Previous Next

410. Split Array Largest Sum

Dec. 15, 2017 | 66.6K views

Average Rating: 3.93 (87 votes)

Given an array which consists of non-negative integers and an integer *m*, you can split the array into *m* non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these *m* subarrays.

Note:
If *n* is the length of array, assume the following constraints are satisfied:

- $1 \le n \le 1000$
- $1 \le m \le \min(50, n)$

Examples:

```
Input:
nums = [7,2,5,10,8]
m = 2

Output:
18

Explanation:
There are four ways to split nums into two subarrays.
The best way is to split it into [7,2,5] and [10,8],
where the largest sum among the two subarrays is only 18.
```

Approach #1 Brute Force [Time Limit Exceeded]

Intuition

Check all possible splitting plans to find the minimum largest value for subarrays.

Algorithm

We can use depth-first search to generate all possible splitting plan. For each element in the array, we can choose to append it to the previous subarray or start a new subarray starting with that element (if the number of subarrays does not exceed `m`). The sum of the current subarray can be updated at the same time.

C++ Java Copy

```cpp
class Solution {
public:
    int ans;
    int n, m;
    void dfs(vector<int>& nums, int i, int cntSubarrays, int curSum, int curMax) {
        if (i == n && cntSubarrays == m) {
            ans = min(ans, curMax);
            return;
        }
        if (i == n) {
            return;
        }
        if (i > 0) {
            dfs(nums, i + 1, cntSubarrays, curSum + nums[i], max(curMax, curSum + nums[i]));
        }
        if (cntSubarrays < m) {
            dfs(nums, i + 1, cntSubarrays + 1, nums[i], max(curMax, nums[i]));
        }
    }
    int splitArray(vector<int>& nums, int M) {
        ans = INT_MAX;
        n = nums.size();
        m = M;
        dfs(nums, 0, 0, 0, 0);
        return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n^m)$. To split `n` elements into `m` parts, we can have $\binom{n-1}{m-1}$ different solutions. This is equivalent to n^m.
- Space complexity : $O(n)$. We only need the space to store the array.

Approach #2 Dynamic Programming [Accepted]

Intuition

The problem satisfies the non-aftereffect property. We can try to use dynamic programming to solve it.

The non-aftereffect property means, once the state of a certain stage is determined, it is not affected by the state in the future. In this problem, if we get the largest subarray sum for splitting `nums[0..i]` into `j` parts, this value will not be affected by how we split the remaining part of `nums`.

To know more about non-aftereffect property, this link may be helpful :
http://www.programering.com/a/MDOzUzMwATM.html

Algorithm

Let's define `f[i][j]` to be the minimum largest subarray sum for splitting `nums[0..i]` into `j` parts.

Consider the `j`th subarray. We can split the array from a smaller index `k` to `i` to form it. Thus `f[i][j]` can be derived from `max(f[k][j - 1], nums[k + 1] + ... + nums[i])`. For all valid index `k`, `f[i][j]` should choose the minimum value of the above formula.

The final answer should be `f[n][m]`, where `n` is the size of the array.

For corner situations, all the invalid `f[i][j]` should be assigned with `INFINITY`, and `f[0][0]` should be initialized with `0`.

C++ Java Copy

```cpp

class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        int n = nums.size();
        vector<vector<int>> f(n + 1, vector<int>(m + 1, INT_MAX));
        vector<int> sub(n + 1, 0);
        for (int i = 0; i < n; i++) {
            sub[i + 1] = sub[i] + nums[i];
        }
        f[0][0] = 0;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                for (int k = 0; k < i; k++) {
                    f[i][j] = min(f[i][j], max(f[k][j - 1], sub[i] - sub[k]));
                }
            }
        }
        return f[n][m];
    }
};
```

Complexity Analysis

- Time complexity : $O(n^2 * m)$. The total number of states is $O(n * m)$. To compute each state `f[i][j]`, we need to go through the whole array to find the optimum `k`. This requires another $O(n)$ loop. So the total time complexity is $O(n^2 * m)$.
- Space complexity : $O(n * m)$. The space complexity is equivalent to the number of states, which is $O(n * m)$.

Approach #3 Binary Search + Greedy [Accepted]

Intuition

We can easily find a property for the answer:

> If we can find a splitting method that ensures the maximum largest subarray sum will not exceed a value `x`, then we can also find a splitting method that ensures the maximum largest subarray sum will not exceed any value `y` that is greater than `x`.

Lets define this property as `F(x)` for the value `x`. `F(x)` is true means we can find a splitting method that ensures the maximum largest subarray sum will not exceed `x`.

From the discussion above, we can find out that for `x` ranging from `-INFINITY` to `INFINITY`, `F(x)` will start with false, then from a specific value `x0`, `F(x)` will turn to true and stay true forever.

Obviously, the specific value `x0` is our answer.

Algorithm

We can use Binary search to find the value `x0`. Keeping a value `mid = (left + right) / 2`. If `F(mid)` is false, then we will search the range `[mid + 1, right]`; If `F(mid)` is true, then we will search `[left, mid - 1]`.

For a given `x`, we can get the answer of `F(x)` using a greedy approach. Using an accumulator `sum` to store the sum of the current processing subarray and a counter `cnt` to count the number of existing subarrays. We will process the elements in the array one by one. For each element `num`, if `sum + num <= x`, it means we can add `num` to the current subarray without exceeding the limit. Otherwise, we need to make a cut here, start a new subarray with the current element `num`. This leads to an increment in the number of subarrays.

After we have finished the whole process, we need to compare the value `cnt` to the size limit of subarrays `m`. If `cnt <= m`, it means we can find a splitting method that ensures the maximum largest subarray sum will not exceed `x`. Otherwise, `F(x)` should be false.

C++ Java Copy

```cpp
#define LL long long
class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        LL l = 0, r = 0;
        int n = nums.size();
        for (int i = 0; i < n; i++) {
            r += nums[i];
            if (l < nums[i]) {
                l = nums[i];
            }
        }
        LL ans =  r;
        while (l <= r) {
            LL mid = (l + r) >> 1;
            LL sum = 0;
            int cnt = 1;
            for (int i = 0; i < n; i++) {
                if (sum + nums[i] > mid) {
                    cnt ++;
                    sum = nums[i];
                } else {
                    sum += nums[i];
                }
            }
            if (cnt <= m) {
                ans = min(ans, mid);
```

Complexity Analysis

- Time complexity : $O(n * log(sum\ of\ array))$. The binary search costs $O(log(sum\ of\ array))$, where `sum of array` is the sum of elements in `nums`. For each computation of `F(x)`, the time complexity is $O(n)$ since we only need to go through the whole array.
- Space complexity : $O(n)$. Same as the Brute Force approach. We only need the space to store the array.

Rate this article:

Previous Next

Comments: 51 Sort By

Type comment here... (Markdown is supported)

Preview Post

HawaiianCalm ★ 196 August 20, 2019 4:37 AM

It took me a while to understand what binary search was actually binary searching for. Now that I understand, it's actually pretty simple:

- Set the search range between min=(largest single value) and max=(sum of all values).
  The min starts there because we're looking for the sum of the largest group in the final set of

Read More

165 Share Reply

SHOW 12 REPLIES

snandi1603 ★ 50 June 17, 2019 5:36 AM Report

Such a Pathetic explanation!!!!!

37 Share Reply

SHOW 2 REPLIES

toplanzi ★ 82 May 10, 2018 1:06 AM

For the binary search case I think the time complexity is actually: O(n*log(SumOfArray-MaxElementInArray)) since the search is made between sum of the array elements (right), and max element in the array (left)
Also space complexity should be O(1) since we don't need to create a new array.

18 Share Reply

SHOW 1 REPLY

wcarvalho ★ 37 December 29, 2017 10:38 PM

Question #2: I also don't understand why mid, a value that is never explicitly constrained to be a sum of the subarrays, will correspond to the sum of the subarrays. Thanks!

11 Share Reply

SHOW 2 REPLIES

mishuman ★ 79 February 20, 2019 6:37 PM

I wish all articles in leetcode was so nicely explained, comprehensible and precise.

23 Share Reply

SHOW 2 REPLIES

Evercode ★ 126 May 3, 2019 8:14 AM Report

Approach #2 is actually confusing. The author mix base condition with the transition. Took me some time to figure out. This is more clear

```
        for (int i=1;i<=n;i++)
```

Read More

9 Share Reply

LLLLLL ★ 124 December 16, 2018 8:38 PM

Python solution for approach 2 throws TLE.

6 Share Reply

SHOW 3 REPLIES

JustKeepCodinggg ★ 63 January 23, 2020 1:59 AM

I don't know why leetcode solution explanations are so poor. And let's not even mention the code readability! Leetcode needs to do a better job at filtering and editing these articles.
Here is the same question solved on geek for geeks with great explanation on how the Approach 3 works! - https://www.geeksforgeeks.org/split-the-given-array-into-k-sub-arrays-such-that-maximum-sum-of-all-sub-arrays-is-minimum/

Read More

3 Share Reply

SHOW 2 REPLIES

jackzhao-mj ★ 9 January 30, 2019 7:36 AM

Here's a Python version of the DP approach:

```
class Solution(object):
    def splitArray(self, nums, m):
```

Read More

3 Share Reply

SHOW 2 REPLIES

w238liu ★ 4 September 9, 2019 1:40 AM

Just want to improve the 2nd Approach from two aspects. First, since the minimax sum of k splits only depends on the case of k-1 splits, we may reduce the space complexity from O(m*n) to O(n). Second, the minimax sum of subarrays of nums[:j-1] must be less than or equal to that of nums[:j], so we can apply a binary search instead of the linear search when updating the dp array. The time complexity is thus reduced from O(n^2*m) to O((nlog n) * m). Below shows my python implementation.

Read More

2 Share Reply

‹ 1 2 3 4 5 6 ›