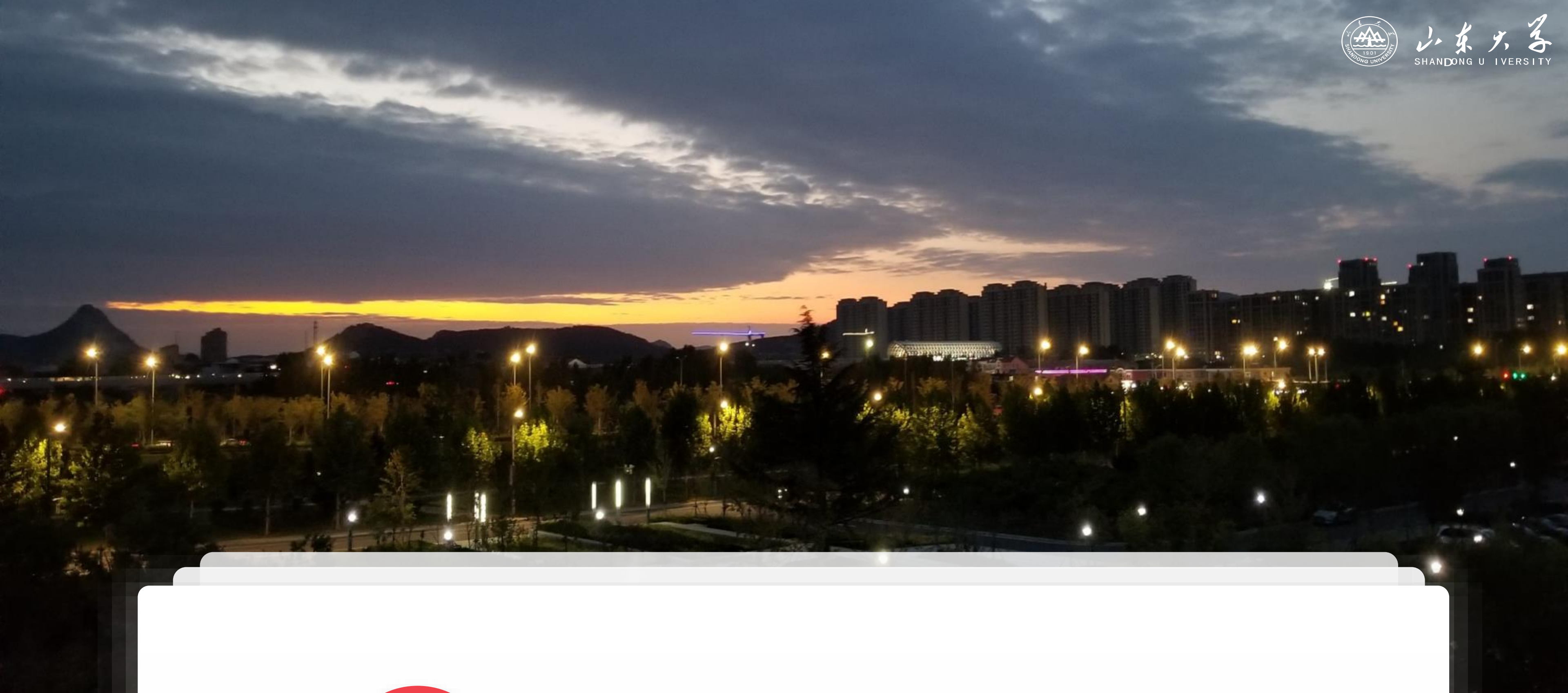
程序设计思维与实践

Thinking and Practice in Programming

贪心、二分 | 内容负责：师浩晏



1 贪 心 算 法 初 试

The beginning of the greedy algorithm

贪心热身

- 部分背包问题
  - 有 n 个物品，第 i 个物品的重量为 w_i，价值为 v_i。
  - 你有一个容量为 C 的背包，可以往里面放物品，尽量让总价值最高。
  - 每个物品可以只取走一部分，价值和重量按比例计算。
  - 样例：

    n = 5

    v = [5, 1, 3, 2, 5]

    w = [6, 3, 2, 1, 4]

    C = 5

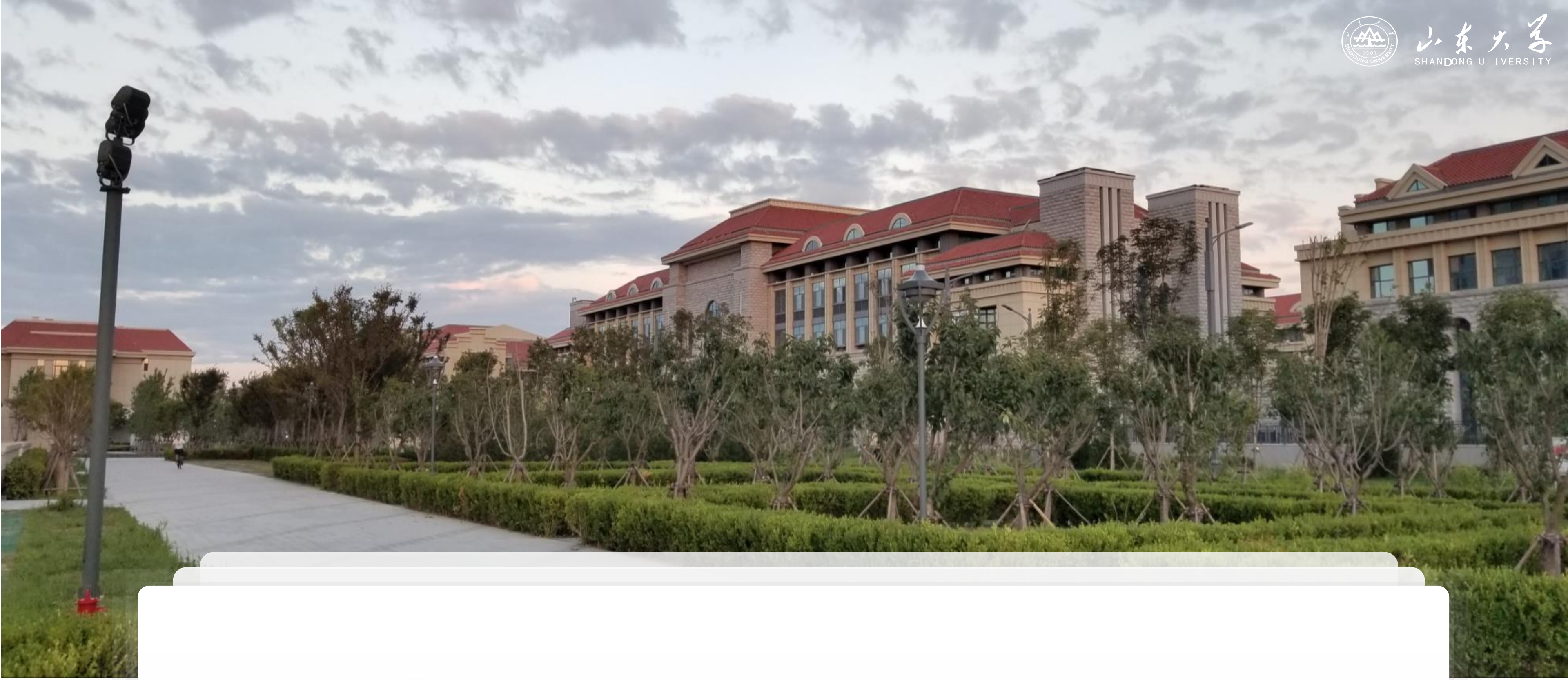
    答案为：？
  - 策略：？
    - 优先拿“单位重量价值”最大的，直到重量和正好为 C。
    - v/w = [0.83, 0.33, 1.5, 2, 1.25]

贪心热身

- 乘船问题
  - 有 n 个人，第 i 个人重量为 W_i。
  - 每艘船的最大载重量均为 C，且最多只能乘 2 个人。
  - 用最少的船装载所有的人。
  - 策略?
    - 最轻的人应该要选能和他一起坐船的人中最重的一个

贪心热身

- 乘船问题
  - 考虑最轻的人 u，他应该和谁一起坐呢？如果每个人都无法和他一起坐船，则唯一的方法就是每人各坐一艘船（想想为什么？）
    否则，他应该选择能和他一起坐船的人中最重的一个 v。
  - 因为剩下的人越轻越好



2 贪 心 算 法

The greedy algorithm

贪心算法

贪心算法（英语：greedy algorithm），是用计算机来模拟一个“贪心”的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

贪心算法

最常见的贪心题型有两种。
1.我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。
2.我们每次都取 XXX 中最大/小的东西，并更新 XXX。

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

贪心算法

最常见的贪心解法有两种。

排序解法

用排序法常见的情况是输入一个包含几个权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

贪心算法

举一个最简单的例子

假设你可以最多拥有n个红包，现在给出了m个红包，每个红包金额已知，怎么选择才能让总价值最大？

选择价值最大的n个红包即可——**排序解法**

假设每天会给你一个新的红包，但你最多只能拥有n个红包。如果现在已经有了n个红包，还想接受新的红包，就要选择其中一个已有的红包退回去。怎样获得最大价值？

无论如何，都收下每一天的新红包。如果总数超过了n，就把现在拥有的价值最少的红包退回——**后悔解法**

由此可见，排序解法里会用到排序，后悔解法里常常用到优先队列
回忆 sort() 和 priority_queue用法

贪心算法

证明方法
贪心算法有两种常用的证明方法：贪心算法领先和交换论证。一般情况下，一道题只会用到其中的一种方法来证明。

考试中不用严格证明，但在写代码之前最好确定自己使用的是正确的策略，否则会白白浪费写代码的时间和提交机会

- 贪心算法领先：每一步都比别的方法优秀，所以整体最优。相当于归纳法
- 交换论证：假设存在最优解，但这个最优解不是贪心解。按照贪心策略修改最优解，发现解法趋向于贪心解，同时不会更差，得出矛盾，从而反证贪心解为最优解

贪心算法

交换论证的详细解释

假设贪心解是A，我们想证明A是最优解

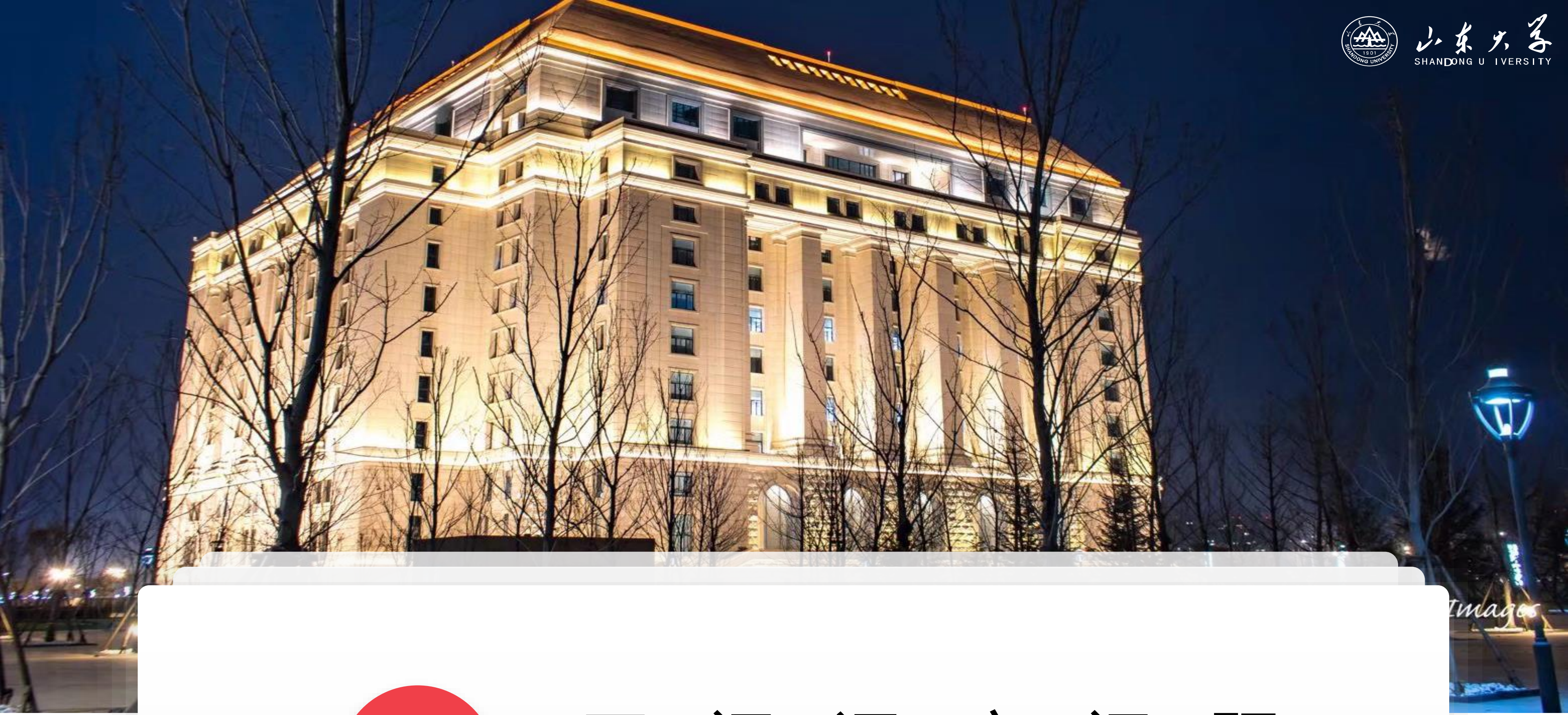
首先假设A不是最优解，那么必然存在解法O1，O1更优，同时O1和贪心解不同

由于O1 不是贪心解，O1 一定存在不满足贪心性质的部分，那么据此可以构造一个修改O1的方法，得到解法O2，O2更接近贪心解，同时O2不比O1更差 。

继续修改O2，得到O3，O4，O5。由于一直按照贪心方法修改，最终解法会变成A，同时解法不会变得更差，即A 不比 O1 差。这和最初假设矛盾

综上，我们只要能得出，按照贪心性质对某个解法进行修改，会让它变得更优，那么贪心解就是正确的

很多时候，这种修改是交换解法序列中某两个元素的顺序

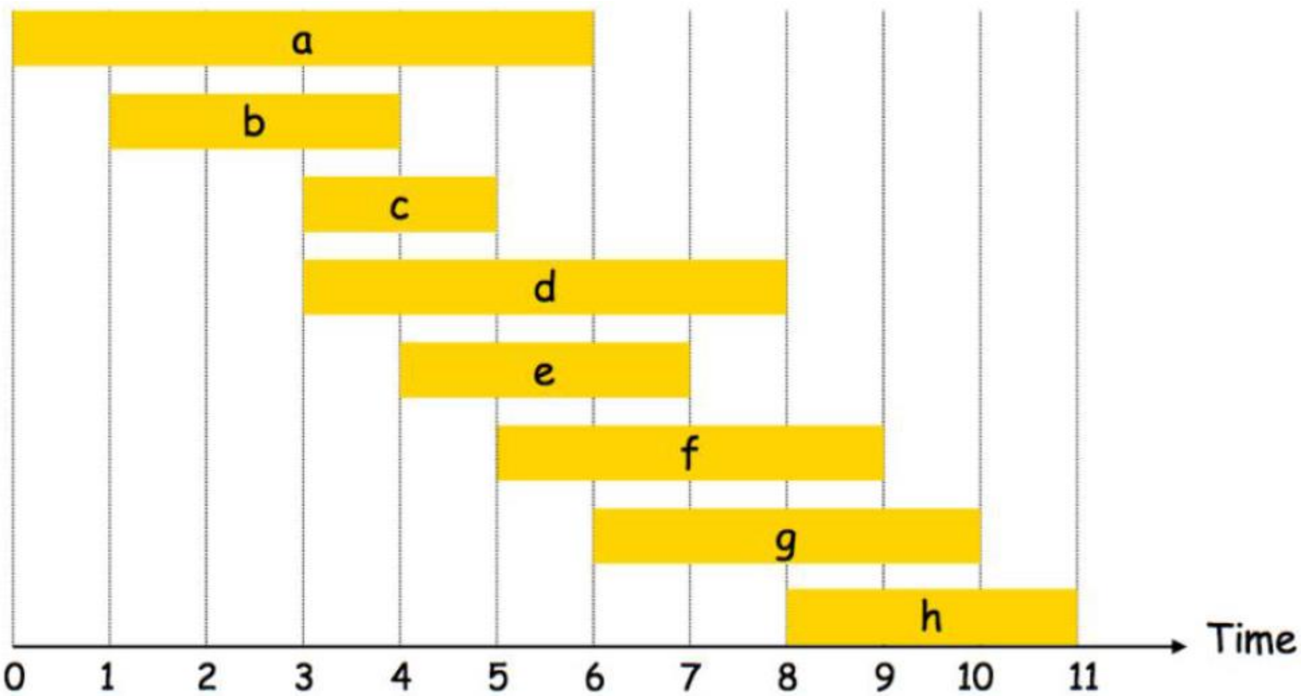


3 区 间 调 度 问 题

Interval scheduling problem

区间调度问题

- 区间调度问题
  - 区间 j 从 S_j 时刻开始，到 F_j 时刻结束 （$F_j > S_j$）
  - 两个区间 i 和 j 相容当且仅当它们不重叠 （$F_i <= S_j$ 或者 $F_j <= S_i$）
  - 目标：找到最多数量的两两不冲突的区间

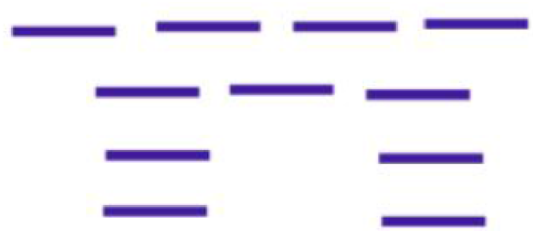


区间调度问题

- 区间调度问题 —— 贪心算法
  - 前面提到了贪心算法要有一个指标，现在我们要选择一个贪心指标，根据它，来优先选择一些区间
  - 我们应该用哪些指标?
  - 最早的开始时间 S_i ?
    - 反例

  - 最短的区间长度 F_i - S_i ?
    - 反例

  - 最少冲突的区间？
    - 反例

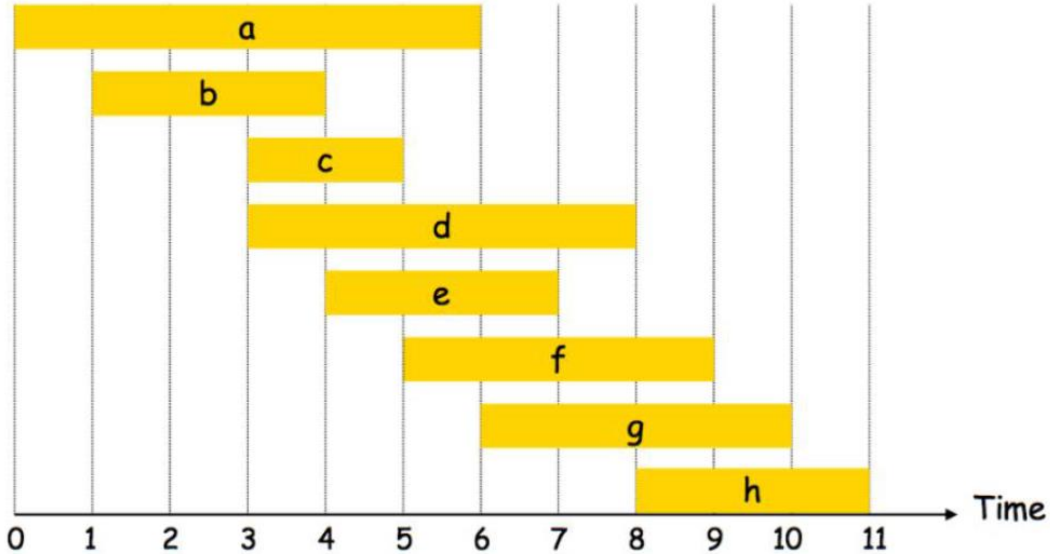


  - 选择 —— 最早的完成时间 F_i 或者最晚的开始时间S_i

区间调度问题

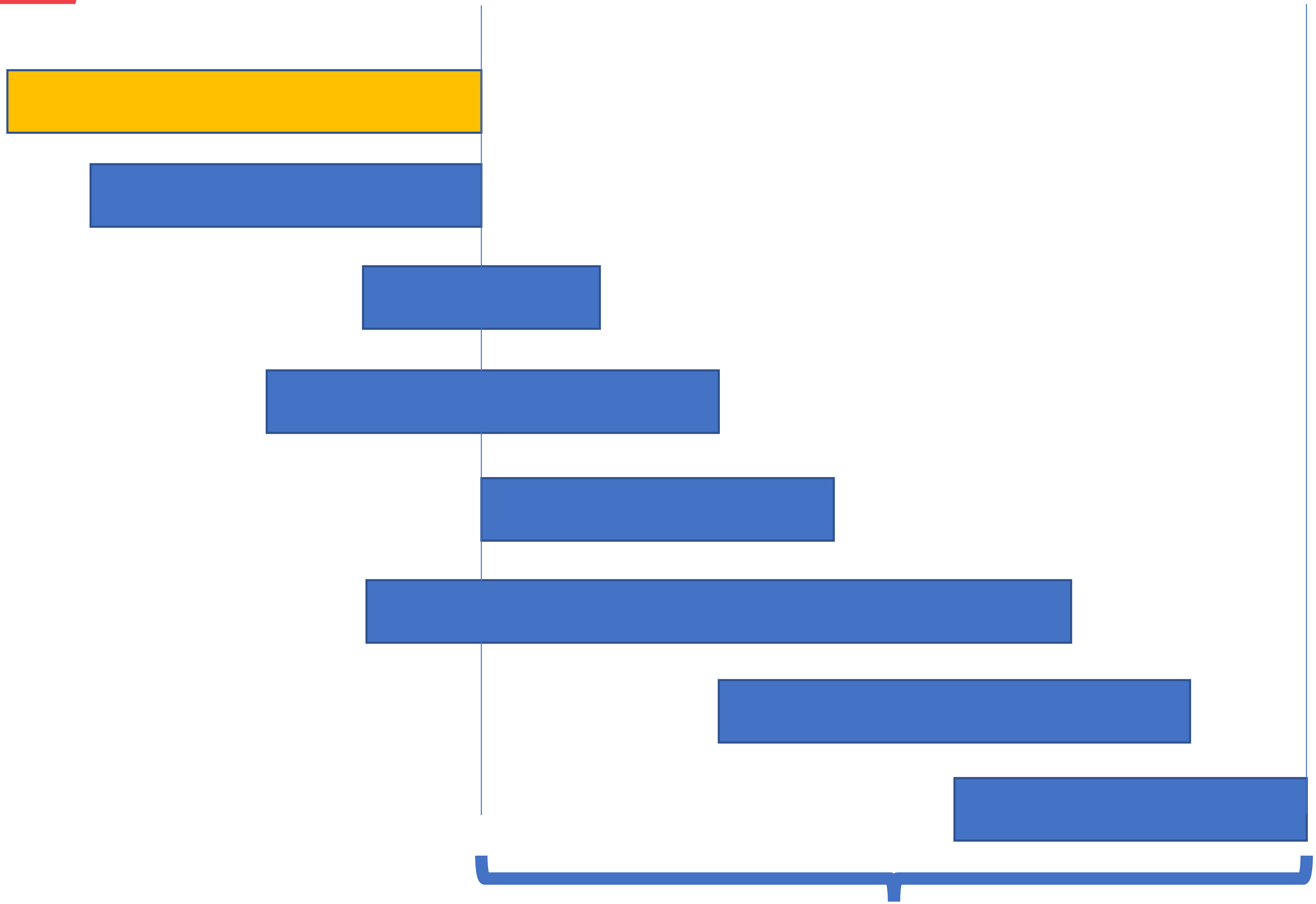
- 区间调度问题 —— 贪心算法
  - 选择 —— 最早的完成时间 F_i
  - 怎么证明?

Init：令 R 是所有区间的集合，A 为空
While R 不空:
    选择 R 中最小结束时间的区间 i
    把 i 加到 A 中
    从 R 中删除与区间 i 不相容的所有区间
Return A

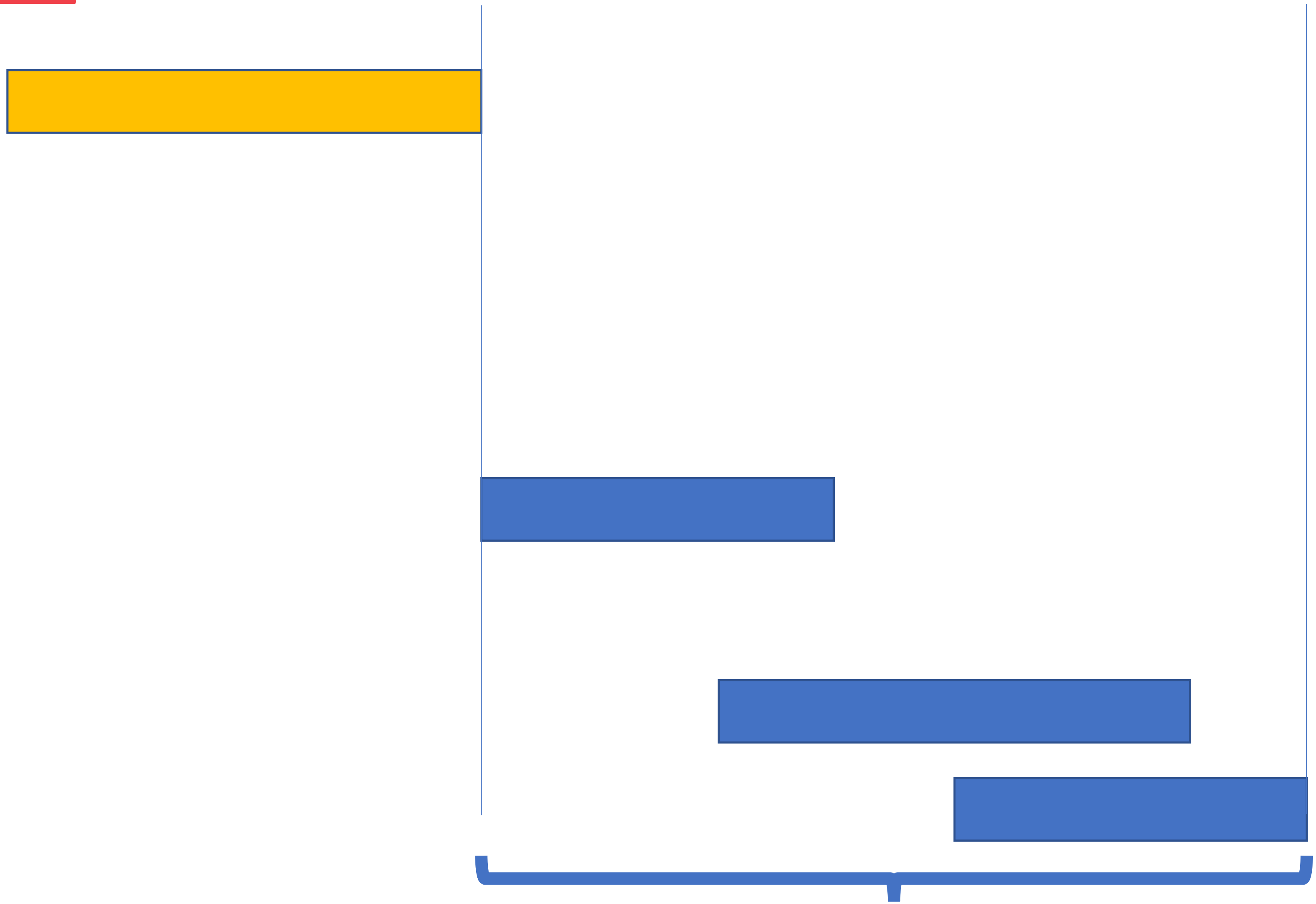


区间调度问题

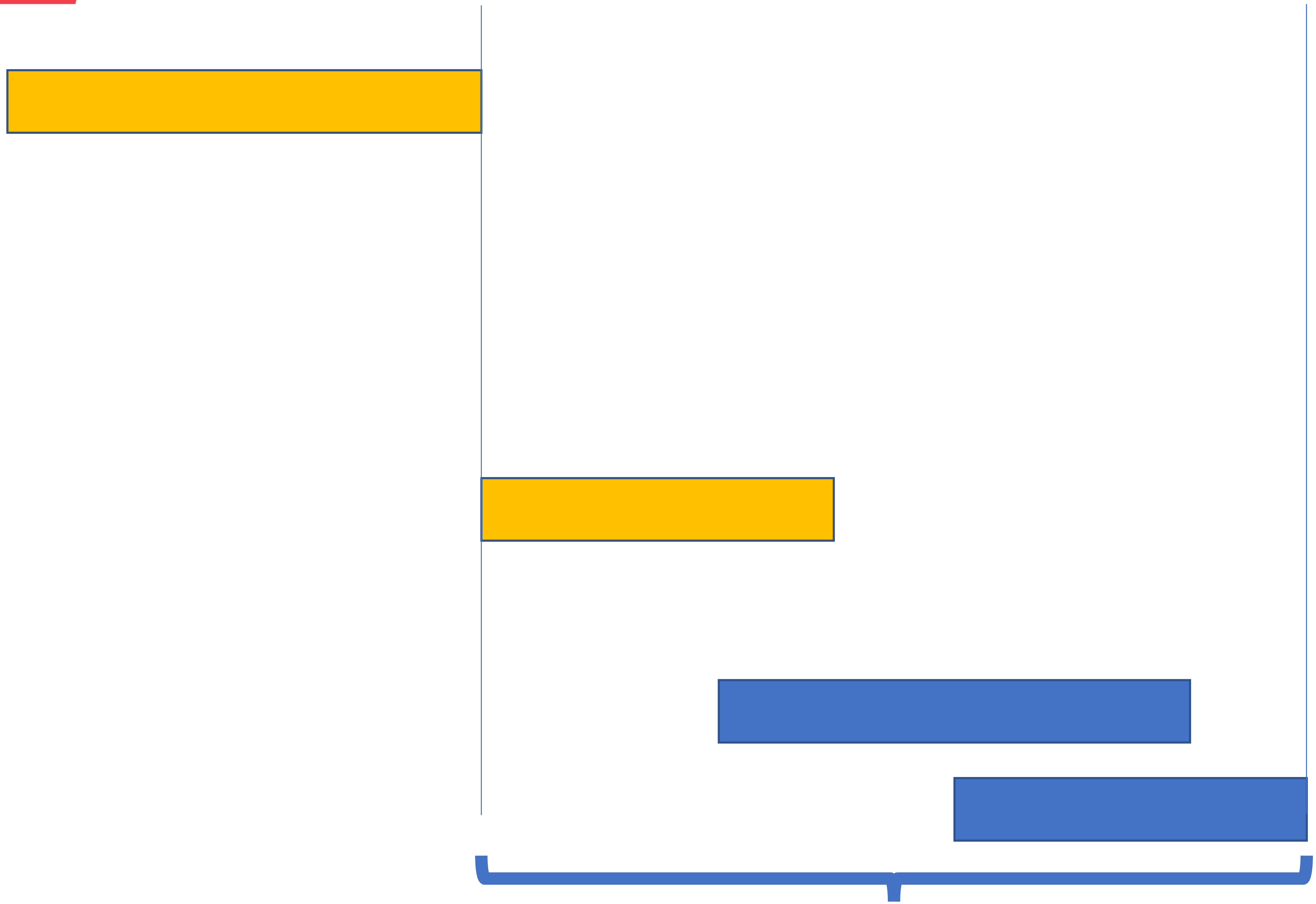
区间调度问题



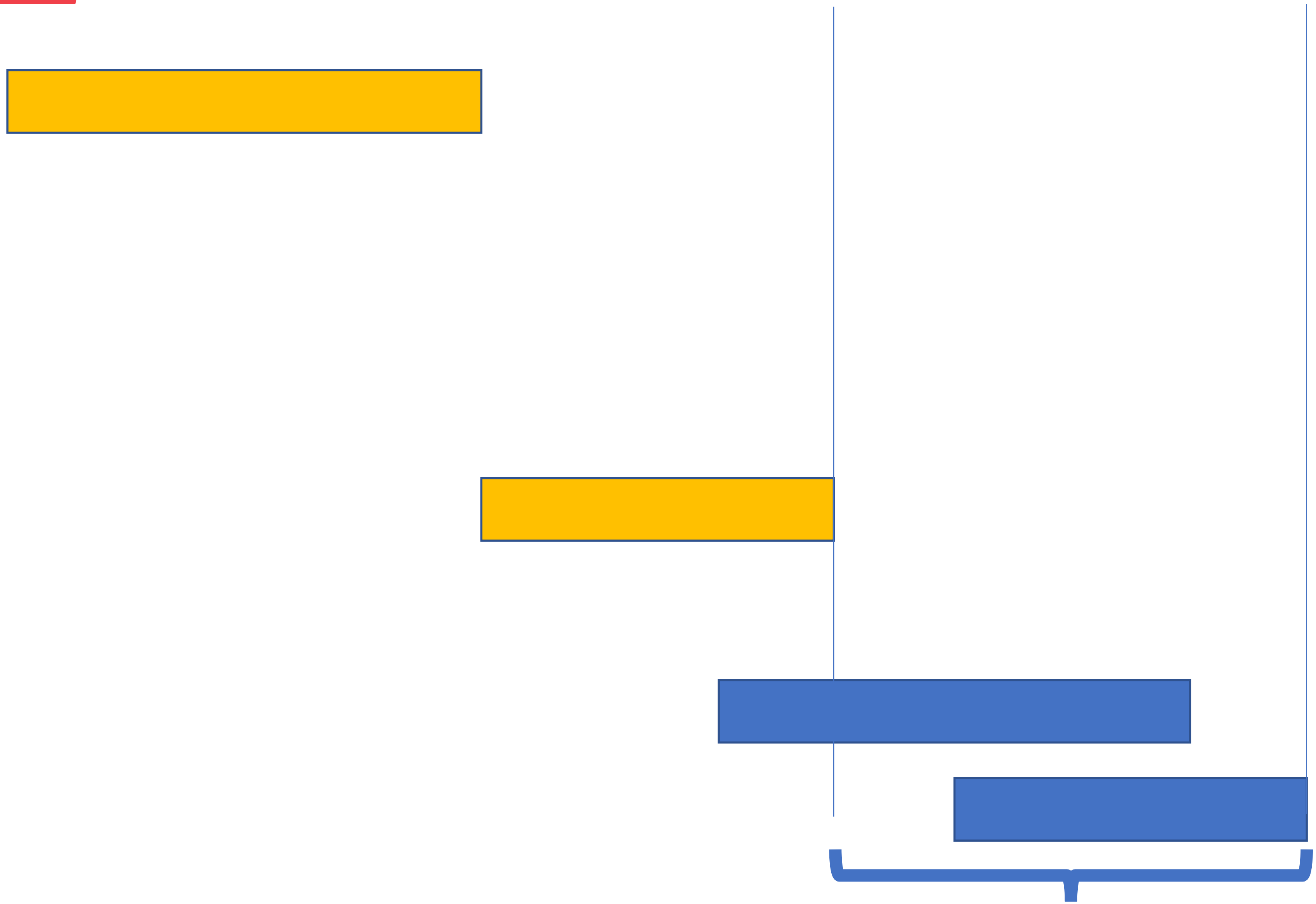
区间调度问题



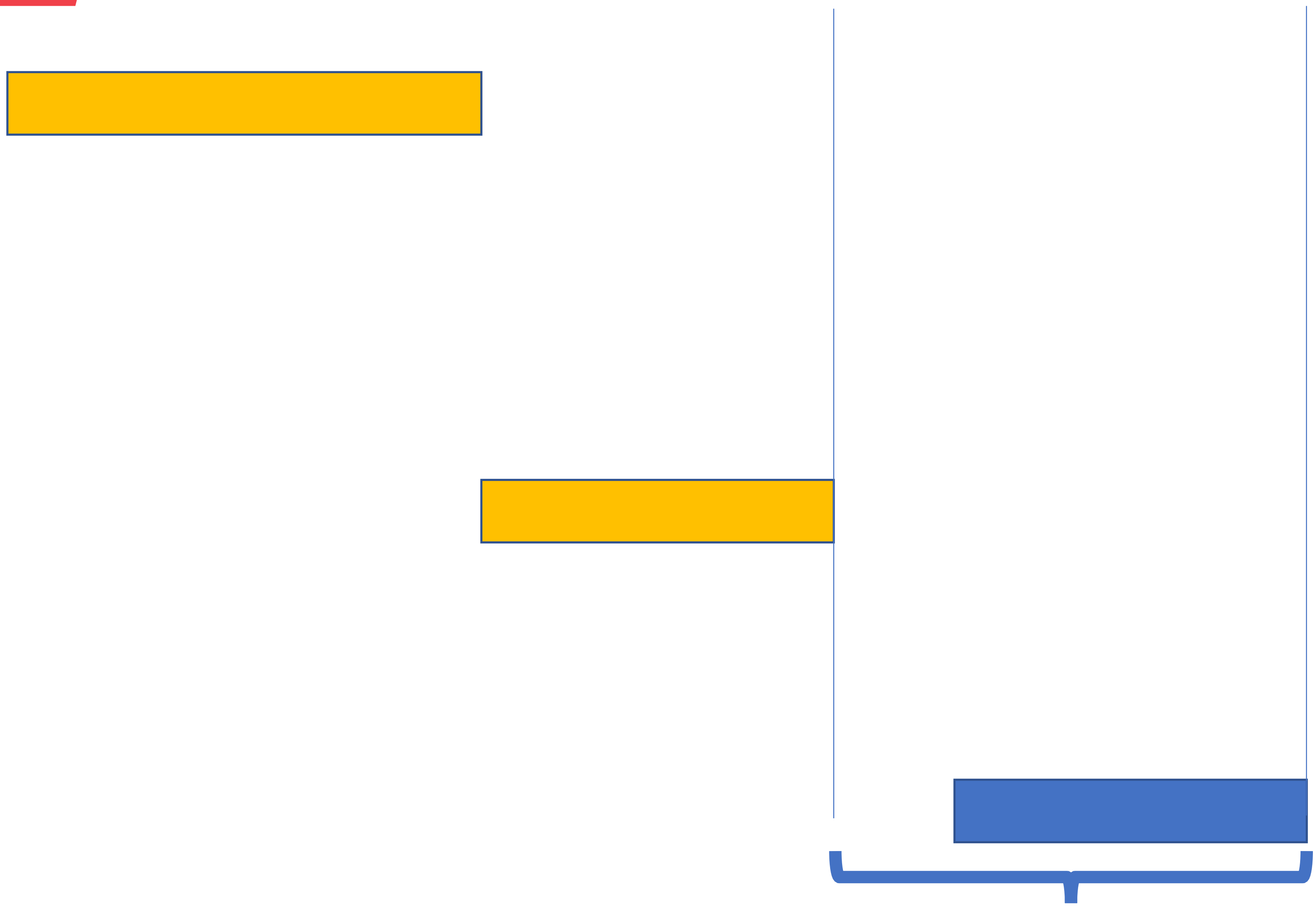
区间调度问题



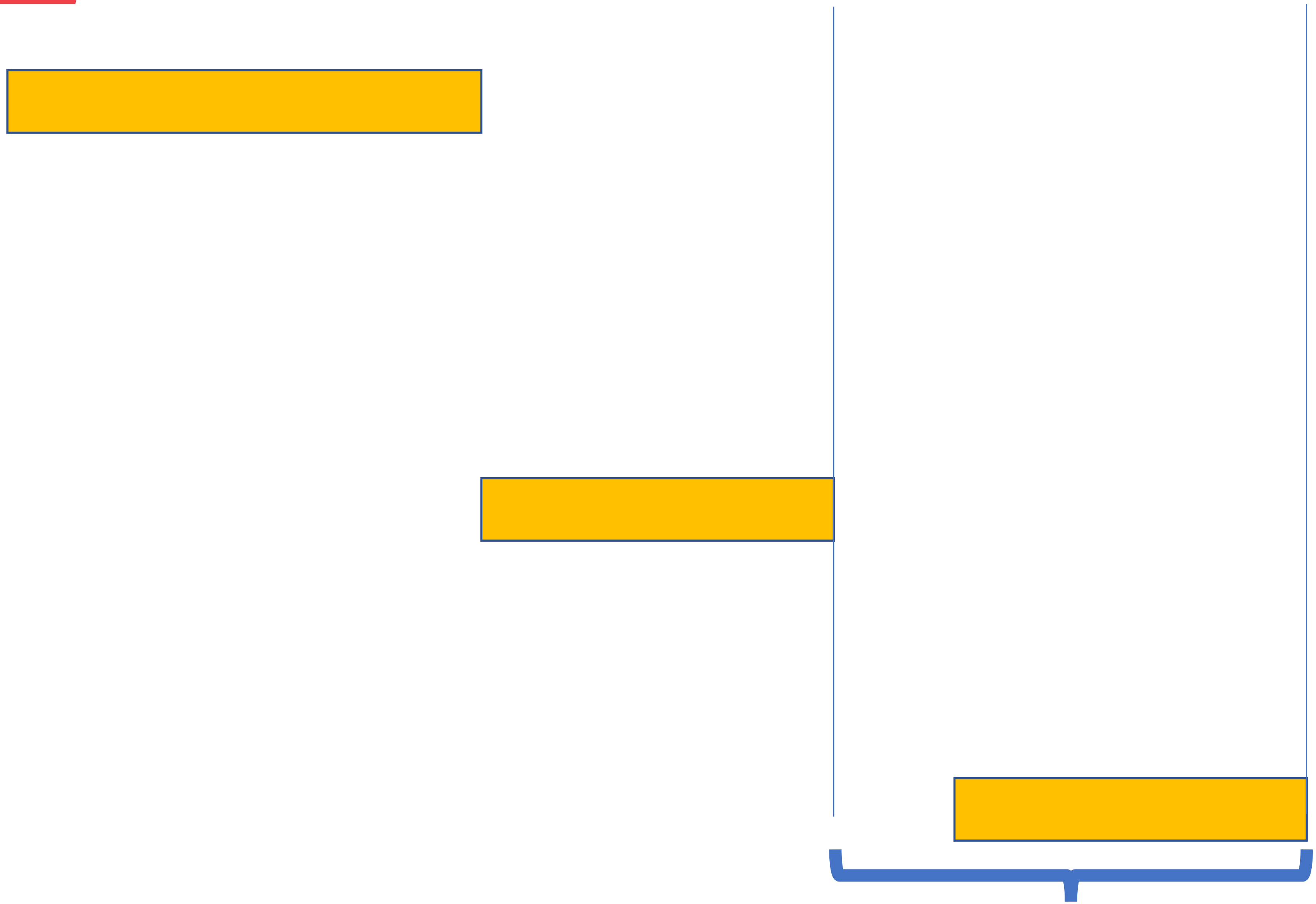
区间调度问题



区间调度问题



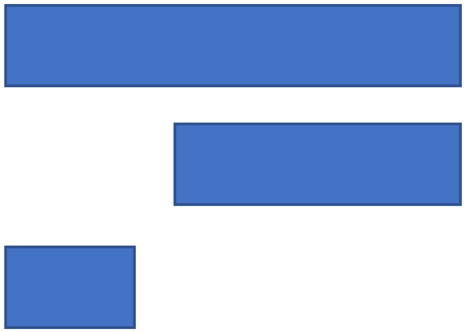
区间调度问题



区间调度问题

- 区间调度问题 —— 贪心算法
- 正确性证明：贪心算法领先
- 每选择一个区间，都会排除掉开始时间小于该区间结束时间的区间
- 选择的区间结束越早，下一步能选择的范围就越大

- 思考：如果两个区间结束时间相同，如何选择



5 工 作 调 度

Work scheduling problem

工作调度

- 问题引入
  - 约翰的工作日从 0 时刻开始，有 1e9 个单位时间。在任一单位时间，他都可以选择编号 1 到 n(1,1e5) 工作中的任意一项工作来完成。工作 i 的截止时间是 D_i(1,1e9)，完成后获利是 P_i(1,1e9)。在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少。

工作调度

做法：

- 先考虑一个简化的版本：

- 给定一个时刻t。那么t时刻前最多可以完成多少个工作？
- 显然是t个，每个时刻都做一个工作

工作调度

做法：

- 先考虑一个简化的版本：

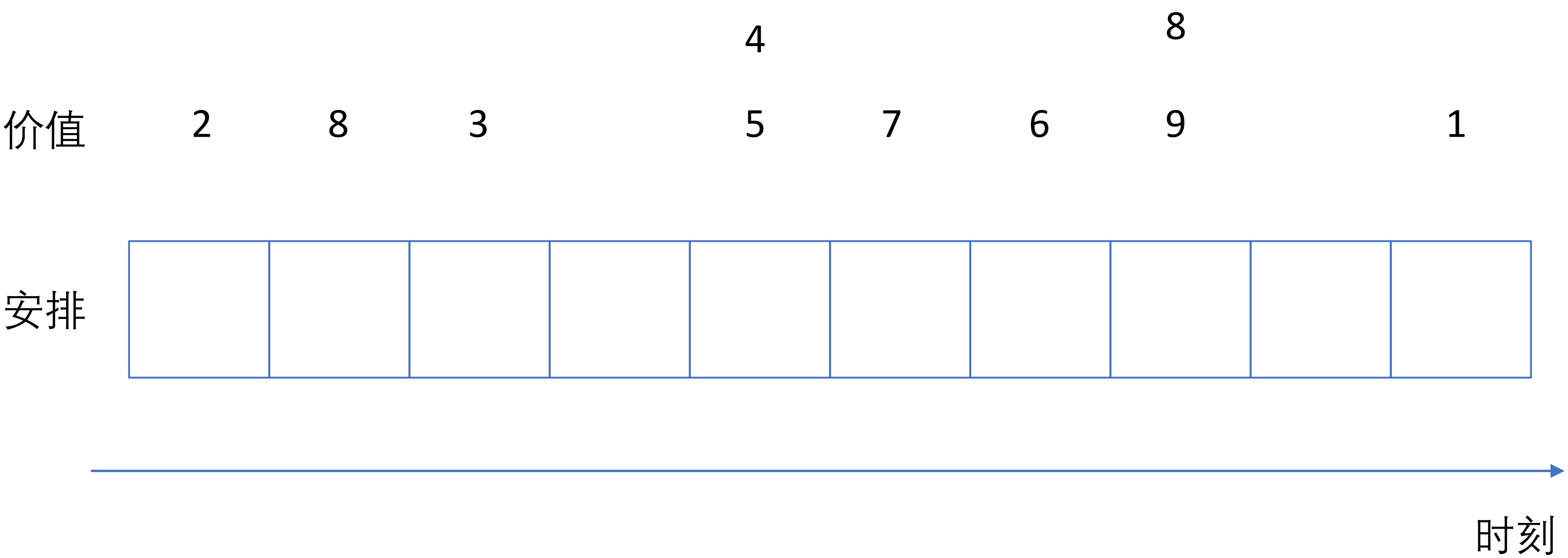
- 假如所有工作的截止时间都是 k 时刻，该如何选择？
- 0时刻到k时刻一共有k个单位时间，选取k个价值最大的工作即可

工作调度

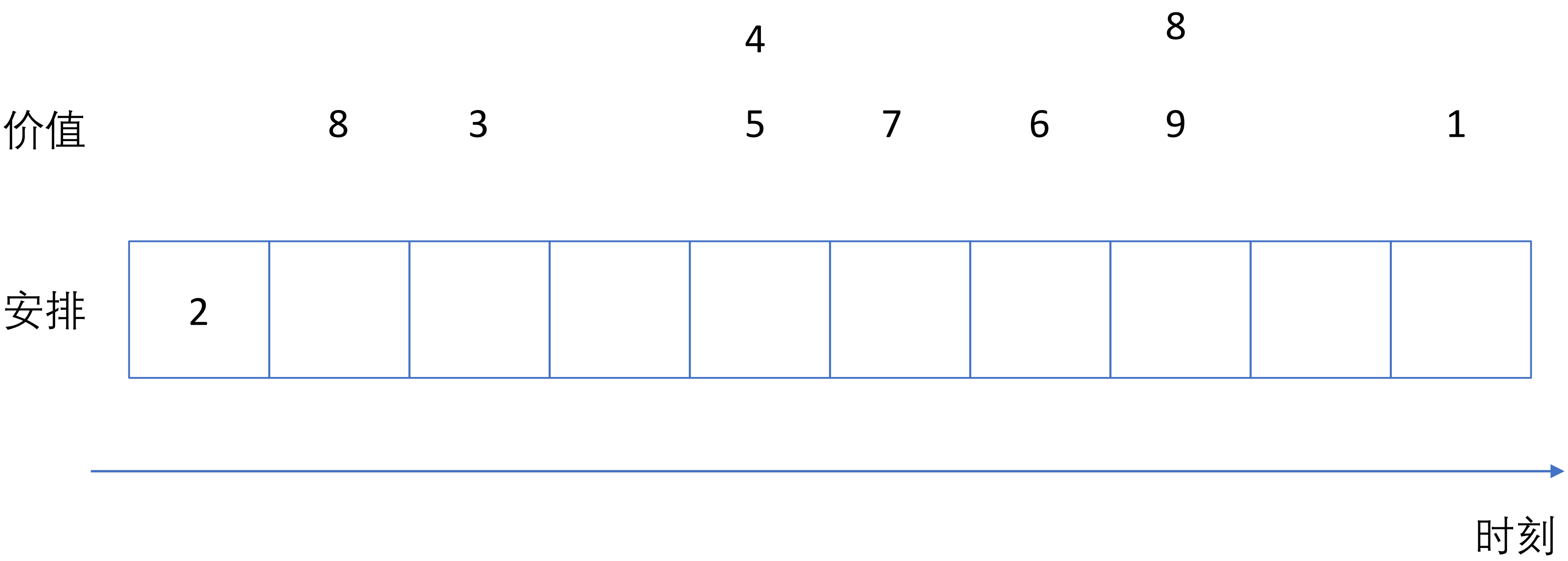
做法:

- 现在问题来了，许多工作的截止时间不同。
- 我们能不能按照截止时间进行排序，然后尽可能的安排工作?

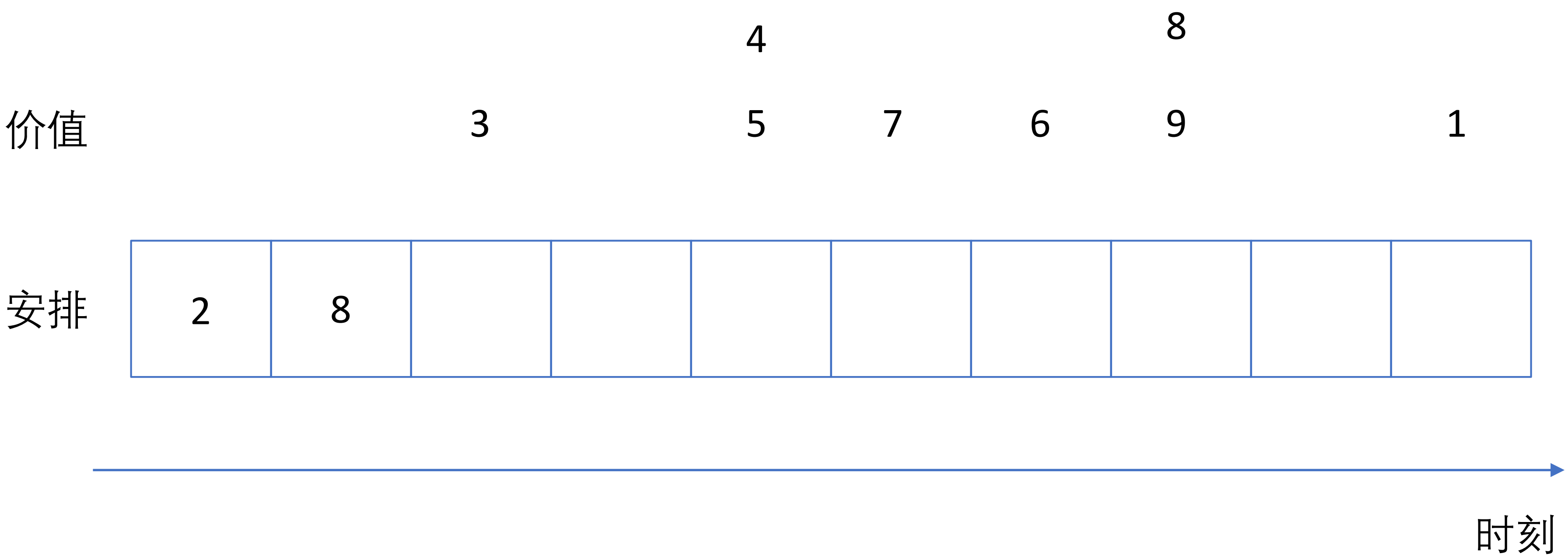
工作调度



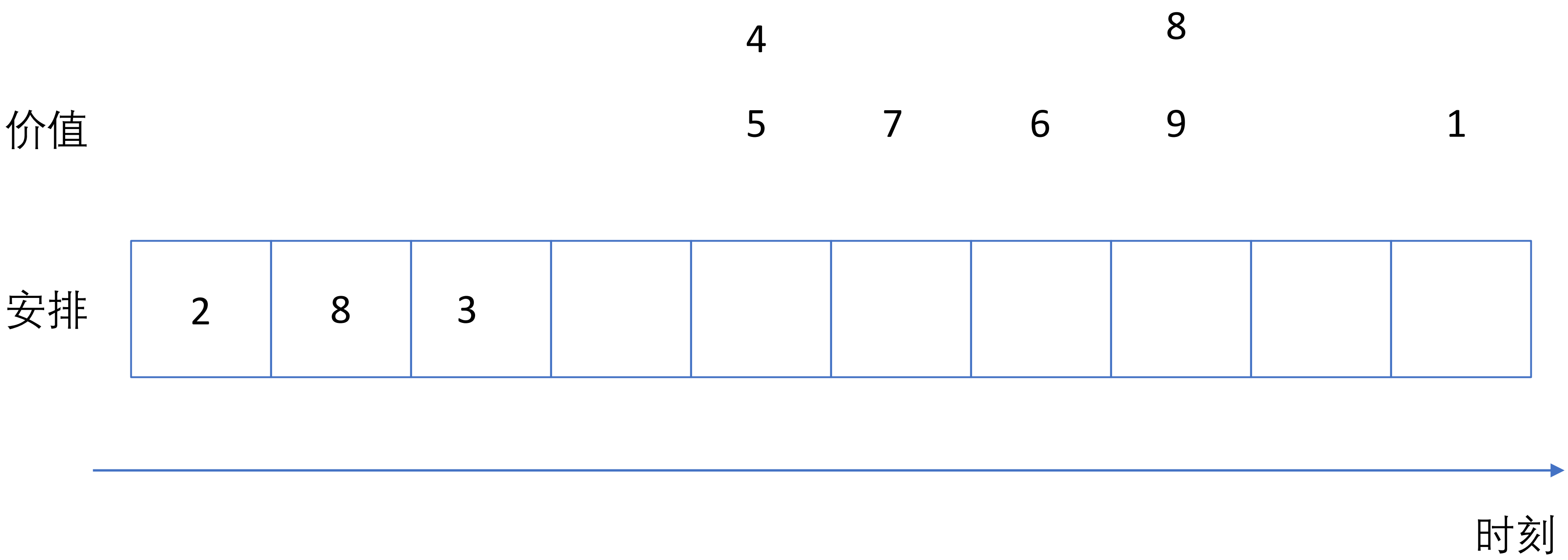
工作调度



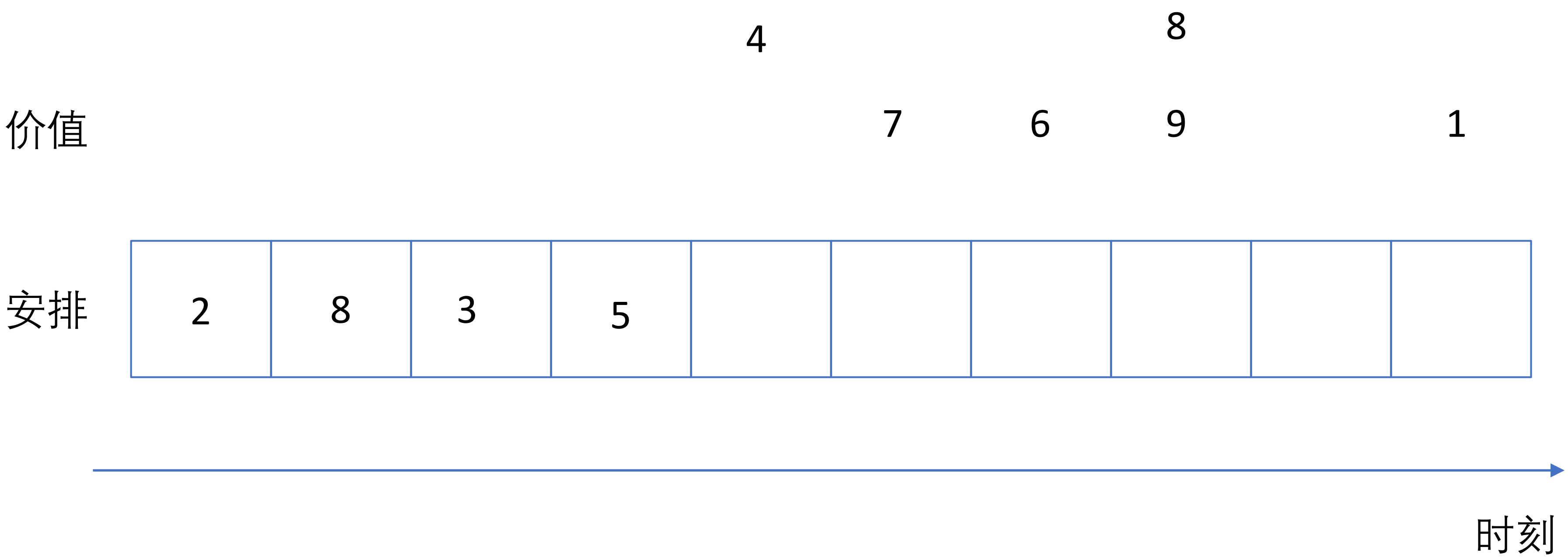
工作调度



工作调度



工作调度

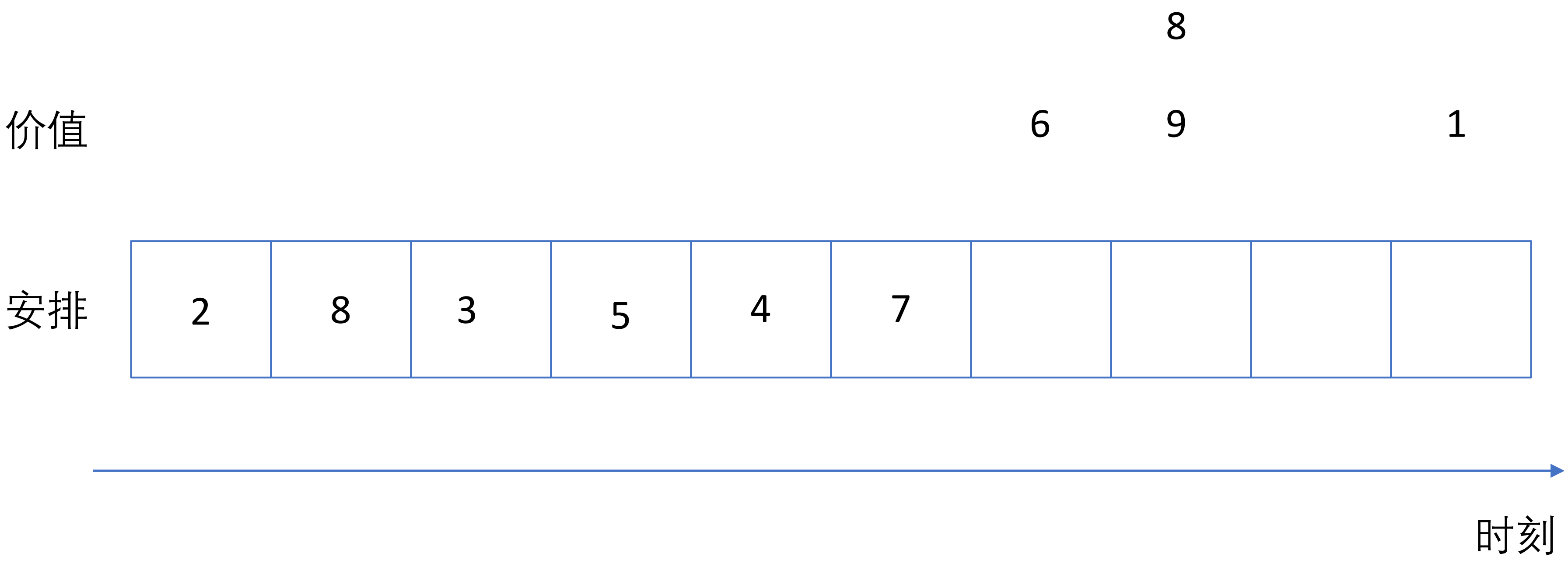


工作调度

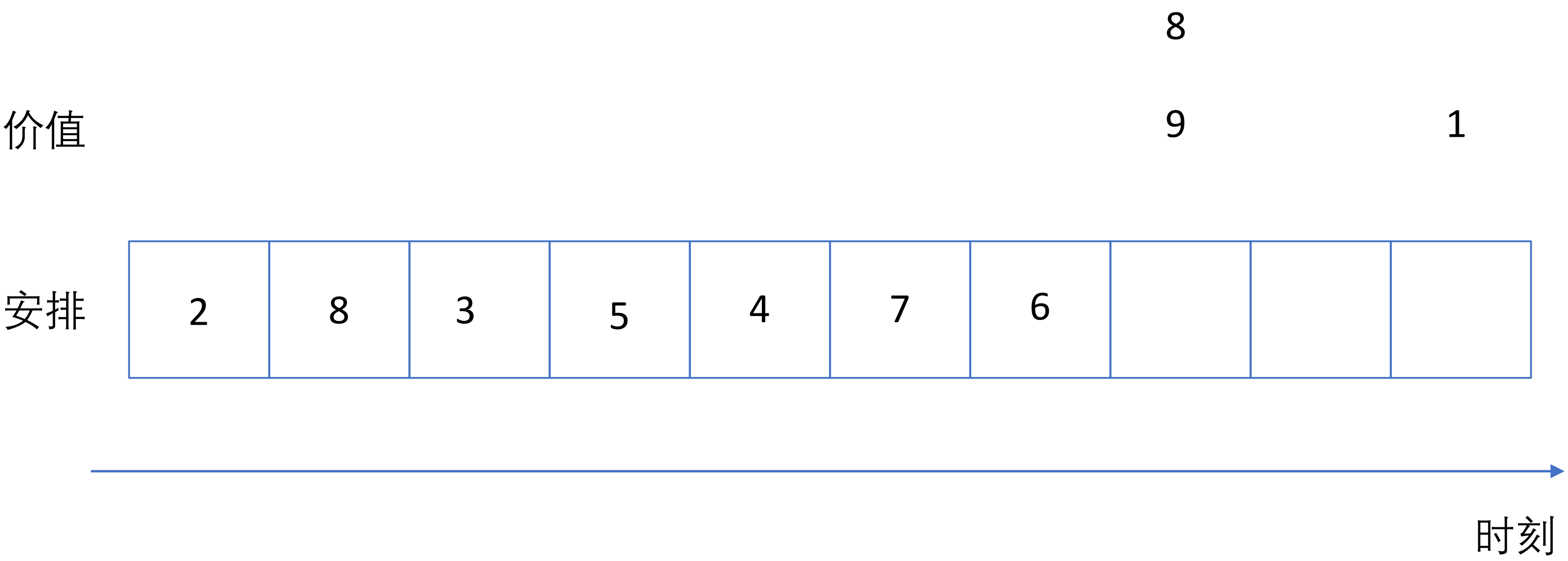
价值						7	6	8 9		1
安排	2	8	3	5	4					

时刻

工作调度



工作调度



工作调度

8

价值 1

安排	2	8	3	5	4	7	6	9		

时刻

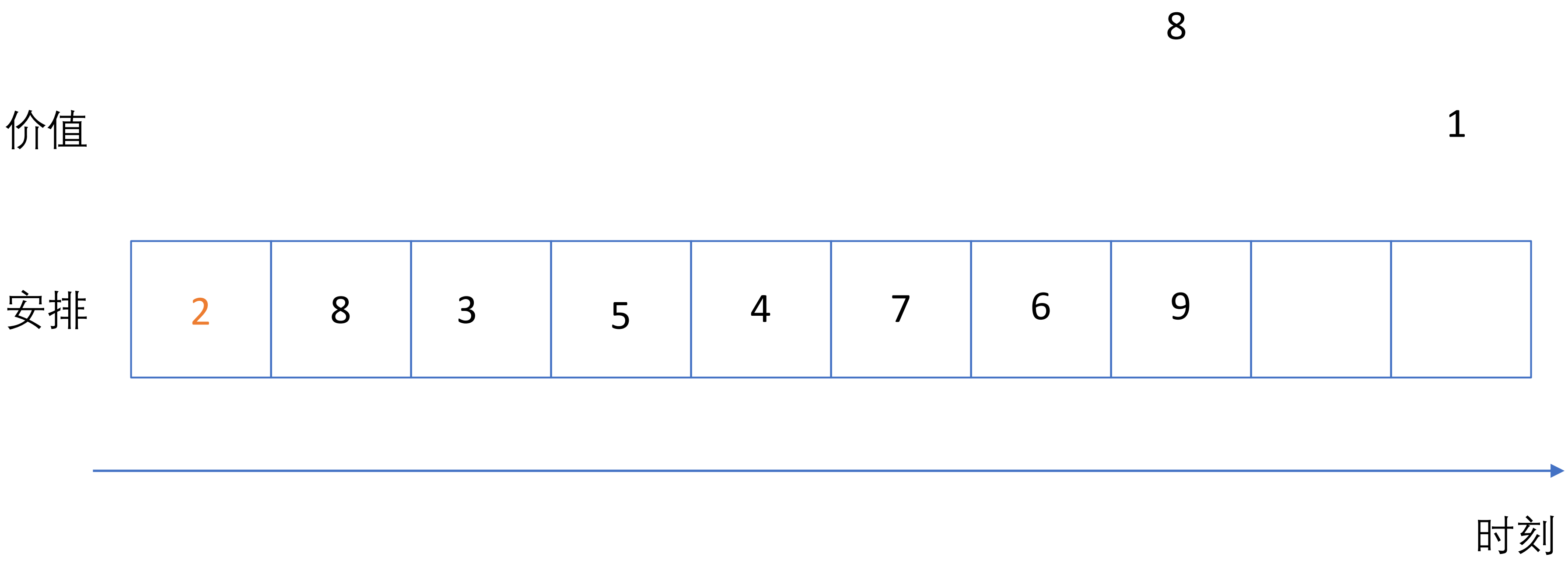
工作调度

做法：

- 现在问题来了，许多工作的截止时间不同。
- 我们能不能按照截止时间进行排序，然后尽可能的安排工作？

- 一些截止较早的工作可能价值不高，但占用了时间
- 反悔操作：如果排不下，就选择一个价值最低的，换成一个截止时间晚但价值高的工作

工作调度



工作调度

价值 1

安排	8	8	3	5	4	7	6	9		

时刻

工作调度

价值

安排	8	8	3	5	4	7	6	9	1	

时刻

工作调度

- 思考：正确性证明?

- 如何实现?
- 先按照截止时间从小到大对所有工作排序，按顺序考虑每个工作
- 用一个数组记录每个时间段的安排，模拟上述过程

工作调度

- 问题：数组要开1e9长度，开不下
- 时间复杂度太高，会超时

- 优化：

- 先按照截止时间从小到大对所有工作排序，按顺序考虑每个工作的安排
- 用一个容器存放暂时确定要做的工作
- 假设当前要工作的截止时间为 t ，那么包括这个工作最多可以做 t 个工作。先把这个工作加入容器，如果容器内工作数量超过t个，则删除容器内价值最小的工作
- 显然，这个容器用优先队列最合适

6 国 王 游 戏

The King’s Game

国王游戏

- 问题引入
  - 恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。 国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。
  -

$$1 \le n \le 1,000, 0 < a, b < 10000$$

国王游戏

- 解法
  - 相邻的两个位置交换不会影响其他人的答案，只会改变交换位置的两个人
  - 设排序后第 i 个大臣左右手上的数分别为a_i,b_i 。考虑通过邻项交换法推导贪心策略。
  - 用 s 表示第 i 个大臣前面所有人的 左手数字 的乘积，那么第 i 个大臣得到的奖赏就是 s/b_i，第 i+1 个大臣得到的奖赏就是$s*a_i/b_{i+1}$。
  - 如果我们交换第 i 个大臣与第 i+1 个大臣，那么此时的第 i 个大臣得到的奖赏就是 $s*a_{i+1}/b_i$，第 i+1 个大臣得到的奖励就是 s/b_{i+1}。

国王游戏

- 解法
  - 如果交换前更优当且仅当

$$max(\frac{s}{b_i}, \frac{s*a_i}{b_{i+1}}) < max(\frac{s}{b_{i+1}}, \frac{s*a_{i+1}}{b_i})$$

  - 提取出相同的 s 并同时都乘上b_i*b_{i+1}

$$max(b_{i+1}, a_i * b_i) < max(b_i, a_{i+1} * b_{i+1})$$

```
struct uv {
  int a, b;
  bool operator<(const uv &x) const {
    return max(x.b, a * b) < max(b, x.a * x.b);
  }
};
```

4 二分

Binary Search

二分

- 假如一个问题的答案有若干个可能的值。二分法就是不断缩小可能取值的范围（通常是分成两个基本相同大小的部分，排除掉其中一个），逐渐锁定答案。复杂度为 O(log(n))

- 由答案可取值的范围可以分为整数二分和浮点数二分

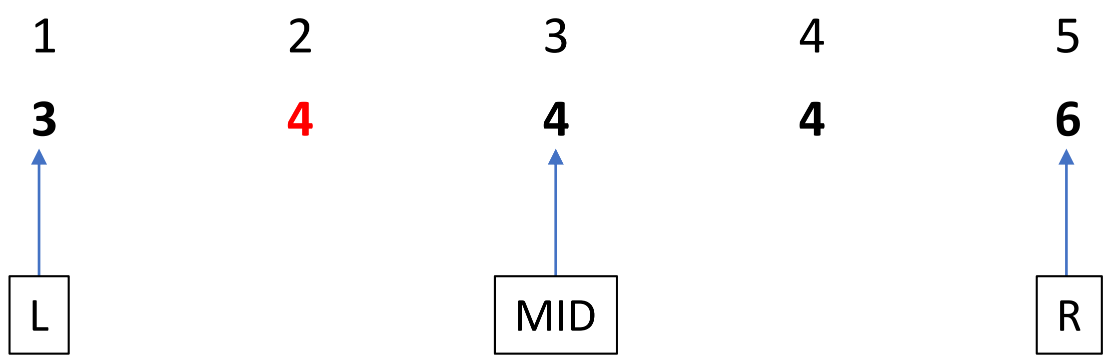
整数二分

- 回顾一下整数二分的最基本问题：给定一个数组，该数组满足单调不降性，从中查找第一个大于等于给定数的位置。若不存在返回数组最后一个位置的下一个位置
- 下标：1，2，3，4，5
- 数组：3，4，4，4，6，查找4
- 返回：   2

整数二分

1	2	3	4	5
3	**4**	**4**	**4**	**6**

整数二分

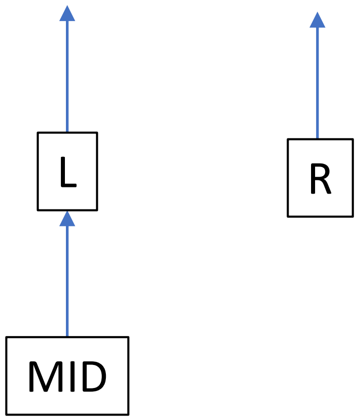


整数二分

1	2	3	4	5
3	**4**	**4**	**4**	**6**
L	MID	R		

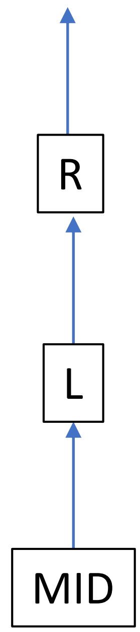
整数二分

1	2	3	4	5
3	**4**	**4**	**4**	**6**



整数二分

1	2	3	4	5
3	**4**	**4**	**4**	**6**



整数二分

- 相信二分的思路大家都比较熟悉。但二分的关键是如何正确地，优美地实现二分查找。

- 二分查找的位置的本质，其实相当于查找一个分界线：

- 查找单调不降数组中第一个大于等于给定值的位置：

- 3，4，4，4，5

- 实际上相当于查找“大于等于给定值”和“小于给定值”两部分的分界，而分界线右侧第一个位置即为所求

- 3，4，4，4，5

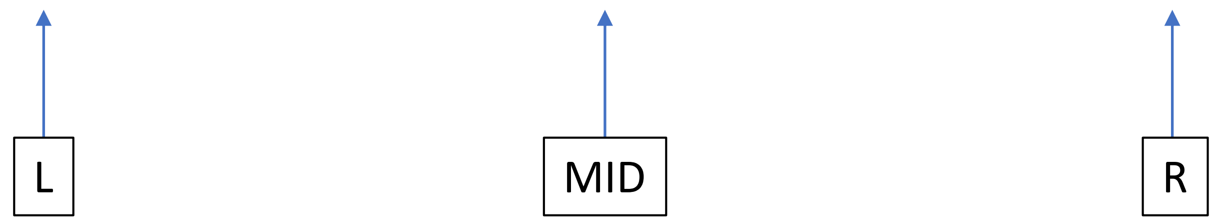
小于4　　大于等于4

整数二分

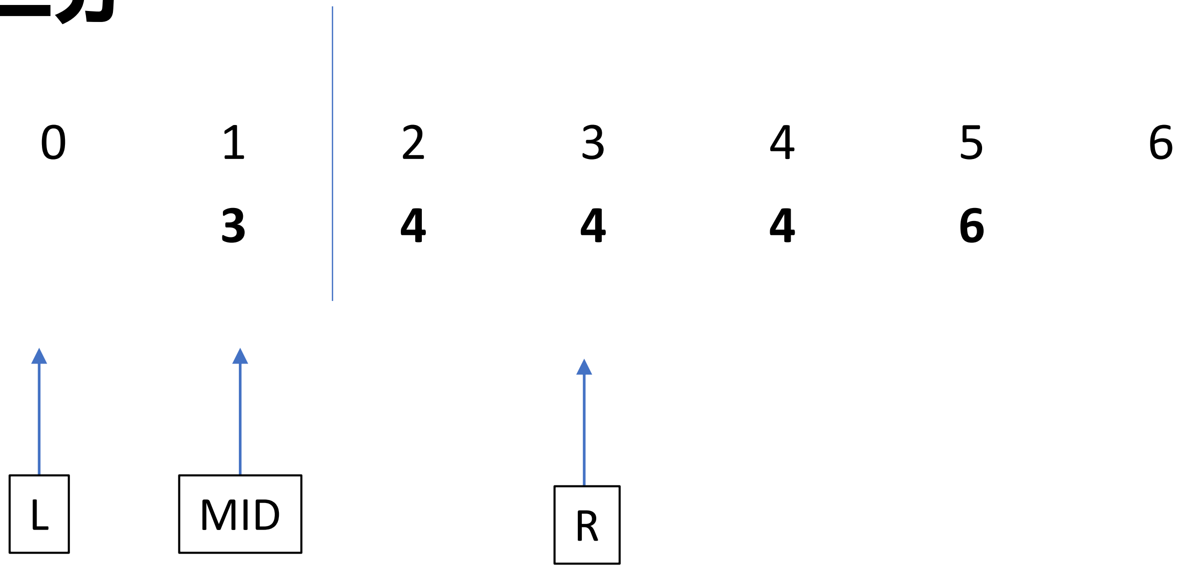
- 一个常用的二分实现方法如下：
- 设置L和R指针，其中L指向第一个元素的前一个位置，R指向最后一个元素的下一个位置。我们让L始终指向分界线以左的部分，让R始终指向分界线以右的部分。在保证这个前提的基础上，让L和R逐渐靠近，当L和R指向相邻两个数时，L和R之间就是分界线
- 那么怎么让L和R逐渐靠近呢？我们每次都取L和R的中间位置M，判断M所指的位置应该在分界线左侧还是右侧。如果是左侧，则把L移动到M的位置，否则把R移动到M的位置

整数二分

0	1	2	3	4	5	6
	3	**4**	**4**	**4**	**6**	



整数二分



整数二分

0	1	2	3	4	5	6
	3	**4**	**4**	**4**	**6**	

L MID R

整数二分

0	1	2	3	4	5	6
	3	**4**	**4**	**4**	**6**	

L R

整数二分

- 代码实现

```
int L = 0, R = n + 1;
while(R - L > 1){
    int MID = (L + R) / 2;
    if(a[MID] >= target) R = MID;
    else L = MID;
}
return R;
```

- 实际上这个过程就是lower_bound所做的。

- 思考：为什么L和R的初值分别要取第一个元素的前一个位置和最后一个元素的后一个位置

- 思考：这样实现的好处？

整数二分

- 任务2
  - 设计一个函数$find(x)$，能够在给定的一组升序数列A中，找到x最后一次出现的位置。如果x不存在输出-1

  - 三步走
  - 1. 把问题转换为查找分界线的位置：即“小于等于给定值”和“大于给定值”两部分的分界线，判断分界线左侧数字即可
  - 2. 让L和R指向第一个元素的前一个位置和最后一个元素的后一个位置
  - 3. 让L和R靠近，直到L和R指向相邻两个数

  - 对于这个问题，需要特判-1的情况

整数二分

- 在一个单调有序(递增或递减) 的区间[a_1,a_n] 中查找元素x，每次将区间分为左右长度相等的两部分，判断解在哪个区间中并调整区间上下界，不断重复直至找到相应的解。
- 时间复杂度 $O(logn)$
- 优于顺序查找 $O(n)$

作用:

1. 查找元素（位置或值）
2. 求满足条件的最值（最大值/最小值）

5

浮点二分

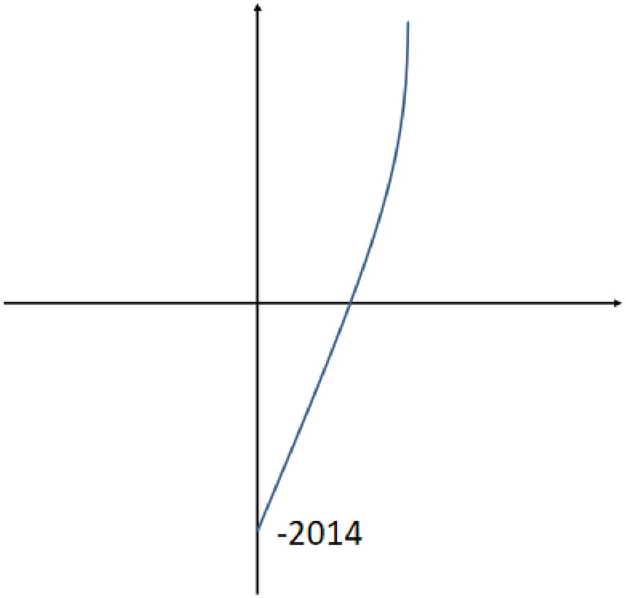
Real Binary

浮点二分

- 1.求方程：
- $x^6 + x - 2014$
- 在$(0, +\infty)$ 的一个解。
- 小数点后精确5位。

浮点二分

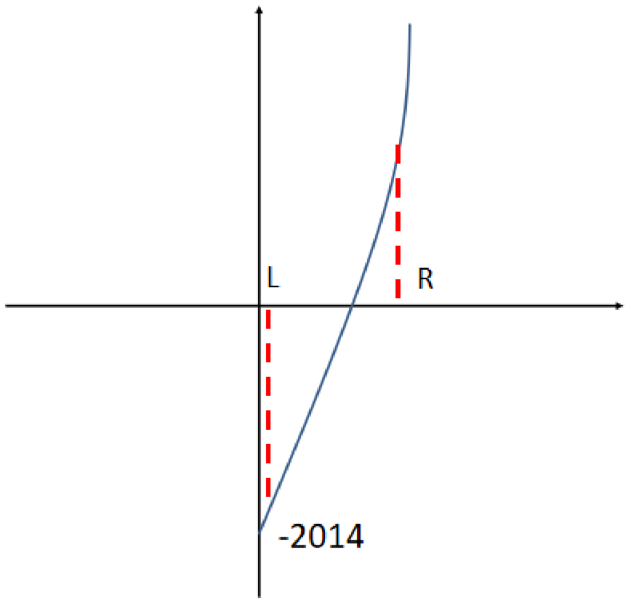
- $f(x) = x^6 + x - 2014$ 在$(0,+\infty)$单调递增



浮点二分

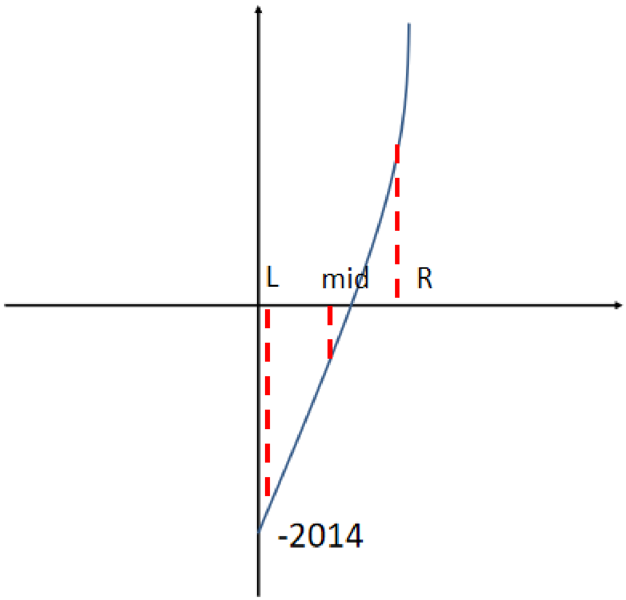
- $f(x) = x^6 + x - 2014$ 在$(0,+\infty)$单调递增

- 估算：
- L = 1.0;   //f(1.0)<0
- R = 10.0;   //f(10.0)>0
- f(L) * f(R) < 0
- 解的区间：[1.0, 10.0]



浮点二分

- L = 1.0;
- R = 10.0;
- mid = (L + R) / 2;
- if (f(mid) * f(R) <= 0)
- 答案在[mid,R];
- else
- 答案在[L,mid];



浮点二分

代码实现1

eps用来控制精度误差，题目要求答案保留小数点后5位，也就是10^{-5}

```
const double eps = 1e-5;
double find() {
    double l = 1, r = 10;
    while(abs(r - l) > eps) {
        double mid = (l + r) / 2;
        if(f(mid) * f(r) <= 0) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return l;
}
```

浮点二分

代码实现2

每一次迭代区间长度减小一半，当区间长度小于10^{-5}时即可停止，需要迭代多少次?

区间长度从10减少到10^{-5}，变化幅度为10^{6} $\log(10^{6}) \sim 20$

浮点二分

代码实现2

每一次迭代区间长度减小一半，当区间长度小于10^{-5}时即可停止，需要迭代多少次?

区间长度从10减少到10^{-5}，变化幅度为10^{6} $\log(10^{6})\sim 20$

```
double find() {
    double l = 1, r = 10;
    for(int i = 1; i <= 20; ++i) {
        double mid = (l + r) / 2;
        if(f(mid) * f(r) <= 0) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return l;
}
```

6 二 分 答 案

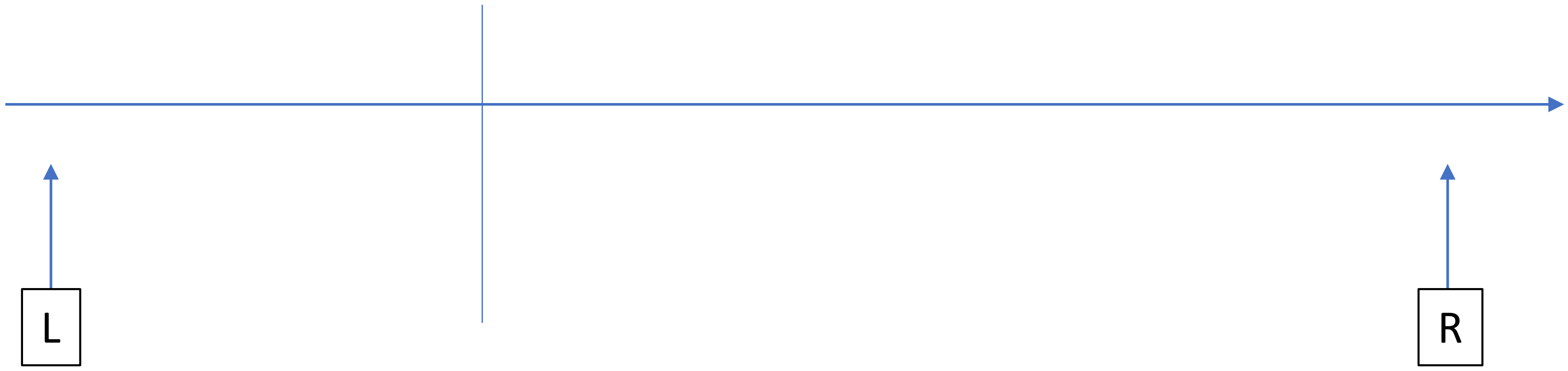
B i s e c t i o n

二分答案

解题的时候往往会考虑枚举答案然后检验枚举的值是否正确。

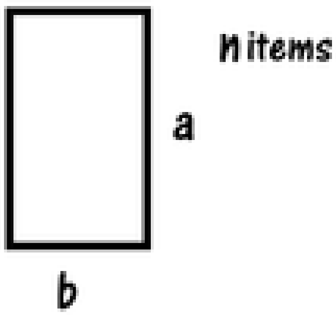
若满足要查找的位置是个分界线，则可以使用二分法找到这个分界线从而求解问题

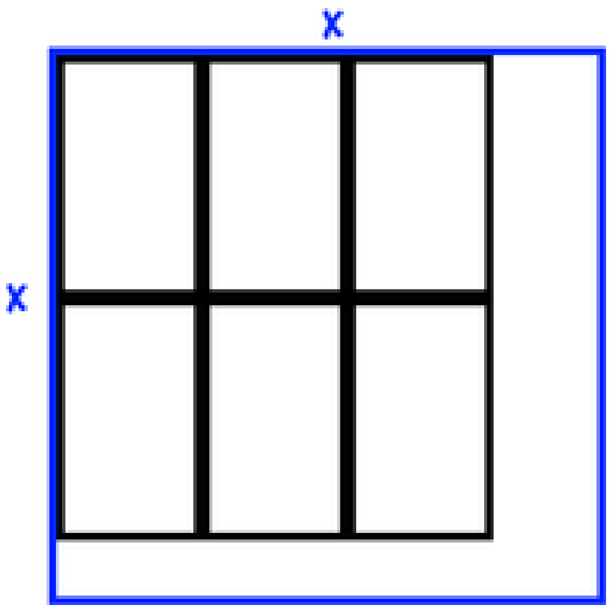
把这里的枚举换成二分，就变成了“二分答案”。



例题

- 有n个axb的矩形，我们要把它们排在一起，不能改变它们的方向。求能包含所有矩形的最小面积的正方形
- 1 <= a, b, n <= 1e9



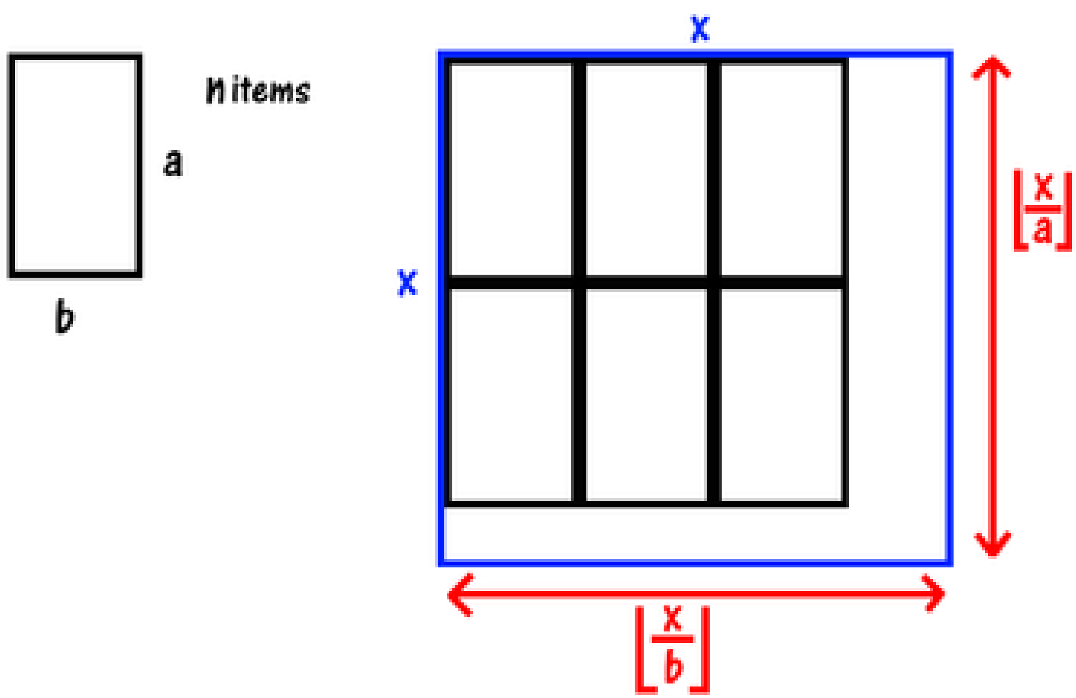


例题

答案可以取的值：大于等于1的整数

分界线：当小于某个边长时，无法包含所有矩形；大于等于某个边长时，可以包含所有矩形。因此，分界线在“不能包含所有矩形”和“能包含所有矩形”之间，答案就是分界线右侧的第一个数

需要根据边长判断是否能包含所有矩形



例题

答案可以取的值：大于等于1的整数

分界线：当小于某个边长时，无法包含所有矩形；大于等于某个边长时，可以包含所有矩形。因此，分界线在“不能包含所有矩形”和“能包含所有矩形”之间，答案就是分界线右侧的第一个数

需要根据边长判断是否能包含所有矩形

```
bool check(long long d){
    long long rows = d / a;
    long long cols = d / b;
    if(rows == 0 || cols == 0) return false;
    //使用除法而不是乘法判断，防止溢出
    return cols >= (n + rows - 1) / rows;
    //x除以y上取整的写法为，(x + y - 1) / y
}
```

```
long long L = 0, R = 1e18;
while(R - L > 1){
    long long M = (L + R) / 2;
    if(check(M))R = M;
    else L = M;
}
return R;
```

感谢收听

Thank You For Your Listening