DP Word Wrapping

Tom Bohbot

April 2021

1. How will you minimize the penalty function:

   My program iterates (let this variable be i) from the end of the words inputted until the first word (inclusive), and has an inner loop (let this variable be j) which iterates from i until either the end of the list of words or if there is no more space available on the line, which ever comes first (therefore the inner loop is bounded by the smaller of two choices). As I iterate i, I check how much room is remaining on the line if I were to add the current word, by summing up the amount of characters in the word and adding it to a counter I have called charsOnLine. If the current word is not the first word of the line, then I add an additional character as each word is separated by one space. If adding the current word would put me over the limit of lineLength, then the previous amount of charsOnLine is the final amount of characters on that line. If nothing was recorded then infinity is returned, as that would mean that this is the first word, and it is greater than the length of the line. If the total amount of characters can fit within the line (this variable is called charsOnLine) then I use dynamic programming to check if inserting this new word into the line will help minimize the penalty. The way I check is by seeing if the remaining free characters (called remChars in recurrence) squared plus penalty[j+1] would be less than what is currently stored at penalty[i]. If this is the case then it replaces penalty[i]. Once j has reached a break condition then the final value for penalty[i] is recorded. The reason for adding penalty[j+1] is so that the first element of the penalty array will be equal to the total minimized penalty. Due to this, if the element being verified is the last element in the list of words then there is no adding penalty[j+1] to its total cost since there is no penalty that comes before it. This process will then repeat words.length*lineLength times as those are the two upper bounds of the loops, resulting in a runtime of O(words.length*lineLength). Once the outer loop terminates the penalty array's ith element will be filled with the penalty of its line plus the penalty of the summation of the rows before it, meaning that the first element will be equal to the total penalty. Having already mentioned the runtime, the total space complexity of this algorithm O(words.length) as I must create an array, called penalty, that is able to store the amount of total rows created which can only be as many as words there are assuming each line can take up 1 line (worst case).

2. Supply a fully-specified recurrence that solves the problem:

$$c[i][j] = \begin{cases} \min \begin{cases} \text{remChars}^2 + \text{penalty}[j+1] \\ \text{penalty}[i] \end{cases} & \text{If charsOnLine <= lineLength and i != j} \\ \min \begin{cases} \text{remChars}^2 \\ \text{penalty}[i] \end{cases} & \text{If charsOnLine <= lineLength and i == j} \\ \infty & \text{if i == j and words[i] > lineLength} \end{cases}$$

3. A brief explanation of how you construct the optimal layout and the “Big- O” space and computation requirements:

   To be able to construct the optimal layout I initially use some information derived from my minimumPenalty method. In the minPenalty method I create an array that tracks which indices belong on which lines in the method. This information is tracked at the same time as when I update penalty[i]. I simply just input into this new array, let us call it indices, indices[i] = to the j of when penalty[i] was updated. Having this information I now know which words are on the same line as the element corresponding to the ith word is the index of the last word on that line. Therefore, I simply add those words to a list in an inner loop and once I exit the inner loop I add it to my hashmap. Once the outer loop resets, the list of words does as well and I am ready to add the next line of words until the last line of words thanks to my indices array. I know once I have reached the last line of words since the element in the indices array will be equal to the index of the last element of words, so I know to break once this occurs. Once I exit the outer loop I have added all the words to the correct line and into my map, and am now able to simply return the map.

   This method runs in O(words.length*lineLength) time as my outer loop iterates from 1 until words.length, and my inner loop iterates from indices[i-1]+1 until there are either no more characters per line or the index, indices[i] is reached which ever occurs first. Since this must be bounded by the smaller term, and lineLength is a smaller bound, it is overall bounded by lineLength. Since the rest of the algorithm is made up of constant time operations, the algorithm is O(words.length*lineLength). Concerning the space complexity, it is O(words.length) as the worst case of this algorithm is that every word takes up its own line. The data structures I use are bounded by the amount of words inputted as well.