算法观光之旅：语言、符号与逻辑
(Volume 1：逻辑与结构)

hotteano
南方科技大学

July 9, 2025

0.1 前言：让我们宏观地看算法

历史上许多天才为我们铺下了一段宏伟壮观的道路——这条路就是算法的历史。从欧几里得到 Knuth、Tarjan、Floyd，我们在算法的路上已经行走很远。但我们深深知道——以人类有限的年华，决无法穷竭世界上一切的算例。因为问题无穷无尽，因此我们的算法也应当与时俱进。

本书的一个目的是，让读者在算法发展的历程上观光游览，无需费尽心思考量算法内部的数学原理，这些内容在经典的教材如《算法导论（第三版）》、《计算机程序设计艺术》、《算法（第四版）》等上都能够找到。互联网如此发达，在观光以后，也应当沿着一条路走下去，细细品味一个算法的天才之处，至少对于设计算法的思想有一个更加深刻的理解。

本书分为十七个章节，前八个章节主要介绍各类算法、算法分析方法和数据结构，第九章主要介绍数理逻辑的基础知识，第十章介绍自动机理论，第十一章介绍数学算法以及组合数学知识，第十二章介绍数值优化方法，第十三章介绍常用的流程优化方法，第十四章介绍近似算法，第十五章介绍多线程算法，第十六章则是介绍复杂度理论，第十七章则是对一些当代算法的介绍。

本书主要涵盖了理论计算机（算法 + 复杂度理论）的入门内容，其中每一章都是值得专门挑出一个学期甚至一年进行深入学习的内容。希望读者在阅读本书的时候有所收获。

本书大量使用 tikz 包进行绘图，这是一个非常高效且实用的绘图包，建议读者在写作的时候也使用这个宏包。

Contents

第一章　初入算法世界：如何分析算法的复杂度？

1.1　渐进分析

对于一个复杂度函数的增长，我们采用同阶估计的想法，采用如下记号：

- $g(n) = \Theta(f(n))$。这等价于对于充分大的 n, 总有 $af(n) \leq g(n) \leq bf(n).\Theta(f(n))$ 被称为函数 $g(n)$ 的渐进紧确界。
- 类似地，$g(n) = O(f(n))$，表明对于充分大的 n, 总有 $g(n) \leq af(n)\, O(f(n))$ 被称为 $g(n)$ 的渐进上界；进一步我们定义，若这个等号取不到，那么我们记这个界为 $o(f(n))$，也就是说，这个界是不够紧的。
- 相反，我们定义 $\Omega(f(n))\, g(n)$ 的下界，$\omega(f(n))$ 为 $g(n)$ 的非紧下界，只需要将 2 中的不等号反过来就好了；

利用上述的符号我们可以简单地表达"$f(n)$ 地增长速率大概是……"。这是因为，在算法领域中，输入的规模往往是不确定的，我们所要做的仅仅是估计主要部分的增长速度，从而界定我们算法的效率；对于低阶的项，我们常常在分析的时候会选择性忽略，除非这些因素已经达到能够影响主要成分（例如背包问题中的巨大的 W 常数）。也就是说，我们在评价一个算法的时候，往往关注时间或空间增长的阶。

1.2　递归树

1.3　主方法

对于递归式 $T(n) = T(n/b) + f(n)$，我们有如下的定理确定 $T(n)$ 的增长阶：

1. 若对于 $\epsilon > 0, f(n) = O(n^{\log_b a - \epsilon})$, 则 $T(n) = \Theta(n^{\log_b n})$；【定理表明，$f(n)$ 比较小的时候，$T(n/b)$ 占主要作用】

2. 若对于 $f(n) = \Theta(n^{\log_b a})$, 则 $T(n) = \Theta(n^{\log_b a} \lg n)$;【二者同阶，那么递归树每一层的代价不可忽略】

3. 对于某个常数 $\epsilon > 0$ 有 $f(n) = \Omega(n^{\log_b a + \epsilon})$, 对于某个常数 $c < 1$ 和所有充分大的 n 都有 $af(n/b) \leq cf(n)$【事实上，这一分句等价于前一分句，因此是多余的条件】，则 $T(n) = \Theta(f(n))$.【递归树每一层的代价占主要作用】

1.4 初等概率论与随机算法分析

我们在入门阶段，一般接触的都是确定型算法，即每一步都是由算法精确描述并确定的。然而，当我们将随机数生成器引入算法中，我们就无法根据算法的每一步精确分析其时间复杂度。此时，使用现代概率论成为了我们的一条出路。在概率论中，数学期望是一种非常有效的分析指标，它能够给出一个随机过程出现某一特定结果的概率。通过对于随机过程的分析，我们能够很好地分析算法运行时间及其运行效率和准确率。

1.4.1 指示器思想

所谓指示器变量估计，指的是将次数变量转化为一系列布尔变量的等价集合。例如，某件事情发生次数的期望 $\rightarrow X_i$:i 事件是否发生，最后再把所有的变量加起来。利用期望的线性性质，$E(\sum X_i) = \sum E(X_i)$, 从而给出问题的一个概率估计。

对于一个随机算法，我们往往利用随机数生成器加入随机性，并利用这个随机数进行选择。由此我们知道，随机算法可能无法一次性解决问题，但是我们可以通过算法设计使得这个算法的正确概率尽可能的高。这种算法对于 NP 问题而言，可以在某种程度上降低复杂度，因为若我们能够得到输出，那么我们可以在多项式时间内验证答案，从而若算法也是多项式的，我们就可以在多项式时间内找到问题的一个答案。

1.5 摊还分析

摊还分析给出的是对于算法整体性能的评估，比起单独计算单一步骤的复杂度，摊还分析从算法整体的性能和效果出发，通过取平均来说明一件事情：即便一个操作组合的某个操作的复杂度很高，其最坏情况下的平均复杂度仍然是可以接受甚至极其理想的。为了整体评估算法的复杂度，我们不能简单地将每一步组合到一起，因为流程中的复杂度会随着数据结构的改变而改变。因此我们有必要引入一些有意思的解决方案：

1.5.1 聚合分析

对于一个操作序列，我们若能够证明 n 个操作在最坏情况下的复杂度为 $T(n)$，那么操作序列的均摊复杂度为 $\frac{T(n)}{n}$。这是非常自然的分析方法。

1.5.2 “核算”法（accounting method）

对于每一步，我们授予“信用”，而这一信用能够被后续的收益补充。也就是说，我们能够容忍一些前置的代价，这些代价能够让我们在后续的操作中获得收益，从而使得整个流程的复杂度趋于可接受范围。

1.5.3 势能法

对于数据结构的特定状态，我们赋予其状态函数——势能，因此我们得出均摊代价 $\hat{c} = c + \Phi_i - \Phi_{i-1}$，其中 Φ 表示势能。每一步操作会改变数据结构的势能，而这个势能差也可以被视为我们前面提到的“信用”，这些势能可以用来填补均摊代价与实际代价之间的鸿沟。我们可以对于数据结构进行观察，从而定义数据结构的势能究竟是什么（一切以实际的情况为准）。该方法使得我们可以分析任意两个步骤之间的均摊复杂度，具有较高的灵活性。

第二章　算法设计的核心思想

2.1　暴力：大力出奇迹

遍历所有答案空间是一定可行的，只要我们能够将这一遍历过程通过循环进行表示，那么我们就能够通过暴力解决这个问题。当然，大部分时候，这一个方法是极其低效的。然而，在我们分析一个算法问题的时候，不妨先想一个暴力的做法，然后再运用已知的算法对这个暴力方法进行优化，这样我们就可以分析清楚问题中一些关键的指标和数据特征。暴力方法的常规操作包括枚举、直接递归、遍历等，而递归问题往往可以通过优化为动态规划问题从而降低时间复杂度。

2.2　分治：分而治之

2.3　贪心：得过且过也能得到完美

贪心策略意指这样的策略：如果我们能够在较小规模下使得问题获得最优解决，那么对于全局我们也能够获得这样的最优解。这基于一个假设：局部最优一定能够满足全局最优。当然，这一策略并非总是奏效，因为大部分的问题并不具有局部最优性。此时我们会考虑一些特殊策略，如动态规划（Dynamic Programming）对这些问题进行搜索。当然，动态规划也是基于一定的局部最优性进行推理。贪心策略是算法设计中最为核心的一种思想，在无法求得最优解或严格求解代价过大的时候，我们也可以通过这种方法逼近最优解，从而以比较小的代价获得比较好的结果。

2.4　二分：分治的特化

什么是二分查找？二分查找是在有序集中根据特定指标对目标对象进行查找的方法，一般而言，我们需要根据数据内容的特殊性质（如，大小关系，余数关系等）在集合上给出一个划分，并以此在每次搜索过程中将搜索范围缩小一半的方法。

由于每一次我们的搜索范围都缩小一半，进而我们通过递归式 $T(n) = T(n/2) + \Theta(1)$ 不难得到这一搜索的复杂度是 $\Theta(\log n)$ 二分查找的关键是寻找分段指标从而作出一个

划分，抽象一点说，我们找出集合上的两个等价类，并将其作为一个划分进行搜索。通过观察数据特点，特别是数学关系，能够让我们更好地优化算法。这也充分表明了二分查找是一个极为灵活的算法，由于分段指标的不确定性，很大程度上二分查找需要长期练习的经验和突如其来的灵感才能够在需要使用的时候发挥作用。二分查找的步骤如下：1. 寻找划分指标（最需要灵感的一处）；2. 设置头尾指针，申请 mid（二分指针）；3. 进入 while 循环，设置终止条件：头指针 = 尾指针；4. 每次循环执行条件判断：计算 mid 指针，即头尾指针相加除以二；二分指针两侧，不符合搜索目标的划分舍去，这一侧的指针移动到 mid 处；注意，为防止无限循环，个人通常习惯左侧指针赋值的时候将其赋为 mid+1，而右侧指针习惯赋为 mid。5. 循环终止，输出头指针或尾指针指向的数据即可；

例如如下的题目：对于一个整数数组 nums，给定一个最大操作数 m，对于一个 nums[i], 单次操作允许将该整数拆分为两个整数，这两个整数的和为 nums[i]，求最终数组中的最大值的最小值。

我们这里将问题转化为：存在一个 y，使得操作数小于等于 m 时，数组中没有比 y 更大的数。这就是一个搜索问题，我们可以利用二分查找搜索这个 y。对于一个 nums[i], 将这个数分解为小于 y 的数需要 $\lfloor \frac{nums[i]-1}{y} \rfloor$ 次，这是因为我们考虑每一次分解都将这个数分解为 y 和另一个数，如此需要 $\lfloor \frac{nums[i]-1}{y} \rfloor$ 次才分解完。依照上面的算法，我们计算总操作次数，若超过，那么应当将左指针向右移动；如果没有超过，说明 y 可以更小，右指针左移。最终我们的复杂度就是 $O(n \log C)$, 其中 C 为数组中最大的数的大小。

第三章　排序：从无序到有序

3.1　朴素排序法：$O(n^2)$

这三种排序是典型的 $O(n^2)$ 复杂度排序，它们均必须对于每一个下标的元素遍历一遍，并且在循环中遍历其他元素进行比较然后将元素放到对应的位置上。其正确性可以利用循环不变式证明，此处略去，详见《算法导论》。

3.1.1　插入排序

插入排序与打牌的时候整理牌型是一致的，对于一个元素，我们顺次遍历直到遇到一个比它大或比它小的元素（取决于最终排序的大小次序），然后我们将元素插入到这个元素的左边或右边（取决于升序降序）；

3.1.2　冒泡排序

冒泡排序类似于泡泡的上升过程，我们不断比较需要排序的元素与其右侧元素的大小关系，按照升序或者降序的原则，判断是否要交换两者。

3.1.3　选择排序

选择排序采用的是寻找最大值或最小值进行排序的思路，实际上，这与堆排序的思路是一致的，但是堆排序运用了堆这种数据结构将寻找最大值的时间复杂度降低到了 $O(\log n)$. 我们后面还会看到许多维护最大值的方法，如单调队列、优先队列、单调栈等等。

3.2　分治排序法：$O(n \log n)$

3.2.1　归并排序

归并排序通常遵循两个步骤：交换 + 合并；首先比较每两个元素的大小关系，如果不符合排序顺序，那么交换二者；然后将排序好的数组两两合并，然后重复这样的比较直到只剩下一个数组；其中归并数组的时候可能会产生较大的空间复杂度，因此需要及

时释放占用的空间以进行排序；归并排序的实现通常在实现有序化和归并以后，递归调用归并排序直到只剩下一个元素；理论时间复杂度上界为 $O(nlog\ n)$ ，这是因为递归树的高度为 $log\ n$ ，并且每一层的时间复杂度为 $O(n)$ ，合并以后就得到了时间复杂度的上界。

3.2.2 快速排序

快速排序的思路一般是：选择一个数（个人喜欢选择下标为中点的数），将大于这个数的数放到右边（假设升序排列），小于这个数的数放到左边。这里，我们可以使用双指针优化这个交换过程，读者可以自行尝试。我们递归执行这个过程，每一次问题规模近似缩小一倍，递归式 $T(n) = 2T(n/2) + (n)$, 利用主定理不难求出复杂度为 $O(nlogn)$. 伪代码描述：

```
QUICK_SORT(A,p,r)
 if p<r
    x=A[r]
    i=p-1
    for j=p tp r-1
        if A[j]<=x//小于等于x，放到左边
            i=i+1
            swap(A[i],A[j])
    swap(A[i+1],A[r])
  q=i+1//上述代码叙述的是对于数组的划分部分
  QUICK_SORT(A,p,q-1)
  QUICK_SORT(A,q+1,r)//递归
```

事实上，对于上述代码还有一个优化版本，读者可以自行思考。

3.2.3 堆排序

堆是一棵完全二叉树，堆排序通常是通过维护堆的最大堆性质来排序数组。最大堆性质是指，在一个子堆中，父节点左右子节点的数均小于父节点，这样，我们保证了堆顶的元素是最大元素。在堆排序中又两个关键步骤，第一个步骤是建立最大堆并维护最大堆性质，具体而言就是建立最大堆并递归维护每一个子堆中的最大堆性质，前一步骤的时间复杂度为 $O(n)$ ，后一步骤的时间复杂度为 $O(log\ n)$.

第二步则是从数组第一个位置开始，设置一个自增变量 i 遍历数组，每一次将堆顶的元素放到第 i 个位置上（将第 i 个元素与堆顶元素交换），每一次交换完以后，维护第 i 个数到第 n 个数的最大堆性质，最终到最后一个元素，这就完成了排序。总体时间复杂度为 $O(nlog\ n)$. 需要注意，建立最大堆为 $O(n)$，但是我们不需要每次都建立最大堆，

否则将会退化为选择排序的 $O(n2)$. 我们只需要维护最大堆即可，而这是 $O(log\ n)$ 的复杂度。我们后面的分析会表明，基于比较的排序的最佳时间复杂度下界就是 $O(nlog\ n)$.

总而言之，堆排序的步骤只有两个：

1. 维护最大堆性质，为了实现这个性质，我们只需要比较父节点和两个子节点，将更大的那个子节点与父节点交换；如果父节点是最大的，那么不变；我们从下到上依次实现这一个操作，我们就可以得到一个最大堆；

2. 交换堆顶元素与第 i 个元素，并将维护最大堆的范围缩小到 i n。堆排序的稳定性不如快速排序和归并排序，因为在交换的过程中很容易因为数据特点增加许多不必要的比较和交换；因此，在常规程序设计中我们一般使用快速排序和归并排序。

3.3 决策树与统计量

3.4 特殊排序法

3.4.1 桶排序

3.4.2 希尔排序

3.4.3 基数排序

3.4.4 计数排序

计数排序通常用于比较密集的正整数排序（稀疏数据可能会导致内存爆炸），一般做法就是，通过数个 $O(n)$ 的数组遍历，将数在一个第二数组（初始化为 0）中，将与该数相同的下标加 1；然后求出前缀和；然后遍历第二数组，将第二数组中每个值的下标存入第三数组，那么第三数组就是有序的第一数组了。

计数排序并非是基于比较的排序，因此，它的时间复杂度的下限并不是 $O(n \log n)$ 但是，由于它只能排序整数，而且在求前缀和的时候实际时间复杂度可能非常巨大，因此只能用来排序比较密集的正整数数组。

3.5 拓扑排序：严格来说不是排序

3.5.1 拓扑序

在日常生活中，我们往往会需要先后做一些事情，有些事情一定要在某些事情之前做完，否则没办法进行下一件事。我们为这种顺序起了一个名，叫做拓扑序。我们将要做的事抽象为一个有向图，然后将其排成一列，并且要求，若一件事情被多件事情指向，那么指向这件事的事件必须要在被指向的事情之前完成。

3.5.2 有向图与流程排序

如何实现拓扑序的输出呢？首先，我们需要进行有向图的构建，这一件事情有很多种办法，但是关键在于，我们要记录每一个结点的入度。然后，我们对这个图进行广度优先搜索，也就是使用一个队列来模拟搜索序列。若我们访问了一个结点，那么这个结点的入度应该-1，若这个结点的入度归零，那么直接输出即可。如是我们就完成输出了有向图的某一种拓扑序。需要注意，拓扑序可能有很多种。

第四章　基本数据结构：有序结构随取随用

4.1　栈

4.1.1　栈

栈是一种经典的数据结构，它可以被看做一种只有一端开口的容器，只能从一端取出或者压入数据。我们一般将这种数据存取规则称为后进先出原则（LIFO）。

实现栈一般为运用数组 + 一个指针（栈顶指针）。

- 压入数据（Push）：在栈顶加入数据，指针 ++；
- 弹出数据（Pop）：指针–；
- 查询是否空栈：指针 ==0；

栈的关键性质就是后进先出，如果在题目或者需求中遇到类似的性质，可以考虑采用栈来模拟输入输出数据的方式。栈的关键性质使之与卡特兰数有关——该数是组合数学中一个非常重要的数列。

4.1.2　单调栈

单调栈是一种非常经典的数据结构，其关键性质在于其中的元素要么单调增，要么单调减。这一性质在查找数组中所有数右侧或者左侧第一个大于这个数的数时可以使查找算法非常高效。构建方法也非常简单，若栈单调增，则若新数大于栈顶，那么压栈，若新数小于栈顶，弹出栈顶直到小于新数，压栈。（视情况而定是弹出栈内还是等待合适的新数）

4.2 队列

4.2.1 队列

队列也是一种经典的数据结构，我们可以想象很多人在排队等电梯，那么当电梯开了以后，排在前面的人进入电梯，排在后面的人往前移动。类似这样的原则被称为先进先出原则（FIFO）。

实现队列的基本想法是用一个数组和头尾两个指针控制队列的长度。

- 弹出：头指针 ++；
- 压入：尾指针处插入数据，尾指针 ++；
- 是否为空：按照上面的实现方法，头指针 < 尾指针则非空（这里，我们初始化的时候头指针 == 尾指针）；

队列的关键性质是先进先出。这一特点在拉取和处理请求的时候非常有效。同时，这一性质也广泛应用在广度优先搜索和 spfa 算法中。

4.2.2 双端队列（deque）

4.2.3 单调队列

单调队列，即具有某种单调性的队列，如从队首到队尾数据始终保持递增或递减。为了维护这个性质，我们在添加新元素的时候（我们假设队列递减），如果元素小于等于最后一个元素，则直接添加，如果元素大于最后一个元素，则向前查找直到找到比它大的元素，删去后面所有的元素。例如，对于滑动窗口最大值问题，我们可以通过单调队列来优化时间复杂度。其实现方法一般是采用一个数组，每一次添加元素查看前面的元素是否出队，然后再考虑新元素的位置。通常使用 while(i+length<j)i++ 的形式缩短队头，再用 while(element>a[i])i–的形式考虑插入的位置，以此维护队列的性质。

4.2.4 优先队列

4.3 链表

4.3.1 链表

链表一般分为单向链表和双向链表。

我们要实现所谓的“链状结构”就必须要实现两点：前后节点可以轻易访问目标节点、插入数据不需要前后移动其他数据地址。

因此我们采用如下方式构建节点（Node）：

1. 数据（Data）：代表节点的值；

2. 指针：单向链表中，一般只有尾指针，该指针指向下一个节点；在双向链表中，存在一个指针指向上一个节点。

特别地，对于头尾节点，头节点只有指针指向下一节点，尾指针只有上一节点的指针指向它。

因此我们定义如下操作（以单向链表为例，双向链表只需要重复前后指针两次操作即可）：

1. 插入：将插入位置的前一节点的尾指针指向新节点，新节点尾指针指向后一节点。

2. 删除：将要删除节点的前一节点的尾指针指向后一指针，删除中间节点的数据和指针；

3. 遍历：从头指针开始，采用一个检测头 det，令 det=head，然后访问 head 的 pointer 就可以访问下一节点。

4.3.2 链表向前星

在 c++ 中，我们可以利用三个数组模拟邻接表，这在处理各类图论问题时有空间优势；head[u] 表示从 u 开始，第一条边的编号；next[i] 表示编号为 i 的边的下一条边，该边与 i 同一起点；to[i] 表示编号为 i 的边的终点；（可加）val[i] 为边的权。以下是第一部分伪代码：

```
head[u]=i=0;
//其余值也初始化为0即可;
ADD(u,v,w) //u,v为边的端点,
w为权值, 无权图可以删去
next[++i]=head[u];
//编号加1, 新边的next指针指向从u开始的第一条边,
表明从u开始的第一条边是新边的下一条边
head[u]=i;//将该边编号为新的i: 注意, 前面已经自增1
to[i]=v;//将边的后驱结点赋值为v
val[i]=w; //加权
//如上添加了一条有向边, 若欲添加无向边, 反过来一次即可;
```

于是我们就可以以 $O(V+E)$ 的代价存储一个图。继续，我们来看看链表向前星是如何存储一张图的？从访问的角度来看看：

```
VISIT_ADJ(u)
  for(int j=head[u];j>0;j=next[j]) print val[j],to[j];
  //分别打印边权和邻接顶点
```

这一串简洁的访问代码，我们看到，对于从 u 开始的边，我们从该节点的第一条边开始，不断令 j=next[j]，这也就使得 j 不断访问有向边的下一条邻接边，直到 j 碰到一个 0（即走不下去了）。我们注意到，ADD 操作中的 i 是每一条边的编号，这一操作使得我们只要输入结点编号 u，就可以通过 next 数组访问所有与结点连接的边，这既保证了信息的完善，又降低了储存图的空间复杂度（对于稀疏图，CFS 结构仍然能够保证算法在其上发挥同等效益的作用）。

4.4 哈希表

散列表，即所谓的哈希表，是一种储存数据并快速查找的方法。如果设计得当，那么相比数组依次查找，哈希表的平均查找时间复杂度将会降低到

哈希表实际上是通过所谓的散列函数为关键字生成一个散列值，并插入散列值对应位置的链表中，因此当我们查找一个关键字时，我们只需要访问散列值对应的链表并向后查找即可。理想情况下，如果我们能为所有的值生成独一无二的散列值，这最好不过。

但是，这将会占据巨大的储存空间，并且在大多数时候，这样的储存是低效而浪费的。因此我们通常只会选取很小一部分作为散列值的值域，并选取适当的散列函数以减少散列值碰撞（即拥有相同的散列值而需要向后遍历链表）。一个比较简单的散列函数是 $h(k)=\lfloor km \rfloor$. 同时还有 $h(k)=\lfloor kA \mod 1 \rfloor$, 这被称为乘法散列，其中 A 是一个适当的常数。散列函数的选取是非常有技巧性的，理论上说，我们通过期望可以最小化哈希值碰撞的概率。

4.5 二叉搜索树

4.5.1 二叉搜索树的构造

当我们需要更好地维护一系列数据并且高效地查找他们的时候，可以采用二叉搜索树进行维护。寻常的二叉搜索树满足以下几个特点：每个节点具有关键字（key）、左子节点（指针）、右子节点（指针）、父节点（指针）、以及一些存储于节点的数据左关键字小于右关键字；如果不存在父节点或子节点，那么其指针设置为 null。我们通常用多个数组模拟这个树状结构，也就是说，首先我们需要一个数组，设他们关键字为 1，2，

3...，然后我们构造左子节点数组：每一个关键字的指针指向另一个关键字，那么我们就可以通过 p=l[key] 的形式访问左子节点的关键字，进而访问其中的数据。右子节点、父节点数组相同。

4.5.2 二叉树的遍历

我们遍历二叉树通常采用递归算法，按照不同的顺序（可以分为先序、中序、后序遍历）对整个二叉树进行遍历。伪代码如下：

```
INORDER TREE WALK(x)
  if x!= null
      INORDER TREE WALK(x.left)
      print x.key
      INORDER TREE WALK(x.right)
```

上述代码描述了中序遍历是如何进行的：我们总是先访问左边的节点，然后再去访问右侧的节点，递归保证了其顺序的正确性。

4.5.3 二叉树的插入与删除

为了执行插入和删除两大操作，我们需要编写 Insert 和 Delete 两个算法，并且需要保证二叉搜索树的性质：左关键字小于右关键字。因此插入算法就呼之欲出了：设我们要插入节点 z，其关键字为 v，我们需要寻找到满足关键字 v 的位置，就要从树根开始遍历，然后根据二叉树的性质向左或者向右递归，寻找适合 z 的位置。伪代码如下：

```
INSERT(T,z)
  y=null
  x=T.root //从根开始遍历
  while x!= null(树非空)
      y=x //储存x的值
      if z.key<x.key
      //若插入节点的关键值小于x的关键字，
      那么由于左子树更小，z一定位于左子树
          x=x.left //访问左子树
      else
          x=x.right
  z.p=y
  //z的父节点一定是它前面的那个节点，也就是y键值对应的节点
  if y==null
      T.root=z //树空则根节点为z
```

```
elseif z.key<y.key
    y.left=z
    //若x的上一个键值大于它，那么将x放在它的左边
else
    y.right=z
    //若x的上一个键值小于它，那么将x放在它的右边
```

4.6 红黑树

4.6.1 红黑树

4.6.2 红黑树的操作

4.6.3 AVL 树

4.7 B 树

4.7.1 B 树结点的构造

4.7.2 插入、查找

4.7.3 split 操作

4.8 VEB 树

4.9 并查集

4.9.1 并查集

并查集是一种搜索策略，当我们的搜索范围比较模糊，或者具有明显的从属关系的时候，可以考虑并查集优化。首先，我们初始化 n 个结点，每个结点具有关键字和指针，事先让每个指针指向自己。

4.9.2 路径压缩

然后，我们考虑“并”操作，将具有关系的结点连接起来。然后是“查”，我们将需要查询关系的两个结点分别向上递归查找父结点，若相同，那么考虑“路径压缩”，也就是将查询路上所有的结点的父结点递归设置为根节点，如此，我们将整个树的深度压

缩到了 2 层，这使得我们的多次查询时间复杂度大大降低。以下是一段路径压缩代码示例：

```
int search(int x){  //f[x]表示x的父结点
  if(x == f[x]){
    return x
  }
  else{
      f[x] = search(f[x]);
      return f[x]
  }
}

//三目表达式简化版本
int search(int x){
  return x==f[x]? x : (f[x] = search(f[x]);)
}
```

4.9.3 按秩合并

同时，我们可以考虑“按秩合并”：一开始，所有的结点秩为 1，将秩小于等于一个集合的另一个集合向秩比较大的集合合并（也就是将较小集合的根节点指向秩大的集合的根节点），如此我们就可以进一步将查询的时间复杂度降低到反阿克曼函数级别（Andrew Yao,1985）。

4.10　线段树

4.10.1　线段树的构造

4.10.2　懒标记

4.10.3　线段树的查找

4.10.4　线段树的维护

4.11　斐波那契堆

4.11.1　斐波那契堆的构造

4.11.2　插入与懒操作

4.11.3　查找、合并、Decrease Key

4.11.4　提取最小子结点并动态更新

4.11.5　时间复杂度：摊还分析

4.12　珂朵莉树

第五章　搜索：寻找戈多

5.1　深度优先搜索

对于一个二叉（或者多叉）搜索树，按照根-叶的顺序，一直增加深度直到搜索到叶；利用一个数组记录该节点已经被搜索过，然后开始回溯（即，返回上一个父节点检测是否有叶节点，若没有，继续向上回溯直到有新的分支），然后在新的分支重复搜索过程。深度搜索一般应用于搜索对应序列的过程或者一些寻路过程。例如，在 Trie 树前缀匹配中，通常采用深度优先搜索对字符串的前缀进行匹配。深度优先搜索通常利用一个栈模拟搜索序列，已搜索的弹出，未搜索的压入栈。以下是一个简单的示意：

```
DFS(i)
  if(i == END) return; // 设置递归条件
  //中间可以添加需要的代码
  d[i] = true;//记录已经搜索过
    for all i.next: // 查找所有子结点
      DFS(i.next); //访问下一个
  d[i] = false;//回溯
```

5.2　广度优先搜索

对于一个二叉（或者多叉）搜索树，按照逐层搜索的顺序搜索每一个分枝的合法性。如果存在一个判断条件，使得某一条路的合法性是可判断的，那么立刻终止该搜索的进程，称为剪枝。同样，深度优先搜索也能够通过剪枝对搜索过程进行剪枝，对于数据合法性比较稀疏并且可判断非法的问题，能够很好地优化时间复杂度。

广度优先搜索一般应用于地图的最短路搜索，例如在最大流问题中，广度优先搜索会发挥很大的作用。

5.3　A^* 与启发式搜索

第六章　动态规划：具有记忆的探索者

6.1　动态规划思想

6.1.1　动态规划是什么？

动态规划，即 DP（Dynamic Programming）实际上是“有备忘录的搜索”。也就是说，我们将已有的搜索结果记录在一个空间中，在搜索的过程中，通过已有的搜索结果计算新的结果。这与我们高中曾经学过的数列递推类似。事实上，我们在动态规划中也将使用类似的方法，即通过找到对应的“状态转移式”得到最终结果。

6.1.2　状态转移

动态规划中最关键的两个问题是：状态是什么？状态如何转移？

所谓状态，就是待优化目标函数在特定条件下的解。具体而言，比如我们在 CYK 算法中（下面会讲到），P[i][j] 就是字符串从 i 开始长度为 j 的字符串能否被 G 产生，其状态为布尔值。当然，在钢条切割问题中，这个就变成了钢条的总价值；在 01 背包中，就变成了物品的总价值。但是无论如何，如何定义状态，是一个非常需要灵感和经验的事情。

通常而言，对于连续切割问题（如钢条切割），我们采用的是一维动态规划，即仅有一个限定条件；对于离散选择问题，我们通常采用二维动态规划，即具有两个限定条件。限定条件的选择，必须要让最后限定出的状态能够成为更大问题的子答案，进而才能够进行递推优化。

若我们已经找到了合适的状态，那么就需要探索合适的递推关系了。

选择一个合适的递推公式需要首先观察前一状态的可能性：1. 简单递增：直接相加；2. 竞争最大值：计算两种可能，取最大值；3. 观察最优子结构，推导考察对象的性质，对不同情况进行分类（这一步是比较玄学的，需要细致观察与证明）；

对不同情况进行分类，运用不同的递推式，就可以设计出一个动态规划算法。至于这个递推关系具体如何找，可以参考以下的经典案例。通常来说，题目通常只会从这些经典的问题演变过来，而不会直接交给你一个 open question 进行求解，因为很多问题的确没有合适的动态规划算法进行求解。

6.2 经典动态规划类型

6.2.1 背包 dp

我们经常会处理给定操作次数上界求最大收益的问题。类似的问题被称为 0-1 knapsack（0-1 背包问题）。其特点是每个物品（每个操作）只能进行一次，能够进行多次的问题被称为完全背包问题。我们设想这样一个问题，有若干物品，重量分别为 w[i]，价值分别为 p[i]，背包只能装下重量为 W 的物品，如何装才能获得最大价值？

这里，我们立刻想到设一个二维 dp 数组，其中 dp[i][j] 表示：从前 i 个物品里面选取，最大重量为 j 时，背包中的最大价值。由此我们不难得到转移方程

$$dp[i][j] = \begin{cases} \max(dp[i][j], dp[i-1][j-w[i-1]] + p[i-1]) & , choose\ item[i] \\ dp[i-1][j] & , not\ choose\ item[i] \end{cases} \tag{6.1}$$

我们来详细解释一下转移方程的正确性：首先，我们需要一个最大值，因此我们使用了 max 函数求最大值；其中 dp[i][j] 是已经求出的暂定最大值，后一项是需要计算的最大值。其中，我们观察到后一项指的是：拿了 i-1 个物品，重量为 j-w[i-1] 时，拿上重量为 w[i-1], 价值为 p[i-1] 的物品的总价值。我们可以将上述转移方程合并：

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w[i-1]] + p[i-1]) \tag{6.2}$$

需要注意，w[i-1] 表示第 i 个物品的重量，因为我们数组的下标从 0 开始。p 数组同理。我们还有一个一维的优化版本（C++）：

```
int knapsack_1D(
        vector<int>& weights,
        vector<int>& values, int W
    ) {
    int n = weights.size();
    vector<int> dp(W + 1, 0);
    for (int i = 0; i < n; ++i) {
        for (int w = W; w >= weights[i]; --w) {
            //逆序遍历保证每一个物品只选择一遍
            dp[w] = max(dp[w],
```

```
                        dp[w - weights[i]] + values[i]
                    );
            }
        }
        return dp[W];
}
```

在上述代码中，我们观察到 i 实际上对于递推并无贡献，因此我们考虑把 i 优化出数组，从而我们得到了一个一维递推关系。

我们注意到，尽管上述算法似乎是 $O(nW)$ 的，也就是说，它看起来像一个多项式时间算法，但是事实上，W 是一个非常大的数，它随着物品数目的增多（也就是 n 的变化），选与不选造成 W 的大小以 2 为底指数变化，因此实际上在现有算法基础下，该问题是一个 NP 问题。

6.2.2 区间 dp

我们有时需要处理一些区间上两两合并使得目标函数最大的问题。

考虑如下合并问题（NOI1995 简易版，原版为环形）：在某一段路上有若干堆石头，每一堆数量不同，相邻的石堆可以互相合并。每当合并一堆石头，可以获得两堆石头数目之和的分数。求在整条路上合并后分数的最大值。这里我们考虑设置状态为 dp[i][j]，其中由于石堆的相邻性质，我们可以用区间来进行动态规划。

令 dp[i][j] 表示左端点在 i，右端点在 j 的区间上得到的最大分数，我们的目标是求出 dp[1][n]（此处假定数组从 1 开始）。显然，对于状态 dp[i][j]，我们枚举分割点，得到由哪两个区间合并而来，并且还要考虑整个区间合并时的得分，从而得到整个区间合并的总得分。从而我们得到转移方程：

$$dp[i][j] = \max(dp[i][j], dp[i][k] + dp[k+1][j] + \sum_{t=i}^{j} a_t)$$

利用前缀和我们可以用 $O(1)$ 求出后面这个和：

$$dp[i][j] = \max(dp[i][j], dp[i][k] + dp[k+1][j] + S_j - S_i), S_n = \sum_{i=1}^{n} a_i$$

如上，我们只要注意边界处理以及最后的目标值，处理好循环顺序，就可以解决类似的问题。一般地，区间 dp 的转移方程为：

$$dp[i][j] = \max(dp[i][j], dp[i][k] + dp[k+1][j] + cost_{i,j})$$

第七章　图论算法：我们所编制复杂的网啊！

7.1　图上 DFS 与 BFS

7.1.1　图上深度优先搜索

之前我们已经介绍了深度优先搜索，以下是图 G 上深度优先搜索的伪代码：

```
DFS(G)
for v in G.V
  v.color = WHITE //所有结点未被搜索
  v.pre = null //一开始所有的结点的前驱节点为空
time = 0 //全局时间戳，记录搜索的前后顺序，后面有用
for u in G.V
  if u.color = WHITE //对于未被搜索的结点进行搜索
    DFS_VISIT(G,u) //调用搜索函数
DFS_VISIT(G,u)
time += 1//每搜索一步，时间增加
u.d = time //表示结点开始被发现的时间
u.color = GRAY //用灰色表示该深搜树正在搜索
for m in G.adj[u] //对于u的所有邻接结点
  if u == WHITE
    m.pre = u //前驱节点为u
    DFS(G,m)
u.color = BLACK //将结点记录为已经搜索过
time = time + 1
u.f = time //搜索结束时的时间戳
```

7.1.2　图上广度优先搜索

我们之前已经介绍了广度有限搜索，我们给出一个图上的广度优先算法伪代码：

```
BFS(G,s)
for v in G.V-{s}
  u.color = WHITE //所有结点未被搜索
  u.d = MAX //原点s到该点的距离设置为无穷大
  u.pre = null //前驱节点设置为空
s.color = GRAY
s.d = 0
s.pre = null
Q = EMPTY //搜索队列设置为空
ENQUEUE(Q,s) //s进队
while Q != EMPTY
  u = DEQUEUE(Q)
  for v in G.adj[u]
    if v.color == WHITE
      v.color = GRAY
      v.d = u.d + 1
      v.pre = u
      ENQUEUE(Q,u)
 u.color = BLACK
```

7.2 单源最短路

7.2.1 Bellman-Ford 算法

Bellman-Ford 算法的想法时非常简单的，只要不断对图进行松弛操作，若不存在负权环，则根据每次操作距离递减，必然最后得到最短路。

```
Bellman-Ford(G,u,w):
  dis[V]={INT_MAX};
  pre[V]={null};
  for i=1 to |G.V|-1:
  //重复结点数次，若限制路长为k，此处设置k次迭代即可
    for each e(u,v)://对于每一条边，考虑松弛操作
      if(dis[v]>dis[u]+w(u,v))
        dis[v]=dis[u]+w(u,v)
  for each e(u,v):
    if(v.d>u.d+w(u,v))
      return false
```

```
        //存在负权环
        //根据抽屉原理，路径最多包含n个边，否则存在负权环
    return true; //成功搜索到最短路，直接输出dis[target]即可
```

7.2.2 Dijsktra 算法

对于一个无向图 $G=(V,E)$，设其边权为 $e=\{(a,b)|a\in E,b\in E,a\neq b\}$，并且要求权重全为非负的。我们需要设置三个集合：最优路长集合（int，初始化全为无限大，数据不太大时，在 c 中不妨使用 INTMAX）、前一点编号集合（int）、搜索状态集合（bool）我们有一个开始点和一个结束点，递归顺序如下（假设起始点为 O）：

（1）从起始点开始，搜索邻接的点，将未搜索的点更新为最优路长。

（2）计算刚加入节点 A 到临近节点 B 的距离，若路长 A-O+A-B<B-O，更新 B 的前一节点编号为 A，B-O 的距离为 A-O+A-B。

（3）到达终点时，标记终点，递归向前回溯前面点存入数组直到起始点，那么结束点的最优路长就是最短路长，数组就是最短路的各编号的序列。

```
DIJKSTRA(G,u)
//G表示图，后面叙述中，V,w分别代之G.V,G.w，
即顶点集与权重集
  dis[V] = {INT_MAX};
  pre[V] = {-1};
  S = emptySet;
  dis[u] = 0; //到自己的长度一定是0
  while(V!=emptySet)//若结点未全部探索
    a= min(dis[V]);//实际实现的时候需要使用优先队列优化
    S = S + a;
    //将搜索过的结点放入搜索集合，
    实际操作中可以用布尔数组进行标记
    for each v in G.adj[a]:
      if(dis[v] > dis[u] + w[u,v])
        dis[v] = dis[u] + w[u,v];
        pre[v] = u;
        //该操作被称为RELAX即松弛操作，
        将更大的长度更换为更小的长度以保证最优
  for each v in pre[target]:
```

```
// 使用pre是为了打印最短路径上所有结点；
//若不需要，可以不使用pre数组
  print v;
  if(pre[v]!=s)
    v = pre[v];
```

7.3 任意两点最短路

7.3.1 Floyd-Warshall 算法

Floyd-Warshall 算法是解决从任何一个点出发到另一个点最短路的集中解决方案，但是在单源最短路径中，我们尽量使用前面提到的两个算法。

```
FLOYD-WARSHALL(W)
//W为n*n的权重矩阵（w[i][j]表示i结点与j结点之间的权重，
//若不存在，则设为INT_MAX）
D[n][V][V]={INT_MAX};
n=W.rows;
for k=1 to n // 注意，k在第一重循环
  for i=1 to n
    for j=1 to n
      d[k][i][j]=min(d[k-1][i][j],d[k-1][i][k]+d[k-1][k][j]);
       //动态规划，相当于一个大型的松弛操作
return D[n] //返回任意两点之间的最短路二维数组
```

FLoyd 算法中存在三重循环，因此在单源搜索的条件下会生成大量无用的解并消耗大量时间。

7.3.2 Johnson 算法：负权消除

7.4 最小生成树

给定一个有权无向图 G(V,E), 若存在连通无环图 T(V',E') 使得图中每条边的权值 w 总和最小，那么称该图为 G 的最小生成树。为了找到这样一棵树，我们的一个基本原则是：在已有的最小生成树基础上，加入“安全边”。为了描述安全边，我们需要引入两个概念：

1 尊重。若对于任意边 e(u,v), 不存在 $u \in V$, $v \in S - V$，其中切割 C 将图 S 分割为 V 和 S-V，那么称切割 C 尊重 V.

2 轻量级边：切割的所有边中，权值最小的边被称为轻量级边。

进而我们介绍两个基于贪心的最小生成树搜索策略:

7.4.1 Prim 算法

Prim 算法：从任意根节点开始，称当前的生成树为 A，策略为从所有连接 A 和非 A 结点的边中选取一条轻量级边加入，不断检查这条边是否会导致生成环，进而选取次优的边。伪代码如下：

```
MST-PRIM(G,w,r)
for each u in G.V
  u.key=MAX
  u.pre=null
r.key=0
//选取任意根节点
Q=G.V
while Q is not empty
  u=EXTRACT-MIN(Q)
  //找出轻量级边的一个端点,其中u不在A中; 将u加入A中
  for each v in G.adj[u]
  //将每个与u邻接但不属于A的结点的key进行更新,
    if v in Q and w(u,v)<v.key
      v.pre= u //记录“从哪里来的”
      v.key=w(u,v)
      //将下一邻接结点的key直接标记为边权,
      //相当于记忆“从哪一个结点找过来的”
```

7.4.2 Kruskal 算法

Kruskal 算法：该算法首先将所有的顶点视为生成树，然后将所有的边权排序，按照升序依次检查是否属于同一棵生成树（若处于同一树，那么将生成环，因此应当抛弃），若不属于同一生成树，则将两棵树合并。（伪代码可以参考书本，比较简单）实际实现的时候需要使用并查集进行优化（并查集非常适合用于归类和判断是否为同一类的算法当中）。

7.5 最大流

7.5.1 Ford-Fulkerson 方法

7.5.2 Edmond-Karp 算法

7.5.3 最大二分图匹配

7.5.4 推送-重贴标签算法

7.6 强连通量搜索与 2-SAT 问题

7.6.1 Tarjan 算法

第八章　字符串：解析语言有迹可循

8.1　经典的字符串问题：前缀、后缀、子串

8.2　trie 树

8.2.1　基础 trie 树

字典树常用于前缀查找。该树是一个如下的结构：对于输入的每一个单词，首先将每个单词分解为单个字母，然后从第一层开始查找是否有输入单词的首字母，若有，沿着该结点向下；若没有，则新建一个分支。对于每个单词，我们需要标记其单词末尾的位置；如果存在重复，需要设计 cnt 进行计数。字典树可以将复杂的前缀匹配化简为多叉树的搜索，大大降低了前缀匹配统计的时间复杂度。

8.2.2　cow-trie 树

8.3　KMP 算法

KMP（Knuth-Morris-Pratt）算法是一种经典的字符串匹配算法，其根本思想就是记忆化搜索。至于记忆什么呢？按照学院派的讲法，就是目标字符串前缀集与后缀集的最大交集。这是什么意思？所谓前缀集合，就是从第一个字符开始数，数到第 j 个作为一个元素，直到 j 等于最后一个下标；后缀集合，就是从最后一个字符开始数，数到第 j 个下标作为一个元素，直到 j 为第一个下标。

我们不关心字符串具体怎样，我们只关心这样的字符串到底有多长。为什么？我们想象一下，如果我们用一个字符串去跟另一个字符串匹配，最简单的方法就是把需要匹配的字符串和目标字符串从第一个下标开始一一匹配，直到最后。但是这样的方法十分低效，时间复杂度很高。

我们回想一下，我们在整个过程中做了很多低效的操作，例如，我们在一次操作中，本来已经知道某些区域不可能匹配，我们仍然重复操作，白白浪费了许多时间。因此，我

们不妨从上一次匹配的串中，找出最长的、能与目标字符串开头某一段匹配的子串，并从这个子串的开头进行匹配。

我们参考上图 (此处缺一张图)，在（a）中，我们在第 j 个位置发生了失配（也就是字符串不匹配），那么，如果我们能够记住在 j 前面第二长的匹配子串（也就是图中加黑的 abab），那么我们不妨令 i 不动，让 j 移动到 abab 最后一个 b 的后面（也就是“a”）继续匹配，继续重复这个操作，我们最终就可能找到适配的字符串。不难发现，我们要记录的其实就是：从第 j-1 个下标开始往前数，能与从头开始数的字符串匹配多少的问题。我们只需要记录这个长度，然后令 j=p[j-1]，我们的 j 就会跳到 b 的后面。在上面这个例子中，匹配长度为 abab，长度为 4，我们现在的 j=6，那么我们的 j 就会跳转到 p[5]=4 处，恰好正确！我们通常把这样的预处理数组叫做 PMT（图中为 next 数组，我通常为 p[]）。预处理的手段就很简单了，我们将目标字符串复制一份，两个指针从头开始扫描，其中一个字符串作为主字符串，另一个作为目标字符串，如果下一个匹配，那么记录数组下标并 +1 目，存入 p 中；如果不匹配，那么直接记录数组下标，存入 p 中，并且 j=0。

如上图 (此处缺一张图)，我们首先比较指针指向的字符是否相等，如果不相等，i 始终往后移动一位，p[i]=j，然后 j 归零；如果相同，p[i]=j+1，然后两者同时往后移动一位；

8.4 有限状态机

8.5 AC 自动机

8.5.1 什么是 AC 自动机

事实上，AC 自动机是字典树和 KMP 算法思想的结合，它允许在多个样本字符串中寻找与目标串的最长匹配子串，并且时间复杂度远远优于深度优先搜索。

Definition 8.5.1. *AC 自动机是由三元组 AC(N,C,F) 组成的匹配类型数据结构，其中 N 表示结点编号，C 表示用字母标记的指针（指向下一节点），F 表示 fail 指针集合。*

8.6 字符串处理与字典序

第九章　优化策略

9.1　数据预处理

9.1.1　前缀和与差分数组

9.1.2　离散化

9.2　滑动窗口

9.2.1　定长滑动窗口

定长滑动窗口适合维护以下的几类问题：

- 简单的区间最大值求解
- 环状数组求最值（数组复制两倍看窗口【不必要真的赋值两倍，取模循环即可】）
- 时间间隔重新安排的最小次数（将距离视为数，将操作次数看作滑动窗口长度）
- 对于数组首位进行操作的问题（将数组复制两倍，将操作视为取子串；包含首尾的滑动窗口等）
- ……

9.2.2 不定长滑动窗口（队列）

9.3 双指针

9.3.1 头尾针

9.3.2 快慢针

9.4 滚动数组

9.5 位运算

9.5.1 异或应用

9.5.2 汉明距离

9.5.3 格雷码

9.6 快速幂

在迭代和数列问题中中，我们常常会考虑递推公式，如斐波那契数列 $a_n = a_{n-1} + a_{n-2}$, 但是，如果从第一项开始，按照递推公式计算，计算复杂度为 $O(n)$，这对于较大的数字 n 是极大的代价。因此我们考虑将递推转化为幂次，这样我们就可以利用快速幂技巧将 $O(n)$ 优化到 $O(\log n)$。

如果读者学过线性代数,我们知道上面的递推公式等价于矩阵乘法 $\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \end{pmatrix}$, 不妨记 $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, 称为转移矩阵。我们采用快速幂的计算思路：若 n 为偶数，那么只需要计算 $A^{\frac{n}{2}}$, 进一步只根据奇偶性计算 $A^{\frac{n}{4}}$ 或者 $A^{\frac{n-1}{4}}$，如上递归算法就能够计算出 A^n 的值，最后将计算出的 A^n 代入 $\begin{pmatrix} a_n \\ a_n \end{pmatrix} = A^n \begin{pmatrix} a_2 \\ a_1 \end{pmatrix}$, 由于每次的计算复杂度减半，因此时间复杂度为 $O(\log n)$.

其中，快速幂的思路源于计算数的 n 次幂。如计算 2^{100}, 只需要计算 2^{50}, 然后将两个 2^{50} 相乘就可以得到 2^{100}，按照这个思路，只需要计算 2^{25},2^{12}(2^{25-1} 开根号),$2^6, 2^3, 2$ 即可，如上我们不难发现我们不用计算 100 次乘法，而只需要计算 10 次左右乘法就可以了。

9.7 高精度计算——重新写一个 ALU

9.8 在线算法优化

9.8.1 线段树的应用

9.8.2 区间合并

9.8.3 并查集的应用

9.8.4 莫队算法