<https://leetcode.com/problems/boats-to-save-people/description/>

You are given an array people where people[i] is the weight of the ith person, and an infinite number of boats where each boat can carry a maximum weight of limit. Each boat carries at most two people at the same time, provided the sum of the weight of those people is at most limit.

Return the minimum number of boats to carry every given person.

**Example 1:**

**Input:** people = [1,2], limit = 3

**Output:** 1

**Explanation:** 1 boat (1, 2)

**Example 2:**

**Input:** people = [3,2,2,1], limit = 3

**Output:** 3

**Explanation:** 3 boats (1, 2), (2) and (3)

**Example 3:**

**Input:** people = [3,5,3,4], limit = 5

**Output:** 4

**Explanation:** 4 boats (3), (3), (4), (5)

**Constraints:**

1 <= people.length <= 5 \* 10^4

1 <= people[i] <= limit <= 3 \* 10^4

**Attempt 1: 2025-03-16**

**Solution 1: Greedy + Two Pointers (10 min)**

class Solution {

    public int numRescueBoats(int[] people, int limit) {

        Arrays.sort(people);

        int boats = 0;

        int lo = 0;

        int hi = people.length - 1;

        while(lo <= hi) {

            if(people[lo] + people[hi] <= limit) {

                lo++;

            }

            hi--;

            boats++;

        }

        return boats;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/boats-to-save-people/editorial/>

**Approach 1: Greedy (Two Pointer)**

**Intuition**

**If the heaviest person can share a boat with the lightest person, then do so. Otherwise, the heaviest person can't pair with anyone,**

so they get their own boat.

The reason this works is because if the lightest person can pair with anyone, they might as well pair with the heaviest person.

**Algorithm**

Let people[i] to the currently lightest person, and people[j] to the heaviest.

Then, as described above, if the heaviest person can share a boat with the lightest person (if people[j] + people[i] <= limit)

then do so; otherwise, the heaviest person sits in their own boat.

class Solution {

public int numRescueBoats(int[] people, int limit) {

Arrays.sort(people);

int i = 0, j = people.length - 1;

int ans = 0;

while (i <= j) {

ans++;

if (people[i] + people[j] <= limit)

i++;

j--;

}

return ans;

}

}

**Refer to Deepseek**

Okay, I need to solve this LeetCode problem 881, Boats to Save People. Let me read the problem carefully.

The problem says that we have an array called people where people[i] is the weight of the ith person. And we have boats that can carry a maximum weight of 'limit'. Each boat can carry at most two people at the same time, provided the sum of their weights is at most the limit. We need to find the minimum number of boats to carry everyone.

Hmm. So the key points here are that each boat can carry one or two people, and the sum of the weights of the people in the boat can't exceed the limit. And we need the minimum number of boats.

So first, how do I approach this? Let me think. Maybe a greedy approach would work here. Like, pair the heaviest person with the lightest possible person. If they can fit together, then we do that. Otherwise, the heaviest person has to go alone. Then move to the next heaviest.

Yes, that makes sense. Because if the heaviest person can pair with someone, pairing them with the lightest gives the best chance for that other person to also pair with someone else. Otherwise, if you pair the heaviest with a heavier person, you might end up using more boats.

So the steps would be:

1. Sort the array of people's weights.

2. Use two pointers, one at the start (lightest) and one at the end (heaviest).

3. Check if the sum of the two is <= limit. If yes, then both can go in one boat. Move both pointers.

4. If not, the heaviest goes alone. Move the end pointer.

5. Increment the boat count each time.

This way, we efficiently pair people to minimize the number of boats.

To solve this problem, we need to determine the minimum number of boats required to rescue all people given that each boat can carry at most two people and their combined weight must not exceed a specified limit.

**Approach**

The optimal approach involves using a greedy algorithm with two pointers. The key steps are as follows:

Sort the Array: First, sort the array of people's weights. This allows us to efficiently pair the lightest and heaviest people using two pointers.

Two Pointers Technique: Use two pointers, one starting at the beginning (lightest person) and the other at the end (heaviest person) of the sorted array.

Pairing Check: For each iteration, check if the sum of the weights at the two pointers is within the limit. If it is, both people can be carried in one boat, and both pointers move inward. If not, only the heaviest person is carried in a boat, and the end pointer moves inward.

Count Boats: Increment the boat count each time a pair is checked, whether one or two people are carried.

This approach ensures that we use the minimum number of boats by efficiently pairing people.

**Solution Code**

import java.util.Arrays;

public class Solution {

public int numRescueBoats(int[] people, int limit) {

Arrays.sort(people);

int boats = 0;

int left = 0;

int right = people.length - 1;

while (left <= right) {

if (people[left] + people[right] <= limit) {

left++;

}

right--;

boats++;

}

return boats;

}

}

**Explanation**

Sorting: Sorting the array allows us to use the two-pointer technique effectively. By considering the lightest and heaviest remaining person each time, we can make optimal decisions on pairing.

Two Pointers: The left pointer starts at the beginning (lightest person) and the right pointer at the end (heaviest person). We check if they can be paired. If yes, both pointers move inward. If not, only the right pointer moves inward.

Boat Count: Each iteration of the loop processes at least one person (the heaviest if not paired, or both if paired), incrementing the boat count each time.

This method efficiently pairs people to minimize the number of boats used, ensuring an optimal solution with a time complexity of O(n log n) due to sorting, followed by a linear scan O(n).