



Problem Description

The task is to find a subsequence from a given array nums such that the sum of this subsequence's elements is strictly greater than the sum of the elements that are not included in the subsequence. A subsequence is defined as a sequence that can be derived from the array by removing some or no elements without changing the order of the remaining elements.

If there are multiple subsequences that satisfy the condition, the one with the smallest number of elements should be returned.

The constraints for the solution are:

- If there is still more than one subsequence with the minimum number of elements, the subsequence among them with the
- maximum total sum should be returned. The returned subsequence should be sorted in non-increasing order (from the largest to the smallest element).
- It is guaranteed that a unique solution exists for the given constraints.
- Intuition

To arrive at the solution, we should understand the following points:

elements.

 Sorting the array in non-increasing order helps in picking out the subsequence with the maximum possible sum in the smallest size.

A subsequence with the largest possible elements will contribute more towards making its sum greater than the rest of the

- Accumulating the items from the sorted array until the running total sum (t) of these items is greater than the sum of the remaining items (s - t) gives a subsequence that fulfills the condition.
- Now, considering these points, the algorithm follows these steps: Sort the array in non-increasing order.

3. After each addition, check if the sum of elements in this subsequence result (t) is strictly greater than the sum of the remaining

- 4. Continue accumulating elements until the condition is met.
- elements in the original array (s t).
- Once the condition is met, the current subsequence result (ans) contains the minimum number of elements with the maximum sum, sorted in non-increasing order. This is the desired subsequence to be returned.

2. Iterate over the sorted array, adding each element to a running total (t) and storing the element in a subsequence result (ans).

Solution Approach

The solution approach takes advantage of the sorting algorithm and a greedy strategy to create the desired subsequence. Here's a walkthrough of the implementation based on the reference solution provided:

1. First, we need to know the sum of all elements in the given array nums. This is calculated with the expression sum(nums) and stored in the variable s.

2. We then define a variable t to keep track of the total sum of the selected subsequence as we build it. Initially, t is set to 0 since we haven't selected any elements yet.

- 3. The input array nums is then sorted in non-increasing order using sorted (nums, reverse=True). Sorting the array allows us to consider larger elements first, aligning with the greedy approach of maximizing the sum of the subsequence.
- 5. Next, we iterate over the sorted array. For each element x in the array: • We add the element x to the total t and also append x to our answer list ans.

After each addition, we check if the sum t is strictly greater than the sum of the remaining elements in the original array (s -

t). If this condition is true, it means the selected elements in ans now have a sum greater than that of the remaining elements of

4. We initialize an empty list ans to store the elements of the subsequence.

nums, fulfilling our requirement.

sorted in non-increasing order - as dictated by our sorting of nums.

- non-increasing order, the first time this condition is met, we have the smallest subsequence with the greatest total sum.
- 7. Lastly, the subsequence stored in ans is returned. This is the subsequence with the minimum size and maximum total sum,

6. The iteration is terminated immediately once our condition is met (i.e., when t > s - t). Since we are iterating over the array in

By utilizing a greedy algorithmic approach along with elementary data structures like lists and performing array sorting, we have an efficient implementation that ensures the strict constraints of the problem are met and a unique solution is returned.

Let's use a small example to illustrate the solution approach. Consider the array nums = [4, 3, 1, 2]. We need to find a subsequence such that its sum is greater than the sum of the elements not included in it.

3. Sort the array in non-increasing order: sorted_nums = [4, 3, 2, 1].

remaining sum (7 > 3).

Example Walkthrough

4. Initialize an empty list ans to store our subsequence. 5. Start iterating over the sorted array. We pick elements one by one and add to both t and ans.

remaining sum. \circ Second element x = 3: t = 4 + 3 = 7, and t = [4, 3], and remaining sum t = 3. Now, the sum of and is greater than the

def minSubsequence(self, nums: List[int]) -> List[int]:

Initialize the variable to track the running total of the subsequence

Check if the sum of the subsequence is greater than the remaining elements

Append the current number to the result subsequence

if subsequence_sum > total_sum - subsequence_sum:

If true, break out of the loop

6. The condition t > s - t is met, so we stop iterating.

1. Calculate the sum of all elements in nums: s = 4 + 3 + 1 + 2 = 10.

2. Initialize t to 0, which will represent the total sum of our subsequence.

7. The resulting subsequence ans = [4, 3] is already sorted in non-increasing order, which is also the subsequence with the smallest number of elements and the maximum total sum out of all valid subsequences.

We return ans = [4, 3] as our final answer. This subsequence has a sum of 7, which is greater than the sum of the non-included

elements (2 + 1 = 3). It's the smallest such subsequence with the largest sum, fulfilling all the problem's requirements.

 \circ First element x = 4: t = 0 + 4 = 4, ans = [4], and remaining sum s - t = 6. The sum of ans is not yet greater than the

Initialize the variable to store the resulting subsequence result_subsequence = [] # Calculate the total sum of the elements in nums

```
subsequence_sum = 0
13
           # Sort the nums array in decreasing order
14
           for num in sorted(nums, reverse=True):
15
               # Add the current number to the running total
16
```

10

11

12

17

18

19

20

21

22

23

24

25

26

27

Python Solution

class Solution:

from typing import List

total_sum = sum(nums)

break

break;

return result;

subsequence_sum += num

Return the result subsequence

result_subsequence.append(num)

```
28
           return result_subsequence
29
Java Solution
   import java.util.Arrays; // Import Arrays utility for sorting and streaming
   import java.util.List; // Import List interface
   import java.util.ArrayList; // Import ArrayList class for dynamic arrays
   class Solution {
       public List<Integer> minSubsequence(int[] nums) {
           // Sort the array in non-decreasing order
           Arrays.sort(nums);
 9
10
           // Prepare a list to store the result
           List<Integer> result = new ArrayList<>();
11
12
13
           // Calculate the sum of all elements in the array
           int totalSum = Arrays.stream(nums).sum();
14
15
16
           // Initialize a variable to keep track of the running sum of the subsequence
17
           int subsequenceSum = 0;
18
19
           // Iterate over the array from the last element to the first
20
           for (int i = nums.length - 1; i >= 0; i--) {
               // Add the current element to the subsequence sum
21
22
               subsequenceSum += nums[i];
               // Add the current element to the result list
23
24
               result.add(nums[i]);
               // Check if the subsequence sum is greater than the remaining elements' sum
25
               if (subsequenceSum > totalSum - subsequenceSum) {
26
```

// Break the loop if the subsequence sum is greater as we have the minimum subsequence

// Return the result list which now contains the minimum subsequence

C++ Solution 1 #include <vector>

27

28

29

30

31

32

33

34

35

```
2 #include <numeric> // For accumulate
  #include <algorithm> // For sort
   class Solution {
   public:
       // Function to find the smallest subsequence with a sum greater than the sum of the remaining elements.
       vector<int> minSubsequence(vector<int>& nums) {
           // Sort the input vector in non-increasing order.
 9
           sort(nums.rbegin(), nums.rend());
10
11
           // Calculate the total sum of all elements in the vector.
12
           int totalSum = accumulate(nums.begin(), nums.end(), 0);
13
           // This will keep track of the sum of the current subsequence.
14
15
           int subsequenceSum = 0;
16
           // This vector will store the elements of the smallest subsequence.
17
           vector<int> answer;
18
19
           // Iterate over the sorted array to accumulate a sum greater than the half of total sum.
           for (int num : nums) { // Using num instead of x for clarity.
20
21
                subsequenceSum += num; // Add the current element to the subsequence sum.
22
               answer.push_back(num); // Add the current element to the subsequence.
23
               // Check if the current subsequence sum is greater than the sum of the remaining elements.
24
               if (subsequenceSum > totalSum - subsequenceSum) {
25
                    break; // Terminate the loop if the condition is true.
26
27
28
29
           // Return the resulting subsequence.
30
           return answer;
31
32 };
33
Typescript Solution
   function minSubsequence(nums: number[]): number[] {
       // Sort the array in non-increasing order (largest to smallest)
       nums.sort((a, b) \Rightarrow b - a);
       // Calculate the sum of all elements in the array
       const totalSum = nums.reduce((sum, current) => sum + current, 0);
```

27 // An empty subsequence signifies failure to find the desired subset. 28 return []; 29 } 30

Time and Space Complexity

let subsequenceSum = 0;

// Iterate through the sorted array

subsequenceSum += nums[i];

for (let i = 0; i < nums.length; i++) {

// up to the current element

return nums.slice(0, i + 1);

// Add the current element to the subsequence sum

// the sum of the remaining elements in the array

if (subsequenceSum > totalSum - subsequenceSum) {

// In case no subsequence is found that satisfies the condition

// Though this case won't occur as per the problem's constraints,

// we need to return something to satisfy TypeScript's return type.

// Check if the sum of the elements of the subsequence is greater than

// If the condition is met, return the subsequence from the start

8

9

10

11

12

14

15

16

17

18

19

20

21

22

23

24

25

26

Time Complexity The time complexity of the given code is primarily determined by the sorting operation. Here, sorted(nums, reverse=True) sorts the

input list nums in descending order. The sorting algorithm used in Python's sorted function is Timsort, which has a time complexity of O(n log n) for the average and worst case, where n is the number of elements in the input list. After sorting, the code iterates through the sorted list once. This single pass through the list has a time complexity of 0(n).

Therefore, the overall time complexity of the code is dominated by the sorting operation and is $0(n \log n)$.

Space Complexity The space complexity of the code involves the space needed for the output list ans and the temporary space used by the sorting operation.

already been accounted for as O(n).

The output list ans grows proportionally with the number of elements it includes, which in the worst case, can be all the elements of the input list nums. Hence, the space complexity for ans is O(n).

The sorting operation may require additional space for its internal workings (temp arrays for mergesort or hybrid sorts like Timsort). In Python, the sorted function does not sort in-place; instead, it returns a new sorted list. However, since this sorted list is not stored in an additional variable and is used in the same loop for iteration, the dominant additional space is the same as for ans, which has

Therefore, the overall space complexity of the code is O(n).