華中科技大學

课 程 实 验 报 告

课程名称：串行与并行数据结构及算法

专业班级：本硕博 2301

学　　号：U202315763

姓　　名：王家乐

指导教师：陆枫

报告日期：2025.11.17

计算机科学与技术学院

目录

1 无重复排序

1.1 题目描述

给出一个具有 N 个互不相同元素的数组，对它进行升序排序。

1.2 算法流程

使用快速排序算法，递归地将列表进行排序：首先选取列表的第一个元素作为基准值，然后将剩余元素依据与基准值的大小关系分割为两个子列表，接着对这两个子列表分别进行递归排序，最后将排序后的左子列表、基准值和右子列表依次合并，从而得到最终的有序列表。

1.3 时间、空间复杂度

• **时间复杂度**：该算法的时间复杂度在平均和最佳情况下为 $O(n \log n)$。这是因为在理想情况下，每次选择的基准值都能将列表大致平分为两个子列表，使得递归树的深度为对数级别。然而在最坏情况下，例如当输入列表已经有序或逆序时，每次选择的基准值都是当前列表的最小或最大元素，导致分区极不平衡，算法退化为 $O(n^2)$ 的复杂度。

• **空间复杂度**：该算法的空间复杂度主要取决于递归调用的深度。在平均和最佳情况下，递归深度为 $O(\log n)$，因此空间复杂度也为 $O(\log n)$。但在最坏情况下，由于分区不平衡导致递归链变得很长，递归深度会达到 $O(n)$，从而使得空间复杂度上升到 $O(n)$。

1.4 样例数据

图 1-1 无重复排序样例数据

1. 初始调用：quicksort([10, 155, 200, 9, 60, 174, 17, 6, 172, 103])
    - 基准：10
    - 划分：left：[9, 6], right：[155, 200, 60, 174, 17, 172, 103]
    - 递归：quicksort([9, 6]) @ [10] @ quicksort([155, ...])
2. quicksort([9, 6])
    - 基准：9
    - 划分：left：[6], right：[]
    - 递归：quicksort([6]) @ [9] @ quicksort([])
    - 返回：[6, 9]
3. quicksort([155, 200, 60, 174, 17, 172, 103])
    - 基准：155
    - 划分：left：[60, 17, 103], right：[200, 174, 172]
    - 递归：quicksort([60, 17, 103]) @ [155] @ quicksort([200, 174, 172])
4. quicksort([60, 17, 103])
    - 基准：60
    - 划分：left：[17], right：[103]
    - 返回：[17, 60, 103]
5. quicksort([200, 174, 172])
    - 基准：200
    - 划分：left：[174, 172], right：[]
    - 递归：quicksort([174, 172]) @ [200] @ quicksort([])
6. quicksort([174, 172])
    - 基准：174
    - 划分：left：[172], right：[]
    - 返回：[172, 174]

合并结果：

第 5 步合并为 [172, 174, 200]

第 3 步合并为 [17, 60, 103, 155, 172, 174, 200]

第 1 步合并为 [6, 9, 10, 17, 60, 103, 155, 172, 174, 200]

2 最短路

2.1 题目描述

给定一个带权无向图和一个源点，计算从该源点到其他所有节点的最短路径。图中的边权为非负值，需要输出从源点到每个节点的最短距离。

2.2 算法流程

采用 Dijkstra 算法来解决该最短路径问题，具体流程如下：

·初始化：

– 读取节点数 N、边数 M 和源点 T。

– 使用距离数组 dist 记录源点到各节点的当前最短距离。将源点 T 的距离设为 0，其余节点距离均设为 INF（一个足够大的数），表示无限大。

– 使用布尔数组 visited 标记节点是否已找到最短路径，初始时全部设为 false。

– 使用邻接表 adj（一个数组，每个元素是一个列表）存储图的信息。读取 M 条边的信息，对于每条无向边 (u, v, w)，在邻接表中同时添加 u 到 v 和 v 到 u 的边。

·主循环：

– 在所有未被访问的节点中，选择距离源点最近的节点 u，即满足 dist[u] 最小且 visited[u] 为 false 的节点。

– 如果找不到这样的节点（所有可达节点均已访问），则循环结束。

– 将节点 u 标记为已访问，即设置 visited[u] = true。

– 遍历节点 u 在邻接表中的所有邻接节点 v（边权为 w），如果通过节点 u 到达节点 v 的距离（即 dist[u] + w）小于当前记录的距离 dist[v]，则更新 dist[v] = dist[u] + w。

·算法结束： dist 数组中存储的即为源点 T 到所有节点的最短路径长度。对于无法到达的节点，其距离仍为初始值 INF，在输出时转换为 -1。

2.3 时间、空间复杂度

• **时间复杂度**：该算法的时间复杂度为 $O(N^2)$。由 Dijkstra 算法的核心部分决定。主循环最多执行 N 次（N 为节点数），在每次循环中，都需要线性扫描 dist 数组查找当前未访问节点中距离源点最近的节点，其本身的时间复杂度为 O(N)。因此，仅这部分操作的总时间复杂度就是 $O(N^2)$，成为整个算法的性能瓶颈。

• **空间复杂度**：该算法的空间复杂度为 O(N + M)。空间主要消耗在存储图的结构和算法运行所需的数据结构上。邻接表用于存储图的 N 个节点和 M 条边，需要 O(N + M) 的空间。此外，还需要大小为 N+1 的距离数组 dist 和布尔数组 visited，它们分别需要 O(N) 的空间。

2.4 样例数据

图 2-1 最短路样例数据

步骤	已确定集合 S	距离数组 dist
0	{5}	[∞ , ∞ , ∞ , ∞ , 0, 3, 5]
1	{5, 6}	[4, ∞ , 7, ∞ , 0, 3, 5]
2	{5, 6, 1}	[4, ∞ , 7, 7, 0, 3, 5]
3	{5, 6, 1, 7}	[4, 6, 7, 7, 0, 3, 5]
4	{5, 6, 1, 7, 2}	[4, 6, 7, 7, 0, 3, 5]
5	{5, 6, 1, 7, 2, 4}	[4, 6, 7, 7, 0, 3, 5]
6	{5, 6, 1, 7, 2, 4, 3}	[4, 6, 7, 7, 0, 3, 5]

表 2-1 Dijkstra 算法计算过程

2.5 特殊情况

• **不连通节点**：该算法能够正确处理。在初始化阶段，所有节点的距离 dist 都被设置为一个极大的值 INF，只有源点 T 的距离为 0。算法从源点开始，通过边不断探索和更新可达节点的距离。如果一个节点与源点 T 不在同一个连通分量中，那么从 T 出发将永远无法到达该节点。因此，该节点的距离值将始终保持为初始的 INF。最终输出结果时代码会检查每个节点的距离值，如果距离等于 INF，则会输出 -1，明确地表示该节点不可达。

• **负权值的边**：该算法无法保证给出正确的结果。Dijkstra 算法是一个贪心算法，其正确性的前提是图中所有的边权都是非负的。它的核心思想是：每次都选择当前距离源点最近且未被访问的节点，并确定其最短路径。一旦一个节点被标记为 visited，算法就认为已经找到了到该点的最短路径，不会再对其进行更新。如果图中存在负权边，这个贪心策略就会失效，因为通过一个带有负权值的边，可能会使得一个已经被标记为 visited 的节点产生一条更短的路径。在本题中，已明确所有边权均为非负值。

3 最大括号距离

3.1 题目描述

给定一个由`(` 和`)` 构成的字符串，在其中找到所有匹配的子串（即满足闭合定义的串，且外侧包裹一对括号），并输出这些子串中长度最长的那个的长度。

3.2 算法流程

算法的主要思想是使用栈来匹配括号，同时在匹配成功时计算匹配的长度，维护一个最大长度变量。具体流程如下：

• **初始化**：创建一个空栈（stack），用于存储遇到的左括号的位置索引。初始化一个变量 max 为 0，用于记录和更新找到的最大距离。

• **遍历序列**：算法从左到右依次处理序列中的每个元素及其位置索引。

- **处理规则：**
  - 遇到左括号 (0)：将其位置索引压入栈中。
  - 遇到右括号 (1)：检查栈是否为空。如果栈是空的，说明这个右括号没有匹配的左括号，直接忽略它，继续处理下一个元素。如果栈不为空，说明找到了一个匹配的括号对。从栈顶弹出一个位置索引，这个索引是与当前右括号匹配的那个左括号的位置。计算当前右括号的位置与弹出的左括号位置之间的距离（当前位置 - 栈顶位置 + 1）。将这个新计算出的距离与当前记录的最大距离 max 进行比较，如果新距离更大，则更新 max。
- **结束：**遍历整个序列后，变量 max 中存储的值就是所求的最大括号距离。

```
TheLongestDistance S(⟨ (0, x_1), (1, x_2), ⋯ ⟩ ) =
let
    match (i, x), (stack, m) =
        if x = 0 then
            (i ++ stack, m)
        else if |stack| = 0 then
            ([], m)
        else
            let
                h ← hd stack
                ms ← max(m, i - h + 1)
                rem ← tl stack
            in
                (rem, ms)
            end
end
in
    second reduce match ([], 0) S
end
```

3.3 时间、空间复杂度

- **时间复杂度：**遍历序列需要 O(N) 的时间，其中 N 为序列的长度。在遍历过程中，每个字符只进行有限次操作（压栈、弹栈、比较）。因此，总的时间复杂度为 O(N)。
- **空间复杂度：**使用了一个栈 stack，最坏情况下栈的深度为 N（当所有字符都是`(` 时）。因此，空间复杂度为 O(N)。

3.4 样例数据

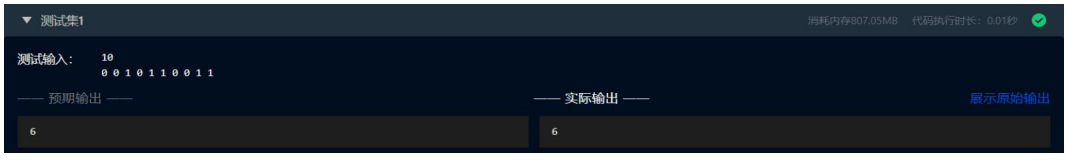

图 3-1 最大括号距离样例数据

位置	字符	stack	length	max
0	0	[0]	0	0
1	0	[1, 0]	0	0
2	1	[0]	2	2
3	0	[3, 0]	0	2
4	1	[0]	2	2
5	1	[ ]	6	6
6	0	[6]	0	6
7	0	[7, 6]	0	6
8	1	[6]	2	6
9	1	[ ]	4	6

表 3-1 最大括号距离计算过程

3.5 特殊情况

·未匹配的右括号：如果当前字符是右括号`)`，但栈为空，则说明没有可匹配的左括号，直接跳过。

·未匹配的左括号：在遍历结束后，栈中可能还剩余未匹配的左括号`(`，这不会影响已经计算的最大匹配长度。

·空串：如果输入的序列为空（N = 0），则输出 0。

4 天际线

4.1 题目描述

给定一组建筑物，每个建筑物由三个整数 (Li , Hi , Ri) 表示，分别表示建筑物的左边界坐标 Li ，高度 Hi ，以及右边界坐标 Ri。所有建筑物都是矩形，且都与地面对齐。计算由这些建筑物形成的天际线，输出应为一系列关键点 (x, height)，表示在横坐标 x 处，天际线的高度变为 height。

4.2 算法流程

采用“分而治之”的策略来解决天际线问题。

• **分解（Divide）**：首先，算法将输入的建筑列表递归地对半切分，直到每个子问题只包含一个建筑或没有建筑。

• **解决（Conquer）**：对于只包含一个建筑的“基本”子问题，其天际线轮廓可以由两个关键点表示：建筑的左上角 (左边界, 高度) 和右下角 (右边界, 0)。

• **合并（Merge）**：这是算法的核心。它将两个已经计算好的子天际线轮廓（表示为关键点的有序列表）合并成一个。合并过程类似于一次扫描：

– 想象一条扫描线从左到右移动。同时遍历两个子天际线关键点列表。

– 在每个关键点的 x 坐标处，算法会更新来自两个子天际线的当前高度。

– 计算出这两个高度中的较大值，作为合并后轮廓在该点的新高度。

– 只有当这个新的最大高度与前一个点的最大高度不同时，才意味着天际线轮廓发生了变化，此时将这个新的关键点 (x 坐标, 新最大高度) 添加到最终结果中。

– 合并过程持续进行，直到两个子天际线的所有关键点都被处理完毕。

```
Skyline B_s=
if |B_s| = 1 then
    [(l_1, h_1), (r_1, 0)]
else
    let
        len ← |B_s| / 2
        B_l ← Skyline <(l_1, r_1, h_1), ..., (l_len, r_len, h_len)>
        B_r ← Skyline <(l_len+1, r_len+1, h_len+1), ..., (l_n, r_n, h_n)>
    in
        merge B_l B_r
    end
end
```

4.3 时间、空间复杂度

• **时间复杂度**：算法采用了分治策略，将问题递归地分解为两半，贡献了 log n 的因子。在每一层递归中，合并两个子天际线的过程需要线性时间，即 O(n)，因为它需要遍历两个子天际线的所有关键点。因此，总时间复杂度符合分治算法主定理的形式，为 O(n log n)。

• **空间复杂度**：空间复杂度为 O(n)。虽然递归调用本身会产生 O(log n) 的栈深度，但在算法执行过程中，需要存储由子问题生成的天际线点列表。在任何递归

深度上，为中间天际线列表和最终结果列表分配的总空间都与输入建筑物的数量 n 成线性关系。因此，主导空间消耗的是这些列表的存储，导致整体空间复杂度为 O(n)。

4.4 样例数据

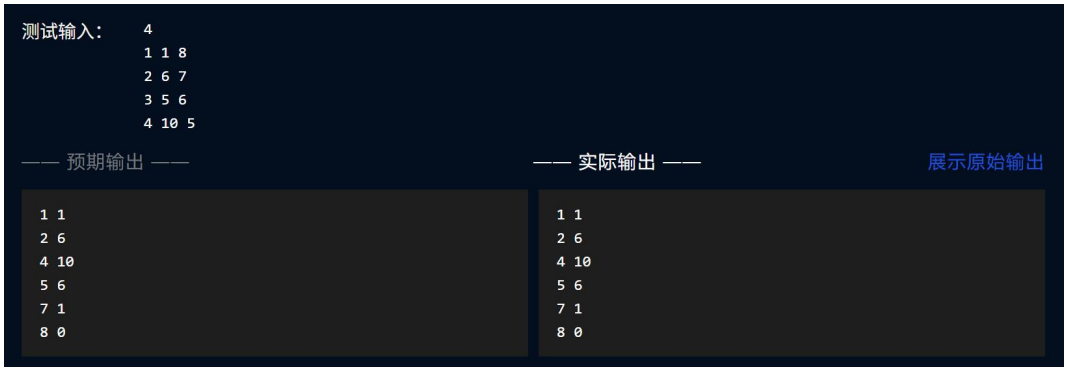

图 4-1 天际线样例数据

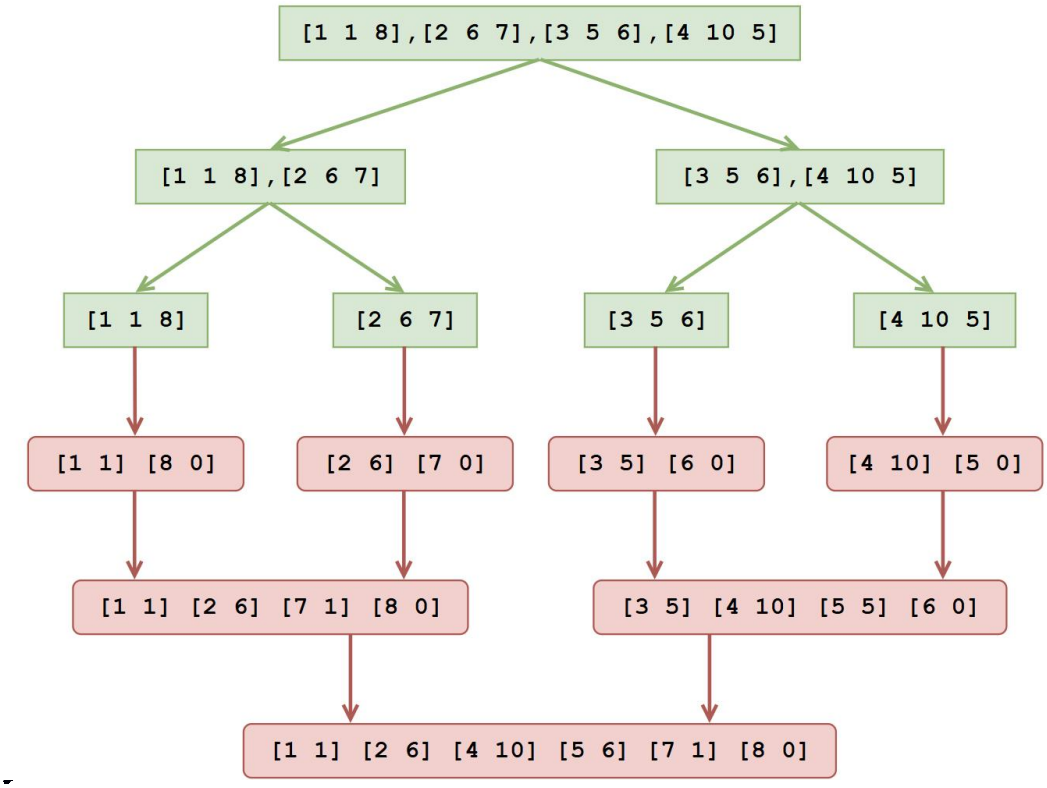

图 4-2 skyline 算法执行过程

4.5 特殊情况

• **高度相同的建筑物**：如果多个建筑物具有相同的高度，算法仍然能够正确处理，因为高度计数数组会累加相同高度的建筑物数量。

5 括号匹配

5.1 题目描述

给定一个由括号组成的序列，判断该序列是否是匹配的。括号用数字表示： 0 代表左括号`(`，1 代表右括号`)`。需要注意的是， 对于序列 0 1 0 1，在本题中也视为匹配的。

5.2 算法流程

为判断括号序列是否匹配，我们使用栈的思想，遍历序列，维护一个计数器 state，表示当前未匹配的左括号数量。具体步骤如下：

• 初始化：将 state 初始化为 0，表示当前没有未匹配的左括号。

• 遍历序列：遇到左括号就将 state 加 1，表示增加一个未匹配的左括号。遇到右括号则判断 state 是否大于 0，如果是，将 state 减 1，表示匹配一个左括号。如果否，说明出现了没有匹配的右括号，序列不匹配，立即返回 -1。

• 遍历结束后，判断 state 的值：等于 0 则所有左括号都被匹配，序列匹配，返回 1。不等于 0 则说明还有未匹配的左括号，序列不匹配，返回 0。

5.3 时间、空间复杂度

• **时间复杂度**：遍历整个括号序列，每个字符只进行一次判断和加减操作，时 间复杂度为 O(N)。

• **空间复杂度**：只使用了一个计数器 state，空间复杂度为 O(1)。

5.4 样例数据

图 5-1 括号匹配样例数据

步骤	字符	state
1	0	state = 1
2	0	state = 2
3	0	state = 3
4	1	state = 2
5	1	state = 1
6	1	state = 0

表 5-1 括号匹配计算过程

5.5 特殊情况

- **遍历过程中 state < 0**：说明出现了没有匹配的右括号，序列不匹配。
- **遍历结束后 state > 0**：说明有未匹配的左括号，序列不匹配。
- **空串**：如果 N = 0，序列为空，state = 0，序列匹配，输出 1。

6 高精度整数

6.1 题目描述

给定两个任意精度的非负整数 a 和 b，满足 $a \geq b$。每个整数由一串数字组成，数字从高位到低位输入，可能包含前导零。要求计算并输出它们的和 $a + b$、差 $a - b$ 和积 $a \times b$。

6.2 算法流程

核心思想是模拟我们手动进行竖式计算的过程。

首先，程序将两个大整数作为数字列表读入，并进行预处理：移除数字前方的无效“0”，然后将整个数字列表反转。这样做的目的是让数字的最低位排在列表的最前面，便于从低位到高位进行计算，处理进位和借位。

- **高精度加法**：从两个数字列表的最低位（也就是列表的头部）开始。将对应位置的两个数字与来自前一位的“进位”（初始为 0）相加。相加结果的个位数成为当

前位的结果，十位数则作为新的“进位”传递给下一位的计算。重复此过程，直到其中一个较短的数字列表被处理完毕。继续将剩余的较长数字列表的每一位与“进位”相加。如果所有数字都处理完后仍有“进位”，则将其作为结果的最高位。

• **高精度减法**（假设被减数大于等于减数）：同样从两个数字列表的最低位开始。用被减数的位减去减数的对应位，再减去来自前一位的“借位”（初始为 0）。如果结果为负数，则需要向高位“借一”，即给当前结果加上 10，并设置“借位”为 1，供下一位计算使用。如果结果为非负数，则它就是当前位的结果，且“借位”为 0。重复此过程直到减数列表处理完毕。之后，继续用被减数列表中剩余的位减去“借位”，直到所有位都处理完成。最后，移除结果中可能出现的前导零。

• **高精度乘法**：该算法模拟了竖式乘法。算法遍历乘数的每一位（从最低位开始）。对于乘数的某一位，用它去乘以整个被乘数（这本身就是一个小的“高精度乘单个数”的过程），得到一个“部分积”。根据当前乘数位的权重（个位、十位、百位……），对这个“部分积”进行“错位”处理，即在末尾补上相应数量的 0。使用前面定义的高精度加法，将这个错位后的“部分积”累加到最终的总结果上。遍历完乘数的所有位后，累加得到的总结果就是最终的乘积。

6.3 时间、空间复杂度

1. **高精度加法：**

   • 时间复杂度：加法需要遍历长度为 n 的数字列表，时间复杂度为 O(n)。

   • 空间复杂度：结果列表的长度最多为 n + 1，空间复杂度为 O(n)。

2. **高精度减法：**

   • 时间复杂度：减法需要遍历长度为 n 的数字列表，时间复杂度为 O(n)。

   • 空间复杂度：结果列表的长度最多为 n，空间复杂度为 O(n)。

3. **高精度乘法：**

   •时间复杂度：乘法采用逐位相乘算法，需要进行 $na \times nb$ 次乘法和加法操作，时间复杂度为 $O(na \times nb)$。由于 na 和 nb 均为 O(n)，因此 总的时间复杂度为 $O(n^2)$。可以使用更高效的算法（如 Karatsuba 算法 或 FFT 算法）将时间复杂度降低到 O(nlog n)。

   • 空间复杂度：中间结果和最终结果长度为 $na + nb$ ，空间复杂度为 O(n)。

6.4 样例数据

图 6-1 高精度整数示例数据

位数	**A** 位	**B** 位	结果（含进位）
个位	3	0	3
十位	3	3	6
百位	2	2	4

表 6-1 加法运算过程

位数	**A** 位	**B** 位	结果（含借位）
个位	3	0	3
十位	3	3	0
百位	2	2	0

表 6-2 减法运算过程

B 位	乘积	temp	result
0	0	0	0
3	699	6990	6990
2	466	46600	53590

表 6-3 乘法运算过程

6.5 特殊情况

- 如果其中一个整数为零，乘积结果应为零。
- 当 a = b 时，减法结果为零。

7 割点与割边

7.1 问题描述

给定一个无向无权连通图，求出该图的所有割点和割边的数目。

7.2 算法流程

该算法通过一次深度优先搜索（DFS）来找出图中的所有割点割边。

• **初始化**：为每个节点准备两个关键数值：一个是“发现时间戳”（dfn），记录节点在深度优先搜索中首次被访问的顺序；另一个是“可追溯到的最早祖先的时间戳”（low），表示该节点及其所有后代节点能通过回边到达的最早的祖先节点。

• **深度优先搜索**：从一个任意的起始节点开始进行深度优先搜索。当访问一个新节点时，立即为它分配一个当前的时间戳，并将其“发现时间戳”和“可追溯时间戳”都设置为这个值。接着，遍历该节点的所有邻居。

• **邻居是未被访问过的新节点**：这说明在搜索树中，该邻居是当前节点的子节点。对这个邻居节点递归执行相同的搜索过程。当递归调用返回后，用邻居节点的“可追溯时间戳”来更新当前节点的“可追溯时间戳”。这是因为如果子节点能到达一个更早的祖先，那么父节点自然也能。

– 判断**关节点**：如果当前节点不是搜索树的根节点，并且其“发现时间戳”小于或等于其子节点的“可追溯时间戳”，那么当前节点就是一个关节点。表示该子节点无法通过其他路径回到当前节点的祖先，必须经过当前节点。如果当前节点是根节点，并且有两个或更多的子树，那么它也是一个关节点。

– 判断**桥**：如果当前节点的“发现时间戳”严格小于其子节点的“可追溯时间戳”，那么连接这两个节点的边就是一座桥。这表示子节点连回到当前节点本身都做不到，只能回到子树内部，因此断开此边会使图分裂。

• **邻居已被访问过**，并且不是当前节点的父节点：这表示发现了一条“回边”，即一条从当前节点指向其某个祖先节点的边。利用这个邻居的“发现时间戳”来更新当前节点的“可追溯时间戳”，因为它提供了一条回到更早时间点的路径。

• **完成**：当整个深度优先搜索结束后，所有被标记为关节点和桥的节点与边就是最终结果。

7.3 时间、空间复杂度

·时间复杂度：建立邻接表需要 O(M) 时间。DFS 遍历每个顶点和边，时间复杂度为 O(N + M)。总时间复杂度为 O(N + M)。

·空间复杂度：邻接表占用 O(N + M) 空间。辅助数组 (dfn, low, point, edge) 占用 O(N + M) 空间。总空间复杂度为 O(N + M)。

7.4 样例数据

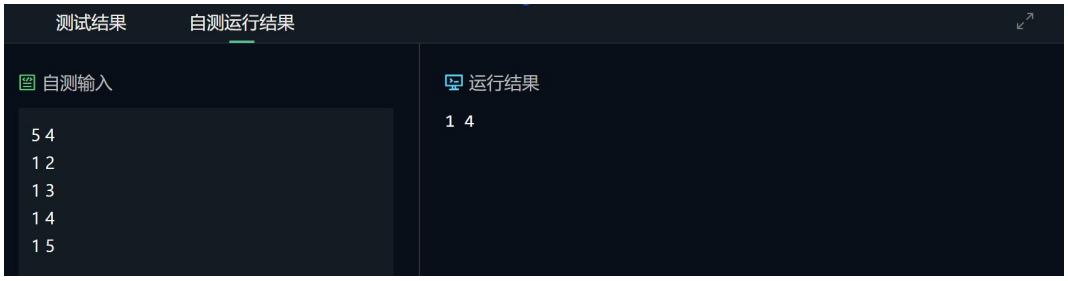

图 7-1 割点与割边样例数据

步骤	当前节点	父节点	dfn	low	point	edge
1	1	-1	dfn[1] = 1	low[1] = 1	-	-
2	2	1	dfn[2] = 2	low[2] = 2	-	edge[1-2] = true
回溯	1	-	-	low[1] = 1	point[1] = true	-
3	3	1	dfn[3] = 3	low[3] = 3	-	edge[1-3] = true
回溯	1	-	-	low[1] = 1	point[1] = true	-
4	4	1	dfn[4] = 4	low[4] = 4	-	edge[1-4] = true
回溯	1	-	-	low[1] = 1	point[1] = true	-
5	5	1	dfn[5] = 5	low[5] = 5	-	edge[1-5] = true
回溯	1	-	-	low[1] = 1	point[1] = true	-

表 7-1 Tarjan 算法计算割点和割边过程

7.5 特殊情况

·完全连通的图中，没有割点和割边，因为每个节点之间都有直接的连接， 不会因去除某个节点或边而割裂图。

· 在环形结构的图中，每条边都连接成一个闭合环，因此也没有割点或割边。

8 静态区间查询

8.1 题目描述

给定一个长度为 N 的数列，和 M 次询问，求出每一次询问的区间内数字的最大值。

8.2 算法流程

使用线段树这种数据结构来高效地解决静态区间最大值查询问题。整个过程分为两个主要阶段：**构建**和**查询**。

• **构建阶段**：首先，算法会根据输入的原始数组递归地构建一棵线段树。树的每个叶子节点对应原始数组中的一个元素。每个非叶子节点（内部节点）代表一个区间，其存储的值是它所代表的区间内所有元素的最大值。这个值是通过合并其左右两个子节点的值（即取两者中的较大者）来计算的。这个构建过程从代表整个数组的根节点开始，不断将区间对半分割，直到区间大小为 1（即叶子节点），然后再逐层回溯计算父节点的值。

• **查询阶段**：当需要查询某个指定区间 [L, R] 的最大值时，算法会从线段树的根节点开始进行遍历。在遍历过程中，对于当前节点区间，有三种情况：

– 完全不包含： 如果当前节点代表的区间与查询区间 [L, R] 没有任何交集，则忽略这个节点。

– 完全包含：如果当前节点代表的区间被查询区间 [L, R] 完全覆盖，那么该节点预存的值就是这个子问题的答案，直接返回该值，无需再向下遍历。

– 部分重叠： 如果当前节点代表的区间与查询区间 [L, R] 只有部分重叠，则算法会递归地进入该节点的左、右两个子节点继续进行查询。

• **最后**，将所有从递归调用中返回的结果进行比较，取其中的最大值作为查询区间 [L, R] 的最终答案。

8.3 时间、空间复杂度

• **时间复杂度**：构建每个节点的值只需一次操作，整个数列大小为 N，线段树包含 2N - 1 个节点，时间复杂度为 O(N)。每次查询过程中，递归查找左右子树的

过程类似于二分搜索，共有 M 次查询，因此查询总时间复杂度 为 O(Mlog N)。综合来看，算法的总时间复杂度为 O(N + Mlog N)。

• **空间复杂度**：构建树的节点数量为 2N - 1，所以线段树的空间复杂度为 O(N)。递归调用树的深度为 O(log N)，因此调用栈的最大空间复杂度也为 O(log N)。综上，空间复杂度为 O(N)。

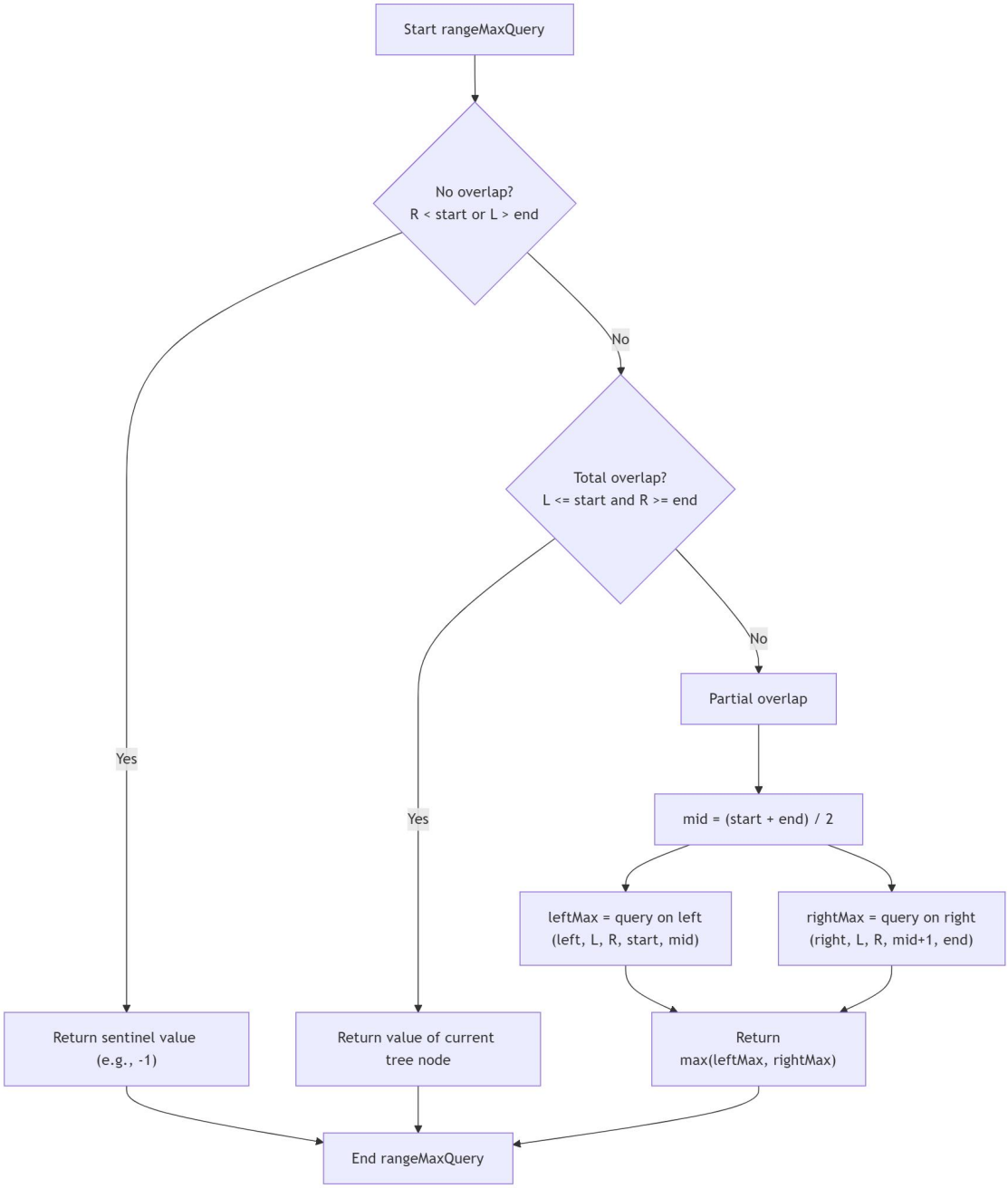

图 8-1 动态区间查询算法流程图

8.4 样例数据

图 8-2 静态区间查询样例数据

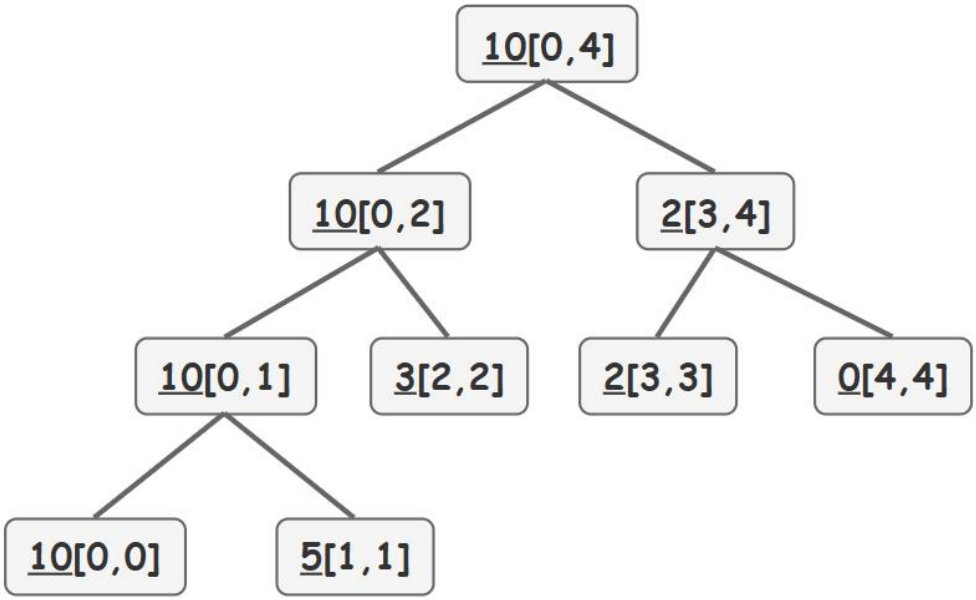

图 8-3 样例数据线段树

7.5 特殊情况

- 如果区间查询范围超出了序列的范围，返回-1 表示无效查询。
- 对于包含负数的序列，线段树仍能正确处理。
- 如果区间的左右端点相同，查询结果即为单个元素的值。

9 素性测试

9.1 题目描述

给定一个任意大的正整数 N，判断它是否是素数。

9.2 算法流程

使用 Miller-Rabin 算法。

- 检查 N 是否在一个已给定的素数表中。如果存在，则直接返回True。
- 将 N-1 进行因式分解，找到最大2^k 的因子和一个奇数 m 使得$N-1 = 2^k \cdot m$。
- 定义一个快速幂函数quickPow 用于计算$a^b \mod N$。
- Miller-Rabin 素性测试
  - 对于一个测试基a（从素数表中选取），计算a^k 的结果。
  - 若$a^m \equiv 1 \pmod p$ 或者$a^{2^{\wedge}s \cdot m} \equiv -1 \pmod p$ 其中$0 \le s < k$。只要有一个等式成立，那么后面不断平方结果也是 1。如果一个都没有成立则说明一定不是质数。
- 如果所有的测试基a 都通过，则认为N 是素数，否则不是素数。

9.3 时间、空间复杂度

- **时间复杂度**：检查素数表的时间复杂度为 O(1)。在最坏情况下（即 N 较 大），分解 N-1 的时间复杂度为 O(log N)。快速幂计算的时间复杂度为 O(log k)，其中 k 是指数。Miller-Rabin 进行 O(log N) 次的测试，因此总时间复杂度为 O(klog N)。
- **空间复杂度**：主要的空间消耗来自于存储常数大小的素数表和递归调用栈。 空间复杂度为 O(1)。

9.4 样例数据

- 10000003 不在素数表中。
- 有 N-1 = 10000002 = 2^1 • 5000001，得出 k = 1, m = 5000001。

图 9-1 素性测试样例数据

- 如果 $t \equiv 1 \bmod 10000003$ 或 $t \equiv 10000002 \bmod 10000003$，则继续。
- 否则，进行迭代，计算 $t^2 \bmod 10000003$。
- 如果在任何迭代中 $t \equiv 10000002 \bmod 10000003$，则返回素数。
- 最终根据测试输出 False，表示 10000003 不是素数。

9.5 特殊情况

当 N 是合数的时候，也有可能骗过测试算法，所以当 N 很大时，可能现有的小素数表不足以 100% 保证判断正确。