CS 124 Problem Set 4
HARVARD ID: 10939860

1. **Solution.**

To find indices i and j to maximize the largest contiguous sum, we will keep two arrays, $currentMax$, or C, and $maxSoFar$, or M. $C[i]$ will output the value of the maximum contiguous subarray that ends at index i, and $M[i]$ will output the globally best maximum contiguous subarray that begins and ends anywhere before or at index i. To begin, let $C(-1) = M(-1) = 0$, in order to ensure that the maximum subarray before you begin reading the array is 0. To find the indices that the algorithm is looking for, start with two variables i and j initiated to value -1. They will represent the beginning and end of the maximum subarray. Define the recursion as follows:
$C(k) = max(C(k-1) + A[k], A[k])$
$M(k) = max(M(k-1), C(k))$

To further illustrate how this recursion operates, consider the following pseudocode:
maxSoFar = 0
currentMax = 0
for len(array):
currentMax = max(a[k], a[k] + currentMax)
maxSoFar = max(maxSoFar, currentMax)
endfor

At each step, the algorithm will determine whether the subarray ending at index k is larger if it was the kth array element added to the largest subarray ending at the previous index, or if it was simply the kth element alone. Then, the algorithm will determine whether the maximum of those two options is the largest possible subarray, and if it is, then update the maxSoFar variable to be that value. Thus, the algorithm will recurse through the array, searching and considering every possible subarray (including negative numbers), comparing each of them to each other, and thus in the end find the maximum subarray possible over the entire list.

When the maximum option for M is $C(k)$, then we know that the algorithm must update i and j. From here, if $C(k) > M(k-1)$ and $C(k-1) + A[i] > A[i]$, then we update j to equal k and keep i the same, since we add the kth element to the subarray. Further, if $C(k) > M(k-1)$ and $C(k-1) + A[i] < A[i]$, then the new subarray becomes only the element $A[i]$, so update $i = j = k$. If $C(k) < M(k-1)$, the algorithm keeps the same subarray, so maintain the same i and j, as they are optimal. This updating of i and j will make sure that the indices are always optimal, and will thus return the correct indices of the maximum contiguous subarray once the algorithm has finished reading through the entire array.

To analyze the running time of this algorithm. we see that the array will simply be read through once, with the operation for each member of the array A taking constant time, so we have $O(n)$ time complexity for the algorithm. The space complexity will be constant, as we only keep track of four variables: currentMax, maxSoFar, i, and j. Thus, the space complexity is $O(1)$.

2. **Solution.**

To solve this problem, we will use an algorithm that stores two tables, each of size $n * k$. One

table stores all of the minimum imbalances, and the other stores the index of the first value in the last partition. Let $M(a, b)$ be the minimum imbalance where a is the array up to index a and b is the number of partitions, and let $C(i, n)$ be the cost of a single partition according to the equation in the problem. Our case case for all recursions will be the case with any number of elements and only one partition, so the minimum imbalance will be the cost of the whole array. ***$I(a, b)$??? Define the recursion as follows: **what is the answer in the end?
$M(a, b) = min_i(max(M(i-1, b-1), C(i, a)))$
$I(a, b) = argmin_i(max(M(i-1, b-1), C(i, a)))$

The goal for this problem is to figure out if the last element in the array should be in a partition of its own or be stuck in the previous partition, and work recursively from there. We need to find where the boundary between the kth and $(k+1)$st partition should be. We start by not taking into account the last partition ($i-1$ to i, which is separated by $k-1$ partitions), finding the imbalance from 1 to $i-1$, and then the cost from i to the end of the array. We take the maximum of these two values to determine which will cost more, and then find the i that minimizes that value. This will start with the last partition and move backwards, ensuring that every imbalance is exhausted and we end up with the total minimum imbalance. At every step, the algorithm will determine if the last element should be part of its own partition or be in the previous partition, so it will recursively find every optimal boundary to minimize the total imbalance.

The space complexity of this algorithm will be $O(nk)$, since we just have to store the minimum imbalance table and the index table. The time complexity will be $O(n^2k)$, since it takes linear time to fill up each spot in the two tables, and there are $2nk$ spaces.

For the second case, we simply adapt our recursions to take the sum of $M(i-1, b-1)$ and $C(i, a)$ instead of the maximum, but otherwise keep the recursion perform the same way.

3. **Solution.**

To find the minimum sized blanket for a tree, we can think of this process as a decision on each node to include it in the blanket or not. Consider three recursions, where one will be the optimal solution for a certain node that determines whether or not to include it in the blanket, one will be the case in which that node is included in the blanket, and one will be when that node is not included in the blanket. Name these recursions X_{opt}, X_{in}, and X_{out}, respectively, and consider the recursion equations as follows:
$X_{opt}(r) = sum(min(len(X_{in}(r.children)), len(X_{out}(r.children))))$
$X_{in}(r) = S \cup r \cup X_{opt}(r.children)$
$X_{out}(r) = sum(X_{in}(r.children))$

As a base, initialize all functions to have a value of 0 with trees of size 0 or 1 nodes. To show why this algorithm works, we will consider it step by step starting from the bottom. In X_{out}, if a node is not included in the blanket, then all of its children must be included, so include them in the blanket and recurse from there. In X_{in}, if a node is included in the blanket, then we take the union of that node, the set of vertices we already have in the blanket, and the optimal solution for that node's children, since the children may or may not be included in the blanket, and the recursion continues. In X_{opt}, we take the sum of the minimum of either the lengths of the recursive call on X_{in} or the lengths of the recursive call on X_{out}. In easier language, the function will take the union of all optimal choices for children until there are no children remaining. We can calculate the size of the

blanket for both of the options to include or exclude any node, and then choose the minimum option after exhausting all possible options. Each of the three functions will recurse down to the bottom of the tree, and X_{opt} will decide the minimum option at every point, thus providing the minimum blanket at the end when $X_{opt}(root)$ is originally called.

The running time will be $O(|V|)$. The X_{in} and X_{out} functions will each occur in linear time, since they will visit each node in the tree once, and X_{opt} will occur in constant time as it simply takes the minimum. Thus, the total time will be linear as a function as the number of nodes, $O(|V|)$. The space complexity will also be linear, because only the optimal value must be remembered for every single node, so it is also $O(|V|)$.

4. **Solution.**

To approach this problem, consider two arrays. One array will be C. Let $C[j]$ be the optimal penalty of printing the words from 1 to j, such that word j ends a line. The other array will be P, which is a parallel array to point to the variable in the recursion for C that gives the optimal value for $C[j]$. To illustrate this, consider the following recursion:
Let $s(i,j) = (M - j + i - \sum_{k=i}^{j} l_k)^3$
$C[j] = min_{1 \leq i \leq j}(C[i-1] + s(i,j)), C[0] = 0.$
Thus, P points to the i that gives the optimal value for $C[i-1]$. So, the last line of the paragraph begins at $P[n]$ and ends at n, the previous line goes from $P[P[n]-1]$ through $P[n]-1$, and so on.

This algorithm will find the optimal line breaks by exhausting all possible breaks for each word, and testing whether any given words from i to j yield a proper line, and comparing them all to each other. It recursively checks for the start word for each line that will minimize the penalty for each line and thus the penalty for the total paragraph. Starting from the end of the paragraph, it will find the optimum way to print the last line, the last two lines, etc. until the whole paragraph is optimized.

For $M = 40$, the minimal total penalty that my algorithm calculated was around 68,000. For $M = 70$, it was around 390,000. I'm fairly sure that my algorithm is not quite right, though please see my code to see my logic and progress after much effort!

For the given algorithm, the space complexity will be $O(n)$, since it is only storing the arrays, and the time complexity will be $O(n^2)$ due to the recursion over all i to optimize every line break.

I collaborated on this problem set with Kate Schember and Caroline Teicher.