Word_wrap_editorial

Friday, November 12, 2021

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 ddddd" and line width as 6. Greedy method will produce following output.

aaa bb
cc
ddddd

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
ddddd

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.

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>

Method 2 (Recursive Approach with memoization)

The problem can be solved using a divide and conquer (recursive) approach. The algorithm for the same is mentioned below:

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>

```
1. We recur for each word starting with first word, and remaining
length of the line (initially k).
2. The last word would be the base case:
    We check if we can put it on same line:
        if yes, then we return cost as 0.
        if no, we return cost of current line based on its remaining
length.

3. For non-last words, we have to check if it can fit in the current
line:
    if yes, then we have two choices i.e. whether to put it in same
line or next line.

        if we put it on next line: cost1 = square(remLength) + cost of
putting word on next line.
        if we put it on same line: cost2 = cost of putting word on
same line.
        return min(cost1, cost2)
    if no, then we have to put it on next line:
        return cost of putting word on next line
```

4. We use memorization table of size n (number of words) * k (line length), to keep track of already visited positions.

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>

```
/*package whatever //do not write package name here */

import java.io.*;
import java.util.Arrays;

public class WordWrapDpMemo {



    private int solveWordWrap(int[] nums, int k) {
        int[][] memo = new int[nums.length][k + 1];
        for (int i = 0; i < nums.length; i++) {
            Arrays.fill(memo[i], -1);
        }
        return solveWordWrapUsingMemo(nums, nums.length, k, 0, k, memo);
    }



    private int solveWordWrap(int[] words, int n, int length, int wordIndex,
int remLength, int[][] memo) {

        //base case for last word
        if (wordIndex == n - 1) {
            memo[wordIndex][remLength] = words[wordIndex] < remLength ? 0 :
square(remLength);
            return memo[wordIndex][remLength];
        }

        int currWord = words[wordIndex];
        //if word can fit in the remaining line
        if (currWord < remLength) {
            return Math.min(
                    //if word is kept on same line
                    solveWordWrapUsingMemo(words, n, length, wordIndex + 1,
remLength == length ? remLength - currWord : remLength - currWord - 1, memo),
                    //if word is kept on next line
                    square(remLength) + solveWordWrapUsingMemo(words, n,
length, wordIndex + 1, length - currWord, memo)
            );
        } else {
            //if word is kept on next line
            return square(remLength) + solveWordWrapUsingMemo(words, n,
length, wordIndex + 1, length - currWord, memo);
        }
    }



    private int solveWordWrapUsingMemo(int[] words, int n, int length, int
wordIndex, int remLength, int[][] memo) {
        if (memo[wordIndex][remLength] != -1) {
            return memo[wordIndex][remLength];
        }

        memo[wordIndex][remLength] = solveWordWrap(words, n, length,
wordIndex, remLength, memo);
        return memo[wordIndex][remLength];
    }



    private int square(int n) {
        return n * n;
    }



    public static void main(String[] args) {
        System.out.println(new WordWrapDpMemo().solveWordWrap(new int[]{3, 2,
2, 5}, 6));
    }
}
```

Output

10

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>

Method 3 (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.

$$c[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} (c[i-1] + l_c[i, j]) & \text{if } j > 0 \end{cases}$$

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 ith word (i starts from 1) in theinput sequence.

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>

```
// A Dynamic programming solution for
```

```java
// Word Wrap Problem in Java
public class WordWrap
{

    final int MAX = Integer.MAX_VALUE;

    // A utility function to print the solution
    int printSolution (int p[], int n)
    {
        int k;
        if (p[n] == 1)
        k = 1;
        else
        k = printSolution (p, p[n]-1) + 1;
        System.out.println("Line number" + " " + k + ": " +
                    "From word no." +" "+ p[n] + " " + "to" + " " + n);
        return k;
    }



// l[] represents lengths of different words in input sequence.
// For example, l[] = {3, 2, 2, 5} is for a sentence like
// "aaa bb cc ddddd". 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[][] = new int[n+1][n+1];

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

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

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

        // 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 (int i = 1; i <= n; i++)
        {
            extras[i][i] = M - l[i-1];
            for (int 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 (int i = 1; i <= n; i++)
        {
            for (int j = i; j <= n; j++)
            {
                if (extras[i][j] < 0)
                    lc[i][j] = MAX;
                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 (int j = 1; j <= n; j++)
        {
            c[j] = MAX;
            for (int i = 1; i <= j; i++)
            {
                if (c[i-1] != MAX && lc[i][j] != MAX &&
                   (c[i-1] + lc[i][j] < c[j]))
                {
```

```
                    c[j] = c[i-1] + lc[i][j];
                    p[j] = i;
                }
            }
        }

        printSolution(p, n);
    }

    public static void main(String args[])
    {
        WordWrap w = new WordWrap();
        int l[] = {3, 2, 2, 5};
        int n = l.length;
        int M = 6;
        w.solveWordWrap (l, n, M);
    }
}

// This code is contributed by Saket Kumar
```

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 cane be optimized to O(n) (See the reference 2 for details)

From <https://www.geeksforgeeks.org/word-wrap-problem-dp-19/>