# When Array Meet DP

The target of such problem is to find out the continuous sub array, or subsequence which met some condition. We may need to combine the array and the DP in the solution.

The critical point is to associate each element with an **accumulated state** and keep track on only meaningful state.

## 659. Split Array into Consecutive Subsequences

Medium

You are given an integer array sorted in ascending order (may contain duplicates), you need to split them into several subsequences, where each subsequences consist of at least 3 consecutive integers. Return whether you can make such a split.

**Example 1:**

**Input:** [1,2,3,3,4,5]

**Output:** True

**Explanation:**

You can split them into two consecutive subsequences :

1, 2, 3

3, 4, 5

**Example 2:**

**Input:** [1,2,3,3,4,4,5,5]

**Output:** True

**Explanation:**

You can split them into two consecutive subsequences :

1, 2, 3, 4, 5

3, 4, 5

**Example 3:**

**Input:** [1,2,3,4,4,5]

**Output:** False

**Note:**

1. The length of the input is in range of [1, 10000]

/// <summary>

/// Leet code #659. Split Array into Consecutive Subsequences

///

/// You are given an integer array sorted in ascending order (may contain

/// duplicates), you need to split them into several subsequences, where

/// each subsequences consist of at least 3 consecutive integers.

/// Return whether you can make such a split.

///

/// Example 1:

/// Input: [1,2,3,3,4,5]

/// Output: True

/// Explanation:

/// You can split them into two consecutive subsequences :

/// 1, 2, 3

/// 3, 4, 5

///

/// Example 2:

/// Input: [1,2,3,3,4,4,5,5]

/// Output: True

/// Explanation:

/// You can split them into two consecutive subsequences :

/// 1, 2, 3, 4, 5

/// 3, 4, 5

///

/// Example 3:

/// Input: [1,2,3,4,4,5]

/// Output: False

///

/// Note:

/// 1. The length of the input is in range of [1, 10000]

/// </summary>

bool LeetCode::isPossible(vector<int>& nums)

{

unordered\_map<int, priority\_queue<int, std::vector<int>, std::greater<int>>> heap\_map;

for (size\_t i = 0; i < nums.size(); i++)

{

if (heap\_map.count(nums[i] - 1) == 0)

{

heap\_map[nums[i]].push(1);

}

else

{

int length = heap\_map[nums[i] - 1].top();

heap\_map[nums[i] - 1].pop();

if (heap\_map[nums[i] - 1].empty())

{

heap\_map.erase(nums[i] - 1);

}

heap\_map[nums[i]].push(length + 1);

}

}

for (auto itr : heap\_map)

{

if (itr.second.top() < 3) return false;

}

return true;

}

## 446. Arithmetic Slices II - Subsequence

Hard

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 (P0, P1, ..., Pk) such that 0 ≤ P0< P1 < ... < Pk < N.

A **subsequence** slice (P0, P1, ..., Pk) of array A is called arithmetic if the sequence A[P0], A[P1], ..., A[Pk-1], A[Pk] is arithmetic. In particular, this means that k ≥ 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 -231 and 231-1 and 0 ≤ N ≤ 1000. The output is guaranteed to be less than 231-1.

**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]

/// <summary>

/// Leet code #446. Arithmetic Slices II - Subsequence

///

/// 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

/// (P0, P1, ..., Pk) such that 0 ≤ P0 < P1 < ... < Pk < N.

///

/// A subsequence slice (P0, P1, ..., Pk) of array A is called

/// arithmetic if the sequence A[P0], A[P1], ..., A[Pk-1], A[Pk]

/// is arithmetic. In particular, this means that k ≥ 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 ≤ N ≤ 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]

/// </summary>

int LeetCode::numberOfArithmeticSlicesII(vector<int>& A)

{

int count = 0;

vector<unordered\_map<int, int>> arithmeticCount(A.size());

for (int i = 0; i < (int)A.size(); i++)

{

for (int j = i - 1; j >= 0; j--)

{

int diff = A[i] - A[j];

arithmeticCount[i][diff] += 1;

if (arithmeticCount[j].count(diff) > 0)

{

arithmeticCount[i][diff] += arithmeticCount[j][diff];

count += arithmeticCount[j][diff];

}

}

}

return count;

}

## 1027. Longest Arithmetic Sequence

Medium

Given an array A of integers, return the **length** of the longest arithmetic subsequence in A.

Recall that a *subsequence* of A is a list A[i\_1], A[i\_2], ..., A[i\_k] with 0 <= i\_1 < i\_2 < ... < i\_k <= A.length - 1, and that a sequence B is *arithmetic* if B[i+1] - B[i] are all the same value (for 0 <= i < B.length - 1).

**Example 1:**

**Input:** [3,6,9,12]

**Output:** 4

**Explanation:**

The whole array is an arithmetic sequence with steps of length = 3.

**Example 2:**

**Input:** [9,4,7,2,10]

**Output:** 3

**Explanation:**

The longest arithmetic subsequence is [4,7,10].

**Example 3:**

**Input:** [20,1,15,3,10,5,8]

**Output:** 4

**Explanation:**

The longest arithmetic subsequence is [20,15,10,5].

**Note:**

1. 2 <= A.length <= 2000
2. 0 <= A[i] <= 10000

/// <summary>

/// Leet code #1027. Longest Arithmetic Sequence

///

/// Given an array A of integers, return the length of the longest arithmetic

/// subsequence in A.

///

/// Recall that a subsequence of A is a list A[i\_1], A[i\_2], ..., A[i\_k] with

/// 0 <= i\_1 < i\_2 < ... < i\_k <= A.length - 1, and that a sequence B is

/// arithmetic if B[i+1] - B[i] are all the same value (for 0 <= i <

/// B.length - 1).

///

/// Example 1:

///

/// Input: [3,6,9,12]

/// Output: 4

/// Explanation:

/// The whole array is an arithmetic sequence with steps of length = 3.

///

/// Example 2:

///

/// Input: [9,4,7,2,10]

/// Output: 3

/// Explanation:

/// The longest arithmetic subsequence is [4,7,10].

///

/// Example 3:

///

/// Input: [20,1,15,3,10,5,8]

/// Output: 4

/// Explanation:

/// The longest arithmetic subsequence is [20,15,10,5].

///

/// Note:

///

/// 1. 2 <= A.length <= 2000

/// 2. 0 <= A[i] <= 10000

/// </summary>

int LeetCode::longestArithSeqLength(vector<int>& A)

{

int result = 0;

vector<unordered\_map<int, int>> arithmeticCount(A.size());

for (size\_t i = 0; i < A.size(); i++)

{

for (size\_t j = 0; j < i; j++)

{

int diff = A[i] - A[j];

if (arithmeticCount[j].count(diff) > 0)

{

arithmeticCount[i][diff] = max(arithmeticCount[i][diff], arithmeticCount[j][diff] + 1);

}

else

{

arithmeticCount[i][diff] = max(arithmeticCount[i][diff], 2);

}

result = max(result, arithmeticCount[i][diff]);

}

}

return result;

}

## 1063. Number of Valid Subarrays

Hard

Given an array A of integers, return the number of **non-empty continuous subarrays** that satisfy the following condition:

The leftmost element of the subarray is not larger than other elements in the subarray.

**Example 1:**

**Input:** [1,4,2,5,3]

**Output:** 11

**Explanation:** There are 11 valid subarrays: [1],[4],[2],[5],[3],[1,4],[2,5],[1,4,2],[2,5,3],[1,4,2,5],[1,4,2,5,3].

**Example 2:**

**Input:** [3,2,1]

**Output:** 3

**Explanation:** The 3 valid subarrays are: [3],[2],[1].

**Example 3:**

**Input:** [2,2,2]

**Output:** 6

**Explanation:** There are 6 valid subarrays: [2],[2],[2],[2,2],[2,2],[2,2,2].

**Note:**

1. 1 <= A.length <= 50000
2. 0 <= A[i] <= 100000

/// <summary>

/// Leet code #1063. Number of Valid Subarrays

///

/// Given an array A of integers, return the number of non-empty continuous

/// subarrays that satisfy the following condition:

///

/// The leftmost element of the subarray is not larger than other elements

/// in the subarray.

///

/// Example 1:

///

/// Input: [1,4,2,5,3]

/// Output: 11

/// Explanation: There are 11 valid subarrays: [1],[4],[2],[5],[3],[1,4],

/// [2,5],[1,4,2],[2,5,3],[1,4,2,5],[1,4,2,5,3].

///

/// Example 2:

///

/// Input: [3,2,1]

/// Output: 3

/// Explanation: The 3 valid subarrays are: [3],[2],[1].

///

/// Example 3:

///

/// Input: [2,2,2]

/// Output: 6

/// Explanation: There are 6 valid subarrays: [2],[2],[2],[2,2],[2,2],[2,2,2].

///

///

/// Note:

///

/// 1. 1 <= A.length <= 50000

/// 2. 0 <= A[i] <= 100000

/// </summary>

int LeetCode::validSubarrays(vector<int>& nums)

{

int result = 0;

stack<int> dp;

for (size\_t i = 0; i < nums.size(); i++)

{

if (dp.empty())

{

dp.push(nums[i]);

}

else

{

while (!dp.empty())

{

if (dp.top() > nums[i])

{

dp.pop();

}

else

{

break;

}

}

dp.push(nums[i]);

}

result += dp.size();

}

return result;

}

# When Array Meet Two Pointers

The group of this problem has a common pattern is to calculated sum of a sub array, with either target for longest length or shortest length.

First to calculate the sum of a subarray we need accumulated sum, so sum of subarray from x to y is sum[y] – sum[x], and to keep the accumulated sum with shortest length, using two pointer is a solution, but we should keep the sum[j] > sum[i] for all i < j.

Another scenario is to get accumulated sum above a target, we need to keep all the steps, with the descending accumulated sum down to the hell.

## 1124. Longest Well-Performing Interval

Medium

We are given hours, a list of the number of hours worked per day for a given employee.

A day is considered to be a *tiring day* if and only if the number of hours worked is (strictly) greater than 8.

A *well-performing interval* is an interval of days for which the number of tiring days is strictly larger than the number of non-tiring days.

Return the length of the longest well-performing interval.

**Example 1:**

**Input:** hours = [9,9,6,0,6,6,9]

**Output:** 3

**Explanation:** The longest well-performing interval is [9,9,6].

**Constraints:**

* 1 <= hours.length <= 10000
* 0 <= hours[i] <= 16

/// <summary>

/// Leet code #1124. Longest Well-Performing Interval

///

/// We are given hours, a list of the number of hours worked per day for a

/// given employee.

/// A day is considered to be a tiring day if and only if the number of hours

/// worked is (strictly) greater than 8.

/// A well-performing interval is an interval of days for which the number of

/// tiring days is strictly larger than the number of non-tiring days.

/// Return the length of the longest well-performing interval.

///

/// Example 1:

/// Input: hours = [9,9,6,0,6,6,9]

/// Output: 3

/// Explanation: The longest well-performing interval is [9,9,6].

///

/// Constraints:

/// 1. 1 <= hours.length <= 10000

/// 2. 0 <= hours[i] <= 16

/// </summary>

int LeetCode::longestWPI(vector<int>& hours)

{

int n = hours.size();

vector<int> dp(n, -1);

int sum = 0;

int result = 0;

for (int i = 0; i < (int)hours.size(); i++)

{

if (hours[i] <= 8)

{

sum--;

}

else

{

sum++;

}

if (sum > 0) result = i + 1;

else if (sum == -n) continue;

else

{

// for zero we do not need to store the position

if (sum < 0) if (dp[sum + n] == -1) dp[sum + n] = i;

if (dp[sum + n - 1] != -1)

{

result = max(result, i - dp[sum + n - 1]);

}

}

}

return result;

}

## 862. Shortest Subarray with Sum at Least K

Hard

Return the **length** of the shortest, non-empty, contiguous subarray of Awith sum at least K.

If there is no non-empty subarray with sum at least K, return -1.

**Example 1:**

**Input:** A = [1], K = 1

**Output:** 1

**Example 2:**

**Input:** A = [1,2], K = 4

**Output:** -1

**Example 3:**

**Input:** A = [2,-1,2], K = 3

**Output:** 3

**Note:**

1. 1 <= A.length <= 50000
2. -10 ^ 5 <= A[i] <= 10 ^ 5
3. 1 <= K <= 10 ^ 9

/// <summary>

/// Leet code #862. Shortest Subarray with Sum at Least K

///

/// Return the length of the shortest, non-empty, contiguous subarray of A

/// with sum at least K.

///

/// If there is no non-empty subarray with sum at least K, return -1.

///

/// Example 1:

/// Input: A = [1], K = 1

/// Output: 1

///

/// Example 2:

/// Input: A = [1,2], K = 4

/// Output: -1

///

/// Example 3:

/// Input: A = [2,-1,2], K = 3

/// Output: 3

///

/// Note:

/// 1. 1 <= A.length <= 50000

/// 2. -10 ^ 5 <= A[i] <= 10 ^ 5

/// 3. 1 <= K <= 10 ^ 9

/// </summary>

int LeetCode::shortestSubarray(vector<int>& A, int K)

{

int result = -1;

vector<int> sums(A.size() + 1);

int total = 0;

for (size\_t i = 0; i <= A.size(); i++)

{

sums[i] = total;

if (i < A.size()) total += A[i];

}

deque<int> window;

size\_t index = 0;

for (size\_t i = 0; i < sums.size(); i++)

{

if (window.size() > 1)

{

int last = window.back();

window.pop\_back();

while (!window.empty() && sums[window.back()] >= sums[last])

{

window.pop\_back();

}

window.push\_back(last);

}

window.push\_back(i);

while (window.size() > 1)

{

int first = window.front();

int last = window.back();

if (sums[last] - sums[first] >= K)

{

result = (result == -1) ? (last - first) : min(result, last - first);

window.pop\_front();

}

else

{

break;

}

}

}

return result;

}

