DP and Graph Algorithms
Problem Set 2 – CS 6515/4540 (Fall 2025)

Answers to problem set 2, question 4.

4 DP: Balancing Array

This problem can be solved using dynamic programming. Below I will outline my algorithm. For any index i $\in [0, n-1]$ in the array and for any j $\leq$ k, I define T[i][j] as the minimum achievable largest sum in the array $A[0, ..., i]$ partitioned into j contiguous non-empty subarrays.
Then we can define our induction hypothesis as:

$$T[i][j] = \min_{p \in [j-1, i-1]} (\max(T[p][j-1], RangeSum(p+1, i))$$

Here, $RangeSum(a, b)$ gives us the sum of elements in A from index a to b. This can be computed in $O(1)$ if we pre-compute and store the cumulative sums in A. If we know the cumulative sum, defined as $CumSum(i)$ is cumulative sum up till index i, then $Range(a, b) = CumSum(b) - CumSum(a-1)$.

What our hypothesis is implying is, given we know the minimum achievable largest sum in all possible subarrays of the array $A[0, ..., i-1]$, if we want to compute the new result up till the i^{th} element, we have to figure out a partition point, p, such that we're minimising the max sum obtained after adding the new element to out last subarray.

In its naive form this algorithm is $O(n^2 k)$, but we can make an improvement here. Note the following observation: $T[p][j-1]$ increases as p increases, on the contrary $RangeSum(p+1, i)$ decreases as p increases. Therefore, using binary search we can identify the optimal inflection point, p such that $T[p][j-1] \geq RangeSum(p+1, i)$, where $p \in [j-1, i-1]$. This reduces the complexity to $O(nklog(n)$.

1. Using the above algorithmic description I define:

   $T[i][j]$ = minimum achievable largest sum when partitioning the subarry $A[0, ..., i]$ into j subarrays.

2. What our hypothesis is implying is, given we know the minimum achievable largest sum in all possible subarrays of the array $A[0, ..., i-1]$, if we want to compute the new result up till the i^{th} element, we have to figure out a partition point, k, such that we're minimising the max sum obtained after adding the new element to out last subarray.
   Formally, for $1 \leq i \leq n$ and $1 \leq j \leq k$:

   $$T[i][j] = \min_{p \in [j-1, i-1]} (\max(T[p][j-1], RangeSum(p+1, i))$$

   where

   $$RangeSum(A[p+1, .., i]) = \sum_{t=p+1}^{i} A[t]$$

   and we use **binary search to compute** the $\min\limits_{p \in [j-1, i-1]}$ as outlined above.

   Base cases:

   - $T[i][1] = \sum_{t=0}^{i} A[t], \quad \forall i \in [0, n-1],$
   - $T[i][0] = 0$

The final answer is $T[n-1][k]$.

This is correct as when we add the i^{th} element we have to naively look at all our subarrays and identify if we can get a new minimum achievable largest sum by considering the last subarray from the partition point p. Note the following observation: $T[p][j-1]$ increases as p increases, on the contrary $RangeSum(p+1, i)$ decreases as p increases. Therefore, using binary search we can identify the optimal inflection point, p such that $T[p][j-1] \geq RangeSum(p+1, i)$, where $p \in [j-1, i-1]$. Once we have this optimal p we need only check the $p-1$, and $p+1$ indices to verify they're not better partition points.

3. Based on how we've defined the entries of the table, we see that for any i, j we're doing maximum $log(n)$ work to find the optimal partition point, following which taking the max is just an $O(1)$ operation. We'll have a total of $n * k$ entries in our table, therefore the time complexity is $O(nk \log n)$.
As the remark states, we can design a purely binary search based algorithm that has time complexity $O(nlogn)$.