# 413. 等差数列划分

如果一个数列 至少有三个元素，并且任意两个相邻元素之差相同，则称该数列为等差数列。

例如，[1,3,5,7,9]、[7,7,7,7] 和 [3,-1,-5,-9] 都是等差数列。
给你一个整数数组 nums ，返回数组 nums 中所有为等差数组的 子数组 个数。

子数组 是数组中的一个连续序列。

输入：nums = [1,2,3,4]
输出：3
解释：nums 中有三个子等差数组：[1, 2, 3]、[2, 3, 4] 和 [1,2,3,4] 自身。

# 【负雪明烛】暴力 => 双指针 => 递归 => 动态规划

https://leetcode-cn.com/problems/arithmetic-slices/solution/fu-xue-ming-zhu-bao-li-shuang-zhi-zhen-d-fc1l/

本题要我们求数组中有多少个等差数列。
本题解分成了 4 个部分：暴力 => 双指针 => 递归 => 动态规划。

# 暴力

最容易想到的就是暴力解法：判断所有的子数组是不是等差数列，如果是的话就累加次数。

```
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& A) {
        const int N = A.size();
        int res = 0;
        for (int i = 0; i < N - 2; i++) {
            for (int j = i + 1; j < N; j++) {
                if (isArithmetic(A, i, j))
                    res ++;
            }
        }
        return res;
```

```
    }
private:
    bool isArithmetic(vector<int>& A, int start, int end) {
        if (end - start < 2) return false;
        for (int i = start; i < end - 1; i++) {
            if (A[i + 1] * 2 != A[i] + A[i + 2])
                return false;
        }
        return true;
    }
};
```

- 时间复杂度：O(N ^ 3) 遍历所有的子数组，需要有两重循环，时间复杂度是 O(N^2)；判断每个子数组是不是等差数列，时间复杂度是 O(N);所以总的时间复杂度是 O(N ^ 3)
- 空间复杂度：O(1)

# 双指针

在上面的暴力解法中，我们对每个子数组都进行了是否为等差数列的判断。

- 其实，如果我们已经知道一个子数组的前面部分不是等差数列以后，那么后面部分就不用判断了。
- 同时，我们知道等差数列的所有的相邻数字的差是固定的。

因此，对于每个起始位置，我们只需要向后进行一遍扫描，直到不再构成等差数列为止，此时已经没有必要再向后扫描。
这个思路其实就是双指针（滑动窗口）的解法。

```
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& A) {
        const int N = A.size();
        int res = 0;
        for (int i = 0; i < N - 2; i++) {
            int d = A[i + 1] - A[i];
            for (int j = i + 1; j < N - 1; j++) {
                if (A[j + 1] - A[j] == d)
                    res ++;
                else
                    break;
            }
        }
```

```
        return res;
    }
};
```

- 时间复杂度：O(N^2) 空间复杂度：O(1)

# 递归

从上面的思路中，我们已经逐渐的抽象出一个思路了：固定起点，判断后面的等差数列有多少个。

类似的思路，我们可以构造出「自顶向下」的递归解法：定义递归函数 slices(A, end)的含义是区间 A[0, end] 中，以 end 作为终点的，等差数列的个数。

A[0, end]内的以 end 作为终点的等差数列的个数，相当于在 A[0, end - 1]的基础上，增加了 A[end] 。

有两种情况：

1. A[end] - A[end - 1] == A[end - 1] - A[end - 2]时，说明增加的A[end]能和前面构成等差数列，那么 slices(A, end) = slices(A, end - 1) + 1；
2. A[end] - A[end - 1] != A[end - 1] - A[end - 2]时， 说明增加的 A[end]不能和前面构成等差数列，所以slices(A, end) = 0；

最后，我们要求的是整个数组中的等差数列的数目，所以需要把
0 <= end <= len(A - 1)
0<=end<=len(A−1) 的所有递归函数的结果累加起来。

```python
class Solution(object):
    def numberOfArithmeticSlices(self, A):
        N = len(A)
        self.res = 0
        self.slices(A, N - 1)
        return self.res

    def slices(self, A, end):
        if end < 2: return 0
        op = 0
        if A[end] - A[end - 1] == A[end - 1] - A[end - 2]:
            op = 1 + self.slices(A, end - 1)
            self.res += op
        else:
```

```
            self.slices(A, end - 1)
        return op
```

- 时间复杂度：O(N^2)因为递归函数最多被调用 N 次，每次调用的时候都需要向前遍历一次。
- 空间复杂度：O(N)，递归栈的最大深度是 N。

# 动态规划

上面的递归的解法，是「自顶向下」的思路。如果转成「自底向上」的思路，就变成了动态规划。
类似于递归解法，我们定义
dp[i] 是以 A[i] 为终点的等差数列的个数。

类似于上面的递归思路，有两种情况：

1. A[i] - A[i - 1] == A[i - 1] - A[i - 2]时，说明增加的A[i]能和前面构成等差数列，那么 dp[i] = dp[i - 1] + 1;
2. A[i] - A[i - 1] != A[i - 1] - A[i - 2]时， 说明增加的 A[i]不能和前面构成等差数列，所以dp[i] = 0;

动态规划的初始状态：
dp[0] = 0, dp[1] = 0

最后，我们要求的是整个数组中的等差数列的数目，所以需要把
0 <= i <= len(A - 1) 的所有 dp[i] 的结果累加起来。

```python
class Solution(object):
    def numberOfArithmeticSlices(self, A):
        N = len(A)
        dp = [0] * N
        for i in range(1, N - 1):
            if A[i - 1] + A[i + 1] == A[i] * 2:
                dp[i] = dp[i - 1] + 1
        return sum(dp)
```

- 时间复杂度 O(N)；
- 空间复杂度：O(N)

由于 dp[i] 只和 dp[i - 1] 有关，所以可以进行状态压缩，只用一个变量 k 来表示以 A[i] 为终点的等差数列的个数。
计算的方式仍然不变。

```python
class Solution(object):
    def numberOfArithmeticSlices(self, A):
        count = 0
        k = 0
        for i in xrange(len(A) - 2):
            if A[i + 1] - A[i] == A[i + 2] - A[i + 1]:
                k += 1
                count += k
            else:
                k = 0
        return count
```

- 时间复杂度：O(N)
- 空间复杂度：O(1)