算法竞赛 解决方案介绍

A. 有趣的路径

最快的解决方案：**UTokyo：时间操纵器 (0:12)**

A.有趣的路径

问题陈述

给定的是一个有 n 个顶点和 m 条边的 DAG。

如果每条路径的起点都是源点（顶点 1），终点都是汇点（顶点 n），

- 那么可能的最长路径序列是什么？

至少有一条边不包含在前面的路径中

A.有趣的路径

解决方案

- 如果从源头无法到达水槽，答案就是 0。

A.有趣的路径

解决方案

- 如果从源头无法到达水槽，答案就是 0。
- 设 N 为可从源点和汇点到达的顶点数（在后一种情况下为反向图）。

A.有趣的路径

解决方案

- 如果从源头无法到达水槽，答案就是 0。
- 设 N 为可从源点和汇点到达的顶点数（在后一种情况下为反向图）。
- 设 M 为此类边的数量。

A.有趣的路径

解决方案

- 如果从源头无法到达水槽，答案就是 0。
- 设 N 为可同时从源点和汇点到达的顶点数（后一种情况下为反向图）。
- 设 M 为此类边的数量。答案是 $M - N + 2$。

A.有趣的路径

为什么?

- 如果有一条未被访问的边，我们可以扩展路径序列。

A.有趣的路径

为什么?

- 如果有一条未被访问的边，我们可以扩展路径序列。

  如果新路径访问了 v 个未访问过的顶点，那么它必须至少访问 $v + 1$ 条未访问过的边（这样就足够了）。

A.有趣的路径

为什么?

- 如果有一条未被访问的边，我们可以扩展路径序列。
- 如果新路径访问了 v 个未访问过的顶点，那么它必须至少访问 $v+1$ 条未访问过的边（这样就足够了）。

所以 $N-2=\sum_p v$，$M=\sum_p (v+1)=p+\sum_p v$，因此 $p=M-N+2$。

A.有趣的路径

为什么?

- 如果有一条未被访问的边，我们可以扩展路径序列。
- 如果新路径访问了 v 个未访问过的顶点，那么它必须至少访问 $v+1$ 条未访问过的边（这样就足够了）。

所以 $N-2=\sum_p v$，$M=\sum_p (v+1)=p+\sum_p v$，因此 $p=M-N+2$。

复杂性

A.有趣的路径

为什么?

- 如果有一条未被访问的边，我们可以扩展路径序列。
- 如果新路径访问了 v 个未访问过的顶点，那么它必须至少访问 $v+1$ 条未访问过的边（这样就足够了）。

所以 $N - 2 = \sum_p v$，$M = \sum_p (v+1) = p + \sum_p v$，因此 $p = M - N + 2$。

复杂性

$O(n+m)$

B. 咆哮 III

最快的解决方案：**无 :(**

B.咆哮 III

问题陈述

给定的是一棵树，在它的一些顶点上有标记。对于每一个独立的顶点，我们都应假设它是根，并解决下面的问题：

B.咆哮 III

问题陈述

给定的是一棵树，在它的一些顶点上有标记。对于每一个独立的顶点，我们都应假设它是根，并解决下面的问题：

在一次移动中，我们可以选择一个包含令牌的顶点（与根顶点不同），并将该令牌向根顶点移动一条边。只有当目标顶点不包含令牌时，我们才能这样做。我们必须计算可能的最大移动次数。

一根

- 让我们从最深的顶点开始考虑。

B.咆哮 III

一根

- 让我们从最深的顶点开始考虑。

  如果顶点不包含令牌，但在其子树的某个地方至少有一个令牌，那么最佳做法是将路径上的所有令牌移到最深的一个顶点，然后移向根。

B.咆哮 III

一根

让我们从最深的顶点开始考虑。

如果顶点不包含令牌，但其子树中至少有一个令牌，那么最佳做法是将路径上的所有令牌移到最深的一个顶点，然后移向根。

证明

B.咆哮 III

一根

- 让我们从最深的顶点开始考虑。

  如果顶点不包含令牌，但其子树中至少有一个令牌，那么最佳做法是将路径上的所有令牌移到最深的一个顶点，然后移向根。

证明

交换论据。

B.咆哮 III

一根

- 让我们从最深的顶点开始考虑。

  如果顶点不包含令牌，但其子树中至少有一个令牌，那么最佳做法是将路径上的所有令牌移到最深的一个顶点，然后移向根。

证明

交换论据。

新举措

我们可以把这一系列移动看作是将最深的标记直接移动到子树的根部。

B.咆哮 III

B.咆哮 III

一根

让我们从最深的顶点开始考虑。

如果顶点不包含令牌，但其子树中至少有一个令牌，那么最佳做法是将路径上的所有令牌移到最深的一个顶点，然后移向根。

证明

交换论据。

新举措

我们可以把这一系列移动看作是将最深的标记直接移动到子树的根部。
使用分段树，我们可以在 $O(\log(n))$ 的时间内找到子树上最深的标记。

B.咆哮 III

多根

- 让我们为任何根模拟一下上述过程。

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根部的标记，并将其移回原来的位置）

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。
- 请注意，我们计算答案的顶点不一定是数据结构的根点

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。

  请注意，对于我们的数据结构来说，我们正在计算答案的顶点不一定是根顶点--我们可以以 $O(\log(n))$ 的复杂度 "重根 "树（因为新根顶点会更靠近有向子树，离其他顶点更远）。

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。
- 请注意，对于我们的数据结构来说，我们正在计算答案的顶点不一定是根顶点--我们可以以 $O(\log(n))$ 的复杂度 "重根 "树（因为新根顶点会更靠近有向子树，离其他顶点更远）。
- 如果我们计算了某个顶点 v 的答案，又想计算它的邻居 u 的答案，该怎么办？

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。
- 请注意，对于我们的数据结构来说，我们正在计算答案的顶点不一定是根顶点--我们可以以 $O(\log(n))$ 的复杂度 "重根 "树（因为新根顶点会更靠近有向子树，离其他顶点更远）。
- 如果我们计算了某个顶点 v 的答案，而我们想计算它的邻居 u 的答案，该怎么办？ - 只有两次移动可能不同！

多根

- 让我们为任何根模拟一下上述过程。
- 请注意，这些操作是可逆的（我们可以选择带至根目录的标记，并将其移回原来的位置）--我们可以回滚这些操作。
- 请注意，对于我们的数据结构来说，我们正在计算答案的顶点不一定是根顶点--我们可以以 $O(\log(n))$ 的复杂度 "重根 "树（因为新根顶点会更靠近有向子树，离其他顶点更远）。
- 如果我们计算了某个顶点 v 的答案，而我们想计算它的邻居 u 的答案，该怎么办？ - 只有两次移动可能不同！

让我们回滚这两个动作（首先回滚将令牌指向 v，然后指向 u），然后将代币从正确的子树带到 v 和 u。

复杂性

如果我们在树上维护一棵分段树，我们就能及时执行每个操作 $O(\log(n))$，因此最终复杂度为 $O(n \log(n))$。

C. 雷达

最快的解决方案：**UTokyo：时间操纵器 (0:07)**

C.雷达

问题陈述

给定的是一块 $n \times n$ 的正方形棋盘。对于棋盘上的每一个小格，我们都知道在其中建造一个雷达的成本，这个雷达将覆盖以该小格为中心边长为 n 的正方形。覆盖整个棋盘的最小成本是多少？

C.雷达

第一个简单的观察结果

我们需要覆盖黑板的每个角落。

C.雷达

第一个简单的观察结果

我们需要覆盖黑板的每个角落。

第二个简单的观察

如果我们覆盖了所有的角落，就肯定覆盖了整个棋盘。

C.雷达

第一个简单的观察结果

我们需要覆盖黑板的每个角落。

第二个简单的观察

如果我们覆盖了所有的角落，就肯定覆盖了整个棋盘。

结论

我们可以只关注角落。

C.雷达

算法

我们维持覆盖每个边角掩膜的成本

C.雷达

算法

我们维持覆盖每个角的掩膜的成本--$DP[16]$ 将有所帮助。

C.雷达

算法

我们维持覆盖每个角的掩膜的成本--*DP*[16] 将有所帮助。

要考虑一个掩膜角为 *m*、成本为 *x* 的单元，我们需要放宽动态程序设计的条件

C.雷达

算法

我们维持覆盖每个角的掩膜的成本--$DP[16]$ 将有所帮助。

要考虑一个掩码角为 m、成本为 x 的单元，我们需要放宽动态编程--对于每个 i，我们执行 $DP[i|m] = \min(DP[i|m], DP[i] + x)$.

C.雷达

算法

我们维持覆盖每个角的掩膜的成本--*DP*[16] 将有所帮助。

要考虑一个掩码角为 *m*、成本为 *x* 的单元，我们需要放宽动态编程--对于每个 *i*，我们执行 *DP*[*i*|*m*] = min(*DP*[*i*|*m*], *DP*[*i*] + *x*)。

复杂性

与电路板大小成线性关系 - $O(n^2)$.

D. Xor 分区

最快的解决方案：**港湾.空间：** **P+P+P (0:08)**

D.Xor 分区

问题陈述

给定的是一个整数序列 a_1 , a_2 , ..., a_n 。

D.Xor 分区

问题陈述

给定的是一个整数序列 a_1 , a_2 , ..., a_n
序列区间的值是其元素的 xor 值。

D.Xor 分区

问题陈述

给定的是一个整数序列 a_1 , a_2 , ..., a_n
序列区间的值是其元素的 xor 值。

将序列分割成区间的值是每个区间值的乘积。

D.Xor 分区

问题陈述

给定的是一个整数序列 a_1 , a_2 , ..., a_n
序列区间的值是其元素的 xor 值。

将序列分割成区间的值是每个区间值的乘积。

计算 a 的所有分区值之和。

D.Xor 分区

慢速解决方案

- $dp[i]$ - $a_1, a_2, \ldots$ 的所有分区值之和。，a_i

D.Xor 分区

慢速解决方案

- $dp[i]$ - $a_1, a_2, \ldots$ 的所有分区值之和。，a_i
- 遍历分区中最后一个区间的长度。

D.Xor 分区

慢速解决方案

- $dp[i]$ - a_1 , a_2 , . . . 的所有分区值之和。，a_i
- 遍历分区中最后一个区间的长度。

  这些分段的数值总和--最后一个分段的数值乘以前缀所有分段的数值总和。

D.Xor 分区

慢速解决方案

- $dp[i]$ - a_1 , a_2 , . . . 的所有分区值之和。, a_i
- 遍历分区中最后一个区间的长度。
- 这些分段的数值总和--最后一个分段的数值乘以前缀所有分段的数值总和。

$$dp[i] = \sum_{j=0}^{i} dp[j] - XOR(a_{j+1} , a_{j+2} , \ldots , a_i)$$

D.Xor 分区

慢速解决方案

- $dp[i]$ - a_1 , a_2 , . . . 的所有分区值之和。, a_i
- 遍历分区中最后一个区间的长度。
- 这些分段的数值总和--最后一个分段的数值乘以前缀所有分段的数值总和。

$$dp[i] = \sum_{j=0}^{i} dp[j] - XOR(a_{j+1} , a_{j+2} , \ldots , a_i)$$

复杂度 $O(n^2)$ - 太慢。

D.Xor 分区

更好的想法

- 我们可以使用乘法的分配律，而不是加法。

D.Xor 分区

更好的想法

- 我们可以利用乘法的分配性质来计算加法。分别考虑最后一个区间的每个位。

更好的想法

- 我们可以利用乘法的分配性质来计算加法。分别考虑最后一个区间的每个位。

$$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] - 2^b - [\text{b 设为 } XOR(a_{j+1}, a_{j+2}, \ldots, a_i)]$$

更好的想法

- 我们可以利用乘法的分配性质来计算加法。分别考虑最后一个区间的每
- 个位。

$$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] - 2^b - [\text{b 设为 } XOR(a_{j+1}, a_{j+2}, \dots, a_i)]$$

设 $pref_i = XOR(a_1, a_2, \dots, a_i)$

更好的想法

- 我们可以利用乘法的分配性质来计算加法。分别考虑最后一个区间的每个位。

$$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^{b} \cdot [\text{b 设为 } XOR(a_{j+1}, a_{j+2}, \ldots, a_i)]$$

- 设 $pref_i = XOR(a_1, a_2, \ldots, a_i)$

$$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^{b} \cdot [\text{b的状态在}前缀_i\text{和}前缀_j\text{中不同}]$$

D.Xor 分区

$b_j=0$

D.Xor 分区

算法

我们可以计算 $DP_2[i][b][2]$ - $j \le i$ 的 $DP[j]$ 的总和，这样在前 i 位 b 设置或不设置。用它来更新和计算 DP 也很容易。

D.Xor 分区

算法

我们可以计算 $DP_2[i][b][2] - j \leq i$ 的 $DP[j]$ 的总和，这样在前 i 位 b 设置或不设置。用它来更新和计算 DP 也很容易。

内存优化

我们只能记住 DP 的最后一层。$_2$

D.Xor 分区

算法

我们可以计算 $DP_2[i][b][2]$ - $j \leq i$ 的 $DP[j]$ 的总和，这样在前 i 位 b 设置或不设置。用它来更新和计算 DP 也很容易。

内存优化

我们只能记住 DP 的最后一层。$_2$

复杂性

$O(n - \log(max(a_i)))$

E. 图案搜索 II

最快的解决方案：**UTokyo：时间操纵器 (1:21)**

E.模式搜索 II

问题陈述

给定的是二进制字母表上的字符串 t。我们必须从无限斐波那契单词中选择一个与之相等的子序列，使得第一个位置和最后一个位置之间的距离最小。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

如果我们在 S_k 中，那么我们要么在它的左边部分（S_{k-1}），要么在它的右边部分（S_{k-2}），要么在它们两个中。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

如果我们在 S_k 中，那么我们要么在它的左边部分（S_{k-1}），要么在它的右边部分（S_{k-2}），要么在它们两个中。

我们可以将左边部分改写为 $S\ S_{k-2k-3}$，右边部分改写为 $S\ S\ S_{k-4k-5k-4}$。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

如果我们在 S_k 中，那么我们要么在它的左边部分（S_{k-1}），要么在它的右边部分（S_{k-2}），要么在它们两个中。

我们可以将左边部分改写为 $S\ S_{k-2k-3}$，右边部分改写为 $S\ S\ S_{k-4k-5k-4}$。如果 $|S_{k-4}| \geq 3n$，那么肯定 S_{k-3} 在左边部分足够，S_{k-4} 在右边部分足够，因此我们适合 S_{k-2}。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

如果我们在 S_k 中，那么我们要么在它的左边部分（S_{k-1}），要么在它的右边部分（S_{k-2}），要么在它们两个中。

我们可以将左边部分改写为 $S\ S_{k-2k-3}$，右边部分改写为 $S\ S\ S_{k-4k-5k-4}$。如果 $|S_{k-4}| \geq 3n$，那么肯定 S_{k-3} 在左边部分足够，S_{k-4} 在右边部分足够，因此我们适合 S_{k-2}。

我们适合 S_k，且 $|S_{k-4}| \geq 3n \rightarrow$ 我们适合 S_{k-1}。

无限词

斐波那契单词中的每 3 个连续字符都包含两个字母，因此结果不会超过 $3n$。

如果我们在 S_k 中，那么我们要么在它的左边部分（S_{k-1}），要么在它的右边部分（S_{k-2}），要么在它们两个中。

我们可以将左边部分改写为 $S\ S_{k-2k-3}$，右边部分改写为 $S\ S\ S_{k-4k-5k-4}$。如果 $|S_{k-4}| \geq 3n$，那么肯定 S_{k-3} 在左边部分足够，S_{k-4} 在右边部分足够，因此我们适合 S_{k-2}。

我们适合 S_k，且 $|S_{k-4}| \geq 3n \rightarrow$ 我们适合 S_{k-1}。

无限词

我们不必过多地寻找最佳子序列。

E.模式搜索 II

慢速解决方案

迭代 S 中子序列应从哪个位置开始。

慢速解决方案

迭代 S 中子序列应该从哪个位置开始。在向右迭代的同时，我们可以贪婪地匹配 t 中的字符，找到可能的最早的最后位置。

E.模式搜索 II

慢速解决方案

迭代 S 中子序列应从哪个位置开始。在向右迭代的同时，我们可以贪婪地匹配 t 中的字符，找到可能的最早的最后位置。

加速

我们需要能够回答以下问题：如果我们想从第 i 个字符开始将 t 的字符匹配到 S_k，我们会匹配多少个字符？

E.模式搜索 II

慢速解决方案

迭代 S 中子序列应从哪个位置开始。在向右迭代的同时，我们可以贪婪地匹配 t 中的字符，找到可能的最早的最后位置。

加速

我们需要能够回答以下问题：如果我们想从第 i 个字符开始将 t 的字符匹配到 S_k，我们会匹配多少个字符？

这将帮助我们快速跳过 S 的片段。

E.模式搜索 II

慢速解决方案

迭代 S 中子序列应该从哪个位置开始。在向右迭代的同时，我们可以贪婪地匹配 t 中的字符，找到可能的最早的最后一个位置。

加速

我们需要能够回答以下问题：如果我们想把 t 中从第 i 个开始的字符匹配到 S_k，我们会匹配多少个字符？

这将帮助我们快速跳过 S 的片段。所有这些查询的答案都可以很容易地计算出来

E.模式搜索 II

慢速解决方案

迭代 S 中子序列应该从哪个位置开始。在向右迭代的同时，我们可以贪婪地匹配 t 中的字符，找到可能的最早的最后位置。

加速

我们需要能够回答以下问题：如果我们想从第 i 个字符开始将 t 的字符匹配到 S_k，我们会匹配多少个字符？

这将帮助我们快速跳过 S 的片段。

所有这些查询的答案都可以很容易地计算出来--如果我们用 $DP[i][k]$ 表示上述问题的答案，那么

E.模式搜索 II

$DP[i][k] = DP[i][k-1] + DP[i + DP[i][k-1]][k-2]$ 成立。

E.模式搜索 II

快速解决方案

我们可以在恒定时间内计算上述数组中的每个单元格，因此我们可以在 $O(n - \log(n))$ 的时间内计算所有单元格。

E.模式搜索 II

快速解决方案

我们可以在恒定时间内计算上述数组中的每个单元格，因此我们可以在 $O(n - \log(n))$ 的时间内计算所有单元格。

利用它，我们可以在 $O(\log(n))$时间内为每个可能的起始位置找到最早可能的最后位置。

E.模式搜索 II

快速解决方案

我们可以在恒定时间内计算上述数组中的每个单元格，因此我们可以在 $O(n \cdot \log(n))$ 的时间内计算所有单元格。

利用它，我们可以在 $O(\log(n))$时间内为每个可能的起始位置找到最早可能的最后位置。

复杂性

动态编程和查找所有子序列的时间为 $O(n \cdot \log(n))$

每个

F. 瀑布矩阵

最快的解决方案：**添加列车队 (1:41)**

F.瀑布矩阵

问题陈述

我们想创建一个 $n \times n$ 矩阵，其中所有列和行的值都不递增。对于矩阵中的某些单元格子集，我们被告知其中应包含的内容。对于每一个单元格，惩罚都是所需值与矩阵中值之间的绝对差。我们必须最小化惩罚总和。

F.瀑布矩阵

准备工作

我们可以在不改变答案的情况下移动给定的单元格，使所有单元格都位于不同的行和列中。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

一个障碍

如果只计算一个障碍物，即位置上的障碍物，我们就能找到一个最佳矩阵。
$x + \frac{1}{2}$
。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

一个障碍

如果只计算一个障碍物，即位置上的障碍物，我们就能找到一个最佳矩阵。

$x + \frac{1}{2}$ 我们可以将 $\leq x$ 的数变为 0，将 $> x$ 的数变为 1。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

一个障碍

如果只计算一个障碍物，即位置上的障碍物，我们就能找到一个最佳矩阵。

$x + \frac{1}{2}$。我们可以将 $\leq x$ 的数变为 0，将 $> x$ 的数变为 1。

我们正在寻找一个从矩阵右上角到左下角的 "边界"。

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

一个障碍

如果只计算一个障碍物，即位置上的障碍物，我们就能找到一个最佳矩阵。

$x + \frac{1}{2}$。我们可以将 $\leq x$ 的数变为 0，将 $> x$ 的数变为 1。

我们正在寻找一个从矩阵右上角到左下角的 "边框"--对于每个单元格，我们都知道它要位于边框的哪一边

F.瀑布矩阵

有什么区别?

对于任意整数 x，我们可以想象在 $x + \frac{1}{2}$ 位置的数线上有一个 "障碍"。
$|a - b|$ 是 a 和 b 之间的障碍数。

一个障碍

如果只计算一个障碍物，即位置上的障碍物，我们就能找到一个最佳矩阵。

$x + \frac{1}{2}$。我们可以将 $\leq x$ 的数变为 0，将 $> x$ 的数变为 1。

我们正在寻找一条从矩阵右上角到左下角的 "边界"--对于每个单元格，我们都知道它想位于边界的哪一边--对于位于错误一边的每个单元格，我们都要支付一个单位的罚金。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。

  对于列之间的每一条边，我们都要记住如果边框到达该处的罚则

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。

  对于列之间的每一条边，我们都要记住如果边线到达那里时的惩罚值，这些值是不递减的。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。

  对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果它们不是递减的，而某些边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。
- 对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果它们不是递减的，而某些边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。
- 在经过重要单元格时，我们要么将后缀增加 1，要么将前缀增加 1。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。
- 对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果它们不是递减的，而某些边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。

  在后一种情况下，我们可能需要将某些区间减少 1，以保持惩罚不递减。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。
- 对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果不是，而某条边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。
- 在后一种情况下，我们可能需要将某些区间减少 1，以保持惩罚不递减。

我们可以使用多集来存储结果增加的地方。

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。
- 对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果不是，而某条边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。
- 在后一种情况下，我们可能需要将某些区间减少 1，以保持惩罚不递减。
- 我们可以使用多集来存储结果增加的地方。

边境恢复

我们可以逆转这一过程，恢复最佳边界

F.瀑布矩阵

扫线

- 我们将矩阵从上到下逐行添加。
- 对于列之间的每一条边，我们都要记住如果边线经过那里时的惩罚值--这些值是不递减的。如果不是，而某条边的惩罚值低于其左边的边，我们就会减少左边边的惩罚值。
- 在后一种情况下，我们可能需要将某些区间减少 1，以保持惩罚不递减。

我们可以使用多集来存储结果增加的地方。

边境恢复

我们可以将这一过程反过来，恢复出最佳边界--这将告诉我们哪些单元格应 $\leq x$，哪些单元格应 $> x$。

F.瀑布矩阵

主要意见

x 的最佳边界将低于 $x + 1$ 的最佳边界并位于其右侧

F.瀑布矩阵

主要意见

x 的最优边界将在 $x + 1$ 的最优边界的下方和右侧 - 我们可以分别求出它们，并将结果相加！

主要意见

x 的最优边界将在 $x + 1$ 的最优边界的下方和右侧 - 我们可以分别求出它们，并将结果相加！

这样，我们就能在 $O(n^2 \log(n))$ 的时间内得到正确的算法。

F.瀑布矩阵

分而治之

对于任意 x，我们可以检查哪些单元格应该大于 x，哪些不应该。

F.瀑布矩阵

分而治之

对于任意 x，我们可以检查哪些单元格应该大于 x，哪些不应该。

我们可以针对 x 的 "中间 "值运行算法，并在两边递归分割，每个单元格只传递到一边。

分而治之

对于任意 x，我们可以检查哪些单元格应该大于 x，哪些不应该。

我们可以针对 x 的 "中间 "值运行算法，并在两边递归分割，每个单元格只传递到一边。

将有 $O(\log(n))$ 层递归，每层递归总共包含 n 个单元，因此需要 $O(n \log(n))$ 个单元来考虑所有单元。

F.瀑布矩阵

分而治之

对于任意 x，我们可以检查哪些单元格应该大于 x，哪些不应该。

我们可以针对 x 的 "中间 "值运行算法，并在两边递归分割，每个单元格只传递到一边。

将有 $O(\log(n))$ 层递归，每层递归总共包含 n 个单元，因此需要 $O(n \log(n))$ 个单元来考虑所有单元。

复杂性

如果我们使用的数据结构（如多集合）每次操作的复杂度为 $O(\log(n))$，那么最终的复杂度为 $O(n \log^2(n))$。

G. 拼图 II

最快的解决方案：**添加列车队 (2:04)**

G. 拼图 II

问题陈述

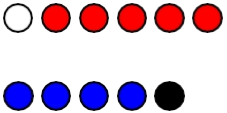
在一次移动中，我们可以从第一个序列中选择一个长度为 k 的循环段，从第二个序列中选择一个长度相同的循环段，然后将它们互换。我们需要在最多 n 次移动中使两个序列都成为单色。

G. 拼图 II

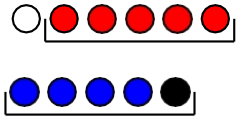
怎么办？

我们可以采取哪些有组织的行动？

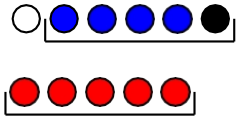
G. 拼图 II



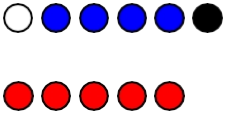
G. 拼图 II



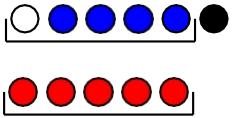
G. 拼图 II



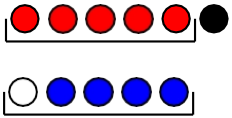
G. 拼图 II



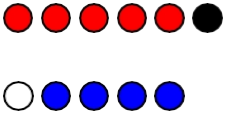
G. 拼图 II



G. 拼图 II



G. 拼图 II



G. 拼图 II

怎么办？

通过两次移动，我们可以将一个元素从第一个序列移动到第二个序列，再将一个元素从第二个序列移动到第一个序列。

G. 拼图 II

怎么办？

在两步棋中，我们可以将一个元素从第一个序列移动到第二个序列，再将一个元素从第二个序列移动到第一个序列。由于我们可以选择哪个序列是白棋，我们可以这样做最多 $\frac{n}{2}$ 这样的操作，结果是 $\leq n$ 次移动。

G. 拼图 II

我们该开始了吗?

有经验的人应该会发现使用了一些 BST 的溶液会导致

$O(n \cdot \log(n))$ 复杂性.. .

G. 拼图 II

我们该开始了吗?

有经验的人应该能发现使用某些 BST 的解决方案，其复杂度为 $O(n \cdot \log(n))$。. .不过，我们还是应该寻找更简单、更快速的方法。

G. 拼图 II

聪明的方法

让我们在第一个序列的第一个元素上设置一个大小为 $k + 1$ 的滑动窗口。

G. 拼图 II

聪明的方法

在第一个序列的第一个元素上设置大小为 $k + 1$ 的滑动窗口。在第二个序列的最后一个元素上设置大小为 k 的滑动窗口。

G. 拼图 II

聪明的方法

在第一个序列的第一个元素上设置大小为 $k + 1$ 的滑动窗口。在第二个序列的最后一个元素上设置大小为 k 的滑动窗口。

我们将把两个窗口都存储在 deques 上--所述操作可以在恒定时间内完成。

G. 拼图 II

聪明的方法

在第一个序列的第一个元素上设置大小为 $k + 1$ 的滑动窗口。在第二个序列的最后一个元素上设置大小为 k 的滑动窗口。

我们将把两个窗口都存储在 deques 上--所述操作可以在恒定时间内完成。我们还可以在恒定时间内将窗口向左和向右移动一个。

G. 拼图 II

聪明的方法

在第一个序列的第一个元素上设置大小为 $k + 1$ 的滑动窗口。在第二个序列的最后一个元素上设置大小为 k 的滑动窗口。

我们将把两个窗口都存储在 deques 上--所述操作可以在恒定时间内完成。我们还可以在恒定时间内将窗口向左和向右移动一个。

只要我们对第一个窗口的第一个元素不满意，就可以向右滑动，向左滑动第二个窗口。

G. 拼图 II

聪明的方法

在第一个序列的第一个元素上设置大小为 $k + 1$ 的滑动窗口。在第二个序列的最后一个元素上设置大小为 k 的滑动窗口。

我们将把两个窗口都存储在 deques 上--所述操作可以在恒定时间内完成。我们还可以在恒定时间内将窗口向左和向右移动一个。

只要我们对第一个窗口的第一个元素不满意，就可以向右滑动，向左滑动第二个窗口。

我们最多会移动两个窗口 $O(n)$ 次。

G. 拼图 II

复杂性

使用自选的 BST 可达到 $O(n \cdot \log(n))$，使用一些技巧则可*达到* $O(n)$。

H. 天气预报

最快的解决方案：**LNU：LNU 种马 (0:17)**

H.天气预报

问题陈述

我们给定了一个整数序列和一个数字 k。我们需要将这个序列划分为 k 个区间，使所有区间的最小平均数最大。

H.天气预报

主要意见

如果我们能让所有均值≥ x，那么如果 $x \geq y$，我们肯定能让所有均值≥ y。

H.天气预报

主要意见

如果我们能让所有均值≥ x，那么如果 $x \geq y$，我们肯定能让所有均值≥ y。

结论

二进制搜索找出答案。

H.天气预报

我们能让所有手段 $\geq x$ 吗?

我们能让所有手段 $\geq x$ 吗

- $\frac{a_1+\ldots+a_n}{n} \geq x \iff a_1 + \ldots + a_n \geq nx \iff (a_1 - x) + \ldots + (a_n - x) \geq 0$

我们能让所有手段 ≥ x 吗

- $\frac{a_1+\ldots+a_n}{n} \geq x \iff a_1 + \ldots + a_n \geq nx \iff (a_1 - x) + \ldots + (a_n - x) \geq 0$
- 从序列中的每个数字中减去 x。

我们能让所有手段 $\geq x$ 吗?

- $\frac{a_1+\ldots+a_k}{l} \geq x \iff a_1 + \ldots + a_l \geq x \cdot l \Rightarrow (a_1 - x) + \ldots + (a_l - x) \geq 0$ 从序列中的每个数中减去 x。
- 计算前缀和并设置序列 $0 = e_0, e_1, e_2, e_3, ., e_k = n$ 为分区中区间的两端。

我们能让所有手段 $\geq x$ 吗?

- $\frac{a_1+\ldots+a_k}{l} \geq x \iff a_1 + \ldots + a_l \geq x \cdot l \Rightarrow (a_1 - x) + \ldots + (a_l - x) \geq 0$ 从序列中的每个数中减去 x。
- 计算前缀和并设置序列 $0 = e_0, e_1, e_2, e_3, ., e_k = n$ 为分区中区间的两端。
- 这些点的前缀和必须构成一个非递减序列。我们必须选择其中至少 $k + 1$ 个点（包括 0 和 n）。

H.天气预报

我们能让所有手段 ≥ x 吗?

- $\frac{a_1+\ldots+a_k}{l} \geq x \iff a_1 + \ldots + a_l \geq x \cdot l \Rightarrow (a_1 - x) + \ldots + (a_l - x) \geq 0$ 从序列中的每个数中减去 x。
- 计算前缀和并设置序列 $0 = e_0, e_1, e_2, e_3, ., e_k = n$ 为分区中区间的两端。
- 这些点的前缀和必须构成一个非递减序列。我们必须选择其中至少 $k + 1$ 个点（包括 0 和 n）。

解决方案

最长递增序列。

我们能让所有手段 ≥ x 吗?

- $\frac{a_1+\ldots+a_k}{l} \geq x \iff a_1 + \ldots + a_l \geq x \cdot l \Rightarrow (a_1 - x) + \ldots + (a_l - x) \geq 0$ 从序列中的每个数中减去 x。
- 计算前缀和并设置序列 $0 = e_0, e_1, e_2, e_3, ., e_k = n$ 为分区中区间的两端。
- 这些点的前缀和必须构成一个非递减序列。我们必须选择其中至少 $k + 1$ 个点（包括 0 和 n）。

解决方案

最长递增序列。

复杂性

$O(n \cdot \log(n) \cdot \log(precision))$

H.天气预报

I. 雇佣军

最快的解决方案**UTokyo：时间操纵器 (4:10)**

I.雇佣军

问题陈述

给我们一条笔直的道路，在这条道路上我们只能向右移动。这条路上有 n 座城市，居住在第 i 座城市的雇佣兵的参数对为（s_i , m_i ）。每两个相邻的城市之间都有一个商店，可以在商店里购买一件物品，每件物品都会给雇佣兵的两个统计数据增加一些值。当佣兵从一个城市移动到另一个城市时（只能向右移动），他可以在每个商店购买一件物品，这些物品的加成值会累积起来。我们必须考虑怪物攻击的情况--怪物可能会攻击某个城市，而怪物的参数有三个--A、B 和 C。如果一个雇佣兵能到达怪物所在的位置，并拥有 $A - S + B - M \geq C$ 的统计值（S, M），那么他就能打败怪物。我们必须找出能打败每个怪物的最右边的雇佣兵。

I.雇佣军

这是几何图形吗?

让我们把雇佣兵和物品理解为坐标平面四分之一处的矢量

I.雇佣军

这是几何图形吗?

让我们把佣兵和物品理解为坐标平面四分之一处的矢量--打败怪物就意味着位于给定的半平面内。

I.雇佣军

这是几何图形吗?

让我们把佣兵和物品理解为坐标平面四分之一处的矢量--打败怪物就意味着位于给定的半平面内。

这是凸面船体吗?

要检查一个怪物是否能被打败，我们只需要考虑能到达这个怪物身边的雇佣兵的统计数据的右上角凸壳。

有组织的方法

让我们在城市序列上建立一棵分段树。

I.雇佣军

有组织的方法

让我们在城市序列上建立一棵分段树。

在每个基段中，让我们计算一下通过该基段时可能获得的奖金的凸壳。

有组织的方法

让我们在城市序列上建立一棵分段树。

在每个基段中，让我们计算一下通过该基段时可能获得的奖金的凸壳。
一段的凸面体是其两个子段的凸面体的闵科夫斯基和

有组织的方法

让我们在城市序列上建立一棵分段树。

在每个基段中，让我们计算一下通过该基段时可能获得的奖金的凸壳。

一个线段的凸壳是它的两个子线段的凸壳的闵科夫斯基和，我们可以在线性时间内合并它们。

有组织方法第二部分

对于每个基段，我们还可以计算出雇佣兵在离开该基段时从该基段开始的可能统计数据的凸壳。

I.雇佣军

有组织方法第二部分

对于每个基段，我们还可以计算出雇佣兵在离开该基段时从该基段开始的可能统计数据的凸壳。
这样的雇佣兵可以从正确的分区开始

I.雇佣军

有组织方法第二部分

对于每个基段，我们还可以计算出雇佣兵在离开该基段时从该基段开始的可能统计数据的凸壳。

这样的雇佣兵可以从右边的分区或左边的分区开始，用右边分区的物品来加强自己的实力

有组织方法第二部分

对于每个基段，我们还可以计算出雇佣兵在离开该基段时从该基段开始的可能统计数据的凸壳。

这样的雇佣兵可以从右边的分段开始，也可以从左边的分段开始，用右边区间的物品来加强自己的实力（同样是闵科夫斯基和）。

有组织方法第二部分

对于每个基段，我们还可以计算出雇佣兵在离开该基段时从该基段开始的可能统计数据的凸壳。

这样的雇佣兵可以从右边的分段开始，也可以从左边的分段开始，用右边区间的物品来加强自己的实力（同样是闵科夫斯基和）。

一组点的凸壳也可以在线性时间内计算出来。

攻击场景

当怪物攻击时，让我们把前缀分成基本段，然后从右向左考虑。

I.雇佣军

攻击场景

当怪物攻击时，让我们把前缀分成基本段，然后从右向左考虑。

要检查该段中是否有雇佣兵可以打败它，我们可以对船体进行二元搜索（或三元搜索）。

I.雇佣军

攻击场景

当怪物攻击时，让我们把前缀分成基本段，然后从右向左考虑。

要检查该段中是否有雇佣兵可以打败它，我们可以对船体进行二元搜索（或三元搜索）。

如果是，我们就深入树中。

I.雇佣军

攻击场景

当怪物攻击时，让我们把前缀分成基本段，然后从右向左考虑。

要检查该段中是否有雇佣兵可以打败它，我们可以对船体进行二元搜索（或三元搜索）。

如果是，我们就深入树中。

如果没有，我们就必须考虑佣兵团可以购买的物品。

I.雇佣军

攻击场景

当怪物攻击时，让我们把前缀分成基本段，然后从右向左考虑。

要检查该段中是否有雇佣兵可以打败它，我们可以对船体进行二元搜索（或三元搜索）。

如果是，我们就深入树中。

如果没有，我们就必须考虑佣兵可以购买的物品。它们对怪物可能产生的最大影响（我们应该将 *C* 参数降低多少的信息）也可以通过二进制搜索找到。

优化

描述的 $O(\log(n))$ 基段方法进行二进制搜索

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

让我们来看看按照半平面角度排序的怪物吧

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

让我们考虑一下按半平面角度排序的怪物--所有凸壳上的最佳点只会向一个方向移动

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

让我们考虑一下按照半平面角度排序的怪兽们--所有凸壳上的最佳点只会向一个方向移动--我们只需移除

如果我们将它们存储为列表，就可以从凸壳中提取非最佳向量。

I.雇佣军

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

让我们考虑一下按照半平面角度排序的怪兽们--所有凸壳上的最佳点只会向一个方向移动--我们只需移除

如果我们将它们存储为列表，就可以从凸壳中提取非最佳向量。

复杂性

提及优化后，我们只需将输入的大小乘以一个对数，即段树的高度和按角度对怪物和项目进行排序的需要。

优化

描述的 $O(\log(n))$ 基段方法是二进制搜索，我们可以做得更好。

让我们考虑一下按照半平面角度排序的怪兽们--所有凸壳上的最佳点只会向一个方向移动--我们只需移除

如果我们将它们存储为列表，就可以从凸壳中提取非最佳向量。

复杂性

提及优化后，我们只需将输入的大小乘以一个
对数--分段树的高度和对怪物进行分类的需要，以及
项的角度。我们最终会得到 $O((n + \sum r_{ii} + q) - \log(n + \sum r_{ii}))$ 的一致性。

J. 多边形 II

最快的解决方案：**Harbour.Space：P+P+P (3:24)**

问题陈述

随机变量 X_1 , X_2 , ..., X_n , 其中 $X_i = U(0, 2^a \text{ i})$.求我们能构建一个边长为 X_i 的非整数多边形的概率。

三角形不平等

当且仅当对于某些 i

$$x_i \geq \sum_{j/=i} x_j$$

三角形不平等

当且仅当对于某些 i

$$x_i \geq \sum_{j} x_j$$

。

$j/=i$

回答

$$1 - \sum_{i} p(x_i \geq \sum_{j} x_j)$$

- 坏事件是不相连的。 $j/=i$

有用的阶梯

$$p(x_i \geq \sum_{j/=i} x_j) = p(2^{a_i} \geq \sum_j x_j)$$

有用的阶梯

$$p(x_i \geq \sum_{j/=i} x_j) = p(2^{a_i} \geq \sum_j x_j)$$

证明

$$X_i \sim 2 X^{a_i}{}_i$$

有用的阶梯

$$p(x_i \geq \sum_{j/=i} x_j) = p(2^{a_i} \geq \sum_j x_j)$$

证明

$$X_i \sim 2 X^{a_i}{}_i$$

$$p(x_i \geq \sum_{j/=i} x_j) = p(2^{a_i} - x_i \geq \sum_{j/=i} x_j) = p(2 \geq x^{a_i} \sum_j)$$

主旨

让 Y_i 成为一个只有两个可能值的随机变量：

$$p(y_i = 0) = \frac{1}{2}, \; p(y_i = 2^i) = \frac{1}{2}$$

主旨

让 Y_i 成为一个只有两个可能值的随机变量：

$$p(y_i = 0) = \frac{1}{2}, p(y_i = 2^i) = \frac{1}{2}$$

比特分解

$$x_i = u(0, 1) + Y_0 + Y_1 + \ldots + Y_{a_i - 1}$$

动态编程

$DP[i][j]$ - 在决定所有 $U(0, 1)$, Y_0 , Y_1 , . .. 之后，我们携带 j 比特（值为 2^j ）的概率。

Y_{i-1}

动态编程

$DP[i][j]$ - 在决定所有 $U(0, 1)$, Y_0 , Y_1 , . .. 之后，我们携带 j 比特（值为 2^i ）的概率。

$Y_{0 \ldots i-1}$

过渡

k_i - Y 类型变量的数量$_i$

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

动态编程

$DP[i][j]$ - 在决定所有 $U(0, 1)$, Y_0 , Y_1 , . .. 之后，我们携带 j 比特（值为 2^i ）的概率。

$Y_{0 \ldots i-1}$

过渡

k_i - Y 类型变量的数量$_i$

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

初始化 - 仅 U（0，1）

$\sum_{i=0}^{j} i$ $DP[0][i] = n$ 维多面体的体积 $x^{\sum_i} < j$ 且 $0 \le x_i \le 1$。

动态编程

$DP[i][j]$ - 在决定所有 $U(0, 1)$, Y_0 , Y_1 , . .. 之后，我们携带 j 比特（值为 2^i ）的概率。

$Y_{0 \ldots i-1}$

过渡

k_i - Y 类型变量的数量$_i$

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

初始化 - 仅 U（0，1）

$\sum_{i=0}^{j} DP[0][i] = n$ 维多面体的体积 $x_i < j$ 且 $0 \le x_i \le 1$。

关于有多少 $x_i \le 1$ 不符合包含-排除原则。

复杂性

$O(max(a_i) - n)^2$

复杂性

$O(max(a_i) \cdot n^2)$ 或 $O(max(a_i) \cdot n \cdot \log(n))$（使用 FFT）。

K. 权力划分

最快的解决方案：**添加列车队 (1:18)**

K.权力划分

问题陈述

给定的是一个序列 b_1 , b_2 , ... b_n ， 其形式为 2^{a_1} , 2^{a_2} , ..., 2 .^{a_n}

区间 [l , r] 是好$\Leftarrow\Rightarrow$ $b_l + b_{l+1} + \ldots .+ b_r = 2^k$ （对于 k$\in$N）

计算将序列划分为良好区间（素数模）的次数。

K.权力划分

工作流程

K.权力划分

工作流程

- 查找所有好的间隔

K.权力划分

工作流程

- 找到所有好的区间--分而治之。

K.权力划分

工作流程

- 找到所有好的区间 - 除法与征服。计算所有好的分区

K.权力划分

工作流程

- 找到所有好的区间--分而治之。

计算所有好的分区 - 动态编程

总和 S 的表示形式，并用 b 将其递增，

总和 S 的表示形式，并用 b 将其递增$_i$

S	101110
B_i	000100
$S + b_i$	110010

总和 S 的表示形式，并用 b 将其递增$_i$

S	101110
B_i	000100
$S + b_i$	110010

时间 - 摊销 $O(1)$

总和 S 的表示形式，并用 b 将其递增$_i$

S	101110
B_i	000100
$S + b_i$	110010

时间--摊销 $O(1)$（潜力--设置位数）。

K.权力划分

常数

P - 大质数，例如 $2^{61} - 1$。

C_0，C_1，. .，$c\,6_{10+20}$ - 随机系数。

K.权力划分

常数

P - 大质数，例如 $2^{61} - 1$。

$C_0, C_1, \ldots, c\,6_{10+20}$ - 随机系数。

散列

$S = b \sum_{i\,i} - 2^i$

K.权力划分

常数

P - 大质数，例如 $2^{61} - 1$。

$C_0, C_1, \ldots, c\,6_{10+20}$ - 随机系数。

散列

$S = \sum_i b_i \cdot 2^i$

$h(S) = \sum_i b_i - c_i \text{ modulo } P.$

常数

P - 大质数，例如 $2^{61} - 1$。

$C_0, C_1, \ldots, c\,6_{10+20}$ - 随机系数。

散列

$S = \sum_i b_i \cdot 2^i$

$h(S) = \sum_i b_i \cdot c_i \text{ modulo } P.$

在进行二进制重述的同时，我们还将跟踪总和的哈希值（仍按恒定时间摊销）。

常数

P - 大质数，例如 $2^{61} - 1$。

$C_0, C_1, \ldots, c\,6_{10+20}$ - 随机系数。

散列

$S = \sum_i b_i \cdot 2^i$

$h(S) = \sum_i b_i \cdot c_i \text{ modulo } P.$

在进行二进制重述的同时，我们还将跟踪总和的哈希值（仍按恒定时间摊销）。

碰撞概率

$S_1 \neq S_2 \Rightarrow P(h(S_1) = h(S_2))_P = \frac{1}{}$

K.权力划分

所有良好的间隔 - 分而治之

所有良好的区间 - 将区间分为 L 和 R

-

K.权力划分

所有好的区间 - 将区间分为 L 和 R 在 L

-
- 上递归

所有好的区间 - 将区间分为 L 和 R 在 L

-
- 上递归
- R 上的递推

所有好的区间 - 将区间分为 L 和 R 在 L

-
- 上递归
- R 上的递推
- 间隔 $suf_L + pref_R = 2^k$

间隔 $suf_L + pref_R = 2^k$

间隔 $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$（情况 $pref_R < suf_L$ 类似）。

K.电力分部

间隔 $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$（情况 $pref_R < suf_L$ 类似）。

主要意见

对于$_R$，suf_L 的可能值只有一个。

间隔 $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$（情况 $pref_R < suf_L$ 类似）。

主要意见

对于$_R$，suf_L 的可能值只有一个。

前额	0101110
$_R$	1000000
2^k	0010010
苏$_L$	

间隔 $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$（情况 $pref_R < suf_L$ 类似）。

主要意见

对于$_R$，suf_L 的可能值只有一个。

前额	0101110
$_R$	1000000
2^k	0010010
苏$_L$	

计算 $h(suf)_L$

a - $前缀_R$ 中最右边的设置位。

b - $前缀_R$ 中最左边的设置位。

间隔 $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$（情况 $pref_R < suf_L$ 类似）。

主要意见

对于$_R$，suf_L 的可能值只有一个。

前额	0101110
$_R$	1000000
2^k	0010010
苏$_L$	

计算 $h(suf)_L$

a - *前缀*$_R$ 中最右边的设置位

。

b - *前缀*$_R$ 中最左边的设置位

。

$$h(suf_L) + h(pref_R) = c + \sum_{i=a}^{b} c_i$$

。

间隔 suf_L + $pref_R$ = 2^k - 算法

- 记忆 $h(suf_L)$ - 从 $h(suf_L)$ 到 L 的（散列）映射。

间隔 suf_L + $pref_R$ = 2^k - 算法

- 记忆 $h(suf_L)$ - 从 $h(suf_L)$ 到 L 的（散列）映射。遍历 $pref_R$，保留 S、$h(S)$、a 和 b 的散列。

间隔 suf_L + $pref_R$ = 2^k - 算法

- 记忆 $h(suf_L)$ - 从 $h(suf_L)$ 到 L 的（散列）映射。遍历
- $pref_R$，保留 S、$h(S)$、a 和 b 的裂缝。检查是否存在

好的 suf_L。

间隔 $suf_L + pref_R = 2^k$ - 算法

- 记忆 $h(suf_L)$ - 从 $h(suf_L)$ 到 L 的（散列）映射。遍历
- $pref_R$，保留 S、$h(S)$、a 和 b 的裂缝。检查是否存在

好的 suf_L。

碰撞概率

$$\sum_{suf_L} \sum_{pref_R} P(h(suf_L) = h(2^k - pref_R)) \leq \frac{n^2}{P}$$

K.权力划分

复杂性

- 一级分而治之：

复杂性

- 一级分而治之：$O(n)$ (hashmap) 或 $O(n \log n)$ (map)。

复杂性

- 一级分而治之：$O(n)$ (hashmap) 或 $O(n \log n)$ (map)。整体分而治之：

复杂性

- 一级分而治之：$O(n)$ (hashmap) 或 $O(n \log n)$ (map)。整体分而治之：$O(n \log n)$ (hashmap) 或 $O(n \log^2 n)$ (map)。

复杂性

- 一级分而治之：$O(n)$ (hashmap) 或 $O(n \log n)$ (map)。整体分而治之：
- $O(n \log n)$ (hashmap) 或 $O(n \log^2 n)$ (map)。动态编程：

复杂性

- 一级分而治之：$O(n)$ (hashmap) 或 $O(n \log n)$ (map)。整体分而治之：
- $O(n \log n)$ (hashmap) 或 $O(n \log^2 n)$ (map)。动态编程：O（良好区间数）$= O(n \log n)$。

L. 和弦

最快的解决方案：**Harbour.Space：P+P+P (2:36)**

L.和弦

问题陈述

将圆上的 $2n$ 个点随机配对，形成 n 条弦。我们需要找到最大的弦子集，使其中没有两点相交。

更简洁的外观

我们可以在任何位置切圆

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

确定性解决方案

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

确定性解决方案

DP[*l*][*r*] - 如果只从区间中选择，不相交和弦的最大数目

[*l*, *r*]。

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

确定性解决方案

$DP[l][r]$ - 如果只从区间中选择，不相交和弦的最大数量 $[l, r]$。

要计算 $DP[l][r]$，我们肯定要考虑 $DP[l][r-1]$。

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

确定性解决方案

$DP[l][r]$ - 如果只从区间中选择，不相交和弦的最大数量

$[l, r]$。

要计算 $DP[l][r]$，我们肯定要考虑 $DP[l][r-1]$。

如果在点 r 上，某条弦结束，其起点在点 $left_r$ 上，且 $left_r \geq l$ 成立，我们需要考虑 $DP[l][left_r - 1] + 1 + dp[left_r + 1][r-1]$。

更简洁的外观

我们可以在任何位置切圆--现在所选的任意两个区间要么是不相交的，要么其中一个必须包含在另一个中。

确定性解决方案

$DP[l][r]$ - 如果只从区间中选择，不相交和弦的最大数量

$[l, r]$。

要计算 $DP[l][r]$，我们肯定要考虑 $DP[l][r-1]$。

如果在点 r 上，某条弦结束，其起点在点 $left_r$ 上，且 $left_r \geq l$ 成立，我们需要考虑 $DP[l][left_r - 1] + 1 + dp[left_r + 1][r-1]$。

所描述的动态编程在 $O(n^2)$ 时间内计算出正确答案，并在 $DP[1][2n]$ 中返回。

L.和弦

为什么是随机性?

和弦之间经常会出现混乱的交错

L.和弦

为什么是随机性？

和弦之间的相交往往是杂乱无章的，很难选择一个大的子集，使其中没有两个和弦相交。

L.和弦

为什么是随机性?

和弦之间的相交往往是杂乱无章的--很难选择一个大的子集，使其中没有两个相交--答案会很小!

为什么是随机性?

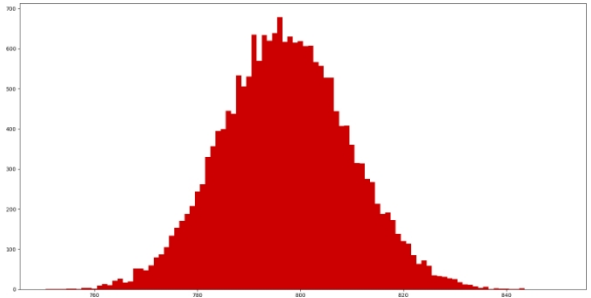
和弦之间的相交往往是杂乱无章的--很难选择一个大的子集，使其中没有两个相交--答案会很小!

最好根据经验来验证--对于 $n = 10^5$ ，答案不会离 800 太远。

为什么是随机性?

和弦之间的相交往往是杂乱无章的--很难选择一个大的子集，使其中没有两个相交--答案会很小!

最好根据经验来验证--对于 $n = 10^5$ ，答案不会离 800 太远。



如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。

如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。我们还知道

$DP[l-1][r] \geq DP[l][r]$。

如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。我们还知道

$DP[l-1][r] \geq DP[l][r]$。

对于每一个 r，我们只能记住哪些 ls 的答案会增加!

如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。我们还知道

$DP[l-1][r] \geq DP[l][r]$。

对于每个 r，我们只能记住哪些 ls 的答案会增加!

通过压缩方形数组，我们可以将存储信息的大小减小到

$O(n-ans)$

如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。我们还知道

$DP[l-1][r] \geq DP[l][r]$。

对于每一个 r，我们只能记住哪些 ls 的答案会增加!

通过压缩方形数组，我们可以将存储信息的大小减小到

$O(n-ans)$ 在 $O(ans)$ 的时间内，我们还可以计算出固定 r 的所有结果。

如何使用?

对于固定的 r 和任意的 l，$DP[l][r]$不会很大。我们还知道

$DP[l-1][r] \geq DP[l][r]$。

对于每一个 r，我们只能记住哪些 ls 的答案会增加!

通过压缩方形数组，我们可以将存储信息的大小减小到

$O(n-ans)$ 在 $O(ans)$ 的时间内，我们还可以计算出固定 r 的所有结果。

复杂性

最终，我们的时间和内存复杂度为 $O(n-ans)$。

M. 排列平衡

最快的解决方案：**Harbour.Space：P+P+P (1:31)**

M.排列平衡

问题陈述

排列 p 的余数定义为 $|p_i - i|$ 的和。我们必须找出余额等于 b *的第* k *个*词性最小的 n *元素*排列。

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

1 1 2 2 3 3 4 4 5 5

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

1 1 2 2 3 3 4 4 5 5

动态编程

每一对都将从正确数字的角度进行创作。

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

1 1 2 2 3 3 4 4 5 5

动态编程

每一对都将从正确数字的角度创建。假设 *DP*[*i*][*j*] 是在大小为 *i* 的前缀上创建 *j* 对的方法数（在上述序列中）。

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

1 1 2 2 3 3 4 4 5 5

动态编程

每一对都将从正确数字的角度创建。假设 *DP*[*i*][*j*] 是在大小为 *i* 的前缀上创建 *j* 对的方法数（在上述序列中）。

从前缀 *i* 到 *i* + 1，我们可以将该数字与之前未配对的数字之一配对 - 我们知道配对的方法有多少种。

M.排列平衡

简约外观

排列是将数值分配到不同的位置 - 假设有 *n* 个红色数字（位置）和 *n* 个蓝色数字（数值），并将它们排序为数字。

1 1 2 2 3 3 4 4 5 5

动态编程

每一对都将从正确数字的角度创建。假设 *DP*[*i*][*j*] 是在大小为 *i* 的前缀上创建 *j* 对的方法数（在上述序列中）。

从前缀 *i* 到 *i* + 1，我们可以将该数字与之前未配对的数字之一匹配 - 我们知道这样做的方法有多少种。我们也可以不配对，假设我们想将它与右边的数字配对。

M.排列平衡

平衡如何?

这种动态编程只需计算所有的排列组合

M.排列平衡

平衡如何?

这种动态编程只需计算所有的排列组合--我们可以扩展状态。

M.排列平衡

平衡如何?

这种动态编程只需计算所有的排列组合--我们可以扩展状态。

设 $DP[i][j][l]$ 是在大小为 i 的前缀上创建 j 对的方法数，使得它们成对的数字之和等于 l。

M.排列平衡

平衡如何?

这种动态编程只需计算所有的排列组合--我们可以扩展状态。

设 $DP[i][j][l]$ 是在大小为 i 的前缀上创建 j 对的方法数，使得它们成对的数字之和等于 l。

知道了右数之和，我们也就知道了左数之和，因此我们就知道了平衡。

M.排列平衡

平衡如何?

这种动态编程只需计算所有的排列组合--我们可以扩展状态。

设 $DP[i][j][l]$ 是在大小为 i 的前缀上创建 j 对的方法数，使得它们成对的数字之和等于 l。

知道了右数之和，我们也就知道了左数之和，因此我们就知道了平衡。

这种动态编程有 $0(n^4)$ 个状态，我们计算每个状态的时间不变。

那么第 k 个词性最小的词呢?

在后续位置，我们将尝试手动插入后续值，并计算有多少种方法可以完成排列，使其达到所需的平衡。

M.排列平衡

那么第 k 个词性最小的词呢?

在后续位置，我们将尝试手动插入后续值，并计算有多少种方法可以完成排列，使其达到所需的平衡。

如何计算以 $[4, 1]$ 开头的五元素排列，并达到所需的平衡?

M.排列平衡

那么第 k 个词性最小的词呢?

在后续位置，我们将尝试手动插入后续值，并计算有多少种方法可以完成排列，使其达到所需的平衡。

如何计算以 $[4, 1]$ 开头的五元素排列并达到所需的平衡?

1 1 2 2 3 3 4 4 5 5

M.排列平衡

那么第 k 个词性最小的词呢?

在后续位置，我们将尝试手动插入后续值，并计算有多少种方法可以完成排列，使其达到所需的平衡。

如何计算以 [4, 1] 开头的五元素排列并达到所需的平衡?

1 1 2 2 3 3 4 4 5 5

↓

(1, 4) (1, 2)

M.排列平衡

那么第 k 个词性最小的词呢?

在后续位置，我们将尝试手动插入后续值，并计算有多少种方法可以完成排列，使其达到所需的平衡。

如何计算以 [4, 1] 开头的五元素排列并达到所需的平衡?

1 1 2 2 3 3 4 4 5 5

↓

(1, 4) (1, 2)

2 3 3 4 5 5

M.排列平衡

复杂性

在每个位置上，我们将尝试最多插入一次每个值

M.排列平衡

复杂性

在每个位置上，我们将尝试插入每个值最多一次 - $O(n^2)$ 次，我们将运行 $O(n^4)$ 算法。

M.排列平衡

复杂性

在每个位置上，我们将尝试插入每个值最多一次 - $O(n^2)$ 次，我们将运行 $O(n^4)$ 算法。

最终的复杂度为 $O(n^6)$ 。