

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.

The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

The extra spaces includes spaces put at the end of every line except the last one. The problem is to minimize the following total cost.

Cost of a line = (Number of extra spaces in the line)<sup>3</sup>

Total Cost = Sum of costs for all lines

For example, consider the following string and line width  $M = 15$

"Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines

Geeks for Geeks

presents word

wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively. So optimal value of total cost is  $0 + 2^2 + 3^3 = 13$

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

**1)** There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces =  $3 + 0 + 0 = 3$ . Total cost =  $3^3 + 0^3 + 0^3 = 27$ .

**2)** There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces =  $1 + 1 + 1 = 3$ . Total cost =  $1^3 + 1^3 + 1^3 = 3$ .

Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.

### Method 1 (Greedy Solution)

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "aaa bb cc dddd" and line width as 6. Greedy method will produce following output.

aaa bb  
cc  
dddd

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is  $0 + 64 + 1 = 65$ .

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

aaa  
bb cc

```

dddd

```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is  $27 + 1 + 1 = 29$ .

Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

### Method 2 (Dynamic Programming)

The following Dynamic approach strictly follows the algorithm given in solution of Cormen book. First we compute costs of all possible lines in a 2D table  $lc[][]$ . The value  $lc[i][j]$  indicates the cost to put words from  $i$  to  $j$  in a single line where  $i$  and  $j$  are indexes of words in the input sequences. If a sequence of words from  $i$  to  $j$  cannot fit in a single line, then  $lc[i][j]$  is considered infinite (to avoid it from being a part of the solution). Once we have the  $lc[][]$  table constructed, we can calculate total cost using following recursive formula. In the following formula,  $C[j]$  is the optimized total cost for arranging words from 1 to  $j$ .

$$c[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \leq i \leq j} (c[i-1] + lc[i, j]) & \text{if } j > 0. \end{cases}$$

The above recursion has **overlapping subproblem property**. For example, the solution of subproblem  $c(2)$  is used by  $c(3)$ ,  $C(4)$  and so on. So Dynamic Programming is used to store the results of subproblems. The array  $c[]$  can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel  $p$  array that points to where each  $c$  value came from. The last line starts at word  $p[n]$  and goes through word  $n$ . The previous line starts at word  $p[p[n]]$  and goes through word  $p[n] - 1$ , etc. The function `printSolution()` uses  $p[]$  to print the solution.

In the below program, input is an array  $l[]$  that represents lengths of words in a sequence. The value  $l[i]$  indicates length of the  $i$ th word ( $i$  starts from 1) in the input sequence.

```

// A Dynamic programming solution for Word Wrap Problem
#include <limits.h>
#include <stdio.h>
#define INF INT_MAX

// A utility function to print the solution
int printSolution (int p[], int n);

// l[] represents lengths of different words in input sequence. For example,
// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc dddd". n is size of
// l[] and M is line width (maximum no. of characters that can fit in a line)
void solveWordWrap (int l[], int n, int M)
{
    // For simplicity, 1 extra space is used in all below arrays

    // extras[i][j] will have number of extra spaces if words from i
    // to j are put in a single line
    int extras[n+1][n+1];

    // lc[i][j] will have cost of a line which has words from
    // i to j
    int lc[n+1][n+1];

    // c[i] will have total cost of optimal arrangement of words
    // from 1 to i
    int c[n+1];

    // p[] is used to print the solution.
    int p[n+1];

    int i, j;

    // calculate extra spaces in a single line. The value extra[i][j]

```

```

// indicates extra spaces if words from word number i to j are
// placed in a single line
for (i = 1; i <= n; i++)
{
    extras[i][i] = M - l[i-1];
    for (j = i+1; j <= n; j++)
        extras[i][j] = extras[i][j-1] - l[j-1] - 1;
}

// Calculate line cost corresponding to the above calculated extra
// spaces. The value lc[i][j] indicates cost of putting words from
// word number i to j in a single line
for (i = 1; i <= n; i++)
{
    for (j = i; j <= n; j++)
    {
        if (extras[i][j] < 0)
            lc[i][j] = INF;
        else if (j == n && extras[i][j] >= 0)
            lc[i][j] = 0;
        else
            lc[i][j] = extras[i][j]*extras[i][j];
    }
}

// Calculate minimum cost and find minimum cost arrangement.
// The value c[j] indicates optimized cost to arrange words
// from word number 1 to j.
c[0] = 0;
for (j = 1; j <= n; j++)
{
    c[j] = INF;
    for (i = 1; i <= j; i++)
    {
        if (c[i-1] != INF && lc[i][j] != INF && (c[i-1] + lc[i][j] < c[j]))
        {
            c[j] = c[i-1] + lc[i][j];
            p[j] = i;
        }
    }
}

printSolution(p, n);
}

```

```

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

```

```

// Driver program to test above functions
int main()
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}

```

Output:

```
Line number 1: From word no. 1 to 1  
Line number 2: From word no. 2 to 3  
Line number 3: From word no. 4 to 4
```

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(n^2)$  The auxiliary space used in the above program can be optimized to  $O(n)$  (See the reference 2 for details)

**References:**

[http://en.wikipedia.org/wiki/Word\\_wrap](http://en.wikipedia.org/wiki/Word_wrap)