Problem Set 4
Julia Pearl

I collaborated with Rodrigo Daboin Sanchez, Carissa Wu, Marwa, and Noah Epstein. All code and problems were done independently.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail. As always, try to make your answers as clear and concise as possible.

1 Consecutive Maximum Sum

Suppose we have an array A containing n numbers, some of which may be negative. We wish to find indices i and j so that

$$\sum_{k=i}^{j} A[k]$$

is maximized. That is, you want a consecutive group of numbers with the maximum sum. Find an algorithm that runs in time O(n), and also give the space used by your algorithm. Note: you should do an example or two; in particular, you should see that it is not the case that if you know the maximum consecutive sum of the first n 1 numbers, and the nth number, you can then find the maximum consecutive sum of the first n numbers.

Restating problem:
The task of our problem is to identify the indices whose consecutive values yield the maximum sum. This can be done using recursion and memoization.

Dynamic Programming Storage:
We will setup an array D with a size n. All indexes of D will store the maximum consecutive sum that completes at that index and that corresponding sum's start index. The highest value of D will be equal to the maximum consecutive sum in A.

Each entry of D will look like: D[a] = [max sum for a, start index i]

Defining the Recursion:
We will define our recursion for our DP algorithm as the following:
D[j][0], the max sum for a, equals:

$$D[j] = max \begin{cases} D[a-1][0] + A[a] & \text{iff } a > 0 \\ A[i] & \text{for all a} \end{cases}$$

Recursive Base Case:
Our base case of the recursion (the lowest case that we will build our recursion from), will be D[0] = [A[0], 0].

Final Return Value:
At the end, we will go through our D array and find the largest sum value, and return the index of the D array as our ending index and the array stored for that sum's corresponding start index as the start index.

Algorithm Outline:
We can define the algorithm as the following:
1) Set our first index of D array, D[0] to be equal to the first value of our A array and the starting index 0: [A[0], 0].

2) For all n spots in our D array, starting with index 1, run the recursive step.
For clarity's sake, let's run through the recursive step for an index a. We will set D[a] to be equal to the max value of the max sum stored in the a-1th index of the D array plus the ath index of the A array or just the value of ath index of the A array.
3) After we have filled up our D array with max sum values, let's go through and find the index of the D array with the largest max sum value which has an index j and a starting index stored as i.
4) We return that (i, j).

Time and Space Complexity:
Looking at this algorithm, the most significant space usage is from the D array which uses **O(n)** space. The algorithm goes through the entire array A once and goes through D once after, and both of those arrays have a size n, so the algorithm runs in **O(n)** time.

Proof of Correctness:

Proof. We can prove the correctness of this algorithm using induction.

We can induct upon number of indexes whose values in the D array are correct. In each index of the D array, there is stored the maximum consecutive sum and that corresponding sum's start index.

Note: each maximum consecutive sum per index is a sum that terminates addition at that index. So a maximum sum value of index i will only count consecutive sums of indexes smaller than or equal to i.

In the D array, as long as each index's corresponding max sum and start index are correct, then when we return the largest max sum of D and the corresponding indexes of this sum, we will be returning the correct information regarding the largest consecutive sum across the entire array.

Let's say we have an array A size x. We make a D array that is also size x.

Base Case 0: The first index we update with the correct max sum is index 0. We will set D[0] = [A[0], 0]. These values stored in D are correct because there are no indexes of the array that are smaller than 0. Thus, the maximum correct sum that completes at 0 is just the value of A[0] and the starting index is also 0.

Base Case 1: The second index we update with the correct max sum is index 1. We will compare D[0][0] + A[1] with A[1] and pick the maximum.
Case I: If A[1] > D[0][0] + A[1] (which means that A[0] was a negative number), then D[1] will not include A[0] in its sum. D[1] will be equal to [A[1], 1].

Case II: If A[1] < D[0][0] + A[1], then D[1] will store this value as the maximum sum with a starting index of 0: D[1] = [D[0][0] + A[1], 0].

Inductive Hypothesis: All data stored in D an index y in which y <x - 1 are correct.

Inductive Step: We will prove that the y + 1 index of D is correct.

Case I: If A[y + 1] > D[y][0] + A[y + 1], then D[y] will start a new max sum at index y + 1. Thus, D[y] will equal [A[y], y]. We know that D[y][0] is correct, so this update must be correct.

Case II: If A[y + 1]<D[y][0] + A[y + 1], then D[y + 1] will sum its value with the max consecutive sum of the previous index D[y][0]. The starting index will be D[y][1] or the yth starting index. Ultimately, D[y + 1] = [D[y][0] + A[y + 1], D[y][1]]. Again, we know that D[y][0] and D[y][1] is correct, so this update must be correct.

Thus, in both cases, updated the y + 1 index will yield the correct values stored in D. So ultimately, when D is scanned for the highest consecutive max value with a corresponding start and given finish index, it will yield the correct values. ■

2 Minimum Maximum Deviation

A challenge that arises in databases is how to summarize data in easy-to-display formats, such as a histogram. A problem in this context is the minimal imbalance problem. Again suppose we have an array A containing n numbers, this time all positive, and another input k. Consider k indices j1, j2, . . . jk that partition the array into k + 1 subarrays A[1, j1], A[j1 + 1, j2], . . . , A[jk + 1, n]. The weight w(i) of the ith subarray is the sum of its entries. The imbalance of the partition is

$$\max_i \mid w(i) - (\sum_{l=1}^{n} A[l])/(k+1) \mid \tag{1}$$

That is, the imbalance is the maximum deviation any partition has from the average size. Give an algorithm for determining the partition with the minimal imbalance given A, n, and k. (This corresponds to finding a histogram with k breaking points, giving k + 1 bars, as close to equal as possible, in some sense.) Also give the space used by your algorithm. Explain how your algorithm would change if the imbalance was redefined to be

$$\sum_i \mid w(i) - (\sum_{l=1}^{n} A[l])/(k+1) \mid \tag{2}$$

Restating the problem:
We will begin by restating the problem: we are looking to add k bars to a list size n, making sure that each k bar added minimizes that maximum deviation between partitions.

Recursion Method:
We will attack this problem recursively, going top down. Every time we add a bar, we will check the minimum of every possible placement of that bar, and for each placement, calculate the maximum deviation for the placement of that bar and all possible bars before it.

We will do so by creating a 2d array called D[n elements, k partitions]. Each D element will be represented by a tuple of (minimum imbalance for given bars, index to the right of the last bar placed) For an arbitrary D[x, y], we can think of x as the elements left to divide (or partition), and y as the bars left to place. We can define the recursion mathematically as the following:

$$D[x,y][0] = \min_{\text{for } i \in [k+1,x]} \begin{cases} \max \begin{cases} \sum_{l=i}^{x} el_l - \frac{sum}{y+1} \\ D[i-1, y-1][0] \end{cases} \end{cases}$$

Whenever we do reset a D value, we will set D[x, y][1] = i.

Explaining the Lower Restriction of i:
This will compare all possible bar placements for k bars and n elements. The restriction of i being greater than y is so that each bar will allow room for all other bars to their right. To give an example of this, let's say there was an array size 6 with 4 total bars to be placed. This recursion places bars right to left. (Remembering that we are 1 indexing our array and that we cannot place a bar at anything below our 1st index) If the 4th bar were to be placed in an index less than the 5th index, then there would not be room for 3 other bars to be placed.

Description (in steps) of the Algorithm:
Let's walk through this algorithm.
1) Let's initialize our D array to be D[size n, size k] with values of (∞, NULL). Let's also calculate the

total sum of the array A and store it. Let's make a variable called avg and set it equal to the sum of the total array divided by k + 1.
2) We will setup the recursion as outlined above. We will set D[x, y] equal to D[n , k]. If y is equal to 0, then our algorithm will return the sum of all elements 0 to x minus our average variable. This will cause our recursion to complete. We will set D[x, y][0] equal to the minimum, for i values in between k + 1 and n, of the maximum of the sum of elements in between k + 1 and n minus our avg variable and D[i - 1, k - 1].
3) Whenever we set the value of a minimum i, we will store that i value in the associated D[x, y][1] value.
4) After D is filled by the recursive step, we will make a (1 indexed) array size k called Bars. We will make a counter variable Count and set it to k. We will set D[x, y] to equal D[n, k]. While Count is greater than 0, we will set Bar[Count] equal to D[x, y][1]. If Count is greater than 1, we will then set D[x, y] equal to D[Bar[Count] - 1, Count - 1]. We will subtract one from Count. When this has completed, Bars will be filled with all indexes of the bars.
5) We will return Bars.

Time complexity and space complexity:
Our algorithm has a space complexity of O(NK) because of the 2D array–we are considering all possible bar placements. Time complexity, our algorithm has an O(K*(N-K)) complexity because for each k partition placed, we are considering N-K options.

Proof of correctness:

Proof. We will run induction on the y value of each D[x, y] value to prove the correctness of this algorithm. We will choose arbitrary x values, except note that all values of x must be greater than y. The arbitrary value of x will detract from the importance of the placement of the yth bar within the range of x for each x. If every update for any arbitrary x and for all possible values of y to the D array is correct, then the algorithm should return a correct final partition placement once the D[n, k] index is filled finally.

We can assume that the AVG variable of total sum divided by k + 1 is correct by basic arithmetic.

Base Case 0:
The first base case is the placement of 0th bar in an array with an arbitrary element X as the last element, which could be represented in the D array as D[X, 0]. The algorithm will not search for an optimal placement of the 0th bar, so it will store the sum of all elements to the left of x + 1 (so including x) starting at element 1 in D[x, 0][0]. We note that this storage value is not dependent upon X. The sum of the elements minus the avg variable is the correct value to be placed in a D[x, 0][0] index for any arbitrary x.

Base Case 1:
The second base case is the placement of the 1st bar in an array with arbitrary element x as the last element, which could be represented in the D array as D[x, 1]. The only restriction on the value of x is that it must be greater than 1. i Will have a value in between 2 and x. We will compare every possible placement of the 1st bar at position i, searching for the minimum of the maximum of the sum of the elements to the right of i and D[i - 1, 0][0]. We know that for any value in the D array with 0 partitions, the element is stored correctly. We can assume that the sum of elements in the range [i, x] minus the avg variable is correct. Thus, the maximum value per index i will be correct and the comparison of each i index will be correct in yielding the lowest max deviation value. When we accept the minimum, we will store the correct value for i in D[x, 1][1].

Inductive Hypothesis: We can assume that for y - 1 bars in which y ¡ k, all values stored in D with

arbitrary element x D[x, y - 1] are correct.

Inductive Step: We will prove that the D[x, y] element in the array is correct.

When examining D[x, y][0], we will review all values of i in the range of [y + 1, x]. For each i value, we will calculate the maximum between the sum of the elements between i and x minus the avg variable. This is simple addition and will yield the correct result. We know that the value stored in D[i - 1, y - 1][0] is correct because of the inductive step. Thus, choosing the maximum for each i will yield the correct result. Choosing the minimum for all of i's maximums will also yield the correct result. D[x,y][1] will store the minimum i value. Thus, we know that the D[x, y] index in the D array will be correct.

Because we have proven the correctness of every partition placement in subarrays that are to be stored in D, we have proven the correctness of this algorithm. ■

Considering a sum instead of a max: If we were to consider the minimum imbalance as the sum of all deviations, then we can simply alter our recursive definition to be the following:

$$D[x,y][0] = \min_{\text{for } i \in [k+1,x]} \left\{ \sum_{a=i}^{x} el_a - \frac{sum}{y+1} + D[i-1, y-1][0] \right.$$

3 DP on Trees

Dynamic programming can also work well on tree structures. Let us call a subset S of vertices a blanket if every edge is adjacent to at least one vertex of S. Suppose that we wish to find the mininum sized blanket for a tree. Find an algorithm that finds a minimum-sized blanket of a tree and runs in time O(—V —); also give the space used by your algorithm

Restating the Problem
Let's begin by restating the problem. We are looking for the minimum number of vertices required to create a blanket over a tree, or a minimum-sized blanket.

Methodology of the Dynamic Programming Solution
Explaining the methodology of the dynamic programming solution will begin with explaining how we intend to traverse the tree and store values of minimum spanning blankets. The algorithm will begin by arbitrarily choosing a node V and traversing to its children and grandchildren and storing the smallest blanket values per each ancestral line travelled. We will store information in an array D that is of size V. D is populated by tuples carrying the current min value blanket for that corresponding node and a boolean statement that tells whether or not each node is included in the final blanket or (number minval, boolean).

Population of D and its Recursion
D is recursed, starting at the root, with each D entry calling for the values of children and grandchildren. The recursion statement is the algorithm's decision, made at every vertex, to include that vertex in the blanket or to not include that vertex in the blanket but definitely include that vertex's children in the blanket. Whenever an inclusion statement is made for any generation, the recursion is run upon the next generation: if the vertex would be included, then the recursion would be run on the children, and if the vertex would not be included but the children would be included in the blanket, then the recursion would be run on the grandchildren.

We state the recursion of D below:

If D were to have a children and b grandchildren:

$$D[V][0] = min \begin{cases} 1 + \sum_{k=1}^{a} D[child_k][0] \\ k + \sum_{k=1}^{b} D[grandchild_k][0] \end{cases}$$

The base case of this recursion is that if a vertex were to have less than or equal to one neighbor, then that vertex would have an individual blanket of 0 and it itself would not be included in the blanket.

Determining Ancestry
We run into the problem in this algorithm of how to determine per node, which of its neighbors are children and which are parents. A great way to counter this (other than some assumption that we know who is who), is to use a slightly altered BFS. First, we'd make an array called Visited size V, and we'd populate it with -1's. Our BFS would look like the following: we'd start at an arbitrary node X, and give that node a layer 0. Then, we'd go to that node's neighbors. If a neighbor has a Visited[neighbor] value of -1, then we'd change that Visited[neighbor] to equal Visited[X] + 1. Then we'd repeat that process for each neighbor's neighbors, correctly incrementing each layer by 1. Then, in our recursion, we'd know each vertex's children by checking for each vertex, the number of neighbors neighbors with levels larger than the initial vertex. Then, our algorithm would cleanly traverse down the tree and would not

rewrite itself. This BFS has a runtime of O(V + E) which would be problem if we weren't running the BFS on a tree. Because it's a tree E = V - 1, so V + E = O(2V) = O(V). Timewise, we are still in the clear.

Defining the Algorithm
Now that we've given a broad description of the algorithm, let's define it step by step to give a clearer perception of what we are going to do.

fun MIN BLANKET ALG (Graph G):
1) Initialized the Visited array of size S with values of -1. We'd pick an arbitrary node X, set Visited[X] value to 0, and run ALTERED BFS to populate Visited.
2) Initialize array D of size S, of type (number, boolean).
3) Run the REC STEP(X, 0) to populate D.
4) Make an empty Blanket Array. Go through the first index of the D array (the boolean statements). If an index of D has a true boolean statement, add that index (representing the corresponding vertex) to the Blanket Array.
5) Return the Blanket Array!!

fun REC STEP(Vertex X, int Parent Layer):
1) X has N neighbors. If N is equal to 1, set D[V] equal to (0, false). Then, return 0. (This is our base case).
2) If Visited[X] is less than or equal to the Parent Layer, return 0. This step allows us to run the following recursion step on all neighbors of vertexes because, even if we do look at the parent of the vertex we are running the recursion step on, we give that parent a value of 0.
3) Else, D[X][0] is equal to the minimum of a. or b.:
a: 1 + for every Vertex i of the N neighbors, run REC STEP (i, Visited[X]).
b: N - 1 + for each Vertex i of the N neighbors, for each Vertex j of i's neighbor's, run REC STEP (j, Visited[i])
. 4) If a was smaller, set D[X][1] to true. If b was smaller, set D[X][1] to false. 5) Return D[X][0].

fun ALTERED BFS(X):
1) V has N neighbors. If V has no neighbors, break. If V has one neighbor with a Visited value not equal to -1, break.
2) For each vertex i of V's neighbors, if Visited[i] has a value equal to -1, set Visited[i] = Visited[V] + 1. Run BFS(i).

Runtime:
Looking at MIN BLANKET ALG, we can go through this step by step. Initializing the Visited array and running ALTERED BFS is O(V). Initializing the D array is O(1). When the REC STEP is run for all vertices of the tree, the recursive algorithm it traverses the tree in one direction only before returning values up the tree. This is O(V) time because we have guaranteed that we aren't rewriting our D array. From there, we go through D one more time which is another O(V). Taking out constant factor values, this MIN BLANKET ALG has a runtime of **O(V)**.

Space:
We are using an array of tuples size V to for DP information. We also have a Visited array size V. We use **O(V)** space.

Proving correctness:

Proof. We can use proof by induction upon the minimum blanket values stored in each level of the tree. If we have a tree with n layers, we will prove that each minimum blanket value stored at each of D's indexes is correct. We build minimum blankets starting at the root, yet we begin to fill in values of the blanket at the bottom of the tree. So, proving the correctness of each blanket value starting at the bottom of the tree will prove the ultimate correctness of the algorithm.

Base Case 0: We are focusing on the minimum blanket of the bottom layer of a tree with n layers. This layer is madeup of single vertices with no children. Our algorithm's base case will give these vertices a blanket size of 0 and will not include these vertices in the tree. Thus, D[a last layer vertex] = (0, false). The algorithm's blanket values made at the 1st or bottom layer of the tree are correct.
Base Case 1: We are focusing on a the bottom 2 layers of the tree of n total layers. We can think of the bottom two layers as a series of subtrees. Our algorithm will look at each root of these subtrees, and it will traverse to each of the children. Every vertex of the children layer (or bottom layer) is our Base Case 0, with values in the D array of (0, false). Thus, on the root layer (one above the bottom layer), the minimum comparison will be between 1 + 0 and the number of children of the last layer + 0. The number of children of the last layer has to be greater than or equal to one, so algorithm will yield a blanket size of 1. So, D[root of 2 layer subtree] = (1, true) or (1, false) (in the case in which there is only one child; the difference between these two solutions in that case is arbitrary). It will either include the root or, if the root only has one child, the child of the tree.

Inductive Hypothesis: Because we have shown the functionality of the algorithm for the base cases and the correctness of D storage for the base cases, we can state that for each k - 1 bottom layers of a tree with total n layers (in which k ¡= n), the algorithm will yield correct minimum blankets for all vertices included in the range of these layers and store correct values in the D array.

Inductive Step: We are trying to define the D values of the vertices (or roots) at the bottom k layers. Let's pick an arbitrary vertex of this kth to bottom layer and call it M. We know that the sum of all of M's children's minimum blankets stored in D are correct, so $\sum_{k=1}^{a} D[child_k][0]$ must be correct. We know that the sum of all of M's grandchildren's minimum blankets stored in D are correct, so $\sum_{k=1}^{b} D[grandchild_k][0]$ is correct as well (if M has no children or grandchildren, then these sum values will be 0). Thus the following equation:

$$D[M][0] = min \begin{cases} 1 + \sum_{k=1}^{a} D[child_k][0] \\ k + \sum_{k=1}^{b} D[grandchild_k][0] \end{cases}$$

will yield the correct minimum. Because M is arbitrary, we can assume that all vertices of the bottom k level of the tree will store the correct minimums. Because we have proven our base cases and we have proven that each layer's accuracy is dependent upon the accuracy of prior layers (or it's children and grandchildren), then we have proven the correctness of this algorithm. ■

4 Neat Paragraphs

(20 points) Suppose we want to print a paragraph neatly on a page. The paragraph consists of words of length ‘1, ‘2, . . . , ‘n. The maximum line length is M. (Assume ‘i M always.) We define a measure of neatness as follows. The extra space on a line (using one space between words) containing words ‘i through ‘j is M - j + i - $\sum_{k=i}^{j} l_k$. The penalty is the sum over all lines except the last of the cube of the extra space at the end of the line. This has been proven to be an effective heuristic for neatness in practice. Find a dynamic programming algorithm to determine the neatest way to print a paragraph. Of course you should provide a recursive definition of the value of the optimal solution that motivates your algorithm. Also give the space used by your algorithm.

For this problem, besides explaining/proving your algorithms as for other problems on the set, you should also code up your algorithm to print an optimal division of words into lines. The output should be the text split into lines appropriately, and the numerical value of the penalty. You can use any coding language you wish. You should assume that a word in this context is any contiguous sequence of characters not including blank spaces. You should not work with a partner on this coding problem. After coding your algorithm, determine the minimal penalty for the following review of the Season 1 Buffy DVD, apparently written by Ryan Crackell for the Apollo Guide, for the cases where M = 40 and M = 72. We will try to put the text of the review on the class page as well.

Restating the problem:
This algorithm solves the problem of finding the optimal line breaks in an inputted text file to minimize the total penalty. Penalty is defined as the extra space on the end of every line except the last cubed. The extra space can be calculated by subtracting the inputted max character per line value by the total characters of words and total spaces.

Concept Behind the Recursion:
The recursion we will use to solve this problem is very similar to what we used in the second part of problem 2. However, instead of placing k bars in between a series of values, we are placing an optimal amount of line spaces in between a series of values with an inputted max line. Besides from these differences, we can use the same concept of setting each value in our storage array based upon the minimum of all possible total sums achieved with each line break. (The min of all sums concept is what we can pull from 2).

Storage of Recursion:
While we do not know how many line breaks there will be, the worse case for the number of line breaks is the case in which every word is on its own line. Thus, we are going to make a D array of size n to hold penalties. Each value stored in the D array at arbitrary index I represents the optimal penalty sum if the last line break were to be placed after I. The algorithm will define build optimal penalties per line break and will store the final, total, lowest penalty in the last element: D[n-1]. The recursive function stated below will populate D.

Populating our Storage Array D/our recursive equation:
We populate D using the recursive equation. The recursive equation uses a penalty equation that accounts for the special case of not counting the penalty of the last line. We define the penalty equation and the recursive equation as the following:

We define
PENALTY EQUATION:

$$pen(j,i) = \begin{cases} (M - j + i - \sum_{k=i}^{j})^3 & \\ 0 & \text{if } j = n-1 \text{ and } (M - j + i - \sum_{k=i}^{j})^3 >= 0 \end{cases}$$

As mentioned previously, this penalty function factors in the case of the last line that is within the maximum boundaries having a penalty of 0.
The basic recursion (without specifications on j or i (essentially edge or base cases)) is:

$$D[j] = \min_{\text{for } i \in [0,j]} \Big\{ pen(j,i) + D[i-1]$$

Implementing D with base cases and edge cases yields:

$$D[j] = \min_{\text{for } i \in [0,j]} \begin{cases} pen(j,i) & \text{if } i = 0 \text{ and } pen(j,i) >= 0 \\ pen(j,i) + D[i-1] & \text{elif } pen(j,i) >= 0 \\ \infty & \text{else} \end{cases}$$

Tracing back to collect optimal line breaks and the final print:
When we need to collect the optimal line breaks from the D array, we can do so by creating an array called Indexes that is also size n. We initialize each value of Indexes with -1. Whenever we update a value of D[j] for an arbitrary j, we save the corresponding i value (the optimal starting index for that j line break) in Indexes[j]. When we need to collect all final optimal line breaks, we can trace back our indexes. We would start at Indexes[n - 1], then go to Index[Indexes[n-1]], and continue this pattern until we reach a value of -1. We can store these values (there are less than n of them). When we print out the words of the Text array, we can insert a line break after every time we reach any of these stored values (optimal index break values).

Outline of the Problem:
I have outlined and numbered the steps of the algorithm below for implementation of the DP Problem.

NEAT PARAGRAPHS ALGORITHM ():
1) Convert the text file to an array Txt. Create D and Indexes array. Fill D with infinities and Indexes with -1's.
2) Run RECURSION STEP on the last variable in Txt, populating Indexes and D.
3) Traceback Indexes starting at the last variable and store these optimal values in an array.
4) Print the last value of D as our total penalty.
5) Print the Txt array, inserting line breaks at our optimal values stored.

RECURSION STEP (Index X):
1) If X ¡= -1, return 0.
2) Set iterator i to be equal to X.
3) Set a MIN value variable to infinity and an INDEX variable to -1.
4) While the PEN(X, i) is greater than or equal 0, and i is greater than or equal to zero:
a. If i is equal to zero, return PEN(X, i).
b. Else, if D[i - 1] has a value of infinity, run RECURSION STEP(i - 1). c. If the PEN(X, i) plus D[i - 1] is less than MIN, set INDEX equal to i and change MIN to equal PEN(X, i) plus D[i - 1]. 5) Set D[X] equal to MIN and set Indexes[X] equal to INDEX.

PEN (X, i):
1) Add the number of characters of values from the TXT array in the range of [i, X]. Subtract this value

from M, subtract X, and add i. Cube this value.
2) If this value is greater than or equal to zero and X is equal to n - 1, return 0.
3) Else, return the value calculated in step 1.

Runtime and space:
Total runtime and space used per step: NEAT PARAGRAPHS ALGORITHM walk-through: Space for array Txt, D, and Indexes totals to O(3N). Creation of optimal values in an array has an upper bound of O(N). Thus, the total space used is O(4N) which is ultimately O(N).

Runtime for conversion of file to array is O(N). Tracing the Indexes array backwards has an upper bound runtime of O(N) (each access is O(1) just at most there are N or slightly less than N line breaks). Printing the Txt array has an upper bound of O(N + N) given the enters we have to input. Thus, without considering recursion, the runtime is O(4N) or O(N). Looking at the recursive step, we define every word's minimum penalty value in D by all N possibilities of a line break ending at each word. Just considering the while loop that starts at n, reduces an interator to 0, and runs that for every value of n, it is clearer to see that this step has a runtime of O(N^2). Thus, because O(N^2) dominates O(N), the overall runtime of this algorithm is O(N^2).

Proof of correctness:

Proof. We can prove correctness using induction on the index values of the D array, specifically basing correctness upon the correctness at each index. If every index value of D yield is correct, then the algorithm will return a correctly optimal line break placement. We will prove this for a list size n.

Base Case 0: We will focus on updating the 0th index of the D array. i has a range of [0, 0] and j = 0, thus D[0] will be equal to the output of the penalty function, PEN(0, 0). There are two cases for the return value of this function.
Case I: If n == 1, PEN(0, 0) will return 0, and D[0] will equal to 0. We have arrived at the last line and the last line has a penalty equal to 0. D only contains 1 word.

Case II: If n > 1, D[0] will hold the penalty value of the first word being on its own line.

Both of these cases yield the correct D[0] value.

Base Case 1: We update the 1st index of the D array. Looking at the recursion equation, j = 0 and i is in the range [0, 1]. D[1] will be equal to the minimum of PEN(0, 1) and PEN(1,1) + D[0]. This is essentially comparing placing the second word of the array on a the same line as the first word verses on the second line. In all cases, D[1] will contain the lesser penalty value of these two options. It will yield the correct result.

(Note* If the Text array were to only contain two words, then the lesser of the penalties would be the 0 value.)

PEN(j, i) Function:
Because the PEN function is just simple character summations or the assignment of 0 penalty if it's j index is is equal to the last word in the Text file, we know this function will return correct results.

Inductive Hypothesis: For all index values less than k in which n ¿= k, the D array holds the correct minimum penalty value.

Inductive Step: We are updating D[k] and looking at all i values in the range [0, k]. Because, unlike Base Case 1, we don't know how many values of i we will be comparing, we can choose a minimum variable MIN to keep track of the smallest values produced by all values of i. Just like the algorithm states, we will initialize min with positive infinity. As long as each update of MIN is correct, D[k] will ultimately be correct.

Case 1: i == 0, and PEN(k, i) is not negative. This is the case where i is the first word in the text file and k words can fit on the first line of the paragraph. Because it's the first line, the only penalty to consider is the first line's. If k is the last word of the text file, then PEN(k, i) will be equal to 0. Regardless of whether k is the last word of the text file or not, we can set MIN = min(PEN(k, i), MIN). We know this is the correct comparison to be making.
Case 2: i is not the first word in the text file, and the penalty of k and i are not negative. Because i is not the first word in the text file, we need to consider the penalty of k and i and the penalty of all words that came before i. Thus, we will compare with MIN PEN(k, i) plus the value stored in D[i - 1]. We know that PEN(k, 1) will yield a correct result and we use our inductive hypothesis to claim that D[i - 1] has a correctly stored value. Thus, when we set MIN = min(penalty k and i + D[i - 1], MIN), we know this is the correct comparison.
Case 3: The last case is when we have chosen an i whose penalty value of k and i is negative. This will yield a result of infinite. We will set MIN = min(infinity, MIN). This is the correct comparison.

Because all 3 types of comparisons are correct, we know that MIN value will be correct after all iterations of i. Thus, with the final update of D[k] to equal MIN, we will have stored a correct D[k] value. Thus, we have proven our base cases and our inductive step, ultimately proving the correctness of this algorithm. ■

The code below is the algorithm described above with the outputs of maxes 40 and 72 following it.
CODED ALGORITHM

```
import sys
from array import *

class pset_4:

    def __init__(self):
        self.Max = 0
        self.ret = []
        self.D = []
        self.indexes = []
        self.cuts = []

    def run(self):
        self.ret = self.read_words('buffy.txt')
        if (len(sys.argv) != 2):
            print("Usage: python filename.py max")
        self.Max = int(sys.argv[1])
        for i in range (0, len(self.ret)):
            self.D.append(((self.Max + 1)**3)*len(self.ret))
```

```
            self.indexes.append(-1)
        self.recurse_line(len(self.ret) - 1)
        iterator = self.indexes[len(self.ret) - 1]

        while (iterator != -1):
            self.cuts.append(iterator)
            iterator = self.indexes[iterator]
        print("Total Penalty:", self.D[len(self.ret) - 1])
        self.printin()

    def recurse_line (self, last):
        if (last == -1):
            return 0
        i = last
        if (self.penalty(i, last) < 0):
            return self.D[last]
        min_val = ((self.Max + 1)**3)*len(self.ret)
        index = -1
        while (self.penalty(i, last) >= 0 and i >= 0):
            if (i == 0):
                return (self.penalty(i, last))
            else:
                if (self.D[i - 1] == ((self.Max + 1)**3)*len(self.ret)):
                    self.D[i - 1] = self.recurse_line(i - 1)
                val = self.penalty(i, last) + self.D[i - 1]
                if (val < min_val):
                    min_val = val
                    index = i - 1
            i = i - 1
        self.D[last] = min_val
        self.indexes[last] = index
        return self.D[last]

    def penalty(self, start, last):
        sum = 0
        for i in range (start, last + 1):
            sum += len(self.ret[i])
        val = (self.Max - last + start - sum)**3
        if ((last == len(self.ret) - 1) and val >= 0):
            return 0
        return ((self.Max - last + start - sum)**3)

    def read_words(self, words_file):
        with open(words_file, 'r') as f:
            ret = []
            for line in f:
                ret += line.split()
            return ret
```

```
    def printin(self):
        old = 0
        new = len(self.cuts) - 1
        while (new >= 0):
            for i in range (old, self.cuts[new] + 1):
                print(self.ret[i], end =" ")
            print('')
            old = self.cuts[new] + 1
            new = new - 1
        for i in range(self.cuts[0] + 1, len(self.ret)):
            print(self.ret[i], end =" ")
        print(' ')

if __name__ == '__main__':
    pset4 = pset_4()
    pset4.run()
```

MAX 40

Here is the output of the coded algorithm when run with the Buffy file with a max of 40:

Total Penalty: 2183
Buffy the Vampire Slayer fans are
sure to get their fix with the DVD
release of the show's first season.
The three-disc collection includes all
12 episodes as well as many extras.
There is a collection of interviews
by the show's creator Joss Whedon in
which he explains his inspiration for
the show as well as comments on the
various cast members. Much of the same
material is covered in more depth with
Whedon's commentary track for the show's
first two episodes that make up the
Buffy the Vampire Slayer pilot. The
most interesting points of Whedon's
commentary come from his explanation
of the learning curve he encountered
shifting from blockbuster films like Toy
Story to a much lower-budget television
series. The first disc also includes
a short interview with David Boreanaz
who plays the role of Angel. Other
features include the script for the
pilot episodes, a trailer, a large photo
gallery of publicity shots and in-depth
biographies of Whedon and several of the

show's stars, including Sarah Michelle
Gellar, Alyson Hannigan and Nicholas
Brendon.

MAX 72

Here is the output of the coded algorithm when run with the Buffy file with a max of 40:
Total Penatly: 2104
Buffy the Vampire Slayer fans are sure to get their fix with the DVD release of the show's first season. The three-disc collection includes all 12 episodes as well as many extras. There is a collection of interviews by the show's creator Joss Whedon in which he explains his inspiration for the show as well as comments on the various cast members. Much of the same material is covered in more depth with Whedon's commentary track for the show's first two episodes that make up the Buffy the Vampire Slayer pilot. The most interesting points of Whedon's commentary come from his explanation of the learning curve he encountered shifting from blockbuster films like Toy Story to a much lower-budget television series. The first disc also includes a short interview with David Boreanaz who plays the role of Angel. Other features include the script for the pilot episodes, a trailer, a large photo gallery of publicity shots and in-depth biographies of Whedon and several of the show's stars, including Sarah Michelle Gellar, Alyson Hannigan and Nicholas Brendon.