1218. Longest Arithmetic Subsequence of Given Difference

Given an integer array `arr` and an integer `difference`, return the length of the longest subsequence in `arr` which is an arithmetic sequence such that the difference between adjacent elements in the subsequence equals `difference`.

A **subsequence** is a sequence that can be derived from `arr` by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: `arr = [1,2,3,4], difference = 1`
Output: 4
Explanation: The longest arithmetic subsequence is `[1,2,3,4]`.

Example 2:

Input: `arr = [1,3,5,7], difference = 1`
Output: 1
Explanation: The longest arithmetic subsequence is any single element.

Example 3:

Input: `arr = [1,5,7,8,5,3,4,2,1], difference = -2`
Output: 4
Explanation: The longest arithmetic subsequence is `[7,5,3,1]`.

Constraints:

- $1 \leq$ `arr.length` $\leq 10^5$
- $-10^4 \leq$ `arr[i], difference` $\leq 10^4$

Solution

For this problem, let's think of building the subsequence one integer at a time. If our subsequence is currently ending with the integer k, our next integer will have to be $k +$ `difference`. Since subsequences must follow the relative order of the original array, we want to pick the next closest value of $k +$ `difference` and append it to our subsequence. We can also observe that appending the closest value of $k +$ `difference` will give us more options for the next addition to our subsequence.

To find the longest subsequence however, we can try all possible starting positions for our subsequence and construct it greedily with the method mentioned above. Out of all subsequences, we'll return the length of the longest one.

Example

For this example, we'll start the sequence at index 1 and `difference` is 2.

Our subsequence starts with 3 and we're looking for $3 +$ `difference` $= 5$. It appears at indices 4, 7, and 9. We want the next closest position so we'll pick the 5 at index 4. We apply the same idea to then pick 7 at index 5 and finally 9 at index 11.

Let N represent `arr.length`.

Since building the subsequence takes $\mathcal{O}(N)$ and we build $\mathcal{O}(N)$ subsequences (one for each starting position), this algorithm runs in $\mathcal{O}(N^2)$.

For a faster solution, we'll use dynamic programming. We know that to extend subsequence ending with k, we'll find the next closest element with value $k +$ `difference` and add it into our subsequence. We can also think of this idea from the other direction. To find the subsequence ending with $k +$ `difference` at some position, we'll look for the previous closest element k. Then, we'll take the longest subsequence ending with that specific element k and append $k +$ `difference` to obtain our desired subsequence. This idea uses solutions from sub-problems to solve a larger problem, which is essentially dynamic programming.

Let `dp[i]` represent the length of the longest subsequence with the last element at index i.

Let `j` represent the previous closest index of the value `arr[i] - difference`. If `j` exists, then `dp[i] = dp[j] + 1` since we take the longest subsequence ending at index `j` and append `arr[i]` to it. Otherwise, our subsequence is just `arr[i]` itself so `dp[i] = 1`.

We can maintain the previous closest index of integers with a hashmap. The hashmap will use a key for the integer and the value will be the closest index of that integer. Everytime we process `dp[i]` for some index i, we'll update `arr[i]` into our hashmap. Our final answer is the maximum value among all values in `dp`.

Time Complexity

Since we take $\mathcal{O}(1)$ to calculate `dp[i]` for one index i, our time complexity is $\mathcal{O}(N)$ since `dp` has a length of $\mathcal{O}(N)$.

Time Complexity: $\mathcal{O}(N)$

Space Complexity

Our `dp` array and hashmap both use $\mathcal{O}(N)$ memory so our space complexity is $\mathcal{O}(N)$.

Space Complexity: $\mathcal{O}(N)$

```cpp
class Solution {
   public:
    int longestSubsequence(vector<int>& arr, int difference) {
        int n = arr.size();
        unordered_map<int, int> prevIndex;  // keeps track of closest index of integer
        vector<int> dp(n);
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int prevNum = arr[i] - difference;
            if (prevIndex.count(prevNum)) {  // checks if prevNum was processed
                dp[i] = dp[prevIndex[prevNum]] + 1;
            } else {
                dp[i] = 1;
            }
            prevIndex[arr[i]] = i;  //  the closest previous index of arr[i] is always i at this point
            ans = max(ans, dp[i]);
        }
        return ans;
    }
};
```

```java
class Solution {
    public int longestSubsequence(int[] arr, int difference) {
        int n = arr.length;
        HashMap<Integer, Integer> prevIndex = new HashMap(); // keeps track of closest index of integer
        int[] dp = new int[n];
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int prevNum = arr[i] - difference;
            if (prevIndex.containsKey(prevNum)) { // checks if prevNum was processed
                dp[i] = dp[prevIndex.get(prevNum)] + 1;
            } else {
                dp[i] = 1;
            }
            prevIndex.put(arr[i], i);  //  the closest previous index of arr[i] is always i at this point
            ans = Math.max(ans, dp[i]);
        }
        return ans;
    }
}
```

```python
class Solution:
    def longestSubsequence(self, arr: List[int], difference: int) -> int:
        n = len(arr)
        prevIndex = {}  # keeps track of closest index of integer
        dp = [0] * n
        ans = 0
        for i in range(n):
            prevNum = arr[i] - difference
            if prevNum in prevIndex:  # checks if prevNum was processed
                dp[i] = dp[prevIndex[prevNum]] + 1
            else:
                dp[i] = 1
            prevIndex[arr[i]] = i  #  the closest previous index of arr[i] is always i at this point
            ans = max(ans, dp[i])
        return ans
```