Algorithms Programming Project II
Dynamic Programming

March 2023

1 Team Members

Haolan Xu: UFID-34326768
Yujie Wang: UFID-59913548

2 Design and Analysis of Algorithms

In our project, our outputs starting from 1 instead of 0, which means we are 1-base instead of 0-base.

2.1 Problem 1

2.1.1 ALG 1

Analysis of algorithm:

The method is:

1.Read the input dimensions m and n, and the minimum number of trees h.

2.Read the input matrix p, and convert each element to 1 if the number of trees is greater than or equal to h, otherwise keep it as 0.

3.Initialize the maximum side length (ans) of the square to be found as 0, and initialize the upper left corner (ansx, ansy) of the square to (1, 1).

4.Iterate through each cell (i, j) in the matrix, considering it as the upper left corner of the potential square.

5.For each cell, calculate the side length l and check if the square with side length l is within the grid boundaries. If yes, check if all the cells within the square have a value of 1 (i.e., all cells require at least h trees).

6.If a valid square is found, update the maximum side length (ans) and the upper left corner (ansx, ansy) of the square.

7.Print the upper left corner and lower right corner of the square with the maximum side length.

The algorithm checks every possible square within the grid to find the largest one that satisfies the given condition. It iterates through all cells in the matrix, considering them as the upper left corner of potential squares, and explores all possible side lengths to find the largest square.

And the **time complexity** is $\Theta(m * n * min(m, n)^3)$. There are m * n cells in the matrix. For each cell (i, j), the algorithm checks all possible square side lengths. In the worst case, this step contributes a factor of min(m, n). For each side length, the algorithm checks whether all cells within the square have a value of 1, which takes $\Theta(min(m, n)^2)$ time. So, the time complexity is $\Theta(m * n * min(m, n) * min(m, n)^2) = \Theta(m * n * min(m, n)^3)$.

The example of algorithm:

Consider the following input:

Table 1: ALG 1 Example Parameter

m	n	h
5	5	2

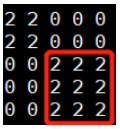

Figure 1: ALG1 Input Matrix P

After ALG1 algorithm, we can get the result that the max size is 3, the upper left corner is (3,3), the lower right corner is (5,5).

```
[wangyuj@flip3 ~/test2]$ g++ Task1.cpp
[wangyuj@flip3 ~/test2]$ ./a.out
5 5 2
2 2 0 0 0
2 2 0 0 0
0 0 2 2 2
0 0 2 2 2
0 0 2 2 2
3 3 5 5
```

Figure 2: ALG1 Output

2.1.2 ALG 2

Analysis of algorithm:

The method is:

1.Read the input dimensions m and n, and the minimum number of trees h.

2.Read the input matrix p, and convert each element to 1 if the number of trees is greater than or equal to h, otherwise keep it as 0.

3.Initialize the maximum side length (ans) of the square to be found as 0, and initialize the upper left corner (ansx, ansy) of the square to (1, 1).

4.Iterate through each cell (i, j) in the matrix, considering it as the upper left corner of the potential square.

5.For each cell, calculate the side length l and check if the square with side length l is within the grid boundaries. If yes, check if all the cells in **the new right and lower bounds** of the square have a value of 1 (i.e., all cells require at least h trees).

6.If a valid square is found, update the maximum side length (ans) and the upper left corner (ansx, ansy) of the square.

7.Print the upper left corner and lower right corner of the square with the maximum side length.

The algorithm checks every possible square within the grid to find the largest one that satisfies the given condition. It iterates through all cells in the matrix, considering them as the upper left corner of potential squares, and explores all possible side lengths to find the largest square. **It optimizes the search by only checking the new right and lower bounds for each increment in side length.**

And the **time complexity** is $\Theta(m * n * min(m, n)^2)$. There are m * n cells in the matrix, and for each cell, the algorithm checks all possible side lengths, which is min(m, n) in the worst case. For each side length, the algorithm checks only the new right and lower bounds of the square, which takes O(min(m, n)) time. Therefore, the overall time complexity is $\Theta(m*n*min(m, n)*min(m, n))$ = $\Theta(m * n * min(m, n)^2)$.

The example of algorithm:

Consider the following input:

Table 2: ALG 2 Example Parameter

m	n	h
5	5	2

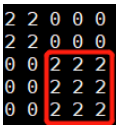

Figure 3: ALG2 Input Matrix P

After ALG2 algorithm, we can get the result that the max size is 3, the upper left corner is (3,3), the lower right corner is (5,5).

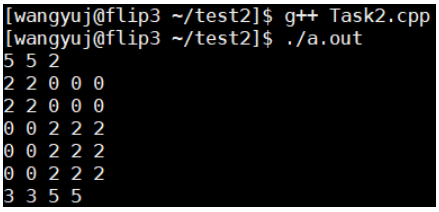

Figure 4: ALG2 Output

2.1.3 ALG 3

Analysis of algorithm:

The method is:

1.Read the dimensions of the grid (m, n), the minimum number of trees required (h), and the matrix p representing the minimum number of trees that must be planted on each plot.

2.Create a new m * n matrix p and fill it with values based on the input matrix. If the value in the input matrix is greater than or equal to h, set the corresponding value in the new matrix p to 1, otherwise set it to 0. This step converts the problem into finding the largest square with all 1s.

3.Create another m * n matrix dp. This matrix will be used to store the largest square size ending at each position (i, j) in the grid.

4.Iterate over the grid using two nested loops for i and j (from 1 to m and 1 to n, respectively). For each position (i, j), update the dp matrix using the following rule: dp[i][j] = p[i][j] ? min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1 : 0. If the value in the p matrix is 1, set the dp matrix value to the minimum of the adjacent positions' values (above, left, and diagonally above-left) plus 1. Otherwise, set the dp matrix value to 0.

5.Keep track of the largest square size found so far (ans) and its ending position (bottom-right corner) in the grid (ansx, ansy). Update these values whenever a larger square is found during the iteration.

6.After the entire grid has been iterated over, calculate the bounding indices of the largest square by subtracting the largest square size from the ending position coordinates.

7.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1, if p[i][j] >= h

dp[i][j] = 0, otherwise

This relation states that if p[i][j] is 1 (meaning the plot has at least h trees), then dp[i][j] is equal to the minimum of the squares ending at the adjacent positions (left, above, and diagonally above-left) plus 1. Otherwise, dp[i][j] is set to 0.

The dynamic programming approach used in the code iteratively calculates the largest square ending at each position (i, j) and updates the maximum size and its position accordingly. Since the code iterates over the entire grid, it is guaranteed to find the largest square with the required property.

The time complexity of the code is $\Theta(m * n)$ because it iterates over the entire grid once while calculating and updating the dp matrix. The nested loops for i and j run m and n times, respectively.

The example of algorithm:

Consider the following input:

Table 3: ALG 3 Example Parameter

m	n	h
5	5	2

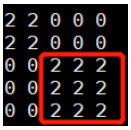

Figure 5: ALG3 Input Matrix P

After ALG3 algorithm, we can get the result that the max size is 3, the upper left corner is (3,3), the lower right corner is (5,5).

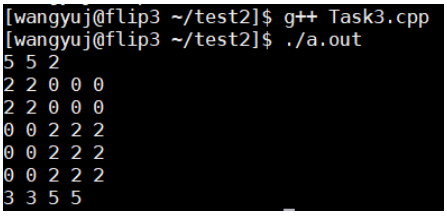

Figure 6: ALG3 Output

2.2 Problem 2

2.2.1 ALG 4

Analysis of algorithm:

The method is:

1.Read the dimensions of the grid (m, n), the minimum number of trees required h, and the grid values representing the minimum number of trees that must be planted on each plot.

2.Convert each value in the input matrix p[i][j] to 1 if the value is greater or equal to h, otherwise set it to 0. This simplifies the problem by finding the maximum square with all 1s except corner plots.

3.Initialize a new matrix dp with dimensions m * n. For each plot (i, j) in the grid, set dp[i][j] to the number of consecutive plots with value 1 in the same column ending at the plot (i, j). This is done using the recursive formulation described earlier.

4.Initialize variables ans, ansx, and ansy to store the maximum side length of the square found and the lower left corner indices.

5.Iterate through all possible lower left corners of potential squares, represented by (i, l). For each lower left corner: **a**.Iterate through all possible lower right corners of potential squares, represented by (i, r). **b**.Calculate the maximum edge length possible for the current lower left and lower right corners using the dp matrix and the current value of r and l. **c**.If the calculated maximum edge length is greater than the current ans value, update ans, ansx, and ansy to store the new maximum side length and the corresponding lower left corner indices.

6.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

dp[i][j] = dp[i-1][j] + 1, if p[i][j] >= h

dp[i][j] = 0, otherwise

The base case for the dynamic programming formulation in this code is when i = 1, which represents the first row of the grid. For the first row, we don't have any previous row to refer to, so we directly initialize the dp array based on the values of the p array:

dp[1][j] = p[1][j], for all j = 1, 2, ..., n.

The algorithm calculates the dp matrix using the given recursive formulation, considering each plot (i, j) in the grid. It iterates through all the lower left corners of potential squares, represented by (i, l), and then iterates through all the lower right corners, represented by (i, r). By calculating

the minimum edge length possible at each step, the algorithm finds the largest square with the required condition.

And the **time complexity** is $\Theta(m * n^2)$. Initializing and filling the matrix p takes $\Theta(m * n)$ time. Filling the dp matrix takes $\Theta(m * n)$ time. In the worst case, there are three nested loops: one iterating through rows (O(m)), one iterating through the lower left corner's column (O(n)), and one iterating through the lower right corner's column (O(n)). This results in a time complexity of $O(m * n^2)$. Therefore, the overall time complexity is $\Theta(m * n^2)$.

The example of algorithm:

Consider the following input:

Table 4: ALG 4 Example Parameter

m	n	h
7	7	2

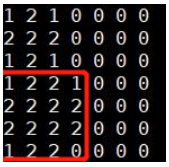

Figure 7: ALG4 Input Matrix P

After ALG4 algorithm, we can get the result that the max size is 4, the upper left corner is (4,1), the lower right corner is (7,4).

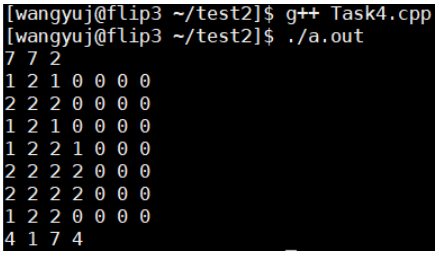

Figure 8: ALG4 Output

2.2.2 ALG 5

Memoization:

Analysis of algorithm:

The method is:

1.Read the dimensions (m and n), minimum number of trees (h), and the values of the minimum number of trees for each plot (p[i][j]).

2.Convert the grid p[i][j] to a binary grid, where 1 indicates that p[i][j] >= h and 0 otherwise.

3.Initialize the base cases for the DP function. If a plot is on the top or left border, set its DP value to 1.

4.Define the DP function to calculate the size of the largest square that has its lower right corner at position (x, y) and satisfies the problem constraints. The DP function uses memoization to store intermediate results for efficient computation.

5.Iterate over all positions (i, j) in the grid (excluding the top and left borders). For each position, call the DP function and update the answer (largest square size found so far) if the current position has a larger square size.

6.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

The recursive formulation is expressed through the function DP(x, y), which returns the size of the largest square that has its lower right corner at (x, y) and satisfies the problem's constraints. The formulation for DP(x, y) is as follows:

DP(x, y) = 1, if x = 1 or y = 1

DP(x, y) = 2, if p[x][y-1] = 0 or p[x-1][y] = 0 or p[x-L][y-L+1] = 0 or p[x-L+1][y-L] = 0

DP(x, y) = L + 1, otherwise

Where L = min(DP(x-1, y), DP(x, y-1), DP(x-1, y-1)).

The base cases handle the upper and left border plots, which must have a square with a length of 1. The recursive cases determine the optimal substructure by taking the minimum of the three previously solved subproblems and adding 1.

The algorithm is correct because it iteratively builds the solution based on the optimal substructure property. The memoization ensures that overlapping subproblems are not recomputed, leading to an efficient solution. By keeping track of the largest square found so far and its lower right corner, the algorithm can identify the largest square satisfying the constraints.

And the **time complexity** is $\Theta(m * n)$. Because each cell in the grid is visited once, and the memoization ensures that overlapping subproblems are not recomputed.

The example of algorithm:

Consider the following input:

Table 5: ALG 5A Example Parameter

m	n	h
9	9	2

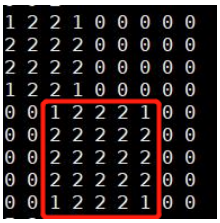

Figure 9: ALG5A Input Matrix P

After ALG5A algorithm, we can get the result that the max size is 5, the upper left corner is (5,3), the lower right corner is (9,7).

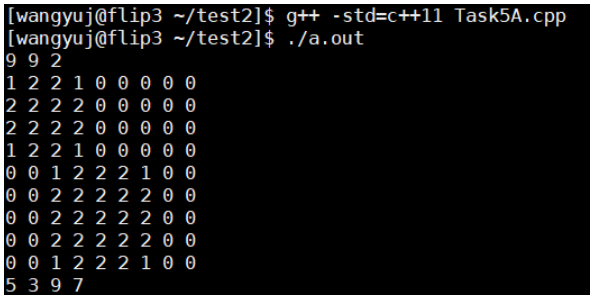

Figure 10: ALG5A Output

BottomUp:

Analysis of algorithm:

The method is:

1.Read the dimensions (m and n), minimum number of trees (h), and the values of the minimum number of trees for each plot (p[i][j]).

2.Convert the grid p[i][j] to a binary grid, where 1 indicates that p[i][j] >= h and 0 otherwise.

3.Initialize the base cases for the dp array. If a plot is on the top or left border, set its dp value to 1.

4.Iterate over all positions (i, j) in the grid (excluding the top and left borders). For each position, calculate the size of the largest square that has its lower right corner at position (i, j) and satisfies the problem constraints. Update the dp array with the calculated size.

5.Keep track of the answer (largest square size found so far) and its lower right corner (ansx, ansy) during the iteration.

6.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

dp[i][j] = 1, if i = 1 or j = 1

dp[i][j] = 2, if p[i][j-1] = 0 or p[i-1][j] = 0 or p[i-L+1][j-L] = 0 or p[i-L][j-L+1] = 0

dp[i][j] = L + 1, otherwise

Where L is min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

The correctness of the algorithm comes from the fact that it starts from the base cases (plots on the top and left borders) and builds the solution for larger squares based on the optimal solutions of smaller overlapping subproblems (squares). The code iterates over all possible squares and ensures that all the plots enclosed within the square, except the corners, meet the minimum requirement of h trees, and the corners can have any number of trees.

And the **time complexity** is $\Theta(m * n)$. Because each cell in the grid is visited once.

The example of algorithm:

Consider the following input:

Table 6: ALG 5B Example Parameter

m	n	h
9	9	2

Figure 11: ALG5B Input Matrix P

After ALG5B algorithm, we can get the result that the max size is 5, the upper left corner is (5,3), the lower right corner is (9,7).

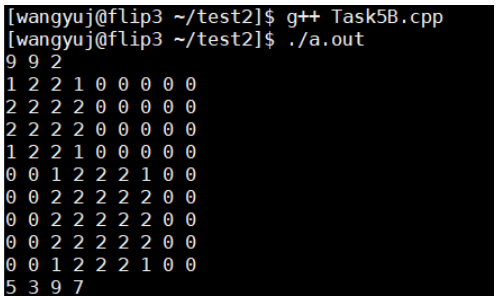

Figure 12: ALG5B Output

Difference between Memoization and BottomUp

In **Memoization**: Use an unordered_map called M to store the DP values. And use a macro #define dp(a, b) M[a * N + b] to access the DP values. This is a form of memoization. The DP function is called int DP(int x, int y) and it is a **top-down** approach.

In **BottomUp**: Use a 2D array called dp[N][N] to store the DP values. Access the DP values directly through array indices. The DP values are updated directly within the main function, which is a **bottom-up** approach.

2.3 Problem 3

2.3.1 ALG 6

Analysis of algorithm:

The method is:

1.Read input values for m, n, h, and k.

2.Initialize the grid with binary values: 1 if the number of required trees is greater than or equal to h, and 0 otherwise.

3.Initialize variables to store the answer, ans (maximum side length), ansx, and ansy (coordinates of the upper left corner of the square).

4.Loop through all possible upper left corners (i, j) of the square.

5.Loop through all possible side lengths l, starting from ans+1 (to find a larger square).

6.Calculate the number of enclosed plots with less than h trees (cnt) for each square.

7.If cnt <= k, update ans, ansx, and ansy.

8.Print the upper left corner and lower right corner of the square with the maximum side length.

The code exhaustively searches all possible square areas within the grid, considering all possible side lengths and upper left corner positions. It ensures the number of enclosed plots with less than h trees is less than or equal to k. Therefore, the algorithm is correct as it explores all possibilities.

And the **time complexity** is $\Theta(m * n * min(m, n)^3)$. Reading the input grid takes $\Theta(m * n)$ time. The nested loops iterate over all possible upper left corners and side lengths, which can take $\Theta(m * n * min(m, n))$ time. The innermost loop to count plots with less than h trees takes $\Theta(l^2)$ time, where l is the side length. In the worst case, l = min(m, n), so this loop takes $\Theta(min(m, n)^2)$ time. Overall, the time complexity of the code is $\Theta(m * n * min(m, n)^3)$.

The example of algorithm:

Consider the following input:

Table 7: ALG 6 Example Parameter

m	n	h	k
5	5	2	3

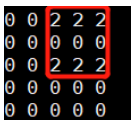

Figure 13: ALG6 Input Matrix P

After ALG6 algorithm, we can get the result that the max size is 3, the upper left corner is (1,3), the lower right corner is (3,5).

```
[wangyuj@flip3 ~/test2]$ g++ Task6.cpp
[wangyuj@flip3 ~/test2]$ ./a.out
5 5 2 3
0 0 2 2 2
0 0 0 0 0
0 0 2 2 2
0 0 0 0 0
0 0 0 0 0
1 3 3 5
```

Figure 14: ALG6 Output

2.3.2 ALG 7

Memoization:

Analysis of algorithm:

The method is:

1.Read input values for m, n, h, and k.

2.Initialize the grid with binary values: 1 if the number of required trees is less than h, and 0 otherwise.

3.Calculate the prefix sum of the grid to efficiently compute the sum of values in a given range.

4.Initialize memoization tables L and U for storing the last 1 on the left and upper sides, respectively.

5.Define the DP function to calculate the side length of the largest square ending at (x, y) and with at most K plots having less than h trees.

6.The DP function computes the result based on the values in the memoization tables L and U, and the DP values for (x-1, y-1).

7.Iterate through all possible lower right corners (i, j) and all possible values of l (0 to k) to find the largest square.

8.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

DP(x, y, K) = max(DP(x-1, y-1, i)+1 for all i which 0<=i<=K, x>0, y>0)

Here, DP(x, y, K) denotes the side length of the largest square ending at (x, y) and with at most K plots having less than h trees. The function iterates through all possible values of i from 0 to K and finds the maximum square size by using the DP values of the previous (x-1, y-1) position.

The code uses dynamic programming to explore all possible squares in the grid while keeping track of the number of plots with less than h trees. The memoization tables L and U enable the efficient calculation of the DP values. Therefore, the algorithm is correct as it systematically explores all possibilities while avoiding redundant calculations.

And the **time complexity** is $\Theta(m * n * k)$. Reading the input grid takes $\Theta(m * n)$ time. Computing the prefix sum of the grid takes $\Theta(m * n)$ time. Filling the memoization tables L and U takes O(m * n) time in the worst case. The DP function is called $\Theta(m * n * k)$ times, and its complexity is O(1) with memoization. Overall, the time complexity of the code is $\Theta(m * n * k)$.

The example of algorithm:

Consider the following input:

Table 8: ALG 7A Example Parameter

m	n	h	k
5	5	2	3

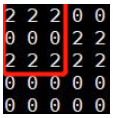

Figure 15: ALG7A Input Matrix P

After ALG7A algorithm, we can get the result that the max size is 3, the upper left corner is (1,1), the lower right corner is (3,3).

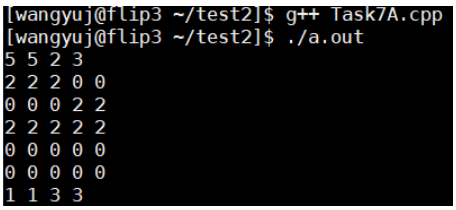

Figure 16: ALG7A Output

BottomUp:

Analysis of algorithm:

The method is:

1.Read the dimensions (m, n), the minimum number of trees required (h), and the maximum number of plots with less than h trees allowed (k).

2.Resize and initialize matrices p, L, U, and dp to their required dimensions.

3.Loop through the grid and read the input values for each plot. For each plot at (i, j): **a.**If the number of trees required for the plot is less than h, set p[i][j] to 1, else set it to 0. **b.**Calculate the accumulated sum of plots with less than h trees by updating p[i][j] with the sum of p[i-1][j], p[i][j-1], and p[i][j], and subtracting p[i-1][j-1]. **c.**Update L[i][j+1] and U[i+1][j] based on whether p[i][j] is 1 or 0.

4.Initialize the answer variables ans, ansx, and ansy to store the maximum square length and the coordinates of the lower-right corner of the square.

5.Loop through the grid and for each plot at (i, j): **a.**Loop through all possible values of t (0 to k) and perform the following updates: **i.**Call the update function with i, j, and dp[i-1][j-1][t] + 1. **ii.**Find the nearest left and upper continuous block of plots with less than h trees and call the update function with i, j, and j - L[i][j] or i - U[i][j] accordingly. **b.** Loop through all possible values of l (0 to k), and if dp[i][j][l] ¿ ans, update ans, ansx, and ansy.

6.Print the upper left corner and lower right corner of the square with the maximum side length.

Mathematical Recursive Formulation:

Base case: If x = 0 or y = 0, DP(x, y, K) = 0

DP(x, y, K) = max(DP(x-1, y-1, i)+1 for all i which 0<=i<=K, x>0, y>0)

The correctness of this formulation is based on the optimal substructure property. The solution for the largest square ending at (x, y) can be constructed from the optimal solutions of its smaller subproblems, in this case, the squares ending at (x-1, y-1). By exploring all the possibilities of i and finding the maximum, we are guaranteed to find the optimal solution for the current position (x, y).

The **time complexity** of this solution is $\Theta(m * n * k)$, because it iterates through all the rows (m), columns (n), and maximum number of plots with less than h trees allowed (k). It processes each plot and calculates the DP values and accumulated sums.

The example of algorithm:

Consider the following input:

Table 9: ALG 7B Example Parameter

m	n	h	k
5	5	2	3

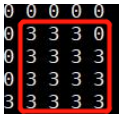

Figure 17: ALG7B Input Matrix P

After ALG7B algorithm, we can get the result that the max size is 4, the upper left corner is (2,2), the lower right corner is (5,5).

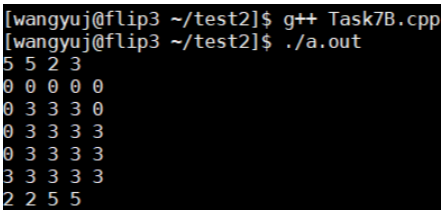

Figure 18: ALG7B Output

Difference between Memoization and BottomUp

In **ALG7A** uses memoization to calculate the LeftOne(int x, int y) and UpperOne(int x, int y) functions, which store intermediate results in the L and U matrices. This approach is top-down since it starts with a high-level problem and breaks it down into subproblems, using memoization to store intermediate results and avoid redundant calculations.

In **ALG7B**, on the other hand, does not use memoization for the LeftOne and UpperOne functions. Instead, it directly updates the L and U matrices during input processing. However, it still uses dynamic programming to fill in the dp matrix, which is a bottom-up approach. In bottom-up dynamic programming, the problem is solved by building up solutions to smaller subproblems first and using these solutions to construct solutions for larger subproblems.

3 Experimental Comparative Study

3.1 The approach

Create a script to generate random input files for various grid sizes (m x n) and tree values (h). Make sure to generate input files that cover a wide range of cases, from small to large grid sizes.

Run our code for each input file and record the running time for each task (Task 1, Task 2, Task 3, etc.). In our project we use a timer to measure the running time.

Then we create two-dimensional plots for each task comparison group (Tasks 1-3, Tasks 4-5B, and Tasks 6-7B). The x-axis should represent the input size (m * n), and the y-axis should represent the running time.

To avoid overshadowing, we create additional plots for tasks with a significant difference in running time.

3.2 The result

3.2.1 Task 1,2,3:

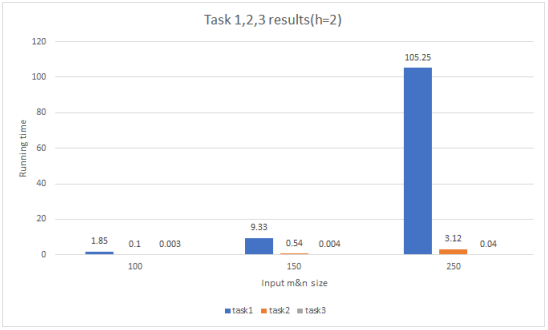

Figure 19: Task 1,2,3 Comparison

3.2.2 Task 4,5A,5B:

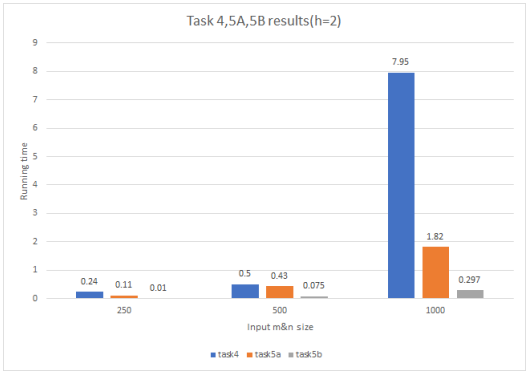

Figure 20: Task 4,5A,5B Comparison

3.2.3 Task 6,7A,7B:

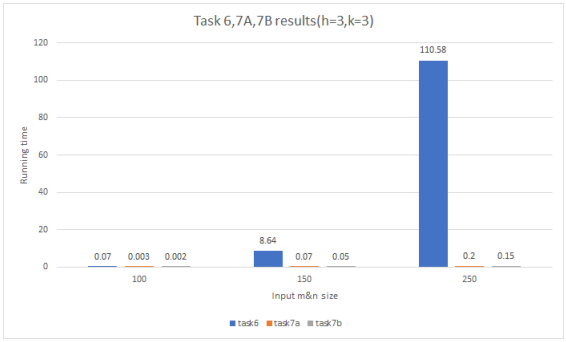

Figure 21: Task 6,7A,7B Comparison

3.2.4 Task 2,3:

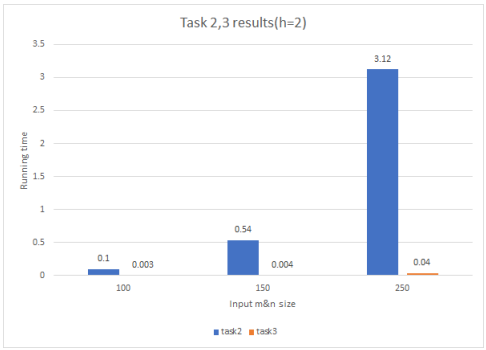

Figure 22: Task 2,3 Comparison

3.2.5 Task 5A,5B:

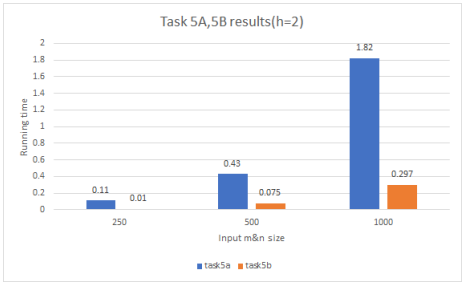

Figure 23: Task 5A,5B Comparison

3.2.6 Task 7A,7B:

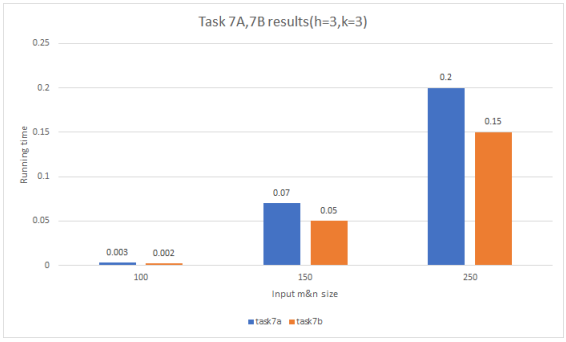

Figure 24: Task 7A,7B Comparison

4 Conclusion

Throughout this project assignment, we've been exposed to various algorithmic concepts and techniques, including dynamic programming, memoization, bottom-up approaches. By working on these tasks, we've gained valuable experience in applying these methods to solve different types of problems.

Regarding ease of implementation, the Brute Force is most straightforward approach to solve the problem. But after comprehending the dynamic programming, we can use dp to solve these problems more efficiently. And in dp, tasks using top-down approaches like memoization might have been easier to implement as they often build on a more intuitive thought process. On the other hand, bottom-up approaches may require more effort in identifying the optimal substructure and carefully designing the loops and indices to fill in the DP table.

Some potential technical challenges faced during the project might include:

Understanding the problem requirements: Clearly comprehending the problem statement and requirements is essential to develop an accurate and efficient solution.

Identifying the correct approach: Choosing the most suitable technique among the various methods like memoization, bottom-up can impact the complexity and efficiency of your solution.

Debugging and testing: Ensuring that the implemented code is correct and efficient can be time-consuming. Identifying edge cases and potential pitfalls requires a deep understanding of the problem and the chosen technique.

Analyzing time complexity: Accurately analyzing the time complexity of our solution is crucial to understanding its performance and making potential improvements.

Generating input files and performance comparison: Creating input files with various sizes and generating plots to compare the performance of different tasks can be a challenge.

Despite these challenges, working on this project has provided us with hands-on experience in applying various algorithmic techniques, developing our problem-solving skills, and deepening our understanding of time complexity analysis.