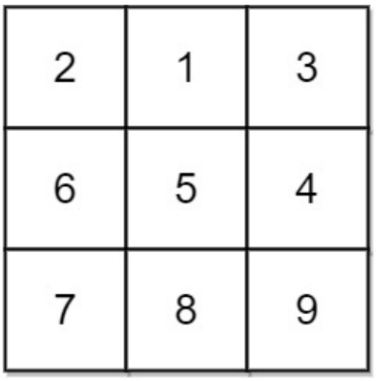
Minimum Falling Path Sum

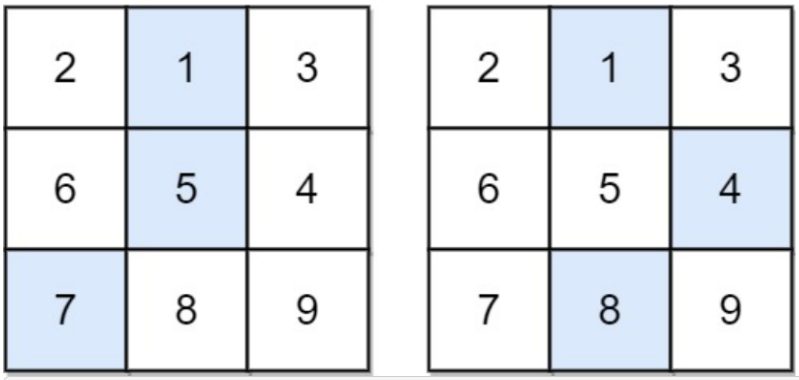
Solution

Given an `n x n` array of integers `matrix`, return *the* ***minimum sum*** *of any* ***falling path*** *through* `matrix`.

A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position `(row, col)` will be `(row + 1, col - 1)`, `(row + 1, col)`, or `(row + 1, col + 1)`.

Example 1:

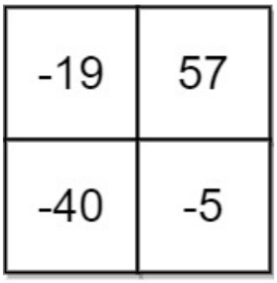


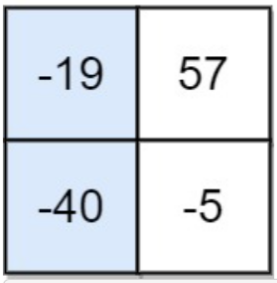



```
Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]
Output: 13
Explanation: There are two falling paths with a minimum sum as shown.
```

Example 2:

Example 2:






```
Input: matrix = [[-19,57],[-40,-5]]
Output: -59
Explanation: The falling path with a minimum sum is shown.
```

Constraints:

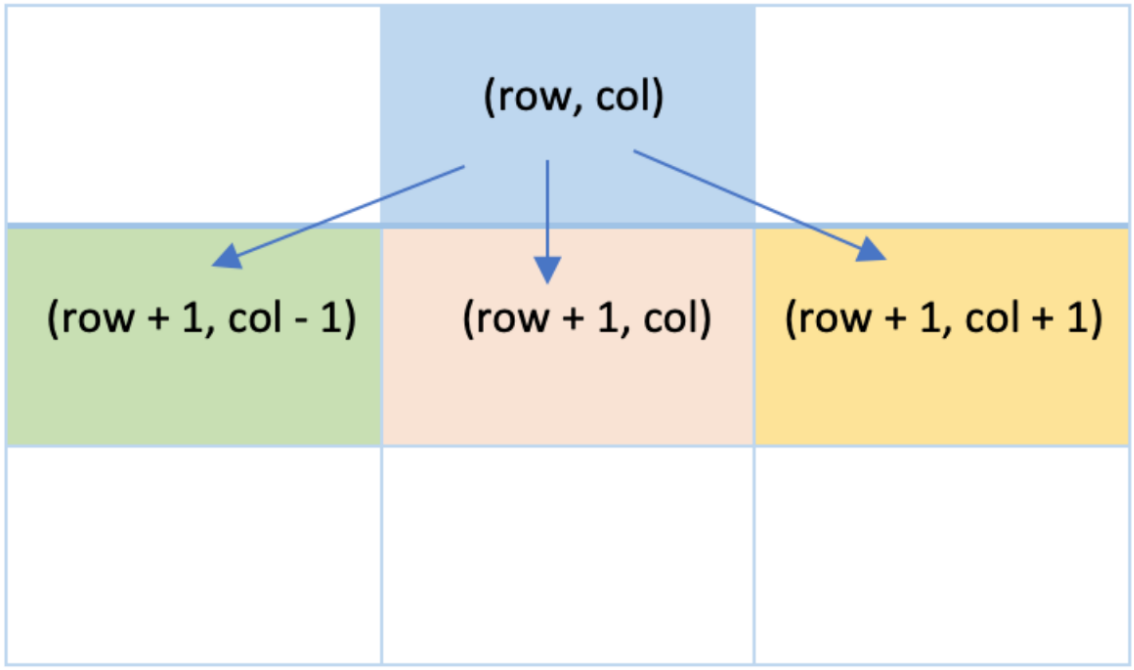
- `n == matrix.length == matrix[i].length`
- `1 <= n <= 100`
- `-100 <= matrix[i][j] <= 100`

Solution

Overview

Given a 2D `matrix(row, col)`, we have to find the sum of the minimum falling path in a matrix. To begin with, let's try to understand, what is a falling path? To put it in simple words, it is a path that satisfies the following criteria,

1. A falling path is a path that begins at **any** cell in the first row of the matrix and ends at **any** cell in the last row of the matrix.
2. From a certain cell `(row, col)` in the falling path, we can only move to 3 possible cells, `(row + 1, col)`, `(row + 1, col + 1)`, `(row + 1, col - 1)`.



As the name suggests, the falling path sum is the sum of values of all the cells in the chosen path. Our goal is to find the minimum sum from all possible paths. Let's consider different approaches that can be used to solve this problem. We will begin with the brute force approach and optimize it using dynamic programming.

Approach 1: Brute Force Using Depth First Search

Intuition

Brute Force is generally the straightforward approach to solving the problem. You can think of the brute force approach as

"Given pen and paper in your hand, how will you try to solve the problem"?

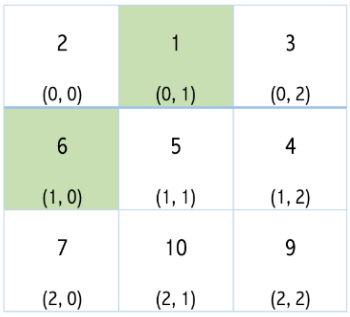
Though the brute force approach is considered to be the most exhaustive and unoptimized approach, implementing this approach first gives a deeper understanding of the basis of the problem, which can further help to shape a better-optimized solution.

Let's understand the brute force approach with the following example,

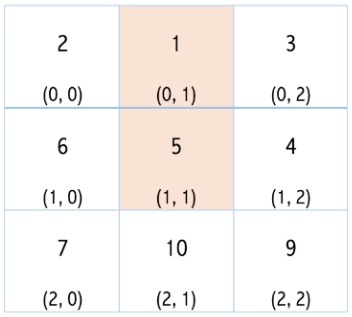
2 (0, 0)	1 (0, 1)	3 (0, 2)
6 (1, 0)	5 (1, 1)	4 (1, 2)
7 (2, 0)	10 (2, 1)	9 (2, 2)

> Before reading the following section, try to find the minimum falling path for the above matrix on your own.

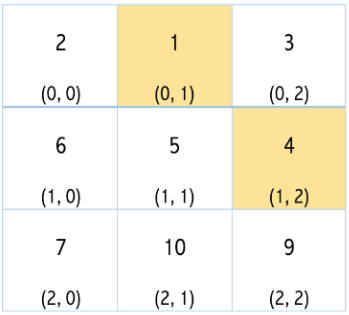
In this example, if we start at cell `(0, 1)`, the next cell in the path could be `(1, 0)`, `(1, 1)`, or `(1, 2)`. From the current position, we cannot determine which path will give us the minimum path sum, so let's try all the possible paths.



Current Path Sum = (1 + 6) = 7



Current Path Sum = (1 + 5) = 6



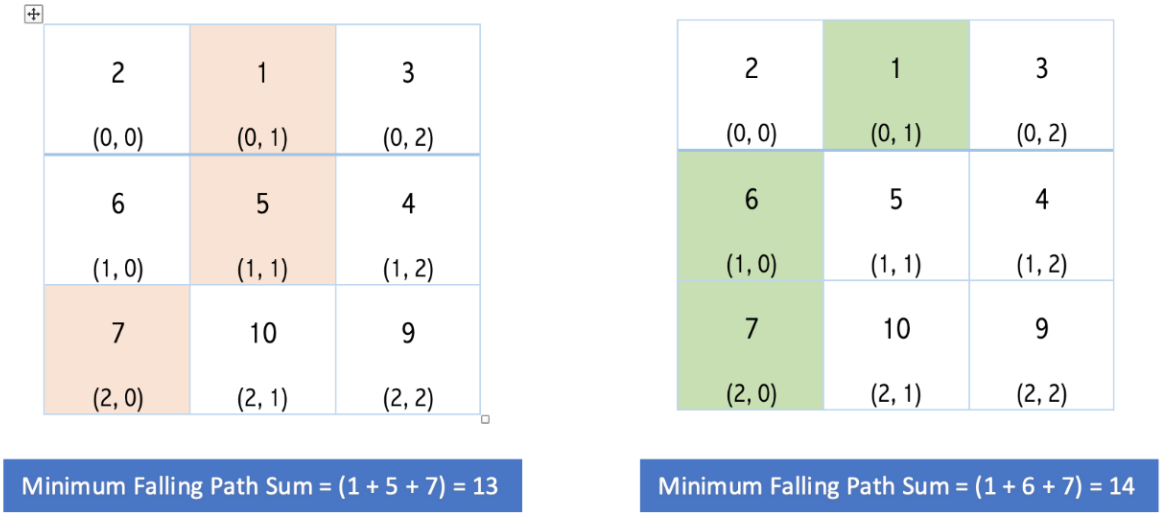
Current Path Sum = (1 + 4) = 5

The minimum path sum from cell `(0, 1)` is "5" - going from `(0, 1)` to `(1, 2)`. Does that mean that we can greedily choose this path and move ahead, discarding the other paths? Let's find the minimum falling path if we choose the path with the sum "5" i.e `(0, 1)` to `(1, 2)`.

2 (0, 0)	1 (0, 1)	3 (0, 2)
6 (1, 0)	5 (1, 1)	4 (1, 2)
7 (2, 0)	10 (2, 1)	9 (2, 2)

Minimum Falling Path Sum = (1 + 4 + 9) = 14

Thus, the minimum falling path sum is "14". Now, let's explore other paths starting from cell `(0, 1)`.



The minimum falling path sum is now "13" i.e `(0, 1)` -> `(1, 1)` -> `(2, 0)`.

This proves that the greedy approach will not work for this problem. A sub-path with the lowest subset-sum cannot guarantee the lowest sum for the entire path. We must explore all possible paths to find the one with the smallest sum.

Since we have to try all paths from any cell, the solution must generate all the possible falling paths and track the one with a minimum sum. To try all possible combinations, we can perform a depth-first search on the matrix.

> The Depth First Search is an exhaustive search process wherein we will traverse all the cells in a path. On reaching the end of the path, we must undo our last step in the current path and try a different possible next step to extend the path.

Note: This approach is a brute force solution, and it is not expected to pass all test cases. This approach is included because, often, an intuitive way to approach problems like this one is to start by building a brute force solution and then modify it to achieve an optimized solution.

Algorithm

1. Implement a Depth First Search algorithm, by defining a recursive function, `findMinFallingPathSum(row, col)`, that recursively explores all the paths from the current cell (defined by parameters `row` and `col`).
    - Define Base Case: In any recursive function, we must define the terminating condition i.e the base case. When the terminating condition is satisfied, we will exit the recursive search process. The base cases are as follows,
        - The `row` or `col` values are not within the matrix boundaries.
        - We have reached the last row. In this case, we will return the value of the current cell and not make any other recursive calls.
    - *Recursively explore all paths:* If the base case is not satisfied, it means that we have not reached the end of our current path, and we must try all options to extend our path and find the one with the minimum sum:

```
minimumPath = Minimum(findMinFallingPathSum(row + 1, col + 1),
                      findMinFallingPathSum(row + 1, col),
                      findMinFallingPathSum(row + 1, col - 1))
```

2. Now that we have defined the recursive function, we must find the minimum falling path for all possible starting cells. A starting cell is any cell in the top row.

   For this, we have to iterate using a *for* loop and find the minimum falling path for cell in 0^{th} row and columns ranging from 0 to $\text{matrix.length} - 1$. Define a variable `minFallingSum` to track the minimum of all the falling paths found so far and return the result.

Implementation

C++:

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int minFallingSum = INT_MAX;
        for (int startCol = 0; startCol < matrix.size(); startCol++) {
            minFallingSum =
                min(minFallingSum, findMinFallingPathSum(matrix, 0, startCol));
        }
        return minFallingSum;
    }

    int findMinFallingPathSum(vector<vector<int>>& matrix, int row, int col) {
        // check if we are out of the left or right boundary of the matrix
        if (col < 0 || col == matrix.size()) {
            return INT_MAX;
        }
        // check if we have reached the last row
        if (row == matrix.size() - 1) {
            return matrix[row][col];
        }
```

```
        // calculate the minimum falling path sum starting from each possible
        // next step
        int left = findMinFallingPathSum(matrix, row + 1, col);
        int middle = findMinFallingPathSum(matrix, row + 1, col + 1);
        int right = findMinFallingPathSum(matrix, row + 1, col - 1);

        return min(left, min(middle, right)) + matrix[row][col];
    }
};
```

Complexity Analysis

Let N be the length of `matrix`.

- Time Complexity: $O(N \cdot 3^N)$ The solution takes the form of a 3-ary recursion tree where there are 3 possibilities for every node in the tree. The time complexity can be derived as follows,

- The maximum depth of the recursion tree is equal to the number of rows in the matrix i.e N.
- Each level (`level`) of the recursion tree will contain approximately 3^{level} nodes. For example, at level 0 there are 3^0 nodes, for level 1, 3^1 nodes, and so on. Thus, the maximum number of nodes at level N would be approximately 3^N.
- Thus the time complexity is roughly, $O(N \cdot 3^N)$.

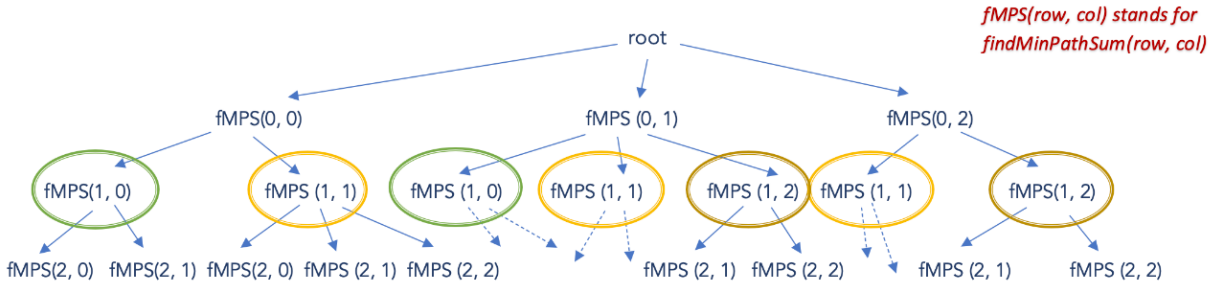
The time complexity is exponential, hence this approach is exhaustive and results in *Time Limit Exceeded (TLE)*.

- Space Complexity: $O(N)$ This space will be used to store the recursion stack. As the maximum depth of the tree is N , we will not have more than N recursive calls on the call stack at any time.

Approach 2: Top Down Dynamic Programming

Intuition

The brute force approach is exhaustive. To come up with the optimized solution for the problem, let's take a deeper look at the following recursion tree,



In the above recursion tree, we can identify the repetitive sub-paths (circled in the same color). For example, `findMinPathSum(1, 0)` is calculated twice, `findMinPathSum(1, 1)` is calculated three times, and so on.

Repeated calculation of the same subproblems is the root cause of the exponential time complexity in the previous approach. Although, what if our algorithm could remember the result for a subproblem when it is computed the first time and reuses the stored result every other time?

> Pretend you are on a treasure hunt. On reaching point A, you travel to the destination and don't find anything there. You go back to some other path which again takes you to point A. You wouldn't explore the same path from point A again. You would say, "I have been here before; I know where this path goes."
>
> How can we make our algorithm think the same way? We can do so by marking every path we have visited so that if we reach the same path again, we know the result!!

In Dynamic Programming, when a recursive problem solves the same subproblem multiple times, it has the Overlapping Subproblem property. Such problems can be optimized using a dynamic programming technique called Memoization.

As in the previous approach, each call to `findMinFallingPathSum` will return the minimum falling path sum between the current cell and the bottom of the `matrix`. However, in this approach, we will store the result of each call in the new parameter `memo`, and when we revisit this cell in a subsequent call, we will be able to reuse the stored result.

Algorithm

1. In order to record the results of computation for every cell, maintain a 2-dimensional matrix named `memo` where the value of `memo[row][col]` would give the minimum falling path starting from the cell `(row, col)`.
2. Implement a Depth First Search algorithm, by defining a recursive function, `findMinFallingPathSum(row, col)`, that recursively explores all the paths from the current cell (defined by parameters `row` and `col`).
    - Define Base Case: In any recursive function, we must define the terminating condition i.e the base case. When the terminating condition is satisfied, we will exit the recursive search process. The base cases are as follows,
        - The `row` or `col` values are not within the matrix boundaries.
        - We have reached the last row. In this case, we will return the value of the current cell and not make any other recursive calls.
    - *Recursively explore all paths:* If the base case is not satisfied, it means that we have not reached the end of our current path, and we must try all options to extend our path and find the one with the minimum sum:

```
minimumPath = Minimum(findMinFallingPathSum(row + 1, col + 1),
                      findMinFallingPathSum(row + 1, col),
                      findMinFallingPathSum(row + 1, col - 1))
```

3. To avoid repetitive computation of the results as in the brute force approach, we make use of stored results as follows,
    - Before recursively computing the result for the current cell, check if the `memo` has the result for the current cell. If so, return the result, otherwise, proceed with the recursive call to compute the result.
    - After computing the result, store the result in the `memo[row][col]`.
4. Iteratively find the minimum falling path for all possible starting cells i.e cells in 0^{th} row and columns ranging from 0 to $\text{matrix.length} - 1$. Track the minimum value in the variable `minFallingSum` and return the result.

Implementation

C++:

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int minFallingSum = INT_MAX;
        vector<vector<optional<int>>> memo(
            matrix.size(), vector<optional<int>>(matrix.size(), nullopt));

        // start a DFS (with memoization) from each cell in the top row
        for (int startCol = 0; startCol < matrix.size(); startCol++) {
            minFallingSum = min(minFallingSum, findMinFallingPathSum(
                                        matrix, 0, startCol, memo));
        }
        return minFallingSum;
    }

    int findMinFallingPathSum(vector<vector<int>>& matrix, int row, int col,
```

```
                        vector<vector<optional<int>>>& memo) {
        // base cases
        if (col < 0 || col == matrix.size()) {
            return INT_MAX;
        }
        // check if we have reached the last row
        if (row == matrix.size() - 1) {
            return matrix[row][col];
        }
        // check if the results are calculated before
        if (memo[row][col] != nullopt) {
            return (memo[row][col]).value_or(0);
        }

        // calculate the minimum falling path sum starting from each possible
        // next step
        int left = findMinFallingPathSum(matrix, row + 1, col - 1, memo);
        int middle = findMinFallingPathSum(matrix, row + 1, col, memo);
        int right = findMinFallingPathSum(matrix, row + 1, col + 1, memo);

        int minSum = min(left, min(middle, right)) + matrix[row][col];

        memo[row][col] = minSum;
        return minSum;
    }
};
```

Complexity Analysis

Let N be the length of `matrix`.

- Time Complexity: $O(N^2)$

  For every cell in the matrix, we will compute the result only once and update the `memo`. For the subsequent calls, we are using the stored results that take $O(1)$ time. There are N^2 cells in the matrix, and thus N^2 dp states. So, the tim complexity is $O(N^2)$.

- Space Complexity: $O(N^2)$

  The recursive call stack uses $O(N)$ space. As the maximum depth of the tree is N, we can't have more than N recursive calls on the call stack at any time. The 2D matrix `memo` uses $O(N^2)$ space. Thus, the space complexity is $O(N) + O(N^2) = O(N^2)$.

Approach 3: Bottom-Up Dynamic Programming (Tabulation)

Intuition

The memoization technique follows the top-down approach. We start by finding the result for the original matrix and recursively move towards computing the result of smaller subproblems.

There is yet another technique to implement Dynamic Programming problems called bottom-up dynamic programming also known as, Tabulation. In the bottom-up approach, we start by finding the result of the smallest subproblem and iteratively move towards larger sub-problems. Being non-recursive in nature, this approach does not require maintaining a separate internal call stack for storing the intermediate state of recursion.

The results of subproblems are stored in a 2-dimensional table, `dp`. Here, the smallest sub-problem would be a matrix with only one row. Thus, we will first calculate the result for each cell in the last row (base case), call this the n^{th} row, and store the results in `dp`. Next, when we move to the $(n-1)^{th}$ row, the results for the next row (i.e. the n^{th} row) are already present. In this way, in every iteration, we are calculating and storing the result for the subsequent problem.

While this approach does not improve the time or space complexity compared to the top-down approach, it does act as a stepping stone towards the next approach where we will improve the space complexity.

Algorithm

Bottom-up Dynamic Programming follows an iterative approach to solving the problem. We have to start by finding the minimum falling path for the smallest possible matrix i.e the matrix having a single cell and one by one move towards the rest of the cells in the matrix.

1. To compute the minimum falling path for a certain cell `(row, col)`, we must have pre-computed values for the minimum falling path for cells `(row + 1, col - 1)`, `(row + 1, col)` and `(row + 1, col + 1)`. For this, we will iterate from $(n-1)^{th}$ row to 0^{th} row and from 0^{th} column to $(n-1)^{th}$ column.

   > Note: The order of iterating the columns does not matter in this case. Even if we iterate from $(n-1)^{th}$ to 0^{th} column, the results would be the same.

2. Build a 2D matrix `dp` and compute the minimum falling path of current `row` and `col` as,

   ```
   dp[row][col] = min(dp[row + 1][col - 1],
                      dp[row + 1][col],
                      dp[row + 1][col + 1]) + value of current (row, col) in matrix
   ```

   Note: We must handle the edge cases, for example, if the col value is `0` or `n - 1`.

3. Once we have the value of the minimum falling path from every cell, we must get the result from the start cell. The start cell can be any cell on the first row. Thus, we will iterate over all the cells in the first row, and return the minimum value.

Implementation

C++:

```
class Solution {
public:
```

```
    int minFallingPathSum(vector<vector<int>>& matrix) {
        vector<vector<int>> dp(matrix.size() + 1,
                               vector<int>(matrix.size() + 1, 0));
        for (int row = matrix.size() - 1; row >= 0; row--) {
            for (int col = 0; col < matrix.size(); col++) {
                if (col == 0) {
                    dp[row][col] = min(dp[row + 1][col], dp[row + 1][col + 1]) +
                                   matrix[row][col];
                } else if (col == matrix.size() - 1) {
                    dp[row][col] = min(dp[row + 1][col], dp[row + 1][col - 1]) +
                                   matrix[row][col];
                } else {
                    dp[row][col] =
                        min(dp[row + 1][col],
                            min(dp[row + 1][col + 1], dp[row + 1][col - 1])) +
                        matrix[row][col];
                }
            }
        }
        int minFallingSum = INT_MAX;
        for (int startCol = 0; startCol < matrix.size(); startCol++) {
            minFallingSum = min(minFallingSum, dp[0][startCol]);
        }
        return minFallingSum;
    }
};
```

Complexity Analysis

Let N be the length of `matrix`.

- Time Complexity: $O(N^2)$
    - The nested for loop takes (N^2) times to fill the `dp` array.
    - Then, takes N time to find the minimum falling path.
    - So, Time Complexity $T(n) = O(N^2) + O(N) = O(N^2)$
- Space Complexity: $O(N^2)$. The additional space is used for `dp` array of size N^2.

Approach 4: Space Optimized, Bottom-Up Dynamic Programming

Intuition

The tabulation approach used $O(N^2)$ space to track the minimum falling path starting from every cell. But, in the end, to calculate the final result, we only need the minimum falling path of all the cells in the first row.

Furthermore, to calculate the values for cells in the row `n`, we only need the values of the `n + 1` row. So, after finding the results for row `n`, we no longer need to store the values for the `n + 1` row. As such, we can optimize the space complexity of the tabulation approach as follows,

- Calculate the values for all the cells in the current row `n` based on results for the cells in the `n + 1` row. Let's define a 1-dimensional array, `dp`, to store the results of the `n + 1` row.
- As we move to the `n - 1` row, the current row becomes the `n + 1` row. So, we can discard the current values of `dp`, and begin to store the results for the current row in the `dp` array.

In this way, instead of maintaining a 2-dimensional array the size of a `matrix`, we can just define two 1-dimensional arrays, one that stores the value of the current row and the other that has the results of the previous row.

Algorithm

1. Define a 1-dimensional array, `dp`, initialized with all values set to `0`,
2. For every row, define a 1-dimensional array, `currentRow`, to store the results of all the columns in the current row.
3. Calculate the values for all the columns in the current row, `currentRow` based on the values of the previous row stored in array `dp`,

   ```
   currentRow[col] = min(dp[col - 1],
                         dp[col],
                         dp[col + 1]) + value of current (row, col) in matrix
   ```

   Note: We must handle the edge cases, for example, if the col value is `0` or `n - 1`.
4. Once all the values of the current row are calculated, assign the values of `currentRow` to the `dp` array. So that the values in `dp` servers as the values of the previous row for the next iteration.

   In some programming languages, this can be done by just changing the pointer reference instead of copying the values one by one.
5. The process is repeated for all the rows, and at the end, we will have the results for the cells in the first row, in the `dp` array, and we can find the minimum falling path.

Implementation

C++:

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        vector<int> dp(matrix.size() + 1, 0);
        for (int row = matrix.size() - 1; row >= 0; row--) {
            vector<int> currentRow(matrix.size() + 1, 0);
            for (int col = 0; col < matrix.size(); col++) {
```

```
            if (col == 0) {
                currentRow[col] =
                    min(dp[col], dp[col + 1]) + matrix[row][col];
            } else if (col == matrix.size() - 1) {
                currentRow[col] =
                    min(dp[col], dp[col - 1]) + matrix[row][col];
            } else {
                currentRow[col] =
                    min(dp[col], min(dp[col + 1], dp[col - 1])) +
                    matrix[row][col];
            }
        }
        dp = currentRow;
    }
    int minFallingSum = INT_MAX;
    for (int startCol = 0; startCol < matrix.size(); startCol++) {
        minFallingSum = min(minFallingSum, dp[startCol]);
    }
    return minFallingSum;
  }
};
```

Complexity Analysis

Let N be the length of `matrix`.

- Time Complexity: $O(N^2)$
    - The nested for loop takes (N^2) time.
    - Then, it takes Ntime to find the minimum falling path.
    - So, Time Complexity $T(n) = O(N^2) + O(N) = O(N^2)$
- Space Complexity: $O(N)$.
    - We are using two 1-dimensional arrays `dp` and `currentRow` of size N.

Comment:

In approach 4, I think its not space optimized, as we need to initialize O(n) vector, everytime.
Here is my approach, which is space optimized to O(1).

```
class Solution {
public:
    int minFallingPathSum(vector<vector<int>>& matrix) {
        int n = matrix.size();

        int mini = INT_MAX;
        for(int i=1; i<n; i++){
            for(int j =0; j<n; j++){
                if(j == 0){
                    matrix[i][j] += min( matrix[i-1][j],  matrix[i-1][j+1]);
                }
                else if(j == n-1){
                    matrix[i][j] += min( matrix[i-1][j],  matrix[i-1][j-1]);
                }
                else{
                    matrix[i][j] += min( matrix[i-1][j], min(matrix[i-1][j-1], matrix[i-1][j+1
                }
            }
        }

        for(int i = 0; i<n; i++) {
            mini = min(mini, matrix[n-1][i]);
        }
        return mini;
    }
};
```