Max Kipust

Nick Mulhern

Chengyi Ma

# Lab 4

## Summary report

We implemented our solution to the word wrap problem in Java. The problem wasn’t communicated clearly. When reading the assignment page it said to minimize the sum of the extra spaces at the end of the lines. This is trivial with a greedy algorithm approach, but when emailing the professor and the TA I found out that you actually want the minimum of the sum of the square of the extra spaces at the ends of the lines. This would require DP and was never stated on the assignment. If you continue to use this assignment in the future, please change the assignment paper to specify so.

## Algorithm

I used a Dynamic Programming approach to solve this problem. It was split up into 3 parts.

1. Part 1 calculates the extra spaces on a line if you were to include words i to j on that line.
2. Part 2 takes the results from part 1 and cleans them up, it’ll make them MAX if they are negative since that means that we can’t fit those words on the line, it’ll also return 0 for the last line since we want to ignore the last line, otherwise it’ll just square the number. We need to do this outside of part 1 since part 1 relies on itself. If we wrote part 1 recursively then we wouldn’t need to but we would need to recalculate each cell multiple times.
3. Part 3 finds the lowest combination that’ll cover all of the words
4. Part 4/getSolution() is a little bit at the end thatll convert our solution array into a more printable format.

public Integer[] solve(){  
 //calulate the cost of having words i to j in a line PART 1  
 Integer extraSpace[][] = new Integer[words.length][words.length];  
 for(int i = 0; i< words.length; i++){  
 extraSpace[i][i] = lineLength - words[i].length();  
 for(int j = i+1; j<words.length; j++){  
 extraSpace[i][j] = extraSpace[i][j-1] - words[j].length()-1;  
 }  
 }  
  
 //calculate the square of the cost of the line PART 2  
 for(int i =0; i< extraSpace.length; i++){  
 for(int j = i; j<extraSpace[i].length; j++){  
 extraSpace[i][j] = cleanSquareNum(extraSpace[i][j], j);  
 }  
 }  
  
 Integer totalCost[] = new Integer[words.length+1];  
 Integer solution[] = new Integer[words.length+1];  
 //find the minimum cost for puting words on each line PART 3  
 totalCost[0] = 0;  
 for (int j = 1; j <= words.length; j++) {  
 totalCost[j] = Integer.*MAX\_VALUE*;  
 for (int i = 1; i <= j; i++) {  
 if (totalCost[i-1] != Integer.*MAX\_VALUE* && extraSpace[i-1][j-1] != Integer.*MAX\_VALUE* && (totalCost[i-1] + extraSpace[i-1][j-1] < totalCost[j])){  
 totalCost[j] = totalCost[i-1] + extraSpace[i-1][j-1];  
 solution[j] = i;  
 }  
 }  
 }  
  
 getSolution(solution, words.length);  
 return lineNums;  
}

*/\*\*  
 \* calulates the cost of the line by squaring it  
 \** ***@param*** *num the remaining spaces in the line  
 \** ***@param*** *line the line number  
 \** ***@return*** *the cost of the line  
 \*/*

public Integer cleanSquareNum(int num, int line){  
 if(num < 0){  
 //if we cant fit the number on the line then we return the max  
 return Integer.*MAX\_VALUE*;  
 }  
 if(line == words.length-1){  
 //if the word is on the last line then we dont give it a cost since we can ignore the extra spaces on the line  
 return 0;  
 }  
 //otherwise square the number  
 return num\*num;  
}

*/\*\*  
 \* converts the solution to a list of line numbers.  
 \* It will set the actual solution to lineNums. this is done to save me a ton of pain.  
 \** ***@param*** *solution the solution we get from the dp solution  
 \** ***@param*** *size the size that we are looking at  
 \** ***@return*** *the new size  
 \*/*

int getSolution (Integer solution[], int size) {  
 //line number is size-1  
 //words from solution[size] to size  
 int k;  
 if (solution[size] == 1)  
 k = 1;  
 else  
 k = getSolution(solution, solution[size]-1) + 1;  
 for(int i = solution[size]-1; i< size; i++){  
 lineNums[k] = size - solution[size];  
 }  
 return k;  
}

## Code

The code is located in Lab4/src/

## Results

|  |  |
| --- | --- |
| Word Count | Time to format (seconds) |
| 564 | 0.044 |
| 782 | 0.067 |
| 1188 | 0.162 |
| 1564 | 0.452 |
| 1782 | 0.866 |

## DP Tables

We used multiple tables in our solution, and we weren’t sure which ones you wanted so here are all of them. Please look at the code or algorithm to see what they are used for.

**Extra Space**

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 0 | -3 | -9 |
| null | 4 | 1 | -5 |
| null | null | 4 | -2 |
| null | null | null | 1 |

**Clean Extra Space**

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 0 | 2147483647 | 2147483647 |
| null | 16 | 1 | 2147483647 |
| null | null | 16 | 2147483647 |
| null | null | null | 0 |

**Total Cost**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 9 | 0 | 10 | 10 |

**Solution**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| null | 1 | 1 | 2 | 4 |

**lineNums**

|  |  |  |  |
| --- | --- | --- | --- |
| null | 0 | 1 | 0 |

## Execution

I implemented a few methods to execute the code.

1. You can pass in the number of characters in a line and a list of words to include
2. You can pass in the number of characters in a line and a path to a file with words in it.

It is regular java code so you can build and run it with javac and java or with an IDE (I would recommend IntelliJ since it is already in that format). It has a main class of Main. I put the input that I used in Lab4/input.txt