<https://github.com/doocs/leetcode/blob/main/solution/2300-2399/2355.Maximum%20Number%20of%20Books%20You%20Can%20Take/README_EN.md>

You are given a 0-indexed integer array books of length n where books[i] denotes the number of books on the ith shelf of a bookshelf.

You are going to take books from a **contiguous** **section** of the bookshelf spanning from l to r where 0 <= l <= r < n. For each index i in the range l <= i < r, you must take **strictly fewer** books from shelf i than shelf i + 1.

Return the maximum number of books you can take from the bookshelf.

**Example 1:**

**Input:** books = [8,5,2,7,9]

**Output:** 19

**Explanation:**

- Take 1 book from shelf 1.

- Take 2 books from shelf 2.

- Take 7 books from shelf 3.

- Take 9 books from shelf 4.

You have taken 19 books, so return 19.

It can be proven that 19 is the maximum number of books you can take.

**Example 2:**

**Input:** books = [7,0,3,4,5]

**Output:** 12

**Explanation:**

- Take 3 books from shelf 2.

- Take 4 books from shelf 3.

- Take 5 books from shelf 4.

You have taken 12 books so return 12.

It can be proven that 12 is the maximum number of books you can take.

**Example 3:**

**Input:** books = [8,2,3,7,3,4,0,1,4,3]

**Output:** 13

**Explanation:**

- Take 1 book from shelf 0.

- Take 2 books from shelf 1.

- Take 3 books from shelf 2.

- Take 7 books from shelf 3.

You have taken 13 books so return 13.

It can be proven that 13 is the maximum number of books you can take.

**Constraints:**

1 <= books.length <= 10^5

0 <= books[i] <= 10^5

**Attempt 1: 2024-08-28**

**Solution 1: DFS + Memoization (180 min)**

**Refer to chatGPT by converting below DP back into DFS**

To transform this dynamic programming (DP) solution into a pure Depth-First Search (DFS) approach for Leetcode 2355, we need to make the following changes:

Recursive Structure: Replace the iterative DP structure with a recursive DFS approach. Each recursive call will try to find the maximum number of books starting from a given index i.

Memoization: Use memoization to store already computed results for each index to avoid recalculating the same subproblem, thus improving efficiency.

Simulating Stack with DFS: Use recursion to simulate the logic of the stack that keeps track of the nearest valid previous index (j) from which we can start taking books.

**Pure DFS Solution**

Here's the pure DFS version of the solution based on the provided DP approach:

class Solution {

public long maximumBooks(int[] books) {

int n = books.length;

long[] memo = new long[n];

Arrays.fill(memo, -1); // Initialize memoization array with -1

long maxBooks = 0;

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

maxBooks = Math.max(maxBooks, dfs(i, books, memo));

}

return maxBooks;

}

private long dfs(int i, int[] books, long[] memo) {

if (i < 0) {

return 0; // Base case: Out of bounds

}

if (memo[i] != -1) {

return memo[i]; // Return the already computed value for index i

}

long maxBooks = 0;

// Find the nearest index j such that we can take books from j+1 to i

int j = -1;

for (int k = i - 1; k >= 0; k--) {

if (books[k] < books[i] - (i - k)) {

j = k;

break;

}

}

int lastTook = books[i] - (i - j) + 1;

long sum = 0;

if (lastTook > 1) {

// Sum of books[i] + (books[i] - 1) + ... + lastTook

sum = (long) (books[i] + lastTook) \* (i - j) / 2;

} else {

// Sum of 1 + 2 + ... + books[i]

sum = (long) books[i] \* (books[i] + 1) / 2;

}

if (j >= 0) {

sum += dfs(j, books, memo); // Include books from left subproblem

}

maxBooks = sum;

memo[i] = maxBooks; // Memoize the result for the current index

return maxBooks;

}

}

**Explanation of the DFS Solution**

Initialization: We initialize an array memo to store the maximum number of books that can be taken starting from each index. The array is filled with -1 to indicate uncomputed states.

Recursive DFS Function:

Base Case: If the index is less than 0, return 0 because we are out of bounds.

Memoization Check: If the result for the current index i is already computed, return it to avoid redundant computation.

Finding Valid **j**: We iterate from i-1 down to 0 to find the nearest valid index j such that books[j] < books[i] - (i - j). This simulates the logic of finding the nearest smaller value index using a stack.

Calculate Books: Calculate the number of books that can be taken starting at i:

If lastTook > 1, calculate the sum of an arithmetic series starting with lastTook to books[i].

If lastTook <= 1, calculate the sum directly using books[i] \* (books[i] + 1) / 2.

Recursively Add: If a valid j is found (j >= 0), add the maximum books that can be taken from j using a recursive DFS call.

Memoize the Result: Store the result in the memo array to reuse it later.

Iterate Over All Indices: In the main function, iterate over all indices to find the maximum number of books that can be taken, using the DFS function for each index.

Return the Result: Return the maximum value found, which represents the maximum books that can be taken according to the rules.

**Why This Works**

DFS Recursion: The recursion effectively simulates the iterative DP solution, where each call explores the maximum number of books that can be taken starting from a particular index.

Memoization: The use of a memo array avoids recalculating the same result multiple times, thus improving efficiency.

Simulation of Stack: The logic to find j (the nearest valid smaller value) is embedded in the DFS, replacing the need for an explicit stack.

This approach mirrors the logic of the DP solution while providing a clear recursive structure to explore all possible subproblems.

**Solution 2: DP (180 min)**

**Refer to**

<https://walkccc.me/LeetCode/problems/2355/#__tabbed_1_2>

class Solution {

public long maximumBooks(int[] books) {

// dp[i] := the maximum number of books we can take from books[0..i] with taking all of

// books[i]

long[] dp = new long[books.length];

Deque<Integer> stack = new ArrayDeque<>(); // the possible indices we can reach

for (int i = 0; i < books.length; ++i) {

// We may take all of books[j], where books[j] < books[i] - (i - j).

while (!stack.isEmpty() && books[stack.peek()] >= books[i] - (i - stack.peek()))

stack.pop();

// We can now take books[j + 1..i].

final int j = stack.isEmpty() ? -1 : stack.peek();

final int lastTook = books[i] - (i - j) + 1;

if (lastTook > 1)

// books[i] + (books[i] - 1) + ... + (books[i] - (i - j) + 1)

dp[i] = (long) (books[i] + lastTook) \* (i - j) / 2;

else

// 1 + 2 + ... + books[i]

dp[i] = (long) books[i] \* (books[i] + 1) / 2;

if (j >= 0)

dp[i] += dp[j];

stack.push(i);

}

return Arrays.stream(dp).max().getAsLong();

}

}

**Another DP style**

**Refer to**

<https://algo.monster/liteproblems/2355>

**Problem Description**

In this problem, we are provided with an integer array called books, where each element books[i] represents the number of books on the ith shelf in a bookshelf. The shelves are indexed starting from 0.

The task is to take books from a continuous section of the shelf, where we start from shelf l and end at shelf r inclusive, such that 0 <= l <= r < n. However, while taking books from these shelves, we must adhere to a specific rule: from each shelf i within the section l <= i < r, we must take fewer books than from the shelf i+1. In other words, the number of books taken should strictly increase with each shelf as you move from left to right within the chosen section.

Our goal is to determine the maximum number of books we can take from the shelves while following this rule.

**Intuition**

To solve this problem, we need to find a way to take as many books as possible from each shelf, taking into account the increasing sequence constraint and without missing any of the shelves.

A simple greedy approach of just taking as many books as possible from each shelf won't work here because of the constraint that the number of books taken must strictly increase from one shelf to the next. Also, directly trying to find the maximum for each possible section for l to r can lead to a time-consuming solution, since there could many possible sections.

The solution involves dynamic programming, and the intuition for the solution comes from realizing that the maximum number of books we can take from a shelf i is influenced by the number of books available on shelf i as well as the restrictions applied by the shelves on its left (due to the strictly increasing order requirement).

We transform the problem into a smaller sub-problem by noting that the number of books that we can take from the shelf i is bounded by the difference between the number of books on that shelf and its index i (since the counts must increase). Use this transformed nums array to maintain this offset.

Next, we create a monotonically increasing stack stk that helps us quickly determine the leftmost shelf from which we can start taking books to satisfy the strictly increasing condition up to the current shelf. We iterate through each shelf, and update the left bounds accordingly.

Lastly, we apply dynamic programming (dp) to find the solution. The dp array keeps track of the maximum number of books we can take from the bookshelf ending at each index. For each shelf i, we calculate the maximum number of books we can take from this shelf and potentially the sequence of shelves to its left by using the left bounds and the dp of the previous shelf in the bounds. We keep track of the overall maximum as we calculate the dp for each shelf, which will be our final answer.

The code enforces the constraints by limiting the number of books we can take from each shelf i and ensures strictly increasing order by using the left bounds and the offsets in the nums array. It uses a key dynamic programming insight to build upon the results of smaller sections of shelves to find the answer for larger sections, ultimately leading to the maximum number of books we can take from any section of the bookshelf.

**Solution Approach**

The solution approach uses dynamic programming, an array transformation, a monotonically increasing stack for efficient retrieval of data, and careful calculation of the boundaries for taking books.

Firstly, a transformed array nums is created from the original books array by subtracting the index from the book count for each shelf, i.e., nums[i] = books[i] - i. This transformation is important because it helps in efficiently enforcing the strictly increasing book count condition.

The next step is to find the left boundary for each shelf, at or beyond which we must start taking books to ensure the increasing sequence rule. This is accomplished by using a monotonically increasing stack stk. Iterating through the nums array:

For every element v at index i, we pop from the stack while its top element is greater than or equal to v. This ensures that our stack stk is always in increasing order.

We then set left[i] to the top of the stack stk if the stack is not empty; this represents the index of the previous smaller or equal element in nums. If the stack is empty, it means there is no previous element in the nums array smaller or equal to nums[i], which implies that we can take books[i] books from shelf i.

Push the current index i onto the stack.

Next begins the dynamic programming phase, where we initialize a dp array to store the maximum number of books that can be taken ending at each shelf index. dp[0] starts as books[0] because that's the maximum we can take from the first shelf.

As we go through each shelf i, with a variable j representing left[i], which is the left boundary index — we calculate:

The count cnt of the maximum contiguous books that we can take from the current shelf, which is either books[i] or a truncated count if j is not too far from i.

A series, u, representing the count from which we would have to start taking books from the current shelf to maintain the increasing sequence, which is calculated as v - cnt + 1.

A sum s, which is the sum of the series from u to v calculated as (u + v) \* cnt // 2.

The dp[i] is then set to this sum s plus the dp of the previous boundary j if j is not -1.

As we compute dp[i] for each shelf, we keep track of the maximum sum ans found at any stage, which, once the iterations are complete, holds the answer to the problem — the maximum number of books that can be taken from the bookshelf.

**Example Walkthrough**

Imagine we have a books array [4, 5, 4, 3, 6, 2], which represents the number of books on each respective shelf of a bookshelf.

Following the solution approach:

**Step 1: Transform the Array**

Create a transformed nums array by subtracting the index from the number of books on each shelf.

Original books array: [4, 5, 4, 3, 6, 2]

Transformed nums array: [4 - 0, 5 - 1, 4 - 2, 3 - 3, 6 - 4, 2 - 5]

Resulting nums array: [4, 4, 2, 0, 2, -3]

**Step 2: Use a Stack to Define Left Boundaries**

Start iterating through nums and use a stack stk to keep track of indices of a monotonically increasing series.

Iteration through nums:

i = 0, nums[i] = 4: Stack is empty, so push i to the stack. stk = [0].

i = 1, nums[i] = 4: The top of the stack is equal to nums[i]; no stack operation needed. stk = [0, 1].

i = 2, nums[i] = 2: Pop 1 from the stack because nums[2] < nums[1]. The stack top is now less than nums[i], so push 2 to the stack. stk = [0, 2].

i = 3, nums[i] = 0: Continue popping until the stack is empty since all elements are larger or equal. Push 3 to the stack. stk = [3].

i = 4, nums[i] = 2: No pops needed, push 4 to the stack. stk = [3, 4].

i = 5, nums[i] = -3: Pop all elements since they are greater. stk ends empty.

Left bounds array after iteration: left = [-1, 0, 0, -1, 3, -1] (If the stack is empty the left bound is -1)

**Step 3: Dynamic Programming to Calculate Maximum Books**

Initialize a dp array to store the maximum books we can take, starting with dp[0] as books[0] which is 4. Now iterate to calculate dp[i].

Let's process each shelf:

i = 1: left[1] = 0, Calculate available books from current shelf books[1]. dp[1] = dp[0] + books[1] dp[1] = 4 + 5 = 9

i = 2: left[2] = 0, Calculate books as sequence starting from 1 (since 2 books taken from shelf 0). cnt = 1 (Because taking 2 books would violate increasing condition) dp[2] = dp[left[2]] + cnt dp[2] = 4 + 1 = 5

i = 3: left[3] = -1, It means take all available books books[3]. dp[3] = books[3] = 3

i = 4: left[4] = 3, Calculate books as sequence starting from 1. cnt = 1 (Only one book can be taken) dp[4] = dp[left[4]] + cnt dp[4] = 3 + 1 = 4

i = 5: left[5] = -1, Take available books books[5]. dp[5] = books[5] = 2

The dp array after processing: [4, 9, 5, 3, 4, 2]

The maximum number of books that can be taken is the largest number in dp, which is 9 (which is the sum of books from shelf 0 to shelf 1).

So, following the approach outlined in the solution, we were able to find that the maximum number of books we can take while maintaining the strictly increasing condition is 9.

**Solution Implementation**

import java.util.Arrays;

import java.util.ArrayDeque;

import java.util.Deque;

class Solution {

public long maximumBooks(int[] books) {

int n = books.length;

int[] adjustedBooks = new int[n];

// Adjust the book counts by subtracting the current index from the book count at each position

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

adjustedBooks[i] = books[i] - i;

}

// Initialize an array to track the nearest smaller value to the left

int[] leftSmallerIndex = new int[n];

Arrays.fill(leftSmallerIndex, -1);

// Monotonically decreasing stack to find the nearest smaller values

Deque<Integer> stack = new ArrayDeque<>();

// Iterate through each adjusted book count to populate leftSmallerIndex

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

// Pop all elements that are greater than or equal to the current element

while (!stack.isEmpty() && adjustedBooks[stack.peek()] >= adjustedBooks[i]) {

stack.pop();

}

// If stack isn't empty, it means we found a smaller element to the left

if (!stack.isEmpty()) {

leftSmallerIndex[i] = stack.peek();

}

// Push the current index onto the stack

stack.push(i);

}

long maxBooks = 0; // To store the maximum book count

long[] dp = new long[n]; // Dynamic programming array to store the maximum books at each position

dp[0] = books[0]; // Base case: At position 0, max books equal to the book count at the 0th position

// Iterate through each position to find the maximum number of books

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

int leftIndex = leftSmallerIndex[i];

int bookCount = books[i];

int count = Math.min(bookCount, i - leftIndex);

int startValue = bookCount - count + 1;

long sum = (long)(startValue + bookCount) \* count / 2; // Calculate the sum of the arithmetic series

dp[i] = sum + (leftIndex == -1 ? 0 : dp[leftIndex]); // Calculate dp value for the current position

maxBooks = Math.max(maxBooks, dp[i]); // Update maxBooks if current dp value is larger

}

// Return the maximum books that can be obtained

return maxBooks;

}

}

**Refer to**

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)