Sebastian Loaiza
Adrian Lozada
Vincent Tran

COT 4400

# Project 2 Report

1. How can you break down a large problem instance into one or more smaller instances? Your answer should include how the minimum absolute difference for the original problem is constructed from the minimum solutions to the subproblems and why this breakdown makes sense.

    To address this problem effectively, we can decompose it into smaller, more manageable subproblems. Each subproblem involves finding the minimum difference for an individual entry within the matrix. We approach this by considering all potential paths originating from the specific entry and selecting the path yielding the smallest difference. For instance, imagine we're positioned at a particular entry within an matrix . Our task is to identify which adjacent entry to this position has the smallest difference relative to our sequence value. If the entry is in an edge of the matrix then the neighbors that are outside the matrix have a difference of infinity. We then accumulate this minimum difference as we proceed from one entry to the next.

    The algorithm examines each entry in the matrix and chooses the one with the smallest cumulative difference. At every step, the difference for the current entry is determined by adding it to the minimum difference from the adjacent entries. This process is repeated for every entry, ensuring the algorithm considers all possible positions and directions. As a result, the algorithm guarantees that it finds the minimum possible difference for the entire matrix by exhaustively checking each entry and all paths stemming from it.

    This method is effective because it systematically explores all entries and their neighbors, ensuring no possibility is overlooked. It's the same as a thorough search where each step is optimized by choosing the most promising direction, leading to an overall minimum difference for the sequence and matrix combination.

2. What recurrence can you use to model this problem using dynamic programming?

Let $f(n,\ m,\ z)$ be the function that calculates the minimum sum difference between the sequence $string$ and the values in the $matrix$, starting at position $(n,\ m)$ in the matrix and considering the $z$-th element in the sequence $string$. The function is defined as follows:

$$f(n,m,z) = \begin{cases} 0 & \text{if } z \geq \text{size of } string, \\ \infty & \text{if } n \geq \text{rows or } m \geq \text{columns or } n < 0 \text{ or } m < 0, \\ |matrix[n][m] - string[z]| + \min & \text{otherwise} \\ \quad \begin{cases} f(n-1,m,z+1), & \text{go down} \\ f(n+1,m,z+1), & \text{go up} \\ f(n,m-1,z+1), & \text{go left} \\ f(n,m+1,z+1) & \text{go right} \end{cases} & \end{cases}$$

The algorithm will call the recurrence $f$ for every entry in the matrix.

3. What are the base cases of this recurrence?

There are two base cases. The first stops the function from proceeding if it reaches the last element of the sequence or string. The second stops the function from proceeding if the current entry is in an edge of the matrix.

4. Give pseudocode for a memoized dynamic programming algorithm for the problem of identifying the minimum possible alignment difference

**Algorithm 3:** Function f

```
Input: Integers n, m, z
Input: Integer Sequence s
Input: Matrix matrix
Input: DP Table dp
Output: Minimum absolute difference for entry (n, m) starting with index z in s
if z ≥ size of s then
    return 0                                          // Base case: reached end of sequence
if n ≥ number of rows in matrix ∨ m ≥ number of columns in matrix ∨ n < 0 ∨ m < 0 then
    return ∞ // Base case: out of matrix bounds
if dp[n][m][z] ≠ −1 then
    return dp[n][m][z]                                // Subproblem already solved
dp[n][m][z] ← abs(matrix[n][m] − s[z]) + min{
    f(n − 1, m, z + 1, s, matrix, dp),                                  // Go down
    f(n + 1, m, z + 1, s, matrix, dp),                                  // Go up
    f(n, m − 1, z + 1, s, matrix, dp),                                  // Go left
    f(n, m + 1, z + 1, s, matrix, dp)                                   // Go right
}
return dp[n][m][z]
```

**Algorithm 4:** Memoization Wrapper

```
Input: Sequence s
Input: Matrix matrix
Output: Minimum absolute difference for all entries
Initialize dp as a 3D vector with dimensions [matrix.size()][matrix.front().size()][s.size()],
  filled with -1
min ← ∞
for i ← 0 to matrix.size() − 1 do
    for j ← 0 to matrix.front().size() − 1 do
        min ← min{f(i, j, 0, s, matrix, dp), min}                  // Find min for each entry
return min
```

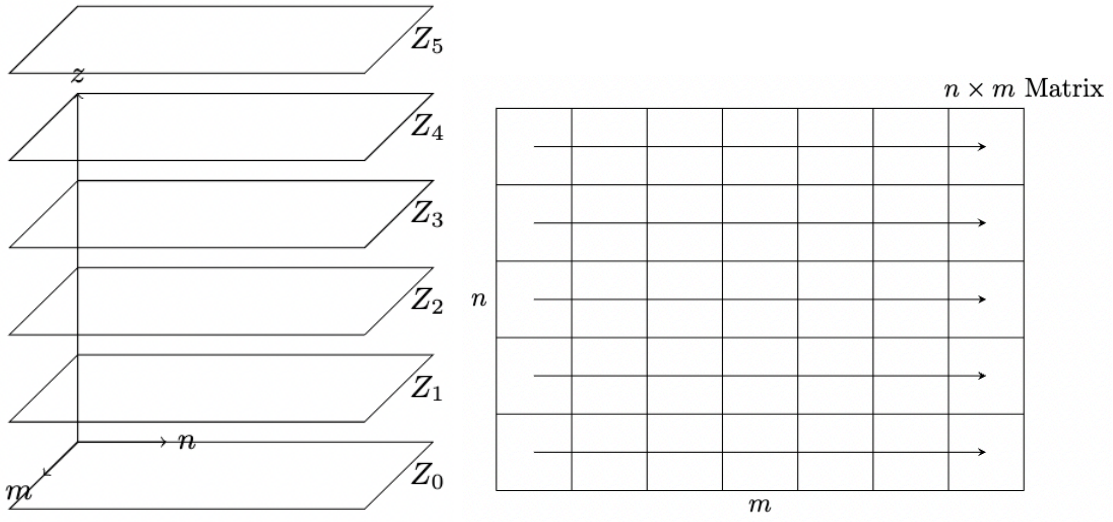
5. What is the time complexity of your memoized algorithm? Justify your answer.

The dynamic programming table, which has three dimensions, can be used to calculate the complexity of the memoized algorithm. The dimensions of the dynamic programming table are $matrix.row * matrix.columns * string.size$. Since we visit each entry of the table one time, the worst-time complexity of the memoized algorithm would be $O(rows * columns * size)$ or $O(n * m * z)$, where $n$ is the number of rows, $m$ is the number of columns, and $z$ is the size of the string.

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (f(i,j,0)) = n * m * k = O(n * m * k).$$

6. Describe a valid iteration order you would recommend for an iterative algorithm for this problem in pseudocode.

A valid iteration order would be to start from the first layer and move upwards. Inside each layer, any iteration order is valid. However, we can move left to right and then to the next row.




**Input:** A 3D DP table with dimensions z-layers, n-rows, and m-columns

```
1 for layer ← 0 to z − 1 do
      // Iterate through each layer from bottom to top
2     for row ← 0 to n − 1 do
          // Iterate through each row of the current layer
3         for column ← 0 to m − 1 do
              // Iterate through each column of the current row from left to right
4             UpdateDPTable(layer, row, column) // Perform the required operation
```

7. Give pseudocode for an iterative algorithm for the problem

The algorithm works by iterating through the layers starting at the penultimate. It then traverses the matrices starting at the top left, moving rightward, and down the rows. The algorithm calculates the minimal difference for the entry by finding the neighbor with the smallest difference and adding it to the difference of the current entry. The algorithm searches for the neighbor's smallest difference by looking at the layer below the current and selecting the smallest difference among the four neighbors. After the algorithm iterates through all the layers, it searches for the smallest difference. The algorithm finds the smallest difference by iterating through the last layers, the topmost one, and selecting the smallest value there.

**Algorithm 5:** Iterative

```
Input: Sequence s
Input: Matrix matrix
Output: Minimum sum difference for the sequence and matrix
Initialize dp as a 3D vector with dimensions [size of s + 1] × [number of rows in matrix] ×
  [number of columns in matrix], filled with 0
for layer ← 1 to size of dp do
    for row ← 0 to number of rows in matrix do
        for column ← 0 to number of columns in matrix do
            dp[layer][row][column] ← |matrix[row][column] - s[size of  s - layer]|
              + min{
                if row + 1 < number of rows in matrix then  dp[layer − 1][row + 1][column]
                else  ∞
                , if row > 0 then  dp[layer − 1][row − 1][column]
                else  ∞
                , if column + 1 < number of columns in matrix then
                  dp[layer − 1][row][column + 1]
                else  ∞
                , if column > 0 then  dp[layer − 1][row][column − 1]
                else  ∞
            }
min ← ∞
for i ← 0 to number of rows in matrix do
    for j ← 0 to number of columns in matrix do
        min ← min(min, dp[front][i][j])
return min
```

8. How could you modify your solution to the problem of identifying the minimum absolute difference to actually locate the alignment in the grid? You may answer this question by giving pseudocode for an algorithm that computes and returns the alignment, or you may describe (in English) how to modify the iterative or memoized algorithm.

We can backtrack through the dynamic programming table to find the path or alignment. Therefore, I simply need to add some code to the iterative algorithm to find the path. We start at the layer where the solution is located, which is at the topmost layer of the table. We then locate the smallest value and move down a layer to the same n and m position in the matrix. In this layer, we search for the neighbor with the smallest difference and move to it. We then can output or store the neighbor's direction relative to the previous one, like 'L,' if the neighbor is left to the previous entry. We can do this repeatedly until we reach the penultimate layer. At the end we have a string that contains the direction for the minimal difference.

**Algorithm 7:** FindPath

```
Input: DP table dp, Path array path, Matrix matrix
Output: Path string path_s
path_s ← empty string
Initialize coordinate to store four directions' next step coordinates
for each step i from 1 to dp.size − 2 do
    Initialize up, down, left, right to maximum integer value
    // Check possible moves from the current position
    Update up, down, left, right with values from dp if moves are valid
    Update coordinate with corresponding positions
    // Choose the direction with the minimum value
    smallest ← minimum of up, down, left, right
    // Update the path and path string
    if up is smallest then
        Add 'U' to path_s and update path[i]
    else if down is smallest then
        Add 'D' to path_s and update path[i]
    else if left is smallest then
        Add 'L' to path_s and update path[i]
    else
        Add 'R' to path_s and update path[i]
return path_s                          // Return the constructed path string
```

9. Can the space complexity of your iterative solution for finding the minimum score or alignment be reduced? Why or why not?

Yes, the space complexity of the algorithm can be reduced. We can reduce it by simply using two layers instead of $(string.size + 1)$. The reason why is that we only need the previous layer to calculate the current one. Hence, we can iterate in between the two layers to get the answer.

However, if we reduce the space complexity, we will not be able to determine the problem's path or alignment.

**Algorithm 1:** IterativeWithReducedSpace

**Input:** Sequence $s$ of integers
**Input:** Matrix *matrix* of integers
**Output:** Minimum sum difference between elements of $s$ and *matrix*

```
min ← ∞
Initialize dp as a 3D vector with dimensions [2] × [number of rows in matrix] × [number of
  columns in matrix], filled with 0s
for i ← 1 to length of s do
    for j ← 0 to number of rows in matrix do
        for k ← 0 to number of columns in matrix do
            dp[i mod 2][j][k] ← |matrix[j][k] − s[length of s − i]| + min{
                if j + 1 < number of rows in matrix then  dp[(i - 1) mod 2][j + 1][k]
                else  ∞
                , if j > 0 then  dp[(i - 1) mod 2][j- 1][k]
                else  ∞
                , if k + 1 < number of columns in matrix then  dp[(i - 1) mod 2][j][k + 1]
                else  ∞
                , if k > 0 then  dp[(i  - 1) mod 2][j][k - 1]
                else  ∞
            }
    for row ← 0 to number of rows in matrix do
        for col ← 0 to number of columns in matrix do
            min ← min(min, dp[(i - 1) mod 2][row][col])
return min
```