CSE 417: Algorithms and Computational Complexity **Winter 2012**

Homework 5

Due Wednesday, 2/22/12

This week, instead of written problems, we have a programming assignment to practice dynamic programming. Supporting files and materials can be found at `http://www.cs.washington.edu/education/courses/cse417/12wi/homework/homework5.html`. Here you will also find a link to turn in your code via Catalyst.

Problem (30 points)

In this problem, we'll explore how to break a paragraph of words into lines of text. If we're given a list of words, such as [“`It`”, “`was`”, “`the`”, “`best`”, “`of`”, “`times.`”], there are various ways to break it into lines of text, such as

```
It was the
best of
times.
```

or

```
It was
the best
of times.
```

Usually, each line can only fit a certain number of characters (where a character is a letter of the alphabet, a punctuation mark, or a space). Let L be the maximum number of characters that can fit on a line. So for example, if $L = 9$, then the second example above is a legal way to break up the text, but the first is not.

What is a good algorithm to decide where to put the line breaks? An approach that many text editors take is greedy: fit as many words as possible on the first line, then as many as possible on the second line, and so on until all the words are used. If this looks familiar, it's because it's just the greedy “CD packing” algorithm from Problem 1 of Homework 3, where the lines correspond to the CDs and the words correspond to the songs. Based on the result of that problem, we know that this algorithm minimizes the total number of lines used.

However, minimizing the total number of lines is not the only objective one might consider. Let the words that we need to format be w_1, w_2, …, w_n, and let the length of word i be denoted by $\ell(w_i)$. Suppose that words $i, i+1, \ldots, j$ form a line of text. The *length* of this line is $r(i,j) = (j-i) + \sum_{k=i}^{j} \ell(w_k)$, since the number of spaces needed to separate the $j-i+1$ words in the line is $j-i$ and the total length of those words is $\sum_{k=i}^{j} \ell(w_k)$. Define the *slack* of the line, $s(i,j) = L - r(i,j)$, to be the leftover space on the line.

Given these definitions, we might ask for an algorithm that minimizes the sum of the slacks for each line, except the last. Conveniently, it turns out the same greedy algorithm described above also minimizes this objective function. (Proving this would be a good exercise to check your understanding.) However, minimizing the sum of the slacks of each line is a bit simplistic. A more sophisticated objective is to minimize the sum of the *squares* of the slacks for each line, except the last. This would more heavily penalize large slacks and would therefore tend to reduce the "raggedness" of the paragraph. Variants of this type of objective function are used by many typesetting systems, including TEX, the software used to typeset this document.

With the sum of squared slacks objective, the greedy method fails. For example, consider the words ["aa", "bb", "cc", "dd", "eeeeeeee"] and a line length $L = 8$. The greedy method would output

```
aa bb cc
dd
eeeeeeee
```

which has a total penalty under the sum of squared slacks objective of $0^2 + 6^2 = 36$, whereas the optimal solution is

```
aa bb
cc dd
eeeeeeee
```

which has a total penalty of $3^2 + 3^2 = 18$.

Although the greedy algorithm fails for this objective, it turns out that there is an $O(n^2)$ dynamic programming algorithm that chooses line breaks to minimize the sum of squared slacks objective. In this problem, you are to implement such an algorithm in Java.

Formally, given the input $w_1, \ldots, w_n$ and a line length L such that $\ell(w_i) \leq L$ for all i, the algorithm is to output a set of *line breaks* $b_1, b_2, \ldots, b_m$. For all i, $1 \leq i \leq m$, b_i is a number between 1 and n that indicates that the i-th line begins at word b_i. To illustrate this concept, note that the second example above would have $b_1 = 1$, $b_2 = 3$, and $b_3 = 5$ since the first line starts with the first word, the the second line starts with the third word, and the third line starts with the fifth word. Note that since the first word must begin the first line, b_1 is always equal to 1.

These line breaks must be chosen so that no line is longer than L. That is, for $1 \leq i \leq m$, we must have $r(b_i, b_{i+1} - 1) \leq L$. (For convenience, we have implicitly defined $b_{m+1} = n + 1$.) Given this constraint, the algorithm must choose the line breaks to minimize the sum of squared slacks penalty function defined by

$$p = \sum_{i=1}^{m-1} \left[s(b_i, b_{i+1} - 1)\right]^2 \quad . \tag{1}$$

Note that this objective function does not take into account the slack of the last line since we don't usually care about its length.

The following notation will prove useful. For $1 \leq i \leq n$, let $q(i)$ be the index of the last word with which we can legally end a line starting with word w_i. That is, let

$$q(i) = \begin{cases} n & \text{if } r(i,n) \leq L \\ \max\{j \geq i : r(i,j) \leq L\} & \text{otherwise} \end{cases} .$$

To solve a problem via dynamic programming, we must come up with an ordered set of subproblems and a recurrence that relates those subproblems. We'll start you off with both of these. First, the subproblems: for $1 \leq i \leq n$, let $P(i)$ be the value, according to the objective p, of the optimal set of line breaks on the words $w_i, \ldots, w_n$. (Thus, $P(1)$ is the value of the optimal solution for our original problem.) Second, the recurrence relation:

$$P(i) = \begin{cases} 0 & \text{if } r(i,n) \leq L \\ \min_{i \leq j \leq q(i)} \left(s(i,j)^2 + P(j+1)\right) & \text{otherwise} \end{cases} . \tag{2}$$

This recurrence is based on choosing the best word with which to break the first line.

Your task: Build a Java program called `ParagraphFormatter` that can be run from the command line. It should take two parameters, `filename` and `length`. The `filename` parameter should point to a text file containing a paragraph of text in any format. The `length` parameter is the maximum allowable line length. Given these inputs, the program should print the paragraph, breaking lines to minimize the sum of squared slacks penalty function. For example, the following command would read a file called `tale_of_two_cities.txt` and output the words in that file with the optimal set of line breaks (given a maximum line length of 80), according the sum of squared slacks objective:

```
java ParagraphFormatter tale_of_two_cities.txt 80
```

To help, we've provided a set of resources on the course web page. These include:

- A file `ParagraphFormatter.java`. This contains supporting functions to parse the input and output the result. As the file is currently, it will output all of the words from the input file on a single line. Your task is to add code to this file to choose the optimal line breaks. Note that the provided code is structured so that you do not actually need to do any of the formatting yourself—you just need to output the list of line breaks $b_2, \ldots, b_m$ and then use the provided helper functions to turn that into text. (Since b_1 is always 1, you do not need to output it.) There are more instructions in the comments of the provided code.

  Although you will be implementing the heart of the algorithm, it should not actually be that much code. Our solution adds less than 30 lines of code to `ParagraphFormatter.java`.

- Two sample input files, `tale_in.txt` and `david_in.txt`. Both of these files consist of one paragraph, formatted according to the greedy line breaking strategy. For `tale_in.txt`, the maximum line length is 80 characters, and for `david_in.txt`, the maximum line length is 60 characters.

- Two corresponding output files, `tale_out.txt` and `david_out.txt`. These are the same paragraphs formatted according to the sum of squared slacks objective. (Note that they look less ragged!) The line breaks shown on these files should be the same line breaks you would see running your program on the sample input files. You can use these to test your program. (Of course, just because the program works correctly on two inputs doesn't mean that it is correct for all inputs.)

What you must turn in: Your `ParagraphFormatter.java` file, with the code implementing the dynamic programming algorithm added. This code should compile and run as specified. We will evaluate your solution in three ways:

- whether it runs correctly on the provided test instances,
- whether it runs correctly on our own test instances (not on the website), and
- whether you've demonstrated understanding of dynamic programming by the structure of the code.

The last criterion is more qualitative. It is a chance for partial credit if your code does not run correctly on the provided test instances. Because we will be looking at your code, it is important that it be clear and well-commented.

Extra Credit (10 pts)

The provided code actually supports a third, optional, command-line argument, which can be one of two values, "`last`" or "`decreasing`". The extra credit consists of implementing these options, both of which will involve modifying recurrence (2). If you choose to do the extra credit, just add the necessary code to the same `ParagraphFormatter.java` file that you turn in for the main part of the assignment. Follow the instructions given in the comments of the file. Your solutions will be evaluated solely on their ability to output the correct answer on our test instances. *Hint: The first extra credit problem is easier than the second.*

1. (5 pts) In the discussion above, our objective ignored the slack in the last line of the paragraph. However, this might not be what we want. Specifically, we might wish to minimize the objective
$$p' = \sum_{i=1}^{m} [s(b_i, b_{i+1} - 1)]^2 \quad .$$
instead of the objective (1). (The difference between the two objectives is the upper bound in the summation.) Add functionality to `ParagraphFormatter.java` to optimize this objective if the third parameter has the value "`last`". The file `tale_out_ec1.txt` included in the supporting materials can help you test your solution—it contains the target output of running

```
java ParagraphFormatter tale_in.txt 80 last
```

2. (5 pts) The allowed length of the lines might not be the same for all lines. This could be the case, for example, if we were formatting paragraphs to wrap around a figure. So, suppose that instead of being given a single line length L, we are given a line length L_k for each line k. Now, for an output to be legal, line k must have a total length less than L_k. The slack function is different as well. For the k-th line, the slack $s(i, j)$ is $L_k - r(i, j)$ instead of $L - r(i, j)$. Given the new slack function, the objective is the same as (1), that is, we still want to minimize the sum of the squared slacks over all but the last line.

   To make things concrete, suppose that $L_k = L - k + 1$, for some parameter L. (Thus, the allowed length of each line is one less than the preceding line: the first line can be length L, the second can be length $L - 1$, and so on.) Add functionality to `ParagraphFormatter.java` to optimize this objective for L_k defined as above if the third parameter has the value "`decreasing`". The file `tale_out_ec2.txt` included in the supporting materials can help you test your solution—it contains the target output of running

```
java ParagraphFormatter tale_in.txt 65 decreasing
```