1. Let $f(i)$ be a function that computes the size of the maximum subarray that includes the ith element of array A. We want to compute $\max_{1 \le i \le n} f(i)$.
   A recurrence relation that would compute f is the following:
   $$f(i) = \begin{cases} \max(f(i-1) + A[i], A[i]) & i > 1 \\ \max(A[1], 0) & i = 1 \end{cases}$$
   and then in order to get the maximum subarray all we need to do is keep track of the largest $f(i)$ we've seen so far and the index associated with it as well as the last time that $f(i)$ was negative, which will tell us when the max subarray begins. When $f(i)$ is negative, the largest subarray ending including element i is negative (and so is $A[i]$) and so that ith value won't be in our maximum subarray because we could always just take no elements and then have a subarray that summed to 0.
   The correctness of this recursion is clear by induction - assuming $f(i-1)$ is the size of the largest subarray until $i-1$, then the largest subarray including $f(i)$ has to have either size $A[i]$ or $f(i-1)+A[i]$ because we have to include the element and can either make it a part of the subarray we already had or discard the subarray we had and start a new maximum subarray at position i. The base case is also correct because either the largest subarray over one element is either 0 if the value is negative or it's just that element. My algorithm runs in $O(n)$ time and uses $O(1)$ (constant) space by storing only $f(i-1)$ and the largest value on each iteration. $f(i)$ corresponds to max_including_i in the pseudocode below.

```
max_overall = 0
max_including_i = 0
start = 0
end = 0
for i from 1 to n:
        max_including_i = max(max_including_i + A[i], A[i])
        if max_including_i > max_overall:
                max_overall = max_including_i
                if max_including_i = A[i]:
                        start = i
                end = i
return start, end, max_overall
```

2. Let $B(i, j)$ calculates the size of the imbalance of the first i elements of A partititoned into $j+1$ subarrays. We ultimately want to find $B(n, k)$. Let Imb(A[i,j]) calculate the difference between the weight of the subarray from i to j of A and the average of all the elements in A.
   A recurrence that will calculate $B(n, k)$ is the following:

   $$B(n, k) = \min_{k-1 \le j < n} [\max(B(j, k-1), \text{Imb}(A[j+1, n]))]$$

   For the base case, if $n \le k$ then $B(n, k)$ is equal to which ever element of A is the closest to the average of all the elements. The way to understand this recurrence is that either the most imbalanced subarray is the $k+1$st subarray or it is one of the first k subarrays among the first j elements of A. Since we are trying to minimize the imbalance of the partition, we iterate over all j between $k-1$ and n to place the kth partition point and by induction this will always find the smallest possible overall imbalance - whenever we find the index that minimizes this function, we need to keep track of it so that we will have the breaking points when the algorithm terminates. To actually run the algorithm, first calculate and store $\frac{1}{k+1} \sum_{l=1}^{n} A[l]$ and then fill an n by k sized array where every cell (i, j) of that array contains the value for $B(i, j)$. As we go along in order to make computing Imb(A[j+1,n]) more efficient we should store the previous value and then when we need to increase the size of the partition

as we iterate over j just add whatever the next element is to the current value of the imbalance of that subarray. We can fill this array row by row and in the worst case, it will take $O(n)$ steps to calculate the particular value for a cell by looking of the values of $B(n,k)$ we've already computed and stored in the table and so this will take $O(kn^2)$ time to complete.

If instead we defined the imbalance of the partition to be the sum over all i, then the recurrence would instead be:

$$B(n,k) = \min_{k-1 \le j < n} [B(j, k-1) + \text{Imb}(A[j+1, n])]$$

Other than redefining B(n,k), the algorithm would remain the same - it would still use $O(kn)$ space and run in $O(kn^2)$ time.

3. First run a depth first search on the tree, and using the topological order we can identify the root of the tree. This will take $O(|V| + |E|)$ time, but because this is a tree, $|E| = |V| - 1$ so the DFS runs in $O(|V|)$ time. From the DFS, we can identify the root vertex and starting from the root we can then calculate our blanket. One important thing to note is that in our optimal tree blanket, no leaves (vertices with no children) will be included. To see this, assume that a leaf was included in a potential minimal tree blanket. If its parent vertex were already in this blanket we can just remove this leaf from the blanket. If the parent were not in the blanket, include it and if it's the root of the tree stop, otherwise treat it like we just treated the leaf and work our way back to the root, removing and adding vertices from the blanket as necessary. At the end of this process our blanket will have one fewer vertex and so a leaf cannot be in a minimal blanket. Therefore when we try to find the blanket, once we include a vertex in the tree, we can essentially remove that node and edge from the tree and if that forces the parent node to become a leaf, then that parent node will not be in the blanket. Another important fact that is clearer to see is that if all the children of a vertex are in the minimal blanket then the parent will not be because all those edges will already be covered.
   Now to determine if a vertex is in the minimal blanket, check two things. First, if the node has no children (it's a leaf), then it is not in the blanket and second if all of the node's children are in the tree, then it's not in the blanket. Otherwise, let the vertex be in the blanket. Starting with the root and a bit vector initialized to all 0's that will store the vertices in the blanket, we just do one pass down the tree and each time we see that a vertex is in the tree we can update that vertex's position in the bit vector to a 1. When we come to a vertex who children haven't already been visited, first traverse down all of its children to see if they're in the blanket and then use the criteria I just mentioned to determine if the current node is in the blanket. This runs in $O(|V|)$ (the pass to do the DFS and the second pass to calculate the blanket are both $O(|V|)$) and uses an additional $O(|V|)$ space to store the blanket. Every time we determine if a vertex is in a blanket, change it's space in the bit vector for the blanket to a 1 so that all the other vertices will have access to it and we won't have to recompute those values. This runs in $O(|V|)$ time and takes $O(|V|)$ additional space to hold the blanket. If we've only been given adjacency lists (and not some more complicated set of pointers and other structures) which tell us the structure of the tree than as we traverse down the tree, in order to keep track of what's a parent node and a child, whenever we visit a node we should mark it as visited (in another bit vector) and so when we are examining the nodes to connected to one of its children we won't search it again because it will have already been visited.

4. Let $\text{cost}(i,j)$ be the cube of the number of spaces left at the end of a line of length M if words i to j are put onto the line. If there are no spaces left at the end of the line, then the cost should be set to infinity because we'll never be able to put this sequence of words into a line. If word j is the last word in the text, then the cost is 0 because we are not penalizing any spaces at the end of the text on the last line.
   The optimal cost of the first j words can be computed as follows:

$$\text{opt\_cost}[j] = \begin{cases} 0 & j = 0 \\ \min\limits_{j - \lceil M/2 \rceil \le i \le j} \text{opt\_cost}[i-1] + \text{cost}(i,j) & \text{otherwise} \end{cases}$$

So the way to break down this recurrence relation is that when we try to add the jth word, it can either be added to a new line of its own ($i = j$) or it can be added with some other words to a new line. Essentially though, opt_cost[i-1] gives us the minimal cost for the first $i - 1$ words spread across some number of lines and then $cost(i, j)$ gives the penalty accrued over the last line. We ultimately want opt_cost[n] where n is the number of words. Because we minimize the total error every time we calculate opt_cost[j] and find optimal way to break up the first j words, by induction it will be possible to do the same for all the n words and each time we find the optimal cost we can keep track of where the line break was separately. This can be done in $O(n)$ space and $O(nM)$ time. In my implementation, I relied on the fact that spaces[i, j] = spaces[$i + 1, j$] - length[i] - 1, where spaces[i, j] represents the number of spaces that words i to j leave at the end of the line (it is not an array), and so I never had to store all n^2 values that would give the number of spaces and thus cost of putting words i to j on the same line. In the actual Python code, opt_cost[-1] = 0, not 0 because arrays are 0 indexed in python and so the first word is actually word 0 not word 1 as I assumed in the recurrence above. The $O(n)$ spaces comes from the fact that I stored the lengths of all the words and the words themselves in two arrays and that the array that keeps track of the optimal costs is also of length n. The algorithm runs in $O(nM)$ time because to compute opt_cost[j] we have to examine all opt_cost[i] between $j - \lceil M/2 \rceil$ and j. We don't need to check any word before the $j - \lceil M/2 \rceil$ word because every line can only have $\lceil M/2 \rceil$ words since every word is separate by a space and a word has to be at least 1 character. The way the final for-loop of my implementation works is that it first assumes that word j will be on it's own line and then it works its way backwards adding the word that came before until it minimizes the error that having j words would have and stores the linebreak.

For $M = 72$, my Python implementation gives an error of 2104 on buffy.txt and for $M = 40$, I got an error of 2183. To run my Python script, type "python linebreaks.py filename M" and it will output the correctly formatted paragraph as well as the total error accrued over the lines.