1. Problem overview

This problem is about finding an optimal solution for word warping/text justification. There are few rules to follow.

1. Text should not go beyond the margin
2. No of spaces at the end of the line should be minimum(spaces at the end of a line identifies as **Slack** of a line)

Following diagrams illustrate the concerns that we need to consider when developing the solutions

000 0000 0000 0000
0000 0000 000 000 00
00 000 0000 000 00
000 000 000 000 00000

figure 1

According to the figure 1, when wrapping the texts we should not go beyond the margin. If line margin exceeds, the last word need to be in next line

000 00000 000
000 000 000 000000
000 000 00000 00
000 000 000000

figure 2

According to the figure 2, we can see the slack(spaces at the end of the line) in line 1, line 3 and line 4 is too high, we should eliminate these type of scenario. Our goal is to minimize the spaces at the end of the line. So our optimal solutions should satisfy the above give two criterias.

To minimized/optimize the slack we are using some penalty kind of mechanism. Give a penalty for each space at the end of the line. In real scenario we cube the penalty. For an instance if there is 2 spaces at the end, the penalty is 2*2*2, if 5 spaces the penalty will be 5*5*5. The purpose of cube the penalty is reducing the spaces between words(idea is to highly discouraging the big spaces). Our goal is to wrap the text with minimum no of penalties.

2. How to find an optimal solution

Word wrapping is an optimization problem. Depending on what needs to be optimized for, different algorithms are used. Basically there are two approaches(ways) that can be use to solve this problem.

1. Greedy approach
2. Dynamic programming approach

2.1 Greedy approach

A simple way to do word wrapping is to use a greedy algorithm that puts as many words on a line as possible. In greedy strategy, we have to find a greedy choice. In here obvious greedy choice will be adding maximum no of words per line. This algorithm always uses the minimum possible number of lines but may lead to lines of widely varying lengths.

For example consider text "aaa bb cc ddddd" and line width as 6. Following is sample output when using greedy algorithm with this text.

aaa bb
cc
ddddd

figure 3

In this example, in first line we can put two words "aaa bb" since max line width is 6, in second line can only put one word "cc" if we put word "ddddd" in to second line, the line width will be exceed than 6. So word "ddddd" goes to third line.

This solution is not the optimal/ best solution. According to to the above output we can see greedy algorithm produces more extra spaces at the end of the line(slack in second line).

There are lot of applications that use greedy strategy for wrap/justify words in a text. **Microsoft word** and ***Openoffice word*** are some examples. The above mentioned problem(extrac spaces between words/ high amount of slack) exists in those application since they are using greedy algorithm.

2.2 Dynamic programming approach

Following arrangement has more balanced spaces than above arrangement (figure 3). To address the drawback in greedy approach we have to use dynamic programming approach.

aaa
bb cc
ddddd

figure 4

Following is the penalty criteria analyze of greedy and dynamic programming approaches for above example

Greedy solution (in figure 3)

Line 1	0 space	penalty 0*0*0 = 0
Line 2	4 spaces	penalty 4*4*4 = 64
Line 3	1 space	penalty 1*1*1 = 1

Total penalties = 0 + 64 + 1 (65)

Dynamic programming solution (in figure 4)

Line1	3 spaces	penalty 3*3*3 = 27
Line 2	1 spaces	penalty 1*1*1 = 1
LIne 3	1 space	penalty 1*1*1 = 1

Total penalties = 27 + 1 + 1 (29)

According to the above penalty comparison we can conclude that dynamic programming approach produce the minimum no of penalties(29). So dynamic programming is the technique we have to use when finding an optimal solution for text wrapping/ justification problem.

There are some applications which are using dynamic programming to text justification. One of the most popular one is **LaTex** document preparation system (LaTex defined the penalty for space as cubic function).

3. Dynamic programming properties

To apply a dynamic programming technique to solve a problem, two criteria should be satisfied. (Actually the problem should have two properties)

1. Optimal substructure
2. Overlapping of subproblems

The text justification problem have above two properties, that's why we can apply dynamic programming to this problem.

3.1 Optimal substructure

In dynamic programming the problem can divide into several problems. By finding the optimal solutions for given subproblems we can find the optimal solution for the whole problem.

3.2 Overlapping subproblems

In dynamic programming, the problem contains special property call overlapping subproblems. It means the solutions of same sub problem need again again(repetitively). So we can store the answers for the subproblems and reuse them in future(without solving the subproblem). Normally storing the answers of subproblems in a table. This storing and reusing technique known as **Memoization**. In memoization, when problem solved once the answer store in a memoization table. When the same subproblem comes again, the answer taken from the table instead of solving the problem.

This technique saves the total cost of the problem. So we can identify dynamic programming as a careful brute force technique(We have to solve each and every sub problem, but we solve some of them and reuse their answers in other subproblems)

4. Word wrapping with dynamic programming

Following is a brief description of the problem. It describe using standard notations. These notations are using when developing and analysing the solution.

Input (word list)	W {w1, w2, w3, w4, … , wn}
Margin (max line width)	M
Characters in a line	Ci (i line no)
No of penalties of a line	Slack
Output (line breaks)	L (w1, w2, w3 … wl1), (wl1 + 1, wl1 + 2, wl1 + 3, … , wl2), (wl2 + 1, wl2 + 2, wl2 + 3, … , wl3), … … … (wlx + 1, wlx + 2, wlx + 3, … , wn)

In this solution we give a list of input words and max line width to the program. These fields denotes with ***W*** and ***M*** respectively. Program output will be a list of line breaks. According to above description output(line breaks) would be like below.

1. First line starts at W[1] and end with W[l], second line starts with W[l1] + 1(l1 denotes line1). So after W[l1] there is a line break
2. Line 2 start with W[l2] + 1(l2 denotes line 2) end with W[l3]. So W[l3] is a line break
   …..
   …..
3. Last line start with W[lx] + 1(lx defines the last line, since we don't know how many lines are we define line no as x) and end with wn(n'th word)

We are denoting characters in a line with *C*i. i represents line no. For example *C1* (Characters in line 1), *C2* (Characters in line 2)

There are two conditions that need to be satisfied in the solution

1. No line exceeds the margin(max length of the lines).
   Ci <= M for all i
2. Should minimize the sum of slack in each line. We define slack as a cubic function of the spaces.
   min Σ(M - Ci) ^ 3

5. Recurrence relations

To solve the problem using dynamic programming first we have to define a variable that represent the solution. This variable can be use to create the recurrence relation of the problem. We defined a Variable **Best(n)**.

Best(n), defines the smallest penalty which the first n words can be typeset. In other words it defines what is the best way to typeset n words.

Before creating a recurrence relation first we further analyze the problem with **Best(n).** Figure 5 describes the problem with respect to **Best(n)**.

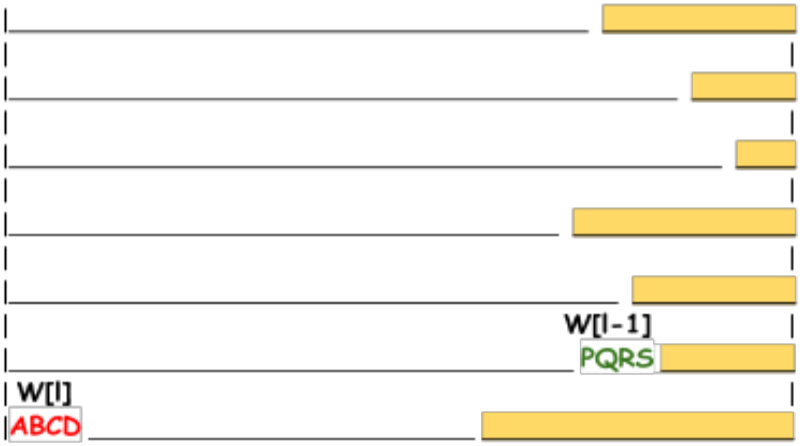

Figure 5

In here, each line contains some blank spaces(denote with color). It is the **slack** and our goal is to minimize the slack.

"ABCD" is the first word of the last line(denote it as **W[l]** - this is the l'th word). We can define our best solution in terms of the **W[l].** Depending on what is the first word of the last line we can define slack of the last line.(denotes with **S[l, n]**. According to **W[l]** and **S[l, n]** we can define the best way to typeset n words.

Best way to typeset n words is equals to best way to typeset l -1 words plus slack of last line. (**W[l - 1]** is the word before **W[l]**, so **W[l - 1]** is the last word of previous line). We can describe it as below.

Best (n) = Best (l - 1) + slack(l) ^ 3

So what are the candidates for l. l could be first word, l could be second word etc. So there are n candidates for l. According to this n candidates we can define the recurrence relation of the problem as below.

Recurrence relation

$$\text{Best}(n) = \begin{cases} \text{for } i=1 \text{ to } n \quad \min \{\text{Best}(i) + (S[(i+1), n])^{\wedge} 3\} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

In above relation **S[(i+1), n]** defines slack when typesetting the line that begin with word (i + 1) and ends with word n. Figure 6 further describe the above recurrence relation.

Figure 6

6. Implementation

In this dynamic programming solution we have to calculate some extra thing. It is the slack. So this problem is a one dimensional dynamic programming problem with auxiliary variable. There are two parts of the algorithm.

1. Create slack table
2. Find best line breaks with respect to slack

Following is the pseudocode of the algorithm. More details about this two phases of the algorithm described in next section

Pseudocode

```
make slack table S[i, j]                    <----------------- part (1)


for i -> 0 to n
      best(i) = for j -> 0 to i             <----------------- part (2)
                  min{best(j) + }
```

6.1 Slack table

First part of the algorithm is creating the slack table. Figure 7 shows the idea behind the slack. So S[i, j] denotes the slack when line starts with word i (W[i]) and ends with word j (W[j])

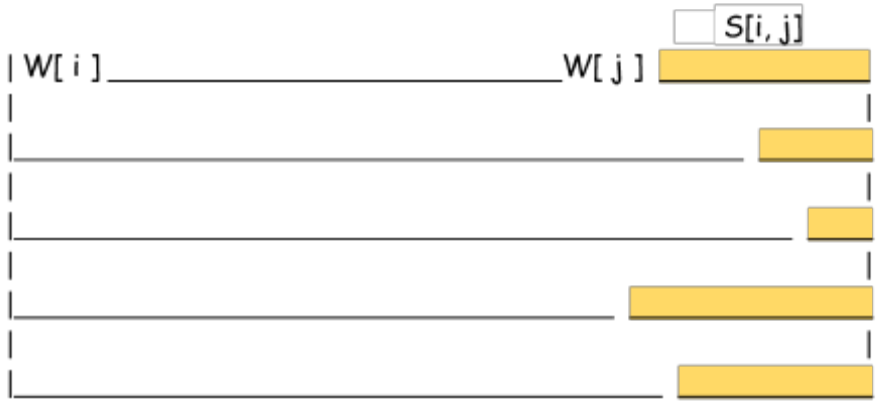

Figure 7

The simplest case of the slack is S(1, 1). It denotes slack when line start with word 1 and ends with word 1(only one word). Figure 8 illustrate it.

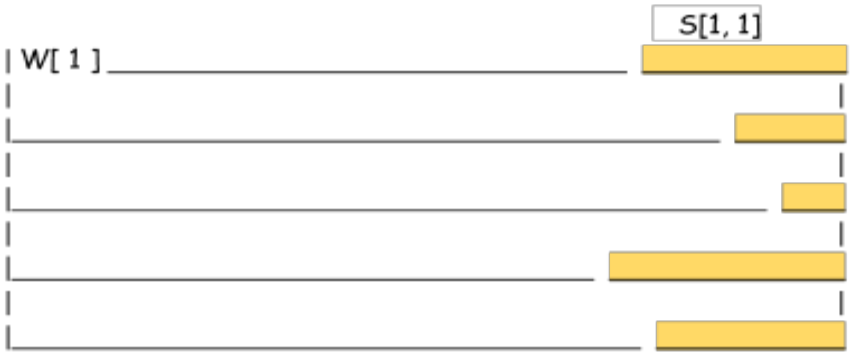

Figure 8

We calculate the slack values and put in a table for future use. Following are some examples of calculated slack values. Figure 9 illustrate how slack table arrange the slack values.

Slack(1, 1) = |M| - |W1|

Slack(1, 2) = Slack(1, 1) - 1 - |W2|

Slack(1, 3) = Slack(1, 2) - 1 - |W3|

Slack(2, 2) = |M| - |W2|

Slack(2, 3) = Slack(2, 2) - 1 - |W3|

S[1, 1]	S[1, 2]	S[1, 3]	S[1, 4]	---------	S[1, n]
	S[2, 2]	S[2, 3]	S[2, 4]	---------	S[2, n]
		S[3, 3]	S[3, 4]	---------	S[3, n]
			S[4, 4]	---------	S[4, n]

				---------	S[n, n]

Figure 9

6.2 Slack function implementation

To implement slack we use two dimensional array. Iterate through each and every word and find the slack. Every steps in the algorithm described in figure 11.

Pseudocode

$$S(i, j) = \begin{cases} |M| - |W_i| & \text{if } i = j \\ S(i, j-1) - 1 - |W_j| & \text{otherwise} \end{cases}$$

Implementation

```
/**
 * Initialize slack table according to the content
 * in input words.
 *
 *            |M| - |Wi|              if i = j
 * S(i, j) =
 *           S(i, j-1) - 1 - |Wj|    otherwise
 *
 * @param words input words
 * @param margin maximum line margin
 *
 * @return slack table(array)
 */
private static int[][] initSlack(String []words, int margin) {
    // slack table is two dimensional Array
    int [][]slack = new int[words.length + 1][words.length + 1];

    // initialize slack
    for (int i=1; i<=words.length; i++) {
        slack[i][i] = margin - words[i - 1].length();

        for (int j=i+1; j<=words.length; j++) {
            slack[i][j] = slack[i][j-1] - words[j - 1].length() - 1;
        }
    }

    return slack;
}
```

Figure 11

In this program "**initSlack()**" function take care of initializing the slack table with respect to input words and max line width. According to the above program, we can see that slack function produce an **n * n** array(n is the no of words in input text/paragraph).

Example

Input words - "**one could imagine some of these features being contextual**"

Output - Figure 12

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 12, 9, 3, -1, -4, -6, -11, -16, -18, -21, -25, -31]
[0, 0, 13, 7, 3, 0, -2, -7, -12, -14, -17, -21, -27]
[0, 0, 0, 10, 6, 3, 1, -4, -9, -11, -14, -18, -24]
[0, 0, 0, 0, 12, 9, 7, 2, -3, -5, -8, -12, -18]
[0, 0, 0, 0, 0, 13, 11, 6, 1, -1, -4, -8, -14]
[0, 0, 0, 0, 0, 0, 14, 9, 4, 2, -1, -5, -11]
[0, 0, 0, 0, 0, 0, 0, 11, 6, 4, 1, -3, -9]
[0, 0, 0, 0, 0, 0, 0, 0, 11, 9, 6, 2, -4]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 14, 11, 7, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13, 9, 3]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 6]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10]
```

Figure 12

6.3 Find best linebreaks(best(n))

The second part of the algorithm is finding the cost and line break positions according to the badness defined in slack table. Following is the pseudocode of the algorithm, algorithm implementation is in figure 15.

Pseudocode

$$
best(i) = \begin{cases} 0 & \text{if } i=0 \\ \min_{j = 0 \text{ to } i} \{Best(j) + (S[j, i])^{\wedge} 3\} & \text{otherwise} \end{cases}
$$

According to the pseudocode we can identify that this algorithm contains all the two properties that required in dynamic programming(Optimal substructure and overlapping subproblems)

Overlapping subproblems

We can represent this property by using a tree(represent recurrence relation in a tree). For an instance when need to find best(3) the structure of the problem will looks like below(Figure 13)

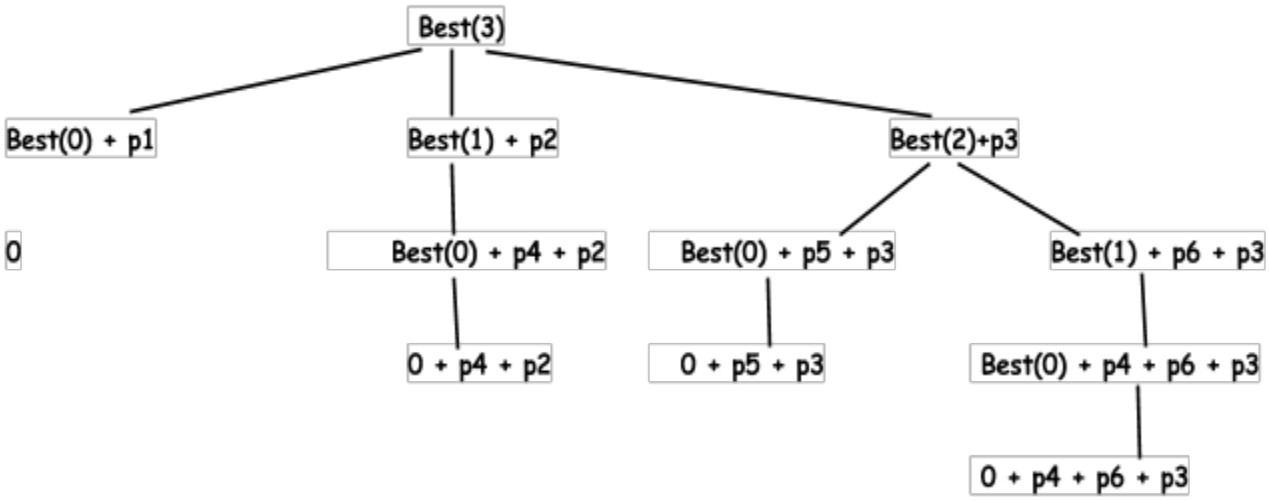

Figure 13

p1, p2 .. pi denote penalty value(S[i, j] ^3). For the simplicity we define it as pi

According to the Figure 13, following are the overlapping subproblems which are exists when calculating Best(3)

Best(0) overlapped in 4 places
Best(1) overlapped in 2 places

We are storing these subproblem answers and reuse them in future occurrences. This technique identified as **Memoization**

Optimal substructure

The second property is optimal substructure property. Following acyclic graphs shows the optimal substructure property of the solution.

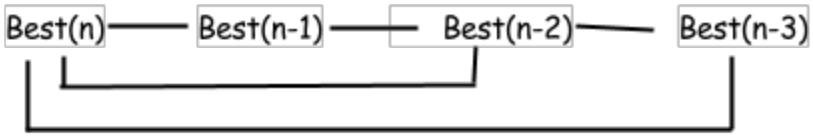

Figure 14

According to figure 13 we can identify that Optimal solution of Best(n) can be constructed with optimal solution of Best(n-1), Optimal solution of Best(n-2) can be constructed with Best(n-2) and so on. This is the optimal substructure property of this solution.

Implementation

The implementation of the algorithm is in figure 15. The function "**findBestLineBreaks()**" takes an input word count and slack table. According to the slack table content algorithm iteratively identifies best positions to take line breaks(which minimize the penalties).

If don't use memoization, algorithm will works as a brute force(Its a brute force strategy to find the line breaks). In brute force strategy we are going each and every word and find weather is it a start of a line. So there is 2 ^ n of sub problems there. Its algorithm would be like below

```
// n is input word count and s i slack
findBestLineBreaks(n, s)
        for i = 1 to n
                for j = 0 to i
                        min {findBestLineBreaks (j) + (S[j, i]) ^ 3}
```

Brute force approach, running time will be ***O*(2 ^ n).**

In here, instead of calculating each and every subproblem we storing the solved subproblem answers in "**bestValues"** array(Place we are doing memoization). This program contains n*n (n is no of words in input texts/paragraph) loop iterations. So it will cost O(n ^ 2)

```
/**
 * Find best possible line breaks(fist words of the line) for given n words
 * which minimizing the total badness(slack). We are using previously calculated
 * slack table values to find best solution
 *
 *           0                              if i = 0
 * best(i) =
 *           j = 0 -> i
 *               min{best(j) + S(j + 1, i)}   otherwise
 *
 * @param wordCount length of the words(this can identifies as n)
 * @param slack slack table
 *
 * @return line breaks
 */
private static int[] findBestLineBreaks(int wordCount, int [][]slack) {
    int []bestValues = new int[wordCount + 1];
    int []lineBreaks = new int[wordCount + 1];
    bestValues[0] = 0;

    for(int i=1; i<=wordCount; i++) {
        int min = INFINITY;
        int tmp;
        int choice = 0;

        // find min cost values, its is the best value
        for (int j=0; j<i; j++) {
            // we not allow negative costs,
            // negative costs considers as infinity
            if (slack[j + 1][i] < 0) {
                // ignore here
                tmp = INFINITY;
            } else if(j == wordCount - 1) {
                // last line cost is 0
                tmp = 0;
            } else {
                // rest of the line cost is "min{best(j) + S(j+1, i)}"
                tmp = bestValues[j] + ((slack[j + 1][i]) * (slack[j + 1][i]) * (slack[j + 1][i]));
            }

            // refine min value
            if (tmp < min) {
                min = tmp;
                choice = j;
            }
        }

        bestValues[i] = min;
        lineBreaks[i] = choice;
    }

    return lineBreaks;
}
```

Figure 15

6.4 Test data

Example 1

```
Input words : She is happy but is a blue gal. I am all gone.
Line margin : 15
Output:
She is happy
but is a blue
gal. I am all
gone.
Cost:
43
```

Example 2

```
Input words : aaa bb cc ddddd
Line margin : 6
Output:
aaa
bb cc
ddddd
Cost:
28
```

Example 3

```
Input words : Find the best line breaks
Line margin : 10
Output:
Find the
best line
breaks
Cost:
9
```

Running time analysis

There are two parts of the algorithm. Following are the running time complexities of these two parts.

Slack table function (part 1)

$$S(i, j) = \begin{cases} |M| - |W_i| & \text{if } i = j \\ S(i, j-1) - 1 - |W_j| & \text{otherwise} \end{cases}$$

According to the slack algorithm described in figure 11, it creates n*n table. So loop iterates over n*n times. Total complexity will be like below

No of loops iterations	= n * n
Total running time	= Θ(n ^ 2)

Best line break finding function (part 2)

$$Best(n) = \begin{cases} 0 & \text{if } n = 0 \\ \text{for } k=1 \text{ to } n \\ \quad \min \{Best(k) + (S(k+1), n)\ \hat{}\ 3\} & \text{otherwise} \end{cases}$$

According to the recurrence relation of the problem

No of runs of (1) loop	= n
no of sub problems(see section 3)	= n
Total running time	= Θ(n ^ 2)

Total complexity

Slack function + Line breaks function = $O(n^2) + O(n^2)$

$= 2O(n^2)$

$= \Theta(n^2)$