<https://leetcode.com/problems/arithmetic-slices/description/>

An integer array 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, [1,3,5,7,9], [7,7,7,7], and [3,-1,-5,-9] are arithmetic sequences.

Given an integer array nums, return the number of arithmetic subarrays of nums.

A subarray is a contiguous subsequence of the array.

**Example 1:**

**Input**: nums = [1,2,3,4]

**Output**: 3

**Explanation**: We have 3 arithmetic slices in nums: [1, 2, 3], [2, 3, 4] and [1,2,3,4] itself.

**Example 2**:

**Input**: nums = [1]

**Output**: 0

**Constraints:**

1 <= nums.length <= 5000

-1000 <= nums[i] <= 1000

**Attempt 1: 2023-03-02**

**Solution 1: Native DFS (30 min)**

**Wrong Solution**

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int count = 0;

        for(int i = 0; i < n - 2; i++) {

            count += helper(nums, i, n);

        }

        return count;

    }

    private int helper(int[] nums, int index, int n) {

        // The wrong base condition which failed on an early stop:

        // Test out by: nums = {1,2,3,4}, expected = 3, output = 2, missing {1,2,3,4}

        // as one arithmetic subarrays

        // -> if we have below condition, it will only find and stop at {1,2,3}

        // and failed to continue find {1,2,3,4} because of an early stop

        // by directly 'return 1', the correct base condition is just not

        // stop here and not directly 'return 1' here, it will allow an

        // accumulate '+ 1' in later recursion by invoke at the end of

        // current DFS recursion such as here '1 + helper(nums, index + 1, n)'

        if(index < n - 2 && (nums[index + 1] - nums[index] == nums[index + 2] - nums[index + 1])) {

            return 1;

        }

        if(index == n - 2 || nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]) {

            return 0;

        }

        return 1 + helper(nums, index + 1, n);

    }

}

**Correct Solution**

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // The difference between two adjacent number should

            // apply to whole subarray

            int diff = nums[i + 1] - nums[i];

            // For any candidate start index 'i', its next and

            // next's next index is 'i + 1' and 'i + 2'

            count += helper(nums, i + 2, i + 1, n, diff);

        }

        return count;

    }

    private int helper(int[] nums, int thirdIndex, int secIndex, int n, int diff) {

        if(thirdIndex == n || nums[thirdIndex] - nums[secIndex] != diff) {

            return 0;

        }

        return 1 + helper(nums, thirdIndex + 1, thirdIndex, n, diff);

    }

}

============================================================================

// We can save one more element during DFS recursion:

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // The difference between two adjacent number should

            // apply to whole subarray

            int diff = nums[i + 1] - nums[i];

            // For any candidate start index 'i', its next and

            // next's next index is 'i + 1' and 'i + 2', and

            // since relation between 'i + 1' and 'i + 2' is

            // permanentely stable, we don't need to pass in

            // both in DFS recursion, only need one represents

            // the both 2nd and 3rd element in {i'th, i+1'th, i+2'th}

            count += helper(nums, i + 2, n, diff);

        }

        return count;

    }

    private int helper(int[] nums, int thirdIndex, int n, int diff) {

        if(thirdIndex == n || nums[thirdIndex] - nums[thirdIndex - 1] != diff) {

            return 0;

        }

        return 1 + helper(nums, thirdIndex + 1, n, diff);

    }

}

============================================================================

// Promote to even less parameters in DFS recursion and its more intuitive

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // We can even remove 'diff' and convert 'thirdIndex' to

            // regular intuitive way as 'firstIndex'(rename as 'index')

            // DFS recursion base condition correspondingly change from

            // 'thirdIndex == n' to 'firstIndex == n - 2'(or 'index == n - 2')

            count += helper(nums, i, n);

        }

        return count;

    }

    private int helper(int[] nums, int index, int n) {

        if(index == n - 2 || nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]) {

            return 0;

        }

        return 1 + helper(nums, index + 1, n);

    }

}

Time Complexity: O(N^2)

Space Complexity: O(N)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        Integer[] memo = new Integer[n + 1];

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // The difference between two adjacent number should

            // apply to whole subarray

            int diff = nums[i + 1] - nums[i];

            // For any candidate start index 'i', its next and

            // next's next index is 'i + 1' and 'i + 2'

            count += helper(nums, i + 2, i + 1, n, diff, memo);

        }

        return count;

    }

    private int helper(int[] nums, int thirdIndex, int secIndex, int n, int diff, Integer[] memo) {

        if(thirdIndex == n || nums[thirdIndex] - nums[secIndex] != diff) {

            return 0;

        }

        if(memo[thirdIndex] != null) {

            return memo[thirdIndex];

        }

        return memo[thirdIndex] = 1 + helper(nums, thirdIndex + 1, thirdIndex, n, diff, memo);

    }

}

============================================================================

// We can save one more element during DFS recursion:

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        Integer[] memo = new Integer[n + 1];

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // The difference between two adjacent number should

            // apply to whole subarray

            int diff = nums[i + 1] - nums[i];

            // For any candidate start index 'i', its next and

            // next's next index is 'i + 1' and 'i + 2', and

            // since relation between 'i + 1' and 'i + 2' is

            // permanentely stable, we don't need to pass in

            // both in DFS recursion, only need one represents

            // the both 2nd and 3rd element in {i'th, i+1'th, i+2'th}

            count += helper(nums, i + 2, n, diff, memo);

        }

        return count;

    }

    private int helper(int[] nums, int thirdIndex, int n, int diff, Integer[] memo) {

        if(thirdIndex == n || nums[thirdIndex] - nums[thirdIndex - 1] != diff) {

            return 0;

        }

        if(memo[thirdIndex] != null) {

            return memo[thirdIndex];

        }

        return memo[thirdIndex] = 1 + helper(nums, thirdIndex + 1, n, diff, memo);

    }

}

============================================================================

// Promote to even less parameters in DFS recursion and its more intuitive

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        Integer[] memo = new Integer[n + 1];

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // We can even remove 'diff' and convert 'thirdIndex' to

            // regular intuitive way as 'firstIndex'(rename as 'index')

            // DFS recursion base condition correspondingly change from

            // 'thirdIndex == n' to 'firstIndex == n - 2'(or 'index == n - 2')

            count += helper(nums, i, n, memo);

        }

        return count;

    }

    private int helper(int[] nums, int index, int n, Integer[] memo) {

        if(index == n - 2 || nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        return memo[index] = 1 + helper(nums, index + 1, n, memo);

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/arithmetic-slices/solutions/1816907/dp-based-approach-recursion-memoization-c-clean-code/>

class Solution {

public:

int solve(vector<int>& nums, int idx, int prev, int d, int n)

{

if(idx == n || nums[idx] - prev != d) return 0;

return 1 + solve(nums, idx+1, nums[idx], d, n);

}

int numberOfArithmeticSlices(vector<int>& nums) {

int n = nums.size();

if(n < 3) return 0;

int countSubarray = 0;

for(int i=0; i<n-2; i++)

{

int d = nums[i+1] - nums[i];

countSubarray += solve(nums, i+2, nums[i+1], d, n);

}

return countSubarray;

}

};

**Solution 3: 1D DP (10 min)**

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int[] dp = new int[n];

        int count = 0;

        for(int i = n - 3; i >= 0; i--) {

            if(nums[i + 2] - nums[i + 1] == nums[i + 1] - nums[i]) {

                dp[i] = dp[i + 1] + 1;

            }

            count += dp[i];

        }

        return count;

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

============================================================================

// The above 1D DP solution translates from below Native DFS solution:

// bottom: n - 2 -> 0 (mapping to dp[n - 2] = 0, dp[n - 1] = 0)

// top: 0 to n - 3 when 'nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]' -> 1

// (mapping to i from n - 3 to 0, when (nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]) {dp[i] = dp[i + 1] + 1}

class Solution {

    public int numberOfArithmeticSlices(int[] nums) {

        int n = nums.length;

        if(n < 3) {

            return 0;

        }

        int count = 0;

        // Enumerate all indexes from 0 to n - 3 as candidate

        // start index of a arithmetic subarray

        for(int i = 0; i < n - 2; i++) {

            // We can even remove 'diff' and convert 'thirdIndex' to

            // regular intuitive way as 'firstIndex'(rename as 'index')

            // DFS recursion base condition correspondingly change from

            // 'thirdIndex == n' to 'firstIndex == n - 2'(or 'index == n - 2')

            count += helper(nums, i, n);

        }

        return count;

    }

    private int helper(int[] nums, int index, int n) {

        if(index == n - 2 || nums[index + 1] - nums[index] != nums[index + 2] - nums[index + 1]) {

            return 0;

        }

        return 1 + helper(nums, index + 1, n);

    }

}

**Refer to**

<https://leetcode.com/problems/arithmetic-slices/solutions/215861/detailed-explanation-two-dp-solutions/>

1st DP Solution: time O(n^2), space: O(n^2)

We find the sub problem:

Assume A[i:j] (both include A[i] and A[j]) is an arithmetic slice, then we have:

if A[i]-A[i-1] = = A[i+1]-A[i], then A[i-1:j] is an arithmetic slice;

if A[j+1]-A[j] = = A[j]-A[j-1], then A[i:j+1] is an arithmetic slice.

use dp[i][j] to memorize whether A[i:j] is an arithmetic slice, and count to count the num of arithmetic slices:

public int numberOfArithmeticSlices(int[] A) {

int n=A.length;

if(n<3){return 0;}

boolean[][] dp=new boolean[n][n]; //initial value is false

int count=0;

for(int i=0;i<n-3+1;i++){

if((A[i+1]-A[i])==(A[i+2]-A[i+1])){

dp[i][i+3-1]=true;

count++;

}

}

for(int k=4;k<=n;k++){

for (int i=0;i<n-k+1;i++){

int j=i+k-1;

if(dp[i+1][j]==true&&(A[i+1]-A[i]==A[i+2]-A[i+1])){

dp[i][j]=true;

count++;

}else if(dp[i][j-1]==true&&(A[j]-A[j-1]==A[j-1]-A[j-2])){

dp[i][j]=true;

count++;

}

}

}

return count;

}

2nd DP Solution: time O(n), space O(n)

We can find another sub problem: assume dp[i] is the number of arithmetic slices which are end with A[i]. then we have:

dp[i]=(A[i]-A[i-1] = = A[i-1]-A[i-2])? 1+dp[i-1] : 0, the code:

public int numberOfArithmeticSlices(int[] A) {

int n=A.length;

if(n<3){return 0;}

int[] dp=new int[n];

dp[0]=0;

dp[1]=0;

int sum=0;

for(int i=2;i<n;i++){

if((A[i]-A[i-1])==(A[i-1]-A[i-2])){

dp[i]=dp[i-1]+1;

}else{

dp[i]=0;

}

sum+=dp[i];

}

return sum;

}

up to now, time complexity is O(n), but space complexity is also O(n). In fact, we only need a curr to memorize the num of arithmetic slices which end with current A[i] and a sum to memorize num of all curr. that is @icl7722's solution, which time complexity is O(n) and space complexity is O(1).

public int numberOfArithmeticSlices(int[] A) {

int curr = 0, sum = 0;

for (int i=2; i<A.length; i++)

if (A[i]-A[i-1] == A[i-1]-A[i-2]) {

curr += 1;

sum += curr;

} else {

curr = 0;

}

return sum;

}