446. Arithmetic Slices II - Subsequence

Jan. 1, 2018 | 15.5K views

Average Rating: 3.91 (23 votes)

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequences:

```
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
```

The following sequence is not arithmetic.

```
1, 1, 2, 5, 7
```

A zero-indexed array A consisting of N numbers is given. A **subsequence** slice of that array is any sequence of integers (P_0, P_1, ..., P_k) such that $0 \le P_0 < P_1 < ... < P_k < N$.

A **subsequence** slice (P_0, P_1, ..., P_k) of array A is called arithmetic if the sequence $A[P_0]$, $A[P_1]$, ..., $A[P_{k-1}]$, $A[P_k]$ is arithmetic. In particular, this means that $k \ge 2$.

The function should return the number of arithmetic subsequence slices in the array A.

The input contains N integers. Every integer is in the range of -2^{31} and $2^{31}-1$ and $0 \le N \le 1000$. The output is guaranteed to be less than $2^{31}-1$.

Example:

```
Input: [2, 4, 6, 8, 10]

Output: 7

Explanation:
All arithmetic subsequence slices are:
[2,4,6]
[4,6,8]
[6,8,10]
[2,4,6,8]
[4,6,8,10]
[2,4,6,8,10]
[2,6,10]
```

Approach #1 Brute Force [Time Limit Exceeded]

Intuition

Enumerate all possible subsequences to see if they are arithmetic sequences.

Algorithm

We can use depth-first search to generate all subsequences. We can check a Subsequence is arithmetic or not by its definition.

C++ Java

Copy

```
#define ll long long
class Solution {
public:
  int n;
  int ans;
  void dfs(int dep, vector<int>& A, vector<ll> cur) {
    if (dep == n) {
      if (cur.size() < 3) {
        return;
      }
      for (int i = 1; i < cur.size(); i++) {
        if (cur[i] - cur[i - 1] != cur[1] - cur[0]) {
          return;
        }
      }
      ans ++;
      return;
    }
    dfs(dep + 1, A, cur);
    cur.push_back(A[dep]);
    dfs(dep + 1, A, cur);
  }
  int numberOfArithmeticSlices(vector<int>& A) {
    n = A.size();
    ans = 0;
    vector<ll> cur;
    dfs(0, A, cur);
    return ans;
  }
};
```

Complexity Analysis

- Time complexity : $O(2^n)$. For each element in the array, it can be in or outside the subsequence. So the time complexity is $O(2^n)$.
- Space complexity : $O(n)$. We only need the space to store the array.

Approach #2 Dynamic Programming [Accepted]

Intuition

To determine an arithmetic sequence, we need at least two parameters: the first (or last) element of the sequence, and the common difference.

Algorithm

Starting from this point, we can easily figure out that one state representation that may work:

> `f[i][d]` denotes the number of arithmetic subsequences that ends with `A[i]` and its common difference is `d`.

Let's try to find the state transitions based on the representation above. Assume we want to append a new element `A[i]` to existing arithmetic subsequences to form new subsequences. We can append `A[i]` to an existing arithmetic subsequence, only if the difference between the sequence's last element and `A[i]` is equal to the sequence's common difference.

Thus, we can define the state transitions for the element `A[i]` intuitively :

> for all `j` < `i`, f[i][A[i] - A[j]] += f[j][A[i] - A[j]].

This demonstrates the appending process above to form new arithmetic subsequences.

But here comes the problem, initially all `f[i][d]` are set to be `0`, but how can we form a new arithmetic subsequence if there are no existing subsequences before?

In the original definition of arithmetic subsequences, the length of the subsequence must be at least `3`. This makes it hard to form new subsequences if only two indices `i` and `j` are given. How about taking the subsequences of length `2` into account?

We can define `weak arithmetic subsequences` as follows:

> **Weak arithmetic subsequences** are subsequences that consist of at least two elements and if the difference between any two consecutive elements is the same.

There are two properties of weak arithmetic subsequences that are very useful:

- For any pair `i, j (i != j)`, `A[i]` and `A[j]` can always form a weak arithmetic subsequence.
- If we can append a new element to a weak arithmetic subsequence and keep it arithmetic, then the new subsequence must be an arithmetic subsequence.

The second property is quite trivial, because the only difference between arithmetic subsequences and weak arithmetic subsequences is their length.

Thus we can change the state representations accordingly:

> `f[i][d]` denotes the number of weak arithmetic subsequences that ends with `A[i]` and its common difference is `d`.

Now the state transitions are quite straightforward:

> for all `j` < `i`, f[i][A[i] - A[j]] += (f[j][A[i] - A[j]] + 1).

The `1` appears here because of the property one, we can form a new weak arithmetic subsequence for the pair `(i, j)`.

Now the number of all weak arithmetic subsequences is the sum of all `f[i][d]`. But how can we get the number of arithmetic subsequences that are not `weak`?

There are two ways:

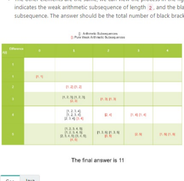
- First, we can count the number of `pure weak` arithmetic subsequences directly. The `pure weak` arithmetic subsequences are the arithmetic subsequences of length `2`, so the number of `pure weak` arithmetic subsequences should be equal to the number of pairs `(i, j)`, which is $\binom{n}{2} = \frac{n(n-1)}{2}$.
- Second, for the summation `f[i][A[i] - A[j]] += (f[j][A[i] - A[j]] + 1)`. `f[i][A[i] - A[j]]` is the number of existing weak arithmetic subsequences, while `1` is the new subsequence built with `A[i]` and `A[j]`. Based on property two, when we are appending new elements to existing weak arithmetic subsequences, we are forming arithmetic subsequences. So the first part, `f[i][A[i] - A[j]]`, is the number of new formed arithmetic subsequences, and can be added to the answer.

We can use the following example to illustrate the process:

```
[1, 1, 2, 3, 4, 5]
```

We need to count the answer for the above sequence.

- For the first element `1`, there is no element in front of it, the answer remains `0`.
- For the second element `1`, the element itself with the previous `1` can form a pure weak arithmetic subsequence with common difference `0` : `[1, 1]`.
- For the third element `2`, it cannot be appended to the only weak arithmetic sequence `[1, 1]`, so the answer remains `0`. Similar to the second element, it can form new weak arithmetic subsequences `[1, 2]` and `[1, 2]`.
- For the forth element `3`, if we append it to some arithmetic subsequences ending with `2`, these subsequences must have a common difference of `3 - 2 = 1`. Indeed there are two: `[1, 2]` and `[1, 2]`. So we can append `3` to the end of these subsequences, and the answer is added by `2`. Similar to above, it can form new weak arithmetic subsequences `[1, 3]`, `[1, 3]` and `[2, 3]`.
- The other elements are the same, we can view the process in the figure below. The red bracket indicates the weak arithmetic subsequence of length `2`, and the black bracket indicates the arithmetic subsequence. The answer should be the total number of black brackets.



The final answer is 11

C++ Java

Copy

```
#define ll long long
class Solution {
public:
  int numberOfArithmeticSlices(vector<int>& A) {
    int n = A.size();
    ll ans = 0;
    vector<map<ll, int>> cnt(n);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < i; j++) {
        ll delta = (ll)A[i] - (ll)A[j];
        int sum = 0;
        if (cnt[j].find(delta) != cnt[j].end()) {
          sum = cnt[j][delta];
        }
        cnt[i][delta] += sum + 1;
        ans += sum;
      }
    }
    return (int)ans;
  }
};
```

Complexity Analysis

- Time complexity : $O(n^2)$. We can use double loop to enumerate all possible states.
- Space complexity : $O(n^2)$. For each `i`, we need to store at most `n` distinct common differences, so the total space complexity is $O(n^2)$.

Rate this article:

Previous Next