

# **Lecture 9**

## **KMP, Greedy**

吳蔚琪 2022.12.9

# 991 《数据结构与算法》考纲

## 9. 算法基础

- (1) 字符串模式匹配算法
- (2) 贪心法、分治法、动态规划的基本概念
- (3) 计算复杂度类别的基本概念，NP-Complete问题

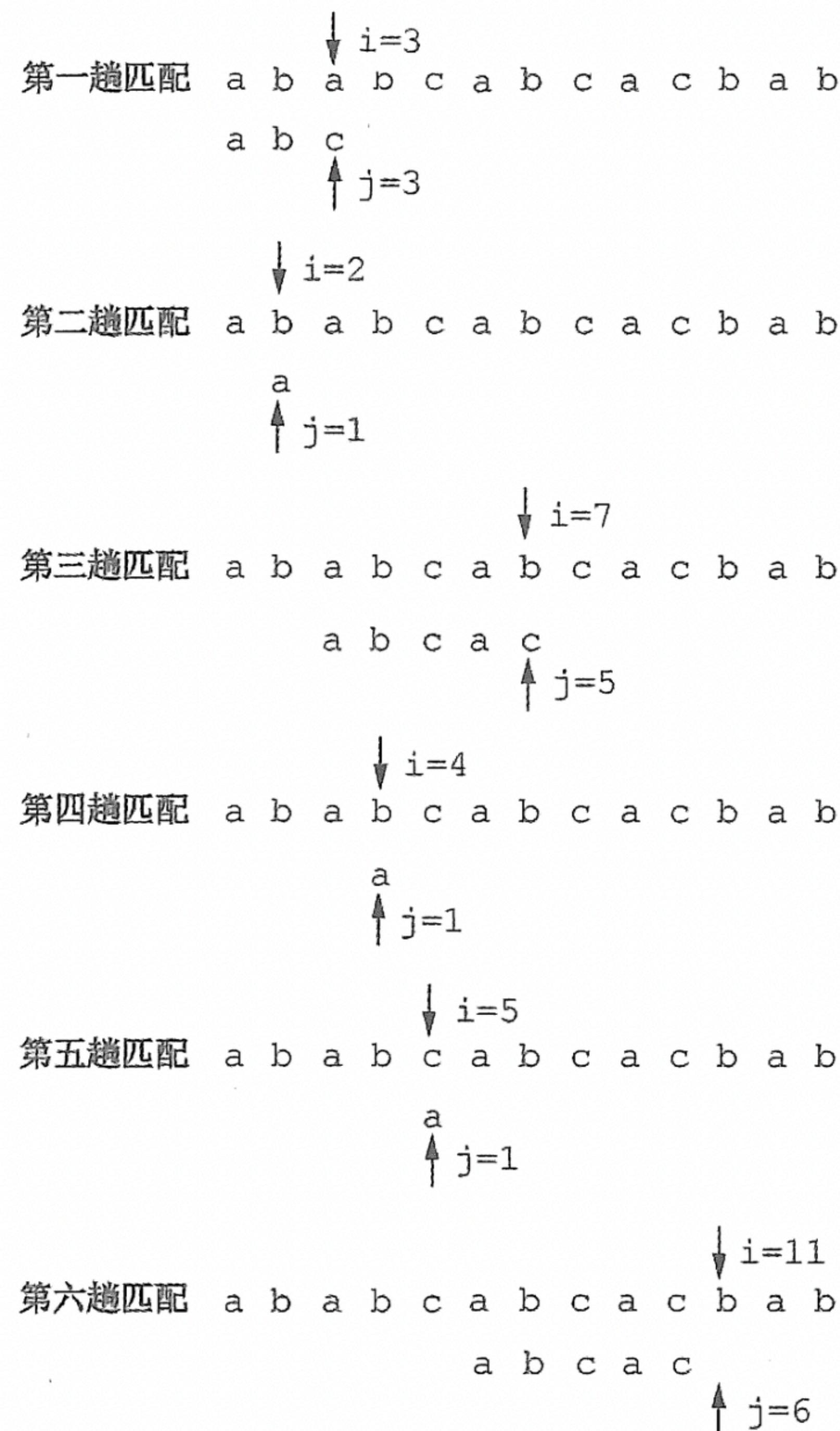
# Pattern Searching

# Brute Force

## 简单模式匹配算法

- 基本思想：
  - 从主串的第一个位置起和模式串的第一个字符开始比较
  - 如果相等，则继续逐一比较两个字符串的后续字符
  - 否则接着从主串的第二个字符开始，再重新用上一步方法与模式串的字符做比较
  - 依次类推，直到比较完模式串中的所有字符。
- 算法的时间复杂度：
  - 最坏情况 $O(mn)$ ，即前面的匹配每次都是最后一个字符不匹配，只在最后才匹配成功

模式串：abcac



# KMP(Knuth-Morris-Pratt)算法

## Motivation

主串	a	b	a	b	c	a	b	c	a	c	b	a	b
	a	b	c	a	c								
	a	b	c	a	c								
	a	b	c	a	c								
	a	b	c	a	c								

图 4.3 失配后移动情况

主串	a	b	a	b	c	a	b	c	a	c	b	a	b
	a	b	c	a	c								
	a	b	c	a	c								

图 4.4 直接移动到合适位置

# KMP算法

## 相关概念

- 字符串的前缀和后缀：
  - 如果有字符串A，存在 $A=B+S$ ，其中S是任意的非空字符串，称B为A的前缀；
  - 如果有字符串A，存在 $A=S+B$ ，称B为A的后缀。
  - 例如有字符串S="bsabs"
    - 其前缀包括{b, bs, bsa, bsab}
    - 其后缀包括{sabs, abs, bs, s}

# KMP算法

## 相关概念

- PMT (Partial Match Table) 部分匹配表
  - PM: 字符串的前缀和后缀集合中最长相等前后缀的长度
  - P="abcac"
    - j=1 'a' {} {} o
    - j=2 'ab' {a} {b} o
    - j=3 'abc' {a, ab} {bc, c} o
    - j=4 'abca' {a, ab, abc} {bca, ca, a} 1
    - j=5 'abcac' {a, ab, abc, abca} {bcac, cac, ac, c} o

编号	1	2	3	4	5
S	a	b	c	a	c
PM	0	0	0	1	0

# KMP算法

## 算法流程

主串	a	b	a	b	c	a	b	c	a	c	b	a	b
子串	a	b	c										

第一趟匹配过程：

发现 c 与 a 不匹配，前面的 2 个字符 'ab' 是匹配的，查表可知，最后一个匹配字符 b 对应的部分匹配值为 0，因此按照下面的公式算出子串需要向后移动的位数：

$$\text{移动位数} = \text{已匹配的字符数} - \text{对应的部分匹配值}$$

因为  $2 - 0 = 2$ ，所以将子串向后移动 2 位，如下进行第二趟匹配：

主串	a	b	a	b	c	a	b	c	a	c	b	a	b
子串			a	b	c	a	c						

第二趟匹配过程：

发现 c 与 b 不匹配，前面 4 个字符 'abca' 是匹配的，最后一个匹配字符 a 对应的部分匹配值为 1， $4 - 1 = 3$ ，将子串向后移动 3 位，如下进行第三趟匹配：

主串	a	b	a	b	c	a	b	c	a	c	b	a	b
子串				a	b	c	a	c					

第三趟匹配过程：

子串全部比较完成，匹配成功。整个匹配过程中，主串始终没有回退，故 KMP 算法可以在  $O(n + m)$  的时间数量级上完成串的模式匹配操作，大大提高了匹配效率。

# KMP算法

## Next数组

- 核心公式：移动位数 = 已匹配字符数 - 最后一个匹配位置的对应匹配值
  - 当前字符在模式串中的位置：j
  - 已匹配字符数/最后一个匹配位置：j - 1
  - $\text{Move} = (j - 1) - \text{PM}[j - 1]$
  - PM右移 -» next
    - $\text{Move} = (j - 1) - \text{next}[j]$
    - Why  $\text{next}[0]=-1$ ? 第一个元素匹配失败，子串右移1位

编号	1	2	3	4	5
S	a	b	c	a	c
next	-1	0	0	0	1

# KMP算法

## Next数组

- $\text{Move} = (j - 1) - \text{next}[j]$
- $j' = j - \text{Move} = j - (j - 1) - \text{next}[j] = \text{next}[j] + 1$

编号	1	2	3	4	5
S	a	b	c	a	c
next	-1	0	0	0	1

- Next数组每项+1
- $j' = \text{next}[j]$

编号	1	2	3	4	5
S	a	b	c	a	c
next	0	1	1	1	2

# KMP算法

## Next数组的一般公式?

主串	$s_1$	...	...	...	...	...	$s_{i-k+1}$	...	$s_{i-1}$	<u><math>s_i</math></u>	...	...	...	...	...	$s_n$
子串			$p_1$	...	$p_{k-1}$	...	$p_{j-k+1}$	...	$p_{j-1}$	<u><math>p_j</math></u>	...	$p_m$				
右移							$p_1$	...	$p_{k-1}$	$p_k$	...	...	$p_m$			

图 4.5 模式串右移到合适位置（阴影对齐部分表示上下字符相等）

$$\text{next}[j] = \begin{cases} 0, & j=1 \\ \max\{k \mid 1 < k < j \text{ 且 } 'p_1 \dots p_{k-1}' = 'p_{j-k+1} \dots p_{j-1}'\}, & \text{当此集合不空时} \\ 1, & \text{其他情况} \end{cases}$$

# KMP算法

## Next数组的代码实现

- $\text{next}[1] = 0$
- Assume  $\text{next}[j] = k$ ,  $\text{next}[j+1] = ?$

(1) 若  $p_k = p_j$ , 则表明在模式串中

$$'p_1 \dots p_{k-1} p_k' = 'p_{j-k+1} \dots p_{j-1} p_j'$$

并且不可能存在  $k' > k$  满足上述条件, 此时  $\text{next}[j+1] = k+1$ , 即

$$\text{next}[j+1] = \text{next}[j] + 1$$

(2) 若  $p_k \neq p_j$ , 则表明在模式串中

$$'p_1 \dots p_{k-1} p_k' \neq 'p_{j-k+1} \dots p_{j-1} p_j'$$

# KMP算法

## Next数组的代码实现

- $\text{next}[1] = 0$
- Assume  $\text{next}[j] = k$ ,  $\text{next}[j+1] = ?$ 
  - 新前缀和新后缀进行模版匹配

j	1	2	3	4	5	6	7	8	9
模式	a	b	a	a	b	c	a	b	a
next[j]	0	1	1	2	2	3	?	?	?

图 4.6 求模式串的 next 值

图 4.6 的模式串中已求得 6 个字符的 next 值, 现求  $\text{next}[7]$ , 因为  $\text{next}[6]=3$ , 又  $p_6 \neq p_3$ , 则需比较  $p_6$  和  $p_1$ (因  $\text{next}[3]=1$ ), 由于  $p_6 \neq p_1$ , 而  $\text{next}[1]=0$ , 所以  $\text{next}[7]=1$ ; 求  $\text{next}[8]$ , 因  $p_7=p_1$ , 则  $\text{next}[8]=\text{next}[7]+1=2$ ; 求  $\text{next}[9]$ , 因  $p_8=p_2$ , 则  $\text{next}[9]=3$ 。

# KMP算法

## Next数组的代码实现

```
void get_next(String T,int next[]){
    int i=1, j=0;
    next[1]=0;
    while(i<T.length){
        if(j==0 || T.ch[i]==T.ch[j]){
            ++i; ++j;
            next[i]=j; //若 pi=pj, 则 next[j+1]=next[j]+1
        }
        else
            j=next[j]; //否则令 j=next[j], 循环继续
    }
}
```

# KMP算法

## 算法流程

- 匹配时操作与暴力方法相同，都是接着对比下一字符；
- 如果在模式串第j个字符出现失配情况，保持主串*i*不变，模式串*j*=**next[j]**，然后接着比较；
- **next[j] = 0**: 已匹配相等序列中没有相等的前后缀，将子串首字符右滑动到主串*i*位置进行下一趟比较
- **next[j] != 0**: 已匹配相等序列中存在最大相等前后缀，将子串向右滑动到和该相等前后缀对齐，再从主串*i*位置开始进行下一趟比较

# KMP算法

## 伪代码

```
int Index_KMP(String S, String T, int next[]) {
    int i=1, j=1;
    while(i<=S.length&&j<=T.length) {
        if(j==0||S.ch[i]==T.ch[j]) {
            ++i; ++j; //继续比较后继字符
        }
        else
            j=next[j]; //模式串向右移动
    }
    if(j>T.length)
        return i-T.length; //匹配成功
    else
        return 0;
}
```

# KMP算法

## 进一步优化\*

主串	a	a	a	b	a	a	a	a	b
模式	a	a	a	a	b				
j	1	2	3	4	5				
next[j]	0	1	2	3	4				
nextval[j]	0	0	0	0	4				

图 4.7 KMP 算法进一步优化示例

当  $i=4$ 、 $j=4$  时， $s_4$  跟  $p_4$  ( $b \neq a$ ) 失配，如果用之前的  $next$  数组还需要进行  $s_4$  与  $p_3$ 、 $s_4$  与  $p_2$ 、 $s_4$  与  $p_1$  这 3 次比较。事实上，因为  $p_{next[4]}=3=p_4=a$ 、 $p_{next[3]}=2=p_3=a$ 、 $p_{next[2]}=1=p_2=a$ ，显然后面 3 次用一个和  $p_4$  相同的字符跟  $s_4$  比较毫无意义，必然失配。那么问题出在哪里呢？

问题在于不应该出现  $p_j=p_{next[j]}$ 。理由是：当  $p_j \neq s_j$  时，下次匹配必然是  $p_{next[j]}$  跟  $s_j$  比较，如果  $p_j=p_{next[j]}$ ，那么相当于拿一个和  $p_j$  相等的字符跟  $s_j$  比较，这必然导致继续失配，这样的比较毫无意义。那么如果出现了  $p_j=p_{next[j]}$  应该如何处理呢？

如果出现了，则需要再次递归，将  $next[j]$  修正为  $next[next[j]]$ ，直至两者不相等为止，更新后的数组命名为  $nextval$ 。计算  $next$  数组修正值的算法如下，此时匹配算法不变。

# KMP算法

## Summary

- Brute Force:  $O(mn)$
- KMP:  $O(m+n)$ 
  - 主串不回溯
- However...
- 一般情况下，BF实际执行时间近似为 $O(m+n)$ ，因此KMP算法只在主串与模式串存在很多“部分匹配”时才显得比普通算法快得多

# KMP算法

## One Last Example

- $S = \text{'aababaabaac'}$
- $P = \text{'aabaaac'}$
- PMT
  - $j=1$  'a'  $\{\}$   $\{\}$  0
  - $j=2$  'aa'  $\{a\}$   $\{a\}$  1
  - $j=3$  'aab'  $\{a, aa\}$   $\{ab, b\}$  0
  - $j=4$  'aaba'  $\{a, aa, aab\}$   $\{aba, ba, a\}$  1
  - $j=5$  'aabaa'  $\{a, aa, aab, aaba\}$   $\{abaa, baa, aa, a\}$  2
  - $j=6$  'aabaaac'  $\{a, aa, aab, aaba, abaaa\}$   $\{abaac, baac, aac, ac, c\}$  0

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1	2	1	2	3

# KMP算法

## One Last Example

- $S = \text{'aababaabaac'}$
- $P = \text{'aabaac'}$

编号	1	2	3	4	5	6
S	a	a	b	a	a	c
next	0	1	2	1	2	3

2) 利用 KMP 算法的匹配过程如下。

第一趟：从主串和模式串的第一个字符开始比较，失配时  $i=6$ 、 $j=6$ 。

主串	a	a	b	a	a	b	a	a	b	a	a	c
	a	a	b	a	a	c						

第二趟： $\text{next}[6]=3$ ，主串当前位置和模式串的第 3 个字符继续比较，失配时  $i=9$ 、 $j=6$ 。

主串	a	a	b	a	a	b	a	a	b	a	a	c
	a	a	b	a	a	c						

第三趟： $\text{next}[6]=3$ ，主串当前位置和模式串的第 3 个字符继续比较，匹配成功。

主串	a	a	b	a	a	b	a	a	b	a	a	c
	a	a	b	a	a	c						

Greedy

# Greedy Algorithm

---

- A greedy algorithm is an algorithm that constructs an object X one step at a time, at each step choosing the **locally best** option.
- In most of cases, greedy algorithms construct the **globally best** object by repeatedly choosing the locally best option.
- **How to define best?**
- The example you have learned:
  - Coin changing
  - Interval scheduling
  - Scheduling to minimizing lateness
  - Huffman coding
  - Dijkstra's Algorithm (Shortest Path)
  - Prim's, Kruskal's Algorithm (Minimum Spanning Tree)

# Greedy Algorithm

---

- The key point:
  - The keyword: “Greedy Algorithm”, “best”, “sort”,, .....
  - Analyze time complexity by pseudocode.
- How to prove correctness? Hard!
  - Greedy algorithms are often used to solve optimization problems: you want to maximize or minimize some quantity subject to a set of constraints.
  - When you are trying to write a proof that shows that a greedy algorithm is correct, you often need to show two different results.
  - First, you need to show that your algorithm produces a **feasible** solution, a solution to the problem that obeys the constraints.
  - Next, you need to show that your algorithm produces an **optimal** solution, a solution that maximizes or minimizes the appropriate quantity.

# Greedy Algorithm

---

- How to write the format of Greedy Algorithm? “Greedy stays ahead”
- **Define Your Solution.** Your algorithm will produce some object  $X$  and you will probably compare it against some optimal solution  $X^*$ . Introduce some variables denoting your algorithm's solution and the optimal solution.
- **Define Your Measure.** Your goal is to find a series of measurements you can make of your solution and the optimal solution. Define some series of measures  $m_1(X), m_2(X), \dots, m_n(X)$  such that  $m_1(X^*), m_2(X^*), \dots, m_k(X^*)$  is also defined for some choices of  $m$  and  $n$ . Note that there might be a different number of measures for  $X$  and  $X^*$ , since you can't assume at this point that  $X$  is optimal.
- **Prove Greedy Stays Ahead.** Prove that  $m_i(X) \geq m_i(X^*)$  or that  $m_i(X) \leq m_i(X^*)$ , whichever is appropriate, for all reasonable values of  $i$ . This argument is usually done **inductively**.
- **Prove Optimality.** Using the fact that greedy stays ahead, prove that the greedy algorithm must produce an optimal solution. This argument is often done by **contradiction** by assuming the greedy solution isn't optimal and using the fact that greedy stays ahead to derive a contradiction.

# Greedy-Interval Problems

## Contents

- Interval Merge
- Interval Scheduling
- Interval Selection
- Interval Covering
- Interval Partition

# Interval Merge

## [LeetCode 56] Merge-Intervals

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]`。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

示例 1：

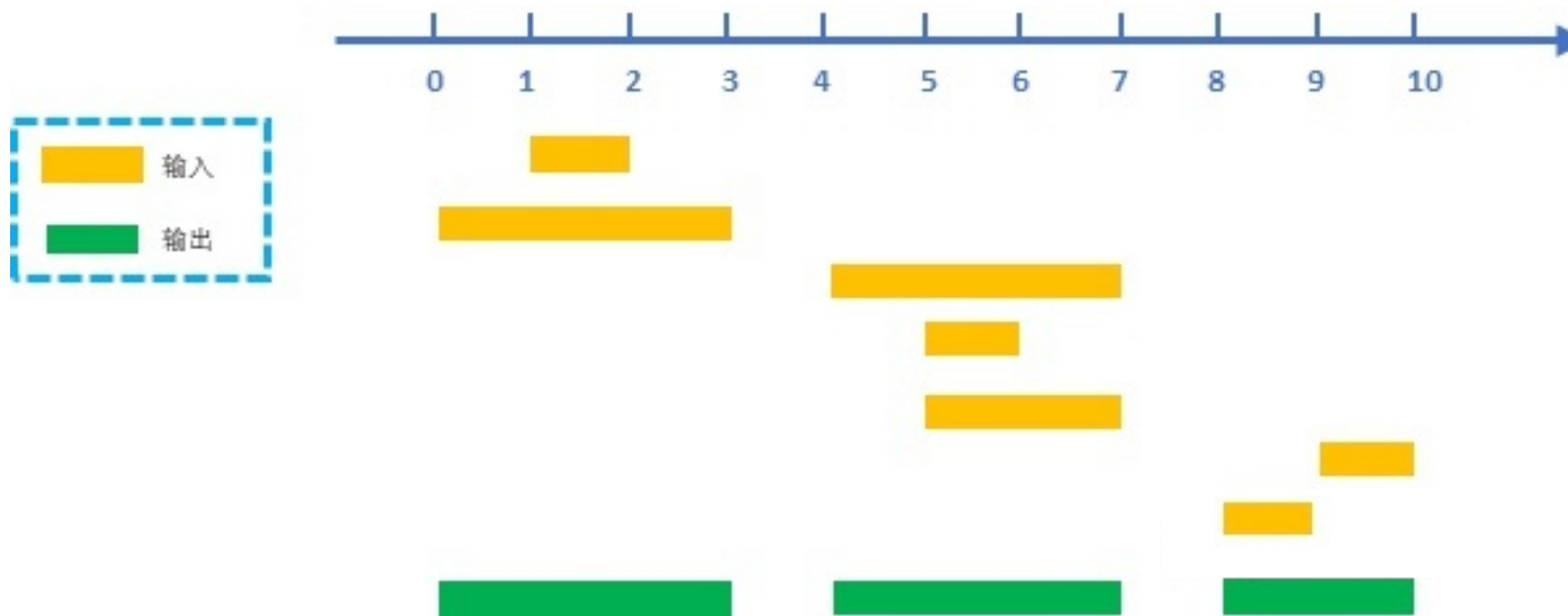
```
输入: intervals = [[1,3],[2,6],[8,10],[15,18]]  
输出: [[1,6],[8,10],[15,18]]  
解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

示例 2：

```
输入: intervals = [[1,4],[4,5]]  
输出: [[1,5]]  
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

# Interval Merge

[LeetCode 56] Merge-Intervals



# Interval Merge

[LeetCode 56] Merge-Intervals

Python3 | C++ | Java

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.size() == 0) {
            return {};
        }
        sort(intervals.begin(), intervals.end());
        vector<vector<int>> merged;
        for (int i = 0; i < intervals.size(); ++i) {
            int L = intervals[i][0], R = intervals[i][1];
            if (!merged.size() || merged.back()[1] < L) {
                merged.push_back({L, R});
            }
            else {
                merged.back()[1] = max(merged.back()[1], R);
            }
        }
        return merged;
};
```

# Interval Scheduling

## [LeetCode 435] Non-overlapping-Intervals

给定一个区间的集合 `intervals`，其中 `intervals[i] = [starti, endi]`。返回 需要移除区间的最小数量，使剩余区间互不重叠。

示例 1:

输入: `intervals = [[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。

示例 2:

输入: `intervals = [[1,2], [1,2], [1,2]]`

输出: 2

解释: 你需要移除两个 `[1,2]` 来使剩下的区间没有重叠。

示例 3:

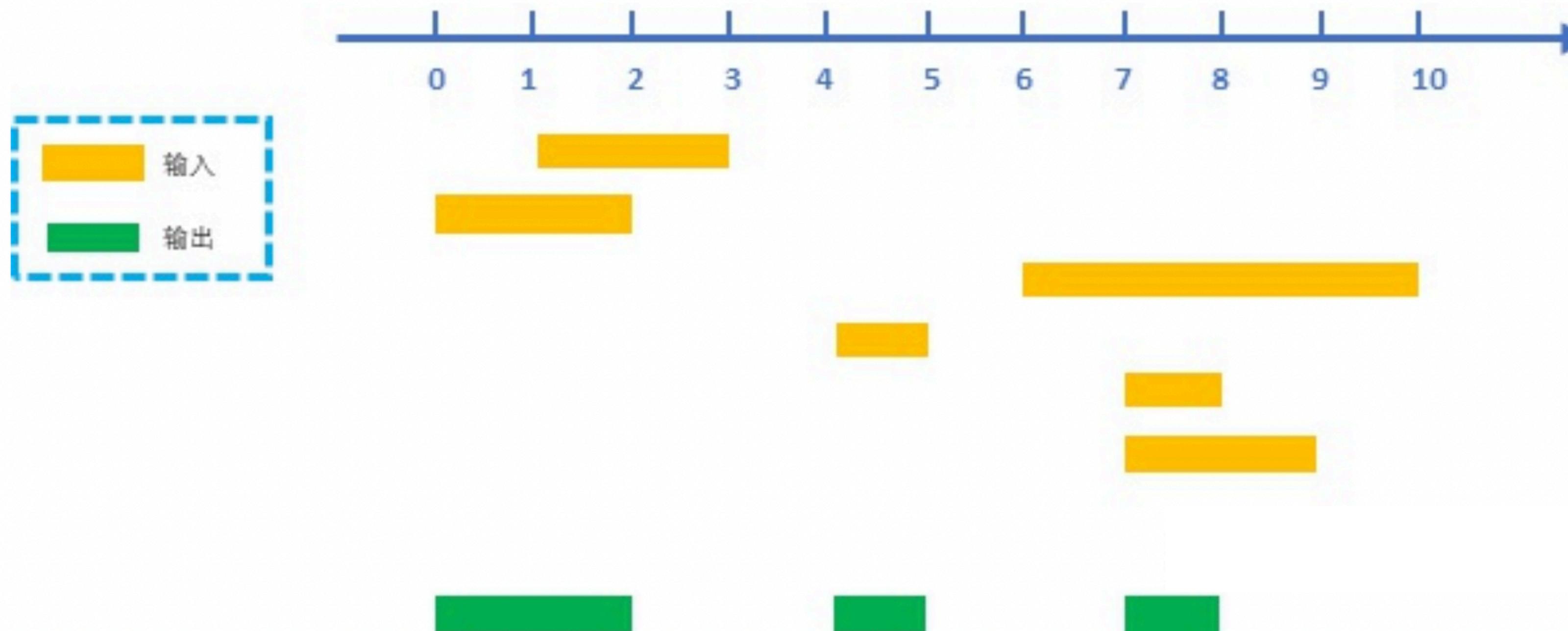
输入: `intervals = [[1,2], [2,3]]`

输出: 0

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

# Interval Scheduling

[LeetCode 435] Non-overlapping-Intervals



# Interval Scheduling

## [LeetCode 435] Non-overlapping-Intervals

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty()) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), [](const auto& u, const auto& v) {
            return u[1] < v[1];
        });

        int n = intervals.size();
        int right = intervals[0][1];
        int ans = 1;
        for (int i = 1; i < n; ++i) {
            if (intervals[i][0] >= right) {
                ++ans;
                right = intervals[i][1];
            }
        }
        return n - ans;
    }
};
```

# Interval Selection

## [LeetCode 452] minimum-number-of-arrows-to-burst-balloons

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points`，其中 `points[i] = [xstart, xend]` 表示水平直径在 `xstart` 和 `xend` 之间的气球。你不知道气球的确切 y 坐标。

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。

给你一个数组 `points`，返回引爆所有气球所必须射出的**最小** 弓箭数。

示例 1：

输入: `points = [[10,16],[2,8],[1,6],[7,12]]`

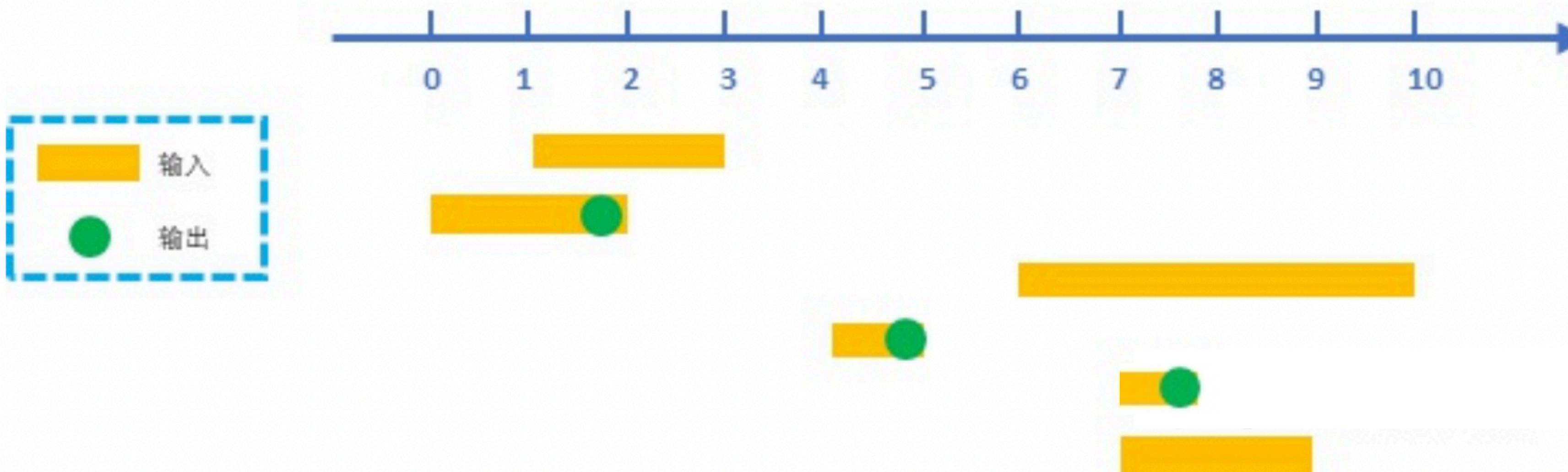
输出: 2

解释: 气球可以用2支箭来爆破:

- 在 `x = 6` 处射出箭，击破气球 `[2,8]` 和 `[1,6]`。
- 在 `x = 11` 处发射箭，击破气球 `[10,16]` 和 `[7,12]`。

# Interval Selection

[LeetCode 452] minimum-number-of-arrows-to-burst-balloons



# Interval Selection

[LeetCode 452] minimum-number-of-arrows-to-burst-balloons

```
class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.empty()) {
            return 0;
        }
        sort(points.begin(), points.end(), [] (const vector<int>& u, const vector<int>& v) {
            return u[1] < v[1];
        });
        int pos = points[0][1];
        int ans = 1;
        for (const vector<int>& balloon: points) {
            if (balloon[0] > pos) {
                pos = balloon[1];
                ++ans;
            }
        }
        return ans;
    }
};
```

# Interval Covering

## [LeetCode 1024] video-stitching

你将会获得一系列视频片段，这些片段来自于一项持续时长为 `time` 秒的体育赛事。这些片段可能有所重叠，也可能长度不一。

使用数组 `clips` 描述所有的视频片段，其中 `clips[i] = [starti, endi]` 表示：某个视频片段开始于 `starti` 并于 `endi` 结束。

甚至可以对这些片段自由地再剪辑：

- 例如，片段 `[0, 7]` 可以剪切成 `[0, 1] + [1, 3] + [3, 7]` 三部分。

我们需要将这些片段进行再剪辑，并将剪辑后的内容拼接成覆盖整个运动过程的片段 (`[0, time]`)。返回所需片段的最小数目，如果无法完成该任务，则返回 `-1`。

示例 1：

输入: `clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]]`, `time = 10`

输出: 3

解释:

选中 `[0,2]`, `[8,10]`, `[1,9]` 这三个片段。

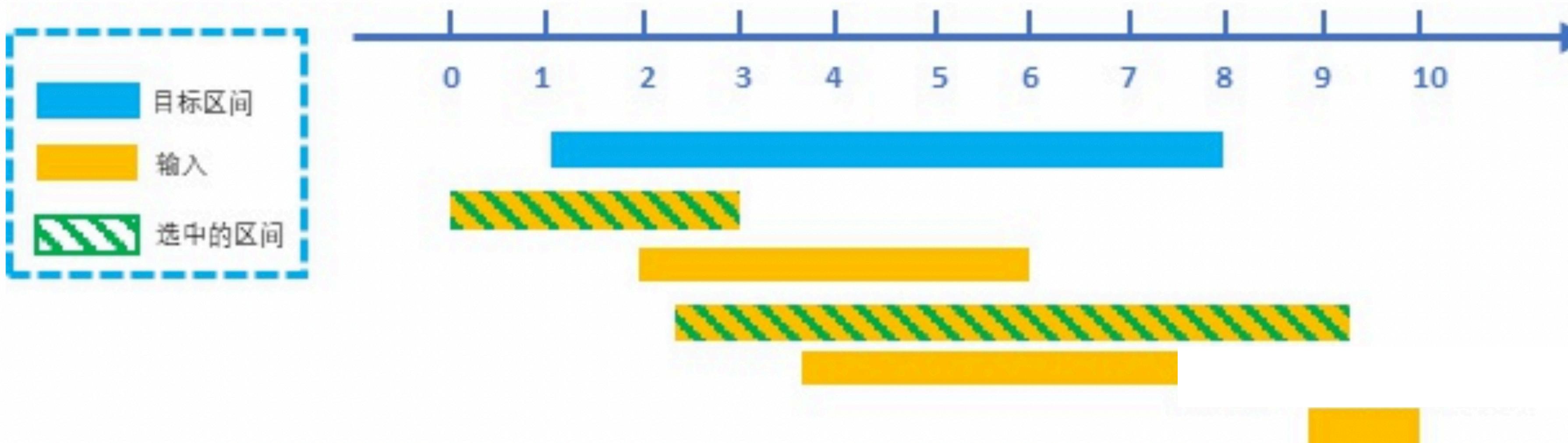
然后，按下面的方案重制比赛片段：

将 `[1,9]` 再剪辑为 `[1,2] + [2,8] + [8,9]`。

现在手上的片段为 `[0,2] + [2,8] + [8,10]`，而这些覆盖了整场比赛 `[0, 10]`。

# Interval Covering

[LeetCode 1024] video-stitching



# Interval Covering

## [LeetCode 1024] video-stitching

```
class Solution {
public:
    int videoStitching(vector<vector<int>>& clips, int time) {
        vector<int> maxn(time);
        int last = 0, ret = 0, pre = 0;
        for (vector<int>& it : clips) {
            if (it[0] < time) {
                maxn[it[0]] = max(maxn[it[0]], it[1]);
            }
        }
        for (int i = 0; i < time; i++) {
            last = max(last, maxn[i]);
            if (i == last) {
                return -1;
            }
            if (i == pre) {
                ret++;
                pre = last;
            }
        }
        return ret;
    }
};
```

# Interval Partition

[LeetCode 253] meeting-rooms

描述

中文

给定一系列的会议时间间隔intervals，包括起始和结束时间  $[[s_1, e_1], [s_2, e_2], \dots]$  ( $s_i < e_i$ )，找到所需的最小的会议室数量。

您在真实的面试中是否遇到过这个题?

是

题目

样例

样例1

输入： intervals = [(0,30),(5,10),(15,20)]

输出： 2

解释：

需要两个会议室

会议室1:(0,30)

会议室2:(5,10),(15,20)

样例2

输入： intervals = [(2,7)]

输出： 1

解释：

只需要1个会议室就够了

[https://blog.csdn.net/weixin\\_44413191](https://blog.csdn.net/weixin_44413191)

# Interval Partition

[LeetCode 253] meeting-rooms



# Interval Partition

[LeetCode 253] meeting-rooms

```
class Solution {
public:
    int minMeetingRooms(vector<vector<int>>& intervals) {
        if(intervals.empty()) return 0;
        sort(intervals.begin(), intervals.end(), [&](auto a, auto b){
            if(a[0] == b[0])
                return a[1] < b[1];//开始时间一样, 先结束的在前
            return a[0] < b[0];//开始早的在前
        });
        priority_queue<int, vector<int>, greater<int>> q;//小顶堆, 存放会议室结束时间, 小的在上
        q.push(intervals[0][1]);
        for(int i = 1; i < intervals.size(); ++i)
        {
            if(intervals[i][0] >= q.top())//最早结束的会议室可用, 占用它
            {
                q.pop();
            }
            q.push(intervals[i][1]);
        }
        return q.size();
    }
};
```

# Greedy-Interval Problems

## Summary

题型	简述	排序端点	排序顺序
区间合并	合并所有有交集的区间	左端点	从小到大
选择不相交区间	最多选多少互不相交的区间	右端点	从小到大
区间选点	最少选多少点可以覆盖所有区间	右端点	从小到大
区间覆盖	最少选多少区间可以覆盖目标区间	左端点	从小到大
区间分组	最少分多少组可以使每组内区间互不相交	左端点	从小到大