873. Length of Longest Fibonacci Subsequence

Medium Array Hash Table Dynamic Programming Leetcode Link

Problem Description

The problem provides us with a sequence of positive integers that is strictly increasing - meaning each number is greater than the previous one. We are asked to find the length of the longest subsequence that is Fibonacci-like within this sequence. A subsequence is considered Fibonacci-like if it satisfies two conditions: It must have at least three numbers, and each number in the subsequence (after the second) must be the sum of the two preceding numbers. To clarify with an example, if our sequence is `[1, 2, 3, 4, 5, 6, 7, 8]`, a Fibonacci-like subsequence could be `[1, 2, 3, 5, 8]`. If no such subsequence exists, the result should be `0`.

Intuition

The solution uses the concept of *dynamic programming* to build up a table of solutions to subproblems, which in this case, are the lengths of the longest Fibonacci-like subsequences ending at different positions in the original sequence.

First, we create a mapping of each number in the sequence to its index to speed up the search for a number's existence within the sequence.

Since we need at least three numbers to form a Fibonacci-like sequence, the default length for any pair of starting numbers is 2 (which is not a valid Fibonacci-like subsequence by itself, but is the basis for building longer ones).

The key intuition is that if `arr[k] + arr[j] == arr[i]` and `k < j < i`, then the value at `dp[j][i]` (the length of the subsequence ending with `arr[j]` and `arr[i]`) can be updated to `dp[k][j] + 1` (which extends the subsequence that ends with `arr[k]` and `arr[j]` by adding `arr[i]`).

We iterate through each pair of numbers, updating the dynamic programming table `dp` whenever we find two numbers that add up to a third one in the sequence. The answer is the maximum value in the table `dp` that represents the length of the longest Fibonacci-like subsequence we have found.

At each step, we are effectively looking backwards from a potential ending number `i` to see if we can form a Fibonacci-like sequence by finding two previous numbers `j` and `k` that add up to `arr[i]`. If we find such a pair, we know we can extend the subsequence that ended at `arr[j]` to now end at `arr[i]` and hence update our `dp` table accordingly.

Solution Approach

The solution approach takes advantage of a dynamic programming paradigm and hash mapping. We use these tools to systematically build a solution that finds the longest Fibonacci-like subsequence within an array of strictly increasing positive integers. The following steps outline how the code implements this solution:

1. **Hash Map Construction**: We first construct a hash map (`mp`) to store each array value (`v`) and its respective index (`i`). This greatly optimizes our solution as it allows us to check whether a number exists in the array in constant time.

   ```
   mp = {v: i for i, v in enumerate(arr)}
   ```

2. **Dynamic Programming Table Initialization**: We create a 2D table (`dp`) where `dp[j][i]` will hold the maximum length of a Fibonacci-like subsequence which ends with the elements `arr[j]` and `arr[i]`. Initially, we set all pairs (`j`, `i`) in `dp` to `2`, which is the minimum length of any subsequence consisting of two elements.

   ```
   dp = [[0] * n for _ in range(n)]
   for i in range(n):
       for j in range(i):
           dp[j][i] = 2
   ```

3. **Updating the DP Table**: We then use two nested loops to iterate through the array to find and extend existing Fibonacci-like subsequences. For each pair of indices `(j, i)`, we calculate the potential preceding number `d` by subtracting `arr[j]` from `arr[i]`.

   ```
   for i in range(n):
       for j in range(i):
           d = arr[i] - arr[j]
   ```

   We then check if this number `d` exists in the array (and thus in the hash map). If it does and the index of `d` (`k`) is less than `j`, we are in a position to extend the subsequence that previously ended with `arr[k]` and `arr[j]`. The length of the new subsequence will be `dp[k][j] + 1`.

   ```
   if d in mp and (k := mp[d]) < j:
       dp[j][i] = max(dp[j][i], dp[k][j] + 1)
   ```

4. **Finding the Answer**: While updating the `dp` table, we keep track of the maximum length found so far (`ans`). Whenever we update an entry in the `dp` table, we compare it with the current maximum length and update `ans` if necessary.

   ```
   ans = max(ans, dp[j][i])
   ```

5. **Returning the Result**: After iterating over all pairs and updating the `dp` table, `ans` contains the length of the longest Fibonacci-like subsequence. If no such subsequence exists longer than two elements, `ans` will be `0`. Therefore, we return `ans` as the result.

   ```
   return ans
   ```

It's interesting to note that the dynamic programming table is not fully utilized. Most of its default-initialization values remain unchanged, especially near the diagonal where `i` and `j` are close. The only entries that receive meaningful updates reflect valid subsequences of length greater than two. The efficient use of the hash map allows for quick validation of potential Fibonacci-like sequences, which is central to the dynamic programming update step.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using a small sequence of positive integers: `[1, 3, 7, 11, 12, 14, 18]`.

First, we initialize the hash map `mp` to map each number to its index in the array.

```
mp = {1: 0, 3: 1, 7: 2, 11: 3, 12: 4, 14: 5, 18: 6}
```

Next, we initialize the dynamic programming table `dp`. Since the array length `n` is 7, `dp` is a 7×7 2D array where all pairs (`j`, `i`) are initially set to a subsequence length of 2.

Now, we start iterating over the pairs of indices `(j, i)`. We're looking for whether the difference `d = arr[i] - arr[j]` exists in the array and corresponds to an index `k` such that `k < j`.

When `i = 3` and `j = 2`, `arr[i] = 11`, `arr[j] = 7`, so `d = 11 - 7 = 4`. There is no `4` in the array, so we move on.

Next significant update happens when `i = 4` and `j = 1`. Here `arr[i] = 12` and `arr[j] = 3`, so `d = 12 - 3 = 9`. The number `9` does not exist in the array either. Again, we proceed forward.

When `i = 6` and `j = 5`, `arr[i] = 18` and `arr[j] = 14`, so `d = 18 - 14 = 4`. Since `4` is not present in our array, no update in `dp` occurs.

However, the first update occurs when `i = 5` and `j = 1`, then `arr[i] = 14`, `arr[j] = 3`, and hence `d = 14 - 3 = 11`. We see that `11` is in the array at index `3`. Therefore, `k = 3` and `k < j`. We update `dp[j][i] = dp[k][j] + 1 = dp[3][1] + 1 = 2 + 1 = 3`.

Now `dp` looks like this (a snippet of relevant parts):

```
dp[1][5] = 3   # This tells us there's a subsequence ending at arr[1] (3) and arr[5] (14) with length 3.
```

Given this `dp` update, our current maximum `ans` becomes `3`.

By the end of the iteration, since no other updates result in a longer subsequence, the longest Fibonacci-like subsequence we have is `[3, 11, 14]`. Hence, the answer `ans` is `3`. If there were no updates that result in a value greater than `2`, `ans` would remain `0`, indicating no valid Fibonacci-like subsequences exist beyond two elements.

Finally, we return `ans`, which is `3` in this case.

Python Solution

```python
class Solution:
    def lenLongestFibSubseq(self, arr):
        # Create a hash map to store value to index mappings for quick access
        value_to_index = {value: index for index, value in enumerate(arr)}
        n = len(arr)

        # Initialize a 2D array for dynamic programming, setting initial subsequence sizes to 2
        dp = [[2 for _ in range(n)] for _ in range(n)]

        # This variable will keep track of the longest fib sequence found
        longest_fib_sequence = 0

        # Iterate over pairs of numbers in the array to build up sequences
        for i in range(n):
            for j in range(i):
                # Find the previous number in the potential Fibonacci sequence
                difference = arr[i] - arr[j]
                # Check if the number exists in our array and precedes the second number (j)
                if difference in value_to_index and value_to_index[difference] < j:
                    # The k index refers to the position of the previous number
                    prev_index = value_to_index[difference]
                    # Update the DP table by extending the sequence found from prev_index and j
                    dp[j][i] = max(dp[j][i], dp[prev_index][j] + 1)
                    # Update the answer if a longer sequence is found
                    longest_fib_sequence = max(longest_fib_sequence, dp[j][i])

        return longest_fib_sequence

```

Java Solution

```java
class Solution {
    public int lenLongestFibSubseq(int[] arr) {
        // The length of the input array
        int length = arr.length;
        // Create a HashMap to store the index of each value in the array for quick lookup
        Map<Integer, Integer> indexMap = new HashMap<>();
        for (int i = 0; i < length; ++i) {
            indexMap.put(arr[i], i);
        }
        // Initialize the dynamic programming table where dp[i][j] will store
        // the length of the Fibonacci-like sequence ending with arr[i] and arr[j]
        int[][] dp = new int[length][length];
        // Initialize the table with the minimum possible sequence length, which is 2
        for (int i = 0; i < length; ++i) {
            for (int j = 0; j < i; ++j) {
                dp[j][i] = 2;
            }
        }
        // This will hold the length of the longest Fibonacci-like subsequence
        int longestSequenceLength = 0;
        // Iterate over all pairs of indices to build the longest sequence
        for (int i = 0; i < length; ++i) {
            for (int j = 0; j < i; ++j) {
                // The potential previous value in the Fibonacci-like sequence
                int prevValue = arr[i] - arr[j];
                // Check if the needed previous value is in the array
                if (indexMap.containsKey(prevValue)) {
                    // The index k of the needed previous value
                    int k = indexMap.get(prevValue);
                    // Ensure the previous value's index comes before j
                    if (k < j) {
                        // Update the sequence length considering the triplet (arr[k], arr[j], arr[i])
                        dp[j][i] = Math.max(dp[j][i], dp[k][j] + 1);
                        // Update the global maximum length if necessary
                        longestSequenceLength = Math.max(longestSequenceLength, dp[j][i]);
                    }
                }
            }
        }
        // If the longestSequenceLength remains 2, no Fibonacci-like sequence is found,
        // so we return 0 as specified in the problem statement
        return longestSequenceLength > 2 ? longestSequenceLength : 0;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Function that returns the length of the longest Fibonacci-like subsequence.
    int lenLongestFibSubseq(vector<int>& arr) {
        // Hash map that associates each value in arr with its index.
        unordered_map<int, int> indexMap;
        int n = arr.size();
        // Fill the index map with value-index pairs.
        for (int i = 0; i < n; ++i) {
            indexMap[arr[i]] = i;
        }

        // Initialize a 2D vector 'dp' with 2s, where dp[j][i] will store
        // the length of the longest Fibonacci-like subsequence ending with arr[j] and arr[i].
        vector<vector<int>> dp(n, vector<int>(n, 2));

        // Variable to store the answer: the length of the longest subsequence found.
        int maxLength = 0;

        // Double loop to find Fibonacci-like subsequences ending with pairs (arr[j], arr[i]).
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                // Calculate the potential previous Fibonacci number in the sequence.
                int prevNum = arr[i] - arr[j];
                // Check if this number exists in our map (hence, in the array).
                if (indexMap.count(prevNum)) {
                    // Get the index 'k' of the found previous number.
                    int k = indexMap[prevNum];
                    // Ensure this index 'k' comes before index 'j' to maintain the sequence ordering.
                    if (k < j) {
                        // Update the length of the subsequence ending with (arr[j], arr[i]).
                        dp[j][i] = max(dp[j][i], dp[k][j] + 1);
                        // Update the answer if we found a longer subsequence.
                        maxLength = max(maxLength, dp[j][i]);
                    }
                }
            }
        }

        // If no sequence is found, maxLength would be 2, as initialized. But according to the problem,
        // the sequence should have at least 3 numbers to count, so return 0 in that case.
        return maxLength > 2 ? maxLength : 0;
    }
};

```

Typescript Solution

```typescript
/**
 * Returns the length of the longest Fibonacci-like subsequence of `arr`.
 * If there is no such subsequence, returns 0.
 * A sequence is considered a Fibonacci-like sequence if `X[i] + X[i-1] = X[i+1]` for all `i + 1 < X.length`.
 * @param {number[]} arr - An array of positive integers
 * @return {number} - The length of the longest Fibonacci-like subsequence in `arr`
 */
function lenLongestFibSubseq(arr: number[]): number {
    const indexMap: Map<number, number> = new Map();
    const n: number = arr.length;
    const dp: number[][] = Array.from({ length: n }, () => new Array(n).fill(2));
    let maxSequenceLength: number = 0;

    // Populate the map with the value to index mappings for quick access
    for (let i = 0; i < n; ++i) {
        indexMap.set(arr[i], i);
    }

    // Construct the dp table where dp[j][i] will represent the length of the longest
    // Fibonacci-like subsequence ending with arr[j] and arr[i].
    for (let i = 0; i < n; ++i) {
        for (let j = 0; j < i; ++j) {
            const potentialPreviousValue: number = arr[i] - arr[j];

            // We only need to investigate if the potential previous value exists in the array and comes before 'j'
            if (indexMap.has(potentialPreviousValue)) {
                const k: number = indexMap.get(potentialPreviousValue)!;
                if (k < j) {
                    dp[j][i] = Math.max(dp[j][i], dp[k][j] + 1);
                    maxSequenceLength = Math.max(maxSequenceLength, dp[j][i]);
                }
            }
        }
    }

    // If we have only computed the initial value of 2 for all pairs,
    // it means no Fibonacci-like sequence has been found.
    return maxSequenceLength > 2 ? maxSequenceLength : 0;
}

```

Time and Space Complexity

The given code is designed to find the length of the longest Fibonacci-like subsequence in a given array `arr`. We analyze both the time complexity and the space complexity of the code.

Time Complexity:

Initially, the given code creates a hashmap (`mp`) from each element to its index with a time complexity of `O(n)`, where `n` is the length of the array `arr`.

The main part of the algorithm consists of nested loops. The outer loop runs `n` times, and for each iteration, the inner loop runs at most `n` times, yielding `n^2` iterations for the nested loops.

Inside the inner loop, the code performs constant-time operations such as dictionary look-up (to check if `d` exists in `mp`), assignments, and arithmetic operations. The look-up in `mp` takes `O(1)` on average. Therefore, the inner block within the inner loop also executes in constant time.

Considering these nested loops as the dominant factor, the time complexity is `O(n^2)`.

Space Complexity:

For space complexity, the code uses:

- A hashmap `mp` with at most `n` entries, thus `O(n)` space.
- A 2D array `dp` of size `n * n`, accounting for `O(n^2)` space.

Therefore, the space complexity is dominated by the 2D array `dp`, making the overall space complexity of the code `O(n^2)`.

In conclusion, the code has a **time complexity of `O(n^2)`** and a **space complexity of `O(n^2)`**.