Omer Seyfeddin Koc
EECE 7205: Fundamentals of Computer Engineering
October 25, 2023

Project 1 (15 pt.)

Submission instruction:

- This is an individual project, and each student must implement the codes independently.
- In your first submission attempt on Canvas, upload a zip file including your source codes and executable file.
- In your second submission attempt, upload a pdf file as project report.
- The project report should have the following parts:
  1) Pseudo codes of your dynamic programming algorithm.
  2) Analysis of the running time asymptotically.
  3) Grouping results of several input examples including the one that A={3,9,7,8,2,6,5,10,1,7,6,4} and M=3.
  4) Source codes.

Problem Description:
You are given an input array $A[1, \ldots, N]$. A grouping of the array A is described by an array $G[1, \ldots, M]$, where the array A is partitioned into M groups, the 1st group consists of the first $G[1]$ elements of array A, the 2nd group consists of the next $G[2]$ elements, and so forth. Define array $B[1, \ldots, M]$ such that $B[j]$ is the summation of the elements in the j-th group of array A. Use a dynamic programming algorithm to find a grouping of array A with M groups such that we maximize the minimum element of array B.

```
Max-min-grouping $(A, N, M)$
{



return $G[1, \ldots, M]$
}
```

Hint:

- **The optimal subproblem property:** suppose the optimal solution to Max-min-grouping (A, N, M) is $G[1, \ldots, M] = [n_1, n_2, \ldots, n_{M-1}, n_M]$. Then $G[1, \ldots, M-1]$ is the optimal solution to the subproblem Max-min-grouping $(A, N - n_M, M - 1)$.

Question 1. Pseudo Code

The pseudocode for the dynamic programming algorithm designed for this problem is as follows:

Algorithm 1 Max-Min-Grouping(A, N, M)

```
Let $dp[\,][\,]$                                  ▷ Create a 2D array to store intermediate results
Let $P[\,][\,]$                                   ▷ Create a 2D array to store optimal positions for division
Let $G[\,]$                                       ▷ Create an array to store group sizes

// Base case: initialize the first row of dp
$dp[0][0] = A[0]$
for $i \leftarrow 1$ to $N-1$ do
    $dp[0][i] = dp[0][i-1] + A[i]$
end for

// Dynamic programming to fill dp and P tables
for $j \leftarrow 1$ to $M-1$ do
    for $i \leftarrow j$ to $N-1$ do
        for $k \leftarrow j-1$ to $i$ do
            $temp = \min(dp[j-1][k], dp[0][i] - dp[0][k])$
            if $temp > dp[j][i]$ then
                $dp[j][i] = temp$
                $P[j][i] = k$
            end if
        end for
    end for
end for

// Calculate group sizes based on positions stored in the P table
$idx = N-1$
for $j \leftarrow M-1$ downto $0$ do
    $G[j] = idx - P[j][idx]$
    $idx = P[j][idx]$
end for

$G[0] = G[0] + 1$

return $G$
```

The Max Min Grouping Algorithm aims to divide an array (A) into M groups in a way that minimizes the maximum sum within any group. The algorithm utilizes dynamic programming to achieve this goal. Here is a step-by-step explanation of the pseudocode:

- **Initialization:**
  - `dp`[ ][ ]: A two-dimensional array used to store intermediate results.
  - `P`[ ][ ]: A two-dimensional array used to store optimal positions for division.
  - `G`[ ]: An array used to store the sizes of the groups.
- **Base Case Initialization:**
  - Initialize the first row of the `dp` array by calculating the cumulative sum of the array elements up to the current index.
- **Dynamic Programming to Fill `dp` and `P` Tables:**
  - Iterate through the arrays and calculate the optimal partition positions. For each group and position, find the minimum value between the maximum sum of the previous group and the

sum of elements from the current position. Update the `dp` array with this minimum value and store the partition position in the `P` array.

- **Calculate Group Sizes:**
  - Trace back from the last group to the first group using the partition positions stored in the `P` array. Determine the size of each group based on the partition positions and store the group sizes in the `G` array.
- **Return Result:**
  - The algorithm returns the `G` array, which represents the sizes of the groups after minimizing the maximum sum within each group.

Question 2. Analysis of the Running Time Asymptotically

The given dynamic programming algorithm for the *Max Min Grouping* problem has a time complexity of $O(N^2 \cdot M)$, where N is the size of the input array and M is the number of groups.

1) **Initialization of `dp` and `P` Arrays:** The initialization step takes $O(N \cdot M)$ time, as we iterate through N for each of the M groups.
2) **Filling the `dp` and `P` Tables:** The nested loops that fill the `dp` and `P` tables run in $O(N^2 \cdot M)$ time. This is because there are three nested loops: the outer loop runs M times, the middle loop runs N times, and the innermost loop runs at most N times as well.
3) **Calculating the Group Sizes:** The final loop that calculates the group sizes runs in $O(M)$ time, as it iterates through the number of groups.

Therefore, the overall time complexity of the algorithm is $O(N^2 \cdot M)$.This analysis assumes that the operations involved in the calculations inside the nested loops take constant time. If the operations inside the loops are more complex and depend on the size of the input, the time complexity might differ accordingly.

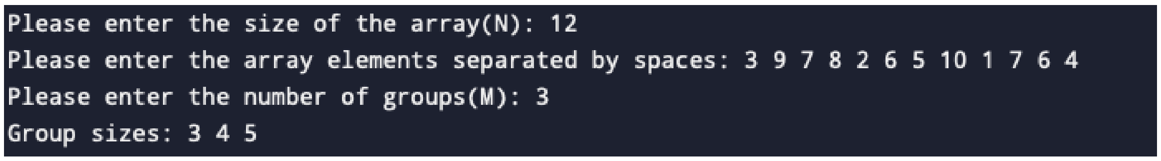
Question 3. Input Examples

The Max Min Grouping algorithm aims to divide an array into several groups, minimizing the maximum sum within any group. Below are examples of input arrays, the number of groups to be formed, and the resulting group sizes after applying the algorithm.

Example 1:

- **Array Size:** 12
- **Array Elements:** 3 9 7 8 2 6 5 10 1 7 6 4
- **Number of Groups:** 3

Interpretation: In this case, the algorithm processes an array of size 12 with elements {3, 9, 7, 8, 2, 6, 5, 10, 1, 7, 6, 4} and forms 3 groups. After processing, the resulting group sizes are {3, 4, 5}.This example is one that we solved in class. As indicated in the output of the algorithm I created, the group sizes to be formed are 3, 4, 5. This demonstrates that it produces the same solution as the one we discussed in class.

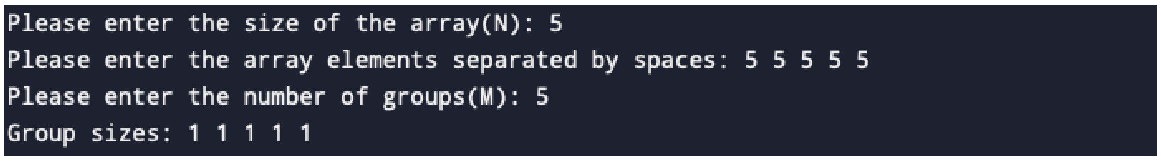


```
Please enter the size of the array(N): 12
Please enter the array elements separated by spaces: 3 9 7 8 2 6 5 10 1 7 6 4
Please enter the number of groups(M): 3
Group sizes: 3 4 5
```

Example 2:

- **Array Size:** 5
- **Array Elements:** 5 5 5 5 5
- **Number of Groups:** 5

Interpretation: For an array of size 5 with all elements being 5, the algorithm creates 5 separate groups, each containing only one element. Hence, the group sizes are {1, 1, 1, 1, 1}.

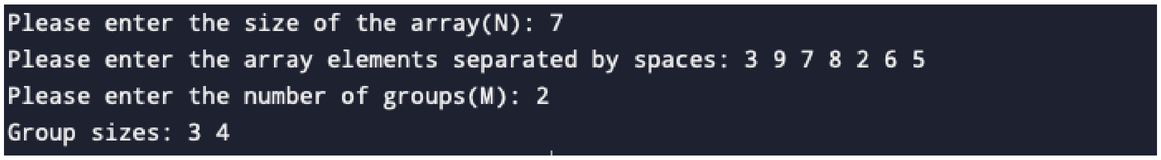


```
Please enter the size of the array(N): 5
Please enter the array elements separated by spaces: 5 5 5 5 5
Please enter the number of groups(M): 5
Group sizes: 1 1 1 1 1
```

Example 3:

- **Array Size:** 7
- **Array Elements:** 3 9 7 8 2 6 5
- **Number of Groups:** 2

Interpretation: In this scenario, the algorithm processes an array of size 7 with elements {3, 9, 7, 8, 2, 6, 5} and divides them into 2 groups. After processing, the resulting group sizes are {3, 4}.

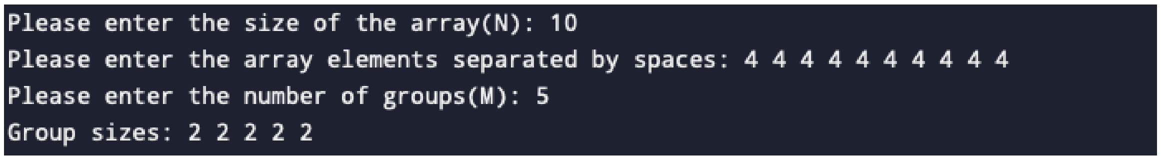


```
Please enter the size of the array(N): 7
Please enter the array elements separated by spaces: 3 9 7 8 2 6 5
Please enter the number of groups(M): 2
Group sizes: 3 4
```

Example 4:

- **Array Size:** 10
- **Array Elements:** 4 4 4 4 4 4 4 4 4 4
- **Number of Groups:** 5

Interpretation: For an array where all elements are 4 and the number of groups is 5, the algorithm creates 5 groups, each containing 2 elements. Therefore, the group sizes are {2, 2, 2, 2, 2}.

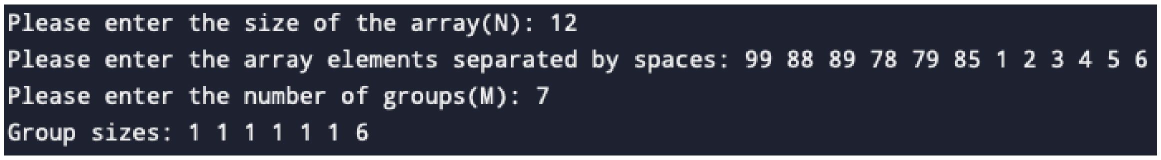


```
Please enter the size of the array(N): 10
Please enter the array elements separated by spaces: 4 4 4 4 4 4 4 4 4 4
Please enter the number of groups(M): 5
Group sizes: 2 2 2 2 2
```

Example 5:

- **Array Size:** 12
- **Array Elements:** 99 88 89 78 79 85 1 2 3 4 5 6
- **Number of Groups:** 7

Interpretation: In this case, the algorithm processes an array of size 12 with elements {99, 88, 89, 78, 79, 85, 1, 2, 3, 4, 5, 6} and divides them into 7 groups. After processing, the resulting group sizes are {1, 1, 1, 1, 1, 1, 6}.

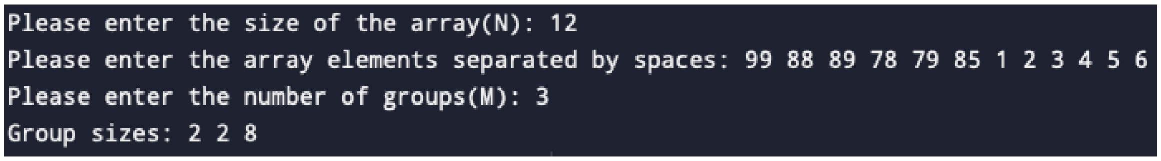


```
Please enter the size of the array(N): 12
Please enter the array elements separated by spaces: 99 88 89 78 79 85 1 2 3 4 5 6
Please enter the number of groups(M): 7
Group sizes: 1 1 1 1 1 1 6
```

Example 6:

- **Array Size:** 12
- **Array Elements:** 99 88 89 78 79 85 1 2 3 4 5 6
- **Number of Groups:** 3

Interpretation: In this scenario, the algorithm processes an array of size 12 with elements {99, 88, 89, 78, 79, 85, 1, 2, 3, 4, 5, 6} and forms 3 groups. After processing, the resulting group sizes are {2, 2, 8}.



```
Please enter the size of the array(N): 12
Please enter the array elements separated by spaces: 99 88 89 78 79 85 1 2 3 4 5 6
Please enter the number of groups(M): 3
Group sizes: 2 2 8
```

Question 4. Source Code

The code for this algorithm written in C++ is as follows:

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Function max_min_grouping takes a vector A, representing an array of integers,
// Two integers N and M as input.
vector<int> max_min_grouping(vector<int> A, int N, int M) {
    vector<vector<int>> dp(M, vector<int>(N, 0));
    vector<vector<int>> P(M, vector<int>(N, 0));
    vector<int> G(M, 0);

    // Base case: initialize the first row of dp.
    dp[0][0] = A[0];
    for (int i = 1; i < N; ++i) {
        dp[0][i] = dp[0][i - 1] + A[i];
    }

    // Dynamic programming to fill dp and P tables.
    for (int j = 1; j < M; ++j) {
        for (int i = j; i < N; ++i) {
            for (int k = j - 1; k <= i - 1; ++k) {
                // Calculate the minimum of two values:
                // 1. Previous group's maximum sum (dp[j - 1][k])
                // 2. Sum of elements from k+1 to i (dp[0][i] - dp[0][k])
                int temp = min(dp[j - 1][k], dp[0][i] - dp[0][k]);
                // If the temporary value is greater than the current value in dp table
    , update dp[j][i] and store the position in P table.
                if (temp > dp[j][i]) {
                    dp[j][i] = temp;
                    P[j][i] = k;
                }
            }
        }
    }

    // Calculate the group sizes based on the positions stored in the P table.
    int idx = N - 1;
    for (int j = M - 1; j >= 0; --j) {
        G[j] = (idx) - P[j][idx];
        idx = P[j][idx];
    }
    G[0] = G[0] + 1;

    // Return the final group sizes.
    return G;
}

// The main function takes user input for the array size, elements, and number of
    groups.
int main() {
    while (true) {
        int N, M;
        cout << "Please enter the size of the array(N): ";
        cin >> N;

        vector<int> A(N);
        cout << "Please enter the array elements separated by spaces: ";
        for (int i = 0; i < N; ++i) {
            cin >> A[i];
        }

        cout << "Please enter the number of groups(M): ";
        cin >> M;

        // Call the max_min_grouping function and get the group sizes.
        vector<int> G = max_min_grouping(A, N, M);

        // Output the group sizes.
        cout << "Group sizes: ";
        for (int i = 0; i < G.size(); ++i) {
            cout << G[i] << " ";
        }
        cout << endl;
    }
    return 0;
}
```