**45. 跳跃游戏 II**

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4]

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

|  |
| --- |
| class Solution {  public:  int jump(vector<int>& nums) {  if (nums.size() < 2) {  return 0;  }  int curt\_max\_reach = nums[0];  int next\_max\_reach = nums[0];  int leap = 1;  for (int i = 1; i < nums.size(); ++i) {  if (i > curt\_max\_reach) {  ++leap;  curt\_max\_reach = next\_max\_reach;  }  next\_max\_reach = max(next\_max\_reach, nums[i] + i);  }  return leap;  }  };  可以使用搜索完成。但标准做法是贪心，从当前位置跳到下一个位置时，选择下一个能跳到最远位置的地方。 |

**55. 跳跃游戏**

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

输入: [2,3,1,1,4]

输出: true

解释: 我们可以先跳 1 步，从位置 0 到达 位置 1, 然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: [3,2,1,0,4]

输出: false

解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。

|  |
| --- |
| class Solution {  public:  bool canJump(vector<int>& nums) {  if (nums.size() < 2) {  return true;  }  int maxReach = nums[0];  for (int i = 0; i < nums.size(); ++i) {  if (i > maxReach) {  return false;  }  maxReach = max(maxReach, nums[i] + i);  }  return true;  }  }; |

**134. 加油站**

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

如果题目有解，该答案即为唯一答案。

输入数组均为非空数组，且长度相同。

输入数组中的元素均为非负数。

示例 1:

输入:

gas = [1,2,3,4,5]

cost = [3,4,5,1,2]

输出: 3

解释:

从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油

开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油

开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油

开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油

开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油

开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。

因此，3 可为起始索引。

示例 2:

输入:

gas = [2,3,4]

cost = [3,4,3]

输出: -1

解释:

你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。

我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油

开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油

开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油

你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。

因此，无论怎样，你都不可能绕环路行驶一周。

如果A->B无法完成到达，那么中间也没有答案（之前累积的结果都是正的，最后一个位置让它不再为正数；如果把起始指针向后移动，情况只会更糟）；如果气体总和大于消耗量，那么必然有解。

|  |
| --- |
| int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {  int rest = 0, run = 0, start = 0;  for (int i = 0; i < gas.size(); ++i){  run += (gas[i] - cost[i]);  rest += (gas[i] - cost[i]);  if (run < 0){  start = i + 1;  run = 0;  }  }  return rest < 0 ? -1: start;  } |

车能开完全程需要满足两个条件：

1. 车从i站能开到i+1。

2. 所有站里的油总量要>=车子的总耗油量。

那么，假设从编号为0站开始，一直到k站都正常，在开往k+1站时车子没油了。这时，应该将起点设置为k+1站。

问题1: 为什么应该将起始站点设为k+1？

因为k->k+1站耗油太大，0->k站剩余油量都是不为负的，每减少一站，就少了一些剩余油量。所以如果从k前面的站点作为起始站，剩余油量不可能冲过k+1站。

问题2: 为什么如果k+1->end全部可以正常通行，且rest>=0就可以说明车子从k+1站点出发可以开完全程？

因为，起始点将当前路径分为A、B两部分。其中，必然有(1)A部分剩余油量<0。(2)B部分剩余油量>0。

所以，无论多少个站，都可以抽象为两个站点（A、B）。(1)从B站加满油出发，(2)开往A站，车加油，(3)再开回B站的过程。

重点：B剩余的油>=A缺少的总油。必然可以推出，B剩余的油>=A站点的每个子站点缺少的油。

有一个环形路上有n个站点； 每个站点都有一个好人或一个坏人； 好人会给你钱，坏人会收你一定的过路费，如果你带的钱不够付过路费，坏人会跳起来把你砍死； 问：从哪个站点出发，能绕一圈活着回到出发点?

首先考虑一种情况：如果全部好人给你 的钱加起来 小于 坏人收的过路费之和，那么总有一次你的钱不够付过路费，你的结局注定会被砍死。

假如你随机选一点 start 出发，那么你肯定会选一个有好人的站点开始，因为开始的时候你没有钱，遇到坏人只能被砍死；

现在你在start出发，走到了某个站点end，被end站点的坏人砍死了，说明你在 [start, end) 存的钱不够付 end点坏人的过路费，因为start站点是个好人，所以在 (start, end) 里任何一点出发，你存的钱会比现在还少，还是会被end站点的坏人砍死；

于是你重新读档，聪明的选择从 end+1点出发，继续你悲壮的征程； 终于有一天，你发现自己走到了尽头（下标是n-1)的站点而没有被砍死； 此时你犹豫了一下，那我继续往前走，身上的钱够不够你继续走到出发点Start?

当然可以，因为开始已经判断过，好人给你的钱数是大于等于坏人要的过路费的，你现在攒的钱完全可以应付 [0, start) 这一段坏人向你收的过路费。 这时候你的嘴角微微上扬，眼眶微微湿润，因为你已经知道这个世界的终极奥秘：Start就是这个问题的答案。

**378. 有序矩阵中第K小的元素**

给定一个 n x n 矩阵，其中每行和每列元素均按升序排序，找到矩阵中第 k 小的元素。

请注意，它是排序后的第 k 小元素，而不是第 k 个不同的元素。

示例：

matrix = [

[ 1, 5, 9],

[10, 11, 13],

[12, 13, 15]

],

k = 8,

返回 13。

|  |
| --- |
| class Solution {  private:  int m, n;  public:  int kthSmallest(vector<vector<int>>& matrix, int k) {  m = matrix.size();  n = matrix[0].size();  int l = matrix[0][0], h = matrix[m - 1][n - 1];  while(l < h){  int mid = (l + h) / 2;  int rank = get\_rank(matrix, mid);  if(rank >= k)  h = mid;  else  l = mid + 1;  }  return l;  }  private:  int get\_rank(const vector<vector<int>>& matrix, int target){  int res = 0;  for(int i = 0; i < m && matrix[i][0] <= target; i ++)  for(int j = 0; j < n && matrix[i][j] <= target; j ++)  res ++;  return res;  }  }; |

|  |
| --- |
| class Solution {  public:  int kthSmallest(vector<vector<int>>& matrix, int k) {  priority\_queue<int> q;  int cnt = 0;  int n = matrix.size();  for(auto& row: matrix)  for(auto it: row){  if(cnt < k){  cnt++;  q.push(it);  }else{  if(q.top() > it) {  q.pop();  q.push(it);  }  }  }  return q.top();  }  }; |

**870. 优势洗牌**

给定两个大小相等的数组 A 和 B，A 相对于 B 的优势可以用满足 A[i] > B[i] 的索引 i 的数目来描述。

返回 A 的任意排列，使其相对于 B 的优势最大化。

示例 1：

输入：A = [2,7,11,15], B = [1,10,4,11]

输出：[2,11,7,15]

示例 2：

输入：A = [12,24,8,32], B = [13,25,32,11]

输出：[24,32,8,12]

|  |
| --- |
| class Solution {  public:  vector<int> advantageCount(vector<int>& A, vector<int>& B) {  multiset<int> s;  for (int a : A) s.insert(a);  vector<int> res (A.size(), 0);  for (int i = 0; i < A.size(); ++i) {  auto it = s.upper\_bound(B[i]);  if (it == s.end()) it = s.begin();  res[i] = \*it;  s.erase(it);  }  return res;  }  }; |

**630. 课程表 III**

这里有 n 门不同的在线课程，他们按从 1 到 n 编号。每一门课程有一定的持续上课时间（课程时间）t 以及关闭时间第 d 天。一门课要持续学习 t 天直到第 d 天时要完成，你将会从第 1 天开始。

给出 n 个在线课程用 (t, d) 对表示。你的任务是找出最多可以修几门课。

示例：

输入: [[100, 200], [200, 1300], [1000, 1250], [2000, 3200]]

输出: 3

解释:

这里一共有 4 门课程, 但是你最多可以修 3 门:

首先, 修第一门课时, 它要耗费 100 天，你会在第 100 天完成, 在第 101 天准备下门课。

第二, 修第三门课时, 它会耗费 1000 天，所以你将在第 1100 天的时候完成它, 以及在第 1101 天开始准备下门课程。

第三, 修第二门课时, 它会耗时 200 天，所以你将会在第 1300 天时完成它。

第四门课现在不能修，因为你将会在第 3300 天完成它，这已经超出了关闭日期。

|  |
| --- |
| class Solution {  public int scheduleCourse(int[][] courses) {  //根据课程结束时间升序排列  Arrays.sort(courses,(a,b) -> (a[1]-b[1]));  //课程用时的大根优先级队列  Queue<Integer> queue = new PriorityQueue<>((a,b) -> (b-a));  int times = 0;  for (int i=0;i<courses.length;i++){  //如果此课程可以学习，则学习，总用时增加，此课程用时入堆  if (times+courses[i][0]<=courses[i][1]){  times += courses[i][0];  queue.add(courses[i][0]);  //如果不能学习此课程，因为此课程结束时间比之前所有的都晚，存在两种情况：  //1.此课程用时比之前某个课程少：则学习此课程，放弃之前用时最长的课程  //2.此课程用时比之前所有课程多：则不学习此课程，可以理解为学习此课程，同时放弃之前用时最长的课程（此课程）  //则此种情况，学习此课程并放弃之前用时最长的课程（总用时减去大根堆堆顶）  }else{  queue.add(courses[i][0]);  times = times + courses[i][0] - queue.poll();  }  }  return queue.size();  }  } |

按照结束时间排序，如果当前课程无法在ddl前完成了，就去掉一个时间最长的课程。贪心的标准做法是按结束时间排序后尽可能选，选不了的就放弃。但是这道题的区别在于它的开始时间并不是确定的。

**948. 令牌放置**

你的初始能量为 P，初始分数为 0，只有一包令牌。

令牌的值为 token[i]，每个令牌最多只能使用一次，可能的两种使用方法如下：

如果你至少有 token[i] 点能量，可以将令牌置为正面朝上，失去 token[i] 点能量，并得到 1 分。

如果我们至少有 1 分，可以将令牌置为反面朝上，获得 token[i] 点能量，并失去 1 分。

在使用任意数量的令牌后，返回我们可以得到的最大分数。

示例 1：

输入：tokens = [100], P = 50

输出：0

示例 2：

输入：tokens = [100,200], P = 150

输出：1

示例 3：

输入：tokens = [100,200,300,400], P = 200

输出：2

|  |
| --- |
| class Solution {  public int bagOfTokensScore(int[] tokens, int power) {  if (null == tokens || 0 == tokens.length) {  return 0;  }  int score = 0;  Arrays.sort(tokens);  int i = 0, j = tokens.length - 1;  while (i <= j) {  while (i <= j && power >= tokens[i]) {  power -= tokens[i++];  score++;  }  if (score == 0 || i >= j) {  break;  }  power += tokens[j--];  score--;  }  return score;  }  }  排序。从左边消耗，能量不够了从右边取，直到无法补充能量，或首尾指针相遇。 |

**955. 删列造序 II**

给定由 N 个小写字母字符串组成的数组 A，其中每个字符串长度相等。

选取一个删除索引序列，对于 A 中的每个字符串，删除对应每个索引处的字符。

比如，有 A = ["abcdef", "uvwxyz"]，删除索引序列 {0, 2, 3}，删除后 A 为["bef", "vyz"]。

假设，我们选择了一组删除索引 D，那么在执行删除操作之后，最终得到的数组的元素是按 字典序（A[0] <= A[1] <= A[2] ... <= A[A.length - 1]）排列的，然后请你返回 D.length 的最小可能值。

示例 1：

输入：["ca","bb","ac"]

输出：1

解释：

删除第一列后，A = ["a", "b", "c"]。

现在 A 中元素是按字典排列的 (即，A[0] <= A[1] <= A[2])。

我们至少需要进行 1 次删除，因为最初 A 不是按字典序排列的，所以答案是 1。

示例 2：

输入：["xc","yb","za"]

输出：0

解释：

A 的列已经是按字典序排列了，所以我们不需要删除任何东西。

注意 A 的行不需要按字典序排列。

也就是说，A[0][0] <= A[0][1] <= ... 不一定成立。

示例 3：

输入：["zyx","wvu","tsr"]

输出：3

解释：

我们必须删掉每一列。

从左到右扫描，如果有一个数字不满足字典序，那么删掉该列，否则保留。

|  |
| --- |
| class Solution {  public:  int res;  int n,w;  int minDeletionSize(vector<string>& A) {  n = A.size();w = A[0].size();  vector<string> tmp(n);  vector<string> cur(n);  for(int i=0;i<w;i++) {  for(int j=0;j<n;j++) cur[j] += A[j][i];  if(judge(cur)) {  res++;  cur = tmp;  }else  tmp = cur;  }  return res;  }  bool judge(vector<string>& cur) {  for(int i=0;i<cur.size()-1;i++) if(cur[i]>cur[i+1]) return true;  return false;  }  }; |

**789. 逃脱阻碍者**

你在进行一个简化版的吃豆人游戏。你从 (0, 0) 点开始出发，你的目的地是 (target[0], target[1]) 。地图上有一些阻碍者，第 i 个阻碍者从 (ghosts[i][0], ghosts[i][1]) 出发。

每一回合，你和阻碍者们\*可以\*同时向东，西，南，北四个方向移动，每次可以移动到距离原位置1个单位的新位置。如果你可以在任何阻碍者抓住你之前到达目的地（阻碍者可以采取任意行动方式），则被视为逃脱成功。如果你和阻碍者同时到达了一个位置（包括目的地）都不算是逃脱成功。

当且仅当你有可能成功逃脱时，输出 True。

示例 1:

输入：

ghosts = [[1, 0], [0, 3]]

target = [0, 1]

输出：true

解释：

你可以直接一步到达目的地(0,1)，在(1, 0)或者(0, 3)位置的阻碍者都不可能抓住你。

示例 2:

输入：

ghosts = [[1, 0]]

target = [2, 0]

输出：false

解释：

你需要走到位于(2, 0)的目的地，但是在(1, 0)的阻碍者位于你和目的地之间。

示例 3:

输入：

ghosts = [[2, 0]]

target = [1, 0]

输出：false

解释：

阻碍者可以和你同时达到目的地。

|  |
| --- |
| class Solution {  public:  bool escapeGhosts(vector<vector<int>>& ghosts, vector<int>& target) {  int min\_dis = INT\_MAX;  int dis = abs(target[0]) + abs(target[1]);  for (auto& ghost : ghosts) {  min\_dis = min(min\_dis, GetDis(ghost, target));  }  return dis < min\_dis;  }  private:  inline int GetDis(vector<int>& dot, vector<int>& target) {  return abs(dot[0] - target[0]) + abs(dot[1] - target[1]);  }  };  算鬼到终点的最近距离，如果自己的距离小于这个距离就可以，否则不行 |

**1147. 段式回文**

段式回文 其实与 一般回文 类似，只不过是最小的单位是 一段字符 而不是 单个字母。

举个例子，对于一般回文 "abcba" 是回文，而 "volvo" 不是，但如果我们把 "volvo" 分为 "vo"、"l"、"vo" 三段，则可以认为 “(vo)(l)(vo)” 是段式回文（分为 3 段）。

给你一个字符串 text，在确保它满足段式回文的前提下，请你返回 段 的 最大数量 k。

如果段的最大数量为 k，那么存在满足以下条件的 a\_1, a\_2, ..., a\_k：

每个 a\_i 都是一个非空字符串；

将这些字符串首位相连的结果 a\_1 + a\_2 + ... + a\_k 和原始字符串 text 相同；

对于所有1 <= i <= k，都有 a\_i = a\_{k+1 - i}。

示例 1：

输入：text = "ghiabcdefhelloadamhelloabcdefghi"

输出：7

解释：我们可以把字符串拆分成 "(ghi)(abcdef)(hello)(adam)(hello)(abcdef)(ghi)"。

示例 2：

输入：text = "merchant"

输出：1

解释：我们可以把字符串拆分成 "(merchant)"。

示例 3：

输入：text = "antaprezatepzapreanta"

输出：11

解释：我们可以把字符串拆分成 "(a)(nt)(a)(pre)(za)(tpe)(za)(pre)(a)(nt)(a)"。

示例 4：

输入：text = "aaa"

输出：3

解释：我们可以把字符串拆分成 "(a)(a)(a)"。

|  |
| --- |
| class Solution {  public:  int longestDecomposition(string text) {  int n=text.size();  for(int i=n-1;i>0;i--) //从后往前找一个与字符串首字母相同的字母  {  if(text[i]==text[0]&&text.substr(0,n-i)==text.substr(i)) //字符串能匹配才行，不行就继续往前  {  if(2\*i>n)  return 2+longestDecomposition(text.substr(n-i,2\*i-n));  else if(2\*i==n) //此时整个字符串正好是左半边==右半边  return 2;  else  return 1;  }  }  return 1;  }  }; |