Word Wrap Problem

Difficulty Level Hard

Frequently asked in Arcesium Factset GreyOrange Microsoft Myntra Ola Cabs PayU

Tags Dynamic Programming

Table of Contents

Problem Statement

The word wrap problem states that given a sequence of words as input, we need to find the number of words that can be fitted in a single line at a time. So, for doing this we put breaks in the given sequence such that the printed document looks nice. The word editors such as Microsoft Office, Libre Office, and other document tools use line breaks to make the document look nice. Here, by nice we mean we place spaces in an evenly spread manner. There should not be lines with many extra spaces and some with small amounts.

Here, we also assume that our word length does not exceed the line size. Just to make things a bit more generic we are considering a const function in the question as (Number of extra spaces)^3, else it would have been too easy. This cost function may vary as per the question asked.

Greedy Approach
Wrong
Cost : 65

Dynamic Programming Approach
Correct
Cost : 28

Example

Here, we will provide number of words, word size, and line size as input.

Input

```
number of words = 4

wordSize = { 3, 2, 2, 3}

lineSize = 6
```

Output

```
28
```

Explanation: We can place 1st word on 1st line with 3 extra space, 2nd and 3rd word on 2nd line with 1 extra space, and the last word on 3rd line with no extra space (we won't consider the spaces after the last word as extra spaces). Thus, using our cost function we find cost as 28.

Input

```
number of words = 5

wordSize = { 1, 1, 1, 1, 1}

lineSize = 10
```

Output

```
36
```

Explanation: Here we can place all words on the same line i.e. 1st line and thus cost = 0.

Approach for word wrap problem

The first approach which comes to mind is a greedy solution where we simply keep on placing the words in a single line. When we cannot place a word on the same line, we move to the second line. This seems to work just fine, but there's a catch. This algorithm will not produce the optimal result because there may be cases such that if we change the extra spaces we may end up with a better global solution.

Output using Greedy Approach

Input

```
"cat is an animal", line size = 6
```

Output

```
65
```

Just for simple understanding, we have shown the input as words instead of wordSize.

Explanation: cat is_

an____

animal

Here, there are 4 extra spaces on the second line and 1 on the third line.

Calculation: 4^3+1^3 = 65

There exists a better solution,

cat___

is an_

animal

Giving us a total cost of 28 (3^3 + 1^3 + 0^3), which is better than 65.

Now, we will solve the word wrap problem using Dynamic Programming. We know that our problem needs a global optimum solution and our previous algorithm was trying to give us local optimum as a result. Here we will find the extra space taken in each line, and will thus find the cost for each row respectively. We will maintain a matrix extraSpace which will tell the extraSpace left in a line, of words from i to j are laced on the same line. Then further we will use this extraSpace matrix to find the minimum cost.

Code

C++ Code for word wrap problem

Code

```
#include <bits/stdc++.h>
using namespace std;

int wordWrap (int wordSize[], int n, int lineSize)
{
  int extraSpace[n+1][n+1];
  int minCost[n+1];

  for (int i=1;i<=n;i++)
  {
    extraSpace[i][i] = lineSize - wordSize[i-1];
    for (int j=i+1;j<=n;j++)
      extraSpace[i][j] = extraSpace[i][j-1] - wordSize[j-1] - 1;
  }

  minCost[0] = 0;
  for (int i = 1; i <= n; i++)
  {
    minCost[i] = INT_MAX;
    for (int j = 1; j <= i; j++)
    {
      int cost; // stores cost of putting words[i,j] in single line

      // if extraSpace contains -ve number, then we can't place
      //words[i,j] in a single line, else if we placed our last
      //word, then we don't consider extraSpace, else calculate
      //cost
      if(extraSpace[j][i]<0)cost = INT_MAX;
      else if(i==n && extraSpace[j][i]>=0)cost = 0;
      else cost = extraSpace[j][i]*extraSpace[j][i]*extraSpace[j][i];

      if (minCost[j-1] != INT_MAX && cost != INT_MAX
      && minCost[j-1] + cost < minCost[i])
        minCost[i] = minCost[j-1] + cost;
    }
  }

  return minCost[n];
}

int main()
{
  int t,n,lineSize;
  while(t--)
  {
    cin >> n >> lineSize;
    int wordSize[n];
    for(int i=0;i<n;i++)
      cin >> wordSize[i];
    int lineSize1=lineSize;
    cout<<wordWrap(wordSize, n, lineSize);
    cout<<endl;
  }
}
```

Input

```
1
4
3 2 2 3
6
1
1 1 1 1
10
5
1 1 1 1 1
```

Output

```
28
4
```

Java Code for word wrap problem

Code

```
import java.util.*;
import java.lang.*;
import java.io.*;

public class Main
{
  public int wordWrap (int wordSize[], int n, int lineSize)
  {
    int extraSpace[][] = new int[n+1][n+1];
    int minCost[] = new int[n+1];

    for (int i=1;i<=n;i++)
    {
      extraSpace[i][i] = lineSize - wordSize[i-1];
      for (int j=i+1;j<=n;j++)
        extraSpace[i][j] = extraSpace[i][j-1] - wordSize[j-1] - 1;
    }

    minCost[0] = 0;
    for (int i = 1; i <= n; i++)
    {
      minCost[i] = Integer.MAX_VALUE;
      for (int j = 1; j <= i; j++)
      {
        int cost; // stores cost of putting words[i,j] in single line

        //if extraSpace contains -ve number, then we can't place
        //words[i,j] in a single line, else if we placed our last
        //word, then we don't consider extraSpace, else calculate
        //cost
        if(extraSpace[j][i]<0)cost = Integer.MAX_VALUE;
        else if(i==n && extraSpace[j][i]>=0)cost = 0;
        else cost = extraSpace[j][i]*extraSpace[j][i]*extraSpace[j][i];

        if (minCost[j-1] != Integer.MAX_VALUE
        && minCost[j-1] + cost < minCost[i])
          minCost[i] = minCost[j-1] + cost;
      }
    }

    return minCost[n];
  }

  public static void main(String args[])
  {
    Scanner sc = new Scanner(System.in);

    int t = sc.nextInt();
    while(t--> 0 ) {
      int n = sc.nextInt();
      int wordSize[] = new int[n];
      for(int i=0;i<n;i++)
        wordSize[i] = sc.nextInt();

      int lineSize = sc.nextInt();
      int ans = new Main().wordWrap(wordSize, n, lineSize);
      System.out.println(ans);
    }
  }
}
```

Input

```
1
4
3 2 2 3
6
1
1 1 1 1
10
5
1 1 1 1 1
```

Output

```
28
4
```

Complexity Analysis

Time Complexity: O(n^2)

Here, since our outer loop runs from 1 to n and our inner loop runs from 1 to i, we have a polynomial time complexity of O(n^2).

Space Complexity: O(n^2)

Here, our extraSpace array is a 2D array which is of size N*N, which gives us polynomial space complexity of O(N^2).

Dynamic Programming Interview Questions
Arcesium, Dynamic Programming, Factset, GreyOrange, Hard, Microsoft, Myntra, Ola Cabs, PayU
‹ The Painter's Partition Problem
› Maximum weight transformation of a given string