Fundamentals of Computer Engineering: Project 1

Name: Rohit Gurusamy Anandakumar
Date: 3/3/2024

1. **Dynamic Programming Algorithm Pseudo Code:**

```
Algorithm max_min_grouping(A, N, M)
Input:
- A: Input array
- N: Size of input array
- M: Number of groups

Output:
- Print the size of each group in the optimal partitioning
- Print the maximum minimum sum

Declare B as a 2D array of size [N][M]
   // Stores the maximum minimum sum for each partition
Declare P as a 2D array of size [N][M]
// Stores the partition index that leads to the maximum minimum sum
Declare C as an integer
   // Initialize B array
for i = 0 to N-1:
     B[i][0] = sum(A, 0, i)

// Dynamic programming loop
for j = 1 to M-1:
     for i = j to N-1:
        B[i][j] = 0
        for k = j-1 to i-1:
            C = min(B[k][j-1], sum(A, k+1, i))
            if C > B[i][j]:
                B[i][j] = C
                P[i][j] = k

// Construct the grouping
i = N - 1
j = M - 1
Declare groupSizes as an array of size [M]
while i >= 0 and j >= 0:
     groupSizes[j] = i - P[i][j]
     if j != 0:
         i = P[i][j]
     if j == 0:
         groupSizes[j] = i + 1
     j -= 1

// Print the size of each group in the optimal partitioning
for k = 0 to M-1:
     Print "Group " + (k+1) + " size: " + groupSizes[k]

// Print the maximum minimum sum
Print "MAX MIN Sum: " + B[N-1][M-1]
```

2. **Run Time Analysis:**

Dynamic Programming Loop:

- The outer loop runs for M iterations.
- The middle loop runs for N iterations.
- The innermost loop runs for N - j iterations in the worst case, which is O(N).
- Therefore, the main loop has a time complexity of O(M * N^2).
- Constructing the Grouping: The section that constructs the grouping does not exceed O(M) iterations since we are constructing M groups.
- Printing the Group Sizes: Printing the group sizes is O(M).

Overall, the dominant factor in the time complexity is the main Algorithm(Max_min_grouping () function). Hence, the asymptotic runtime complexity of the program is O(M * N^2), where M is the number of groups we want to create, and N is the size of the input array

3. **Grouping Results with Several inputs:**

   a. **Example1**

```
Enter the size of array A: 12
Enter the elements of array A: 3 9 7 8 2 6 5 10 1 7 6 4
Enter the number of groups (M): 3
Group 1 size: 3
Group 2 size: 4
Group 3 size: 5
MAX MIN Sum: 19
PS C:\Users\HP>
```

   b. **Example2:**

```
Enter the size of array A: 5
Enter the elements of array A: 5 5 5 5 5
Enter the number of groups (M): 4
Group 1 size: 1
Group 2 size: 1
Group 3 size: 1
Group 4 size: 2
MAX MIN Sum: 5
```

   c. **Example 3:**

```
Enter the size of array A: 10
Enter the elements of array A: 68 3 7 6 6 6 8 8 8 8
Enter the number of groups (M): 5
Group 1 size: 1
Group 2 size: 3
Group 3 size: 2
Group 4 size: 2
Group 5 size: 2
MAX MIN Sum: 12
```

d. **Example 4:**

```
Enter the size of array A: 10
Enter the elements of array A: 1 1 1 1 1 55 65 35 45 85
Enter the number of groups (M): 3
Group 1 size: 7
Group 2 size: 2
Group 3 size: 1
MAX MIN Sum: 80
```

Source Code:

```
#include <iostream>
#include <climits>

using namespace std;

// Function to calculate the sum of elements in the array A from index
'start' to index 'end'
int sum(int A[], int start, int end) {
    int total = 0;
    for (int i = start; i <= end; i++) {
        total += A[i];
    }
    return total;
}

// Function to find the maximum minimum sum partitioning
void max_min_grouping(int A[], int N, int M) {
    // 2D array to store the maximum minimum sum for each partition
    int B[N][M];
    // 2D array to store the partition index that leads to the maximum
minimum sum
    int P[N][M];
    int C;

    // Initialize the first column of B with the sum of elements up to that
index
    for (int i = 0; i < N; i++) {
        B[i][0] = sum(A, 0, i);
    }

    // Calculate the maximum minimum sum for each partition
    for (int j = 1; j < M; j++) {
        for (int i = j; i < N; i++) {
            B[i][j] = 0;
            for (int k = j - 1; k < i; k++) {
          // Calculate the maximum minimum sum and update B and P
                C = min(B[k][j - 1], sum(A, k + 1, i));
                if (C > B[i][j]) {
                    B[i][j] = C;
                    P[i][j] = k;
                }
```

```
            }
        }
    }

// Construct the grouping by backtracking from the last element of P
    int i = N - 1;
    int j = M - 1;
    int groupSizes[M];
    while (i >= 0 && j >= 0) {
        groupSizes[j] = i - P[i][j];
        if (j != 0)
            i = P[i][j];
        if (j == 0) {
            groupSizes[j] = i + 1;
        }
        --j;
    }

    // Print the size of each group in the optimal partitioning
    for (int k = 0; k < M; k++) {
        cout << "Group " << k + 1 << " size: " << groupSizes[k] << endl;
    }

    // Print the maximum minimum sum
    cout << "MAX MIN Sum: " << B[N - 1][M - 1] << endl;
}

int main() {
    int N, M;
    cout << "Enter the size of array A: ";
    cin >> N;
    int * A = new int[N];
    cout << "Enter the elements of array A: ";
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }
    cout << "Enter the number of groups (M): ";
    cin >> M;
    max_min_grouping(A, N, M);
    delete[] A;
    return 0;
}
```