* **最大重叠区间个数**

1. 问题描述

有一个聚会，许多人来参加。一个记录仪记录下了每个人到达聚会的时间和离开聚会的时间。也就是说每个人在聚会的时间为[]，求问这个聚会至少能同时接纳多少人？（即这个聚会中最多能同时接纳多少人？）

1. 问题分析

这是最大重叠区间的一个具体样例，这种题的做法就是将所有的开始时间和结束时间（无论哪个事件的或人的达到结束时间）进行排序，如果是开始时间则计数加1，如果是结束时间则计数减1。在此过程中的计数的最大值便是最大重叠区间数目。

（通俗理解：只要一直是开始时间，那就说明遇到的区间一直在重叠，没有结束。当遇到一个结束时间时，就说明目前重叠的区间的数目到达一个极大值，不能再加入了，否则开始时间还会在结束时间之前出现的。）

PS：在排序的时候，会有开始时间和结束时间重叠的情况，总共分为三种情况：

1. 两个开始时间重叠（即相等），这种情况不影响结果，因为无论哪个在前计数都要加1
2. 两个结束时间重叠（即相等），这种情况不影响结果，因为无论哪个在前计数都要减1
3. 开始时间和结束时间重叠（即相等），这种情况出现时，应该让结束时间排靠在前，开始时间排靠在后，否则若将开始时间排靠在前的话，会将两个区间连接但不重叠的情况考虑成两个区间连接但重叠

总的时间复杂度是排序O(nlogn) + 遍历O(n)，n是时间的个数。

1. 代码实现
2. #include <vector>
3. #include <iostream>
4. #include <algorithm>
6. **using** **namespace** std;
8. **struct** node
9. {
10. **int** time;   //time为到达时间或者结束时间
11. **int** flag;  // flag为0则为到达时间，flag为1则为结束时间
12. node(**int** t, **int** f) : time(t), flag(f) {}
13. };
14. **int** main()
15. {
16. **int** n;   // 此聚会来过的总人数
17. cin >> n;
18. vector<node> v;
19. **while** (n--) {
20. **int** s, e;   // 输入每个人的到达时间s和结束时间e
21. cin >> s >> e;
22. v.push\_back(node(s, 0));
23. v.push\_back(node(e, 1));
24. }
26. //按时间排序（所有事件的开始时间和结束时间）
27. //注意排序时的特殊情况：当时间相等时，让结束时间在前
28. sort(v.begin(), v.end(), [](**const** node& a, **const** node& b) {
29. **if** (a.time == b.time) **return** a.flag > b.flag;
30. **return** a.time < b.time;
31. });
33. //遇到开始时间加1，遇到结束时间减1，找个过程中的最大值
34. **int** ans = 0, res = -1;
36. **for** (**int** i = 0; i < v.size(); i++) {
37. **if** (v[i].flag == 0)
38. ans++;
39. **else**
40. ans--;
41. res = max(res, ans);
42. }
44. cout << res << endl;
45. **return** 0;
46. }

* **区间最大重叠**

1. 问题描述

在一维坐标轴上有n个区间段，求重合区间最长的两个区间段。

1. 问题分析

首先按照区间的左端点即start对n个区间段进行排序；然后从前往后遍历所有区间，比较前后两个区间的右端点即end；

假设前后区间分别为[x1,y1]，[x2,y2],因为是顺序遍历，因此x2>=x1，考虑一下情况：

1. 如果y2>=y1，

则在[x2,y2]后面的区间和[x1,y1]的重叠部分不会超过这个区间，因为他们的x>x2，而重叠的区间大小为(y1-x+1)或者为0；因此与区间[x1,y1]重叠的大小最大为y1-x2+1或者0。(当x2>y1时，两个区间不相交，即为0)

1. 如果y2<y1，

那么只能说区间[x2,y2]包含在[x1,y1]里面，这样的话，跟区间[x1,y1]重叠的大小至少为y2-x2+1，而区间2的大小即为y2-x2+1，此时重叠区间大小便是区间2的大小。但是此时存在其他区间与[x1,y1]的重叠面积更大的情况。所以在没找到一个区间的结束时间大于y1前（和1）情况一致），得一直查找比较。直到找到这个区间，那么现在的最大重叠面积便是当前走过的所有区间的最大重叠面积。

总的时间复杂度为：排序O(nlogn)+遍历O(n)

1. 代码实现
2. #include <iostream>
3. #include <vector>
4. #include <algorithm>
6. **using** **namespace** std;
8. **struct** node {   //区间数据结构，start是开始时间，end是结束时间
9. **int** start;
10. **int** end;
11. };
13. **int** longestOverlap(vector<node> &v) {  // 计算区间最大重叠面积
14. **int** n = v.size();
15. **int** maxOverlap = 0;
17. sort(v.begin(), v.end(), [](**const** node& a, **const** node& b) {  //按开始时间进行从小到大排序
18. **return** a.start < b.start;
19. });
21. node pre = v[0];
22. node cur;
23. **for** (**int** i = 1; i < n; i++) {
24. cur = v[i];
25. **if** (cur.end >= pre.end) {  //如果区间2的end大于或等于区间1的end，那么区间1和其他所有区间的最大重叠即等于它和区间2的重叠
26. maxOverlap = max(maxOverlap, max(pre.end - cur.start + 1, 0));
27. pre = cur;
28. }
29. **else** { //如果区间2的end小于区间1的end，此时的重叠就是区间2的长度。但是要找到和区间1最大重叠的区间，还得找到一个区间的end大于或等于区间1的end
30. maxOverlap = max(maxOverlap, cur.end - cur.start + 1);
31. }
32. }
34. **return** maxOverlap;
35. }
37. **int** main()
38. {
39. **int** n;
40. cin >> n;
41. vector<node> v(n);   //得到输入
42. **for** (**int** i = 0; i<n; i++) {
43. cin >> v[i].start >> v[i].end;
44. }
45. cout << longestOverlap(v) << endl;  //返回结果

48. **return** 0;
49. }

* **合并区间**

1. 问题描述

给出一个区间的集合，请合并所有重叠的区间。（LeetCode 56.）

1. 问题分析

通过以上两个问题的分析后，将所有区间按照开始时间从小到大排序。不同于区间最大重叠面积，由于其计算最大重叠面积，极值终止条件是只要找到一个区间y2>=y1便可，计算重叠要么是0（没有重叠）要么是真实重叠长度。**此题条件稍有不同**，只要x2<=y1那便必有重叠，此时合并即可；否则不合并。

具体操作是，先排序。将第一个节点入栈，目前的区间和栈顶计算重叠，若重叠，则出栈将新合并的区间入栈，之后如此重复；否则不重叠时，将目前区间继续入栈，继续之后的循环。（排序后，目前节点和栈顶不重叠，那么后面其他所有区间肯定与之不重叠）

总的时间复杂度是排序O(nlogn) + 遍历O(n)，n是区间的个数；空间复杂度是O(m)，m是非重叠的区间个数。

1. 代码实现
2. **class** Solution {
3. **public**:
4. //首先按开始时间进行排序，然后用栈进行操作
5. //如果x2<=y1则重叠进行合并；否则x2>y1则不进行合并
6. vector<vector<**int**>> merge(vector<vector<**int**>>& intervals) {
7. vector<vector<**int**>> res;
8. **int** n = intervals.size();
9. **if** (!n) **return**{};
11. sort(intervals.begin(), intervals.end(), [](**const** vector<**int**>& a, **const** vector<**int**>& b) {
12. **return** a[0] < b[0];
13. });
15. vector<**int**> pre = intervals[0];
16. res.push\_back(pre);
17. vector<**int**> cur;
18. **for** (**int** i = 1; i < n; i++) {
19. cur = intervals[i];
20. pre = res.back();
21. **if** (cur[0] <= pre[1]) {
22. res.pop\_back();
23. res.push\_back({ pre[0], max(pre[1], cur[1]) });
24. pre = cur;
25. }
26. **else**
27. res.push\_back(cur);
28. }
30. **return** res;
31. }
32. };

* **无重叠区间**

1. 题目描述

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。（Leetcode 435.）

1. 题目分析

此题和合并区间题目基本类似。就是将所有区间按照开始时间从小到大排序。只要x2<=y1那便必有一个区间被删除。和合并区间有稍微区别的部分是：

1. 如果 x2 < y1 && y2 <= y1

也就是区间2完全被区间1包含，此时不应该删除区间2，而应该删除区间1

1. 如果x2 < y1 && y2 > y1

此时删除区间2即可

此题和合并区间的还有一个不同是，没有使用栈。合并区间是需要生成新的区间的，而且需要结果，所以必须使用栈。而无重叠区间不需要生成新的区间，只需要删除区间，所以只需要维护一个pre指针即可。

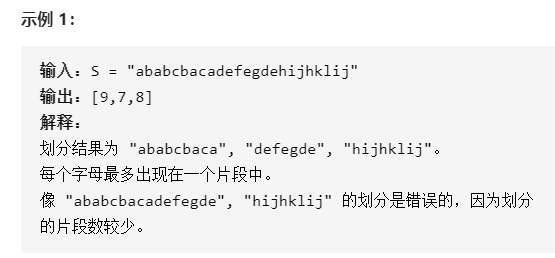
总的时间复杂度是排序O(nlogn) + 遍历O(n)，n是区间的个数；空间复杂度是O(m)，m是非重叠的区间个数。

1. 代码实现
2. **class** Solution {
3. **public**:
4. **int** eraseOverlapIntervals(vector<vector<**int**>>& intervals) {
5. **int** n = intervals.size();
6. **if**(n <= 0) **return** 0;
8. //按开始时间进行排序。如果开始时间相同，则按结束时间在前排序
9. sort(intervals.begin(), intervals.end(), [](**const** vector<**int**>& a,**const** vector<**int**>& b){
10. **if**(a[0] == b[0]) **return** a[1] <= b[1];
11. **return** a[0] < b[0];
12. });
14. **int** res = 0;
16. //维护一个pre指针
17. vector<**int**> cur, pre = intervals[0];
18. **for**(**int** i = 1; i < n; i++){
19. cur = intervals[i];
20. //如果x2 < y1 && y2 >= y1 ，删除区间2
21. **if**(cur[0] < pre[1] && cur[1] >= pre[1]){
22. res++;
23. }
24. //如果 x2 < y1 && y2 < y1，删除区间1
25. **else** **if**(cur[0] < pre[1] && cur[1] < pre[1]){
26. res++;
27. pre =  cur;
28. }
29. //没有重叠时，不需要删除，pre指针下跳
30. **else**{
31. pre = cur;
32. }
33. }
35. **return** res;
36. }
37. };

* **划分字母区间**

1. 题目描述

字符串 S 由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一个字母只会出现在其中的一个片段。返回一个表示每个字符串片段的长度的列表。（LeetCode 763.）



1. 题目分析

此题最初的思路是遍历S找到每个字母出现的开始位置和最后的位置，即相当于得到了多个区间，直接合并区间之后计算每个非重叠区间的长度即可。

优化先计算区间再合并区间的两步思路为一步：找到每个字母的最后出现位置后，从头开始遍历，若当前字母的位置和当前字母的结束位置相同，那么作为单独区间；若当前字母和当前字母结束位置为一个长度大于1的区间，那么继续遍历区间内直到达到结束位置，在此期间，若有字母末尾位置大于此区间，则扩展区间，直到达到此末尾。。。

时间复杂度为O(n)，空间复杂度为O(n)，因为可能所有字母单独成

1. 代码实现
2. **class** Solution {
3. **public**:
4. //这是个合并区间的问题
5. //1. 找出所有不同的字母，然后找出每个字母出现的区间，合并区间
6. //2. 第二种方法是题解给的方法，比较有效简单，和合并区间相比，是直接拓展区间
7. vector<**int**> partitionLabels(string S) {
8. **int** n = S.size();
9. **if**(!n) **return** {};
10. vector<**int**> res;
11. //存储每个字母出现的最后位置
12. vector<**int**> last(26, 0);
13. **for**(**int** i = 0; i < n; i++){
14. last[S[i] - 'a'] = i;
15. }
17. **int** start = 0, end = 0;  //表示一个区间的开始和结尾
18. **for**(**int** i = 0; i < n; i++){
19. end = max(end, last[S[i] - 'a']);
20. **if**(i == end){
21. res.push\_back(i - start + 1);
22. start = i+1;
23. }
24. }
26. **return** res;
27. }
28. };

* **用最小数量的箭引爆气球**

1. 题目描述

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足  xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。（LeetCode 452.）

1. 题目分析

此题怎么说呢？其实做完了以上题目后，这题已经没有任何难度。但它的类型却也不饿能简单的归纳给以上的题目类型。硬性归纳的话，和非重叠区间的个数类似。但是也不完全相同，此题其实本质是：同时重叠的区间的个数。

所以最终的结果其实等于：

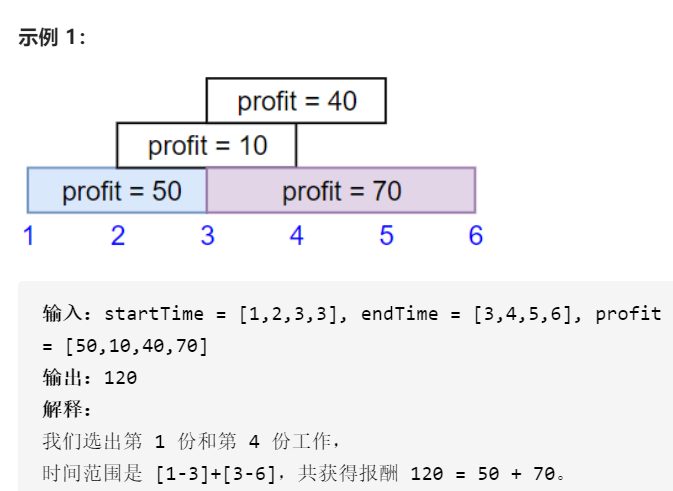
1. 当当前区间的x若大于前面区间的y时，则结果加1
2. 前面有重叠区间，但需要记录他们重叠前的最小的y，因为结果加1时仅当他们同时重叠，若当前区间虽然与他们之中一个有重叠，但不同时重叠时，仍然需要更多一支箭！所以在每个新的重叠区间里要维护一个最小的y
3. 代码实现
4. **class** Solution {
5. **public**:
6. **int** findMinArrowShots(vector<vector<**int**>>& points) {
7. **int** n = points.size();
8. **if**(n <= 1) **return** n;
10. //按开始时间进行排序
11. sort(points.begin(), points.end(), [](**const** vector<**int**>& a, **const** vector<**int**>& b){
12. **if**(a[0] == b[0]) **return** a[1] < b[1];
13. **return** a[0] < b[0];
14. });
16. //合并区间
17. **int** res = 1;
18. **int** minend = INT\_MAX;
19. vector<**int**> pre = points[0], cur;
20. **for**(**int** i = 1; i < n; i++){
21. cur = points[i];
22. **if**(cur[0] > pre[1] || cur[0] > minend){
23. res++;
24. pre = cur;
25. minend = INT\_MAX;
26. }**else**{
27. minend = min(minend, min(pre[1], cur[1]));
28. }
29. }
31. **return** res;
32. }
33. };

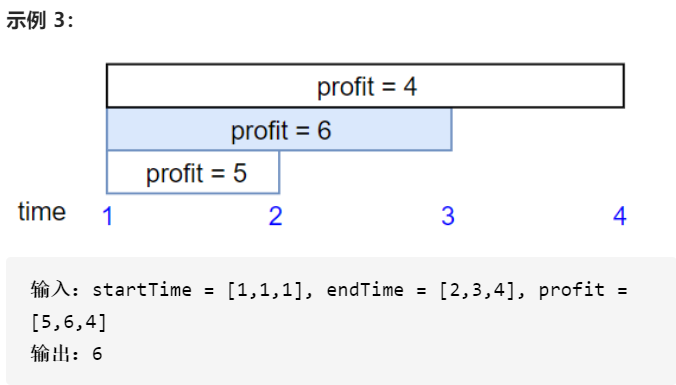
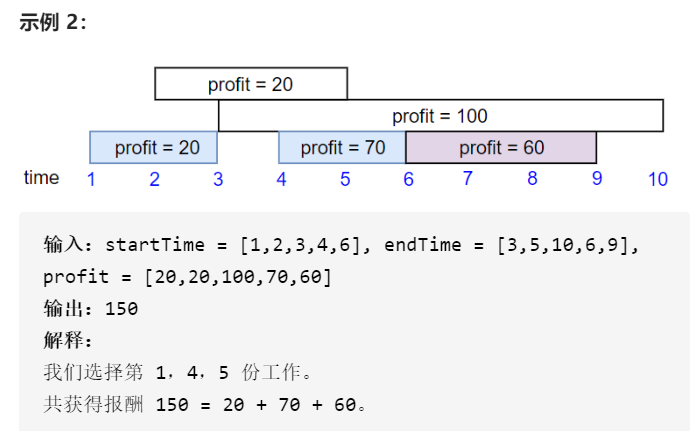
* **规划兼职工作**

1. 题目描述

你打算利用空闲时间来做兼职工作赚些零花钱。

这里有 n 份兼职工作，每份工作预计从 startTime[i] 开始到 endTime[i] 结束，报酬为 profit[i]。给你一份兼职工作表，包含开始时间 startTime，结束时间 endTime 和预计报酬 profit 三个数组，请你计算并返回可以获得的最大报酬。注意，时间上出现重叠的 2 份工作不能同时进行。如果你选择的工作在时间 X 结束，那么你可以立刻进行在时间 X 开始的下一份工作。





1. 题目分析

此题题目具有现实意义。同时和之前的区间题目相比，之间的区间题目不需要详细关注和区分区间之间的组合问题，具体来说是非重叠区间的组合问题。之前的题目是用相同的方式对所有的区间进行操作，而此题需要对区间进行组合，更加难度。

所以没用贪心来解决，而是使用动态规划来解决。

首先对所有区间按结束时间排序。用dp[i]表示到达i工作时所能达到的最大收益，pre[i]表示i工作之前最近能干的工作。所以状态方程就是：

dp[i] = max(profit[i] + dp[pre[i]], dp[i-1])

1. 代码实现
2. **class** Solution {
3. **public**:
4. **int** jobScheduling(vector<**int**>& startTime, vector<**int**>& endTime, vector<**int**>& profit) {
5. **int** n = startTime.size();
6. **if**(!n) **return** 0;
7. //将开始、结束时间和薪酬组织到一起
8. vector<vector<**int**>> job(n, vector<**int**>(3, 0));
9. **for**(**int** i = 0; i < n; i++){
10. job[i][0] = startTime[i];
11. job[i][1] = endTime[i];
12. job[i][2] = profit[i];
13. }
14. //按结束时间排序
15. sort(job.begin(), job.end(), [](**const** vector<**int**>& a, **const** vector<**int**>& b){
16. **return** a[1] < b[1];
17. });
19. //计算pre数组，pre[i]表示i工作前最近能淦的工作
20. vector<**int**> pre(n, -1);
21. **for**(**int** i = 1; i < n; i++){
22. **for**(**int** j = i-1; j >= 0; j--){
23. **if**(job[j][1] <= job[i][0]){
24. pre[i] = j;
25. **break**;
26. }
27. }
28. }
30. //计算dp[i]表示到i工作所能达到的最大收入
31. vector<**int**> dp(n, 0);
32. dp[0] = job[0][2];
33. **for**(**int** i = 1; i < n; i++){
34. **if**(pre[i] >= 0)
35. dp[i] = max(job[i][2] + dp[pre[i]], dp[i-1]);
36. **else**
37. dp[i] = max(job[i][2], dp[i-1]);
38. }
40. **return** dp[n-1];
42. }
43. };