题目地址（881. 救生艇）

https://leetcode-cn.com/problems/boats-to-save-people/

入选理由

1. 和前面的区间不同，这是另外一种贪心的类型。给大家看看不同的题型 难度也是中等

标签

- 贪心

难度

- 中等

题目描述

第 i 个人的体重为 people[i]，每艘船可以承载的最大重量为 limit。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 limit。

返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

示例 1：

输入：people = [1,2], limit = 3 输出：1 解释：1 艘船载 (1, 2) 示例 2：

输入：people = [3,2,2,1], limit = 3 输出：3 解释：3 艘船分别载 (1, 2), (2) 和 (3) 示例 3：

输入：people = [3,5,3,4], limit = 5 输出：4 解释：4 艘船分别载 (3), (3), (4), (5) 提示：

1 <= people.length <= 50000 1 <= people[i] <= limit <= 30000

暴力 DFS（超时）

思路

定义函数 `dfs(people, limit, i, remain)`，其功能是计算从 i 到最后 的 people 在 limit 的限制下，当前一个船还有 remain 位置的情况下需要的最小的船数。那么答案自然就是 `1 + dfs(people, limit, 0, limit)` 。

dfs 中我们最多有两个选择：

- 装 people[i]
- 不装 people[i]

之所以说**最多**有两个选择，是因为存在装不下的情况。

为了防止重复选择，我们需要记录被装运的人，可以使用 visited 的集合来维护，并在 dfs 退出时候撤销 visited，因此这其实就是前面搜索篇讲的暴力回溯。

代码

代码支持： Python3

```
class Solution:
    def numRescueBoats(self, people: List[int], limit: int) -> int:
        visited = set()
        def dfs(people, limit, i, remain):
            if (i >= len(people)):
                return 0
            ans = len(people)
            for j in range(len(people)):
                if j in visited: continue
                visited.add(j)
                # 这个时候需要增加一个新船
                if (people[j] > remain):
                    ans = min(ans, 1 + dfs(people, limit, i + 1, limit - people[j]))
                # 无需增加新船
                else:
                    ans = min(ans, dfs(people, limit, i + 1, remain - people[j]))
                visited.remove(j)
            return ans
        return 1 + dfs(people, limit, 0, limit)
```

复杂度分析

令 n 为 people 的长度。

- 时间复杂度：我们要做的是 dfs 中枚举这两种情况，dfs 的深度（也就是递归树的深度）是 people 的长度，因此总的时间复杂度就是指数级别，即 $O(2^n)$
- 空间复杂度：空间取决于 visited 和 栈的开销，二者最大的情况都不超过 n，因此空间复杂度为 $O(n)$

排序 + 双指针

思路

上面的思路可行，但是太过复杂。而且前面的思路有点像前面我们讲过的 01 背包问题。有没有办法对前面的方法进行优化呢？答案是有的。

由于题目要求船数最少，那显然我们希望每个船都尽可能多装一些重量，这样才能使得结果更优。

每次装人的时候先将最大重量的装上，然后再遍历剩下的人，看剩余容量能否再装下一人，实际上由于题目限定了只能坐两个人，那么我们优先选择较轻的总是没错的，因此轻的更容易被满足并且不会比重的结果差（制定贪心策略）。 最后将已经被装上船的人踢出列表，继续按上述策略装，直到所有人都上船。

每次遍历剩下的人以及将人踢出列表的时间复杂度过高，我们可以采用排序 + 双指针的具体策略来完成。因此这道题大的层面上是**贪心**，具体战术上采用的是排序 + 双指针，这同时也是贪心问题的一个常见的做法。

具体地：

- 先对 people 进行一次排序（不妨进行一次升序）。
- 选择头尾两个人。如果可以同时载就运载这两个人。如果不可行，那么这个重的人和剩下任何人都无法配对，只能自己走了。

采用上面的策略直到全部运走即可。

代码

代码支持：JS，Python，CPP,Java

JS Code：

```
var numRescueBoats = function (people, limit) {
  people.sort((a, b) => a - b);
  let ans = 0,
    start = 0,
    end = people.length - 1;
  while (start <= end) {
    if (people[end] + people[start] <= limit) {
      start++;
      end--;
    } else {
      end--;
    }
    ans++;
  }
  return ans;
};
```

Python3 Code:

```python
class Solution:
    def numRescueBoats(self, people: List[int], limit: int) -> int:
        res = 0
        l = 0
        r = len(people) - 1
        people.sort()

        while l < r:
            total = people[l] + people[r]
            if total > limit:
                r -= 1
                res += 1
            else:
                r -= 1
                l += 1
                res += 1
        if (l == r):
            return res + 1
        return res
```

CPP Code：

```cpp
class Solution {
public:
    int numRescueBoats(vector<int>& people, int limit) {


        sort(people.begin(), people.end());

        int left =0;
        int right = people.size()-1;
        int count =0;
        while(left <=right)
        {
            if(left!=right && people[right]+people[left]<=limit)
            {
                right--;
                left++;
                count++;
            }
            else
            {
                right--;
                count++;
            }
```

```
        }

        return count;
    }
};
```

Java Code:

```
class Solution {
    public int numRescueBoats(int[] people, int limit) {
        Arrays.sort(people);
        int left = 0;
        int right = people.length - 1;
        int res = 0;

        while (left <= right) {
            res++;
            if (people[left] + people[right] <= limit) {
                left++;
            }
            right--;
        }

        return res;
    }
}
```

复杂度分析

- 时间复杂度：$O(nlogn)$

- 空间复杂度：$O(1)$

上一页 下一页