CSCI-UA.0310-001/002 Basic Algorithms November 5, 2016

Text Alignment

Lecturer: Yevgeniy Dodis

Many modern text processors, such as LaTeX (which this text is typeset with), use a sophisticated dynamic-programming algorithm to ensure that lines are well-aligned on the right-hand side of a page. Clearly, the aesthetics of a typeset document depend on choosing good positions for line breaks.

The Text-Alignment Problem

An instance of the *text-alignment problem* is specified by positive integers $l_1, \ldots, l_n$, representing the lengths of the words in an n-word text, and L, corresponding to the maximum line length, as well as by a *penalty function* $P : \mathbb{N} \to \mathbb{R}$. The penalty function is used to characterize the *badness* of a line, i.e., how ill-aligned it is. More precisely, a line consisting of words $l_i, \ldots, l_j$ (with $i \leq j$ and $l_i + \ldots + l_j + j - i \leq L$) has a *gap* of

$$G := L - (l_i + \ldots + l_j + j - i)$$

and badness $P(G)$; the additional term $j - i$ accounts for the spaces between words. The objective is to insert an arbitrary number of line breaks between the n words such that

- there are no empty lines,
- no line has length more than L, and
- the sum of the lines' badnesses is minimized.

Greedy?

A natural inclination is to use a greedy algorithm: if a word fits, add it to the line. However, the following example shows that such a strategy does not work: Assume $P(G) = G^3$ and let $l_1 = 3$, $l_2 = 4$, $l_3 = 1$, $l_4 = 6$, and $L = 10$. The greedy algorithm puts the first three words on the first line and the fourth on the second, incurring penalties of $0^3 + 4^3 = 64$, whereas having two words per line results in a total badness of $2^3 + 2^3 = 16$, which is considerably lower.

A Dynamic-Programming Solution

For $i = 0, 1, \ldots, n$, define $m[i]$ as the optimal badness for aligning $l_1, \ldots, l_i$. Set $m[0] = 0$ and, for values of i such that $l_1 + \ldots + l_i + i - 1 \leq L$, set

$$m[i] = P(L - l_1 + \ldots + l_i + i - 1).$$

For larger values of i, set

$$m[i] = \min_k \big(m[k-1] + P(L - l_k + \ldots + l_i + i - k)\big)$$

recursively, where k ranges over all values k such that $l_k + \ldots + l_i + i - k \leq L$. The intuition behind the recurrence is that one finds the best choice k for starting the last line, adding $m[k-1]$ for the optimal badness of aligning $l_1, \ldots, l_{k-1}$ and the penalty of the last line with words $l_k, \ldots, l_i$. When computing $m[i]$, storing said value k in an auxiliary array $s[\cdot]$ as $s[i] = k$ allows for fast solution recovery once $m[n]$ is computed: The last line contains words $l_{s[n]}$ to l_n; the penultimate line those from $l_{s[s[n]-1]}$ to $l_{s[n]-1}$; etc.

Clearly, array $m[\cdot]$ can be filled in $O(un)$ time, where u is the maximum number of values k that need to be considered when computing an entry $m[i]$. Clearly, $u \leq L$. Under the reasonable assumption that $L = O(1)$, i.e., if L is independent of n, filling $m[\cdot]$ takes *linear* time. Reconstructing a solution from the filled array is easily seen to take at most n steps.

Not counting the final line. It is uncustomary to consider a penalty for the last line as is done by the algorithm above. To that end, consider the following modification: For $i = 0, 1, \ldots, n-1$, compute $m[i]$ as shown above. Then, compute the last entry according to

$$m[n] = \min_k m[k-1],$$

where k again ranges over all values k such that $l_k + \ldots + l_n + n - k \leq L$.