375. Guess Number Higher or Lower II

Medium | Math | Dynamic Programming | Game Theory

Problem Description

Imagine you're playing a guessing game where the objective is to figure out a hidden number and you want to spend as little money as possible in the process. In this game, you will guess numbers between 1 and `n`, and each incorrect guess will cost you the value of the guess in dollars. If you get it right, you win! If you get it wrong, you'll be told if the hidden number is higher or lower than your guess. You want to know the least amount of money required to guarantee a win no matter what number is chosen by your opponent.

Intuition

The approach to solve this problem is to use dynamic programming, a method where we break the problem down into simpler subproblems and build up a solution to the larger problem based on the solutions to the smaller problems.

This problem requires a strategy to minimize the total cost while guaranteeing a win. To achieve this, we need to consider all possible outcomes and make the best worst-case decision at each guess. We think of the game in terms of ranges (from 1 to `n`) and then narrow down the range based on whether we need to guess higher or lower. This approach will help us figure out the minimum cost required to win the game.

We use a 2D array `dp` to record the cost to guarantee a win for every range `[i, j]`. The cost of the guess `k` is the maximum between `dp[i][k-1]` (if the number is lower than `k`) and `dp[k+1][j]` (if the number is higher than `k`) plus the cost `k` of making the guess. Since we want to minimize our cost, we take the minimum over all possible `k`'s for the range `[i, j]`. The answer is then `dp[1][n]` since it's the minimum amount of money required to guarantee a win for the full range.

Solution Approach

The implementation leverages the concept of dynamic programming to develop a strategy that ensures the minimum cost of a guaranteed win. Here's how the solution is built:

1. **Dynamic Programming Table Initialization**: Create a 2-dimensional array `dp` where `dp[i][j]` represents the minimum cost to guarantee a win for the range `[i, j]`. The size of the array is `(n + 10)` by `(n + 10)` to cover all the possible ranges between `1` and `n`, with a bit of extra space to avoid index out-of-bounds errors.
2. **Filling the DP Table**: Iterate over all the possible lengths `l` of the ranges from `2` to `n+1`. For each length, compute the minimum cost for all ranges of that length starting from `1`.
3. **Computing Costs**: For each range `[i, j]` with the starting point `i` and the endpoint `j`, initialize the minimum cost `dp[i][j]` to infinity as a starting point. Then, try every possible guess `k` within the range `[i, j]`. For each guess, calculate the cost `t`, which is the maximum of either guessing too low (`dp[i][k - 1]`) or too high (`dp[k + 1][j]`), plus the cost `k` of making the current guess.
4. **Minimizing Costs**: The value of `dp[i][j]` is updated to the minimum cost out of all the guesses `k`.
5. **Result Extraction**: After populating the `dp` table with the costs for all ranges, the minimum cost to guarantee a win for the full range from `1` to `n` is found in `dp[1][n]`.

The algorithm uses a bottom-up strategy, solving smaller range problems first and using their solutions to address larger ranges. This ensures that at every step, the optimal (minimum) cost for the current range is based on the best decisions made for the previous smaller ranges.

This approach ensures that we achieve a global optimum by making locally optimal choices, thanks to the overlapping subproblems and optimal substructure properties of dynamic programming.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose `n = 4`, so the hidden number is between 1 and 4, and we want to find the minimum amount of money required to guarantee a win.

Step 1: Dynamic Programming Table Initialization
We initialize a 2D array `dp` that has 5×5 dimensions since `n` is 4 and we add 1 to have an inclusive range.

Initially, `dp` looks like this (with 0s where no cost is involved):

```
dp = [
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0],
]
```

Step 2: Filling the DP Table
We consider ranges of lengths from 2 to 5.

Step 3: Computing Costs
When the length is 2, we consider ranges: `[1, 2]`, `[2, 3]`, `[3, 4]`. For example, for the range `[1, 2]`, we can guess 1 or 2. If we guess 1 and the number is 2, we lose $1 and then pay $2 to guess correctly, which sums to $3. If we guess 2 and the number is 1, we lose $2 but then just pay $1 to guess it correctly, summing to $3. So, the minimum cost here is $3.

The table looks partially like this:

```
dp = [
  [0, 0, 0, 0, 0],
  [0, 0, 3, 0, 0],
  [0, 0, 0, 3, 0],
  [0, 0, 0, 0, 3],
  [0, 0, 0, 0, 0],
]
```

Step 4: Minimizing Costs
We move to ranges of length 3 and 4 and update `dp` accordingly. Take range `[1, 3]`. Our guesses are 1, 2, or 3:

- Guess 1: If it's wrong, we lose 1, and the minimum cost to guess between [2, 3] is \3 (from step 3). So, total cost = $1 + $3 = $4.
- Guess 2: If it's wrong, we can face a higher or lower number, resulting in a cost of max([1], [3]) = $2 (always correct on the second guess, 1 or 3). So, total cost = $2 + $2 = $4.
- Guess 3: Similar to guessing 1, total cost = $4.

Therefore, the cost to guarantee a win between `[1, 3]` is $4.

We proceed similarly for all lengths. Finally, for length 4, the range `[1, 4]` has multiple scenarios and we select the minimum cost of all.

Step 5: Result Extraction
After computing all possible ranges, our `dp` array for the range `[1, n]` (which is `[1, 4]` in our example) gives us the answer. For this example, `dp[1][4]` will end up being the minimum cost to guarantee a win for the game.

By building the solution from the smallest range to the largest and choosing the minimum cost at each step, we ensure the minimum amount of money required to guarantee a win for the entire range. This example uses smaller numbers for simplicity, but the actual implementation would handle larger values of `n` following the same process.

Solution Implementation

Python

```python
class Solution:
    def getMoneyAmount(self, n: int) -> int:
        # Initialize the dynamic programming (DP) table with 0's and add extra space to avoid index out of range
        dp_table = [[0] * (n + 10) for _ in range(n + 10)]

        # Loop over the lengths of the ranges of numbers we are considering
        for length in range(2, n + 1):
            # Start index of the current range
            for start in range(1, n - length + 2):
                # End index of the current range
                end = start + length - 1

                # Initialize the DP table entry with infinity to later find the minimum cost
                dp_table[start][end] = float('inf')

                # Check each possible number (k) to guess within the current range
                for k in range(start, end + 1):
                    # The cost of guessing k includes the cost of the worst scenario from both sides
                    # of the range split by k, plus the cost of guessing k
                    cost_when_guessing_k = max(dp_table[start][k - 1], dp_table[k + 1][end]) + k

                    # Update the DP table for the current range with the minimum cost amongst all k's
                    dp_table[start][end] = min(dp_table[start][end], cost_when_guessing_k)

        # The answer is the minimum cost to guess the number in range 1...n
        return dp_table[1][n]
```

Java

```java
class Solution {
    public int getMoneyAmount(int n) {
        // Initialize the dynamic programming table.
        // dp[i][j] represents the minimum amount of money required to
        // guarantee a win for guessing the number between i and j.
        int[][] dp = new int[n + 1][n + 1];

        // Fill the table using the bottom-up dynamic programming approach.
        // 'l' represents the range size from i to j.
        for (int l = 2; l <= n; ++l) {
            // 'i' is the lower bound of the range.
            for (int i = 1; i + l - 1 <= n; ++i) {
                // 'j' is the upper bound of the range.
                int j = i + l - 1;
                // Initialize the value to maximum it could possibly be.
                dp[i][j] = Integer.MAX_VALUE;

                // Test all possible guesses 'k' from i to j and
                // take the cost of the most unfortunate (worst-case) outcome.
                for (int k = i; k <= j; ++k) {
                    // The cost is calculated as the higher cost of two scenarios:
                    // * the cost of guessing a number too low and then solving the range [k+1, j].
                    // * the cost of guessing a number too high and then solving the range [i, k-1].
                    // Then add the cost 'k' of making the guess itself.
                    int cost = Math.max(dp[i][k - 1], dp[k + 1][j]) + k;
                    // Find the minimum cost among all possible guesses 'k'.
                    dp[i][j] = Math.min(dp[i][j], cost);
                }
            }
        }
        // The answer is the minimum cost to guarantee a win when the range is from 1 to n.
        return dp[1][n];
    }
}
```

C++

```cpp
#include <vector>
#include <climits> // For INT_MAX

class Solution {
public:
    int getMoneyAmount(int n) {
        // Initialize a 2D vector dp where dp[i][j] represents the minimal amount
        // of money required to guarantee a win when guessing numbers between i and j.
        std::vector<std::vector<int>> dp(n + 1, std::vector<int>(n + 1));

        // Length of the range we are considering. Starts at 2 since a single number range requires no guesses.
        for (int length = 2; length <= n; ++length) {
            for (int start = 1; start <= n - length + 1; ++start) {
                int end = start + length - 1;

                // Initialize the dp value for the range [start, end] to the highest possible value.
                dp[start][end] = INT_MAX;

                // Try every possible guess (pick 'k') from the range [start, end].
                for (int k = start; k <= end; ++k) {
                    // Compute the cost when choosing 'k'. It is the higher cost of the two ranges
                    // split by 'k' (since we're minimizing the worst-case scenario) plus the cost of 'k'.
                    int costWhenPickingK = k + std::max(
                        k > start ? dp[start][k - 1] : 0, // Cost of the left subrange if k is not the start
                        k < end ? dp[k + 1][end] : 0      // Cost of the right subrange if k is not the end
                    );

                    // Update the minimal cost for the range [start, end].
                    dp[start][end] = std::min(dp[start][end], costWhenPickingK);
                }
            }
        }

        // Return the minimal amount of money required to guarantee a win for the range [1, n].
        return dp[1][n];
    }
};
```

TypeScript

```typescript
// Importing module for max safe integer constant.
import { MAX_SAFE_INTEGER } from "constants";

// Initializes a 2D array 'dp' where dp[i][j] represents the minimal amount
// of money required to guarantee a win when guessing numbers between i and j.
const dp: number[][] = [];

// Method that calculates the minimal amount of money required to guarantee a win.
function getMoneyAmount(n: number): number {
    // Initialize 'dp' with the appropriate dimensions and fill it with zeros.
    for (let i = 0; i <= n; i++) {
        dp[i] = new Array(n + 1).fill(0);
    }

    // Analyzing different range lengths, starting from length 2 since a single number requires no guess.
    for (let length = 2; length <= n; length++) {
        // Iterating over each possible starting point of the range.
        for (let start = 1; start <= n - length + 1; start++) {
            const end = start + length - 1;

            // Initialize the 'dp' value for this range to the highest safe integer to ensure minimization.
            dp[start][end] = MAX_SAFE_INTEGER;

            // Iterate over every possible number 'k' within the current range to simulate a guess.
            for (let k = start; k <= end; k++) {
                // Calculate the cost of picking number 'k', which is k plus the highest cost between the two subranges.
                const costWhenPickingK = k + Math.max(
                    k > start ? dp[start][k - 1] : 0, // Cost for the left subrange, if 'k' is not at the start.
                    k < end ? dp[k + 1][end] : 0      // Cost for the right subrange, if 'k' is not at the end.
                );

                // Store the minimum cost between the current and the calculated cost for this range.
                dp[start][end] = Math.min(dp[start][end], costWhenPickingK);
            }
        }
    }

    // Return the minimal amount of money to guarantee a win for the full range of numbers.
    return dp[1][n];
}

// Example usage: calculate the minimal amount needed to guarantee a win from 1 to 10.
console.log(getMoneyAmount(10));
```

```python
class Solution:
    def getMoneyAmount(self, n: int) -> int:
        # Initialize the dynamic programming (DP) table with 0's and add extra space to avoid index out of range
        dp_table = [[0] * (n + 10) for _ in range(n + 10)]

        # Loop over the lengths of the ranges of numbers we are considering
        for length in range(2, n + 1):
            # Start index of the current range
            for start in range(1, n - length + 2):
                # End index of the current range
                end = start + length - 1

                # Initialize the DP table entry with infinity to later find the minimum cost
                dp_table[start][end] = float('inf')

                # Check each possible number (k) to guess within the current range
                for k in range(start, end + 1):
                    # The cost of guessing k includes the cost of the worst scenario from both sides
                    # of the range split by k, plus the cost of guessing k
                    cost_when_guessing_k = max(dp_table[start][k - 1], dp_table[k + 1][end]) + k

                    # Update the DP table for the current range with the minimum cost amongst all k's
                    dp_table[start][end] = min(dp_table[start][end], cost_when_guessing_k)

        # The answer is the minimum cost to guess the number in range 1...n
        return dp_table[1][n]
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n^3)`. This is because there are three nested loops: the outermost loop ranges over `l`, which goes from `2` to `n`; the second loop ranges over `i`, which goes from `1` to `n - l + 1`. For each combination of `i` and `l`, the innermost loop ranges over `k`, which goes from `i` to `j` (where `j = i + l - 1`). For each value of `k`, the code computes `t` which involves querying and updating values in the `dp` array, which takes constant time. The number of iterations is roughly `n` for the `i` loop, `n` for the `l` loop, and `n` again for the `k` loop, multiplying together for a cubic time complexity.

Space Complexity

The space complexity of the code is `O(n^2)`. We allocate a `dp` array that has `n + 10` rows and `n + 10` columns to ensure that we do not index out of bounds when performing dynamic programming. Though not all of these entries are used, the overall space required is proportional to `n^2` since the bulk of the space is occupied by the `n * n` subarray required to calculate the minimum cost for guessing the number.