第三题 Cross the river

1. 问题描述

题目要求用最少的船把所有的人都运过河，最少的船是关键，换句话说就是尽量做到不浪费船的承载能力，每一步都浪费最少那么整体就是最优解。那么马上就能想到如果将所有乘客按照体重排好序，那么队首和队尾的一组可以组成最省承载能力的一对乘客（如果不超载的话），如果超载的话换句话说就是任何人和体重最大的人一起都超载，那么就让体重最大的单独做一个船，这样也是最优情况。那么我们就可以用每一步都贪心的思想解决这个问题了。

greedy-choice property是避免浪费船的承载量；optimal substructure是选当前最重和最轻的人坐船，超出的话就让最重的人自己坐船。

1. 伪代码实现

|  |
| --- |
| MinBoats(m, n, passenger)# m represents the limit of one boat; n represents the number of passengers; passenger is a list of weight  1：boat = 0;  2：Sort passenger;  3：while passenger is not null do  4： if passenger[0] + passenger[n-1] > m then  5： passenger.pop(-1);  6： else then  7： passenger.pop(0);  8： passenger.pop(-1);  9： n--;  10： end if  11： n--;  12： boat++;  13： if n == 1 then  14： boat++  15： n--;  16： end if  17：end while  18：return boat; |

1. 正确性分析

当只有一个人要过河的时候，需要一个船；当有两个人要过河的时候，如果两个人体重加和未超限制则需要一个船，否则需要两个船；当有n个人要过河的时候，将当前最重的人和最轻的人进行组合，如果未超出限制则这两人过河是当前子问题的最优解（浪费最少），如果超出限制则最重的人单独乘船过河也是当前子问题的最优解（无法乘坐两人）。由每一步的最优解可以归纳出整体问题的最优解。

1. 复杂度分析

排序使用归并排序则当前时间复杂度为O(nlogn)，greedy部分共比较了小于n次，所以时间复杂度小于O(n)。综上所述该算法整体时间复杂度为O(nlogn)。

第五题 Toy Buildings

1. 问题描述

给定一组数组，要求最后操作结束后整个数组是非逆序的，即对于整个数组后一个元素大于等于前一个元素。一步操作指的是找到数组的子数组并将整个子数组加一。由于要求最终的数组是一个非逆序的，不妨从左边开始进行操作。通过分析题目我们可以发现要求找的是操作的最少次数而不是总共的加和。深入分析我们得知当前元素i是否需要进行加一操作完全取决于它前面时候有比它大的元素，这种规则换个角度可以从两者的差值来描述。我们还可以得出结论，对某一子数组进行操作和对这一子数组一直到最后全部进行操作，对于题目结果影响是相同的（因为后面元素是否需要加一是基于差值判断的，对后面全部加一并不改变差值），举个例子会更清晰。

①假设我们有数组[5, 1, 5, 1]，那么第一位肯定是不变的，我们看第二位，此时可以只对第二位进行加四操作，也可以对第二位以及后面全部进行加四操作，这两种操作对于题解是等价最优的，因为后者也并没有改变后面数组的差值情况。

②在另一种情况下我们有数组[5, 1, 1, 5, 1]，此时我们可以只对第二位进行加四，但我们发现如果对第二第三位同时操作将更节省，再结合①，我们知道对第二三位操作和对第二三位以及到最后所有位同时操作是等价的。

③于是我们得到了简洁明了的贪心操作：从左至右，如果第i位小于第i-1位，则将第i到最后一位循环加一直到第i位等于第i-1位。

这样做虽然可行，但是每一步计算operation的时候都要对原数组进行操作，导致在题目网站（找到了一个在线题目网站）上每次都超时。参考了别人的ac思路，其实我的想法就差一点，关键在于我们不需要每次都对数组进行操作了，只要能判断出最终需要的最少操作数即可。具体就是找完整的上升子串作为待操作的子串，从左至右由于当前子串是否需要增加一只与它与前一个元素的差值有关，所以，对于当前上升子串的操作数目就是上一个上升子串的最后一个元素减去当前上升子串首个元素。

贪心策略是从左至右找的上升子串要是最长的（如有[1,2,3,2,1]，则第一个上升子串应该选择[1,2,3]而不是它的子集）。最优子结构是当前的最长上升子串。

1. 伪代码实现

|  |
| --- |
| OldAlgorithm  NonDecrease(buildings, n)#buildings is a list of height, n represents the length of list  1：op = 0;  2：for i = 1 to n-1 do  3： while buildings[i] < buildings[i-1] do  4： Launch add operation for buildings[i,-1];  5： op++;  6： end while  7：end for  8：return op;  NewAlgorithm  NonDecrease(buildings, n)#buildings is a list of height, n represents the length of list  1：op = 0;  2：for every IncreseSubsegment in buildings do  3： op += previous IncreseSubsegment[-1] - current IncreseSubsegment[0];  4：end for  5：return op; |

1. 正确性分析

需要分两部分证明。首先证明初始思路的正确性，由观察分析我们容易得到在问题描述中提到的结论①。在结论①的支持下对于整个数组从左至右找上升子串（此上升子串要是最长的子串，以此满足贪心规则），将该子串到最后进行加一操作直到子串首位与前一个元素值相等，可以得知这步操作是最优的。由每一步的最优操作可以最终得到全局的最优操作。其次对于后面实际能AC的伪代码，其完全依照第一部分的思想，即找到最长的上升子串，然后只需计算操作的次数而不对数组进行实际操作，而实际操作与否并不影响后续子串的结构信息（是否与前一项有差值），所以此算法也是能达到最优解的。

1. 复杂度分析

由于整体只对数组扫描一遍，且计算操作次数的时候只是用前一个上升子串末位和当前上升子串首位进行相减是一步操作，且判断上升子串是否结束时也仅需顺序向后扫描即可，因此最终的时间复杂度是O(n)。

第六题 Maximum Number of Coins You Can Get

1. 问题描述

这是一个取硬币的问题，整体一共有3n堆硬币，每一轮有我指定拿出三堆硬币，并且我能获得这三堆中数量排在中间的堆。问用什么策略能使得最后获得的硬币最多。那么对于取硬币的三个人进行分析，对于Alice会取走当前最大的一堆，不管我们如何设计最终她拿到的总是比我拿得到大，并且无论如何都能拿到当前数组中最大的那一堆（无论到第几轮拿她都能拿到），那么我们不妨每一轮都直接给她最大的一堆，我们拿第二大的那堆；对于Bob他会拿三个里最小的堆，此时我们是可以进行操作的，即每次都选取当前数组里最小的给Bob，这样就能保证他每次拿完对整体利益损失最小（整体利益损失最小就是保持剩余硬币总数最大，即是我们能拿到的硬币越多）。

于是贪心思想就是对整个数组排序后，我们每一次都拿刚好排在Alice后面的那一堆，这样每一步我们都能尽可能拿到最大的（选连续的两堆），又因为Alice不论哪一步都始终将拿到当前数组最大的堆，我们不妨每次都选取最大的给她这样可以更直观。对于剩下一堆我们就拿最小的给Bob这样也是他拿完对整体利益影响最小。

总而言之，对于排序后的数组每次挑选当前最小的一个，和最大的以及次大的作为三个候选数组，重复操作直到数组为空即可。

1. 伪代码实现

|  |
| --- |
| MaxCoins(piles, n)  1：head = 0;  2：tail = 3 \* n - 1;  3：score = 0;  4：Sort(piles);  5：while(head + 1 != tail) then  6： head++;  7： tail -= 2;  8： score += piles[tail+1];  9：end while  10：return score; |

1. 正确性分析

当n为1时即共有三堆硬币时，取最小给Bob，再取连续最大的二元子序列分给Alice和自己，此时算法能得到最优解。假设当完成了n次选择之后算法取得了最优解，则在第n+1次选择时，选择最小的给Bob，选择最大的两个分别给Alice和自己，得到的是当前情况的最优解，所以n+1次选择时算法成立。综上所述，算法是正确的。

1. 复杂度分析

采用归并排序时间复杂度为O(nlogn)，贪心算法中从前向后从后向前的指针只扫了整个数组一次，所以贪心时间复杂度为O(n)。综上该算法的时间复杂度是O(nlogn)。