

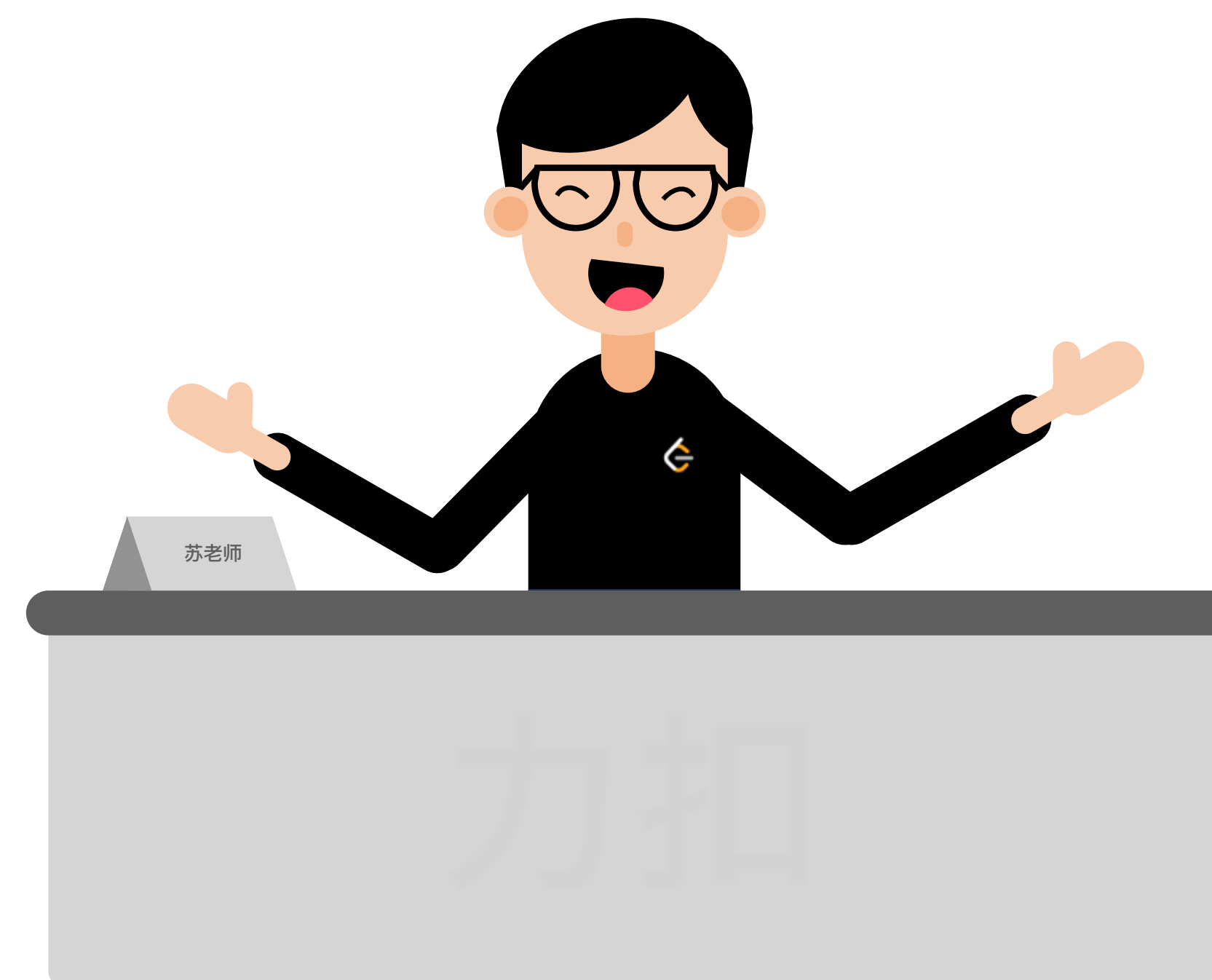
第九课

剖析大厂算法面试真题 - 高频题精讲（二）

大厂面试高频题精讲（二）

- 合并区间+无重叠区间
- 火星字典
- 基本计算器

拉勾



56. 合并区间

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

输入: `[[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: `[[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。

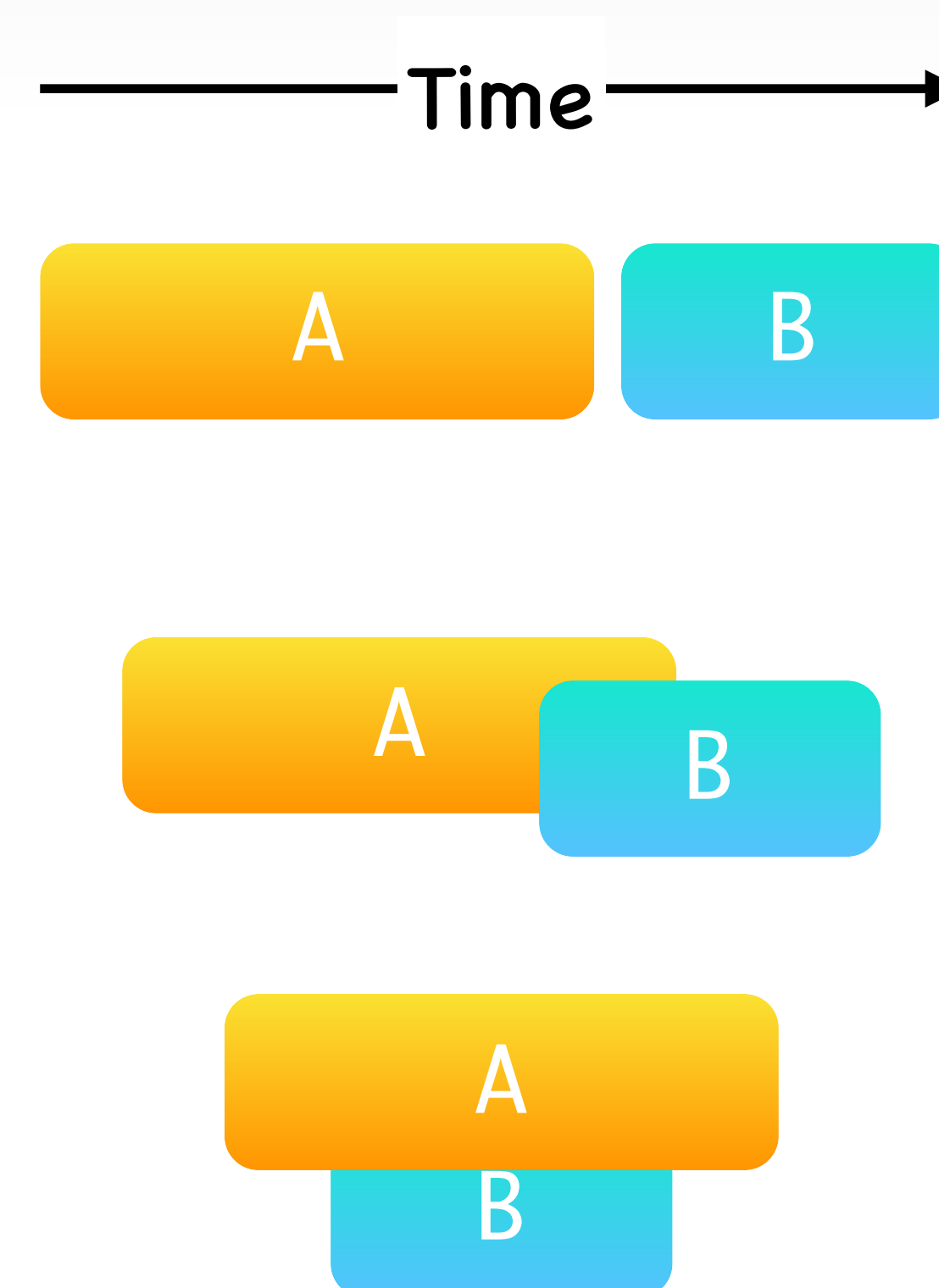
56. 合并区间

解题思路

假设有区间 A 和 B，区间 A 的起始时间要早于 B 的起始时间。

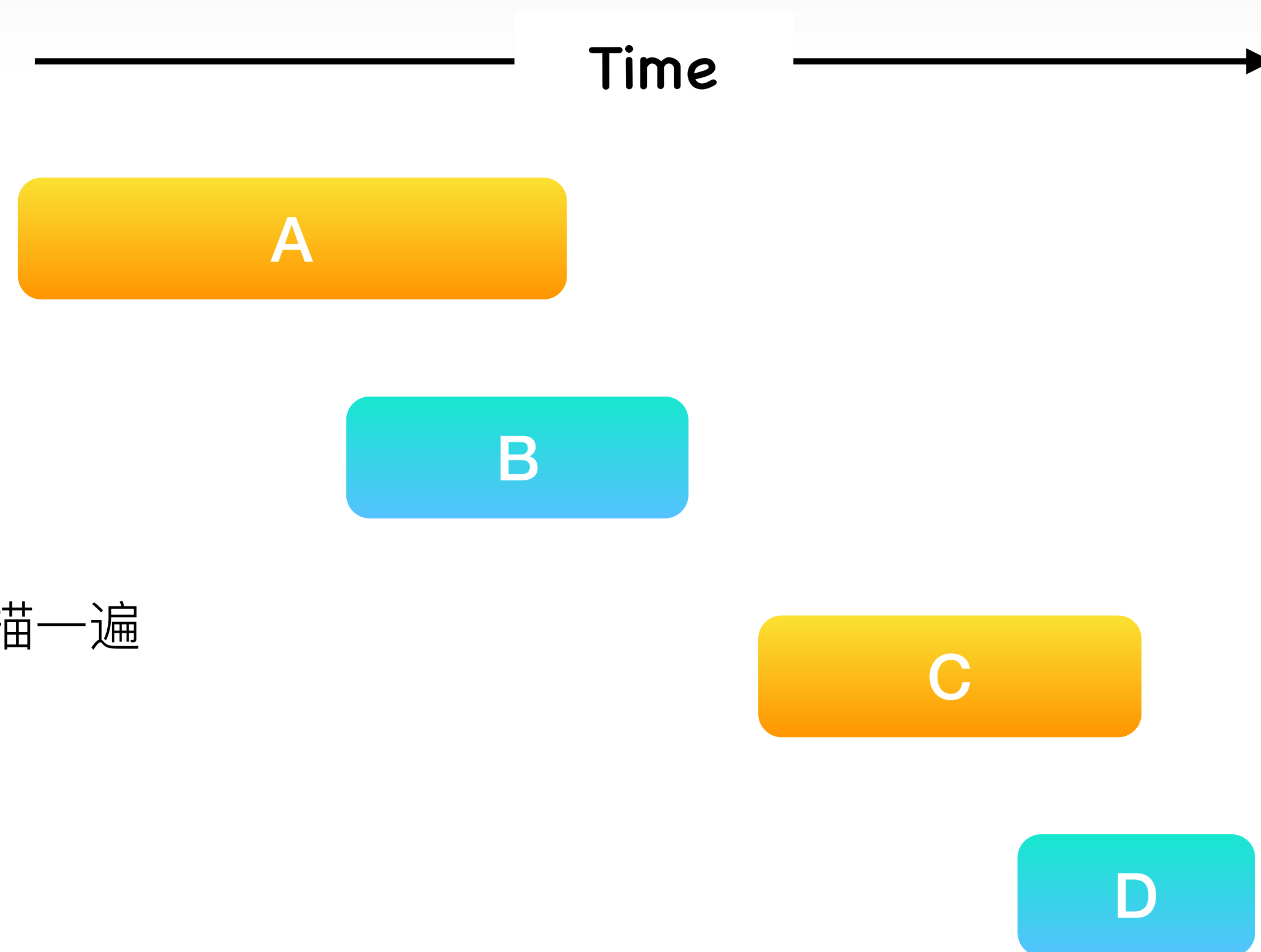
那么它们之间有 3 种可能的重叠关系：

- ▶ 情况一：A 和 B 没有任何重叠部分，不会发生融合
- ▶ 情况二和三：区间有重叠
 - 新区间的起始时间是 A 的起始时间
 - 新区间的终止时间是 A 和 B 终止时间中的最大值
 - 这个就是融合两个区间的最基本的思想



56. 合并区间

给定了 n 个区间，我们应该怎么去有效地融合它们呢？

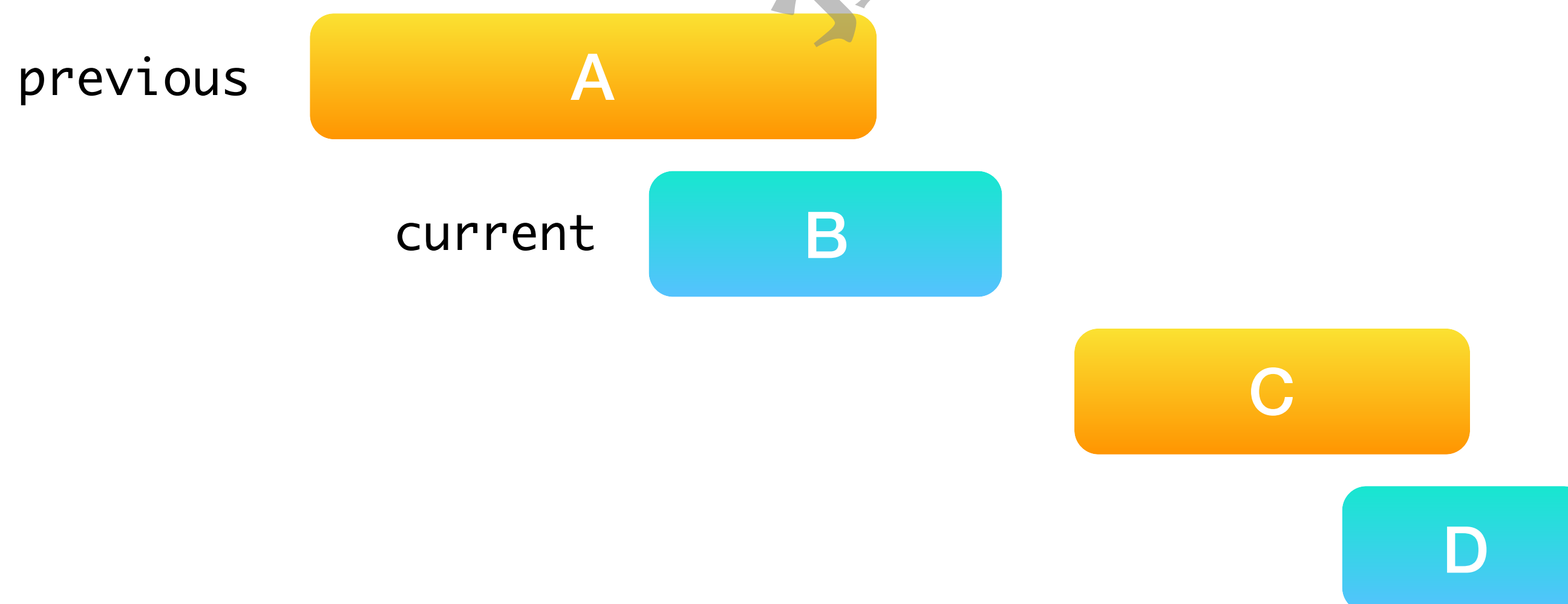


直观有效的做法

- ▶ 先将所有的区间按照起始时间的先后顺序排序，从头到尾扫描一遍

56. 合并区间

- ▶ 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
 - 如果没有融合，当前区间就变成新的 `previous`，下一个区间成为新的 `current`
 - 如果发生了融合，更新前一个区间的结束时间



56. 合并区间

- ▶ 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
 - 如果没有融合，当前区间就变成新的 `previous`，下一个区间成为新的 `current`
 - 如果发生了融合，更新前一个区间的结束时间

`previous`

A'

`current`

C

D

56. 合并区间

- ▶ 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
 - 如果没有融合，当前区间就变成新的 `previous`，下一个区间成为新的 `current`
 - 如果发生了融合，更新前一个区间的结束时间

A'

previous

C

current

D

56. 合并区间

- ▶ 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
 - 如果没有融合，当前区间就变成新的 `previous`，下一个区间成为新的 `current`
 - 如果发生了融合，更新前一个区间的结束时间

A'

C'

```
int[][] merge(int[][] intervals) {  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0],  
i2[0]));
```

```
    int[] previous = null;  
    List<int[]> result = new ArrayList<>();
```

```
    for (int[] current : intervals) {  
        if (previous == null || current[0] > previous[1]) {  
            result.add(previous = current);  
        } else {  
            prev[1] = Math.max(previous[1], current[1]);  
        }  
    }
```

```
    return result.toArray(new int[result.size()][]);  
}
```

- 将所有区间按照起始时间的先后顺序排序
- 定义一个 previous 变量，初始化为 null
- 定义一个 result 变量，用来保存最终的区间结果
- 从头开始遍历给定的所有区间
- 如果这是第一个区间，或者当前区间和前一个区间没有重叠，那么将当前区间加入到结果中
- 否则，两个区间发生了重叠，更新前一个区间的结束时间
- 最后返回结果

56. 合并区间

时间复杂度 $O(n\log(n))$

- ▶ 因为我们一开始要对数组进行排序

空间复杂度为 $O(n)$

- ▶ 因为我们用了一个的 `result` 数组来保存结果

435. 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。

示例 1:

输入: `[[1,2], [2,3], [3,4], [1,3]]`

输出: 1

解释: 移除 `[1,3]` 后，剩余区间无重叠。

示例 2:

输入: `[[1,2], [1,2], [1,2]]`

输出: 2

解释: 你需要移除两个 `[1,2]` 来使剩下的区间无重叠。

示例 3:

输入: `[[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。

435. 无重叠区间

暴力法

▶ 方法一：

- 将各个区间按照起始时间的先后顺序排序
- 找出所有组合
- 分别判断每种组合各个区间有没有互相重叠

▶ 时间复杂度：

排序需要 $O(n\log(n))$ 的时间复杂度

n 个区间中取 1 个，有 C_n^1 种取法

n 个区间中取 2 个，有 C_n^2 种取法

...

n 个区间中取 n 个，有 C_n^n 种取法，也就是 1 种

计入空集合，则共有

$$C_n^0 + C_n^1 + C_n^2 + \dots + C_n^n = 2^n \text{ 种取法}$$

435. 无重叠区间

暴力法

▸ 方法一：

- 分别判断每种组合各个区间有没有互相重叠

▸ 时间复杂度：

判断选出的 k 个区间是否有重叠，需要 $O(k)$ 的时间复杂度

因此暴力解法总体时间复杂度为

$$C_n^0 \times 0 + C_n^1 \times 1 + C_n^2 \times 2 + \dots + C_n^k \times k + \dots + C_n^n \times n$$

$$= n \times 2^{n-1} > n \log(n)$$

$$\approx O(n \times 2^n)$$

435. 无重叠区间

暴力法

▸ 方法二：

- 定义两个变量 `previous` 和 `current`，分别表示前一个区间和当前的区间
 - 如果当前区间和前一个区间没有重叠，则尝试保留当前区间，表明此处不需要删除操作
 - 只有在题目要求最少删除个数的情况下，才不需要做任何删除操作
 - 同时，这种情况下，虽然两个区间没有重叠，也需考虑尝试删除当前区间的情况
 - 然后对比得出需要删除的区间最少的情况

435. 无重叠区间

暴力法

▸ 方法二：



previous



current

435. 无重叠区间

暴力法

▸ 方法二：



previous



current

435. 无重叠区间

暴力法

▸ 方法二：



previous



current

```
int eraseOverlapIntervals(int[][] intervals) {  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0], i2[0]));  
    return eraseOverlapIntervals(-1, 0, intervals);  
}
```

```
int eraseOverlapIntervals(int prev, int curr, int[][] intervals) {  
    if (curr == intervals.length) {  
        return 0;  
    }
```

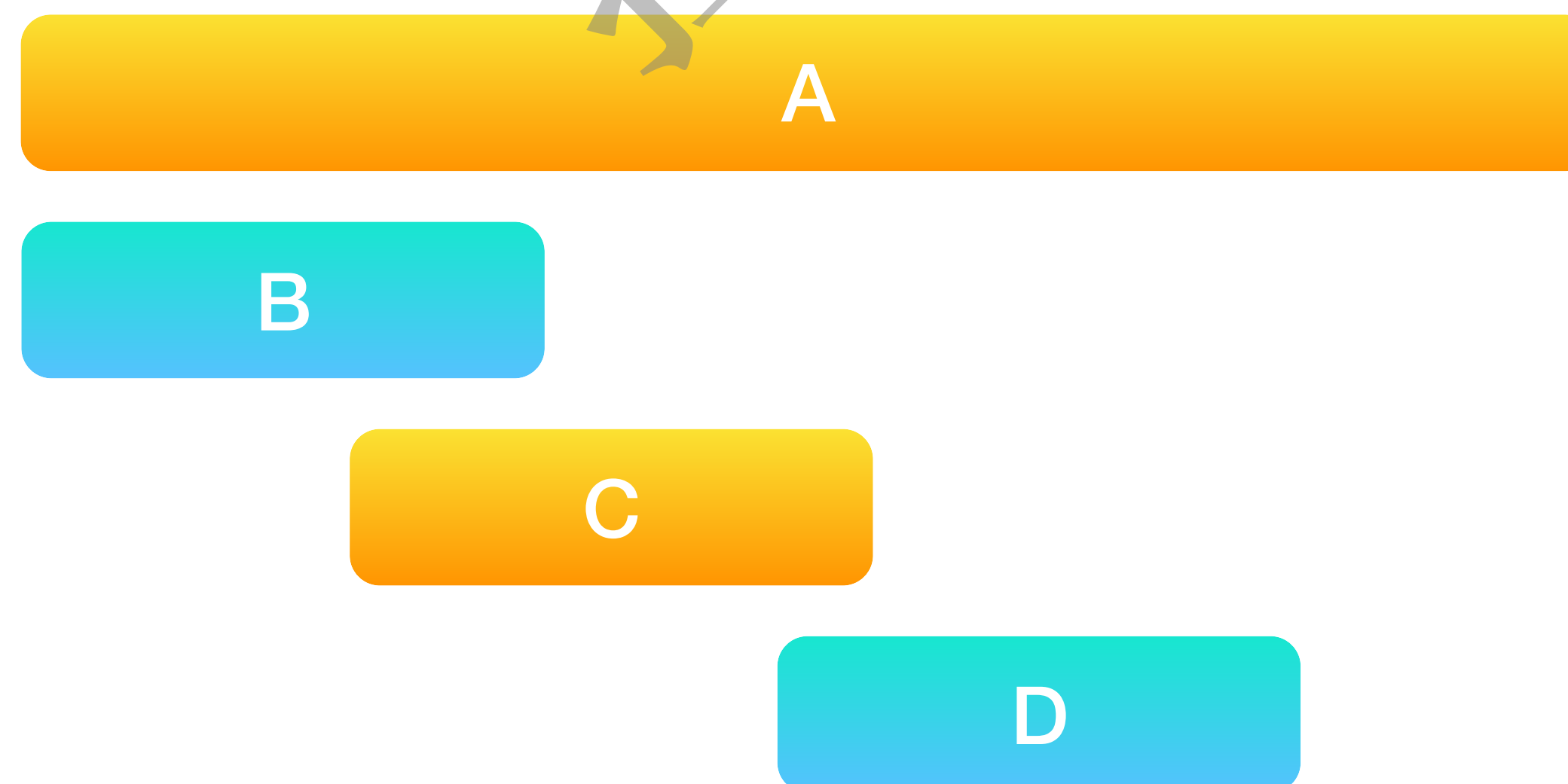
```
    int taken = Integer.MAX_VALUE, nottaken;  
  
    if (prev == -1 || intervals[prev][1] <= intervals[curr][0]) {  
        // 只有当prev, curr没有发生重叠的时候，才可以选择保留当前的区间  
        taken = eraseOverlapIntervals(curr, curr + 1, intervals);  
    }  
  
    // 其他情况，可以考虑删除掉curr区间，看看删除了它之后会不会产生最好的结果  
    nottaken = eraseOverlapIntervals(prev, curr + 1, intervals) + 1;  
  
    return Math.min(taken, nottaken);  
}
```

- 主体函数中，先将区间按照起始时间的先后顺序排序然后调用递归函数
- 递归函数中，先检查所有区间是否已处理完是的话，表明不需要删除操作，直接返回
- 定义 taken 与 nottaken 变量，用来记录：
 - 如果保留当前区间的话，最少需要删除多少其他区间
 - 如果删除当前取件的话，最少需要删除多少区间
- 最后返回两种情况下的最小值，判断是否删除当前区间

435. 无重叠区间

贪婪法

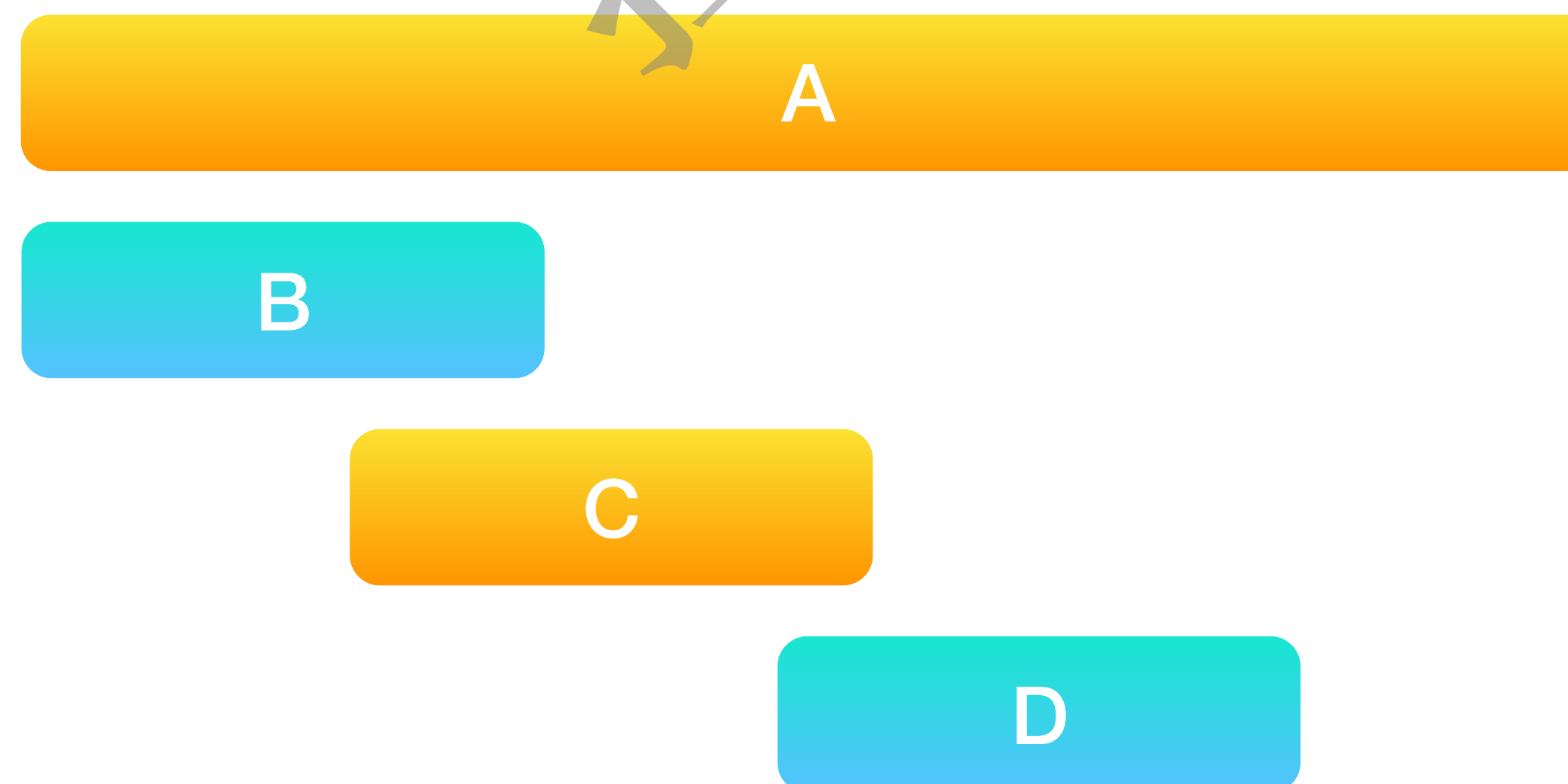
▸ 方法一：按照起始时间排序



435. 无重叠区间

贪婪法

▸ 方法一：按照起始时间排序



```
int eraseOverlapIntervals(int[][] intervals) {  
    if (intervals.length == 0) return 0;  
  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[0],  
i2[0]));  
  
    int end = intervals[0][1], count = 0;  
  
    for (int i = 1; i < intervals.length; i++) {  
        if (intervals[i][0] < end) {  
            end = Math.min(end, intervals[i][1]);  
            count++;  
        } else {  
            end = intervals[i][1];  
        }  
    }  
  
    return count;  
}
```

- 将所有区间按照起始时间的先后顺序排序
- 定义一个 end 变量记录当前的最小结束时间点，定义一个 count 变量记录到目前为止删除了多少区间
- 从第二个区间开始，判断一下当前区间和前一个区间的结束时间
- 如果发现当前区间与前一个区间有重叠，即当前区间的起始时间小于上一个区间的结束时间，则 end 变量记录下两个结束时间的最小值，意味着把结束时间晚的区间删除，计数加一
- 如果没有发现重叠，根据贪心法，更新 end 变量为当前区间的结束时间

435. 无重叠区间

贪婪法

▶ 方法二：按照结束时间排序

- 在给定的区间中，最多有多少个区间相互之间没有重叠的？
- 一旦求出最多有 m 个区间，则最少需要将 $n - m$ 个区间删除

B

C

D

A

```
int eraseOverlapIntervals(int[][] intervals) {  
    if (intervals.length == 0) return 0;  
  
    Arrays.sort(intervals, (i1, i2) -> Integer.compare(i1[1],  
i2[1]));  
  
    int end = intervals[0][1], count = 1;  
  
    for (int i = 1; i < intervals.length; i++) {  
        if (intervals[i][0] >= end) {  
            end = intervals[i][1];  
            count++;  
        }  
    }  
  
    return intervals.length - count;  
}
```

- 将所有区间按照起始时间的先后顺序排序
- 定义一个 end 变量记录当前的结束时间，
定义一个 count 变量记录有多少个没有重叠的区间
- 从第二个区间开始遍历剩下的区间
- 如果发现当前区间与前一个区间结束时间没有重叠，
则计数加一，同时更新一下新的结束时间
- 最后，用总区间的个数减去没有重叠的区间个数，
得到最少要删除的区间个数

269. 火星字典

现有一种使用字母的全新语言，这门语言的字母顺序与英语顺序不同。

假设，您并不知道其中字母之间的先后顺序。但是，会收到词典中获得一个不为空的单词列表。因为是从词典中获得的，所以该单词列表内的单词已经按这门新语言的字母顺序进行了排序。

您需要根据这个输入的列表，还原出此语言中已知的字母顺序。

示例 1:

```
输入:
[
  "wrt",
  "wrf",
  "er",
  "ett",
  "rftt"
]
输出: "wertf"
```

示例 2:

```
输入:
[
  "z",
  "x",
  "z"
]
输出: ""
解释: 此顺序是非法的，因此返回 ""。
```

示例 3:

```
输入:
[
  "z",
  "x"
]
输出: "zx"
```

269. 火星字典

bar

bat

algorithm

cook

cog

- 逐位地比较两个相互挨着的字符串
- 第一个字母出现的顺序越早，排位越靠前
- 当第一个字母相同时，则比较第二个字母，以此类推
- 注意：当两个字符串某个相同的位置出现了不同，即得出它们的顺序，无需继续比较字符串剩余字母

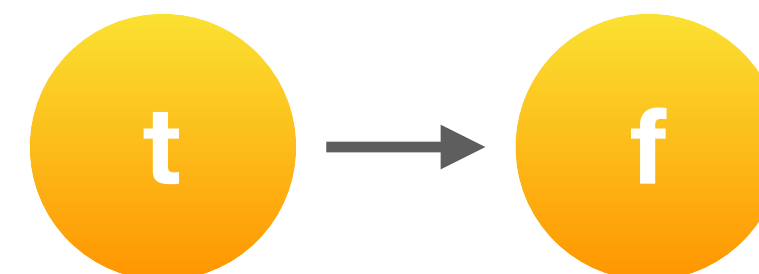
269. 火星字典

示例 1:

```
输入:
wrt
wrfwrt",
er"wrf",
et"er",
rftt"tt",
    "rftt"
]
输出: "wertf"
```

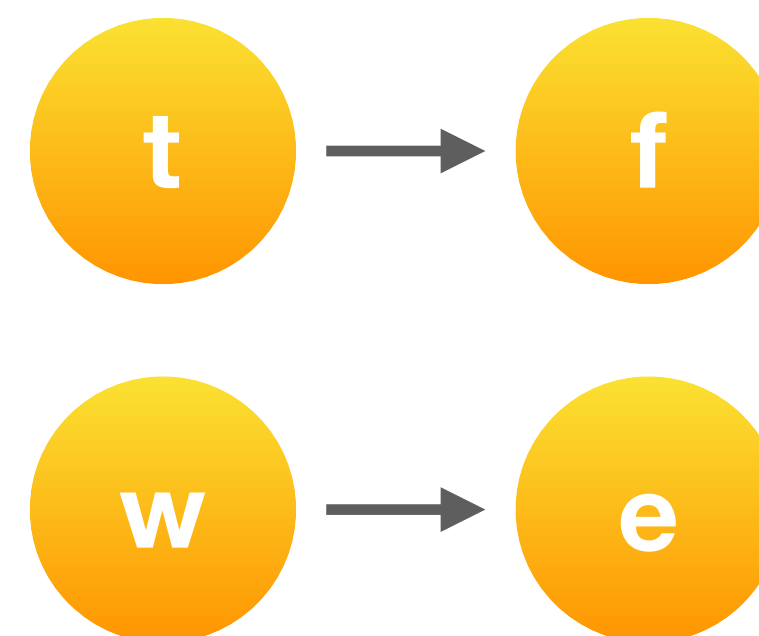
269. 火星字典

wrt
wrf



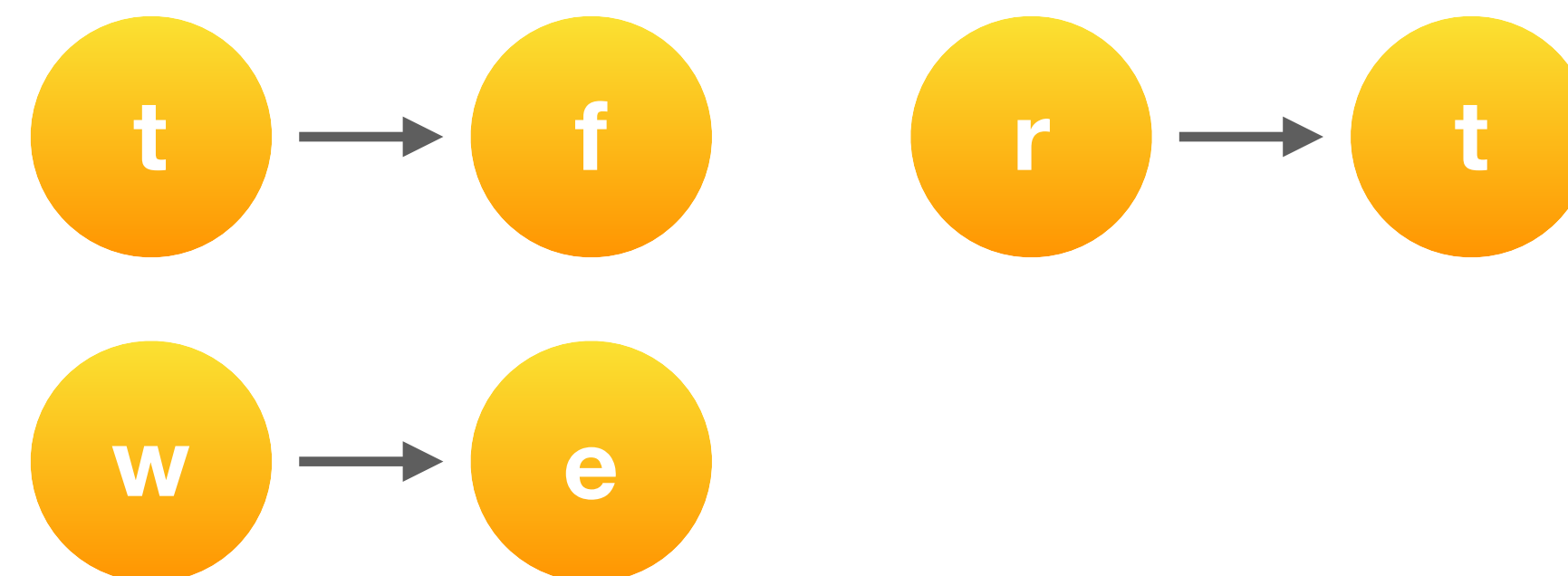
269. 火星字典

wrf
er



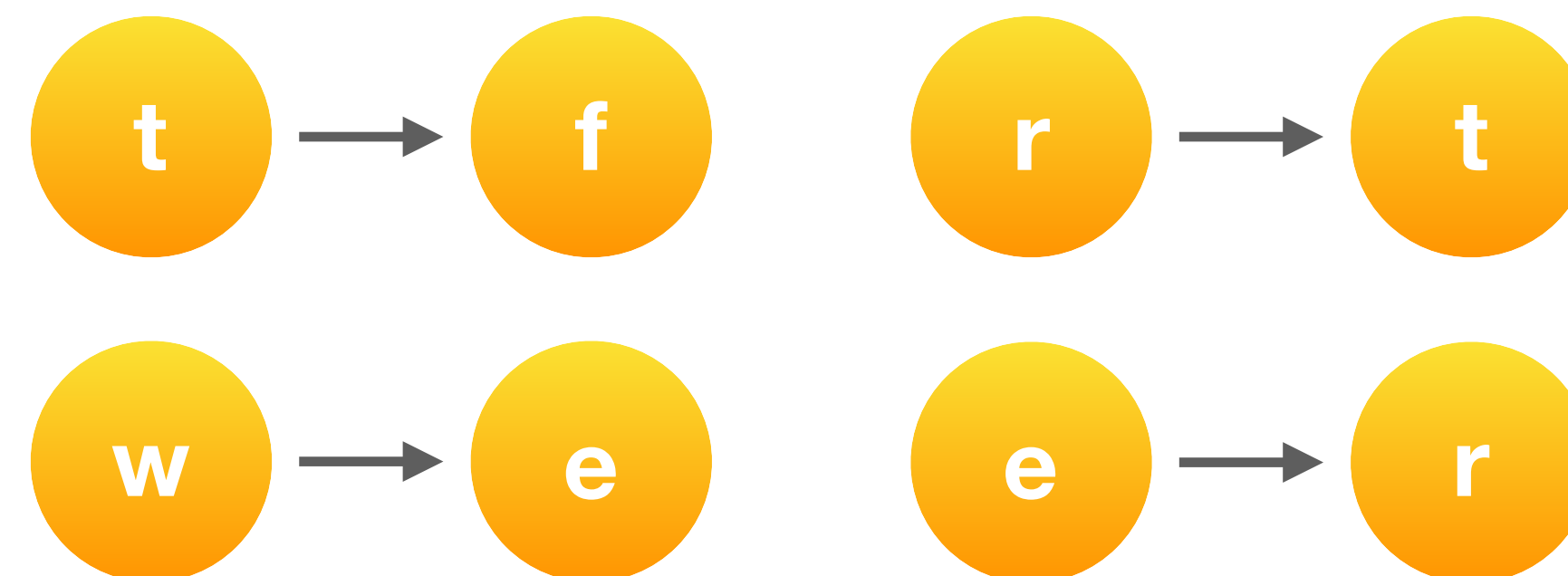
269. 火星字典

er
ett

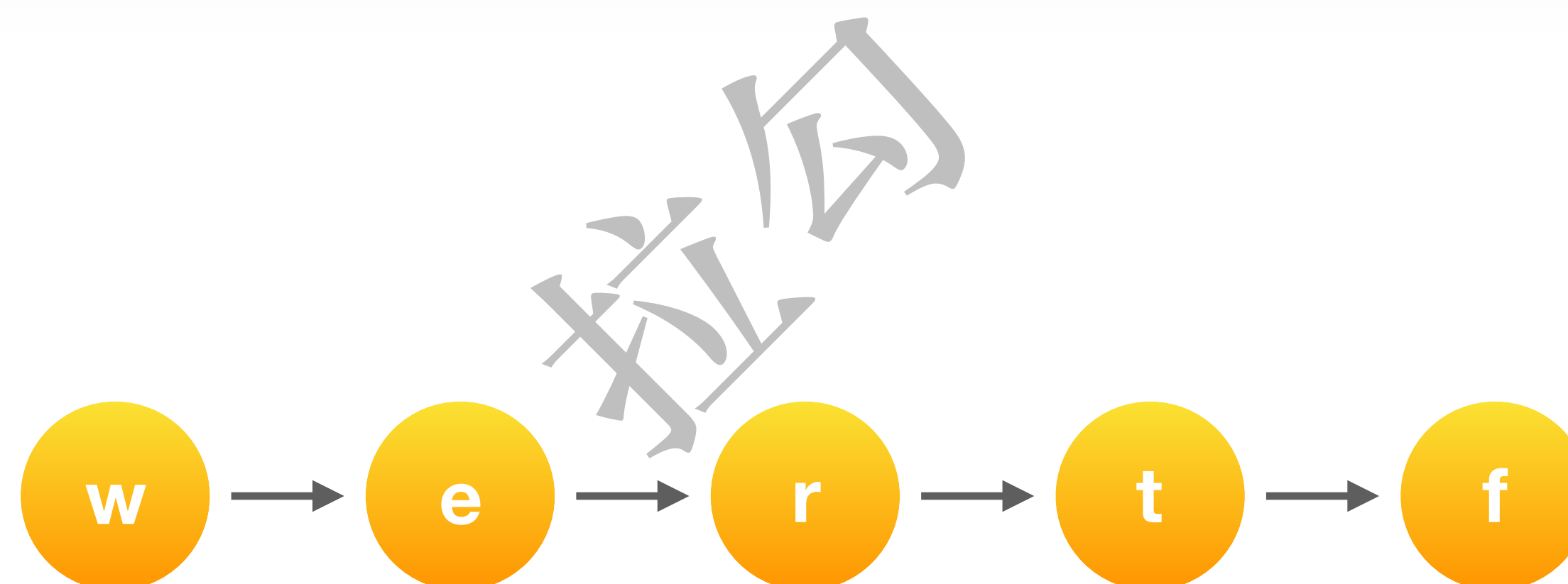


269. 火星字典

ett
rftt



269. 火星字典




```
String alienOrder(String[] words) {  
    if (words == null || words.length == 0)  
        return null;  
  
    if (words.length == 1) {  
        return words[0];  
    }  
}
```

```
Map<Character, List<Character>> adjList =  
new HashMap<>();
```

- 首先根据输入构建一个有向图
 - 输入为空
- 第二步：对该有向图进行拓扑排序
 - 输入的字符串只有一个
- 构建有向图：定义邻接链表 adjList，表示有向图也可使用邻接矩阵

```
for (int i = 0; i < words.length - 1; i++) {  
    String w1 = words[i], w2 = words[i + 1];  
    int n1 = w1.length(), n2 = w2.length();  
  
    boolean found = false;  
  
    for (int j = 0; j < Math.max(w1.length(), w2.length()); j++) {  
        Character c1 = j < n1 ? w1.charAt(j) : null;  
        Character c2 = j < n2 ? w2.charAt(j) : null;  
  
        if (c1 != null && !adjList.containsKey(c1)) {  
            adjList.put(c1, new ArrayList<Character>());  
        }  
  
        if (c2 != null && !adjList.containsKey(c2)) {  
            adjList.put(c2, new ArrayList<Character>());  
        }  
  
        if (c1 != null && c2 != null && c1 != c2 && !found) {  
            adjList.get(c1).add(c2);  
            found = true;  
        }  
    }  
}
```

- 然后两两比较字符串
- 对每个出现的字母都创建一个图里的顶点
- 一旦发现两字母不同，就链接这两个顶点
- 在这里，定义一个 found 变量，
表明一旦出现不同字母，
只需处理好这两个字母，或顶点的关系，
之后的字母不需要考虑

```
Set<Character> visited = new HashSet<>();  
Set<Character> loop = new HashSet<>();  
Stack<Character> stack = new Stack<Character>();
```

```
for (Character key : adjList.keySet()) {  
    if (!visited.contains(key)) {  
        if (!topologicalSort(adjList, key, visited, loop, stack)) {  
            return "";  
        }  
    }  
}
```

```
StringBuilder sb = new StringBuilder();
```

```
while (!stack.isEmpty()) {  
    sb.append(stack.pop());  
}
```

```
return sb.toString();  
}
```

- 接下来就是经典的拓扑排序
- 进行拓扑排序时，
我们需要一个堆栈 `stack` 来记录顶点的顺序。
最后将 `stack` 倒过来输出，
即为最终结果

```
boolean topologicalSort(Map<Character, List<Character>> adjList, char u,
                        Set<Character> visited, Set<Character> loop,
                        Stack<Character> stack) {
    visited.add(u);
    loop.add(u);

    if (adjList.containsKey(u)) {
        for (int i = 0; i < adjList.get(u).size(); i++) {
            char v = adjList.get(u).get(i);

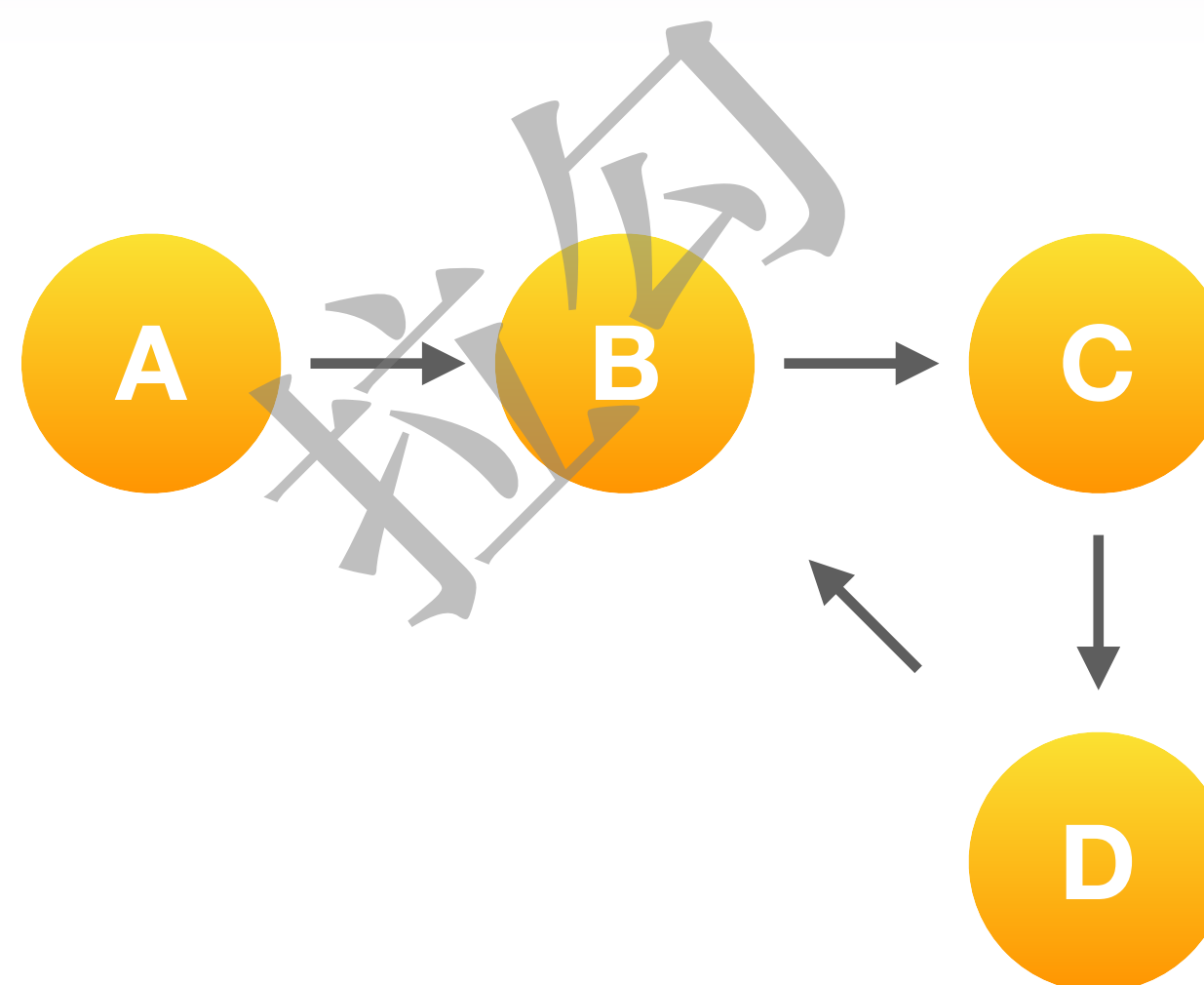
            if (loop.contains(v))
                return false;

            if (!visited.contains(v)) {
                if (!topologicalSort(adjList, v, visited, loop, stack)) {
                    return false;
                }
            }
        }
    }

    loop.remove(u);

    stack.push(u);
    return true;
}
```

- 用深度优先的方法来进行拓扑排序：
 - visited 集合：用来记录已访问过的顶点
 - stack 堆栈：
当从某顶点出发，访问完其他所有顶点，
最后才把当前顶点加入到堆栈，
即要想把该点加入 stack 里，
必须先把其他跟它有联系的顶点都处理完毕
 - loop 集合：有效防止有向图中出现环的情况



```
boolean topologicalSort(Map<Character, List<Character>> adjList, char u,
                        Set<Character> visited, Set<Character> loop,
                        Stack<Character> stack) {
    visited.add(u);
    loop.add(u);

    if (adjList.containsKey(u)) {
        for (int i = 0; i < adjList.get(u).size(); i++) {
            char v = adjList.get(u).get(i);

            if (loop.contains(v))
                return false;

            if (!visited.contains(v)) {
                if (!topologicalSort(adjList, v, visited, loop, stack)) {
                    return false;
                }
            }
        }
    }

    loop.remove(u);

    stack.push(u);
    return true;
}
```

- 将当前节点 u 加入 $visited$ 集合 以及 $loop$ 集合中
- 逐个访问与顶点 u 相邻的其他顶点 v
- 如果在此轮访问过程中, v 其实早已被访问过, 则此处有环出现, 返回 `false`
- 否则, 如果顶点 v 还没有被访问过, 就递归地访问它, 如果在访问顶点 v 时发现了环, 则返回 `false`
- 当这一轮访问结束后, 即从顶点 u 出发 访问完毕所有能访问的点, 将 u 从 $loop$ 集合里删除
- 把 u 加入到堆栈中
- 返回 `true`, 表明此时访问没有发现环

772. 基本计算器 III

实现一个基本的计算器来计算简单的表达式字符串。

表达式字符串可以包含左括号 (和右括号)，加号 + 和减号 -，非负整数和空格。

表达式字符串只包含非负整数， +, -, *, / 操作符，左括号 (，右括号) 和空格。整数除法需要向下截断。

你可以假定给定的字符串总是有效的。

所有的中间结果的范围为 $[-2147483648, 2147483647]$ 。

示例 1:

`"1 + 1" = 2`

`" 6-4 / 2 " = 4`

`"2*(5+5*2)/3+(6/2+8)" = 21`

`"(2+6* 3+5- (3*14/7+2)*5)+3" = -12`

772. 基本计算器 III

解题思路 - Case 1

从一些最基本的情况进行考虑：

如果表达式里只有数字和加法符号，没有减法，也没有空格，并且输入的表达式一定合法，那么应该如何处理？

例如：1+2+10。

解法 - Case 1

这个解法很简单，思路就是一旦遇到了数字就不断地相加。


```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        queue.offer(c);  
    }
```

```
    int num = 0, sum = 0;
```

```
    while (!queue.isEmpty()) {  
        char c = queue.poll();
```

```
        if (Character.isDigit(c)) {  
            num = 10 * num + c - '0';
```

```
        } else {  
            sum += num;  
            num = 0;
```

```
        }
```

```
    }
```

```
    sum += num;
```

```
    return sum;
```

```
}
```

- 首先做一个转换，将字符串的字符放入一个优先队列中
- 定义 num 变量表示当前数字，
定义 sum 变量记录最后的和
- 遍历优先队列，从队列中逐个取出字符
- 如果当前字符是数字，则更新 num 变量
- 如果遇到加号，则把当前 num 加入到 sum 中
num 清零
- 由于字符串的最后没有加号，
最后还要再一次将 num 加入 sum 中
- 返回结果

```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        queue.offer(c);  
    }  
    queue.add('+'); // 在末尾添加一个加号  
  
    while (!queue.isEmpty()) {  
        ...  
    }  
  
    return sum;  
}
```

- 在优先队列的最后添加一个加号

```
int calculate(String s) {  
    ...  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    ...  
}
```

- 拓展：输入时允许空格，如何处理？
- 添加到优先队列时，过滤掉空格即可

772. 基本计算器 III

升级：支持减法

▶ 方法一：借助两个 stack

- 存放数字
- 存放符号

▶ 方法二：将表达式看做 $1 + 2 + (-10)$

$$1 + 2 - 10$$

772. 基本计算器 III

+ 1 + 2 - 10 +

```
sign = +
```

```
num = 0 => 0
```

```
sum = 0 => 1
```

772. 基本计算器 III

+ 1 + 2 - 10 +

sign = + => -

num = 0 => 0

sum = 1 + 2 => 3

772. 基本计算器 III

+ 1 + 2 - 10 +

sign = + => -

num = 0 => 00

sum = 3 + 2 => 3

772. 基本计算器 III

+ 1 + 2 - 10 +

sign = - => +

num = 10 => 00

sum = 3 - 10 => -7


```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    queue.add('+');
```

```
char sign = '+'; // 添加一个符号标志变量
```

```
int num = 0, sum = 0;
```

- 定义一个新变量 `sign` 用来记录当前数字的正负
初始化为正数

```
while (!queue.isEmpty()) {  
    char c = queue.poll();  
  
    if (Character.isDigit(c)) {  
        num = 10 * num + c - '0';  
    } else {  
        // 遇到了符号，表明我们要开始统计一下当前的结果了  
        if (sign == '+') { // 数字的符号是 +  
            sum += num;  
        } else if (sign == '-') { // 数字的符号是 -  
            sum -= num;  
        }  
  
        num = 0;  
        sign = c;  
    }  
}  
  
return sum;  
}
```

- 定义一个新变量 `sign` 用来记录当前数字的正负
初始化为正数
- 遇到符号字符时，判断一下：
 - 如果当前数字的符号是加号，即正数，直接加入至总和
 - 如果当前数字的符号是减号，即负数，
从总和中减去该数
- 最后更新一下当前符号位，并将 `num` 清零

772. 基本计算器 III

升级：支持乘法和除法

▸ 考虑符号优先级

1 + 2 × 10

772. 基本计算器 III

+ 1 + 2 × 10 +

```
sign = +
```

```
num = 0 => 0
```

```
stack = {1}
```

772. 基本计算器 III

+ 1 + 2 × 10 +

sign = * => +

num = 00 => 0

stack = {1} 2} => {1, 20} 10}

772. 基本计算器 III

+ 1 + 2 × 10 +

sum = 1 + 20 = 21

```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    queue.add('+');  
  
    char sign = '+';  
    int num = 0;
```

```
Stack<Integer> stack = new Stack<>();
```

- 定义一个新变量 stack 用来记录要被处理的数

```
while (!queue.isEmpty()) {  
    char c = queue.poll();  
  
    if (Character.isDigit(c)) {  
        num = 10 * num + c - '0';  
    } else {  
        if (sign == '+') {  
            stack.push(num);  
        } else if (sign == '-') {  
            stack.push(-num);  
        } else if (sign == '*') {  
            stack.push(stack.pop() * num);  
        } else if (sign == '/') {  
            stack.push(stack.pop() / num);  
        }  
    }  
}
```

- 定义一个新变量 `stack` 用来记录要被处理的数
- 当遇到的是加号，把当前的数压入堆栈中
- 当遇到的是减号，把当前数的相反数压入堆栈中
- 当遇到的是乘号，把前一个数从堆栈中取出，然后和当前的数相乘，相乘完毕之后再放回堆栈
- 当遇到的是除号，把前一个数从堆栈中取出，然后除以当前的数，运算之后将结果放回堆栈


```
num = 0;  
sign = c;  
}  
}
```

```
int sum = 0;  
while (!stack.isEmpty()) {  
    sum += stack.pop();  
}  
return sum;  
}
```

- 定义一个新变量 `stack` 用来记录要被处理的数
- 当遇到的是加号，把当前的数压入堆栈中
- 当遇到的是减号，把当前数的相反数压入堆栈中
- 当遇到的是乘号，把前一个数从堆栈中取出，然后和当前的数相乘，相乘完毕之后再放回堆栈
- 当遇到的是除号，把前一个数从堆栈中取出，然后除以当前的数，运算之后将结果放回堆栈
- 堆栈里存储的都是需要相加的结果，将它们相加，返回总和即可

772. 基本计算器 III

升级：支持小括号

▶ 考虑符号优先级

- 遇到小括号，需先计算小括号中的内容
- 小括号中的内容，可以利用之前的方法进行计算
 - 遇到一个左括号时，我们可以递归地处理；
 - 遇到一个右括号时，表明小括号中的内容已处理完毕，递归应该返回

```
int calculate(String s) {  
    Queue<Character> queue = new LinkedList<>();  
    for (char c : s.toCharArray()) {  
        if (c != ' ') queue.offer(c);  
    }  
    queue.offer('+');  
  
    return calculate(queue);  
}
```

- 在主函数中调用一个递归函数

```
int calculate(Queue<Character> queue) {  
    char sign = '+';  
    int num = 0;  
  
    Stack<Integer> stack = new Stack<>();  
  
    while (!queue.isEmpty()) {  
        char c = queue.poll();  
  
        if (Character.isDigit(c)) {  
            num = 10 * num + c - '0';  
        }  
        else if (c == '(') {  
            num = calculate(queue);  
        }  
    }  
}
```

- 当遇到一个左括号时，开始递归地调用，求得括号中的计算结果，将它赋值给当前的 num

```
else {  
    if (sign == '+') {  
        stack.push(num);  
    } else if (sign == '-') {  
        stack.push(-num);  
    } else if (sign == '*') {  
        stack.push(stack.pop() * num);  
    } else if (sign == '/') {  
        stack.push(stack.pop() / num);  
    }  
}
```

```
num = 0;  
sign = c;
```

```
if (c == ')') {  
    break;  
}
```

```
}  
}  
}
```

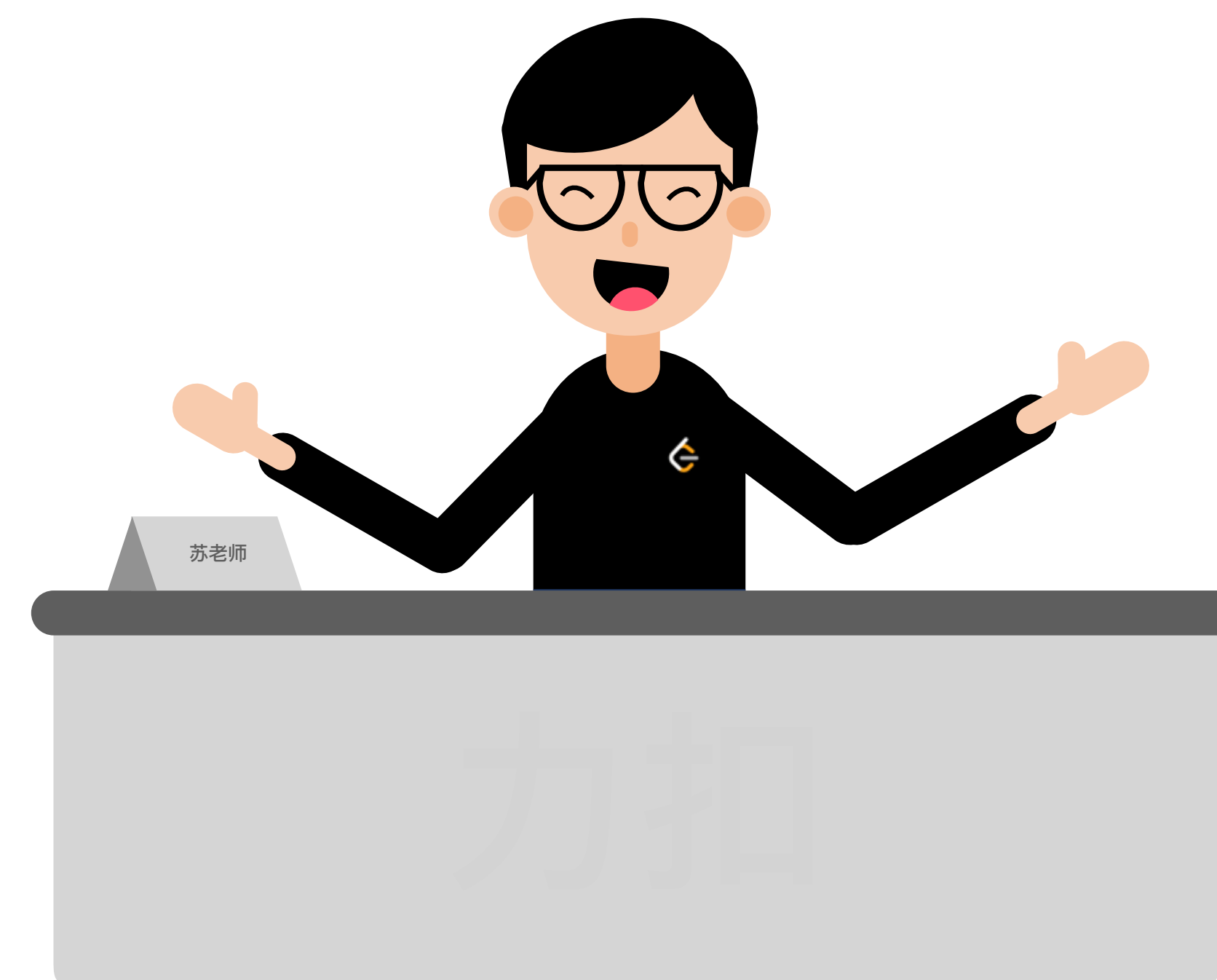
```
int sum = 0;  
while (!stack.isEmpty()) {  
    sum += stack.pop();  
}  
return sum;  
}
```

- 当遇到一个右括号时，就可以结束循环，直接返回当前总和

大厂面试高频题精讲

- 区间问题
- 拓扑排序算法
- 基本计算器

拉勾



Next: 课时 10 《大厂算法面试真题 - 难题精讲（一）》

多加练习，才能更好地巩固知识点。



关注“拉勾教育”
学习技术干货



关注“LeetCode力扣”
获得算法技术干货