

运用贪心算法来做时间管理

Original labuladong labuladong 2020-03-17 16:20

点击上方蓝字设为星标🌟

东哥带你手把手撕力扣~

作者：labuladong

公众号：labuladong

若已授权白名单也必须保留以上来源信息

什么是贪心算法呢？贪心算法可以认为是动态规划算法的一个特例，相比动态规划，使用贪心算法需要满足更多的条件（贪心选择性质），但是效率比动态规划要高。

比如说一个算法问题使用暴力解法需要指数级时间，如果能使用动态规划消除重叠子问题，就可以降到多项式级别的时间，如果满足贪心选择性质，那么可以进一步降低时间复杂度，达到线性级别的。

什么是贪心选择性质呢，简单说就是：每一步都做出一个局部最优的选择，最终的结果就是全局最优。注意哦，这是一种特殊性质，其实只有一小部分问题拥有这个性质。

比如你面前放着 100 张人民币，你只能拿十张，怎么才能拿最多的面额？显然每次选择剩下钞票中面值最大的一张，最后你的选择一定是最优的。

然而，大部分问题都明显不具有贪心选择性质。比如斗地主，对手出对儿三，按照贪心策略，你应该出尽可能小的牌刚好压制住对方，但现实情况我们甚至可能会出王炸。这种情况就不能用贪心算法，而得使用动态规划解决，参见前文 [动态规划解决博弈问题](#)。

一、问题概述

言归正传，本文解决一个很经典的贪心算法问题 Interval Scheduling（区间调度问题）。给你很多形如 `[start,end]` 的闭区间，请你设计一个算法，算出这些区间中最多有几个互不相交的区间。

```
int intervalScheduling(int[][] ints) {}
```

举个例子，`intvs=[[1,3],[2,4],[3,6]]`，这些区间最多有两个区间互不相交，即 `[[1,3],[3,6]]`，你的算法应该返回 2。注意边界相同并不算相交。

这个问题在生活中的应用广泛，比如你今天有好几个活动，每个活动都可以用区间 `[start,end]` 表示开始和结束的时间，请问你今天最多能参加几个活动呢？

如果你学过算法，就可以比别人更高效地规划时间，不是吗？

二、贪心解法

这个问题有许多看起来不错的解决思路，实际上都不能得到正确答案。比如说：

也许我们可以每次选择可选区间中开始最早的那个？但是可能存在某些区间开始很早，但是很长，使得我们错误地错过了一些短的区间。

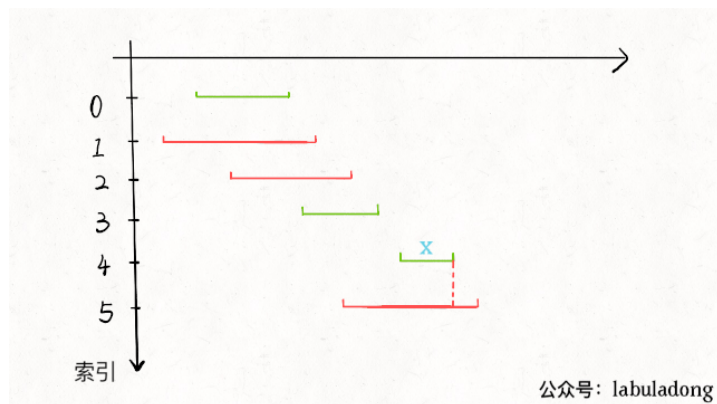
或者我们每次选择可选区间中最短的那个？或者选择出现冲突最少的那个区间？这些方案都能很容易举出反例，不是正确的方案。

正确的思路其实很简单，可以分为以下三步：

1. 从区间集合 `intvs` 中选择一个区间 `x`，这个 `x` 是在当前所有区间中结束最早的（`end` 最小）。
2. 把所有与 `x` 区间相交的区间从区间集合 `intvs` 中删除。
3. 重复步骤 1 和 2，直到 `intvs` 为空为止。之前选出的那些 `x` 就是最大不相交子集。

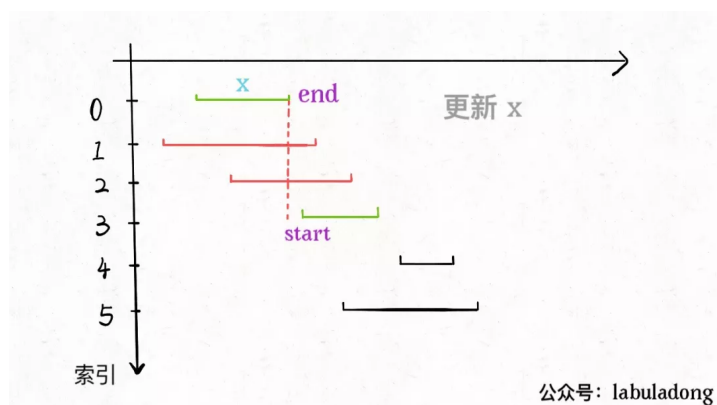
把这个思路实现成算法的话，可以按每个区间的 `end` 数值升序排序，因为这样处理

之后实现步骤 1 和步骤 2 都方便很多：



现在来实现算法，对于步骤 1，由于我们预先按照 **end** 排了序，所以选择 **x** 是很容易的。关键在于，如何去除与 **x** 相交的区间，选择下一轮循环的 **x** 呢？

由于我们事先排了序，不难发现所有与 **x** 相交的区间必然会与 **x** 的 **end** 相交；如果一个区间不想与 **x** 的 **end** 相交，它的 **start** 必须要大于（或等于）**x** 的 **end**：



下面看下代码：

```
public int intervalSchedule(int[][] intvs) {
    if (intvs.length == 0) return 0;
    // 按 end 升序排序
    Arrays.sort(intvs, new Comparator<int[]>() {
        public int compare(int[] a, int[] b) {
            return a[1] - b[1];
        }
    });
    // 至少有一个区间不相交
    int count = 1;
```

```

// 排序后，第一个区间就是 x
int x_end = intvs[0][1];
for (int[] interval : intvs) {
    int start = interval[0];
    if (start >= x_end) {
        // 找到下一个选择的区间了
        count++;
        x_end = interval[1];
    }
}
return count;
}

```

三、应用举例

下面举例几道 LeetCode 题目应用一下区间调度算法。

第 435 题，无重叠区间：

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。

注意：

1. 可以认为区间的终点总是大于它的起点。
2. 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。

我们已经会求最多有几个区间不会重叠了，那么剩下的不就是至少需要去除的区间吗？

```

int eraseOverlapIntervals(int[][] intervals) {
    int n = intervals.length;
    return n - intervalSchedule(intervals);
}

```

第 452 题，用最少的箭头射爆气球：

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 10^4 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 x_{start} , x_{end} ，且满足 $x_{start} \leq x \leq x_{end}$ ，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。

Example:

输入:

`[[10,16], [2,8], [1,6], [7,12]]`

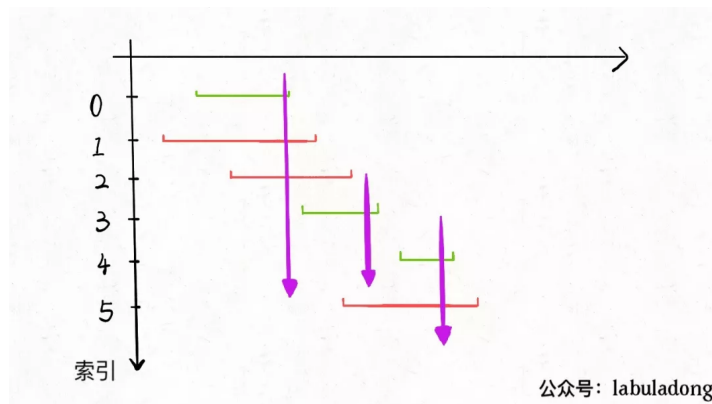
输出:

2

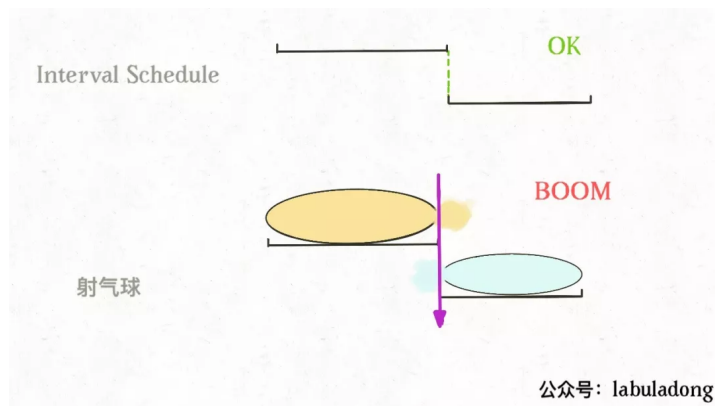
解释:

对于该样例，我们可以在 $x = 6$ （射爆 `[2,8]`, `[1,6]` 两个气球）和 $x = 11$ （射爆另外两个气球）。

其实稍微思考一下，这个问题和区间调度算法一模一样！如果最多有 n 个不重叠的区间，那么就至少需要 n 个箭头穿透所有区间：



只是有一点不一样，在 `intervalSchedule` 算法中，如果两个区间的边界触碰，不算重叠；而按照这道题目的描述，箭头如果碰到气球的边界气球也会爆炸，所以说相当于区间的边界触碰也算重叠：




所以只要将之前的算法稍作修改，就是这道题目的答案：

```
int findMinArrowShots(int[][] intvs) {  
    // ...  
  
    for (int[] interval : intvs) {  
        int start = interval[0];  
        // 把 >= 改成 > 就行了  
        if (start > x_end) {  
            count++;  
            x_end = interval[1];  
        }  
    }  
    return count;  
}
```

这么做的原因也不难理解，因为现在边界接触也算重叠，所以 `start == x_end` 时不能更新区间 `x`。

本文终。对于区间问题的处理，一般来说第一步都是排序，相当于预处理降低后续操作难度。但是对于不同的问题，排序的方式可能不同，这个需要归纳总结，以后再写写这方面的文章。

如果本文对你有帮助，点个在看分个享吧。

往期推荐 

[经典动态规划：0-1 背包问题](#)

[数据结构和算法学习指南](#)

[动态规划解题框架](#)