TEXT ALIGNMENT

Many modern text processors, such as LaTeX(which this is written on) use a sophisticated dynamic programming algorithm to assure that the lines are well aligned on the right side of the page. The problem is where to break the text into lines. Let us $l_1, l_2, \ldots, l_n$ denote the lengths of the words of the text. If $l_i, \ldots, l_j$ are placed on a line we assume they take up space $l_i + \ldots + l_j + j - i$, the extra being the space between the words. (In the special case of a single word l_i the length is simply l_i.) Let L denote the total length of the line. (We'll assume: all $l_i \leq L$, we can never have more than space L on a line, and that words can never be cut.) A line with text $l_i, \ldots, l_j$ then has a *gap* $G = L - (l_i + \ldots + l_j + j - i)$ at the end of the line. We'd like, of course, for G to be zero but we can't always [1] get this. We are given a function $P(x)$ and a line with gap GAP incurs a penalty of $P(GAP)$. (We'll take $P(GAP) = GAP^3$ as an example. We'll say a line with gap G has *badness* $BAD := P(GAP) = GAP^3$. [2] The total badness $(TBAD)$ of a text is the sum of the badnesses of the lines. The object is to split the text into lines so as to minimize the total badnesses.

A natural inclination is to use a *greedy algorithm*: if it fits, put it in. Below is an example (the `|` represents the end of the line) where this does not work. The words have lengths $3, 4, 1, 6$ and the line length $L = 10$. The greedy algorithm would have $3, 4, 1$ on the first line and 6 on the second with gaps $0, 4$ respectively so total badness $0^3 + 4^3 = 64$. If instead we split $3, 4$ and $1, 6$ the gaps are $2, 2$ and the total badness is $2^3 + 2^3 = 16$, much better[3]

```
   NOW SING  |2  8        NOW SING A|0  0
   A MELODY  |2  8        MELODY    |4 64
 total badness  16                     64
```

[1] You might notice that the text you are reading (and most things written in LaTeXor other modern word processors) seems to be nearly perfectly aligned. But if you look very closely you'll see that the space between letters is not perfectly uniform. When LaTeXhas, say, 3 extra spaces at the end of a line it spreads the space out along the whole line by putting extra space between letters. Exactly how it does that is itself an interesting question, but one we do not persue.

[2] So badnesses go $0, 1, 8, 27, 64, 125, \ldots$. A gap of five (badness 125) is then counted as equivalent to nearly five gaps of three so the algorithm will really try to avoid them. The choice of badness function is a subjective decision guided by aesthetic considerations. The algorithm is given a particular badness function and a text to split into sentences.

[3] In actual application space on the last line is not so bad as space in the middle but we ignore that wrinkle in our presentation.

Now for the algorithm. Set $TBAD(i)$ equal to the total badness of the text $l_i, \ldots, l_n$. We shall find $TBAD(i)$ for $i = n, n-1, \ldots, 1$ in *decreasing* order.

`Initialization.` Suppose i is such that $l_i, \ldots, l_n$ fit on one line, i.e., such that $l_i + \ldots + l_n + n - i \leq L$. The best splitting of the text is then to have everything on the same line. This gives a gap $GAP = L - (l_i + \ldots + l_n + n - i)$ and so $TBAD(i) = P(GAP)$.

`The recursion.` Now look at smaller i. We take the i one by one, going down. For a given i let k range over $i, i+1, i+2, \ldots$ for as long as $l_i, \ldots, l_k$ can fit on one line. *Suppose* the first line for text $l_i, \ldots, l_n$ was $l_i, \ldots, l_k$. This line would have $BAD = P(GAP)$ with GAP being the gap of that line. The remaining text $l_{k+1}, \ldots, l_n$ will have minimal total badness $TBAD[k+1]$ *which we have already calculated!* For each of those k calculate $BAD + TBAD[k+1]$. Pick the k that gives the *smallest* sum. Set $TBAD(i)$ equal to that minimal sum.

When i reaches 1 we find $TBAD(1)$ which is the the total badness of of the full text.

We can further find the actual splitting into lines using an auxilliary function $END[i]$, $1 \leq i \leq n$. $END[i]$ will be that k so that in the optimal parsing of $l_i, \ldots, l_n$ the first line is $l_i, \ldots, l_k$. We take the i in descending order. For those i for which $l_i, \ldots, l_n$ fit on one line we set $END[i] = n$. For other i we find the k that minimizes $BAD + TBAD[k+1]$ and set $k = END[i]$. We print out with a simple program:

$i = 1$
WHILE $i \leq k$
    PRINT $l_i, \ldots, l_{END[i]}$ *on one line*
    $i \leftarrow END[i] + 1$
    ENDWHILE

How long does this take. There is a loop on i of length n. There is an inner loop on k. Certainly k takes on at most n values so that this is a $O(n^2)$ algorithm. But in many cases we can say more. Suppose u is the maximum number of words that can fit on a line. Then k takes on at most u values so that this is a $O(un)$ algorithm. If we think of the line size as fixed (rather a natural assumption) and the words as having a minimal length (also natural) then u is a fixed number and so this becomes a *linear* algorithm in the size of the text.