分治算法选讲

Part I

wYYSZL

2026.5.4

本节课内容

1. 分治思想概述
2. 简单应用
3. 主定理
4. 序列贡献分治

前言

主要讲分治思想，以及相关的一些题目。
题单在 vJudge 上面，密码是 TNFStnfs（Taiyuan No.Five School）。
题目的权值大体反映难度，可以看自己的情况来做。
最后，欢迎大家挑战难度 ≥ 30 的题目。

Outline

「分治」，字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并.

分治算法容易理解并且强而有力。这篇讲义分为三个部分介绍，分别是：

- Part I：基础分治思想，主要介绍分治的概念、简单序列分治和主定理。
- Part II：进阶分治算法，介绍“序贡献分治”（所谓 CDQ 分治）、整体二分、线段树分治和笛卡尔树。
- Part III：树上分治，主要是点分治、点分树及其变体。

当然，“分治”这个词有时也用来表示“分类处理”。所以，这份讲义还有 Part IV：名义分治，主要介绍根号分治和衍生的定期重构算法。

以及，分治思想在各种数据结构中都有应用，这些内容就交由它们自己来介绍吧。

目录

基本方法

分治算法的核心思想就是「分而治之」。

大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

基本方法

分治算法的核心思想就是「分而治之」。
大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

显然，分治的关键在于如何良好的分解。具体如何良好分解，我们需要一些练习才能得知，但显然它必须满足一些基本要求：一是子问题必须相互独立，所谓独立，指的是处理这个小规模问题的时候只要知道小规模内的信息即可；二是分解必须有限，也就是最后必须可以到一个边界，且边界很好处理。
可以看到，分治和递归非常类似。

目录

排序

考虑一个非常经典的应用，排序。显然排序问题是满足分治的使用条件的（可以分解，分解后子问题独立）。
快速排序和归并排序都基于分治思想。

归并排序

算法步骤：

1. 分解：从中点将数组分成左右两半。
2. 解决：递归排序左右子数组。
3. 合并：将两个有序子数组合并为一个有序数组。

归并排序直接按下标切分问题，把难点放在了合并。

归并排序

考虑合并，注意性质。实际上就是合并两个有序序列。这其实是容易的。考虑不断取出最小值，最小值一定在某个序列的开头。具体地：

- 使用两个指针分别指向左右子数组的起始位置。
- 每次比较两指针所指元素，将较小的放入辅助数组，并移动该指针。
- 当某一子数组耗尽，将另一子数组剩余元素直接复制到辅助数组。
- 最后将辅助数组写回原数组的对应区间。

归并排序

考虑合并，注意性质。实际上就是合并两个有序序列。这其实是容易的。考虑不断取出最小值，最小值一定在某个序列的开头。具体地：

- 使用两个指针分别指向左右子数组的起始位置。
- 每次比较两指针所指元素，将较小的放入辅助数组，并移动该指针。
- 当某一子数组耗尽，将另一子数组剩余元素直接复制到辅助数组。
- 最后将辅助数组写回原数组的对应区间。

由于层数是 $O(\log n)$ 级别的，时间复杂度 $O(n \log n)$。

快速排序

快排通过一个基准数将序列分为两个较短的序列，具体步骤如下：

1. 选基准：从数组中选择一个元素 x
2. 分区：重新排列数组，使得：左边元素 $\leq x$，右边元素 $> x$，中间放选取的元素
3. 递归排序：对左右子数组递归执行上述过程

快速排序

快排通过一个基准数将序列分为两个较短的序列，具体步骤如下：

1. 选基准：从数组中选择一个元素 x
2. 分区：重新排列数组，使得：左边元素 $\leq x$，右边元素 $> x$，中间放选取的元素
3. 递归排序：对左右子数组递归执行上述过程

一个关键的问题是：选基准中，我们要选择哪个元素。我们考虑随机选一个数，一件很好的事情是，可以证明，这种方法在随机数据下的平均复杂度是 $O(n \log n)$。

快速排序

但考虑这样一个数组，整个数组全部相同，这样，当问题大小为 n 的时候，有 $n-1$ 个数都会归为左边，然后问题变成了处理规模为 $n-1$ 的问题。这样，时间复杂度会退化为 $O(n^2)$。

快速排序

但考虑这样一个数组，整个数组全部相同，这样，当问题大小为 n 的时候，有 $n-1$ 个数都会归为左边，然后问题变成了处理规模为 $n-1$ 的问题。这样，时间复杂度会退化为 $O(n^2)$。

不过处理这种问题是容易的，我们朴素的算法是“左边元素 $\leq x$，右边元素 $> x$，中间放选取的元素”，只要改成“左边元素 $< x$，右边元素 $> x$，中间放 $= x$ 的元素”就好了，中间元素不会再管，也就无所谓了。这个简单的优化有个很帅的名字，叫「三路快速排序」。

结论

在任意数组下，三路快速排序的平均复杂度都为 $O(n \log n)$。

快速排序 - 静态找第 k 大

考虑找第 k 大的数。最简单的方法是先排序，然后直接找到第 k 大的位置的元素。这样做的时间复杂度是 $O(n\log n)$，对于这个问题来说很不划算。

我们可以借助快速排序的思想解决这个问题。考虑快速排序的划分过程，在快速排序的「划分」结束后，数列 $A_p \cdots A_r$ 被分成了 $A_p \cdots A_q$ 和 $A_{q+1} \cdots A_r$，此时可以按照左边元素的个数 $(q-p+1)$ 和 k 的大小关系来判断是只在左边还是只在右边递归地求解。

和快速排序一样，该方法的时间复杂度依赖于每次划分时选择的分界值。如果采用随机选取分界值的方式，可以证明在期望意义下，程序的时间复杂度为 $O(n)$。

快速排序 & 归并排序

快速排序、归并排序的常数远优于堆排序等其他 $O(n \log n)$ 的排序算法。而由于快速排序不需要辅助数组，且空间访问连续，其时间常数也优于归并排序。这也是为什么 std::sort 使用快速排序而不是其他。

快速排序 & 归并排序

快速排序、归并排序的常数远优于堆排序等其他 $O(n \log n)$ 的排序算法。而由于快速排序不需要辅助数组，且空间访问连续，其时间常数也优于归并排序。这也是为什么 std::sort 使用快速排序而不是其他。

实际上，std::sort 使用的算法叫“内省排序”，也就是限制快排深度为 $\log n$，超过即使用堆排序。这保证了 $O(n \log n)$ 的最差复杂度，也保证了较低的常数。
此外，如果数组长度 ≤ 16，std::sort 直接使用插入排序。

Karatsuba 算法

分治还能实现快速的多位数乘法，不需要任何深刻的数学知识，而且实现简单。这种算法称为 Karatsuba 算法，是 Karatsuba 在他读本科的时候发现的。

Karatsuba 算法

分治还能实现快速的多位数乘法，不需要任何深刻的数学知识，而且实现简单。
这种算法称为 Karatsuba 算法，是 Karatsuba 在他读本科的时候发现的。
对于两个 N 位数（或 N 项的多项式）x 和 y，将它们各分成两半：

$$x = a \cdot B^m + b, \quad y = c \cdot B^m + d$$

其中 B 是基数（如 10 或 2），$m = N/2$。传统计算需要 4 次乘法：ac, ad, bc, bd，但注意到：

$$x \cdot y = ac \cdot B^{2m} + [(a+b)(c+d) - ac - bd] \cdot B^m + bd$$

只需要 3 次乘法：ac, bd, $(a+b)(c+d)$。
然后通过加减得到中间项。

Karatsuba 算法

考虑时间复杂度。

感性理解，由于加法很快，或者说“傍着”乘法出现，只要考虑乘法次数即可。

设 $T(n)$ 表示**乘法次数**，则有一个大问题分为三个子问题，有：

$T(n) = 3T(N/2) = 3^2 T(N/4)... = 3^k T(1)$。其中 $2^k = n$。

由是，$k = \log_2 n$，有：

$$3^k = 3^{\log_2 n} = 3^{\frac{\log_3 n}{\log_3 2}} = n^{\frac{1}{\log_3 2}} = n^{\log_2 3}$$

得时间复杂度 $O(n^{\log_2 3}) \approx O(n^{1.585})$。

Karatsuba 算法

严谨一点，考虑每一层的复杂度和这一层级需要执行几次。

对于第 i 层，规模为 $n/(2^i)$，复杂度为 $O(\frac{n}{2^i})$，需要调用 3^i 次，总的复杂度为 $O(n(\frac{3}{2})^i)$。

求和，有：

$$n + 1.5n + (1.5)^2 n + \cdots + (1.5)^{\log_2 n - 1} n = O(n^{\log_2 3})$$

目录

主定理

上面的时间复杂度分析可以形式化为这样的问题。设**操作次数**为 $T(n)$，处理一个大问题需要三个小问题和几次次 $O(n)$ 的加法，所以有 $T(n) = 3T(n/2) + O(n)$，我们想要知道这个「递归方程」的渐进解。

主定理

上面的时间复杂度分析可以形式化为这样的问题。设**操作次数**为 $T(n)$，处理一个大问题需要三个小问题和几次次 $O(n)$ 的加法，所以有 $T(n) = 3T(n/2) + O(n)$，我们想要知道这个「递归方程」的渐进解。

主定理可以解决这样的问题。

主定理

主定理

对于递归式 $T(n) = a\,T(\frac{n}{b}) + f(n)$，**其中** $a \geq 1,\ b > 1$。

1. **若** $f(n) = O(n^{\log_b a - \epsilon}) \quad (\epsilon > 0)$ **则**:
   $\Rightarrow T(n) = \Theta(n^{\log_b a})$
2. **若** $f(n) = \Theta(n^{\log_b a} \log^k n) \quad (k \geq 0)$ **则**:
   $\Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. **若** $f(n) = \Omega(n^{\log_b a + \epsilon})$ **且** $af(n/b) \leq cf(n)\ (c < 1)$ **则**:
   $\Rightarrow T(n) = \Theta(f(n))$

其中 $af(n/b) \leq cf(n)\ (c < 1)$

主定理

主定理

对于递归式 $T(n) = a\,T(\frac{n}{b}) + f(n)$，其中 $a \geq 1,\ b > 1$。

1. $f(n) = O(n^{\log_b a - \epsilon}) \quad (\epsilon > 0) \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a} \log^k n) \quad (k \geq 0) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ 且 $af(n/b) \leq cf(n)\ (c < 1) \Rightarrow T(n) = \Theta(f(n))$

看起来非常抽象，实际上，这个定理就是在比较 $aT(\frac{n}{b})$ 和 $f(n)$ 谁占主导地位。三种种情况分别是 $f(n)$ 比较小，$aT(\frac{n}{b})$ 和 $f(n)$ 差不多大，$f(n)$ 比较大。而比较的依据就是就是看 $f(n)$ 和 $n^{\log_b a}$ 的大小。

主定理

主定理

对于递归式 $T(n) = a\,T(\frac{n}{b}) + f(n)$，其中 $a \geq 1,\ b > 1$。

1. $f(n) = O(n^{\log_b a - \epsilon}) \quad (\epsilon > 0) \Rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a} \log^k n) \quad (k \geq 0) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ 且 $af(n/b) \leq cf(n)\ (c < 1) \Rightarrow T(n) = \Theta(f(n))$

看起来非常抽象，实际上，这个定理就是在比较 $aT(\frac{n}{b})$ 和 $f(n)$ 谁占主导地位。
三种种情况分别是 $f(n)$ 比较小，$aT(\frac{n}{b})$ 和 $f(n)$ 差不多大，$f(n)$ 比较大。而比较的依据就是就是看 $f(n)$ 和 $n^{\log_b a}$ 的大小。
如果你对 O, θ, Ω 这些符号不熟悉，可以简单理解为 $O(T(n))$ 是长远来看 $\leq kT(n)$，θ 是差不多，Ω 是大于；当然，在 OI 阶段，你甚至可以都看成 O。

主定理-例子

看一些例子：

1. **归并排序**：$T(n) = 2\,T(n/2) + n$
2. **二分查找**：$T(n) = T(n/2) + 1$
3. **某种分治**：$T(n) = 2\,T(n/2) + n^2$

主定理-例子

看一些例子：

1. **归并排序**：$T(n) = 2T(n/2) + n$
2. **二分查找**：$T(n) = T(n/2) + 1$
3. **某种分治**：$T(n) = 2T(n/2) + n^2$

解答：

1. $n^{\log_2 2} = n$，$f(n) = n$，**情况** 2（$k = 0$）$\Rightarrow \Theta(n \log n)$
2. $n^{\log_2 1} = 1$，$f(n) = 1$，**情况** 2（$k = 0$）$\Rightarrow \Theta(\log n)$
3. $n^{\log_2 2} = n$，$f(n) = n^2 = \Omega(n^{1+\epsilon})$，**正则条件成立**，**情况** 3 $\Rightarrow \Theta(n^2)$

主定理-Karatsuba 算法

回看之前的 Karatsuba 算法：

递归式：$T(n) = 3T(n/2) + O(n)$ 其中 $a = 3,\ b = 2,\ f(n) = O(n)$.

比较 $f(n)$ 与 $n^{\log_b a}$：

$$f(n) = O(n) = O\left(n^{\log_2 3 - \epsilon}\right), \quad \epsilon = \log_2 3 - 1 \approx 0.585 > 0$$

即 $f(n)$ 多项式小于 $n^{\log_b a}$。

主定理情况 1：

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 3}\right)$$

* 主定理证明

递归式：$T(n) = aT(n/b) + f(n)$，$a \geq 1$，$b > 1$，**设** $n = b^k$。

递归树结构：

- **第** 0 **层（根）**：1 **个问题，规模** n，**代价** $f(n)$。
- **第** 1 **层**：a **个问题，每个规模** $\frac{n}{b}$，**总代价** $a \cdot f\left(\frac{n}{b}\right)$。
- **一般地，第** i **层**：a^i **个问题，每个规模** $\frac{n}{b^i}$，**总代价** $a^i f\left(\frac{n}{b^i}\right)$。
- **叶子层**：$i = \log_b n$，**节点数** $a^{\log_b n} = n^{\log_b a}$，**每个叶子代价** $\Theta(1)$，**总叶子代价** $\Theta(n^{\log_b a})$。

总代价：

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

* 主定理证明

情况 1：存在 $\epsilon > 0$ 使 $f(n) = O\left(n^{\log_b a - \epsilon}\right)$。

- 则

$$a^i f\left(\frac{n}{b^i}\right) = O\left(a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}\right) = O\left(n^{\log_b a - \epsilon} \cdot (b^\epsilon)^i\right)$$

- 求和 $\sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i = O(n^\epsilon)$，故总和为 $O(n^{\log_b a})$。
- 加上叶子项 $\Theta(n^{\log_b a})$，得 $T(n) = \Theta(n^{\log_b a})$。

情况 2：$f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$，$k \geq 0$。

- 代入得

$$a^i f\left(\frac{n}{b^i}\right) = \Theta\left(n^{\log_b a} \cdot i^k\right)$$

- 求和 $\sum_{i=0}^{\log_b n - 1} i^k = \Theta(\log^{k+1} n)$。

* 主定理证明

情况 3：存在 $\epsilon > 0$ 使 $f(n) = \Omega\big(n^{\log_b a+\epsilon}\big)$，且存在 $c < 1$ 使

$$af\Big(\frac{n}{b}\Big) \le cf(n)$$

正则条件推出 $a^i f\big(\frac{n}{b^i}\big) \le c^i f(n)$。

- 求和 $\sum_{i=0}^{\log_b n-1} c^i \le \frac{1}{1-c}$ 有界。
- 故 $\sum_{i=0}^{\log_b n-1} a^i f\big(\frac{n}{b^i}\big) = O(f(n))$。
- 同时，$f(n) = \Omega(n^{\log_b a+\epsilon})$ 意味着叶子项 $n^{\log_b a}$ 可被 $f(n)$ 吸收。
- 因此 $T(n) = \Theta(f(n))$。

关于正则条件

注意到：

小结论

当主定理情况 3 的条件从 $f(n) = \Omega(n^{\log_b a+\epsilon})$ 加强为 $f(n) = \Theta(n^{\log_b a+\epsilon})$ 时，正则条件未必成立，但一定有 $T(n) = \theta(f(n))$。

更方便使用。

另外，如果你想出书的话，一定要记得这里是 Ω。否则，写成 θ 还指着显然成立的式子说“我们不能这样证明，这样过不去，考虑引入正则条件然后换一个方法证明”，可就不太好了。

思考题

B - 黑箱排序

有一个长度为 n 的整数序列 $x_1, x_2, \ldots, x_n$，给定序列长度 n，你自始至终完全不知道序列的内容。你唯一能进行的操作是调用一个函数 $f(i,j)$ $(1 \le i,j \le n)$，它将比较 x_i 和 x_j；若 $x_i > x_j$，则交换两个元素的值（即调用后 $x_i \le x_j$）；否则什么也不做。注意：调用结束后，你无法得知比较的结果，也无法得知是否发生了交换。

构造一系列 f 的操作，使得执行之后最终整个序列非降序排列。请设计一个算法，使得调用的总次数不超过 10^7。

$n \le 10^5$。

目录

序列贡献分治

所谓「序列贡献分治」，一般用于高效计算所有子区间贡献之和的算法思想。它要解决的基本问题是形如

$$\sum_{l=1}^{n}\sum_{r=l}^{n} f(l,r)$$

的区间统计问题，即计算所有子区间的贡献之和。

序列贡献分治

所谓「序列贡献分治」，一般用于高效计算所有子区间贡献之和的算法思想。它要解决的基本问题是形如

$$\sum_{l=1}^{n}\sum_{r=l}^{n} f(l,r)$$

的区间统计问题，即计算所有子区间的贡献之和。

考虑典型的分治流程：假设当前处理区间为 $[l,r]$，取中点（或者其他有良好性质的点）m。这样贡献分为三部分：左右端点都在 $[l,m)$ 内，左右端点都在 $[m,r)$ 内、左端点在 $[l,m)$、右端点在 $[m,r)$ 。前两种分别成为规模减半的子问题，递归处理即可。

至于第三部分，用好“跨过中点的区间一定是个左半区间的后缀拼上一个右半区间的前缀”，或利用双指针，或依次计算每个左/右端点的答案。

例题 C

C - Ada and Greenflies

给出一个序列 $a_1 \sim a_n$，**求**：

$$\sum_{l=1}^{n}\sum_{r=l}^{n}\gcd a_i$$

$n \le 3 \times 10^5$，**值域大小** $|V| \le 10^6$。

例题 C 题解

考虑对序列进行分治。设当前分治区间为 $[L, R]$，中点 $mid = \lfloor \frac{L+R}{2} \rfloor$。考虑如何计算跨过中点的贡献。

注意到固定右端点 r 后，区间的 $\gcd$ 只会有 $\mathcal{O}(\log |V|)$ 种。因为当左端点左移 1 步时，$\gcd$ 要么不变，要么至少除以 2。

一个跨过中点的区间一定是个左半区间的后缀拼上一个右半区间的前缀。因此考虑预处理左半区间所有后缀以及右半区间所有前缀的 $\gcd$ 的出现次数，维护 cl, cr 两个桶。

最后枚举左半区间后缀的 gl，右半区间前缀的 gr，显然贡献为 $\gcd(gl, gr) \cdot cl_{gl} \cdot cr_{gr}$。

例题 C 题解

考虑对序列进行分治。设当前分治区间为 $[L, R]$，中点 $mid = \lfloor \frac{L+R}{2} \rfloor$。考虑如何计算跨过中点的贡献。

注意到固定右端点 r 后，区间的 $\gcd$ 只会有 $\mathcal{O}(\log |V|)$ 种。因为当左端点左移 1 步时，$\gcd$ 要么不变，要么至少除以 2。

一个跨过中点的区间一定是个左半区间的后缀拼上一个右半区间的前缀。因此考虑预处理左半区间所有后缀以及右半区间所有前缀的 $\gcd$ 的出现次数，维护 cl, cr 两个桶。

最后枚举左半区间后缀的 gl，右半区间前缀的 gr，显然贡献为 $\gcd(gl, gr) \cdot cl_{gl} \cdot cr_{gr}$。

粗略分析时间复杂度：对于长度为 N 的分治区间，求所有后缀/前缀的 $\gcd$ 时，每当发现新的数字不能除尽 $\gcd$ 的时候重算，总的重算代价为 $O(\log^2 |V|)$，枚举的代价为 $O(n)$。考虑到 n 比较大的时候前者忽略，n 比较小的时候卡不满，姑且讲分治合并复杂度看作 $O(n + \epsilon)$，总的复杂度 $O(n \log^{1+\epsilon} n)$。

例题 D

D - Imbalanced Array

给出长度为 n 的序列 $a_1 \sim a_n$，求：

$$\sum_{l=1}^{n}\sum_{r=l}^{n}\left(\max_{i=l}^{r} a_i - \min_{i=l}^{r} a_i\right)$$

$n \leq 10^6$。

例题 D - 题解

将式子拆开，分别求解最大值和最小值。以最大值为例，分治求解。设当前区间 $[L, R]$，中点 $mid = \lfloor (L + R)/2 \rfloor$。计算跨过中点的贡献。

枚举左端点，维护右半前缀最大值 $pre_x = \max_{u=mid+1}^{x} a_u$ 及其前缀和 sum_x。从左往右枚举左端点 i（从 mid 向左），维护当前最大值 $mx = \max_{u=i}^{mid} a_u$。存在分界点 k（单调不降），使得：

- 当 $j \in (mid, k)$ 时，区间 $[i, j]$ 的最大值为 mx，贡献为 $(k - 1 - mid) \cdot mx$；
- 当 $j \in [k, r]$ 时，最大值为 pre_j，贡献为 $sum_r - sum_{k-1}$。

时间复杂度 $O(n \log n)$。

例题 D - 题解

将式子拆开，分别求解最大值和最小值。以最大值为例，分治求解。设当前区间 $[L, R]$，中点 $mid = \lfloor (L+R)/2 \rfloor$。计算跨过中点的贡献。

枚举左端点，维护右半前缀最大值 $pre_x = \max_{u=mid+1}^{x} a_u$ 及其前缀和 sum_x。从左往右枚举左端点 i（从 mid 向左），维护当前最大值 $mx = \max_{u=i}^{mid} a_u$。存在分界点 k（单调不降），使得：

- 当 $j \in (mid, k)$ 时，区间 $[i, j]$ 的最大值为 mx，贡献为 $(k-1-mid) \cdot mx$；
- 当 $j \in [k, r]$ 时，最大值为 pre_j，贡献为 $sum_r - sum_{k-1}$。

时间复杂度 $O(n \log n)$。

Hint

分治不是本题的最优解法，单调栈可以做到 $O(n)$。

* 例题 E

E - Min + Sum

给定两个长度为 N 的整数序列 $A = (A_1, A_2, \ldots, A_N)$ 和 $B = (B_1, B_2, \ldots, B_N)$。
请输出满足 $1 \leq l \leq r \leq N$ 的整数对 (l, r) 的个数，使得下列条件成立：

$$\min\{A_l, A_{l+1}, \ldots, A_r\} + (B_l + B_{l+1} + \cdots + B_r) \leq S$$

$N \leq 2 \times 10^5$

* 例题 E 题解

考虑分治。设当前分治区间为 $[l, r]$，中点 $mid = \lfloor \frac{l+r}{2} \rfloor$。

统计跨过中点的区间个数。枚举左端点 i，记录 $s = \sum_{x=i}^{mid} b_x$ 和 $minn = \min_{y=i}^{mid} a_y$。

求合法右端点 j 的个数。

预处理右半部分：

$$sum_j = \sum_{u=mid+1}^{j} b_u, \qquad pre_j = \min_{v=mid+1}^{j} a_v.$$

将右端点 j 分为两类：

- 若 $pre_j \geq minn$，则条件为 $s + sum_j + minn \leq S$，即 $sum_j \leq S - s - minn$。
- 若 $pre_j < minn$，则条件为 $s + sum_j + pre_j \leq S$，即 $sum_j + pre_j \leq S - s$。

* 例题 E 题解

从右往左枚举左端点 i，存在分界点 k（单调不降）使得：

- $j \in (mid, k)$ 时属于第一类；
- $j \in [k, r]$ 时属于第二类。

用平衡树 t_1, t_2 分别维护：

- (mid, k) 中所有 sum_j 的集合；
- $[k, r]$ 中所有 $sum_j + pre_j$ 的集合。

当 k 右移时（原 k 变为新集合的一部分），在 t_1 中插入 sum_k，在 t_2 中删除 $sum_k + pre_k$。查询操作即为统计集合中不超过某定值的元素个数（可用 gnu_pbds::tree 实现）。

时间复杂度 $O(n \log^2 n)$，空间复杂度 $O(n)$。

* 最值分治 - 例题 E 另解

考虑按最小值分治。

统计跨过最小值位置 m 的满足条件的区间个数。遍历 $[l, m]$ 与 $[m, r]$ 中较小的区间，利用启发式合并的性质保证复杂度。设 $s_i = \sum_{j=1}^{i} b_j$。

- 若 $m - l \le r - m$: 遍历左端点 $i \in [l, m]$，则右端点 $j \in [m, r]$ 满足 $a_m + \sum_{x=i}^{j} b_x = a_m + s_j - s_{i-1} \le S$, 即 $s_j \le S + s_{i-1} - a_m$。二分最大的 j_0，则 $j \in [m, j_0]$ 都满足条件，答案加上 $j_0 - m + 1$。
- 若 $m - l > r - m$: 遍历右端点 $i \in [m, r]$，则左端点 $j \in [l, m]$ 满足 $a_m + \sum_{x=j}^{i} b_x = a_m + s_i - s_{j-1} \le S$, 即 $s_{j-1} \ge s_i + a_m - S$。二分最小的 j_0，则 $j \in [j_0, m]$ 都满足条件，答案加上 $m - j_0 + 1$.

总复杂度 $O(n \log^2 n)$.

例题 F

F - Special Segments of Permutation

给定长度为 n 的排列 $p_1, p_2, \ldots, p_n$，求满足下列条件的区间 $[l, r]$ 的个数：

$$l \le r, \quad p_l + p_r = \max_{i=l}^{r} p_i, \quad n \le 2 \times 10^5.$$

例题 F 题解

采用分治算法。设当前区间为 $[L, R]$，中点 $m = \lfloor \frac{L+R}{2} \rfloor$。统计跨过中点的区间。

预处理右半部分的前缀最大值：

$$pre_x = \max_{u=m+1}^{x} p_u, \quad x \in [m+1, R].$$

从右向左扫描左端点 i（从 m 到 L），维护当前后缀最大值：

$$maxn = \max_{u=i}^{m} p_u.$$

存在一个单调不降的分界点 k，使得：

- 当右端点 $j \in (m, k)$ 时，区间 $[i, j]$ 的最大值为 $maxn$；
- 当 $j \in [k, R]$ 时，区间 $[i, j]$ 的最大值为 pre_j。

例题 F 题解

分别统计两类贡献：

- 对于 $j \in (m,k)$，条件为 $p_i + p_j = maxn$，即 $p_j = maxn - p_i$。维护桶 cnt_1 记录 (m,k) 中每个值的出现次数，贡献加 $cnt_1[maxn - p_i]$。
- 对于 $j \in [k,R]$，条件为 $p_i + p_j = pre_j$，即 $pre_j - p_j = p_i$。维护桶 cnt_2 记录 $[k,R]$ 中每个 $pre_j - p_j$ 的出现次数，贡献加 $cnt_2[p_i]$。

当 k 右移时（原位置 k 从第二类移入第一类），更新：

$$cnt_1[p_k] \leftarrow cnt_1[p_k] + 1, \qquad cnt_2[pre_k - p_k] \leftarrow cnt_2[pre_k - p_k] - 1.$$

时间复杂度 $O(n \log n)$。