Jason Caplan
Design and Analysis of Algorithms
Professor Leff
HW13

1. Explanation of dynamic programming approach
   - Let $words_i$ represent the i^{th} word of input of size n, where i ranges from $1 \rightarrow n$, and let m be the allowed line length
   - We can break a given word wrapping problem into subproblems in the following way:
     - The minimum penalty to wrap n words is the minimum over all possibilities of the sum of two values
       - 1) the penalty of the last line, consisting of $words_i \rightarrow words_n$ (where i is the first word on the last line), and:
       - 2) (assuming the last line is as specified in bullet 1), the penalty of the optimal way to set up the previous lines using the remaining words, $words_1 \rightarrow words_{i-1}$
     - We **note** that there is an upper and lower bound to this first value, because the total length of the words on the last line cannot exceed the line length and cannot be less than the length of the n^{th} word
   - Using tabulation, we can start with the base case, i.e. that wrapping just the first word has a penalty of $(m - words_1.length)^2$, and build up solutions for each subsequent word we consider
2. Notation and Recurrence
   - Let $OPT[i]$ hold the minimum penalty for wrapping $words_1 \rightarrow words_i$.
   - Let $p_{i,j}$ be the penalty for placing the words from $words_i \rightarrow words_j$ (inclusive) on one line. We can find p as follows:
     - $charactersUsed_{i,j} = [\sum_{k=i}^{j} words_k.length] + (j - i)$
       - There are (j-i) spaces between the words
     - $p_{i,j} = \begin{cases} (m - charactersUsed_{i,j})^2 & m - charactersUsed_{i,j} \geq 0 \\ \infty & m - charactersUsed_{i,j} < 0 \end{cases}$
   - Using the logic from part 1, we can say that

$$OPT[n] = \begin{cases} \min\limits_{1<i<n} (p_{i,n} + OPT[i-1]) & n > 1 \\ p_{1,1} & n = 1 \end{cases}$$

3. Layout Construction and "Big-O" Space/Computation
   - The layout method works as follows:
     - While filling in the OPT array, we keep a second array of the same size, who's k^{th} element holds the value of i which proved to be the min in the calculation of $OPT[k]$ (i.e. $words_i$ is the beginning of the line that ends with $words_k$).
     - We start with a pointer at the end of the OPT array (which represents the last word of the last line), and find the previous line ending word by moving the pointer to the word just to the left of the first word on the current line (which we can find using the second array described in the previous bullet). We repeat until we have traversed the entire OPT array, gathering all the line ending words. Once we have all the line ending words, it is trivial to build the layout with one traversal of the input

- Space
  - The space required for the OPT array and the secondary array described above, which are borh size n, is $O(n)$. The space required for the layout map is just the size of the map itself, which should also be $O(n)$. We have a total of $O(n)$ space.
- Time
  - Filling in the OPT array is the most expensive operation, (as described above, getting the layout is linear). The outer loop will be $O(n)$, since we trapse through the array from left to right filling it out with the recurrence above. We use the note from part 1 to keep the inner loop of filling in OPT to $O(m)$ as opposed to $O(n)$, because we only need to check "to the left" as far as we can fit more words onto the line, and the calculation of a given $p_{i,j}$ becomes constant once we realize that for a given index k in OPT we don't need to recalculate $p_{i,k}$ from scratch each time we consider a more leftward i value, rather we can keep a running total of $charactersUsed$ as we march to the left looking for the best i (or first word on the line ending with k). Thus, total time is $O(n \times m)$.