Use algorithms to break up paragraphs

How to devise a dynamic algorithm to break a paragraph into neatly arranged lines of text

In this issue...

▸ WHAT'S COVERED

We see paragraphs of text all over the place: in books, in newspapers, on the web and in this very article. Some of them have a ragged right edge, others are justified, but in every case we're very aware when the breaking of the lines looks awkward and there are big blocks of white space. How can you break the lines in a paragraph to produce a more elegant look?

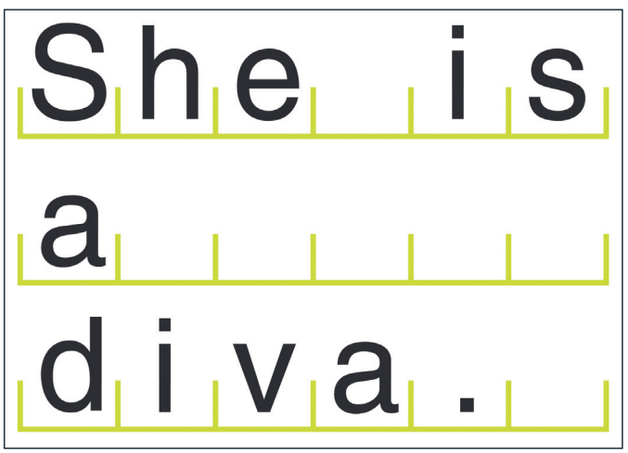
Breaking a paragraph up into lines so that it can be displayed or printed seems to require a fairly simple algorithm. Indeed, the traditional one everyone applies to this problem does a good job if all the words are roughly the same length, but it doesn't do so well if the words vary widely in size.

The traditional algorithm is an example of a 'greedy' algorithm which we'll look at first, then we'll look at a better algorithm which uses dynamic programming.

Making space

With a greedy algorithm, for each new word in our text, we check to see if it can be added to the current line. If it can, we add it, preceding it with a space. If it can't, we start a new line and add the word to it. We continue until all the words in the paragraph have been allocated to the lines.

We'll assume that the text is being displayed in a monospace font, just like the IDE editors we use, and that all lines are the same length in characters. We'll further assume that all words are less than the line length.



▲ **A greedy algorithm is local in scope and doesn't take into account the overall look of the paragraph when it breaks up the lines.**

The reason this is known as a greedy algorithm is that a line will suck up as many words as it can, with no regard for what's gone before or might come after. The algorithm is local in scope: there's no attempt to balance the words to give a better overall look.

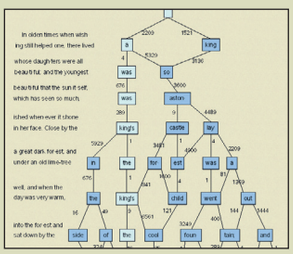
The dynamic programming algorithm will let us justify the text in a paragraph as well. Dynamic programming can be defined as a technique for solving a recursive problem by caching solutions to sub-problems rather than recomputing them. A simple example of this uses the recursive definition of Fibonacci numbers. The nth Fibonacci number is the sum of the (n-1)th and the (n-2)th Fibonacci numbers, with the added assumption that the first and second Fibonacci numbers are both 1. If we directly converted that definition into code, to calculate the sixth Fibonacci number, for example, we would have to calculate the fifth and the fourth and add them together. To calculate the fifth Fibonacci number we'd have to calculate the fourth and the third and add them together. This algorithm would be constantly calculating the same low-order Fibonacci numbers.

A dynamic programming approach to the same problem would cache the Fibonacci numbers as they're calculated. You would, in essence, save a lot of processing at the expense of some extra space.

What we want to do with our justification algorithm is to make

Spotlight on... Knuth-Plass algorithm

The most well-known and feature complete algorithm for breaking paragraphs into lines was devised for the TeX typesetting system by Donald Knuth and Michael Plass, and published in 1981 (their paper is available in Knuth's *Digital Typography*). The Knuth-Plass system uses three attributes of the text. They are called penalty, glue and box. A box is an element (such as a character or word) that can't be subdivided and has a set width. Glue is the space joining two boxes and is assumed to be fairly elastic: it has a default width, but can be shrunk or squeezed. A penalty describes a possible point to break between two lines. This can be large (a bad place) or small (a good place). Using these measures, the Knuth-Plass algorithm proceeds much as the algorithm described in the main text does – although it has rather more data to process. ■



▲ **The Knuth-Plass algorithm works by defining parts of a sentence as box, penalty or glue.**

Nugget

Desktop publishing applications go one step further than word processing programs and attempt to automatically hyphenate words to give a cleaner look to justified paragraphs. To do this they make use of a hyphenation dictionary, which is a list of words with possible hyphenation points marked. This modifies the algorithm pretty drastically, since the hyphen character has a width that must be accounted for when it's used, but is invisible (has width 0) when it's not. ■

the space at the end of each line (the tail space) as small as possible, or to somehow devise an algorithm that prizes short tail spaces over long tail spaces. Obviously, the last line of the justified paragraph doesn't count in this algorithm, since the last line is usually shorter than the others. One way to think about it is to imagine the tail space as being a spring, pulling on the words, and we're trying to minimise the total spring tension over all the lines in the paragraph. To do this, we'll devise a cost function for the tail space in each line and try and minimise the total cost over all the lines. You can think of the greedy algorithm as minimising the cost on a line-by-line basis.

Another feature of dynamic programming solutions is that the problem should have an optimal substructure. This is a problem whose optimal solution also implies an optimal solution for smaller sub-problems. In other words, if we solve sub-problems optimally, this will help solve the main problem optimally as well.

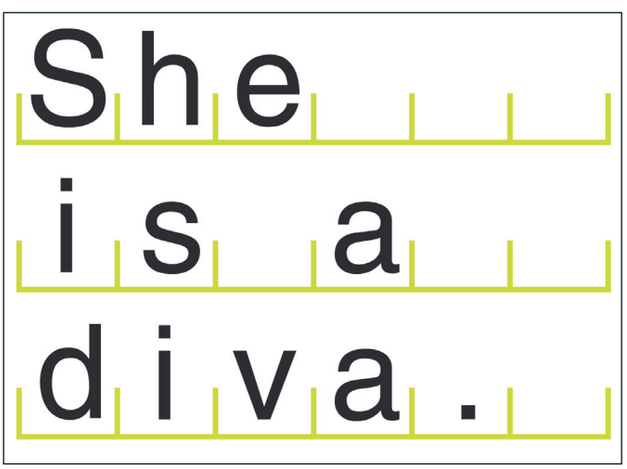
The cost function we'll use is to measure the size of blank space at the end of the line. To make it more significant we'll cube the value. That way, small spaces have low costs but large spaces have huge costs.

Line break

But what is a line and how many words should we put in each line? What we'll do is sequentially number all the words in the paragraph and then create a large matrix – remember, we've been talking about pre-computing data – where each cell (i, j) contains the cost from the ith to the jth word. That is, we put all the words from the ith to the jth on a line and calculate the cost (the remaining space) on the line. Notice that sometimes we can't fit all these words on a line (we're constrained by line length), so we make the cost a very large number (call it infinity) so it will never be used.

The first thing to notice is that the matrix is triangular; that is, i must be less than j. If the rows count i, then the lower half of the matrix would be unused. Furthermore, we assume that the last word on its own on the final line has a cost of zero, since the final line is allowed to be smaller without detriment.

As an example, 'She is a diva.' is the paragraph text and 6 is the line length. The greedy algorithm would break the words in an ugly fashion, as shown on the opposite page. Clearly, dropping the 'is' down a line would produce a more even right edge (see below).



▲ **A dynamic algorithm makes the space at the end of each line as small as possible, producing a more evenly justified paragraph.**

The cost matrix is as follows:

27	0	ffi	ffi
–	64	8	ffi
–	–	125	ffi
–	–	–	0

So, for example, the 27 entry is the cost of just having the word 'She' on a line by itself. Since the length of the tail space is 3, the cost is 3 cubed or 27. Similarly, the 8 entry is the cost of having the words 'is a' on a line by themselves, which is 2 cubed, 2 being the length of the tail space.

Now we have the matrix, how do we use it? We want to devise an algorithm for the cost of the optimal arrangement for the first word to the jth word, as we increase j from 1 to the number of words in the text. We'll call this the best cost for the word. Essentially, we'll calculate the best cost for optimally arranging the first word, then for the first two words, the first three words and so on. There are 2n-1 different ways to arrange the first n words in a set of lines, which could get awfully large pretty quickly. Luckily, we're going to make things easier for ourselves.

Keeping track

We'll use the previously calculated best costs to calculate the best cost for the word we're considering. Suppose this word is the jth word. Arrange the first to the jth word on one line. Calculate its line cost. Arrange the second to the jth word on one line, calculate its line cost and add in the best cost for the first word. Then arrange the third to the jth word on one line. Calculate its line cost and add in the best cost for the second word. Continue like this until you reach the jth word and keep track of the minimum cost. At the end, this will be our best cost. We're trying to put the j words into a justified block, by adding a line break before each of the previous words. The line break will produce an optimally justified block (with its best cost) and a final line (with its line cost). By finding the minimum total cost over all previous words we'll get the best cost for the jth word.

```
We choose to go to the moon. We
choose to go to the moon in this
decade and do the other things, not
because they are easy, but because
they are hard, because that goal will
serve to organize and measure the
best of our energies and skills,
because that challenge is one that we
are willing to accept, one we are
unwilling to postpone, and one which
we intend to win, and the others,
too.

We choose to go to the moon. We
choose to go to the moon in this
decade and do the other things, not
because they are easy, but because
they are hard, because that goal
will serve to organize and measure
the best of our energies and skills,
because that challenge is one that
we are willing to accept, one we are
unwilling to postpone, and one which
we intend to win, and the others,
too.
```

▲ **JFK's famous speech is broken up by a greedy algorithm (top) and a dynamic algorithm (bottom).**

The best cost for optimally arranging the first word on its own is 27. Looking at the first two words, we can either put them on the same line (cost is 0) or on separate lines (cost is 91). The best cost for the first two words is 0. Taking the first three words, we can put them all on the same line (cost is infinity), or use the optimal arrangement for the first word and the next two on their own line (total cost is 35), or use the optimal arrangement for the first two words and put the third word on its own line (cost is 125). The best cost for these is 35.

Finally, we add the fourth and last word to the mix. It turns out that the only way to arrange the fourth word without incurring an infinity cost is to have it on its own on a line. So, we use the optimal arrangement for the previous three words, which is the first word on the first line, and the next two together on the next line. The best cost is 35 and the paragraph will be displayed as shown in the picture on the left of this page.

In comparison, the image above shows part of the speech by John F Kennedy announcing the race to the moon. It's broken into lines, first by the greedy algorithm and then by the dynamic algorithm. ■

Julian M Bucknall is the CTO for Developer Express.
feedback@pcplus.co.uk