where the square of the trailing space of the final line is omitted.

In this part of the coursework, you must give an algorithm which accepts a list of ℓ of word lengths, and computes the value of $\mathtt{vopt}^*(\ell)$, and the breakpoint sequence $\mathtt{opt}^*(\ell)$ which realises this value.

Your algorithm is very likely to use the algorithm of Section 2 as a subroutine, and will probably run in identical asymptotic time to that algorithm.

Your solution should be submitted as the section section of a file named `dp.txt`, `dp.tex` or `dp.pdf`.

4 Dynamic programming implementations for ss and ss^* (10 marks)

After you have developed algorithms to correctly compute $\mathtt{opt}(\ell)$ (Section 2) and $\mathtt{opt}^*(\ell)$ (Section 3), the next stage is to implement these algorithms within the template file `LineFormat.java`, as methods of the following types:

```
public static int[] dynamicLFall (int[] wordLengths)
public static int[] dynamicLFallbutlast (int[] wordLengths)
```

The algorithm for `opt` should be implemented as `dynamicLFall`, and the algorithm for `opt`* implemented as `dynamicLFallbutlast`.

5 Experimental comparisons (10 marks)

In this final section, you should run experiments comparing the performance of your three implementations, and write a short report describing your results.

You should consider a collection of randomly generated sequences of word lengths, and compare the outputs generated by your different algorithms. Some the results of interest would be:

- The typical/average/max difference between $\mathtt{ss}(\ell;\ \mathbf{i})$ when $\mathbf{i}$ is computed by `greedyLF`, in comparison to when it is computed (for the same ℓ) by your dynamic programming method `dynamicLFall`.
- The typical/average/max difference between $\mathtt{ss}^*(\ell;\ \mathbf{i})$ when $\mathbf{i}$ is computed by `greedyLF`, in comparison to when it is computed (for the same ℓ) by your dynamic programming method `dynamicLFallbutlast`.
- The frequency with which $\mathtt{opt}(\ell)$ describes a different breakpoint sequence to $\mathtt{opt}^*(\ell)$.

To set up your experiments, you should define a sensible random method which generates sequences of word lengths which model the English language well (with word lengths between 2 and 7 being common, lengths 1 and 8 being fairly common, and lengths larger than 8 less common). You should base your experiments on reasonably-long sequences of words (which will require 10 or more lines of length 80).

You are welcome to carry out other experimental comparisons and report on those. Your results should be described in a a file named `experiments.txt`, `experiments.tex` or `experiments.pdf`.

6 Your tasks

The first thing you should do is download the file `LineFormat.java` from the course webpage. This file is the starting point for your implementation. It includes declarations for the methods you are required to write, as listed below. It also includes declarations for some methods that *might* be useful (though not required) if implemented, and a few implementations of simple file reading/writing methods.

1. Write a method which implements the $\Theta(n)$ greedy algorithm described in Section 1. It will take as input an array of integers (representing the lengths of the words, in sequential order), and return an array of integers representing the breakpoint sequence of the greedy line format for that sequence of words. *[10 marks]*

   ```
   public static int[] greedyLF(int[] wordLengths)
   ```

2. Describe in detail a $\Theta(n{\cdot}L)$ or $\Theta(n^2)$ dynamic programming algorithm to compute $\mathtt{opt}(\ell)$ (and $v\mathtt{opt}(\ell)$), as discussed in Section 2. *[15 marks]*

   The solution to this part of the coursework should be submitted either in `.txt`, `.tex` or `pdf`, as the first section of a file named `dp` (either `dp.txt`, `dp.pdf` or `dp.tex`).

3. Describe a $\Theta(n \cdot L)$ or $\Theta(n^2)$ algorithm to compute $\mathtt{opt}^*(\ell)$ (and $v\mathtt{opt}^*(\ell)$), as discussed in Section 3. *[5 marks]*

   The solution to this part of the coursework should be included as the second section of your `dp` file (either `dp.txt`, `dp.pdf` or `dp.tex`).

4. Implement methods to realise the algorithms you developed for parts 3 and 4 of this coursework. Each of these methods take as input an array of integers (representing the lengths of the words, in sequential order), and returns an array of integers containing the breakpoint sequence of the line format computed for that sequence of words. The method `dynamicLFall` should implement your dynamic programming algorithm for `opt`, and the method `dynamicLFallbutlast` should compute the solution for `opt`* (when the final line does not contribute to the sum of squares). *[10 marks]*

   ```
   public static int[] dynamicLFall(int[] wordLengths)
   ```

   ```
   public static int[] dynamicLFallbutlast (int[] wordLengths)
   ```

5. Write a short report (1-2 pages) comparing *experimental performance* of your algorithms, as described in Section 5. *[10 marks]*

Please implement *all of your methods* within the `LineFormat.java` file provided. Write the theoretical details of your dynamic programming algorithms in a file called `dp.txt`, `dp.tex` or `dp.pdf`. Give details of your experiments in a file called `experiments.txt`, `experiments.tex` or `experiments.pdf`. Then tar the files together into one file called `work2.tar` and submit it as follows:

```
submit ads cw2 work2.tar
```

The **DEADLINE** is 4pm, Wednesday, 12th November, 2014.
It is a nice idea for you to include some of the data files of text that you used in your experiments in your `tar` file.

Warning: In submitting, I think it's best to `tar` your files together at the command line (but if you're not confident with tar, don't bother doing it). And please do "`more`" on your files to check that you have the right versions to hand (the rules are "what is marked, is what is submitted").

Mary Cryan, 23rd October 2014 (updated 26th)