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 $l_i + \ldots + l_j + j - i - 1$, 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* $G = L - (l_i + \ldots + l_j + j - i + 1)$ at the end of the line. We'd like, of course, for G to be zero but we can't always get this. We are given a function $P(x)$ and a line with gap G incurs a penalty of $P(G)$. For the sake of argument we will specify $P(x) = x^3$. We'll say a line with gap G has *badness* $P(G) = G^3$. [1] The total badness 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. Here is an example where this does not work. Let the words have lengths $3, 4, 1, 6$ with $L = 10$. 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.

Now for the algorithm. Set $m(i)$ equal to the total badness of the text $l_1, \ldots, l_i$. We initialize be $m(0) = 0$. Further initialization (not technically necessary): as long as $l_1, \ldots, l_i$ fit on one line set $m(i)$ to be the badness of that line. We loop on i going up to n. For a given i let k range over $i, i-1, i-2, \ldots$ for as long as $l_k, \ldots, l_i$ can fit on one line. For each of those k calculate $m(k-1)$ plus the badness of the line $l_k, \ldots, l_i$. Pick the k that gives the smallest sum and set $m(i)$ equal to that sum. We can also keep an auxilliary array s setting $s(i) = k$. This has the meaning that in the optimal splitting of text $l_1, \ldots, l_i$ the last line starts with l_k. At the end $m(n)$ denotes the total badness of of the full text. To actually do the splitting we work backwards. The last line goes from word $s(n)$ to word n. Now set $n \leftarrow s(n) - 1$ and the penultimate line goes from word $s(n)$ to word n, etc.

[1] 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.

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.

One final point. Most people don't like to count the badness of the final line. Lets make a modification that takes this into account. First calculate the $m(i)$ as above. Then for $k = n, n-1, \ldots$ as long as $l_k, \ldots, l_n$ can lie on a single line take $m(k-1)$. (I.e., don't add the badness of $l_k, \ldots, l_n$. Pick k with the minimal $m(k)$ and set $s(n) = k$ and make the text of $l_1, \ldots, l_{k-1}$ with badness $m(k)$ followed by the "free" last line $l_k, \ldots, l_n$.