1027. Longest Arithmetic Subsequence

Array   Hash Table   Binary Search   Dynamic Programming

Problem Description

The problem is to identify the length of the longest arithmetic subsequence within a given array of integers, nums. An arithmetic subsequence is a sequence that can be derived by removing zero or more elements from the array without changing the order of the remaining elements, and where the difference between consecutive elements is constant.

For example, if nums is [3, 6, 9, 12], then [3, 9, 12] is an arithmetic subsequence with a common difference of 6 between its first and second elements, and 3 between its second and third elements. The requirement is to find the longest such subsequence.

Key considerations:

- Subsequences are not required to be contiguous, meaning they don't have to occur sequentially in the array.
- The array's elements are not guaranteed to be in an arithmetic sequence order to start with.
- The longest arithmetic subsequence could start and end at any index in the array, with any common difference.

Intuition

To solve this problem, we approach it using dynamic programming, which will help us to build solutions to larger problems based on solutions to smaller, overlapping subproblems.

The intuition is as follows:

- We traverse through the array, and for each element, we check its difference with the elements before it.
- The difference can be negative, positive, or zero; hence, we offset the difference by a certain number to use it as an index. In this case, 500 is used to offset the difference to ensure it can be used as a non-negative index.
- Using a 2D array f, where f[i][j] represents the length of the longest arithmetic subsequence that ends with nums[i] and has a common difference offset by 500.
- For each pair (nums[i], nums[k]) where k < i, we calculate the difference offset j and update f[i][j] accordingly.
- f[i][j] is the maximum of its current value and f[k][j] + 1 because if there exists a subsequence ending with nums[k] having the same common difference, then by adding nums[i], we can extend that subsequence.
- We keep track of the maximum length of any arithmetic subsequence found so far.

This solution leverages the historical information stored in the dynamic programming table to determine the maximum achievable length of an arithmetic subsequence up to the current index. By iterating through all elements and all possible common differences, we find the maximum length of an arithmetic subsequence in the array.

Solution Approach

The implementation of the solution makes use of dynamic programming, a typical algorithmic technique for solving problems by breaking them down into overlapping sub-problems. The core idea is to avoid re-computing the answer for sub-problems we have already solved by storing their results.

Here's a step-by-step explanation of the Solution Approach based on the provided code:

1. **Initial Setup:** Create a 2D array f of size n x 1001, where n is the length of nums. f[i][j] will store the length of the longest arithmetic subsequence up to index i with a common difference offset by 500. Initialize each value in f to 1, because the smallest arithmetic subsequence including any single number is just the number itself, with a length of 1.

2. **Looping Through Pairs:** For each element nums[i], starting from the second element in nums, iterate backwards through all previous elements nums[k] where k < i. This way you are comparing the current element nums[i] with each of the elements before it, one by one.

3. **Compute Difference:** Calculate the difference diff between nums[i] and nums[k], and then offset it by 500 to get a new index j. This offsetting technique is used to convert possible negative differences into positive indices, allowing them to be used as indices for the 2D array f.

4. **Dynamic Programming Update:** For each pair, update f[i][j] to be the maximum value between the existing f[i][j] and f[k][j] + 1. This update reflects the following logic: if an arithmetic subsequence ending at nums[k] with the same common difference exists, then appending nums[i] to it will create or extend an arithmetic subsequence by one element.

5. **Track the Maximum Length:** As the algorithm updates the dynamic programming table f, it also keeps track of the overall maximum length found so far in a variable ans. After considering all pairs and differences, ans will hold the length of the longest arithmetic subsequence in nums.

6. **Return the Result:** Once all elements have been considered, and the dynamic programming table has been fully populated, return the value of ans.

In this particular problem, the use of dynamic programming is crucial because we're looking for subsequences, not subsequences within a contiguous slice of the array. Thus, the computational savings from not re-computing the length at each step are especially significant. The 2D array f is a classic example of a dynamic programming table where each entry builds upon the solutions of the smaller sub-problems.

Example Walkthrough

Let's illustrate the solution approach using a small example with the array nums = [2, 4, 6, 8, 10].

1. **Initial Setup:**
   - Initialize f, a 2D array of size n x 1001, where n is the length of nums. In our case, n is 5, so we have a 5 x 1001 array f.
   - Set each value in f to 1, since any single number is an arithmetic subsequence of length 1.

2. **Looping Through Pairs:** Start with the second element in nums and compare it with each element before it. Here we start with 4.

3. **Compute Difference and Update:**
   - Compare 4 (at index i=1) with 2 (at index k=0).
   - Calculate difference diff = 4 - 2 = 2, offset by 500 to get j = 502.
   - Update f[1][502] to be the maximum of f[1][502] and f[0][502] + 1 which is f[0][502] + 1 because f[0][502] represents the length of the longest subsequence ending with 2 that can be extended by 4.
   - The 2D array f now has f[1][502] = 2.

4. **Continue Looping:**
   - Move to the next element 6 (at index i=2) and compare it with all previous elements.
   - Compare it with 4 (at index k=1), calculate the diff = 6 - 4 = 2, and update the dynamic programming table accordingly.
   - Similarly, compare 6 with 2 (at index k=0) and perform the update since they also form an arithmetic sequence with the same difference.
   - This process continues for 8 and 10, each time updating the dynamic programming table based on the calculated differences.

5. **Track the Maximum Length:**
   - While updating f[i][j], we also keep track of the maximum length so far. After the first iteration with 4, we have ans = 2.
   - When we process 6, we find that f[2][502] becomes 3, updating ans = 3, and so on.

6. **Final Result:**
   - By the time we reach 10, the f table contains f[4][502] = 5, and thus ans = 5.
   - Since we have not found any longer subsequences, the final result, which is the length of the longest arithmetic subsequence in nums, is 5.

Through this example, the dynamic programming table is updated systematically, using the properties of arithmetic subsequences. By the end of the process, f holds all the needed information to determine the length of the longest arithmetic subsequence, which in this case is the entire array itself due to its arithmetic nature.

Python Solution

```python
1  from typing import List
2
3  class Solution:
4      def longestArithSeqLength(self, nums: List[int]) -> int:
5          # Find the length of the input list of numbers
6          num_count = len(nums)
7
8          # Initialize a 2D array with all elements set to 1. The dimensions are 'num_count' by 1001.
9          # The reason for 1001 is to have a range of potential differences between numbers (-500 to 500),
10         # adjusted by adding 500 to avoid negative indices.
11         dp_table = [[1] * 1001 for _ in range(num_count)]
12
13         # 'max_length' will hold the length of the longest arithmetic subsequence found
14         max_length = 0
15
16         # Iterate over all pairs of elements to build the dynamic programming table
17         for i in range(1, num_count):
18             for j in range(i):
19
20                 # Compute the index to represent the difference between elements nums[i] and nums[j]
21                 # the difference 'nums[i] - nums[j]' will be shifted by 500 to make
22                 # sure the index is non-negative
23                 diff_index = nums[i] - nums[j] + 500
24
25                 # Update the dynamic programming table to store the length
26                 # of the longest arithmetic subsequence ending with nums[i]
27                 # with the common difference 'nums[i] - nums[j]'
28                 dp_table[i][diff_index] = max(dp_table[i][diff_index], dp_table[j][diff_index] + 1)
29
30                 # Update the overall maximum length with the current length if it's greater
31                 max_length = max(max_length, dp_table[i][diff_index])
32
33         # Return the length of the longest arithmetic subsequence
34         return max_length
```

Java Solution

```java
1  class Solution {
2      public int longestArithSeqLength(int[] nums) {
3          // The length of the input array.
4          int length = nums.length;
5          // Variable to keep track of the maximum arithmetic sequence length found so far.
6          int maxSequenceLength = 0;
7          // A 2D array to keep track of arithmetic sequence lengths. The second dimension has a size of 1001
8          // to cover all possible differences (including negative differences, hence the offset of 500).
9          int[][] dpTable = new int[length][1001];
10
11         // Loop through the elements of the array starting from the second element.
12         for (int i = 1; i < length; ++i) {
13             // Inner loop through all the previous elements to calculate possible differences.
14             for (int k = 0; k < i; ++k) {
15                 // Calculate the difference between current and previous element,
16                 // and add 500 to handle negative differences.
17                 int diff = nums[i] - nums[k] + 500;
18                 // Update the dpTable for the current sequence. If we've seen this difference before,
19                 // increment the length of the sequence. Otherwise, start with length 1.
20                 dpTable[i][diff] = Math.max(dpTable[i][diff], dpTable[k][diff] + 1);
21                 // Update the maximum length found so far.
22                 maxSequenceLength = Math.max(maxSequenceLength, dpTable[i][diff]);
23             }
24         }
25         // Add 1 to the result as the sequence length is one more than the count of differences.
26         return maxSequenceLength + 1;
27     }
28  }
```

C++ Solution

```cpp
1  #include <vector>
2  #include <cstring>
3  #include <algorithm>
4
5  class Solution {
6  public:
7      int longestArithSeqLength(vector<int>& nums) {
8          int n = nums.size(); // Get the number of elements in nums vector
9
10         // Define the 2D array to store the length of the arithmetic sequence
11         int dp[n][1001];
12         // Initialize the dp array with all elements to 0
13         std::memset(dp, 0, sizeof(dp));
14
15         int longestSequence = 0; // Variable to store the length of the longest arithmetic sequence
16         for (int i = 1; i < n; ++i) { // Iterate over each starting element
17             for (int k = 0; k < i; ++k) { // Compare with each element before the i-th element
18
19                 // Compute the difference between current and previous element
20                 // Offset by 500 to handle negative differences since array indices must be non-negative
21                 int diffIndex = nums[i] - nums[k] + 500;
22
23                 // Look for the existing length of an sequence with this difference
24                 // and extend it by 1. Use a maximum of 1 if no such sequence exists before.
25                 // This effectively builds the arithmetic sequence length (ASL) table.
26                 dp[i][diffIndex] = std::max(dp[i][diffIndex], dp[k][diffIndex] + 1);
27
28                 // Update the longest arithmetic sequence length found so far.
29                 longestSequence = std::max(longestSequence, dp[i][diffIndex]);
30             }
31         }
32         return longestSequence + 1; // ASL is the last index in the sequence + 1 for the sequence start
33     }
34  };
```

Typescript Solution

```typescript
1  function longestArithSeqLength(nums: number[]): number {
2      // The length of the input array
3      const numsLength = nums.length;
4      // Variable to keep track of the longest arithmetic subsequence length
5      let longestSubseqLength = 0;
6      // An array of arrays to store the length of an arithmetic subsequence that ends at index i
7      // with common difference (nums[i] - nums[k]) of 500 offset by 'j'.
8      const dpTable: number[][] = Array.from({ length: numsLength }, () => new Array(1001).fill(0));
9
10     // Iterating through the array starting from the 2nd element
11     for (let i = 1; i < numsLength; ++i) {
12         // Compare the current element with all previous elements to find the longest
13         // arithmetic subsequence that can be formed using this element.
14         for (let k = 0; k < i; ++k) {
15             // Calculate the index 'j' representing the common difference with an offset
16             // to allow negative differences. The offset is chosen as 500, which should be
17             // enough given the problem constraints specifying that elements are in the range -500 to 500.
18             const differenceIndex = nums[i] - nums[k] + 500;
19             // Update the dpTable row for element 'i' at the calculated difference index 'j'.
20             // The value is the maximum length found so far for this difference, plus one for the current pair,
21             dpTable[i][differenceIndex] = Math.max(dpTable[i][differenceIndex], dpTable[k][differenceIndex] + 1);
22             // Update the result with the maximum length found so far,
23             longestSubseqLength = Math.max(longestSubseqLength, dpTable[i][differenceIndex]);
24         }
25     }
26     // Since we counted differences, we add 1 to include the starting element of the subsequence.
27     return longestSubseqLength + 1;
28  }
```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm is $O(n^2 + d)$, where n is the number of elements in nums and d is a constant representing the range of possible differences (limited to 1000 in this case). The primary operation that contributes to the time complexity is the nested loops, with the outer loop running n times and the inner loop also running m times, but only up to the current index of the outer loop. For each pair of elements, we are calculating the difference and updating the state in constant time, which contributes the d factor, but since d is a constant (1001 in this case), we can simplify the time complexity to $O(n^2)$.

Space Complexity

The space complexity of the algorithm is $O(n \cdot d)$, where n is the number of elements in nums and d is the constant range of possible differences. We are using a 2D list f of size n x 1001 to keep track of the lengths of arithmetic sequences. The space usage is directly proportional to the size of this list, so we can conclude that the space complexity is linear with respect to n and constant d, simplifying to $O(n)$.