2013 年信息学奥林匹克

中国国家队候选队员论文集

教练：胡伟栋 唐文斌

编辑：胡伟栋

中国计算机学会

目录

登顶计划

湖南师范大学附属中学 彭天翼

1 题目背景

爬山是一项非常有益于身心健康的运动。可是在爬山时，我们应该采取怎样的策略，才能尽快登上山的顶端呢？基于这个有趣的问题，作者思考出了下面这样一道题目。

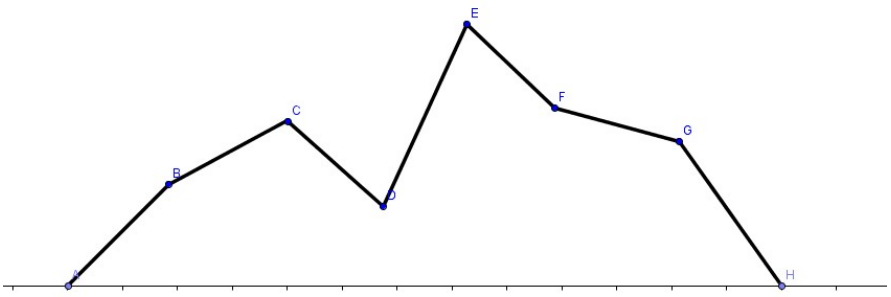
2 题目描述

【资源限制】

时间限制：2s 内存限制：512MB

【问题描述】

二维平面上的山脉由一系列顶点确定：$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$。相邻的两个顶点之间由线段相连（保证$x_i$严格递增），这样就构成了一座连绵的山脉，每个点的y值代表了该点的高度。如下图所示：



我们定义山脉的内部为顶点之间的折线与x 轴的所夹部分（不包括顶点之间的折线）。如果顶点A与顶点B之间的连线段没有穿过山脉的内部，则我们称顶点A能看见顶点B（或B能看见A）。

现在pty从某个顶点出发，想要登到山脉的顶峰（最高点），他只能在顶点之间的折线上行走。经过思考，他将采取如下的一种登山方式：

1. 站在出发点，向左右看去，如果此时能够看到的最高山峰在左侧，则向左侧走去，否则往右侧走去。

2. 在行走的同时，pty仍然观察着此时左右的最高山峰，一旦发现一座比之前看到的都要高的山峰，他将向此时的最高峰走去。

3. 如果存在某个时刻，pty所站立的位置比左右能看到的最高峰都要高，则他到达了山脉的顶峰，此时他的爬山过程结束。

pty想知道，采取如上的策略，从每个顶点出发，到达最高点的路程分别是多少？（平面中两点的距离等于它们之间连线段的长度）

【输入格式】

第一行一个整数n，表示山脉顶点个数。

接下来n行，第i行两个整数x_i,y_i，表示第i个顶点的坐标。

保证x_i严格递增，y_i互不相同(y_1, y_n除外)，x_i,y_i都为非负整数。保证y_1,y_n的值为0。

【输出格式】

输出共n行：每行一个实数。

第i行的实数表示从第i个顶点出发，到达最高点的路程。

如果输出与标准输出的误差不超过$1e-2$，则该测试点得满分，否则得0分。

【样例输入】

6

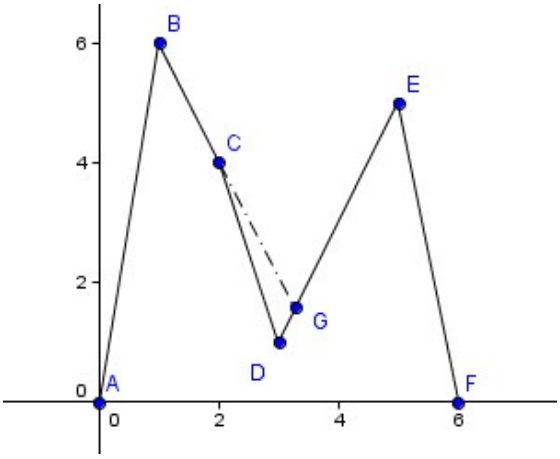
0 0

1 6
2 4
3 1
5 5
6 0

【样例输出】

6.08
0.00
2.24
6.52
9.87
14.97

【样例解释】



路线说明：
A点出发：$A->B$
B点出发：B
C点出发：$C->B$
D点出发：$D->G->D->C->B$
E点出发：$E->D->C->B$

F点出发：$F->E->D->C->B$

从D点出发时，看到的最高点是E，当步行至G点时，发现更高点B，转向后一直步行向B点。从其它点出发后都不需要转向。

【数据规模与约定】

所有的数据满足$xi, yi \leq 100000$。

测试点	限制
1-4	$n \leq 20$
5-8	$n \leq 70$
9-10	$n \leq 100000$且每个顶点都能直接看到最高点
11-14	$n \leq 30000$
15-20	$n \leq 100000$

3 题目分析

3.1 简化问题

题目中要求我们在爬山的任意时刻都要观察两侧当前的最高峰，让我们暂时将问题简单化：变为只有到达顶点时，才观察此时的最高峰。也就是说，现在的观测点，从无限个暂时变为有限的$O(n)$个顶点。

3.2 求一个顶点能看到的最高点

设i号点往左能看到的最高点为L[i]，往右能看到的最高点为R[i]。想象我们站在某个点上，初始时视线水平向左，接下来我们慢慢抬起头，直到我们的视线不再被某座山阻挡，此时我们便找到了L[i]。设当前的顶点为A，L[i]为B。则接下来的两个性质不言而喻：

- 前i个点都处于AB的下方（非严格）
- A，B为前i个点的上凸壳上两个相邻的点

由于A点一定是前i个点的上凸壳的最后一个点，所以B点就是这个上凸壳

的倒数第二个点[1]。于是我们只需要用一个栈维护好前i个点的上凸壳即可，每次加入第i+1个点，同时维护好凸壳的性质。由于总共只有O(n)个顶点，所以我们能在O(n)的时间内求出L[i]和R[i]。

3.3 构建图论模型

3.3.1 暴力做法

得出L[i]与R[i]后，我们有了一个颇为暴力的算法：枚举从每个顶点出发，模拟路线的变化，复杂度为$O(n^2)$或更高。这意味着我们需要继续挖掘其中被重复计算的冗余信息。

3.3.2 建树

令H[i]代表第i个顶点的高度，令T[i] = max(H[L[i]]，H[R[i]])，则T[i]意味着i点能够看到的最高峰的高度。

当我们从A点出发，到达第一个满足$T[B] >= T[A]$的B点时，接下来我们要走的路程和从B点出发走的路程将完全一样。所以A到最高峰的距离就等于B到最高峰的距离再加上A到B的距离。此时让B向A连一条有向边，边权为AB之间的距离。对所有的顶点做完这样的操作以后，我们将得到一棵树（因为最高峰没有入度，其余每个点的入度为1）。从树的根节点走向任意一个点的路径的边权和就等于该点到达最高峰的距离。dfs一遍即可得到每个点的答案。

3.3.3 需要解决的问题

对于一个A点，如何快速求出它往左（或者往右）第一个$T[B] >= T[A]$的点，这是一个十分经典的数据结构问题。我们可以用二分答案加区间RMQ算法解决。接下来介绍一种常数较小的做法。

对所有的顶点按从左往右的顺序建立双向链表。接下来按T值从小到大访问所有的点。每访问到一个点A，则往左第一个满足$T[B] >= T[A]$的B点就在此时双向链表中A的左侧。将A点和B点建边，然后将A点在双向链表中删除。

该算法的复杂度为排序复杂度。

[1]这样一个优美的结论着实令我惊讶。

3.4 回归原问题

至此，我们已经在O(nlogn)的排序时间内解决了一个被简化的问题，接下来让我们回归到原问题。

我们具备了在任意时刻观测最高峰的能力。这会给我们带来什么变化呢？

假设我们现在从A点出发，往右行走，走到某个点B上，发现了左侧的更高峰，转而向左行走。转折点B应该满足怎样的性质呢？设B所在的线段为P_1-P_2。B将满足：

- B点为A点左侧某两个点的连线（设为T_1，T_2）与P_1-P_2的交点
- 前A个点将位于T_1T_2连线的下方（非严格）
- T_1，T_2为前A个点的上凸壳上两个相邻的点

这意味着，有意义的观测点B将是上凸壳上某条边延伸以后与山的折线段的第一个交点。由于前i个点的上凸壳的总边数是O(n)的，所以有意义的观测点也是O(n)的[2]!

那么，如何快速求出这些观测点呢？

T_1与T_2的连线和P_1-P_2有交点，这意味着当P_2加入凸壳后，T_2将要被弹出，而P_1加入凸壳时，T_2没有被弹出。于是我们在维护凸壳的同时，通过求被弹出点和插入点的相关直线的交，就能得到所有观测点。复杂度仍然是O(n)的。

而有限的观测点的问题我们刚刚已经解决了。于是整个问题在O(nlogn)的时间内圆满解决。

4 总结

4.1 灵感来源

本题的灵感来源于对现实生活中爬山问题的思考。用算法知识对现实问题进行分析，不仅能提高我们的算法分析能力，还能反映学习就是为了“学以致用”的本质目的，并且能激发我们对信息学竞赛的兴趣，让我们乐在其中。希望这种思考方式能对大家有所启发。

[2]这又是一个令人兴奋的性质

4.2 解题思路

回顾我们的解题思路：我们先是将问题简单化，变无限为有限，接下来将问题化整为零，一步步地解决。这些都是在解题过程中的一般方法。

下面是作者对解题方法的一些体会，希望与大家分享：

在通常的解题过程中，我们需要在已知条件和所求问题中搭建起一座桥梁。可以利用已学的工具从已知条件顺推，看看我们还能得到什么。也可以从所求问题倒推，看看我们还需要知道什么。当顺推和倒推的大桥有了某个重合点，一条由条件通往问题的桥梁就建好了，问题也就解决了。

可解题的方法是千变万化的，不存在万能的解题秘籍。但值得庆幸是：“当你解决的问题越多，你解决下一个问题的可能性就越大。”所以只要我们勤于练习，乐于思考，解题能力就一定能不断迈上新的台阶。

方格取数

东北师大附中 王康宁

摘 要

2013年集训队互测论文，一道趣题的解题报告。

1 题目

1.1 题目描述

有一个$N*N$的方格表，每个格子里有一个数，第i行第j列的数等于A_i*B_j。一个人从左上角走到右下角，每次只能向右或向下走一步，他会把经过的数累加起来，并使这个总和最小。他决定走T次，第i次从第P_i 行第Q_i列的点走到第R_i行第S_i列的点，问每一次经过的数的最小总和。

1.2 输入格式

第一行包括两个整数N,T，分别表示方格表的大小和询问数量。
第二行N个整数，表示$A_1,A_2,...,A_N$。这一行的数是随机生成的。
第三行N个整数，表示$B_1,B_2,...,B_N$。这一行的数是随机生成的。
接下来T行，每行四个整数$P_i,Q_i,R_i,S_i(P_i \le R_i,Q_i \le S_i)$，含义如题所述。

1.3 输出格式

输出T行，每行一个整数，表示答案。

1.4 输入样例

3 4

1 2 3
2 3 4
1 1 1 1
1 3 1 3
1 1 3 3
1 1 3 1

1.5 输出样例

2
4
29
12

1.6 数据范围

对于20%的数据，$N \leq 100, T \leq 100$。
对于50%的数据，$N \leq 3000, T \leq 1000$。
对于70%的数据，$N \leq 50000, T \leq 20000$。
对于100%的数据，$N \leq 200000, T \leq 50000, 1 \leq A_i, B_i \leq 10^6$。保证询问合法，除样例外，所有的Ai, Bi均为随机生成的。

1.7 时间限制

2秒。

1.8 空间限制

512MB。

2 解题思路

2.1 算法一

对于每一组询问，用动态规划直接求解。

时间复杂度：$O(T*N*N)$
空间复杂度：$O(N*N)$
期望得分：20分

2.2 算法二

上一个算法的询问太慢了，现在需要加速询问。

考虑利用分治思想。每次按照中线将矩形分成两半，用动态规划求出中线上每个点到整个矩形上每个点的答案，并递归处理中线两侧的矩形。如果所询问的起点和终点在中线的两侧，则枚举经过的中线上的点，否则递归到某个子矩形。

如果每次都沿竖线切割的话，设预处理时间复杂度为$Time(N)$。则

$$Time(x) = 2 * Time(x/2) + O(N * N * x) \tag{1}$$

这样预处理时间复杂度为$O(N^3 * logN)$。而如果每次交替沿横线和竖线切割，则

$$Time(x) = 2 * Time(x/2) + O(x^3) \tag{2}$$

预处理时间复杂度为$O(N^3)$。

总时间复杂度：$O(N^3 + T * N)$
空间复杂度：$O(N^3)$
预处理复杂度过高，只能尝试其他方案。

2.3 算法三

考虑分块算法，如果将这个矩形分成S*S块，预处理每个点到块边界的每个点的答案，以及每个块内每对点的答案。

询问时两个点若不在同一块，则枚举其中一个点第一次经过块边界的位置，否则直接回答。

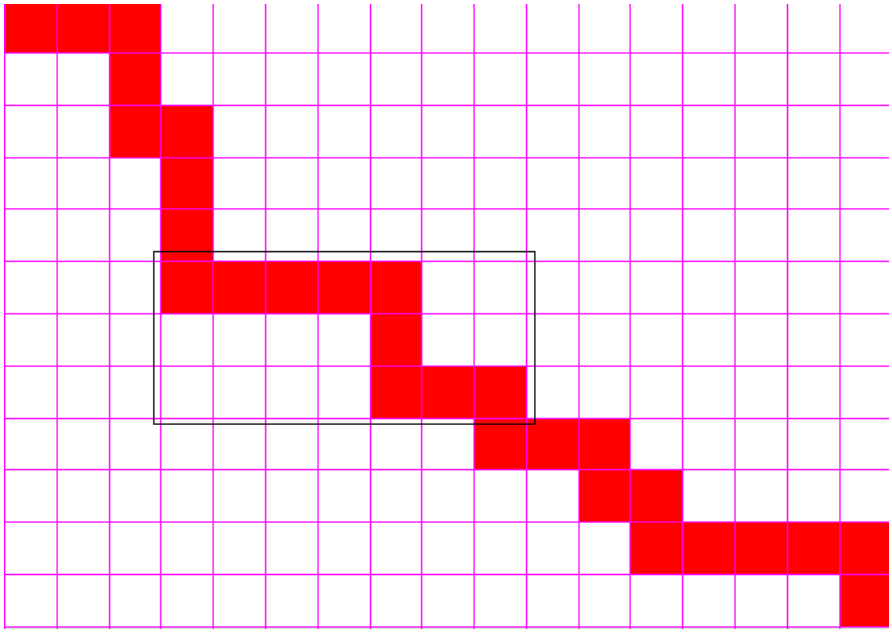
时间复杂度：$O(N^3 * (S + N/S^2) + T * N/S)$
空间复杂度：$O(N^3 * (S + N/S^2))$
复杂度同样过高。

2.4 改进方案

上述算法低效的一个主要原因是没有注意到题目中的一个重要条件：序列A和B均为随机生成的。

经过尝试和观察，可以发现答案的转折点不会很多。

注意一个转折点，和它相邻的两个转折点之间的部分。如图：

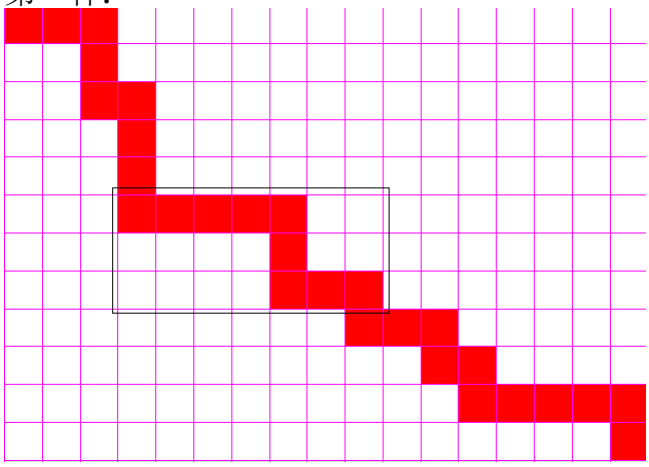


假如这个转折点所在列的权值不小于区域中其左边的某列权值，也不小于区域中其右边的某列权值。下面给出一种不更差的方案。

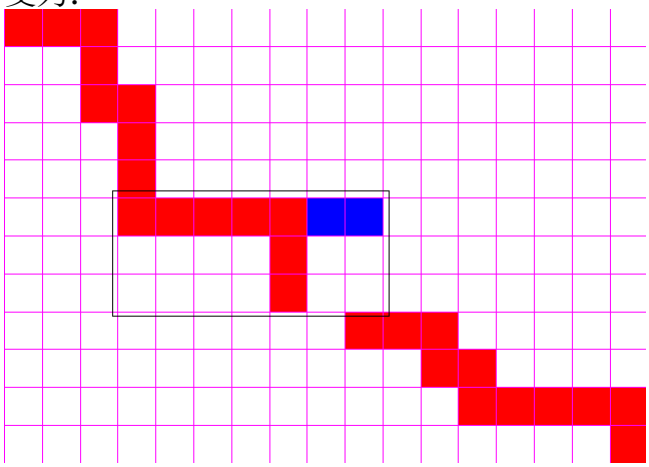
假设图中方框内第五列权值不小于方框最左和最右列的权值。

考虑下列两种变换方案：

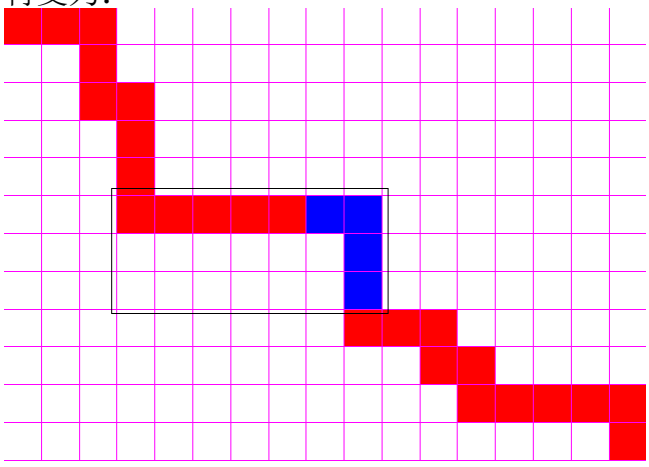
第一种：



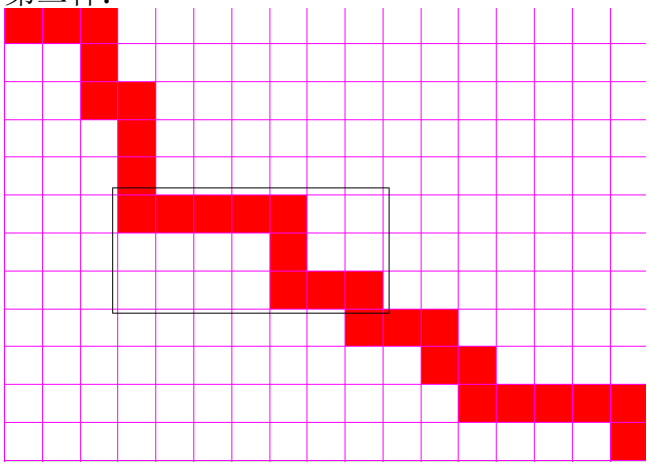
变为：



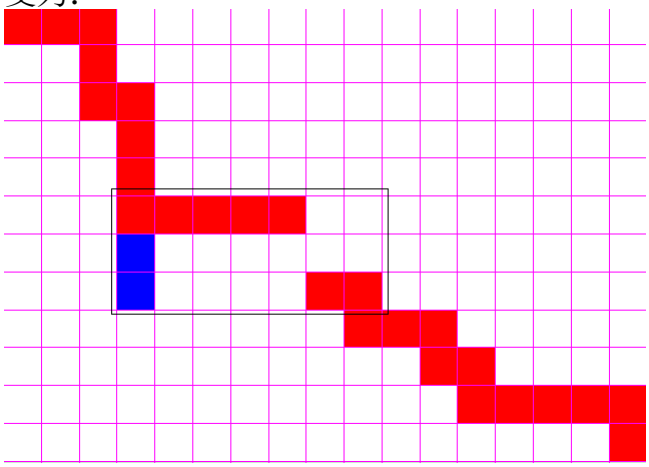
再变为：



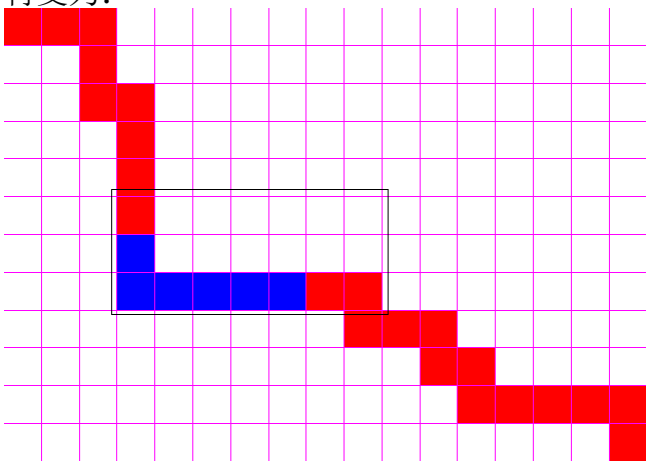
第二种：



变为：



再变为：



如果区域中上面一行权值小于下面一行的话，则经过第一种变换方案答案更优。

如果区域中上面一行权值大于下面一行的话，则经过第二种变换方案答案更优。

如果区域中上面一行权值等于下面一行的话，则经过任何一种变换方案之后答案不会变差。

这样我们就可以消除这个转折点了。

所以，设转折点分别在$L_1, L_2, ..., L_k$行（列），一定存在一种最优方案，使得：L_i行的权值，要么小于L_{i-1}至$L_i - 1$所有行的权值，要么小于$L_i + 1$至L_{i+1}所有行的权值。

若前者成立，则L_{i-1}行的权值一定小于L_{i-2}至$L_{i-1} - 1$所有行的权值，而L_{i-2}行的权值一定小于L_{i-3}至$L_{i-2} - 1$所有行的权值，...，所以L_i行权值比之前所有行的权值都小。

因此，任意一个转折点所在行（列）的权值，要比询问区域中左边所有行（列）权值都小，或者比右边所有行（列）权值都小。

而对于随机数据，第i行的权值比之前所有行权值都小的概率为$1/i$。

因此期望的转折点个数为：$1/1 + 1/2 + ... + 1/N = O(logN)$。

2.5 算法四

我们暴力找出所有可能存在转折点的行和列。

然后动态规划求解，此时，只有可能存在转折点的行和列才为有效状态，期望行数和列数都是$O(logN)$，而期望的乘积等于乘积的期望，所以期望状态数是$O(log^2N)$。

记录数组A和B的部分和来加速转移。

时间复杂度（期望）：$O(T * (N + log^2N)) = O(T * N)$

期望得分：50分

2.6 算法五

上个算法的瓶颈是找出可能存在转折点的行（列），即询问一个序列某区间比之前（之后）所有数都小的数，只要求出每个数之后（之前）第一个比它小的数即可，而这一工作可以使用单调队列来完成。

比如求每个数之后第一个比它小的数，可以从后向前扫描这个序列，将单调队列队尾不小于当前数的部分出队，则这个数之后第一个比它小的数就是当前队尾的数，然后将当前数入队。

预处理只要线性时间，每次询问时间为$O(log^2N)$。

时间复杂度（期望）：$O(N+T*log^2N)$

期望得分：100分

至此，我们已完满解决了这个问题。

3 致谢

感谢CCF提供学习和交流的平台。

感谢孔维玲老师和谷方明老师对我的指导。

感谢帮助过我的同学们。

two strings 试题讨论

杭州学军中学 罗干

1 问题描述

给定两个字符串A和B，有五种操作:
操作1 在A串开头添加一个字符；
操作2 在A串结尾添加一个字符；
操作3 在B串开头添加一个字符；
操作4 在B串结尾添加一个字符；
操作5 询问当前的B串在当前A串中出现的次数。
保证字符均为小写字母，且A、B串初始非空。

1.1 输入输出

1.1.1 输入格式

第一行第二行分别为初始的A串和B串，第三行一个整数m，表示操作的次数，接下来m行，每行表示一个操作，每行第一个数为一个在1-5之间的数字，若其值不为5，则在数字后会有一个小写字母。

1.1.2 输出格式

对于每个询问，每行输出一个整数，表示B串在A串中出现的次数。

1.1.3 数据规模

10%的数据中，最终A串和B串长度之和小于等于200，操作数小于等于10。
30%的数据中，最终A串和B串长度之和小于等于2000，操作数小于等于1000。

100%的数据中，最终A串和B串长度之和小于等于200000，操作数小于等于200000。

1.2 样例

1.2.1 输入

ababc
a
7
5
4 b
5
3 a
1 a
5
5

1.2.2 输出

2
2
1
1

1.2.3 样例解释

第几次询问	A串	B串	B串在A串中出现的次数
第1次	ababc	a	2
第2次	ababc	ab	2
第3次	aababc	aab	1
第4次	aababc	aab	1

1.3 朴素算法

1.3.1 模拟法

模拟每个操作，对于每次询问，直接枚举B串在A串中的位置，然后直接比较。

1.3.2 KMP优化模拟法

在模拟法的基础上，对于询问我们用KMP算法去匹配。

1.4 离线优化模拟法

把所有有关A串的操作全部读入组成一个”大A串”。

那么每次询问时当前的A串就相当于询问”大A串”的某一段中几个有个B串。

每次有三四两种操作就重新做一遍KMP，记下哪几个位置与B串匹配。

那么询问就变成了这个几个位置有几个在L R-Len(B)+1之间。

因为L递减、R递增，所以询问就可以O(1)解决。

1.4.1 各操作时间复杂度

操作1 O(n) →O(1)
操作2 O(n) →O(1)
操作3 O(n)
操作4 O(n)
操作5 O(n) →O(1)

1.5 后缀数组离线处理算法

1.5.1 在线→离线

利用刚刚的离线的思想，将A、B串最终的结果串并在一起，并在中间添加一个分隔符(如‘+’)，用后缀数组来维护后缀之间的关系。

举个例子：

大A串:ABABC
大B串:ABA
最终串: ABABC+ABA
后缀数组排序后:
A
ABA
ABC+ABA
ABABC+ABA
BA
BABC+ABA
BC+ABA
C+ABA
+ABA

1.5.2 求出可行的后缀区间

当前B串在A串第i到第i+Len(B)-1处出现，等同于当前B串与A串第i个后缀的最长公共前缀长度(即Lcp)≥Len(B)。

我们可以用倍增预处理height数组的方法，在log(n)的时间内找到这个区间。

1.5.3 统计合法的A串起始位置

我们定义合法位置为:对于大A串的位置i，[i,i+Len(B)-1]这一段都在当前的A串中。

1.5.4 树状数组维护

我们可以发现:任意一个操作只会改变至多两个位置的合法性，所以我们可以用树状数组来维护后缀数组中某一段有多少个合法位置。

1.5.5 各操作时间复杂度

操作1 O(log(n))

操作2 O(log(n))
操作3 O(log(n))
操作4 O(log(n))
操作5 O(log(n))

1.6 考查要点

对题目本质的深入思考
对各类数据结构的熟练掌握
对各类模拟算法的优化能力
对离线算法的应用能力

Catch The Penguins

杭州学军中学 张闻涛

1 问题简述

给出四维空间上N个企鹅坐标。

有Q个询问，每次给出一个点，问有多少企鹅每一维坐标都小于等于这个点。

样例输入:

```
1
0 0 0 0
2
1 1 1 1.0
1 1 1 -1
```

样例输出:

```
1
0
```

数据范围

共20个数据

数据1~2 N,Q≤5000

数据3~6 N,Q≤10000

数据7~14 N≤30000,Q≤10000

数据15~18 N≤10000,Q≤30000

数据19~20 N,Q≤30000

1.1 算法0

直接暴力枚举，对于每一个询问，一个for循环枚举每只企鹅，判断四维坐标是否都小于等于询问的坐标。

时间复杂度O(n^2)

期望得分10分。

1.2 算法0优化

把企鹅分别按四维坐标排序

对于每个询问，每一维在排好序的这四个数组里二分得出这一维小于询问坐标的企鹅的个数

取其中最小的进行枚举（常数优化）

期望得分30分。

1.3 继续优化?

如果要使用在线算法，似乎没有合适的数据结构能回答四维坐标的询问，难以再继续下去。

我们不妨试着考虑离线处理询问

1.4 分块算法

有了离线的想法，再利用分块的思想，不难想到一种复杂度稍优于暴力法的算法。

1.4.1 离散化+离线处理

分别将询问和企鹅按照第一维坐标排序

枚举询问。

1.4.2 对枚举的询问

加入所有第一维比它小的企鹅。

按照第二维排序

每加入sqrt(n)个企鹅，就暴力进行一次排序
其余：暴力。
这部分总复杂度 O $(n\sqrt{n})$。

1.4.3 进行一次暴力排序时

已经加入排序的企鹅的个数为M
分块
按照第三维坐标排序
按第四维维护前缀的权值线段树
函数式线段树

1.4.4 询问

整块：二分+查询
非整块：暴力

总复杂度O$(n\sqrt{n}log(n))$
期望得分：60~95

1.5 更优算法

通常，分块处理是在一些复杂度更优的算法存在，但思维复杂度比较高时的一个有效代替品。这里，我们可以稍加思考，找到一个更优的分治算法。

1.5.1

将询问和企鹅一起按第一维坐标排序。
按照第二维坐标进行“归并”。

1.5.2

假设排好序后的编号序列为id[L]~id[R]，其中有的是企鹅，有的是询问，令MID为其终点。

1.5.3

递归处理L~MID，MID+1~R部分的答案。（当然，当L，R相同时就可以直接返回）

1.5.4

我们发现我们还没有处理的情况就只有左半部分（L~MID）的企鹅对右半部分（MID+1~R）的询问的影响，这时左右两部分都已经按第二维坐标排好序（为什么已经排好序见后），而它们按第一维排的顺序已经不需要了，我们把左半部分和右半部分按第二维坐标进行归并（有序线性表合并），合并成新的表时，如果添加的是左边的企鹅，就把它三四两维形成的点插入一个“集合”，如果添加的是右边的询问，就在“集合”里询问有多少点在以它三四两维形成的点的左下方，把答案累加到这个询问里。

1.5.5

至于这个“集合”的实现，我们可以用树状数组（或线段树）套平衡树。

总时间复杂度O(nlog^3(n))

1.6 考察点

从在线到离线的思维转换
分块、分治等思想的运用
数据顺序的巧妙处理
数据结构的熟练掌握

浅谈分块思想在一类数据处理问题中的应用

北京市第八十中学　罗剑桥

摘 要

在竞赛和实践中，如何高效地组织和处理大规模数据是一个重要问题。本文主要讨论竞赛中一类与线性序列和树形结构有关的数据处理问题，引入分块思想的概念，探讨分块思想在数据的组织，数据的预处理，以及设计离线算法中的应用，并举例说明如何将数据分块处理而得到时间复杂度低的优秀算法。

1　引言

在一类数据处理问题中，我们需要频繁地统计在规模庞大的数据集合中，满足一定条件的元素的有关信息，如数目或者最值等。传统的数据结构，如线段树，将集合中的所有数据按一定的方式有序地组织起来，使得每次询问可以划分成若干有序部分的信息的和，从而提高效率。

本文介绍一种分块思想，同样是将数据有序化和层次化，但方式有所不同，而且扩展性更强，应用范围更广泛。

分块思想的核心是将一个集合划分成若干个规模较小的子集。一般地，我们倾向于将性质接近的元素分到同一个子集中，并且把每个子集的元素按一定的结构组织起来。

在数据处理问题中，分块思想有着广泛的应用，原因是分块的方法具有一些良好的性质：

若子集规模很小，对每个子集可使用关于子集规模的复杂度较高的算法。

若子集数目很少，可使用与整个集合规模有关的算法分别处理每个子集[3]。

[3]容易发现前两条性质之间存在矛盾。为了均衡，既不能让子集规模过大，也不能让子集数

若每个子集内部的元素具有共同的性质，可以分别设计有针对性的算法。

本文主要讨论分块思想在与线性序列和树形结构有关的数据处理问题中的应用。第 2 节介绍在典型的数据结构中分块的一般方法。第 3 节介绍分块思想在数据的预处理中的应用。第 4 节介绍分块思想在设计离线算法时的应用。

2　数据的分块化

这一节介绍两种常见的数据结构即线性序列和树形结构的一般的分块方法，并分别给出应用分块方法解决的例题。

2.1　线性序列的分块化

在线性序列中，数据集合的所有元素按一定顺序排列。典型的问题是频繁地统计一个连续的子序列中满足一定性质的元素的和，此处“和”表示答案能够直接或间接地分拆成不同部分的答案的和。下面先介绍一种常用的分块方法，然后举例说明如何使用分块的方法高效地解决此类问题。

分块的方法　从序列的第一个元素起，每连续的 S 个元素组成一个块。若最终剩余的元素不足 S 个，令它们组成一个块。

性质2.1. *设 N 为序列的长度，则上述方法中块的数目不超过 $\lfloor N / S \rfloor + 1$。*

性质2.2. *设一次询问操作查询序列中第 L 个元素到第 R 个元素的连续子序列（以下简称区间 $[L, R]$）的信息。那么除了完整包含在内的块中的元素以外，区间 $[L, R]$ 涉及的元素的数目不超过 $2S$。*

根据以上两条性质，如果能够在块的层次维护块内所有元素的信息的和，令块的大小 S 取合适的值，对于任意区间，就能快速找到对应的完整的块与多余的元素，并合并得到整个区间的信息。

目过大。这就是为什么分块算法的时间复杂度往往与 $\sqrt{N}$ 有关的原因。

例题一[4]

有一个 N 个数的序列。每个数都是整数。你需要执行 M 次操作。操作有两种类型:

– $ADD\ D_i\ X_i$ 从第一个数开始，每隔 D_i 个位置的数增加 X_i。

– $QUERY\ L_i\ R_i$ 回答当前序列第 L_i 项到第 R_i 项的和。

数据规模: $1 \leq N,\ M \leq 10^5$，任何时刻数列中每个数的绝对值不超过 10^9。

算法分析

最坏情况下，每次 ADD 操作和 $QUERY$ 操作涉及的元素的数目会达到 N 个，所以单纯的模拟会非常慢。考虑将序列分块，在块的层次记录每个块内所有元素的和。

引理2.1. *若块的大小设为 $\sqrt{N}$* [5]*，则每次查询的时间复杂度是 $O(\sqrt{N})$。*

具体的做法是在查询时枚举每个块，若完整包含在查询的区间内则答案直接加上此块的元素的和。而剩余的元素数目不超过 $2\sqrt{N}$，可直接枚举求和。

不过，修改操作的复杂度依然没有改进。这时还是要用到分块的思想。我们注意到一个事实：

引理2.2. *当且仅当 ADD 操作中 $D_i < \sqrt{N}$ 时，需要更新的元素的数目超过 $\sqrt{N}$ 个。*

因此将 ADD 操作分类：当 $D_i \geq \sqrt{N}$ 时，可以直接更新涉及的元素以及块的和，复杂度是 $O(\sqrt{N})$；当 $D_i < \sqrt{N}$ 时，我们可以只用数组记录每个 D_i 的 ADD 操作的修改量的和，此时更新操作是 $O(1)$ 的。

这时记录的每个元素以及块的和都仅考虑了第一类的修改操作。在查询的时候，我们还要枚举第二类的所有 D_i，并计算每个 D_i 的修改量的和对答案的影响。由于第二类的 D_i 不超过 $\sqrt{N}$ 个，一次查询的时间复杂度依然是 $O(\sqrt{N})$。

[4]**题目来源：原创**

[5]为了方便，文中使用 $\sqrt{N}$ 表示不小于 $\sqrt{N}$ 的最小整数，即 $\lceil\sqrt{N}\rceil$。

定理2.3. 可以在 $O((N+M)\sqrt{N})$ 的时间复杂度内解决此问题。

2.2 树形结构的分块化

在树形结构中，相邻的点之间有很强的相关性。一个直观的想法是希望将树划分成若干个规模较小的连通块。下面介绍一种树的分块方法，对于适当的 S，此方法能够将树划分为 $\Theta(N\ /\ S)$ 个大小为 $\Theta(S)$ 的连通块。其中 N 表示树中的点的数目。

首先引入有根树的 DFS 序列的概念。无根树可以将任意一个点作为根从而转化为有根树。

定义2.1 (树的 DFS 序列)**.** 维护一个序列，最初是空的。从根开始进行深度优先遍历，每遇到一个点进栈或出栈就把它加入序列的末尾。遍历结束后得到的最终序列称为 DFS 序列。

这样，对任意一个树我们能得到一个含有 $2N$ 个元素的 DFS 序列，树中的每个点对应 DFS 序列中的两个元素。DFS 序列具有良好的性质。

引理2.4. DFS 序列中相邻的两个元素的关系仅有三种：要么是同一个点，要么是父子关系，要么是兄弟关系。最后一种关系成立当且仅当 DFS 时前一个元素是出栈的点而后一个元素是入栈的点。

根据引理 2.4，可以证明下面一条性质。

引理2.5. DFS 序列中的第 u 项到第 v 项的连续子序列恰好对应从第 u 项对应的点 D_u 到第 v 项对应的点 D_v 的一条路径。路径上的点除了 D_u 和 D_v 的最近公共祖先（以下简称为 LCA）以外都至少在子序列中出现了一次。

于是得到我们想要的结论：

定理2.6. 使用上一节介绍的分块方法将树的 DFS 序列分成 S 块，那么序列中每一块的所有元素对应的点以及它们的 LCA 就是树上的一个连通块。连通块的数目不超过 $2N\ /\ S$。

这里介绍的分块方法实现起来很简单，也具有分块思想的一般性质即每个块的规模并不大而且块的数目并不多[6]。下面我们来看一道有关的例题。

[6]遗憾的是存在某些点出现在不止 2 个连通块中。

例题二[7]

给出一个 N 个点的树和一个整数 K。每条边有权值。你需要计算对于每个结点 i，其它 $N-1$ 个点到点 i 的距离中第 K 小的值是多少。

数据规模: $1 \leq K < N \leq 50\,000$，边的权值是绝对值小于 $1\,000$ 的整数。

算法分析[8]

朴素的算法是依次将每个点作为根，计算其它 $N-1$ 个点到它的距离，排序以后即可得到答案。时间复杂度是 $O(N^2 \log N)$[9]。

我们将树按之前描述的方法分成若干个大小为 $\Theta(S)$ 的连通块。考虑利用相邻的点之间的相关性更快地计算出以一个连通块内的每个点为根时的答案。

引理2.7. *设点 u 是不在当前连通块内的任意一个点。无论以块内的哪个点作为根，从点 u 到根的路径上经过的第一个块内的点是不变的。*

根据引理 2.7，将不在当前块内的所有点按到根的路径上经过的第一个块内的点（以下简称最近块内祖先）分类。显然，不在块内的点被分成至多 S 类。

引理2.8. *设点 u 和点 v 是不在当前连通块内且属于同一类的任意两个点。设点 p 是点 u 和点 v 的最近块内祖先。则以连通块内的任意一个点为根时，点 u 到根的距离不大于点 v 到根的距离当且仅当点 u 到点 p 的距离不大于点 v 到点 p 的距离。*

首先，将树遍历一遍，计算出每个不在块内的点到最近块内祖先的距离。对每一类的所有点，把它们按到最近块内祖先的距离从小到大排序。接下来，枚举连通块内的每个点作为根，分别计算答案。

定理2.9. *以连通块内的任意一个点 r_i 为根时，可以在 $O(S \log^2 N)$ 的时间复杂度内算出其它点到点 r_i 的距离中第 K 小的值。*

[7]**题目来源：经典问题**

[8]**这道题目有时间复杂度更优但十分复杂的树链剖分的解法，此处介绍的解法为笔者原创**

[9]如果使用基数排序，可以将复杂度降为 $O(N^2)$。

具体的做法如下。设当前以点 r_i 为根。为计算其它点到根 r_i 的距离中第 K 小的值，我们使用二分答案的方法。

设二分的答案为 X 。从点 r_i 开始遍历块内的每个点，计算它们到点 r_i 的距离，并统计其中不超过 X 的值有多少个。然后对块内的每个点 p ，设点 p 到根 r_i 的距离为 D_p，在以点 p 为最近块内祖先的一类点（按到点 p 的距离排序）的有序表中二分查找不大于 $X - D_p$ 的最大的数的位置，从而计算出这一类不在块内的点中与根 r_i 的距离不超过 X 的点的数目。

这样，设（除点 r_i 外）到点 r_i 的距离不超过 X 的点的数目等于 cnt。若 $cnt < K$，说明实际答案大于 X；否则说明实际答案不大于 X。于是每次二分可以将实际答案的范围缩小一半，从而以 $O(S\log^2 N)$ 的时间复杂度算出其它点到点 r_i 的距离中第 K 小的值。

考虑排序的时间复杂度和连通块内的点的数目，

引理2.10. *可以在 $O(N\log N + S^2\log^2 N)$ 的时间复杂度内计算出一个连通块内的每个点的答案。*

定理2.11. *如果令块的大小为 $\Theta(\sqrt{N/\log N})$ ，此算法总的时间复杂度为 $O(N^{1.5}\log^{1.5} N)$。*

2.3 进一步扩展

有兴趣的读者可以自行思考线性序列的分块方法如何推广到 D 维的情况，以及树的分块方法是否适用于更复杂的结构。

3 分块与预处理

预处理是指维护一个数据集合时，在所有询问操作以前对数据进行一些处理，往往记录许多附加信息，在询问时利用已处理过的信息提高时间效率[10]。

如果将分块思想与预处理相结合，能够高效地解决一类范围很广的集合查询问题，这就是本节将讨论的内容。这一节首先说明此类可以使用预处理优化的问题要满足的性质，然后详细介绍在序列和树结构中预处理的一般方法。

[10]如果有修改操作，还要考虑修改对预处理的信息的影响。

3.1 预处理的条件

设维护的数据集合为集合 S。

条件3.1 (可增量). 设两次询问的集合分别为集合 S 的子集 A 和子集 B。则从询问集合 A 的相关信息转化为询问集合 B 的相关信息的复杂度是关于集合 A 与集合 B 的对称差（即 $(A-B)\bigcup(B-A)$）的势的函数。即此转移的复杂度仅与对称差含有的元素的数目有关。

设集合 T 为所有询问可能涉及的集合 S 的子集的集合。

条件3.2 (可预处理). 对于给定的整数 K 和 L，存在一个规模不超过 K 的集合 T 的子集 P，使得对于集合 T 的任意一个元素 X，存在集合 T 的某个元素 Y 属于集合 P 并且集合 Y 与集合 X 的对称差的势不超过 L。

如果对数据集合的询问满足条件 3.1，并且存在合适的 K 和 L 使得数据集合满足条件 3.2，就可以使用分块和预处理的方法提高查询操作的时间效率。

此时我们找到条件 3.2 中的集合 P，并预处理集合 P 中的元素的相关信息。每次询问时找到与查询的元素[11] X 对应的元素 Y。因为元素 Y 的有关信息已经预处理过，并且询问满足条件 3.1，就能以 $O(f(L))$ 的复杂度[12] 将元素 Y 的信息快速地转移为元素 X 的信息。

3.2 线性序列的预处理

本小节考察一类在线性序列上频繁查询某连续子序列的信息并且满足预处理的条件的问题。首先介绍预处理的方法，然后给出一道例题。

我们首先将序列分块。设每个块的大小为 S，则块的数目是 $\Theta(N\ /\ S)$ 的。根据条件 3.1，对任意含有 len 个元素的区间，可在 $O(f(len))$ 的时间复杂度得到询问的相关信息。于是，我们这样预处理任意两块之间[13]的相关信息：

设块 A 为任意的块，考虑按在序列中位置从前到后的顺序依次计算块 A 与它之后的每个块 B_i 之间的信息。根据条件 3.1，计算出块 A 与块 B_i 之间的信息以后，可在 $O(f(S))$ 的复杂度内转化为块 A 与块 B_{i+1} 之间的信息。于是有

[11]此处“元素”指集合 T 的元素，实际上是数据集合 S 的一个子集。
[12]此处 $f(L)$ 指一个以 L 为自变量的函数
[13]从靠前的块的左端点到靠后的块的右端点的连续子序列

定理3.1. 仅考虑位置在块 A 之后的每个块 B_i。可以在 $O(f(N))$ 的复杂度内计算出块 A 与所有块 B_i 之间的信息，如果不考虑记录信息的复杂度。

若记录任意两块之间信息的复杂度可以不考虑的话[14]，我们枚举块 A_i，就能以 $O(Nf(N)/S)$ 的时间复杂度完成分块和预处理的过程。

根据性质 2.2，任意区间 $[L,\ R]$ 的信息可以由某段已预处理过的两块之间的子序列的信息通过添加不超过 $2S$ 个元素得到[15]。从而得到

定理3.2. 对于线性序列上区间统计类型的问题，若问题满足预处理的条件，首先对序列分块和预处理，以后在查询任意区间 $[L,\ R]$ 时，可以在 $\Theta(f(S))$ 的复杂度内得到区间 $[L,\ R]$ 的信息。

例题三[16]

给出一个 N 个数的序列。每个数是 $1 \sim N$ 之间的整数。你需要回答 M 次询问。每次询问给出三个整数 $L_i,\ R_i,\ K_i$，求区间 $[L_i,\ R_i]$ 中出现了恰好 K_i 次的数的数目。

数据规模: $1 \leq N,\ M \leq 10^5$。

算法分析

这道题目中，相邻的区间的答案不能通过记录一些附加信息而快速合并为它们组成的整个区间的答案，因此不能使用线段树等数据结构维护。但是询问满足这样的性质：

引理3.3. 对任意的两个区间 $[L_1,\ R_1]$ 和 $[L_2,\ R_2]$，如果已经记录在区间 $[L_1,\ R_1]$ 中 $1 \sim N$ 之间的每个数的出现次数以及出现次数为 $1 \sim N$ 的数的个数，可以在 $O(|L_1 - L_2| + |R_1 - R_2|)$ 的时间复杂度内转化为区间 $[L_2,\ R_2]$ 中 $1 \sim N$ 之间的每个数的出现次数以及出现次数为 $1 \sim N$ 的数的个数。

因而可以使用预处理的方法解决此问题。

[14]可以采用离线算法或别的手段。

[15]在得到区间 $[L,\ R]$ 的信息以后还要删除这些元素，恢复原来记录的信息。

[16]**题目来源：原创**

将序列分块，令每块的大小为 $\Theta(\sqrt{N})$，则块的数目为 $\Theta(\sqrt{N})$。我们能以 $\Theta(N\sqrt{N})$ 的时间复杂度得出任意两块之间 $1 \sim N$ 之间的每个数的出现次数以及出现次数为 $1 \sim N$ 的数的个数。但是这样记录的信息是 $\Theta(N^2)$ 的，必须想办法减少记录的信息量。

引理3.4. *将所有块从前到后编号为 $1 \sim \sqrt{N}$。则任意一个数 x 在第 i 块到第 j 块的出现次数等于 x 在前 j 块的出现次数减去 x 在前 $i-1$ 块的出现次数。*

我们只维护 $1 \sim N$ 之间的每个数在前 $1 \sim \sqrt{N}$ 块的出现次数，复杂度降为 $\Theta(N\sqrt{N})$。这样做与维护它们在任意两块之间的出现次数是等价的[17]。

引理3.5. *在序列中出现的次数不少于 $\sqrt{N}$ 的数至多有 $\sqrt{N}$ 个。*

在预处理时我们只记录两块之间出现次数在 $1 \sim \sqrt{N}$ 的数的数目。于是记录的复杂度是 $\Theta(N\sqrt{N})$。根据定理 3.2，我们能以 $O(\sqrt{N})$ 的复杂度正确地回答所有 $K_i \leq \sqrt{N}$ 的询问。

对于 $K_i > \sqrt{N}$ 的询问，我们提前处理所有出现次数大于 $\sqrt{N}$ 的数的出现次数的前缀和，询问时以 $O(\sqrt{N})$ 的复杂度枚举这些数，$O(1)$ 判断每个数在询问区间中的出现次数是否等于 K_i。

定理3.6. *上述算法可以在 $O(N\sqrt{N})$ 的时间复杂度内完成预处理。对于每次询问，可以在 $O(\sqrt{N})$ 的复杂度内得到答案。总的时间复杂度和空间复杂度为 $O((N+M)\sqrt{N})$。*

3.3 树形结构的预处理

本小节考察一类树形结构中频繁查询两点之间的路径的信息并且满足预处理的条件的问题。首先将线性序列中分块预处理的方法推广到树上，说明一种在树上标记关键点的方法，然后给出应用此方法的例子。

引理3.7. *树形结构中，任意两点之间的简单路径[18] 是唯一的。*

引理3.8. *对于有根树中的任意点 u 和点 v，设点 p 为它们的 LCA，则点 u 到点 v 的路径恰好由点 u 到点 p 的路径和点 p 到点 v 的路径两部分组成。*

[17]将预处理的第 i 块到第 j 块的每个数的出现次数转化为任意区间 $[L, R]$ 的每个数出现次数时，只需对前 j 块的每个数出现次数做修改，使用时与前 $i-1$ 块的每个数出现次数做差即可。

[18]指经过每个点至多一次的路径。以下“路径”均指简单路径。

我们希望将树中的若干个点标记为关键点，并预处理任意两个关键点之间的路径，使得树上的任意一条较长的路径 p 涵盖了一条已处理过的路径[19] ，并且路径 p 中除此以外的部分的含有的点数较少。下面介绍一种贪心的算法。

对一个有根树，按深度从大到小的顺序枚举树中的每个点并判断是否将其标记为关键点。将任意一个点 u 标记为关键点当且仅当点 u 的子树中存在一个点到点 u 的路径上没有关键点且路径上的点的数目大于 S。 S 是给定的数。

引理3.9. 设 N 为树的点集大小，则贪心算法标记的关键点不超过 N / S 个。

引理 3.9 成立的原因是每个关键点至少是 S 个非关键点到根的路径上的第一个关键点。

引理3.10. 设根结点的深度为 0，则任意一个深度不小于 S 的点的最近的 $S+1$ 个祖先结点（含该点自身）中至少有一个是关键点。

推论3.11. 对于树中的任意一条经过的点数大于 $3S$ 的简单路径，设点 u 和点 v 分别是路径的两个端点。则点 u 到路径上最近的关键点的距离[20]不超过 $2S-2$，点 u 与点 v 分别到路径上最近的关键点的距离之和不超过 $3S-3$。

在标记关键点的贪心算法中，我们只需对每个点记录它的子树内到它的由非关键点组成的路径经过的点数的最大值，就能判断是否要标记它。

定理3.12. 在 $\Theta(N)$ 的时间复杂度内可以将树中的 $\Theta(S)$ 个点标记为关键点。在 $\Theta(S^2 f(N))$ 的复杂度内进行预处理，那么任意一条简单路径中除去已处理过的部分外的点的数目为 $O(N / S)$。

例题四[21]

给出一个 N 个点的树。每个结点的颜色是 $1 \sim M$ 之间的一个整数。

定义从树上任意点 u 到点 v 的旅行收益为 $\sum_{i=1}^{M} \sum_{j=1}^{C_i} V_i W_j$。其中 i 是每种颜色， V_i 是颜色 i 的权值，C_i 是颜色 i 在点 u 到点 v 的简单路径上出现的次数，而$W_1, W_2, \ldots, W_N$是给出的整数。

[19]此条件也可以是任意一条路径与一条已经处理过的路径的对称差较小，但要判断的情况很多，所以程序实现会复杂很多。

[20]指经过的边的数目

[21]**题目来源：**NOI Winter Camp 2013, **糖果公园**

你需要执行 Q 次操作。操作有两种类型：
– $CHANGE\ x_i\ y_i$　　将点 x_i 的颜色修改为 y_i。
– $QUERY\ u_i\ v_i$　　查询从点 u_i 到点 v_i 的旅行收益。

数据规模: $1 \leq N,\ M,\ Q \leq 10^5,\ 1 \leq V_i,\ W_i \leq 10^6$.
本题的时限为 10 秒。

算法分析

将任意一个点作为根。令 $S = \Theta(N^{\frac{2}{3}})$，使用标记关键点的方法对树进行预处理，记录任意两个关键点之间的路径中颜色 $1 \sim M$ 的出现次数和旅行收益。预处理的复杂度是 $\Theta(N^{\frac{5}{3}})$。

对于查询点 u_i 到点 v_i 的路径的操作，先用倍增等高效的算法找到两个点的 LCA。接着，分别从点 u_i 和点 v_i 出发沿着路径移动，找到路径上最接近点 u_i 和点 v_i 的关键点。然后就能利用已经记录的信息，经过若干次增量操作在 $O(N^{\frac{2}{3}})$ 的复杂度内得到答案。

对于修改点 x_i 颜色的操作，枚举任意两个关键点，判断点 x_i 是否在这两个点之间的路径上，如果是的话就要更新记录的信息。

引理3.13. *点 x_i 在点 u 到点 v 的路径上当且仅当点 u 和点 v 的 LCA 是点 x_i 的祖先并且点 x_i 是点 u 或点 v 的祖先。*

如果预处理每个点在 DFS 中进栈和出栈的时刻，那么点 x 是点 y 的祖先当且仅当点 y 进栈的时刻在点 x 的进栈和出栈的时刻之间。我们在预处理时记录任意两个关键点的 LCA，就可以 $O(1)$ 判断点 x_i 是否在它们之间的路径上。因此修改操作的复杂度是 $\Theta(N^{\frac{2}{3}})$。

定理3.14. *可以在 $O((N+Q)N^{\frac{2}{3}})$ 的复杂度内解决此问题。*

4　分块与离线算法

有些数据处理问题并不要求对每次询问即时得到答案，而是一次给出大量

的询问，要求程序尽快计算出这些询问的结果并统一反馈。离线算法就是指预先知道所有询问的内容因而一起解决所有询问的算法。

在离线问题中，因为已经得知所有询问，我们可以把询问按一定的方式组织起来，从而充分利用询问之间的重叠信息。实际上是将所有询问作为数据集合的一部分处理，于是可以用原来处理数据集合的方法处理询问。

本节将用两个例子说明分块思想在设计离线算法时的应用。

例题五[22]

给出一个 N 个数的序列。每个数是 $1 \sim N$ 之间的整数。你需要回答 M 次询问。每次询问给出三个整数 $L_i,\ R_i,\ K_i$，求区间 $[L_i,\ R_i]$ 中出现了恰好 K_i 次的数的数目。

数据规模: $1 \leq N,\ M \leq 10^5$。

题目分析

将序列分块，每个块的大小设为 $\Theta(\sqrt{N})$。块的数目也为 $\Theta(\sqrt{N})$。

一次读入所有的询问，并把询问按查询区间的左端点 L_i 所在的块分类。然后分别处理每一类询问。

考虑左端点在块 A_i 的所有询问。我们把这些询问按右端点在序列中的位置从小到大排序。然后按顺序扫描从块 A_i 的左端点到序列末尾的每个元素，用数组维护每个时刻已扫描过的所有元素中出现次数为 $1 \sim N$ 的数的数目和 $1 \sim N$ 之间的每个数的出现次数。

对于每个询问 $[L_i,\ R_i]$，当我们扫描到询问的右端点 R_i 时可以在 $O(\sqrt{N})$ 的复杂度内将当前维护的信息转化为区间 $[L_i,\ R_i]$ 的信息。在记录询问的答案以后再将信息恢复。每个询问都会被处理一次。

定理4.1. *此算法的时间复杂度为 $O((N+M)\sqrt{N})$，空间复杂度为 $\Theta(N)$。*

在这个例子中，我们采用离线的办法解决了例题三中需要记录的信息量过大的问题。可以看到，分块与离线算法的结合和预处理的方法有一定相似之处。

[22]题目同例题三

而且能使用分块与预处理的方法解决的问题一般可以在离线情况下降低空间复杂度。

下面再看一个树形结构的例子。

例题六[23]

一个 N 个点的树。每个点有权值，权值是小于 2^{31} 的非负整数。你要回答 M 次询问，每次查询点 u_i 到点 v_i 的简单路径中出现的不同点权的数目。

数据范围： $1 \le N \le 40\,000,\ 1 \le M \le 100\,000.$

题目分析

首先将所有点的权值离散化[24]，之后可以认为权值是不超过 N 的整数。使用 3.3 节介绍的树上标记关键点的方法，令 $S = \Theta(\sqrt{N})$，则关键点的数目也为 $\Theta(\sqrt{N})$。根据推论 3.11，每个点与最近的关键点的距离为 $O(\sqrt{N})$。

读入所有的询问并把每个询问按到对应的路径端点 u_i 距离最近的关键点分类。分别处理每一类询问。

考虑处理以关键点 r_i 为与 u_i 距离最近的关键点的一类询问。

以点 r_i 为根深度优先遍历整棵树，同时维护当前已入栈而尚未出栈的所有点中每个权值的出现次数以及不同权值的数目。

引理4.2. *访问任意点 x_i 时，已入栈而尚未出栈的所有点恰好是从点 x_i 到根的路径上的所有点。*

引理4.3. *任意点 i 到任意点 j 的路径等同于点 i 到根的路径加上点 j 到根的路径再减去点 i 与点 j 的 LCA 到根的路径。*

对于每个询问 $(u_i,\ v_i)$，遍历到点 v_i 时维护的栈内的点的信息就是点 v_i 到根的路径上的所有点的信息。此时枚举从点 u_i 到根 r_i 的路径上的每个点 x_i。如果点 x_i 是点 u_i 与点 v_i 的 LCA 则不作改动，否则如果点 x_i 是 LCA 到根 r_i 的路径上的点则减去点 x_i 的权值，否则添加点 x_i 的权值。

[23]**题目来源：** Sphere Online Judge 10707, Count On A Tree II

[24]即建立 N 个点的权值的集合到不超过 N 的自然数的集合的单射。一般使用 $O(N \log N)$ 的排序方法即可。

点 u_i 与根 r_i 的距离是 $O(\sqrt{N})$ 的，因此有

引理4.4. 可以在 $O(\sqrt{N})$ 的时间复杂度内得到一次询问的答案。

定理4.5. 此问题可以在 $O((N+M)\sqrt{N})$ 的时间复杂度和 $O(N)$ 的空间复杂度内解决。

5 总结

形象的说，分块思想的实质是将复杂度为 $O(f(N))$ 的朴素算法转化为复杂度为 $O(g(S)+h(N/S))$ 的算法。其中 S 是块的规模，N / S 是块的数目。只是对于不同的问题，有不同的体现。

以例题一为例，这道题目的参考算法中两处体现了分块思想。一处是对于 $D_i < \sqrt{N}$ 的修改和 $D_i \geq \sqrt{N}$ 的修改分别用两种数据结构维护。另一处是在修改操作很频繁的情况下用分块的结构维护序列的段元素和。在这个特殊问题上，此方法比线段树更加高效，因为修改的数目达到查询的数目的 $O(\sqrt{N})$ 倍。后面的例题也是类似的。不过有时我们将一个问题分割为若干个本质相同的子问题，有时则将一个问题分离为若干个差异较大的子问题。

本文给出了线性序列和树形结构上分块的基本模式，对一类范围很广的问题都是有效的。但是究竟应该如何应用分块思想是因题而异的，同学们不要拘泥于这里给出的几种形式，而要勇于创新。当然，如果存在比分块的方法更高效的算法，也要学会选择，不能墨守成规。希望这些例子能对读者有所启发。

6 参考文献

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. “Introduction to Algorithms (2nd ed.)”, 潘金贵等译，机械工业出版社。

2. Jon Kleinberg, Eva Tardos. “Algorithm Design”, 清华大学出版社。

3. 吴文虎, 王建德.《世界大学生程序设计竞赛(ACM/ICPC)高级教程》(第一册), 中国铁道出版社。

4. 郑暾, 《平衡规划——浅析一类平衡思想的应用》。

5. 漆子超, 《分治算法在树的路径问题中的应用》。

6. 莫涛, 《“小Z的袜子”命题报告》。

7. 顾昱洲, 《“JZPLCM”命题报告》。

8. 徐捷, 《“图中图”命题报告》。

9. 许昊然, 《数据结构漫谈》。

7 致谢

感谢中国计算机学会提供学习和交流的平台。

感谢作者的指导老师贾志勇老师多年以来给予的关心和帮助。

感谢国家集训队教练胡伟栋和唐文斌的帮助。

特别感谢北京的范浩强同学在作者撰写此文的过程中给予的耐心帮助。作者在与他的交流中获益匪浅，受到不少启发。

感谢学长黄纪元同学的帮助。

感谢其他对作者有过帮助和启发的老师和同学们。

最后，感谢我的父母，感谢他们对我无微不至的关心和照顾。

搜索问题中的meet in the middle技巧

南京外国语学校 乔明达

摘 要

meet in the middle是一种常用的搜索优化技巧。部分搜索问题可以使用meet in the middle技巧大幅优化时间复杂度，从而在时限内搜索出结果。

本文分三部分。第一部分将给出一个抽象的有向图模型，以此为基础介绍meet in the middle技巧的思想，并运用该技巧给出针对有向图模型设计的算法。第二部分将简要论证第一部分给出算法的正确性、时间复杂度及其使用条件。第三部分将研究几个使用该技巧的例题，以这些例题为基础，概括得出一个方程模型，再分析该技巧的几个变种。

1 第一部分：有向图模型

1.1 问题的实质

一部分搜索问题的实质是：求有向图G中点A到点B的长度为L的不同路径的数目。这里“长度为L的路径”定义为一个有向边的序列$(E_1, E_2, \ldots, E_L)$，满足有向边E_1的始点为点A，有向边E_L的终点为点B，并且对于$i = 1, 2, \ldots, L-1$，E_i的终点等于E_{i+1}的始点。两条路径$(E_1, E_2, \ldots, E_L)$和$(E'_1, E'_2, \ldots, E'_L)$不同，当且仅当存在$1 \le i \le L$使得$E_i \neq E'_i$。

假设图G中点的最大出度为常数D，描述图G中一个点需要的空间为M，并且计算一个点的某一个后继需要的时间为T。

例如：每个点的标号是一个长度为N的数列$P_1, P_2, \ldots, P_N$，将$P_1, P_2, \ldots, P_{N-1}$或者$P_2, P_3, \ldots, P_N$翻转，可以得到这个点的后继。这里“翻转$P_1, P_2, \ldots, P_{N-1}$”表示：交换$P_1$和$P_{N-1}$，交换$P_2$和$P_{N-2}$，……，交换$P_{\lfloor \frac{N}{2} \rfloor}$和$P_{\lceil \frac{N}{2} \rceil}$。那么：每个点恰有两个后继；描述一个点需要用$O(N)$的空间来存储这个数列；计算某一个

后继需要$O(N)$的时间翻转数列。因此在这个例子中$D = 2$，$M = O(N)$且$T = O(N)$。

根据以上假设，我们分析求解以上问题的朴素DFS算法。这个算法限制DFS的深度不超过L，当搜索深度达到L的时候，将当前点与点B比较，如果是同一个点就将答案增加1。容易证明，在搜索过程中经过的点数不超过$1 + D + D^2 + \cdots + D^L = \frac{D^{L+1}-1}{D-1} = O(D^L)$，并且搜索$L$步之后到达的点数不超过$D^L = O(D^L)$，注意这里多次经过同一个点需要算多次。

由于一个点占据$O(M)$空间，所以比较一个点是否是点B需要$O(M)$时间。另一方面，在搜索中计算每个后继需要$O(T)$时间。因此，朴素DFS算法的时间复杂度为$O(D^L(M + T))$。当L较大的时候，朴素算法的运行时间将会很长，多数情况下不能在时间限制内求出答案，我们需要考虑优化搜索的方法。

在这类问题中，图G中的点非常多，甚至图G可能是无穷图，因此搜索过程中的重复子问题较少，动态规划不是一个有效的解决方法。由于最坏情况下我们不得不遍历整棵搜索树，所以用来优化搜索最优解问题的技巧也不适用于此类求路径数目的问题。我们需要从整体的搜索方法上进行变革。

1.2 meet in the middle的思想

meet in the middle的字面意义是“在中间相遇”。假设有两个人甲和乙，分别从A地和B地出发，向对方的出发点走去。当他们都走了$\frac{L}{2}$ 米的时候恰好“在中间相遇”，那么可以肯定，我们找到了一条从A到B长度为L米的路径。

下面我们考虑如何把这个思想应用到有向图模型。

1.3 针对有向图模型的算法流程

为了便于描述，我们构造一张新图G'。G'中的点与G中相同，但G中一条从X到Y的有向边在G'中对应一条Y到X的有向边。此外，令$L_1 = \lfloor \frac{L}{2} \rfloor$，$L_2 = \lceil \frac{L}{2} \rceil$。通过讨论$L$的奇偶性不难证明$L_1 + L_2 = L$。

算法的第一部分，在图G中从点A出发进行DFS，限制搜索深度不超过L_1。记录下搜索L_1步之后到达的所有点以及到达每个点的次数，记走L_1步到达点P的次数为$Count(P)$。如果搜索L_1步没有到达过点P，那么规定$Count(P) = 0$。

算法的第二部分，在图G'中从点B出发进行DFS，限制搜索深度不超过L_2。假设搜索L_2步之后到达点Q，我们就宣称找到了$Count(Q)$条长度为$L_1 + L_2 = L$的路径，答案累加$Count(Q)$。注意到：搜索L_2步之后我们可能多次到达同一个点，那么每一次到达都要累加$Count(Q)$。

从以上算法流程可以看出，meet in the middle是一种“双向搜索”的技巧。

2 第二部分：算法的分析

2.1 算法的正确性

令$L_1 = \lfloor \frac{L}{2} \rfloor$，$L_2 = \lceil \frac{L}{2} \rceil$。对于图$G$中的某一点$P$，设从点$A$到点$P$长度为$L_1$的不同路径有$f(P)$条，从点$P$到点$B$长度为$L_2$的不同路径有$g(P)$ 条。

引理1. *点A到点B长度为L的不同路径有$\sum_P [f(P)g(P)]$条。*

Proof. 考虑一条长度为L的路径$(E_1, \ldots, E_L)$，其中E_1的始点为A且E_L的终点为B。设E_{L_1}的终点为P，我们称这样一条路径是“途径”点P的。

根据定义，有$f(P)$种不同的路径$(E_1, \ldots, E_{L_1})$，有$g(P)$种不同的路径$(E_{L_1+1}, \ldots, E_L)$，并且这两类路径的任意一种组合，都能通过拼接得到一条满足：从A出发“途径”点P到达点B且长度恰为L的路径。

根据乘法原理，“途径”点P的路径共有$f(P)g(P)$条。由于从A到B长度为L的路径必然“途径”了某个点，所以不同的路径数目就是$\sum_P [f(P)g(P)]$。

□

引理2. *使用第一部分中给出的算法，得出的结果是$\sum_P [f(P)g(P)]$。*

Proof. 根据两次DFS，对于任意点P，算法第一部分中$Count(P) = f(P)$；算法第二部分中，搜索L_2步到达点P的次数为$g(P)$。

因此，对于$g(P) = 0$的点，算法并没有增加答案，相当于给答案加上了$f(P)g(P) = 0$；对于$g(P) > 0$的点，算法$g(P)$次给答案加上$f(P)$，即总共加上了$f(P)g(P)$。对所有点P进行求和，算法得出的结果就是$\sum_P [f(P)g(P)]$。

□

定理1. *第一部分中给出的算法可以正确地求出图G中从点A出发到达点B且长度为L的不同路径的总数。*

Proof. 结合引理1和引理2可知，算法给出的答案和实际上的路径数目都等于$\sum_P [f(P)g(P)]$。 □

2.2 算法的时间复杂度

下面我们来分析算法的时间复杂度。前文中我们已经假设图G中点的最大出度为D，现在我们假设图G'中点的出度均不超过D。另一方面，和前文中的描述相同，我们仍然假设在图G和G'中，描述一个点需要的空间为M，并且计算出某个点的指定后继的时间为T。

算法的时间复杂度主要取决于两部分的效率：一方面是DFS花费的时间，另一方面是程序计算和查询$Count(P)$的时间。以下分析中，由于D为常数，所以指数上的取整符号可以忽略，也就是说我们规定$O(D^{\frac{L}{2}}) = O(D^{\lfloor\frac{L}{2}\rfloor}) = O(D^{\lceil\frac{L}{2}\rceil})$。

根据以上假设，算法第一部分中DFS的代价为$O(D^{\lfloor\frac{L}{2}\rfloor}T)$，第二部分中DFS的代价为$O(D^{\lceil\frac{L}{2}\rceil}T)$。因此整个算法DFS花费的时间为：

$$O(D^{\lfloor\frac{L}{2}\rfloor}T) + O(D^{\lceil\frac{L}{2}\rceil}T) = O(D^{\frac{L}{2}}T)$$

在第一部分中，我们需要统计搜索$\lfloor\frac{L}{2}\rfloor$步之后到达每个点$P$的次数$Count(P)$；在第二部分中，我们需要对搜索$\lceil\frac{L}{2}\rceil$步之后到达的每个点$Q$查询$Count(Q)$。这要求我们寻找一个数据结构，能够快速地插入数据和查询某一数据出现的次数。可以发现，散列表是一个理想的选择。由散列表的性质可知，如果我们使用简单一致散列（simple uniform hashing），并使用链接法解决碰撞，那么平均情况下一次插入和一次查询操作仅需要$O(M)$时间。这是因为在本问题中，比较两个点是否相同需要$O(M)$时间。在第一部分中，我们需要对散列表进行$O(D^{\lfloor\frac{L}{2}\rfloor})$次插入操作；第二部分中，我们需要对散列表进行$O(D^{\lceil\frac{L}{2}\rceil})$次查询操作。因此计算和查询$Count(P)$的时间为：

$$\left[O(D^{\lfloor\frac{L}{2}\rfloor}) + O(D^{\lceil\frac{L}{2}\rceil})\right] \times O(M) = O(D^{\frac{L}{2}}M)$$

结合以上两点，算法的时间复杂度为

$$O(D^{\frac{L}{2}}(M+T))$$

2.3 算法的使用条件

一个问题要使用第一部分中给出的算法，首先必须符合有向图模型。其次，我们在分析该算法时间复杂度的过程中，实际上又加入了若干限制条件。

在之前的分析中，我们假设图G'中点的出度不超过图G中的最大出度。另一方面，图G'中点的出度实际上等于该点在图G中的入度。因此，一般需要保证图G中点的入度和出度是相同数量级的。

另一方面，我们假设能够在$O(T)$时间内访问G'中某个点的指定后继。这一条件等价于要求在$O(T)$时间内访问G中某个点的指定前驱。

在空间消耗方面，为了保证散列表期望$O(M)$的操作时间复杂度，我们需要耗费$O(D^{\frac{L}{2}}M)$的空间。

3 第三部分：应用

3.1 例题1：方程的解数[25]

3.1.1 题目大意

给出整数M，$k_1, k_2, \ldots, k_N$和$p_1, p_2, \ldots, p_N$，求方程$\sum_{i=1}^{N} k_i {x_i}^{p_i} = 0$满足$1 \leq x_i \leq M$ 的整数解的数目。

3.1.2 数据规模

$N \leq 6, M \leq 150$

3.1.3 分析

我们可以将本题转化为有向图模型，从而利用第一部分给出的算法求解。

我们用二元组$(Depth, Sum)$表示图G中的点，因此描述一个点只需要$O(1)$的空间。$Depth$表示之前的项数，Sum表示前$Depth$项的和。出发点为$(0, 0)$，终点为$(N, 0)$。

[25]题目来源：NOI2001第二试

对于满足$Depth < N$的点$(Depth, Sum)$而言，令$k' = k_{Depth+1}, p' = p_{Depth+1}$，则它恰好有$M$个后继，分别为$(Depth+1, Sum+k' \times 1^{p'}), (Depth+1, Sum+k' \times 2^{p'}), \ldots, (Depth+1, Sum+k' \times M^{p'})$。

由于减法是加法的逆运算，因此每个满足$Depth > 0$的点$(Depth, Sum)$也恰好有M个前驱。如果令$k'' = k_{Depth}, p'' = p_{Depth}$，那么这$M$个前驱分别为$(Depth-1, Sum-k'' \times 1^{p''}), (Depth-1, Sum-k'' \times 2^{p''}), \ldots, (Depth-1, Sum-k'' \times M^{p''})$。

为了计算一个点的后继或前驱，我们需要计算某个数的p_i次方，使用快速幂计算则时间复杂度为$O(\log MaxP)$，其中$MaxP$是$p_1, p_2, \ldots, p_N$中的最大值。

经过以上分析，我们可以直接使用第一部分给出的算法，时间复杂度为$O(M^{\frac{N}{2}} \log MaxP)$。

我们回过头来研究这个例题中使用上述算法的实际意义。下面我们研究$N = 6$的情况：

$$k_1 x_1^{p_1} + k_2 x_2^{p_2} + k_3 x_3^{p_3} + k_4 x_4^{p_4} + k_5 x_5^{p_5} + k_6 x_6^{p_6} = 0$$

朴素的算法是枚举方程左边的6个未知数，判断左边的结果是否等于右边的常数，这样时间复杂度为$O(M^6 \log MaxP)$，非常不理想。

仔细想想，上述方程看起来两边非常不平衡，6个未知数均在左边，导致我们不得不枚举6个未知数。将上述方程移项，可以得到：

$$k_1 x_1^{p_1} + k_2 x_2^{p_2} + k_3 x_3^{p_3} = -k_4 x_4^{p_4} - k_5 x_5^{p_5} - k_6 x_6^{p_6}$$

这样方程两边就“平衡”了。我们假设先枚举左侧的三个数，把枚举出的所有结果存在一张表里，再枚举右侧的三个数，每当计算得到和表中某个数相同的结果时，就得到了方程的一组解。在比较计算结果时，我们并不需要比较每一对数是否相同，我们可以利用数据结构（例如散列表）加快统计的效率，从而得到时间复杂度为$O(M^{\frac{N}{2}} \log MaxP)$的算法。实际上，meet in the middle技巧的关键就在于通过优化合并过程减少运算量。

3.2 例题2：ABCDEF[26]

3.2.1 题目大意

给出一个整数的集合S，求有多少组(a,b,c,d,e,f)满足$a,b,c,d,e,f \in S, d \neq 0$且$\frac{ab+c}{d} - e = f$。

3.2.2 数据规模

$|S| \leq 100$

3.2.3 分析

题目中的方程可以通过移项变为$ab + c = d(e + f)$，于是转化为与例题1本质相同的题目，可以得出时间复杂度为$O(|S|^3)$的算法。

3.3 例题3：EllysBulls[27]

3.3.1 题目大意

有一个N位十进制数A，可能存在前导零。现在给出M个条件，第i个条件包含一个N位数X_i和$Same(A, X_i)$的值。$Same(A, B)$表示N位十进制数A和B数字相同的数位数目，例如$Same(\text{"0312"}, \text{"0321"}) = 2$。如果有唯一的$A$满足要求则输出这个解，否则输出表示无解或多解的信息。

3.3.2 数据规模

$N \leq 9, M \leq 50$

3.3.3 分析

为了便于叙述，定义行向量

$$Sum = \begin{pmatrix} Same(A, X_1) & Same(A, X_2) & \cdots & Same(A, X_M) \end{pmatrix}$$

[26]**题目来源：** SPOJ

[27]**题目来源：** TopCoder Single Round Match 572 - Division I, Level Two

记X_i的第j位为$X_{i,j}$，A的第j位为a_j。定义函数$S(a,b)$，其中$0 \le a,b \le 9$，当$a=b$时$S(a,b)=1$，$a \ne b$ 时$S(a,b)=0$。

此外再定义行向量$Cost(i,j) = \begin{pmatrix} S(X_{1,i},j) & S(X_{2,i},j) & \cdots & S(X_{M,i},j) \end{pmatrix}$。这表示，如果$A$的第$i$位是$j$，那么这一位能与$M$个条件中的哪些数匹配。

根据题目的条件，我们可以列出以下方程：

$$Cost(1,a_1) + Cost(2,a_2) + \cdots + Cost(N,a_N) = Sum$$

我们仍然和前两题一样进行移项，令$Mid = \lfloor \frac{N}{2} \rfloor$，得到以下方程：

$$Cost(1,a_1)+\cdots+Cost(Mid,a_{Mid}) = Sum-Cost(Mid+1,a_{Mid+1})-\cdots-Cost(N,a_N)$$

因为a_i有10种可能的情况，所以搜索方程的左边有$O(10^{Mid})$种不同结果，右边有$O(10^{N-Mid})$种不同结果。由于方程两边都是$1 \times M$的行向量，所以每一步计算和在散列表中操作都需要$O(M)$的时间。因此，我们得到了一个时间复杂度为$O(10^{\frac{N}{2}}M)$的算法，可以通过本题的测试数据。

3.4 小结：方程模型

总结以上三个例题，可以发现题目都可以转化为一种方程模型：求方程$\sum_{i=1}^{N} f(x_i) = S$的解数。

在例题1和2中，$f(x_i)$和S为整数，例题3中$f(x_i)$和S为向量。在这三题中，我们的做法都是基于将方程变形为：

$$\sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} f(x_i) = S - \sum_{i=\lfloor \frac{N}{2} \rfloor+1}^{N} f(x_i)$$

我们先搜索方程的左边，将得到的结果记录在散列表中；再搜索方程的右边，并在散列表中查询之前存储的结果。可以看出，以上简要的解题过程与第一部分给出的算法非常相似。并且从例题1的分析过程可以看出，方程模型可以转化为第一部分中给出的有向图模型，因此方程模型是有向图模型的一种特殊情况。对于方程模型的题目，我们比较容易联想到meet in the middle技巧。

运用以上两个模型，我们可以快速地把具体问题模型化，并使用meet in the middle技巧求解。然而，有些题目不能转化为方程模型或有向图模型，从而不能直接使用第一部分中的算法。但是，这些题与上述两个模型具有相似之处，

我们仍然可以对第一部分的算法进行修改，从而使用meet in the middle技巧优化这些问题。下面我们来分析几个这样的例题。

3.5 例题4：Balanced Cow Subsets[28]

3.5.1 题目大意

给出正整数$a_1, a_2, \ldots, a_N$，先从N个数中选出若干个数（至少一个），再把这些数分为两部分，使得两部分的数之和相等，求第一步选数的方案数。

注意：可能有一种选数方案使得存在多种划分方式，但是只能算一种选数方案。

3.5.2 数据规模

$N \leq 20$

3.5.3 分析

我们将题目换一种等价的描述方式：对于数列$x_1, x_2, \ldots, x_N$，其中$x_i \in \{-1, 0, 1\}$。在满足$\sum_{i=1}^{N} x_i a_i = 0$的前提下，求存在多少个不同的非空集合$S = \{i | 1 \leq i \leq N, x_i \neq 0\}$。

这道题类似于之前给出的方程模型，但我们要求的并不是方程的解数，而是方程的解对应的集合的数目，因此算法需要做一些改动。

定义一个新的数列$y_1, y_2, \ldots, y_N$，规定当$x_i = 0$时$y_i = 0$，当$x_i \neq 0$时$y_i = 2^{i-1}$。于是集合S可以用$\sum_{i=1}^{N} y_i$表示。

设M是一个$[0, N]$范围内的整数，将$\sum_{i=1}^{N} x_i a_i = 0$变形为$x_1 a_1 + x_2 a_2 + \cdots + x_M a_M = -x_{M+1} a_{M+1} - x_{M+2} a_{M+2} - \cdots - x_N a_N$。记方程左边为*LeftSum*，右边为*RightSum*。

那么$\sum_{i=1}^{N} y_i$也可以分为两部分：$LeftSet = y_1 + y_2 + \cdots + y_M$和$RightSet = y_{M+1} + y_{M+2} + \cdots + y_N$。

在算法的第一部分中，我们依然像之前的题目一样先搜索方程左边的M个未知数，共有3^M种情况。我们不仅要记录方程左边的*LeftSum*，还要记录对应

[28]**题目来源**：USACO 2012 US Open, Gold Division

的*LeftSet*。

在算法的第二部分中，如果发现*RightSum*和某个*LeftSum*相等，那么相应的*LeftSet* + *RightSet*就是一个符合要求的集合。我们可以用一个0到$2^N - 1$的布尔数组来记录这些集合。

最后，扫描一遍整个布尔数组，就能统计得到答案。

不难发现，第一部分的时间复杂度为$O(3^M)$。第二部分中枚举了$O(3^{N-M})$种情况，但是对于每种情况需要访问对应的所有*LeftSet*。在最坏情况下，我们需要访问$O(2^M)$个*LeftSet*，因此第二部分的时间复杂度为$O(3^{N-M}2^M)$。最后扫描的时间复杂度为$O(2^N)$。

综上，算法的时间复杂度为$O(3^M + 3^{N-M}2^M + 2^N) = O(3^M + 3^{N-M}2^M)$。如果我们取$M = \lfloor \frac{N}{2} \rfloor$，那么就可以得到$O(6^{\frac{N}{2}}) \approx O(2.45^N)$的时间复杂度，足以通过本题的测试数据。但是实际上，如果我们取$M = N \log_{\frac{9}{2}} 3$，可以得到一个更好的时间复杂度$O((3^{\log_{\frac{9}{2}} 3})^N) \approx O(2.23^N)$。

例题4表明，我们在使用meet in the middle技巧时，可以在散列表中存储一些其它的信息，以便于问题的求解。另一方面，使用技巧时如果将搜索的未知数不均匀地分为两部分，可能得到更优的时间复杂度。

3.6 例题5：AlphabetPaths[29]

3.6.1 题目大意

有一个$R \times C$的矩阵，矩阵的每个格子里要么是空的，要么包含一个$0 \sim 20$的整数。求有多少条包含21个格子的路径满足：路径上相邻两个格子必须有一条公共边，并且$0 \sim 20$每个数都在路径上出现一次。

3.6.2 数据规模

$R, C \leq 21$

[29]**题目来源：** TopCoder Single Round Match 523 - Division I, Level Three

3.6.3 分析

这道题看上去并不是之前提到的方程模型，而是类似于第一部分中的有向图模型。但是在这道题中，路径的起点和终点是不确定的，各有$O(RC)$种选择。如果我们枚举了起点和终点，就已经做了$O(R^2C^2)$次枚举，再乘上搜索需要的时间，根本无法通过。

我们刚刚的想法是枚举起点和终点，再从路径的两端向中间搜索。我们发现，实际上“中间点”的选择也是$O(RC)$种，我们为什么不枚举中间点，从中间向两端搜索呢？

于是我们枚举路径的中间点Mid，也就是第11个格子。从Mid出发搜索10步，可以得到一条包含11个格子的路径P，设$Num(P)$为不包括Mid的10个格子里数值的集合。设Mid中的数值为X。那么假如存在两条路径P_1和P_2满足$Num(P_1) \cup Num(P_2) \cup \{X\} = \{0, 1, \ldots, 20\}$，那么将$P_1$和$P_2$拼接起来一定是一条满足要求的路径。注意到这里因为$0 \sim 20$各出现了一次，所以我们可以肯定$P_1$和$P_2$只在$Mid$处相交。我们和上一题一样使用一个二进制数表示集合$Num(P)$，就可以快速地判断$Num(P_1) \cup Num(P_2) \cup \{X\}$是否等于$\{0, 1, \ldots, 20\}$了。

我们估计一下上述算法的运算量。首先枚举的Mid数目为$O(RC)$，其次由于搜索10步，每步可以向四个方向走，所以路径数为4^{10}。实际上这个估计可以更加精确一些，从第二步搜索开始，“往回走”显然是不合法的，因此第二步开始可能的方向只有三个，从而实际的路径数不会超过4×3^9。对于每一条路径，进行散列表的操作和比较都只需要常数时间。综上，我们得到了一个时间复杂度为$O(RC \times 4 \times 3^9)$的算法。尽管$RC \times 4 \times 3^9$可能达到$21^2 \times 4 \times 3^9 = 34720812$，但是考虑到路径不能超出矩阵的边界或者到达没有数字的格子，并且如果路径中某个数字出现了两次可以直接剪枝，实际的搜索量会远小于以上估计。在实际测试中，按照上述算法实现的程序通过了所有测试数据。

例题5表明，在使用meet in the middle技巧时，如果起点和终点未知，并且中间点的数目较少，我们不妨尝试将“在中间相遇”变为“从中间出发”。

3.7 例题6：FencingGarden[30]

3.7.1 题目大意

有N根木棍，长度分别为$L_1, L_2, \ldots, L_N$。你可以选择先将某一根木棍分成两段，长度不一定为整数。从得到的木棍中选出一部分，靠着一堵无限长的墙围一个矩形区域。求当矩形区域的面积达到最大值时，矩形与墙平行的边的长度。

3.7.2 数据规模

$N \leq 40, L_i \leq 10^8$

3.7.3 分析

这道题与之前的所有题目都不同，并不是求方案数，而是求一个最优解。

这道题的一个难点在于切割木棍的方案非常多，枚举这个方案显然不现实。

由于我们把木棍分成了两段，但是围成的矩形有三条边，所以必然有一条边完全是由完整的木棍拼成的。这样一条边有两种可能：与墙平行或者与墙垂直。记这条边的长度为Len。

我们考虑这条边与墙平行的情况，记$S = \sum_{i=1}^{N} L_i$。那么矩形剩下的两条边长度必须相等。我们把仍未使用的木棍排列起来，它们的总长为$S - Len$。在$\frac{S-Len}{2}$处切下一刀，如果这里恰好是两根木棍的连接处，那么我们并没有改变木棍的数量，如果这里是某根木棍中间的某个点，那么我们就把这根木棍切成两段。无论如何，我们总能把剩下的木棍分成长度均为$\frac{S-Len}{2}$的两部分，从而得到一个面积为$Len\frac{S-Len}{2}$的矩形。

同理可得，如果一条长度为Len且与墙垂直的边完全由完整的木棍拼成，那么我们可以得到一个面积为$Len(S - 2Len)$的矩形。

由二次函数的性质可知，第一种情况下$Len = \frac{S}{2}$最优，第二种情况下$Len = \frac{S}{4}$最优。进而我们需要求出，由完整的木棍拼出的长度中，最接近$\frac{S}{2}$和$\frac{S}{4}$的长度分别是多少。

[30]**题目来源：** TopCoder Single Round Match 461 - Division I, Level Three

与之前的题目相同，我们仍然把N根木棍分为两部分，分别有M根和$N-M$根木棍。我们先搜索M根木棍，将得到的长度记录下来。之后我们搜索另外$N-M$根，假设得到了一个长度为T的部分，那么需要在之前搜索的长度中找到一个x，使得$x+T$尽量接近$\frac{S}{2}$或$\frac{S}{4}$，也就是要使x尽量接近$\frac{S}{2}-T$或$\frac{S}{4}-T$。一个较好的解决方式是：先将第一部分中得到的$O(2^M)$个长度排序，第二部分查询时使用两次二分查找。

将$O(2^M)$个数排序的时间复杂度为$O(2^M M)$，在其中进行$O(2^{N-M})$次二分查找的时间复杂度为$O(2^{N-M}M)$。如果我们取$M=\lfloor\frac{N}{2}\rfloor$，那么算法的时间复杂度为$O(2^{\frac{N}{2}}N)$。

例题6表明，运用meet in the middle技巧不仅可以解决计数类问题，还可以结合二分查找等方法，处理一些特殊的最优性问题。

4 总结

从之前的介绍可以看出，meet in the middle是一种十分有效的搜索优化技巧。它的核心思想是将问题分成两部分分别搜索，通过快速地合并两部分搜索结果达到优化算法的目的。虽然我们并不能因此得到一个多项式时间复杂度的算法，但是往往可以使时间复杂度的指数减小一半，在一些问题中这就能使得我们在时限内求出结果。

该技巧的优点在于算法容易实现，算法使用的搜索过程和朴素算法基本相同，仅需要使用一些较为简单的数据结构——例如散列表。该技巧的不足在于使用范围并不是很广，并且优化后的时间复杂度仍然是指数级别，能够有效解决的数据的规模并不能大幅增加。同时，算法必须消耗大量的空间。

有向图模型和方程模型概括了meet in the middle技巧能够解决的问题的普遍特征，使用这两个模型可以解决一类问题。另一方面，仍然有题目不能直接转化为上述模型，需要我们对meet in the middle技巧进行修改。由此看出，同一个技巧在不同题目中的使用方法也是多种多样的。我们需要根据情况灵活地使用meet in the middle 技巧，从而达到优化算法的目的。

5 感谢

感谢亲人和朋友对我的鼓励与支持!

感谢李曙、吴效时、史钋镭等老师对我的指导!
感谢贾志鹏、许昊然等众多学习信息竞赛的同学对我的帮助和启发!

参考文献

[1] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein, Introduction to Algorithms.

[2] vexorian, TopCoder Algorithm Problem Set Analysis.

[3] Bruce Merry, USACO 2012 US Open Gold Division, Balanced Cow Subsets Solution.

[4] Zhipeng Jia, TopCoder SRM 450∼499 Solution.

浅析信息学竞赛中概率论的基础与应用

江苏省扬州中学 胡渊鸣

摘 要

随着信息学竞赛内容的不断充实, 涉及概率的问题不断涌入. 具有一定水平的选手都具备了相当的解决概率问题的能力, 但是一旦询问起解法的稍严格的证明, 很多选手就束手无策了.

本文针对作者自身学习初等概率论过程中的体会, 以及和其他选手交流时发现的概率论难点进行重点分析, 内容包括概率论的基本概念、随机变量与期望、概率转移网络上的相关问题、Markov不等式以及Chebyshev不等式等, 从理论和实践两个紧密联系的层面帮助读者理清概率论的脉络, 深化对涉及概率论的解法的理解.

1 概率

1.1 什么是概率

如果读者仔细想过这个问题, 可能会发现这个问题不太好回答. 直观的说, 概率大的事情发生的可能性就大, 因此概率就是对事情发生的可能性的度量.

概率论的公理化体系会涉及很多测度论的内容, 有兴趣的读者不妨去查阅相关书籍. 由于本文只是从信息学竞赛的角度去讨论概率论, 往往并不涉及一些比较奇异的集合(比如可数个集合的并之类), 也不涉及某些连续的随机问题, 所以在信息学竞赛中用到的概率理论可以大大简化. 不过, 在此还是建议大家在闲暇时去研究一下概率论的公理化体系, 相信会对你对概率论的理解有很大帮助.

1.2 概率空间

竞赛中用到的初等概率论有三个重要成分, 即**样本空间**Ω, **事件集合**F和**概率测度**P. 我们常说的**事件**, 其实是Ω**的某个子集**(请读者时刻牢记这一点). 在竞

赛中, 往往可以认为Ω的每个子集都是一个事件(但在超出竞赛的某些领域, 这样做是有问题的), 所有事件的集合记为F(注意这里是集合的集合). 概率测度P, 是**事件集合到实数的一个函数**. 当然, 并不是所有的概率测度都是合理的. 一个合理的概率测度, 需要满足以下3条**概率公理**:

(1) 对于任意的事件A, 有$P(A) \geq 0$(非负性).

(2) $P(\Omega) = 1$(规范性).

(3) 对于事件A和B, 如果$A \cap B = \Phi$, 有$P(A \cup B) = P(A) + P(B)$(可加性).

我们称符合要求的三元组(Ω, F, P)为**概率空间**. 举例来说, 随机掷一个均匀的骰子, 考虑其向上的面, 我们有**样本空间**$\Omega = \{1, 2, 3, 4, 5, 6\}$, “ 奇数向上”的**事件**是$\{1, 3, 5\}$, **事件集合**$F$为$\Omega$的幂集(所有子集的集合), **概率测度**$P(A) = \frac{|A|}{6}$, 经过验证, 这是一个合理的概率空间.

相信这个概念大家都比较熟悉, 并且也觉得是理所当然的. 在此列出只是考虑到部分读者可能看到后面的内容时会对概率的基础感到困难.

1.3 条件概率

条件概率在OI中其实是非常常用的概念, 但是一不小心就会导致错误. 要解释这个概念并不难, 我们来看一个例子: α大学的学生中有99%是男生, β大学则有99%是女生. 假设两个学校的人数一样多, 我们在两所学校中随机的选出一个学生, 他(她)的性别是男性的概率有多大?

这个问题显然是古典概型的问题, 不难得出男生在所有学生中占50%, 所以选出男生的概率当然是50%.

如果我们与选出的学生进行交谈以后得知他(她)是α大学的, 那么这个概率显然变成99%了. 由此可见, **当我们得到了更多的信息以后, 事件的概率是会改变的**.

记已知事件B发生的情况下, 事件A发生的(条件)概率是$P(A|B)$. 设选出的人是α大学的学生这个事件为U_1, β大学为U_2, 选出的学生是男生的事件为S_1, 女生为S_2. 在上面的这个例子中, 我们可以写作$P(S_1|U_1) = 99\%$.

注意, 很多初学者在理解这个问题的时候, 总是有一个“先后性”的枷锁, 认为事件B一定先于事件A发生. 这在某些情况下是不妥的, 事件只是样本空间的

某个子集, 并没有时间这个属性, 本例中选择学生是一次性完成的. 虽然在某些情况下两个事件之间确实有可能有明显的时间先后关系, 如复习以后考试通过的概率和不复习考试通过的概率明显是不一样的.

计算条件概率的公式如下:

$$P(A|B) = \frac{P(AB)}{P(B)}$$

在研究概率问题时, 我们常常把$A \cap B$写成AB, 或者"A, B". 所以, $P(A \cap B)$, $P(AB)$, $P(A, B)$含义是一样的.

其实在考虑条件概率的时候, 我们是**把事件B看做了新的样本空间**, 由于事件和样本空间都是集合(别忘了同时样本空间本身也是一个事件), 这样做是没有问题的. 新的概率测度往往被称为条件概率, 条件概率公式实质上揭示了**两个样本空间上的概率测度的关系**.

1.4 全概率公式

全概率公式在概率问题中有着举足轻重的作用, 但是很多选手没有深入理解它的含义. 如果$B_1, B_2, B_3, ..., B_n$是概率空间的一个划分, 那么有:

$$P(A) = \sum_k P(A|B_k)P(B_k)$$

这个公式是用分类讨论方法研究概率问题时的基础, 几乎所有的概率问题都涉及这个公式. 刚才我们在得出男生在所有学生中占50% 这个结论的时候, 实际上已经用到了全概率公式了: $P(S_1) = P(S_1|U_1)P(U_1) + P(S_1|U_2)P(U_2) = 99\% \times 50\% + 1\% \times 50\% = 50\%$

1.5 Bayes公式

考虑等式$P(A|B)P(B) = P(AB) = P(B|A)P(A)$, 忽略中间一项, 将两边同时除以$P(B)$, 得到"Bayes公式":

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

该公式在计算**逆向概率**的问题上非常有用.

由于通常已知的往往是一系列的条件概率$P(B|A_k)$, 其中$A_1, A_2, ..., A_n$是样本空间的一个划分, 这种情况下还会将上式和全概率公式联合起来, 得到:

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{\sum_j P(B|A_j)P(A_j)}$$

再来看看上面讲到的那个例子, 现在问题变成:已知选到的同学是男生, 你能否求出他是α大学学生的概率?

现在要求的实际上是$P(U_1|S_1)$利用Bayes公式, 可以求得:

$$P(U_1|S_1) = \frac{P(S_1|U_1)P(U_1)}{P(S_1|U_1)P(U_1) + P(S_1|U_2)P(U_2)} = \frac{99\% \times 50\%}{99\% \times 50\% + 1\% \times 50\%} = 99\%$$

2 随机变量与期望

期望是概率题中又一常出现的概念, 它是随机变量的一个重要属性, 而这一点常常被选手所忽略, 导致期望成了缺乏随机变量这个基础的空中楼阁. 要理解期望到底是什么, 必须先了解随机变量.

2.1 随机变量的定义

随机变量(random variable)是一个有趣的词, 因为它其实并不是随机性的来源(随机性来自样本空间), 也不是变量, 而是定义在样本空间Ω 上的**确定的**实值函数. 不过很多情况下, 我们往往会抛弃样本空间而直接去考虑随机变量的属性(比如说期望), 以至于很多选手只知道期望的概念, 而搞不清随机变量到底是什么. 在这里, 有必要给随机变量下一个清晰的定义:

函数$X : \Omega \to \mathbb{R}$被称为一个随机变量.

在大多数情况下, 有了随机变量以后就可以抛弃对原本样本空间的关注, 而是集中注意于对于每个实值(注意, 本文只考虑离散的情况), 随机变量能够取得该值的概率. 这个过程实际上是将样本空间重新划分的一个过程, 将在这个函数下取得同一实数值的样本空间中的元素合并了. 当然, 读者之前在理解随机变量的时候, 很可能就是直接从其取每个值的概率入手的.

2.2 随机变量的期望

期望(expectation)是对随机变量表现出的平均情况的的一种刻画, 也是在竞赛中最频繁出现的概念. 很多选手是在脱离随机变量的情况下学习这个概念的,

这样做是不可取的. 对于一个随机变量, 定义其期望如下:

$$E[X] = \sum_{\omega} P(\omega)X(\omega) = \sum_{x} xP(X = x)$$

这里, $X = x$表示的是一个事件, 等价于集合$\{\omega|\omega \in \Omega, X(\omega) = x\}$. 前一个式子是从输入(样本空间)的角度入手的定义, 后一个式子则是从不同的输出入手, 将样本空间进行了划分(将输出相同的输入看成一个整体), 即将不同的输出值按概率加权后求和.

由于对于样本空间的某些元素, 随机变量的输出值很可能是相同的, 有时我们就可以不从样本空间的角度去考虑随机变量, 而是直接考虑“随机变量取某个特定的值”这个事件. 不过仍然需要注意的是, 样本空间和概率测度是随机变量的基石, 如果发现问题从期望这个高层无法解决, 我们就需要从底层入手了. 后面的几个涉及期望的性质的证明就是从概率空间入手的, 没有概率空间我们也没有办法证明两个随机变量的独立性.

2.3 随机变量的独立性与乘积的期望

随机变量的独立性是两个随机事件在其输出层面上的属性. 对于两个随机事件X_1, X_2和实数$x_1 \in X_1(\Omega), x_2 \in X_2(\Omega)$ 如果有$P(X_1 = x_1, X_2 = x_2) = P(X_1 = x_1)P(X_2 = x_2)$ 就称X_1, X_2 相互独立.

两个独立的随机变量的重要性质是:其积的期望等于期望的积, 证明如下:

$$\begin{aligned}
E[X_1X_2] &= \sum_{x\in(X_1X_2)(\Omega)} xP(X_1X_2=x) \\
&= \sum_{x\in(X_1X_2)(\Omega)} xP(X_1=x_1)P(X_2=x_2) \\
&= \sum_{x\in(X_1X_2)(\Omega)} \sum_{x_1\in X_1(\Omega)} xP(X_1=x_1)P(X_2=\frac{x}{x_1}) \\
&= \sum_{x\in(X_1X_2)(\Omega)} \sum_{x_1\in X_1(\Omega)} x_1\frac{x}{x_1}P(X_1=x_1)P(X_2=\frac{x}{x_1}) \\
&= \sum_{x_1\in X_1(\Omega)} x_1P(X_1=x_1) \sum_{x\in(X_1X_2)(\Omega)} \frac{x}{x_1}P(X_2=\frac{x}{x_1}) \\
&= \sum_{x_1\in X_1(\Omega)} x_1P(X_1=x_1) \sum_{x_2\in X_2(\Omega)} x_2P(X_2=x_2) \\
&= E[X_1]E[X_2]
\end{aligned}$$

2.4 期望的线性性质

线性性(可加性)是期望的性质中重要的一项. 不管两个随机变量X_1和X_2是否独立, 总有:

$$E[\alpha X_1+\beta X_2]=\alpha E[X_1]+\beta E[X_2]$$

这个性质在竞赛中的应用常常表现为将一个“大”的随机变量分成“ 小”的随机变量的和, 那么“大”的随机变量的期望就是每个“小”的随机变量的期望的和.

2.5 应用:Maze(Adapted from Codeforces 123E)

2.5.1 问题描述

给定一棵树, 从根节点S出发, 到叶子节点T点停止, 求DFS算法的期望步数. 每次DFS将从当前点出发未到达过的点“RandomShuffle” 以后按这个随机顺序往下试探. 注意, DFS时返回(弹栈)的过程也算一步.

2.5.2 解答

作为涉及期望的第一题, 我们有必要把一些细节搞清楚. 先明明确**样本空间**Ω是从S出发, T终止的所有路径的集合. **随机变量**X作用于Ω上, 表示每条路径

的长度. **概率测度**P为Ω的某个子集到$\mathbb{R}$ 的函数$P(A)=\sum_{\omega\in A}P(\omega)$. 这里$P(\omega)$是走出路径$\omega$的概率, 可以计算出来, 也可以避开计算这一概率.

我们要求的是$E[X]$. 不难发现枚举每条路径在数据规模大到一定程度的时候是不可行的, 必须另辟蹊径. 构造一些“ 小”随机变量X_e, 其中e是树的某条边, $X_e(\omega)$表示路径ω 中, 经过边e的次数(来回都算). 考虑到对于$\forall\omega\in\Omega, X[\omega]=\sum_e X_e(\omega)$, 简记作$X=\sum_e X_e$ 利用“ 期望的线性性质”, 有$E[X]=\sum_e E[X_e]$. 这时, 问题就转化成了求$E[X_e]$

我们称S到T的路径为主路径, 将边分成2类, 一类是在必经路径上的, 一类是不在必经路径上的. 先考虑必经路径上的边. 这类边走过的次数一定是1. 一方面由于边在必经路径上, 如果不走这条边就无法到达T; 另一方面一旦走过这条边, 就一定会在反向走这条边之前到达终点, 所以这条边必须且只能走一次. 在考虑不在必经路径上的边e, 这类边有两种情况, 一种是没有走到, 另一种是来回走了2次. 为了求出期望走的次数, 我们必须求出走0次和走2次的概率. 考虑必经路径上和这条边最近的点u, 假设这条边处于以点u的儿子v为根的子树中, 而终点处于以点u的儿子w为根的子树中. u不等于w, 否则u不是必经路径中离e最近的点. 这时我们发现, e到底走了0次还是2次完全取决于u 出发以后是先走了v 还是先走了w. 如果先走了v, 那么以v为根的子树中每条边都走了2次; 如果先走了w, 路径根本不可能进入子树v. 而在一个随机的排列中, 一个元素排在另一个元素前面的概率是$\frac{1}{2}$, 所以这条边走了0次和2 次的概率都是$\frac{1}{2}$(注意此时我们其实是在随机变量的输出层面上考虑问题). 于是, e走过的期望次数就是1.

综上, 每条边走过的期望次数都是1. 于是一个很神奇的结论是, 不管树的形状是如何的, 我们走过的期望边数都是$n-1$.

2.6 全期望公式

首先来考虑一个类似于条件概率的问题:如果给定了更多的信息, 如事件A一定发生, 样本空间Ω上的随机变量X会出现何种变化?

如果将这个受约束的随机变量记作$X|A$, 那么对于$\forall x\in X(\Omega)$, 我们有:

$$P((X|A)=x)=\frac{P(X=x,A)}{P(A)}$$

这就是Ω上的随机变量X在新的样本空间A上, 形成的新的随机变量$X|A$的全部信息.

对于随机变量X,Y, 一个让人有点摸不着头脑的公式是

$$E[E[X|Y]] = E[X]$$

根据上面的分析, $X|Y=y$是在样本空间$Y=y$上面的一个随机变量, 于是$E[X|Y=y]$是一个确定的数值. 如果不明确指定Y的取值, 则$E[X|Y]$是一个新的随机变量, 其期望表示在Y 的各种特定输出y的前提下, $E[X|Y=y]$按照$P(Y=y)$加权的和.

这个公式的意义类似于全概率公式, 使得我们可以对期望问题进行分类讨论. 全概率公式也是这个公式的基础, 下面的推导中第(5)到第(6) 步就用到了全概率公式.

为了便于理解这个公式, 证明如下:

$$\begin{aligned}
E[X|Y] &= \sum_{y\in Y(\Omega)} E[X|Y=y]P(Y=y) && (3)\\
&= \sum_{y\in Y(\Omega)}\sum_{x\in X(\Omega)} xP(X=x|Y=y)P(Y=y) && (4)\\
&= \sum_{y\in Y(\Omega)}\sum_{x\in X(\Omega)} xP(X=x,Y=y) && (5)\\
&= \sum_{x\in X(\Omega)} x\sum_{y\in Y(\Omega)} P(X=x,Y=y) && (6)\\
&= \sum_{x\in X(\Omega)} x\sum_{y\in Y(\Omega)} P(X=x|Y=y)P(Y=y) && (7)\\
&= \sum_{x\in X(\Omega)} xP(X=x) && (8)\\
&= E[X] && (9)
\end{aligned}$$

举个现实中的例子:在全年级的学生(样本空间Ω)中等概率(概率测度$P(A)=\frac{|A|}{|\Omega|}$)任选一人, 询问其上次考试的成绩$X$(随机变量), 求$E[X]$. 第一种方法是直接询问每个同学的成绩然后算出平均数作为$E[X]$. 第二种方法是请求每个班主任算出本班的同学的平均成绩, 然后再按照选到的同学属于每个班的概率(即该班人数除以年级总人数)加权求和.

第二种方法其实是构造了一个新的随机变量Y, $Y(\omega)$表示的是ω所在的班级编号. 这种方法算出来的就是$E[E[X|Y]]$. 显然两种方法求出来的期望是相等的, 这也说明了全概率公式的正确性.

读者可能会注意到其实这里的Y只涉及判断相等的操作, 其值域不一定是实数. 在后面的例子中会再次提到这一点.

3 概率转移网络上的相关问题

概率转移网络是OI中常见的一大类问题的模型, 现在来看看这个模型中的各种问题如何解决.

首先要明白什么是概率转移网络. 概率转移网络(以下简称网络)是一个**有向网络**, 由**点集(状态集)**V, **转移概率矩阵(一个二元函数)**$G : V \times V \to [0, 1]$, 以及**起点**$v_0$组成. 其中, 对于每个$u \in V$, 有$\sum_v G[u, v] \leq 1$.

有了数学上的定义, 再来看看这个模型的实际意义. 一个移动的质点, 一开始(时刻0)位于v_0. 对于每一个时间段, 如果质点位于顶点u, 那么对于任意$v \in V$这个点有$G[u, v]$的概率转移到v, 还有$1 - \sum_v G[u, v]$的概率会消失(或者理解为移动到了一个虚空的点).

熟悉**Markov链**的同学可能会觉得这个模型和Markov链非常相似, 不过考虑到OI中可能遇到的问题类型, 作者将Markov链进行了一些调整. 这个模型具有很强的普适性.

下面来看两个例题.

3.1 例题:Museum(Codeforces 113D)

Petya和Vasya在进行一次旅行, 他们决定去参观一座博物馆. 这座博物馆由m条走廊连接的n间房间, 并且满足可以从任何一间房间到任何一间别的房间, 两个人决定分头行动, 去看各自感兴趣的艺术品. 他们约定在下午六点到一间房间会合. 然而他们忘记了一件重要的事:他们并没有选好在哪儿碰面. 等时间到六点, 他们开始在博物馆里到处乱跑来找到对方. 不过, 尽管他们到处乱跑, 但他们还没有看完足够的艺术品, 因此他们每个人采取如下的行动方法:每一分钟做决定往哪里走, 有p_i 的概率在这分钟内不去其他地方(即呆在房间不动), 有$1 - p_i$的概率他会在相邻的房间中等可能的选择一间并沿着走廊过去. 这里的i指的是当前所在房间的序号. 每条走廊会连接两个不同的房间, 并且任意两个房间至多被一条走廊连接.

两个男孩同时行动. 由于走廊很暗, 两人不可能在走廊碰面, 不过他们可以从走廊的两个方向通行. 此外, 两个男孩可以同时地穿过同一条走廊却不会相遇. 两个男孩按照上述方法行动直到他们碰面为止. 更进一步地说, 当两个人在某个时刻选择前往同一间房间, 那么他们就会在那个房间相遇.

两个男孩现在分别处在a, b两个房间, 求两人在每间房间相遇的概率. ($n \leq 22$)

3.1.1 建模

我们尝试将这个问题转化为上面提到的概率转移网络. 不难发现, 由于两个人物的存在, 我们需要将状态集V定义为$V_0 \times V_0$, 其中V_0为题目中涉及的房间的集合. 通过题目中给出的条件, 我们不难求出每个状态(两个任务的位置)转移到另一个状态的概率, 即矩阵G. 同时, 初始状态$v_0 = (a, b)$. 由于问题的特殊性, 还要再定义停止状态集合$S = \{(a, a)|a \in V_0\}$. 接下来有两种方法来解决这个问题.

3.1.2 解法一:迭代法

如果将S中的状态的转移特殊处理, 将其转移概率除了转移到自己为1外其余全部为0, 不难发现, 我们要求的其实是经过足够多步骤的移动以后, 质点所在的位置(经过足够长时间, 质点一定停留在S中). 记网络中时刻t时, 质点处于每个点的概率为x^t, 其中x^t_u 为质点时刻t 在状态u 的概率, 不难发现, 有$x^{t+1} = Gx^t$. 初始状态x^0中, 只有$x^0_{(a,b)}$ 为1, 其余全部为0.

这时, 需要求的解是x^∞. 这个值没法在有限的时间之内算出来的, 不过可以发现, 当t足够大时, x^t和x^∞ 其实是非常接近的. 我们可以利用快速幂尽可能高地算出G^{2^K}, 然后利用$x^t = G^t x^0$得出一个相当好的近似解.

对于大部分比较弱的数据, 这种方法还是可以接受的, 但是一旦数据比较强, x^t收敛的速度不理想的时候, 该方法就无能为力了. 因此, 必须找一个更好的解法.

3.1.3 解法二:解线性方程组

区别于解法一, 我们采用另一种方法来处理S中的状态, 将其转移概率全部设为0, 即到达了S中的状态以后下一步必然转移到“虚空”. 这样做的目的是便于

列方程.

下面考虑, 落入虚空之前, 质点停留在每个点的次数的期望E_u, 即每个时刻质点位于这个点的概率之和$\sum_t x_u^t$(这里使用了**期望的线性性质**).

对于异于v_0的点u, 我们有:

$$E_u = x_u^0 + \sum_{t=1}^{\infty} x_u^t = x_u^0 + \sum_{t=1}^{\infty} \sum_v x_v^{t-1} G[v,u] = x_u^0 + \sum_v G[v,u] \sum_{t=0}^{\infty} x_v^t = x_u^0 + \sum_v G[v,u] E_v$$

显然, $x_{v_0}^0 = 1$, $x_u^0 = 0, u \neq v_0$.

这样我们就建立了一个方程组, 解之即可. 由于S中的状态只能停留一次, 所以质点停留在这些点的期望次数就等于质点最后一步停留在这个点的概率.

这个方法比上个方法强大许多, 可以通过全部数据, 时间复杂度$O(n^6)$, 不会超时. 相信不少选手知道这个算法并且能够正确实现甚至AC, 但是经过调查, 发现其实大部分选手并没有深入理解这个方程中的未知数的含义.

3.2 例题:走迷宫(SDOI 2012)

Morenan被困在了一个迷宫里. 迷宫可以视为N个点M条边的有向图, 其中Morenan处于起点S, 迷宫的终点设为T. 可惜的是, Morenan 只会从一个点出发随机沿着一条从该点出发的有向边, 到达另一个点. 这样, Morenan走的步数可能很长, 也可能是无限, 更可能到不了终点, 若到不了终点, 则步数视为无穷大. 但你必须想方设法求出Morenan所走步数的期望值.

3.2.1 解法

本问题和上述模型十分接近, 建模不存在难点. 要注意的是, 终点T的出边应该全部删去. 这道题的解法具有一定的技巧. 我们用$E[X_u]$表示从点u出发到达终点T期望需要走的步数. 特别的, $E[X_T] = 0$.

于是, 直觉上, 我们有$E[X_u] = \sum_v G[u,v]E[X_v] + 1$.

这个公式可谓是相当简单, 枚举下一步朝那个方向走, 然后按概率加权再加上走的这一步的1个步数.

可是, 这毕竟只是直观上的一个公式, 虽然它是正确的, 但是我们必须从理论上去证明它, 否则这是不能使人信服的. 证明的时候需要用到全期望公式$E[X_u] = E[E[X_u|Y_u]]$, Y_u是一个随机变量, 表示**从u出发以后第一步到达的节**

点(正如介绍全期望公式时提到的, 这里需要稍微扩展一下随机变量的概念, 这个随机变量是Ω到V的映射). 不难发现$E[X_u|Y_u = v] = E[X_v]$, 带入$E[X_u]$ 就能得到结果. 读者不妨仔细想一下这个问题里面的样本空间和概率测度具体是什么.

有了n个等式, 我们就可以通过解方程解决本题.

4 两个常用不等式

4.1 Markov不等式

对于非负随机变量X和正实数a, 总有以下不等式成立:

$$P(X \geq a) \leq \frac{E[X]}{a}$$

这个不等式说明了随机变量表现出一个较大值的概率和它的期望的关系, 在我们对一个随机变量只知道其期望的时候非常实用. 注意, 大多数情况下等号是取不到的, 而且左边会远小于右边.

4.1.1 应用:对最小圆覆盖随机增量算法运行时间的估计

知道这个算法的同学应该都明白这个算法的期望时间复杂度是$\Theta(n)$的. 但是, 总是有些同学会担心, 虽然期望复杂度是线性的, 某些情况下算法还是会退化成平方的, 导致超时. 这里, 我们来利用Markov不等式分析一下这种情况发生的概率. 用随机变量X表示算法的执行时间, 有

$$P(X \geq n^2) \leq \frac{E[x]}{n^2} = \frac{1}{n}$$

也就是说, 算法的时间复杂度退化成$\Theta(n^2)$的概率是$\Theta(\frac{1}{n})$级别的. 对于$n = 200,000$, $\frac{1}{n} = 5 \times 10^{-6}$, 仅仅是最粗略的估计, 算法退化的概率已经小于每个人死于交通事故的概率了! 不严格的说, 如果你还是不相信这个算法时间复杂度的稳定性, 你这辈子就别出门了. 如果再考虑别的因素, 退化的概率比上面算出的数值往往还会小几个数量级.

这个例子说明, 如果我们给出了算法的期望复杂度, 往往这个算法发生退化的概率是微乎其微的, 所以大家遇到某些运行时间含有随机因素的算法(随机快排, Treap等)大可放心使用.

4.2 Chebyshev不等式

我们称随机变量X的方差$Var[X]$为$E[(X-E[X])^2]$, 标准差σ为$\sqrt{Var[X]}$.

如果我们知道了随机变量的方差, 就可以进一步对随机变量以大于某一幅度偏移其期望的概率进行估计. 对于随机变量X和正实数a, 有:

$$P(|X-E[X]| \geq a) \leq \frac{Var[X]}{a^2}$$

上式也可以利用变量代换$a = c\sigma$写成:

$$P(|X-E[X]| \geq c\sigma) \leq \frac{1}{c^2}$$

上式表明在仅知道标准差的情况下, 随机变量偏移其期望c倍标准差的概率小于等于$\frac{1}{c^2}$.

其实, 这个不等式从直观角度进一步诠释了方差, 即"方差越小的随机变量偏离其期望的幅度越小".

5 总结

本文涉及的初等概率论的内容其实并不复杂, 但是很多初学者学习的时候往往理不清脉络, 忽视其中的逻辑关系, 认为很多简单的定义无需注意, 直接凭自己靠不住的直觉去理解. 本文从"概率" 这个概率论的基石出发, 通过离散型随机变量的讨论, 重点研究了随机变量基本的数字特征:期望, 并应用于解题过程中. 最后, 通过与均值(用一阶统计量——期望表示)和分散程度(用二阶统计量——方差表示)相关的关于随机变量估计的两个不等式说明两个统计特征的用途.

本文涉及的题目不多, 一些内容是选手们知道结论但是可能理解并不深入的细节问题. 很多概率论中的问题具有相当的哲学高度, 如偶然与必然、局部与全局、有限与无限相互之间的关系. 读者在做题之余, 不妨仔细去想想概率论的一些最为基础的问题, 知其然更知其所以然, 这样一定会利用概率论有效提高思辨能力.

由于作者才疏学浅, 如果文章中有错误还请读者们不吝赐教. 文中很多定理和不等式的证明读者可以在任何一本介绍概率论的书上找到, 在此限于篇幅, 不在赘述.

致谢

感谢CCF提供了展示自我的平台.
感谢倪震祥老师长期以来对我的指导.
感谢张煜承, 许昊然, 贾志鹏同学对我的论文提出宝贵意见.
感谢父母, 学校对我的培养.

参考文献

[1] 苏淳, 《概率论》, 科学出版社.

[2] 斯蒂芬 • 弗莱彻 • 休森,《数学桥——对高等数学的一次观赏之旅》, 上海科技教育出版社.

[3] http://tsinsen.com/A1475, Codeforces 113D Museum.

浅谈数据结构题的几个非经典解法

南京外国语学校 许昊然

摘 要

数据结构题是信息学竞赛中十分常见的题目，以线段树、平衡树、分块均衡、“莫队算法”为代表的经典解决方法也早耳熟能详、路人皆知。本文将介绍数据结构题的一些普及度较低，但十分有用的非经典做法。本文的主要介绍内容包括对时间分治算法及其扩展、二进制分组算法、整体二分算法，其中包含作者的很多原创内容和经验总结，希望能起到抛砖引玉的作用。

1 二进制分组算法、对时间分治算法及其扩展

1.1 从动态半平面交谈起

动态半平面交是十分经典的数据结构问题。有一个平面，我们要完成一系列操作，包括：

1. 插入一个半平面
2. 删除某次插入的半平面
3. 查询一个点是否在当前半平面的交集内

这个问题有一个经典但代码极为复杂的算法：利用线段树套可持久化的平衡树进行维护，通过可持久化平衡树的切割与合并来保证时间复杂度，合并时的分类讨论细节极多。

如果把动态半平面交问题比作一个大怪物，那么上述的做法就一定是一根大铁棍了，虽然确实能一棍子把怪物拍死，但未免过于蛮力了。实际上，通过对时间分治算法的扩展，我们可以巧妙地解决这个问题，如同一把锋利的小刀，虽然看似无力，却能直刺怪物的心脏。我们将在下文中逐步解决这个问题。

1.2 “修改独立,允许离线”的数据结构题

我们首先约定，数据结构题中，称要求我们作出回答的操作为“询问”，称会对询问操作的答案造成影响的操作为“修改”。

我们发现，有不少数据结构题都满足以下两个条件。此时，对时间分治算法将是非常有用的利器。

1. 满足修改操作对询问的贡献独立，修改操作之间互不影响效果。

2. 题目允许使用离线算法。

我们应当如何利用这两个条件呢? 我们不妨假设我们要完成的操作序列是S. 那么我们将整个操作序列S等分为两份。

我们可以发现以下两个性质：

1. 显然，后一半操作序列中的修改操作对前一半操作序列中的查询的结果不会产生任何影响。

2. 后一半操作序列中的查询操作只受2方面影响：一是前一半操作序列中的所有修改操作；二是后一半操作序列中，在该询问之前的修改操作。

容易发现，因为后一半操作序列的修改操作完全不会影响前一半操作序列中的查询结果，因此前一半操作序列的查询实际是与后一半操作序列完全独立的，是与原问题完全相同的独立子问题，可以递归处理。

接下来，我们考虑后一半操作序列中的查询。我们发现，影响后一半操作序列答案的因素中，第二部分“后一半操作序列中，在该询问之前的修改操作”也是与前一半序列完全无关的（为什么完全无关? 别忘了我们的前提，修改操作互相独立互不影响，前一半序列中的修改操作不会对后一半序列中的修改操作的效果产生影响!）。因此，这部分因素也是与原问题完全相同的完全独立的子问题，可以递归处理。

我们发现，影响后一半操作序列答案的因素的第一部分“前一半操作序列中的所有修改操作”虽然与前一半序列密切相关，但有一个非常好的性质，就是：影响后半部分任意一个查询答案的东西，始终是一样的! 也就是前半部分操作中所有的修改操作。这时，我们发现原问题的动态修改操作不再存在，而被转化为了离线的、与原问题同样规模的“一开始给出所有的修改”然后“回答若干询问”的更简单问题，从而简化算法。

我们分析一下时间复杂度。不妨设“解决无动态修改操作的原问题”的复

杂度为$O(f(n))$，那么由主定理易得，我们对时间分治的总复杂度将是[31]

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right)+O(f(n)) \\ \text{解得}\quad T(n) &\le O(f(n)\log n) \end{aligned}$$

因此，只要数据结构题满足我们上文假定的两个要求：修改独立，允许离线，我们就可以以一个log的代价，把原问题中的动态修改给去掉，变为没有动态修改的简化版问题，从而使问题得到极大简化。

【例1】共点圆[32]

有一个平面，我们需要支持以下操作（为简化问题，保证所有点都在x轴上方，且横坐标非0）：

1. 给定一点，插入一个圆心位于该点且过原点的圆
2. 查询某点是否在所有圆的内部（含圆周）

本题的官方做法是对圆进行反演后，转化为凸包维护问题，使用splay树动态维护全凸壳，要求支持插入新点和查询点是否在凸包内部的操作，代码较为复杂。官方做法的时间复杂度是$O(n\log n)$。

其实，本题有更简单的做法。我们不妨首先进行一些简单的转化。

“一个点(x_0,y_0)处于圆心(x,y)且过原点的圆的内部”等价于

$$\begin{aligned} (x_0-x)^2+(y_0-y)^2 &\le x^2+y^2 \\ \text{即}\quad 2x_0\cdot x+2y_0\cdot y &\ge x_0^2+y_0^2 \end{aligned}$$

也就是说，符合条件的圆心(x,y)必须位于半平面$2x_0\cdot x+2y_0\cdot y\ge x_0^2+y_0^2$内。于是问题等价于，我们要求支持：

1. 插入一个新点
2. 给定一个半平面，查询是否所有的点都位于这个半平面内

容易发现，要支持的操作满足“修改独立，允许离线”这两个性质（每个插入的点都具有对询问答案的一票否决权，不受其他任何因素干预）。因此，我们直接套用上述上文的对时间分治算法，题目被简化为：

[31]注：事实上主定理要求$O(f(n))\ge O(n)$时该证明才成立。但因为序列的长度已经达到了n，单次处理复杂度几乎不可能小于$O(n)$，如果真的不满足可以具体问题具体分析

[32]题目来源http://tsinsen.com/A1381

开始时给定一堆点，要求回答一系列询问：

给定一个半平面，查询是否所有的点都位于这个半平面内

这个问题做法非常简单，我们不妨考虑求出“在与半平面垂直的方向上的投影最小的点”。如果这个点也在半平面内，那么必然所有点都在半平面内。进而容易发现，只有在凸包上的点才可能成为我们想要求的那个点，因此我们对开始的那堆点求一个凸包。进一步观察发现，我们所求的那个点相邻的两条边的斜率必然正好“夹住”了半平面的斜率，因此我们对凸包的边的斜率进行排序后，二分查找半平面的斜率即可找到我们所求的点，而简单的在$O(n\log n)-O(\log n)$时间内判定。

时间复杂度是$O(n\log^2 n)$，代码非常简单。尽管时间复杂度比官方解法的复杂度多了一个log。

但对时间分治真的不能做到和官方题解一样的复杂度吗？答案是，可以！我们发现，算法的瓶颈在于求凸包、斜率排序和二分查找，这些步骤似乎都难以优化。但我们漏掉了一个重要的性质：我们这是分治算法！我们解决了前一半序列和后一半序列的两个子问题后，已经得到了前一半序列的凸包和后一半序列的凸包，为何不直接把它们线性合并呢？对斜率的排序也可以做类似归并处理。而查询怎么办呢？没关系，我们把查询也按斜率归并排序，在凸包斜率和查询斜率都有序的情况下，只需$O(n)$扫一遍就可以得到我们想求的东西了。于是我们经过诸多优化，终于做到了处理复杂度$O(n)$，乘上分治带来的log后，得到了与官方题解一样的复杂度$O(n\log n)$。当然，这个做法并没有太多的现实意义，因为实际由于常数原因，$O(n\log n)$做法并不比$O(n\log^2 n)$的算法快多少，而代码量就大很多了。

【例2】经典问题

这是我们最初讨论的问题的简化版本。给定一个平面，要求支持：

1. 插入一个半平面
2. 查询一个点是否在当前半平面的交集内

这道题是动态半平面交问题去掉删除操作后的版本。容易发现，这道题也满足“修改独立，允许离线”的性质。运用对时间分治，我们可以把原问题简化为：

开始时给定若干半平面，要求回答一系列询问：

给定一个点，查询该点是否在当前半平面的交集内

这个问题想必大家都会做，直接用半平面交算法求出所有半平面的交集，然后任意找其内部一点，半平面交的顶点关于该点排极角序。查询时，只需二分查找找到查询点所位于的极角区间即可。这个算法复杂度是$O(n\log n)$，因此算上分治复杂度后，解决原问题复杂度是$O(n\log^2 n)$。

相比直接使用平衡树维护半平面交的顶点序，这个方法代码量非常小。而且，也可以使用与上一题中类似的方法优化复杂度至$O(n\log n)$，与平衡树维护法的复杂度相同。

1.3 “修改独立,要求在线”的数据结构题

但有一些数据结构题，虽然满足“修改独立”的性质，但出题人采用各种方法（操作加密、交互题等）强迫选手采用在线算法，此时，对时间分治就无能为力了。但我们依然有一个好方法应对这类问题：二进制分组算法。

我们将一个数拆成2的幂次从大到小的和的形式。示例：

$$\begin{aligned} 21 &= 16+4+1 \\ 22 &= 16+4+2 \\ 23 &= 16+4+2+1 \\ 24 &= 16+8 \end{aligned}$$

这有什么意义呢？我们不妨用上述拆分方法对修改操作分组，比如说，对前21个修改，我们会把前16个修改分为一组，第17～20个修改分为第二组，第21个修改作为第三组（因为21=16+4+1）。我们对每组用数据结构进行维护，并支持在线的查询。不妨假设对n元素的组进行数据结构维护以支持在线查询的复杂度是$O(f(n))-O(g(n))$，按照上述的分组方法，显然最多只会分出$O(\log n)$组，查询时在各组内都查询一遍即可，因此单次查询时间复杂度$O(g(n)\log n)$。为什么这么做是对的？注意大前提，修改对询问贡献独立、修改操作互不影响效果，因此修改操作任意分组、分开处理不会影响答案。

而执行修改操作时怎么做呢？事实上，我们只需考虑当前拆分方案和增加一个修改操作后新的分组方案，显然有一些分组不再存在了，而有一个分组多出来了。我们直接删掉不再存在的分组，暴力建出新增的分组即可（如果感觉难以理解，可以看本文附带的演示文稿，其中包含一个图形化的动画解释）。

我们来分析一下这么做的时间复杂度。我们考虑我们重建的所有组的元素总数。实际上可以发现，当添加第k个修改时，我们重建的组的元素个数是$lowbit(k)$，也就是k的二进制表示下最右侧的1所代表的权值[33]。

考虑$lowbit(k)$的取值及其数量，我们很容易推出，总时间复杂度是[34]

$$\begin{aligned}&\sum_{1\le k\le n} O(f(lowbit(k)))\\ =&\sum_{1\le i\le \log n}\left\lfloor\frac{n}{2^{i+1}}+0.5\right\rfloor * f\left(2^i\right)\\ \le&\sum_{1\le i\le \log n} O(f(n))=O(f(n)\log n)\end{aligned}$$

因此，我们发现，通过二进制分组，我们以一个log的代价，把原问题的动态修改操作给去掉了，转化成了没有修改的简化版问题，并且可以支持原问题的在线算法要求。

【例3】经典问题

这是刚才讨论的例2的强化版本。给定一个平面，要求在线支持：

1. 在线插入一个半平面
2. 在线查询一个点是否在当前半平面的交集内

与上道题一样，这题的操作满足“修改独立”的要求，只是题目要求在线算法。因此我们直接套用二进制分组算法，原问题被转化为：

开始时给定若干半平面，要求在线回答一系列询问：

给定一个点，在线查询该点是否在当前半平面的交集内

这个问题可以直接使用例2中给出的解决方法，因为那个方法就是在线的，该方法时间复杂度为$O(n\log n)-O(\log n)$，因此算上二进制分组的复杂度后，本题算法时间复杂度为$O(n\log^2 n)-O(\log^2 n)$。虽然不及使用经典算法用平衡树维护顶点序的$O(\log n)$每次操作，但实际本算法常数比经典算法小不少，而且代码量非常小，实现简单，几乎没有分类讨论等容易导致bug的内容。

顺带提一个小扩展。如果我们不用二进制分解，而用三进制或更高进制的分解呢？可以发现，随着进制数b的升高，单次查询的复杂度也会升高，但

[33]示例：$22=(10110)_2$所以$lowbit(22)=(10)_2=2$

[34]注：与上文一样，这里也要求$O(f(n))\ge O(n)$时该证明才成立。但因为序列的长度已经达到了n，单次处理复杂度几乎不可能小于$O(n)$，如果真的不满足可以具体问题具体分析

修改的总复杂度会降低！这对于某些非常特殊的题目（比如说题目保证操作中98%都是修改之类）时可能会有所用处。

1.4 对删除操作和变更操作的支持

有一类数据结构题，题目允许离线算法，它的修改操作包括插入操作与删除操作，其中虽然插入对询问的贡献独立，插入操作之间也互不影响效果，但删除操作可以取消某个插入操作的效果，这使得修改操作不再完全独立。比如我们在文章开头提到的“动态半平面交”问题。

这类问题的经典做法往往极为复杂，大多需要树套树等复杂的数据结构，有时还需要可持久化数据结构的帮忙，代码量往往极为惊人，细节也往往十分繁杂。但只要充分发挥对时间分治的威力，我们也可以得到简单优美的做法。

【例4】动态半平面交问题

有一个平面，我们要完成一系列操作，包括：

1. 插入一个半平面
2. 删除某次插入的半平面
3. 查询一个点是否在当前半平面的交集内

这个问题就是我们在文章最初提到的问题。现在，我们终于要解决它了。我们不妨继续考虑对时间分治的做法。现在，修改操作（也就是插入操作和删除操作）不完全独立了，因为现在一个插入操作可能会被后面的删除操作撤销掉。我们不妨仍然试图用对时间分治算法来解决。

我们不妨依然把操作序列等分为两半。我们考虑前一半操作中的查询，它们的答案显然与后半部分的插入和删除操作都完全无关，因此它是与原问题完全相同的子问题，可以递归解决。

考虑后一半操作的查询，它们的答案只与两部分内容相关：

1. 前一半操作序列中的所有插入操作中在该询问之前未被删除的部分
2. 后一半操作序列中在该询问之前的且未被删除的插入操作

我们发现，因为后一半序列中的插入操作显然不会在前一半序列被删除，因此“后一半操作序列中在该询问之前的且未被删除的插入操作”这一部分与前一半操作序列完全无关，也是与原问题完全相同的子问题，可以递归解决。

因此，实际我们要处理的内容是：

“前一半操作序列中的插入操作中未被删除的部分”对“后一半操作序列的询问”的贡献。

因此，我们实际的任务是：

一开始给定一组半平面，要求支持：

1. 删除一个半平面

2. 查询一个点是否在所有的半平面中

很不幸，这个问题想要直接解决是非常困难的。但我们发现这个问题中只有删除操作，没有插入操作。于是一个经典的解决方法派上用场了，那就是：时间倒流！

因为我们使用的是离线算法，我们完全可以预知转化后的问题中“初始的半平面集”“删除和查询序列”等所有我们需要的信息。我们先求出最后没有被删除的半平面是哪些，然后逆序处理操作序列，这样，查询没有变，而删除半平面变成了插入半平面！

利用时间倒流，我们再次把问题转化成了：

有一个平面，要求支持：

1. 插入一个半平面

2. 查询一个点是否在所有的半平面中

是不是很熟悉呢？就是我们刚刚解决的例2的问题。再次对时间分治，可以在$O(n\log^2 n)$或$O(n\log n)$（利用归并）的时间复杂度内解决该问题。于是，原问题得到了解决。

时间复杂度分析：假设当前待处理序列长度为n，那么时间复杂度是

$$T(n)=2T\left(\frac{n}{2}\right)+O\left(n\log^2 n\right)$$

使用主定理易解得

$$T(n)=O\left(n\log^3 n\right)$$

此方法代码量远小于经典的线段树套可持久化平衡树的做法。而且，如果使用上文中较复杂的$O(n\log n)$算法解决转化后的问题，我们可以降低时间复杂度到与经典做法相同的$O(n\log^2 n)$。

通过充分发挥对时间分治的威力，我们以一个简单的做法解决了在经典做法中十分困难的问题。现在，只要插入操作贡献独立，且题目允许离线，即使有删除操作或变更操作（变更操作实际等于先删除原操作，再插入新操作），我

们也可以利用对时间分治与时间倒流，以2个log的时间复杂度代价，把题目简化为没有动态插入、没有动态删除、没有动态变更的完全静态版问题！

那么，如果这道题改成要求在线算法，对时间分治或二进制分组依然可以解决吗？对这个问题，我没有找到解决方法，因为第二步中的“时间倒流”已经决定了题目必须允许离线算法。但至少，在应用一次二进制分组算法进行转化后，我们要支持的操作已经没有动态插入了，对某些题目或许也是一个有用的性质。有没有更强大的解决方法尚待读者探索。

2 整体二分算法及其应用

二分答案是常见的技巧，但在数据结构题中，这个方法往往不奏效，原因是我们往往需要预处理一些东西才能快速回答当前二分点的答案，而任何一个答案都可能被二分到，因此预处理的复杂度就不可接受了。但我们也有一个好方法来应对这类问题，那就是：整体二分。

为了规范问题，我们不妨进行以下约定：称当前二分答案的值为“判定标准”，每个修改，结合具体询问和判定标准，可以算出该修改对该询问的判定答案的贡献。询问的判定答案是各个修改的贡献的和，而每个询问都有一个要求的判定答案，根据要求的判定答案与实际的判定答案的关系，我们可以确定询问的真实答案在当前判定标准之上还是之下。

所谓整体二分，需要数据结构题满足以下性质：

1. 询问的答案具有可二分性
2. 修改对判定答案的贡献互相独立，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线算法

询问的答案具有可二分性显然是前提。我们发现，因为修改对判定标准的贡献相互独立，且贡献的值（如果有的话）与判定标准无关，所以如果我们已经计算过某一些修改对询问的贡献，那么这个贡献永远不会改变，我们没有必要当判定标准改变时再次计算这部分修改的贡献，只要记录下当前的总贡献，在进一步二分时，直接加上的新的贡献即可。

这样的话，我们发现，处理的复杂度可以不再与序列总长度直接相关了，而可以只与当前待处理序列的长度相关。

定义$T(C,S)$表示当前待二分区间长度为C，待二分序列长度为S，不妨设单次处理复杂度为$O(f(n))$，则有[35]

$$\begin{aligned} T(C,S) &= T\left(\frac{C}{2},S_0\right)+T\left(\frac{C}{2},S-S_0\right)+O(f(S)) \\ \text{解之得}\quad T(C,n) &\le O(f(n)\log C) \end{aligned}$$

这样一来，复杂度就可以接受了。通过整体二分算法，我们仅以一个log的代价，便实现了在有预处理的限制下的二分查找，与正常情况下二分查找带来的复杂度相同。

上面的说明可能比较难以理解，因此我提供了整体二分的伪代码框架和例7的参考程序，请务必结合例题仔细理解其正确性与时间复杂度的证明。

【例5】经典问题

给定一个数列，要求回答一系列询问：

给定l,r,k，查询序列的第l个元素到第r个元素组成的子序列的第k大值

我们考虑用整体二分算法解决此题。不妨假设当前整体二分答案为s，则我们只需统计出各个询问区间中大于s的数有多少个即可。如果至少有k个，那么答案必然大于s；否则答案必然小于s，而且，大于s的那部分元素始终对这个询问有着相同的贡献，因此，我们直接把这个询问的k值减掉s，然后分治时不再统计大于s的元素。

那么如何统计出各个询问区间中大于s的元素有多少个呢？很多人的第一想法可能是：开一个数组，如果序列中某个元素大于s，就在数组中对应的位置标记为1，然后求一遍部分和即可。$O(n)-O(1)$，完美的复杂度，不是么？很不幸，还真不是。

如果这么做，我们分析一下总复杂度，会惊讶的发现，总复杂度是：（定义$T(C,S)$表示解决待二分区间长度为C，待二分序列长度为S的问题的复杂度）

$$\begin{aligned} T(C,S) &= T\left(\frac{C}{2},S_0\right)+T\left(\frac{C}{2},S-S_0\right)+O(n) \\ \text{解之得}\quad T(C,n) &= O(nC) \end{aligned}$$

[35]注：与上文一样，这里也要求$O(f(n))\ge O(n)$时该证明才成立。但因为序列的长度已经达到了n，单次处理复杂度几乎不可能小于$O(n)$，如果真的不满足也请具体问题具体分析

Algorithm 1 Divide_And_Conquer(Q, AL, AR)

```
# Q是当前处理的询问序列
# 其中WANT域为要求的判定答案，CURRENT域为已累计的总贡献
# [AL, AR]是当前的答案范围区间
if AL = AR then
  将Q中所有询问的答案均设为AL
  return
end if
AM ← (AL + AR)/2
# Solve是主处理函数，只考虑参数满足判定标准[AL, AM]的修改
# 此时询问序列Q中各个询问的答案被存储至ANS数组
Solve( Q, AL, AM )
# Q1, Q2为两个临时数组，用于划分询问
for i = 1 to Length( Q ) do
  if Q[i].WANT ≤ Q[i].CURRENT + ANS[i] then
    # 当前判定答案不小于要求答案，说明真实答案应当不大于判定答案
    向数组Q1末尾添加Q[i]
  else
    # 当前判定答案小于要求答案，说明真实答案应当大于判定答案
    # 这个算法的关键：把当前贡献累计入总贡献，以后不再重复统计！
    Q[i].CURRENT ← Q[i].CURRENT + ANS[i]
    向数组Q2末尾添加Q[i]
  end if
end for
# 分治，递归处理
Divide_And_Conquer( Q1, AL, AM )
# 注意，这里的调用是Q2而不是Q
# 因为Q1对这部分询问的判定答案的贡献已经被累加到CURRENT域中
Divide_And_Conquer( Q2, AM + 1, AR )
```

与暴力无异！哪里出问题了?

仔细看一看前文中的复杂度式子与这个式子的区别！这是一个初学整体二分的同学非常容易犯的问题。我们的处理复杂度，绝对不能和序列总长n线性相关！只能和当前处理序列区间的长度相关！因此，正确的做法是，对当前处理区间中所有大于s的元素的位置排序，然后二分查找每个询问端点所处的位置。这样复杂度是$O(L\log L)-O(\log L)$，其中L是当前处理序列区间的长度。于是，总复杂度是$O((n+Q)\log n\log C)$，其中C是元素的值的规模。

【例6】矩阵乘法[36]

给定一个矩阵，要求回答一系列询问：

查询某个子矩阵的第k大值

本题做法与上个例题非常相似，只是一维变成了二维。我们应用整体二分算法，问题转化为：

给定一个矩阵，其中若干元素被标记了，要求回答一系列询问：

查询某个子矩阵中被标记元素的个数

额外要求：复杂度不允许与矩阵大小线性相关，只能与被标记元素个数相关

这个问题可以使用二维树状数组解决。这里有个tricky的地方，因为我们的复杂度必须不能与总元素个数线性相关，而只能与被标记的元素个数相关，因此我们做完查询后不能简单地直接清空树状数组（这是与总元素个数相关的操作），而必须一次次逆向修改回去（这才是只与被标记元素个数相关的）以完成清空操作，以待下次继续使用。

时间复杂度$O((n^2+Q)\log^2 n\log C)$。

【例7】ZJOI2013 K大数查询[37]

有n个数集（数集中可以有重复元素）。要求支持：

1. 往第l个数集到第r个数集中都插入一个数k。$(1\le k\le n)$

2. 问你如果把第l个数集到第r个数集中的所有数取出来放在一起，那么这些数中第k大的数是多少。$(1\le k\le 2^{31}-1)$

[36]题目来源http://tsinsen.com/A1333

[37]题目来源：ZJOI2013 Day1 第一题

我们不妨考虑应用一次整体二分算法，那么问题转变为了：

有一个序列，要求支持：

1. 对该序列的一段元素加1

2. 查询该序列的一段元素的和

额外要求：复杂度不允许与序列长度线性相关，只能与操作个数相关

如果没有这个额外要求，想必大家都会做，就是经典的线段树+懒标记。但有了这个额外要求呢？可能有人要说了，使用动态新建结点的线段树云云……

这些都是经典做法，但有更简单的方法吗？如果你想了半天还是没有想法，那么请翻到本文第1页从头再读一遍吧:-)。这个修改操作是满足“修改独立”性质的！直接应用对时间分治，问题被转化为：

开始时给出若干区间，要求回答若干查询：

给定一个区间，要求开始时给出的各个区间与该区间交集的长度的和。

显然每个开始时给出的区间对答案的贡献是一个分段一次函数。直接排序后扫一遍即可。

时间复杂度$O(n\log^3 n)$，如果对时间分治时使用归并排序可优化到$O(n\log^2 n)$。通过整体二分与对时间分治的综合应用，我们成功解决了这道题目。

整体二分算法是一种很有效的处理手段，能大幅简化很多题目。比如说，我们可以把文章开头给出的题目再加强一下，改成要求支持：

1. 插入一个带权的半平面

2. 删除某个插入的半平面

3. 查询覆盖了某个点的半平面中，权值最大的半平面的权值

这个问题如果不用整体二分进行处理的话，几乎无法解决。但整体二分之后，问题就成功转化成了文章开头的动态半平面交问题，从而得到了解决。

3 总结

对时间分治算法、二进制分组算法和整体二分算法，都是解决数据结构题的有效工具。适当地应用这些算法、充分地发挥这些算法的威力，可以大幅降低数据结构题的难度、降低编码难度。这些算法的本质，实际都是通过利用数据结构题的操作中隐藏的特殊性质，例如“修改独立”“贡献独立”“允许离线算法”等，来做到转化问题、简化问题的。这启示我们，要充分挖掘题目特殊

性、利用题目特殊性，从而得到更加优美高效的算法。

4 特别感谢

- 感谢CCF提供了这个交流的平台
- 感谢父母、学校对我的培养
- 感谢贾志鹏、罗雨屏、彭天翼、胡渊鸣、王悦同等诸多通过网络或现实与我交流讨论、给予了我诸多帮助和鼓励的同学
- 感谢罗雨屏、龙浩民等同学在LaTeX方面给我的帮助
- 感谢我的班主任和其他老师对我的支持

参考文献

[1] 《从〈Cash〉谈一类分治算法的应用》陈丹琦

[2] 顾昱洲在WC2013中的授课

重量平衡树和后缀平衡树在信息学奥赛中的应用

杭州外国语学校 陈立杰

摘 要

重量平衡树是一种具有研究和实际价值的平衡树概念，后缀平衡树是一种基于重量平衡树的非常有价值的后缀数据结构，然而在近几年的OI活动中很少看到他们的身影，本文作者为了改善这一状况，从多个角度详细介绍了重量平衡树的概念和各个种类，并且选取了一些经典的应用加以深入探讨。

1 本文结构

本文先简单的介绍重量平衡树的概念，并且将介绍一些重量平衡树。

[1] Treap

[2] Skip-List(跳表)

[3] Scapegoat Tree(替罪羊树)

之后本文将介绍几个重量平衡树的应用

[1] 动态区间k大询问

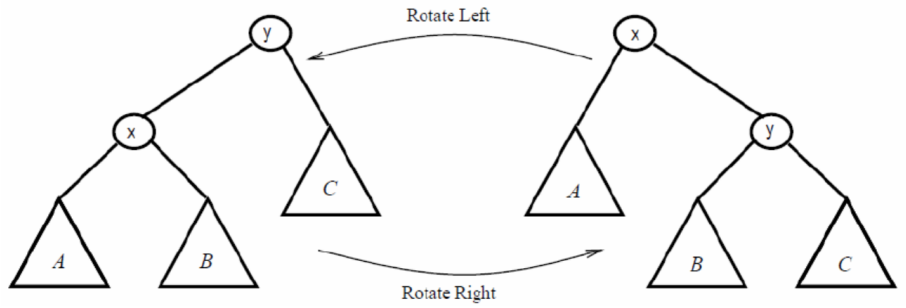
[2] 序列顺序维护问题

接下来本文将介绍可持久化重量平衡树。

最后是本文的重点，介绍使用重量平衡树来维护后缀平衡树这一数据结构，并解决一些问题。

2 重量平衡树的概念

一般的平衡树依赖于一个旋转操作，旋转操作在统计一些可以快速合并的信息的时候是没有问题的，但是对于更加复杂的信息，就会遇到复杂度的问题。



2.1 旋转机制的弊端

考虑一个简单的例子，我们想在一个平衡树的每个节点上维护一个集合，存储子树内部所有的数，此时一次旋转操作的代价可能会达到$O(n)$，传统的旋转平衡树就无法发挥作用了。

而重量平衡树则不同，他每次操作影响的最大子树的大小是最坏或者均摊或者期望的$O(\log n)$。这样即使按照上面所说的方式维护，复杂度也是可以接受的。

2.2 不采用旋转机制的几种重量平衡树

有一些平衡树(或者能实现传统平衡树操作的数据结构)并不依赖于旋转机制，因此也就不会受到旋转机制弊端的影响。

本文选取在OI中有一定实际价值的两者加以介绍。

2.2.1 跳表

关于跳表，已经有国家集训队的前辈在论文中做了详细的讲述，因此本文也不再赘述，有兴趣可以参考附录中的”线段跳表——跳表的一个拓展”。

2.2.2 替罪羊树

替罪羊树依赖于一种暴力重构的操作。

我们定义一个平衡因子a，对替罪羊树的每个节点t，我们都需要满足$\max(size_l, size_r) \leq a \cdot size_t$($l, r$分别表示左右子树)。

他的实现方式也非常的简洁明了，每次我们进行操作之后，找出往上最高的不满足平衡性质的节点，将它暴力重构成完全二叉树。

2.2.3 替罪羊树的复杂度

让我们考虑一个已经被重建过了的子树t，经过计算我们可以发现至少要在里面插入$\Omega(|t|)$ 个节点，他才会被重构（整体的重构，子孙的重构不考虑）。

同时重构的代价是$|t|$的，不妨每次插入一个点的同时，顺便给它的所有祖先都添上1的势能，那么当每个节点被重构的时候，它上面就会有足够的势能来进行重构。

那么插入的均摊复杂度就是$O(\log n)$了。

删除的分析于此基本一样，就留给读者思考了。

同时由于平衡性质，我们可以证明替罪羊树的高度是$\log_{\frac{1}{a}} n$的。那么查找操作自然也是$O(\log n)$ 的了。

2.2.4 平衡因子a的选取

我们可以注意到平衡因子a在略大的情况下，重构操作会变少，因此插入的时间会有所降低，但是树高也因此变大，查找时间会增大。

在小的情况下，重构操作会增多，因此插入的时间会增大，但是树高因此变小，查找时间也变少了。

一般情况下取0.6到0.7左右的平衡因子就能满足大部分的需求了。

2.2.5 替罪羊树的优缺点

替罪羊树虽然不基于旋转机制，但是其思路非常清晰，代码量非常的小，在速度上也不慢，无论是时间复杂度，实现复杂度和思维复杂度都不输给Treap以及Splay这样的传统平衡树。在实际比赛中使用是很明智的选择。

当然替罪羊树由于其平衡机制的限制，并不能支持一些非常复杂的操作，比如Splay中常见的提取一个区间等等。

同时由于他是一个势能的均摊结构，也无法简单的进行可持久化。

并且他在实现中也需要记录$size$域来表示子树大小，当然这在OI中并不是什么大问题。

总体来说替罪羊树在简单的平衡树应用上非常出色。

2.3 采用旋转机制的重量平衡树

即使采用旋转机制，也有很多平衡树是重量平衡的。可以这么理解，大部分旋转都是所谓的"微调"，只在底下发生，所以子树不会太大，高处的旋转则较少发生。所以总体均摊或期望复杂度任然是$O(\log n)$。

2.3.1 Treap

Treap是一个非常常见的数据结构。

顾名思义，Treap=Tree+Heap，它对每个点维护一个随机权值，同时使得这棵树的形状就跟按随机权值顺序从小到大插入的普通BST一样。

我们都知道一个随机数据的情况下，普通BST的期望高度是$O(\log n)$的。那么Treap的期望高度也是$O(\log n)$的，并且跟输入无关。

Treap的一系列操作都是很经典的算法，再次就不再赘述了。

2.3.2 Treap的重量平衡

当我们插入一个数的时候，可能会造成一些旋转，不妨设插入点t转到了祖先k的位置，那么我们需要使用$size_k$的时间来重构，但这种情况发生的条件是点t的随机权值是k子树中最小的一个，那么概率就是$\frac{1}{size_k}$。那么期望的代价就是1。同时一个点最多有$O(\log n)$个祖先，那么总共一次插入的期望代价就是$O(\log n)$。

接下来我们考虑删除一个数x。我们不妨直接重构以该树为根的子树。由于一个点期望有$O(\log n)$个祖先，那么我们可以知道(祖先，孩子)关系总共会有$O(n \log n)$ 对，也就是说每个点的子树的大小的和期望是$O(n \log n)$。因此一个点的子树大小期望是$O(\log n)$。所以删除一个数的复杂度是期望$O(\log n)$。

3 重量平衡树的应用

那么我们来看重量平衡树的几个应用。以此来强调重量平衡树的作用。

3.1 序列顺序维护问题

3.1.1 问题描述

我们现在有一个节点序列，要支持如下两种操作：

[1] 在节点x的后面插入一个新节点y

[2] 询问节点a,b的前后关系。

3.1.2 解法1

我们可以使用简单的平衡树来维护这个序列，插入就直接插入，询问大小关系只要求出两个节点各自的$rank$就能简单实现了。

插入复杂度$O(\log n)$。

询问复杂度$O(\log n)$。

3.1.3 解法2

我们仍然使用平衡树来维护这个序列，不同的是我们对每个点维护一个标记tag_i，询问a,b前后关系时只需要比较tag_a,tag_b的大小就行了。

如何维护这个标记呢，我们不妨让每个点对应一个实数区间，让根对应实数区间$(0,1)$，对于对应实数区间(l,r)的节点i，它的tag_i是$\frac{l+r}{2}$，它的左子树的实数区间是(l,tag_i)，右子树是(tag_i,r)，容易发现这样的tag_i是满足要求的。

那么我们比较的时候只需要比较tag_i就可以了，同时注意到在这里每个点tag_i值的分母是2^{depth_i+1},$depth_i$表示i的深度也就是到根的距离。

那么只要树是平衡的，也就是最深深度不大，精度是有保证的。使用double就可以了。

那么在插入的时候，我们可能要对一整个子树的tag值进行重新计算，只要使用重量平衡树，就能做到$O(\log n)$的复杂度。

询问复杂度是$O(1)$。

3.1.4 解法3

本问题是一个经典问题，黄嘉泰同学查阅了论文并发现存在一个插入和询问都是$O(1)$的优秀算法，不过跟本文主题无关，有兴趣的同学参加附录中的“Two Simplied Algorithms for Maintaining Order in a List”。

3.2 动态区间第k大

3.2.1 题目描述

我们现在有一个长度为n的序列s，要求支持几个操作：

[1] 在第i个数的后面插入数x

[2] 修改s_i的值为v

[3] 删除第i个数

[4] 询问区间$[l,r]$内的第k大数是多少。

3.2.2 题目解法1

我们使用重量平衡树进行维护，每个节点用一颗平衡树保存其子树的中的所有数。

那么插入删除修改都可以在$O(\log^2 n)$的时间内解决。

对于询问操作我们先二分答案a，然后就可以在$O(\log^2 n)$的时间内回答区间中比a小的数有几个，总复杂度就是$O(\log^3 n)$。

3.2.3 题目解法2

我们可以在每个点维护一个权值线段树，从而将复杂度改进到$O(\log^2 n)$。

关于权值线段树，可以参考我去年的论文，附录中的”可持久化数据结构研究”，里面有详细的讲解。

3.2.4 题目解法3

我们另外提出一个有意思的解法，我们不妨对所有权值整体开一个线段树。在权值区间$[l,r)$保存所有权值在$[l,r)$之间的数的位置组成的平衡树。

回答询问的时候就在权值线段树上走一遍，复杂度是$O(\log^2 n)$。

插入数的时候，我们需要在序列中插入该数，同时线段树中包含该数的节点中插入它。

此时我们需要注意到，因为插入了一个数，它后面所有数的位置都+1了，我们显然无法直接修改它们的位置，但是注意到我们并不需要的它们的确切位置！在平衡树中维护只需要能快速比较两个位置前后顺序就可以了！

因此我们使用序列顺序维护问题的解法来维护这个序列，这样我们就能快速比较两个数的前后关系，从而完成平衡树的各种操作。

那么其它操作的复杂度也是$O(\log^2 n)$。

3.3 动态K-D树

K-D树是一个很经典的几何数据结构并支持很多操作。

首先简单回顾一下K-D树的概念。

K-D树依赖于对平面所有点的不断划分，我们每次将所有点按照X或Y轴排序，按照左右或者上下分成尽量均匀的两部分，然后递归划分。

划分的轴的顺序是交替的，首先X轴然后Y轴然后依次类推。

假设我们现在有了个K-D树，我们需要支持插入一个点的操作。

如果我们直接暴力查找点的位置然后插入，K-D树的性质就会被破坏。

我们可以仿照替罪羊树的思想，设定一个平衡因子a，令一个划分最多的部分不能超过a*总大小，这样就能保证插入和询问的复杂度了。

3.4 MAXPATH

这是一道我的原创题

3.4.1 题目描述

平面上有n个点，两个点p_i,p_j之间有有向边的条件是$i<j,|p_ip_j|<R_j$。

求一条最长的路径。

$n \leq 50000$。

3.4.2 解法1

此题有一个基于分治和V图的$O(n \log^3 n)$解法，常数和代码量都十分惊人。

由于跟本文主题无关，就不再赘述，感兴趣的同学可以去翻阅WC2013顾昱洲的讲课课件。

3.4.3 解法2

首先我们进行DP，令dp_i表示以i为结尾的最长链的长度是多少。

那么可以写出DP方程$dp_i = \max dp_j (j < i, |p_i p_j| < R_i)$。

考虑这个式子，第一个想法是维护一个数据结构，支持插入一个带权点和询问一个圆内部最大的点权。

但是这是非常困难的问题。我们不妨换一个思路，我们将之前所有的$j < i$按照dp_j值的顺序排序。

然后我们依次从大到小找出第一个与i的距离小于R_i的。

这样复杂度还是$O(n^2)$的，不过我们注意到此时问题就变得可以套用之前提到过的数据结构了。

我们使用重量平衡树来维护所有j按照dp_j的排序，在节点上维护一个K-D树保存节点内部所有的点。

那么回答询问时，我们在K-D树上询问其中离i最近的点的距离是否小于R_i，就能知道该往哪个孩子走了。

算出dp_i之后我们就直接插入平衡树中就可以了，由于也需要在K-D树中插入，我们还需要使用前面提到的动态K-D树。

K-D树询问最近点的代价可以看作是$O(\sqrt{n})$的，那么总共的复杂度就是$O(n^{1.5} \log n)$。虽然复杂度相较解法1变高了，但是常数和代码复杂度就变少了很多(V图的代码量和代码复杂度相比K-D树非常大)。

4 可持久化重量平衡树

我们来考虑可持久化重量平衡树。由于Treap本身就是重量平衡的，我们只需要实现一个可持久化的Treap就可以了。

关于可持久化Treap的总总细节，可以参考我去年的论文“可持久化数据结构研究”，在此就不再赘述了。

时间复杂度和空间复杂度都是每部期望$O(\log n)$。

替罪羊树由于它是一个均摊的数据结构，如果我们只需要部分可持久化(只能修改当前结构，对历史结构只进行询问)，是可以通过简单的可持久化平衡树方法实现的。

但是如果要实现完全可持久化(对历史结构也要支持修改)，则因为他是均摊$O(\log n)$的，如果一个高代价的操作进行多次，复杂度就被破坏了。

5 后缀平衡树

5.1 后缀平衡树的定义

考虑一个长度为n的字符串s，定义s^i为其由$s_i, s_{i+1}, \cdots, s_n$组成的后缀。

后缀之间的大小由字典序定义，后缀平衡树就是一个平衡树并维护这些后缀的顺序。

5.2 后缀平衡树的构造

接下来我们从在线或离线的角度分别考虑如何构造后缀平衡树。

5.2.1 离线算法

给定一个字符串s，我们先求出它的后缀数组也就是所有后缀的排序，然后根据后缀数组很容易就能构造后缀平衡树。

复杂度$O(n)$或$O(n \log n)$,取决于你使用的求后缀数组的算法。

5.2.2 在线算法

跟后缀树和后缀自动机相同，后缀平衡树也有一个一个个添加字符的在线算法，也因此能解决一些单靠后缀数组无法解决的问题。

跟后缀树和后缀自动机相反的是，后缀平衡树的在线算法不是在字符串最后，而是在字符串开头添加字符。

让我们考虑当前字符串s，并且在其开头加入一个字符c。

那么可以发现实际上就等于在后缀平衡树中插入了一个新的后缀cS。

如果我们能快速比较两个后缀的大小，那么直接套用平衡树的插入算法就行了。

5.2.3 哈希

首先我们可以使用最经典的哈希法，我们预处理一下字符串的哈希，然后就能通过二分在$O(\log n)$ 的时间内求出两个后缀的LCP，从而比较它们的大小。

此时时间复杂度是$O(n \log^2 n)$的，代码复杂度很小，比较好写。

5.2.4 套用序列顺序问题

注意到如果我们使用之前提到的序列顺序问题的解法，就能在$O(1)$的时间内比较两个后缀的大小了。

这样复杂度就变成了$O(n \log n)$。较哈希法有了很大的改进。

5.2.5 可持久化

我们考虑对后缀平衡树进行可持久化，只要我们使用Treap，可以很容易的将算法扩展成为可持久化算法，从而实现可持久化后缀平衡树。

此时时间复杂度不变，空间复杂度变为$O(n \log n)$。

5.3 后缀平衡树的优势

首先后缀平衡树在概念上只是后缀数组和平衡树的一个综合应用，相比复杂的后缀自动机和后缀树更加好理解。

代码复杂度则相对大一些，但是思路很清楚，不容易写错。

同时它相比后缀自动机和后缀树，还支持一些其它的操作。

比如后缀自动机和后缀树虽然都可以在末尾加入一个字符，但是它们无法支持在末尾删除一个字符的操作，相比之下后缀平衡树就可以实现这个操作。

同时后缀平衡树还可以支持动态维护一个Trie的所有后缀，这是后缀自动机不能高效实现的。

还有就是后缀平衡树完全不依赖于字符集的大小，而后缀自动机和后缀树的简单实现都需要考虑这点。

最后，后缀平衡树可以支持可持久化，这是后缀自动机和后缀数望尘莫及的。

接下来我们就通过几个例题来表明后缀平衡树的用处。

6 后缀平衡树的应用

6.1 COT4 online

此题改编自黄嘉泰大神的经典题目COT4。(相对于原题这可以说是一个弱化)。

6.1.1 题目大意

给你一个字符串集合S，一开始里面只有一个空串。你需要支持一些操作：

[1] 取出一个$s \in S$，考虑一个字符c将sc和s都放入集合S。

[2] 对于一个询问串q和一个集合中的字符串$s \in S$，输出q作为s的子串出现了几次。

操作次数$n \leq 200000$,询问串长度和$Q \leq 500000$。

6.1.2 题目解法

其实我们注意到，将sc放入集合S，实际上就是在s后面添加了一个字符，那么我们使用可持久化后缀平衡树维护s的所有后缀，添加时进行可持久化插入即可。

然后是询问一个字符串q出现了几次，注意到这个实际上就是所有字典序比$q\#$小的后缀的数量减去比q小的后缀的数量，$\#$代表一个虚拟的最大字符。

那么我们在平衡树上二分比较查找即可，复杂度是$O(|q|\log n)$，可以接受。

通过维护一些附加信息可以将询问复杂度改进到$O(|q|+\log n)$，不过较为复杂。

操作复杂度是$O(\log n)$，总体复杂度就是$O((n+Q)\log n)$。

6.2 STRQUERY

此题是我的原创题。

6.2.1 题目大意

我们有一个字符串s，我们需要支持如下4个操作:

[1] 在最左端插入或删除一个字符

[2] 在最右端插入或删除一个字符

[3] 在正中间($\frac{|s|}{2}$处)插入或删除一个字符。

[4] 询问字符串q出现了几次。

一共有$n(n \leq 150000)$个操作，所有询问的长度和≤ 1500000。

6.2.2 题目解法

首先，我们的后缀平衡树可以支持在最左端插入或删除一个字符，令这个为结构TL。

那么我们将这个整个反过来，并且将询问串也反过来，就能支持在最右端插入或删除一个字符，令这个为结构TR。

我们现在考虑来实现一个可以在两端删除或插入的结构，我们不妨维护一个结构TL，一个结构TR，并让结构$TB = TL + TR$，那么插入删除分别在TL和TR做，如果其中一个被删到了0个，那么我们可以把另外一个均匀的分成两半，从均摊意义上这个操作不影响复杂度。

同时在TB上询问，我们可以先询问q在TL和TR中出现了几次，然后再询问它在跨越的部分出现了几次，注意到跨越部分长度最多为$2|q| - 2$，直接取出进行KMP即可。

然后我们来实现一个可以在正中间插入的结构TM，我们维护两个TB : l, r，每次插入之后在l和r之间交换几个字符，使得l后面那个位置始终是正中间。这样就能在正中间插入了。

询问跟在TB上询问一样。

那么我们就得出了一个支持此题的结构TM，复杂度是$O(n \log n)$。

感谢

黄嘉泰同学对我的帮助。
CCF为我们提供的这次宝贵的机会。
以及教练们的指点。

参考文献

[1] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[2] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

[3] Michael A. Bender, Richard Cole, Erik D. Demaine,Martin Farach-Colton, and Jack Zito, “Two Simplied Algorithms for Maintaining Order in a List”.

[4] 李骥扬,《线段跳表——跳表的一个拓展》, 国家集训队2009论文。

[5] 陈立杰,《可持久化数据结构研究》, 国家集训队2012作业。

浅谈环状计数问题

江苏省常州高级中学 高胜寒

摘 要

在数学、信息学中，常会遇到计数问题，其中很多还是环状计数问题。解决此类问题常用到burnside与pólya定理，同时需要配合其它算法，如递推等。本文对此进行了简要的分析，并总结出一些经验予以参考。

引言

在信息学的环状计数问题中，常常会有“**旋转**或**翻折**后相同则算同种情况”的条件出现，如：

一个环状n个元素的数列，每个元素都可以染成颜色$1, 2, \cdots, m$，旋转后相同的数列算作同一种方案，求总方案数。

对于这样的计数，常常使用组合数学或者递推解决，而对于的旋转及翻折这些问题，需要推理、优化。如遇到此类问题，在数学推理方面就可以使用本文所提的burnside引理及pólya定理，下面先回顾一下解决此类问题所需要的知识。

1 理论基础

1.1 运算

非空集合S的**运算**(如加法、减法、集合交并等)是从S×S到S的函数$*$[38]，写成a$*$b 或ab，集合S和S的运算记作(S.$*$)。

[38]S×S到S的运算有时称为二元运算，本文所述运算均为此运算。

1. 运算若对$\forall\ a,b\in A$有$a*b\in A$，则称A在$*$下具有**封闭性**。
2. 运算若对$\forall\ a,b,c\in A$有$((a*b)*c)=(a*(b*c))$，则称A在$*$下满足**结合律**。
3. 运算若对$\forall\ a,b\in A$有$a*b=b*a$，则称A在$*$下满足**交换律**。
4. e为S中元素，若对$\forall\ a\in S$有$a*e=e*a=a$则称e为$*$的**单位元**。
5. a,b为S中元素，若$a*b=b*a=e$则称b是在$*$下a的**逆元**(写为a^{-1})。
6. 运算若对$\forall\ a,b,c\in A$当$a*b=a*c$时有$b=c$，则称A在$*$下满足**消去律**

1.2 群

设G是一个具有二元运算的非空集合，若满足

1. 运算满足封闭性
2. 运算满足结合律
3. 单位元属于集合
4. 集合中任一元素的逆元属于集合

则可称G为群。详见1.1。

群G中元素个数记作$|G|$，称为G的**阶**，由此分为**有限群**与**无限群**。

1.2.1 置换

设σ是一个从集合$1,2,\cdots,n$到自身的一一映射，则可称σ为**置换**，形如：

$$\sigma=\begin{pmatrix}1&2&3&\cdots&n\\a_1&a_2&a_3&\cdots&a_n\end{pmatrix}\tag{10}$$

其中a是一个排列，而其中任意一项$\binom{i}{a_i}$都是独立的映射。

由所有置换所构成的集合记作S_n，显然阶为$n!$，不难发现，S_n是一个群，称之为n阶**对称群**。而**置换群**是对称群的子群[39]，其中所有元素都是一种置换。其中包括了单位元

$$\begin{pmatrix}1&2&3&\cdots&n\\1&2&3&\cdots&n\end{pmatrix}$$

任意一种置换有唯一逆元

$$\begin{pmatrix}1&2&3&\cdots&n\\a_1&a_2&a_3&\cdots&a_n\end{pmatrix}^{-1}=\begin{pmatrix}a_1&a_2&a_3&\cdots&a_n\\1&2&3&\cdots&n\end{pmatrix}$$

置换可以做乘法，如$\binom{a}{b}*\binom{b}{c}=\binom{a}{c}$。因为置换有唯一逆元，所以若a,b,c为任意置换，$a*b=a*c$，则$a^{-1}a*b=a^{-1}a*c$得$b=c$。从而此乘法满足消去律。

1.2.2 burnside引理

定义：

1. k不动置换类：给定置换群G中，对于n阶集合X，$k\in[1,n]$，使k不变(即第k项置换为$\binom{k}{k}$)的置换全体，写作Z_k(容易发现，Z_k是G的一个子群)。
2. 等价类：给定作用在集合X上的置换群G，若存在某置换g把元素k变为元素l，则称k与l属同一个等价类，k所属等价类记为E_k。

由此可见，在解决实际问题时，通常是给出了置换群，需要求出在给定集合X中等价类的数目，注意到k不动置换类较为易求得，可以将等价类数用其他可以求得的量所表示。

定理：$\forall k\in[1,n]$有$|E_k|*|Z_k|=|G|$。

这里简单说明一下：设$E_k=a_1,a_2,\cdots,a_n$，令t_j为任意一个G中可将k映射为a_j的置换，由于置换的消去律和封闭性，则t_j*Z_k可以既不重复亦不遗漏的表示所有第k位为a_j的排列。对j求和，即得到以上定理，同时可以发现若$i,j\in E_k$则$|Z_i|=|Z_j|$。

[39]设H是G的子集，若在G的运算下，H是一个群，则称H为G的**子群**

令l为集合X等价类数量，$\sum_{k=1}^{n}|Z_k| = \sum_{k=1}^{l}\sum_{i=1}^{|E_k|}Z_{a_i} = \sum_{k=1}^{l}|G| = l|G|$，即可得到**burnside引理**：设所有可能情况集合X，置换群G，对于每个置换$g \in G$，令X_g为g置换后位置不变的元素集合[40]，则等价类的数量

$$l = \frac{1}{|G|}\sum|X_g| \tag{11}$$

2 特殊情况分析

引言部分所说的**旋转**和**翻折**，正是burnside引理的特殊情况，在此具体讨论一下其推导与应用。

2.1 旋转

发现这是burnside引理的一种特殊情况：对于旋转操作，如果旋转一个单位，可得到置换

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & n-1 & n \\ 2 & 3 & \cdots & n & 1 \end{pmatrix}$$

将此置换自乘k-1次，也就相对应的旋转k格，将得到置换

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ 1+k & 2+k & 3+k & \cdots & n+k \end{pmatrix} \tag{12}$$

这里约定$n+1=1, n+2=2, \cdots, n+k=k$。

这样就形成了一个阶为n的置换群。

pólya定理：设G为对n个对象的置换群，每个对象可用m种颜色染，则不同方案数为：

$$l = \frac{1}{|G|} * (m^{c(g_1)} + m^{c(g_2)} + \cdots + m^{c(g_{|G|})}) \tag{13}$$

c(g_i)为g_i的循环节数。

pólya定理可以解决这些问题。

问题所要求的答案l为$\frac{1}{n} * \sum_{k=1}^{n} X_{\sigma^k}$，$\sigma^k$也就是旋转了k格，而$X_{\sigma^k}$即转过k格后置换为自己，由辗转相除可知其不动元素是循环节为(n,k)的数列。所以整

[40] $\sum_{g\in G} X_g = \sum_{k=1}^{n}|Z_k|$=总不动置换数

理下来为$l=\frac{1}{n}*(\sum_{i=1}^{n}f[(n,i)])$，其中f[i]为递推算出的答案，如pólya染色问题中$f[i]=m^i$，此公式极类似于pólya定理。

继续优化此式子，可以枚举(n,i)情况，n≡0(mod (n,i))，所以枚举n所有因子k，$(n,i)=k$要求

$$\begin{cases}\texttt{n}\equiv\texttt{0(mod k)}\\ \texttt{i}\equiv\texttt{0(mod k)}\\ (\frac{n}{k},\frac{i}{k})=1\end{cases}$$

观察条件三看起来，结合数论知识，可以发现满足要求的i正好就是有$\phi(\frac{n}{k})$个，所以最终的公式可以被写为：

$$l=\frac{1}{n}*\sum_{\texttt{n}\equiv\texttt{0(mod k)}}f[k]*\phi(\frac{n}{k}) \tag{14}$$

实际上，在不了解burnside和pólya时可以这样理解：对于一个长为n的环，计算时会在每一个位置上计算这个环一次，答案是$\frac{f[n]}{n}$，但可能在不同位置将环断开来所形成的数列相同，此仅发生在环存在循环节的情况下。若循环节长度为k，则断开后仅会有k种不同的数列，所以要将循环节长度为k的情况加上去。

枚举所有n的因子作为循环节长度k，循环节长为k则有$\frac{n}{k}$次循环。当成链式结构做完一次长为k的递推，也就是将k复制$\frac{n}{k}$遍使其长为n的种类数。假设长为k的数列最短循环节就是k(即不是更短的循环复制而成，后面将处理这一种情况)，那么其方案数为$\frac{f[k]}{k}$。然而显然某些部分多算了一些，也就是之前括号中所提，如：在计算循环节为6的串时，方案中一定会包含循环节为1,2,3的串，那么可以在算循环节为3的串答案容斥掉它，减去$\frac{f[3]}{6}$。需要使用容斥或者莫比乌斯函数计算，设$\frac{n}{k}=p_1^{c_1}p_2^{c_2}\cdots p_t^{c_t}$，可以推导出f[k]的系数为

$$\sum_{d_i<=c_i}(-1)^{c_1-d_1+c_2-d_2+\cdots+c_n-d_n}\frac{p_1^{d_1}p_2^{d_2}\cdots p_t^{d_t}}{n}$$

仔细观察发现其实上述公式所表达的就是$\frac{\phi(\frac{n}{k})}{n}$，最终得到的答案正好是之前推出的公式。

事实上此方法本质上和pólya定理推导相同。

2.2 翻折

同理可以写出翻转操作的置换，即

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ n & n-1 & n-2 & \cdots & 1 \end{pmatrix} \tag{15}$$

其逆元即为单位元(一个数列翻转数字不变)。

同样，可以理解翻转时的特殊情况：对于长n的链，通常来说每个串都被计算了恰好两次。当原串为回文串时正反相同事实上只计算了一次，加上回文串的情况即可，其实就是把长为

$$\begin{cases} \frac{n}{2}, & \texttt{n≡0(mod 2)} \\ \frac{n+1}{2}, & \texttt{n≡1(mod 2)} \end{cases}$$

的短链翻折至另一边变为长为n的数列。由于n为奇数时在置换的第$\frac{n+1}{2}$项中自己置换自己，所以奇偶略有些差异。

所以可以写出公式：

$$l = \frac{1}{2} * (f[n] + f[\frac{n + odd(n)}{2}]) \tag{16}$$

这里暂定odd(n)为判断奇偶的函数，奇数为1，偶数为0。

2.3 翻折+旋转

此情况将会在以下的具体问题中分析。

3 实例

3.1 Codeforces93D Flags

题意描述

给定一串长为n的数列，用四种颜色（白黄红黑）染。要求相邻位置颜色不同，白黄、红黑不能相邻，且不存在连续黑、红、白所形成的三元组。翻转后相同的数列算为同一种，询问长度在l到r之间的数列有多少种，取模输出。$1 <= l <= r <= 10^9$

算法分析

首先可以把此题转化为求长度在1到n之间数列的种类数。此问题可以用递推解决，因为要解决不存在连续黑、红、白三元组，所以至少要记录最近两项的颜色，从而可以完成转移，递推可以使用矩阵乘法进行优化。根据2.2的总结，需要考虑回文串情况。可以发现，奇数与偶数情况有所差异。

本题较为特殊，显然可以发现当n为偶数时最中间两项颜色相同，为不符合的情况。所以偶数时答案为$\frac{f[n]}{2}$，奇数时答案为$\frac{f[n]+f[\frac{n+1}{2}]}{2}$，最终全部转化为长度在1到n之间不翻转数列的种类数。

3.2 Spoj large

题意描述

t组数据：一张圆桌坐着n个人，人与人之间只考虑性别上的差异，要求不能有超过m个女生连续坐着。圆桌可以旋转，询问排座位方案种类数，取模输出。$1 <= n, m, t <= 1000$

算法分析

考虑此旋转模型，其置换群阶为n，通俗地描述下来就是分别为转$1, 2, 3, \cdots, n$格。先考虑不加旋转操作的情况，暂且忽视圆桌首尾相接，可以记录状态g[i]表示长为i的序列且i为男生方案数，易发现可以部分和优化转移。

下面加上环的性质，首尾相接后不能超过m个女生相连。根据已经算好的g[i]，这里可以采用容斥法，加以递推可得到最终数列f[i]，表示在环大小为i、不能有m个女生连续坐的方案数。接下来可以考虑旋转在其中的作用了，利用2.1推导的公式，枚举n的因子k 由预处理的数组f可统计出答案。

3.3 加强-自创题ring

题意描述

简化题意：求长为n的环状01串，其包含给定长为k的字符串s的数量，取模输出。$1 <= k <= n <= 10^9, k <= 30$

算法分析

考虑此题和上题的区别，给定的串不再有原来那么有规律，可能在解决这个问题时遇到一些麻烦。

可以模仿其做法，但会发现求f[i]时不得不记录第二维状态，这可以利用kmp失配指针可以写出转移。然而加上环的性质后，问题变得更加复杂，如果再采用容斥可能会给计算带来很大麻烦，所以需另辟蹊径。对于一个解，考虑在环拆开后，如果不存在字符串s则必然会有：数列首是s某后缀，数列末是s某前缀。考虑上题，发现只需数列后、前相拼接时女生个数等于k+1即可，并且对于任意这样的的情况，中间n-k-1个座位的可安排方案数均相同。

考虑本题，并不满足以上性质，但可以k^2预处理出一个二维bool数组p记录s的每个后缀与前缀拼接起来是否形成s，对于所有可能的前后缀拼接情况都要做一次递推。由于可能出现某后缀是另一个后缀的前缀这种情况，导致重复计算所以需去重。状态可以记为后缀长度恰为a，前缀长度恰为b。用已有的p数组可以统计相接后形成s的方案数，可以用矩阵乘法快速幂实现。

至此，本题已经基本解决了。但是仍需加上旋转公式以算出答案。考虑旋转的特征，可以注意到一个重要的细节，即当因子长度$l < k$时的处理，考虑到之前所说$f[k]$在公式(5)中的意义，在计算时所得f[l]的答案是将l重复写$\frac{n}{l}$遍的方案数，考虑到$k <= n$，所以可以认为是把长为l的字符串重写了足够多遍。然而字符串s可以被包括在里面，这就是一个经典的字符串循环节问题。利用已经算好的kmp失配指针，可以在几个if内解决此细节。

3.4 继续加强

加强方向

1. 加上翻转操作。

2. 改为多串问题：给定多个串，长为n的环不能包含任意一个字符串。

解决方案

1. 题目如果形成了翻转和旋转的结合，就会出现一些复杂的情况。其实最好的办法还是写出所有可能的置换，在2.1中的n 个置换及其公式。在理解翻转时，可以理解为翻转后再旋转共有n种置换，这样在具体操作起来较为复杂，不如着眼于环，把翻转后在旋转看作是一次翻转，但由于奇偶性的问题产生了两种情况，如：

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 5 & 4 & 3 \end{pmatrix}$$

可以看作以4为中心的翻转，在这样奇数情况下n种置换正是分别以n个元素为中心的翻转。若要是能形成不动置换，则环要关于中心元素对称，n种置换其$\frac{n+1}{2}$个元素均决定了另外的元素，方案数为$n * f[\frac{n+1}{2}]$，再加上旋转部分最终除以$|G|$即2n即可。

但偶数时如：

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 4 \end{pmatrix}$$

出现两种情况，其一为以两元素间为轴翻转，n个两元素间轴中两两同等，如上面例子中既可以以12间为轴，又可从34间翻转，所以有$\frac{n}{2}$种翻转。仅需其中一半，$\frac{n}{2}$个元素即可决定另外的元素。再者，可以以元素为轴，同元素间轴之理可发现有$\frac{n}{2}$种翻转，其中每种置换都可由两个位置翻转，如例子中可以以2或4为轴，所以要由$\frac{n}{2}+1$个元素(因为包含了两个轴)才可决定整个串。方案数是$\frac{n}{2}*(f[\frac{n}{2}]+f[\frac{n}{2}+1])$，加上旋转除以2n得到最终答案。

至于递推部分和加强前相差不大，状态不做修改，依然记录a与b表示后缀与前缀。旋转和翻折操作分开处理，由于翻折操作改变了旋转的拼接方式，所以需改变拼接规则，修改二维bool数组p的值即可完成此题。

至于原题所说的$l < k$的细节问题，翻折操作也是同样处理。由于此部分的效率与总复杂度无关，所以可以使用任意方法判断是否存在长为l的串翻折后包含k，实现起来只是略比旋转困难一些。

此题仍然能很轻松的解决。

2. 单串到多串，很显然递推与前后缀部分只需把kmp算法改为ac自动机即可。因为可能出现因子长度l小于某些串长度k_i，所以可以发现对于每个因子长度l都要重新构建自动机。

   这里提供一种简单的解决方案：对于每个字符串，类似于原题，如果存在一个长为l的串覆盖k_i的长度则将此长为k_i的串砍为长度l，表示不能出现这一个字符串。

4 总结

在竞赛中可能会遇到很多有类似条件的题目，很可能会让人感觉难以入手，甚至可能会有些怪异而难于解决，但是只要把握好置换的本质，及公式背后的每个数字的深层意义，许多看似很难的题目在思考的力量中可能就迎刃而解了。

参考文献

[1] 杨斌斌,《两类环状六角系统的计数》,厦门大学硕士学论文

[2] 董金辉,《正十二面体的旋转群诱导出的置换群的轮换指标》,黄冈师范学院数学信息科学学院学报

[3] Seymour Lipschutz,Marc Lars Lipson 著,曹爱文,曹坤译《Schaum's Outline of Discrete Mathematics》(离散数学)清华大学出版社

[4] 符文杰,《pólya原理及其应用》,2001集训队论文

[5] 陈瑜希,《pólya计数法的应用》,2008集训队论文

分块方法的应用

山东省胜利第一中学 王子昱

摘 要

分块方法是一种通用的、高性价比的问题解决方法，能够在不对问题性质做过多分析的情况下给出效率较高的解法。本文通过若干例题，介绍了分块方法的各种应用，并尝试对其共性进行归纳。

引言

在信息学中有一类问题，要求支持序列（或树等结构）上区间中信息的快速统计，并可能带有各类修改操作。这一类问题的传统解法可以分为利用线段树和对序列分块两类，而线段树的使用常常有局限性，此时可行的做法就只有分块，例如第二节中的问题。另一方面，有的问题虽然可以用线段树或其它方法解决，但分块方法更为简单，有时甚至能带来更好的时间复杂度，例如第一节中的两个问题。

对序列分块的方法体现了均摊制约效率的因素的思想。这一的思想不仅可以用在对数据的维护上。定期重建的方法[41]，以及合并两类算法解决问题的方法都体现了这样的思想。这几类问题在后两节中进行了讨论。

[41]可以看做对时间线进行分块

1 Section 1[42]

1.1 Problem 1 : UNTITLE1[43]

题意

给定一个序列$A[1..n](n \leq 50000)$，设计一个数据结构，支持两种操作：

1. 0 x y k : 将$A[x..y]$增加k。
2. 1 x y : 查询$S[x..y]$的最大值，其中$S[i] = \sum_{j=1}^{i} A[j]$。

题解

本题是一类数据结构问题的代表。它要求维护一个序列，支持对区间的修改和询问。对于这类问题，一般的思路是将序列分为若干区间进行维护。如果区间的信息支持合并，我们可以使用线段树；否则，我们只能使用分块的方法。

本题中，我们尝试维护序列$S[x]$。考虑一次操作$(0, x, y, k)$，假设它涉及的区间为$[l_i, r_i]$，则问题相当于将$[l_1, r_1]$中的$S[i]$增加$(i - l_1) \times k$，将$[l_2, r_2]$中的$S[i]$增加$(l_2 - l_1) \times k + (i - l_2) \times k$，……，将$[l_j, r_j]$中的$S[i]$增加$(l_j - l_1) \times k + (i - l_j) \times k$。注意对于同一个区间$[l, r]$中的$S[i]$，其增量具有$delt + (i - l) \times K$的形式，其中C和delt是常数。所以对每个区间我们维护这两个值作为懒标记。

该区间进行的查询相当于查询$C + max(S[i] + i \times K)$的值。为了快速进行查询，我们考虑可能成为最优解的点的性质。设$y_i = S[i], x_i = i$，忽略对所有点相同的常数C，所求值$z = y_i + k \cdot x_i$可以看做过点(x_i, y_i)，斜率为$-k$的直线在y轴上的截距。查询最大值的操作，相当于寻找一个点集中，过某个点的斜率为$-k$的直线的y截距的最大值。考虑一条斜率为$-k$的直线从y轴无穷远位置向下平移，与点集中的第一个点接触时即停止，如图1所示，此时的y截距显然是最大的。该直线只与点集的上凸壳中的点接触，因此，对于任意的z，只有(x_i, y_i)的上凸壳中的点可能构成答案[44]，区间中需要维护的信息就是其凸壳。求出凸壳

[42]本文中每节介绍的问题在做法上存在共性。显式地用节标题对其进行概括是不必要的，因此节标题采用了"Section n"的简单形式

[43]Source : SPOJ

[44]严格的证明略为繁琐，大家可以自行完成，这里略去

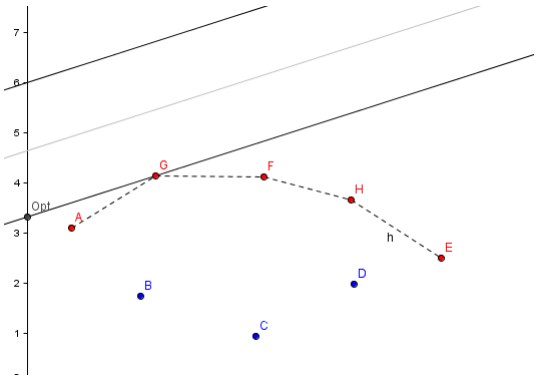


Figure 1: 某一区间对应的点集如图，考虑k=-0.32的情况，最优解对应的点为点G，可能成为最优解的点用红色标出，它们构成了点集的上凸壳

之后，注意到对于凸壳上的点(x,y)，它构成的答案关于x 呈双调。因此，可以使用三分法[45] 进行查询，复杂度为$O(logN)$。

如果使用线段树维护，区间信息的合并无法快速完成[46]。所以我们分块维护。将原序列分为M块，每块中维护$S[]$对应点集的凸壳。修改操作对应于修改至多两个块的一部分，以及修改至多$\frac{N}{M}$块的懒标记。对于修改块的一部分的操作，我们暴力重建该块的凸壳，复杂度为$O(M)$；修改懒标记的操作复杂度为$O(1)\times\frac{N}{M}=O(\frac{N}{M})$，因此，修改的复杂度为$O(M+\frac{N}{M})$。同样地，查询操作对应于查询至多两个块的一部分，以及至多$\frac{N}{M}$块的整体查询。对查询块的一部分的操作，我们暴力完成；整体查询如前所述。总复杂度为$O(M+\frac{NlogN}{M})$。取$M=\sqrt{N}$，得到算法的修改操作复杂度为$O(\sqrt{N})$，查询操作复杂度为$O(\sqrt{N}logN)$。

通过将树划分为$\sqrt{N}$块，这类问题很容易推广到树上，形成的结构称为块状树。实现的细节这里略去。

1.2 Problem 2：静态RMQ问题

题意

给定一个序列$A[1..N]$，每次询问给定l,r，查询$min_{l\le i\le r}(A[i])$。

[45]参见参考文献[4]

[46]凸壳的合并有$logN$的算法，但是过于复杂，可以参考参考文献[1]

题解

$O(N)-O(\sqrt{N})$**做法** 将序列分为$\sqrt{N}$块，将每块中第一个位置称为关键点。预处理出每块内的最小值，并利用这一信息，在$O(N)$时间内预处理出每对关键点之间的最小值；同时预处理出每个位置到离它最近的两个关键点之间的最小值。于是如果询问(l,r)中，l和r不在同一块内，我们可以利用上述信息在$O(1)$时间内解决；否则，我们暴力扫描$A[l..r]$，在$O(\sqrt{N})$时间内得到答案。

$O(NloglogN)-O(1)$**做法** 对于l,r在同一块内的情况，我们没有必要暴力扫描。我们可以对每一块递归地建立上文所述的数据结构，直到待维护的序列长度为1。可以注意到，这一结构相当于一棵每层度数不同的树，其深度为$O(loglogN)$。由于每层上预处理的代价是$O(N)$的，算法的预处理时间共为$O(NloglogN)$。

询问的时候，如果两个端点不在同一块中，我们可以$O(1)$得到答案；否则，我们递归到这一块对应的下一级数据结构中。这个做法是$O(loglogN)$的。但是如果我们每次二分一个最浅的、两端点不在同一块中的层次，询问的复杂度可以降到$O(logloglogN)$ 。

考虑如何得到询问$O(1)$的做法。如果存在整数z使得$N=2^{2^z}$，那么每一层上的块的长度是相同的。考虑长度为L的一个询问，设i 是第一个满足第i 层块长不超过L的整数，于是最浅的、两端点不在同一块中的层次必然是i或者$i-1$，只要分别验证即可。通过预处理每个L对应的i，询问的时间复杂度是$O(1)$ 的。

如果$z>loglogN$，我们可以把每一块的长度L增加到使$\sqrt{L}$是整数的最小值。此时每层块长相同的性质得到保持，上述做法仍然适用。显然，预处理的时间仍为$O(NloglogN)$。

$O(Nlog^*N)-O(1)$**做法** [47] 将序列分为$\frac{N}{logN}$块，预处理每个位置到离他最近的两个关键点之间的最小值。预处理每块内的最小值，并用这$\frac{N}{logN}$个值在$O(N)$时间内建立sparse table。于是对于跨块的询问我们就可以在$O(1)$时间内回答。为了处理块内的询问，我们对每块递归建立这一数据结构，直到当前处理的序列的长度为1。由于每次递归问题的规模降到了对数级，递归的深度是log^*N的；又每层预处理的总代价为$O(N)$，预处理的时间是$O(Nlog^*N)$的。

[47]感谢黄嘉泰告诉我这一算法

询问时用与上一做法类似的手段可以做到$O(1)$。

竞赛中，上述第二个做法的$O(NloglogN) - O(loglogN)$实现最有性价比。在竞赛中，可以利用计算机支持$O(1)$位运算的性质将这个做法优化到$O(1)$，这里不再赘述。对RMQ 问题还存在其他快速做法，这里不再讨论。

另外，上述列举的做法中，第一个和第二个可以扩展到一类一般化的问题：在无修改的情况下，询问区间信息的并，其中并的操作满足区间加法，但不满足区间减法。例如，区间最大连续子段和问题[48]。

2 Section 2

2.1 Problem 3 Fairy[49]

题意

给定一个图，询问存在多少边，使得删掉这条边之后原图为二分图。

顶点数$N \leq 100000$，边数$M \leq 100000$。

题解

本题存在线性做法，但思维和实现的复杂度都比较高。这里我们介绍一种简单暴力的做法。

如果要验证一个图是否为二分图，我们可以用并查集完成：维护每个点到其联通分量中代表元距离的奇偶性，加入边(u, v)时，在并查集中合并点u, v，如果u, v本来在同一联通块中，并且到代表元距离的奇偶性相同，则加入该边会引入奇环，原图不是二分图。

利用这一性质，我们可以将边表分块，对于每一块，首先计算出将此块之外的所有边加入并查集得到的结构；之后对于块中的每一条边，我们将块中除它以外的所有边加入并查集，并判断是否产生矛盾，之后撤销加入本块边的操作。撤销操作可以用栈维护。

注意并查集的$O(\alpha(N))$的复杂度是基于平摊分析得到的，因此此时算法的复杂度不能看做$O(N\sqrt{N}\alpha(N))$，而是$O(N\sqrt{N}logN)$。但是如果我们在将除去

[48]SPOJ GSS1

[49]Source : Codeforces 19E

该块之外的所有边插入之后进行缩点，前后两步的操作就是不相关的了，利用平摊分析可得，此时算法的复杂度为$O(N\sqrt{N}\alpha(N))$。

3 Section 3

3.1 Problem 4 : 糖果公园[50]

题意

给定一棵N个点的树，每个点带权。共有M种不同的权值。Q组操作，每次可以修改一个点的权值，或是询问连接点s_i, t_i 的路径的权值。

其中路径的权定义如下：给定数组V，W，设路径上第i种权值W_i的出现次数为N_w，则该路径的权值为

$$\sum_i \sum_{j=1}^{N_w} W_i V_j$$

数据分为三类：[51]

1. 树是一条链，无修改。
2. 树不是链，无修改。
3. 树不是链，带修改。

本题的标准做法是离线处理询问，并在询问中进行转移，详情参见集训队作业中的冬令营题解。在这里，我们讨论与其不同的一系列在线算法。

3.1.1 链状无修改的情况

对于树为链状，且无修改的情况，我们有以下做法：

将链分为$\sqrt{N}$块，预处理从每块的起点（以下称关键点）出发，到每一个点的答案。这一过程是$O(N\sqrt{N})$的。

考虑一次询问，设其左端点为l，右端点为r，设离l最近的块起点为L，于是$[L, r]$的答案已被求出。我们枚举$[l, L)$中的每一个点，把它纳入统计，并更新

[50]Source : WC2013

[51]去除了不具代表性的一些情况

答案。为了更新答案，我们需要知道某个权值$W[i]$在$(i,r]$中出现的次数，利用前缀和的方法，转化为查询某个权值在$[1,i]$和$[1,r]$中的出现次数的差。因此，我们维护$Okr[i][w]$，表示前i个数中离散化后的权值w出现的次数。

直接用二维数组进行维护的复杂度是$O(N^2)$的，但是注意到$Okr[i]$和$Okr[i-1]$仅在一个位置有差别，因此我们用持久化块状链表维护$Okr[i]$。将$Okr[i]$分为$\sqrt{N}$块，维护指向每块的指针。考虑$Okr[i]$的计算，其中有$\sqrt{N}-1$块和$Okr[i-1]$中相同，我们将指向$Okr[i-1]$中对应块的指针直接复制过来；对于剩下的一块暴力新建。因此构建$Okr[i]$只使用了$O(\sqrt{N})$的额外内存。定位任意一个$Okr[i][j]$的复杂度是$O(1)$的。

于是，预处理的复杂度为$O(N\sqrt{N})$。查询时需要额外考虑至多$\sqrt{N}$个点，其复杂度为$O(\sqrt{N})$。

算法的伪代码如下：

Algorithm 2 Init

```
1: bsize = √n
2: bcnt = [n / bsize]
3: for i = 1 to n step bsize do
4:     for j = i to n do
5:         calculate Ans[i][j]
6:     end for
7: end for
8: for j = 1 to bcnt do
9:     okr[0][j] = new int[bsize + 1]
10: end for
11: for i = 1 to n do
12:     p = [(W[i] − 1) / bsize] + 1
13:     copy okr[i] from okr[i − 1]
14:     okr[i][p] = new int[bsize + 1]
15:     copy okr[i][p] from okr[i][p − 1]
16:     okr[i][p][W[i] mod bsize + 1] + +
17: end for
```

Algorithm 3 Query

Require: the query interval $[l, r]$

Ensure: the answer ans

$L = \frac{l}{bsize} + 1$

if l mod bsize $\neq 0$ **then**

$L = L + 1$

end if

$ans = Ans[L][r]$

for $i = l$ to $L - 1$ **do**

$p = [\frac{W[i]-1}{bsize}] + 1$

q = w mod bsize

$o = okr[r][p][q] - okr[i][p][q]$

$ans = ans + V[o]$

end for

return ans

3.1.2 树状无修改的情况

对于树不是链的情况，做法是类似的。将树分为至多$\sqrt{N}$块，使得每块的高度不超过$\sqrt{N}$[52]。用与上一节同样的方法，预处理出每个关键点出发到任意一个点的答案。与链上类似，我们维护$Okr[i][w]$ 表示从根到点i 的路径上离散化后的权值w的出现次数，于是加入点更新答案的操作可以用类似链上的方法解决。算法的复杂度与链上的情况相同，为$O((N+Q)\sqrt{N})$。

3.1.3 树状带修改的情况

在有修改的情况下，我们考虑对操作分块解决问题。

将所有操作分为$Q^{1/3}$块，统一处理同一块中的操作。对于块中的询问，首先将此块之前的所有修改操作应用到树中，然后利用上一节中的做法，处理出仅考虑此块之前所有修改操作情况下，块中的每一个查询的答案（第一步）；之后对于块中的每一个查询，将同一块中在它之前的所有修改纳入统计中，计算它对答案带来的影响（第二步）。

[52]这样的分块显然总能完成

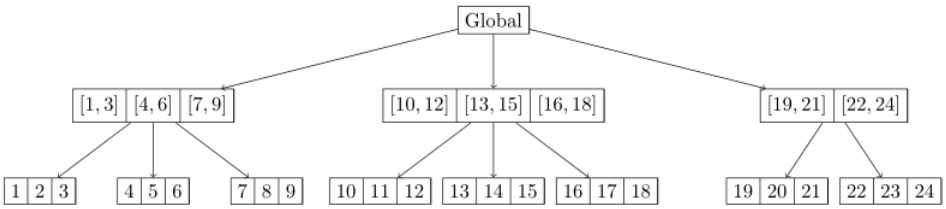


Figure 2: 有26个元素的三层块状表

考虑上面的第一步。我们将树分为$N^{1/3}$块，处理出从每个关键点出发，到达任意顶点的答案。在维护Okr数组的时候，如果我们继续利用持久化块状表，这一步的复杂度就会达到$O(N^{3/2})$，这是难以接受的。所以我们将块状表中的每个块再次分块。[53]将Okr 数组分为$N^{1/3}$组（第一层），每组中的$N^{2/3}$个元素用持久化块状表维护（第二层）。这一结构也可以被看做一个$N^{1/3}$叉树。每次我们只要修改一个（二层的）块状表，并新建一个大小为$N^{1/3}$的（底层）数组，因此，计算Okr数组的复杂度降到了$N^{4/3}$；而在Okr中定位一个元素的复杂度仍然可以看做$O(1)$，因此，处理一个询问的复杂度（在这一步）是$O(N^{2/3})$的。

在第二步中，我们考虑每个在当前询问之前的修改，判断它是否在当前询问的路径上。如果在，就修改答案。使用询问复杂度为$O(1)$的LCA算法，这一步的复杂度为每个询问$O(Q^{2/3})$。因此，算法的总复杂度为$O(Q^{1/3}(N^{4/3}+Q^{4/3}))$。

与标准做法相比，这一做法的优势是它不必离线。利用类似的方法，我们可以在线解决很多“无修改问题可以解决，带修改问题无法直接解决”的问题，或者只有一类操作无法直接解决的问题，例如带插入和修改的区间K值问题。

3.2 类似问题：带插入的区间K值问题[54]

问题描述

给定一个序列$A[1..n]$，支持三种操作：

1. M x y：将$A[x]$修改为y；

[53]对于Okr数组的维护，另外一个解决方案参见参考文献[3]

[54]Source：BZOJ

2. I x y：将y插入到$A[x]$后面；

3. Q l r k：查询$A[l..r]$中的第k小值

操作数q不超过1.75×10^6，原序列长度$n \leq 3.5 \times 10^4$。要求在线算法。

题解

如果只有修改和查询，我们可以用树状数组套线段树在$O((n+q)log^2n)$的时间内解决：树状数组中每个结点维护其对应的区间中所有元素组成的动态线段树；修改时在对应的$logn$棵线段树中修改；询问时二分答案，计算比当前答案小的元素个数，由于所有线段树的结构相同，询问的复杂度为$O(log^2n)$。

在有插入操作的情况下，将所有操作分为m块，将本块之前的所有插入操作应用到序列上，构造当前序列的树状数组套线段树结构。询问时计算考虑本块中插入的情况下，实际的询问区间；在每次二分时扫描所有本块中的插入，修正答案。算法的复杂度为$O(m(nlogn + \frac{q}{m}logn))$，取$m = \sqrt{q}$，得到复杂度为$O((q+n)\sqrt{m}logn)$。

4 Section 4

4.1 Problem 5[55]

题意

给定N个字符串集合，初始时每个集合中仅有一个字符串。要求支持操作：

1. MERGE A B 删除集合A、B，插入集合$A \cup B$。

2. QUERY S s 查询集合S中的串在字符串s中出现的次数。

字符串的总长$Slen$不超过10^5。

[55]Author：**黄嘉泰**

题解

利用二项队列思想维护AC自动机，可以得到$O(NlogN)$的做法。但是这种做法的思维和编程复杂度都比较高。这里介绍一种简单的$O((N+Qlen)\sqrt{Slen})$做法，其中Qlen是询问串的总长度。

设$L=[\sqrt{Slen}]$。将模式串分为长度大于L和小于L的两类，于是长度大于L的字符串至多有$[\frac{Slen}{L}]$个。对每个集合中长度小于L的串，预处理它们的hash 值，并按长度存储在Hash表中；对长度大于L的串，我们计算它们的在KMP算法中的$next$ 数组。

合并时只需要对Hash表进行启发式合并。询问时，对于长度大于L的串，暴力进行KMP，由于它们的$next$ 数组已被计算，这一步的复杂度是$O(qlen\times\frac{Slen}{L})$的，其中$qlen$是当前询问串的长度；对长度小于$L$ 的串，枚举匹配的起始位置，之后枚举模式的长度，在对应的Hash 表中进行查询，复杂度也是$O(qlen\times L)$的。

至此，问题得到解决。

4.2 Problem 6 KAN[56]

题意

如果两个区间有交集，我们称它们互相匹配。

给定N个区间$A[1..N]$。M组询问，每次询问给定一个区间$[l,r]$，确定A中的一个最长连续子序列，使得其中每一个区间都与区间$[l,r]$匹配。

$N\leq 50000, M\leq 200000$。

题解

区间$[l,r]$和$[l',r']$匹配$\Leftrightarrow l\leq r', r\geq l'$。因此，如果要判断一个询问和一个区间集合是否全部匹配，只需要计算该区间集合左端点的最大值（设为$first$）和右端点的最小值（设为$second$），然后与询问的端点进行比较即可。

如果一个询问的答案长度不超过L，我们可以对每个$l\leq L$，预处理出$S[l]$表示长度为l的所有连续区间的$(first,second)$有序对的集合。如果要验证该询问

[56]Source : Algorithmic Engagements 2011

（设为$[l,r]$）的答案是否不小于l，我们只要在$S[l]$中查找$first$不超过r的元素中，$second$的最大值，验证其是否不小于l即可。注意到对于元素a，如果存在b使得$b.first \leq a.first, b.second \geq a.second$，那么元素$a$就是无用的。在$S[l]$中去除所有这样的元素$a$后，按照$(first, second)$排序，每次验证就可以找到$first$不超过$r$的最大元素，验证其$second$是否不小于$l$。于是二分答案，再用上述方法验证即可。这一部分的复杂度是$O(NL + Qlog^2N)$的。

如果询问的答案长度超过L，那么将序列A分为长度不超过L的若干块之后，答案必然横跨至少两块。将序列分块，每块维护块内前i个区间和后i个区间的$(first, second)$对，于是对于每个询问，我们就可以二分得到其在每块中从左端点出发的最长匹配长度，以及从右端点出发的最长（逆序）匹配长度。利用这一信息，我们可以对所有块扫描一遍得到答案。复杂度为$O(Q\frac{N}{L}logN)$。

算法的总复杂度为$O(NL + Q(log^2N + \frac{N}{L}logN))$。取$L = \sqrt{NlogN}$，得到复杂度为$O((N+Q)\sqrt{NlogN} + Qlog^2N)$。

总结

分块方法的优势在于不需要对问题的性质进行过多的分析，具有通用性。其缺陷在于实现比较繁琐，且时间复杂度有时难以承受。在实际应用中，这一方法具有较高的性价比。

致谢

感谢在本文写作过程中提供帮助的罗雨屏、黄嘉泰、邢皓明三位同学。

感谢提供例题的许多同学。

参考文献

[1] 陈立杰，《可持久化数据结构研究》, 2012年集训队作业

[2] 王子昱，《糖果公园 解题报告》, 2013年集训队作业

[3] Anders Kaseorg, “Optimal worst-case fully persistent arrays”, TFP 2009

[4] “Ternary Search”, Wikipedia

浅谈容斥原理

成都七中 王迪

摘 要

本文从计数问题中的容斥原理出发，得出容斥原理的形式，通过一些例题探究了使用它解题的思维方式，然后研究了容斥原理的推广，对容斥原理的一般化作出尝试，最后总结了容斥原理及其运用过程中所体现的思想、方法。

1 引言

在一类组合计数的问题中，我们需要对一些集合的并或交中元素的个数进行统计，而对于这种问题，容斥原理是一种通用的解法，所以在本文的前半部分，我们将探究容斥原理的形式，用容斥原理解决计数问题的分析方法，以及若干有趣的例题。

而容斥原理不仅可以解决组合计数问题，在本文的后半部分，我们将对容斥原理进行推广，可以解决一些数论和概率论中的问题，而通过分析不同问题中容斥原理的形式，我们可以将容斥原理一般化，从更高的层面理解容斥原理。

2 容斥原理

在这一小节中，我们会从一些组合计数问题出发，得出容斥原理的形式并给出证明，然后通过一些例题得出用容斥原理解题的思维方法。

2.1 预备知识

考虑一个简单的问题：某班有a个人擅长唱歌，b个人擅长画画，c个人既擅长唱歌也擅长画画，问多少人有至少一种特长?

通过画文氏图的方法，很容易得出此问题的答案：$a+b-c$。

我们可以得出该问题的一般形式：设一个有限集为U，U中元素有两种性质P_1和P_2，而满足P_1性质的元素组成集合S_1，满足P_2性质的元素组成集合S_2，那么上面的问题相当于是求至少满足两种性质之一的元素个数，可以表示成这样：

$$|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$$

其中$|S|$表示集合S中的元素个数。

我们考虑U中元素有三种性质P_1, P_2, P_3，对应的子集是S_1, S_2, S_3，仍然可以通过画文氏图的方法，得到下面的等式：

$$|S_1 \cup S_2 \cup S_3| = |S_1| + |S_2| + |S_3| - |S_1 \cap S_2| - |S_1 \cap S_3| - |S_2 \cap S_3| + |S_1 \cap S_2 \cap S_3|$$

注意到，我们把**求并集中元素个数**转化成了**求交集中元素个数**，这一步体现了**转化**的思想。

一般地，设U中元素有n种不同的性质，第i种性质称为P_i，满足P_i的元素组成集合S_i，那么

定理2.1.1. *满足$P_1, P_2, \cdots, P_n$中至少一个性质的U中元素的个数是*

$$\begin{aligned} |S_1 \cup S_2 \cup \cdots \cup S_n| &= \sum_i |S_i| - \sum_{i<j} |S_i \cap S_j| + \sum_{i<j<k} |S_i \cap S_j \cap S_k| - \cdots \\ &\quad \cdots + (-1)^{n-1} |S_1 \cap S_2 \cap \cdots \cap S_n| \end{aligned}$$

证明. 考虑一个处于$\bigcup_{i=1}^{n} S_i$中的元素x，它所属m个集合$T_1, T_2, \cdots, T_m$，那么我们计算一下上式右边统计的x个数cnt：

$$\begin{aligned} cnt &= |\{T_i\}| - |\{T_i \cap T_j | i<j\}| + \cdots + (-1)^{m-1} |\{T_1 \cap T_2 \cap \cdots \cap T_m\}| \\ &= C_m^1 - C_m^2 + \cdots + (-1)^{m-1} C_m^m \\ &= -\left(\left(\sum_{i=0}^{m} (-1)^i C_m^i \right) - C_m^0 \right) \\ &= -\Big((1-1)^m - 1 \Big) \\ &= 1 \end{aligned}$$

结合二项式定理即可证明容斥原理的正确性。 □

至此，我们已经知道了用容斥原理计算集合的并中元素的数目的方法。稍加变形我们就能用容斥原理计算集合的交中元素的数目。

我们用$\bigcap_{i=1}^{n} S_i$表示需要计数的交集，令$\overline{S}$表示集合S关于全集U的补集，那么$\overline{S_i}$就表示不满足性质P_i的集合。考虑到需要计数的是满足$P_1, P_2, \cdots, P_n$的元素，我们进行一步**补集转化**，求不满足至少一个性质的U中的元素个数，即$\bigcup_{i=1}^{n} \overline{S_i}$，这个集合的计数方式就和之前类似，所以

定理2.1.2. *满足$P_1, P_2, \cdots, P_n$中所有性质的U中元素的个数是*

$$|S_1 \cap S_2 \cap \cdots \cap S_n| = |U| - |\overline{S_1} \cup \overline{S_2} \cup \cdots \cup \overline{S_n}|$$

综上所述，我们可以利用容斥原理在**求并集元素个数**和**求交集元素个数**这两个问题间**互相转化**，这提示我们，用容斥原理解题，是一个**转换角度**的思维方式。

2.2 经典问题

我们通过几个组合计数的经典题目，来探究如何应用容斥原理。

2.2.1 不定方程非负整数解计数

问题2.2.1. 考虑不定方程

$$x_1 + x_2 + \cdots + x_n = m$$

和n个限制条件$x_i \leq b_i$，其中m和b_i都是非负整数，求该方程的非负整数解的数目。

在解决这个问题之前，这里不加证明地给出一个结论：

定理2.2.1. *不定方程$x_1 + x_2 + \cdots + x_n = m$的非负整数解数目为$C_{m+n-1}^{n-1}$。*

在应用容斥原理前，我们需要找出全集U，以及刻画U中元素的P_i。

- U是满足$x_1 + x_2 + \cdots + x_n = m$的所有非负整数解；
- 对于每个变量i，都对应一个P_i，而P_i代表的性质是$x_i \leq b_i$。

设满足P_i的所有解组成集合S_i，那么我们需要求解的值就是：$|\bigcap_{i=1}^{n} S_i|$。

由之前的知识我们可以写出：$|\bigcap_{i=1}^{n} S_i| = |U| - |\bigcup_{i=1}^{n} \overline{S_i}|$。而$|U|$的值可以由定理2.2.1计算，我们着重考虑后面的部分，而这正是之前容斥原理的一般形式！

通过展开$\bigcup_{i=1}^{n} \overline{S_i}$，问题转化成：对于某几个特定的$\overline{S_i}$，求解满足这些条件的解的数目。一般地，给出$1 \le i_1 < i_2 < \cdots < i_t \le n$，求$|\bigcap_{k=1}^{t} \overline{S_{i_k}}|$。

考虑$\overline{S_{i_k}}$的含义，即满足$x_{i_k} \ge b_{i_k} + 1$的解的数目。而对每一个k，都要满足这个条件，即**部分变量有下界限制**，我们可以在方程的右边减去下界和$\sum_{i=1}^{k} (b_{i_k} + 1)$，那么新方程的解与我们要求的解是一一对应的！而新方程的每个变量都没有上下界限制，所以同样可以用定理2.2.1 求出。

于是我们只需要枚举$\{\overline{S_1}, \overline{S_2}, \cdots, \overline{S_n}\}$的非空子集，进行容斥原理的计算即可。

考虑解题过程，我们先是把问题写成集合的形式，找出全集U，以及我们的解需要满足的性质P_i，然后写出需要求值的式子，用容斥原理进行展开，于是我们可以着眼局部，这时的限制数就大大减少，成为一个个可解的问题，最后我们把答案合并起来就可以了。

2.2.2 错位排列计数

问题2.2.2. 称一个长度为n的排列p为错位排列，当且仅当对所有的$1 \le i \le n$，都满足$p_i \ne i$。给出n，求长度为n 的错位排列的数目。注意排列中1到n的整数都恰好出现1次。

同上题，我们首先分析全集U和性质P_i：

- U表示长度为n的所有排列；
- 对于每个位置i，都对应一个P_i，而P_i代表的性质是$p_i \ne i$。

同样设满足P_i的解组成集合S_i，那么我们需要求的值仍是$|\bigcap_{i=1}^{n} S_i|$！

于是用同样的处理方法，我们写出$|\bigcap_{k=1}^{t} \overline{S_{i_k}}|$，我们考虑$\overline{S_{i_k}}$的含义，即$p_i = i$的排列数目，而对每一个$k$，都确定了排列中一个位置的数，所以共有$t$个位置的数被确定了，而其他位置是没有限制的，所以对应的答案就是$(n-t)!$。

进一步可以推出，只要我们枚举的$\{\overline{S_i}\}$的子集的大小一样，它们对答案的贡献也是一样的！设长度为n的错位排列数是D_n，那么我们有：

$$\begin{aligned} D_n &= n! - \sum_{t=1}^{n} (-1)^{t-1} \sum_{i_1<i_2<\cdots<i_t} (n-t)! \\ &= n! + \sum_{t=1}^{n} (-1)^t C_n^t (n-t)! \\ &= n! + \sum_{t=1}^{n} (-1)^t \frac{n!}{t!} \\ &= n! \sum_{t=0}^{n} \frac{(-1)^t}{t!} \end{aligned}$$

由此我们发现，用容斥原理解决问题的时间复杂度不一定是指数级，我们可以对一些对答案贡献一致的情况进行合并，这样仍能得出高效的算法。

另外，错位排列数D_n也有递推的方法，有兴趣的同学可以另行探究。

2.3 例题解析

下面通过一些例题，看一看容斥原理在信息学中的应用。

2.3.1 HAOI2008 硬币购物

问题2.3.1. 有4种面值的硬币，第i种硬币的面值是c_i。有n次询问，每个询问中第i种硬币的数目是d_i，以及一个购物款s，回答付款方法的数目。数据规模$n \leq 10^3, s \leq 10^5$。

这题初一看是一个经典的多重背包问题，但是经过分析，我们发现单次动态规划的最好复杂度是$O(4s)$，对于多次询问根本无法承受。

但是这题与一般的背包问题有一个明显的不同：硬币（即不同的物品）只有4种。而且，若每次购物没有硬币数目的限制，可以用一个动态规划预处理后$O(1)$回答每组询问。

考虑一次询问，第i种硬币使用的数目是x_i，那么需要满足$c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 = s$，且$x_i \leq d_i$。我们发现，这与之前的不定方程非负整数解计数非常类似，只不过每个变量前有一个系数。

同样我们用容斥原理来处理这个问题，S_i表示满足$x_i \leq d_i$的解的数目，$\overline{S_i}$表示满足$x_i \geq d_i + 1$ 的解的数目，考虑若干$\overline{S_i}$的交集，即一些硬币使用数有下限，我们同样可以从s中减去下界和，问题变成了对于一个s'，若硬币使用数目无限制，有多少种不同的付款方式。而这是一个经典的无限背包问题，可以预处理。

所以对每组询问进行容斥，设最大的s为m，那么总的时间复杂度就是$O(4m + n \cdot 2^4)$。

考虑我们的解题过程，我们首先发现问题的经典算法时间复杂度过高，但是我们抓住了题目的特殊性，通过写出问题的数学形式，通过联想，应用容斥原理把问题拆分，减少了局部问题的限制数，最终解决了问题。

2.3.2 原创题 游戏

问题2.3.2. Alice和Bob在玩游戏。他们有一个n个点的无向完全图，设所有的边组成了集合E，于是他们想取遍E的所有非空子集，对某个集合S有一个估价$f(S)$，这个估价是这样计算的：考虑n个点与S中的边组成的图，我们用m种颜色对所有点染色，其中同一个联通块的点必须染成一种颜色，那么$f(S)$等于这个图的染色方案数。同时，Alice喜欢奇数，所以当$|S|$为奇数时，Alice的分值加上$f(S)$，否则Bob的分值加上$f(S)$。求最后Alice的分值减去Bob的分值的值模$10^9 + 7$的结果。数据规模$n, m \leq 10^6$。

显然我们无法枚举E的所有非空子集；另一方面，对于相同的$|S|$，联通块数目也不尽相同。我们似乎找不到一个突破口。这种情况下，我们就应该写出问题的数学形式，再进行分析。

首先，一个事实是，“同一联通块必须染相同的颜色”与“有边直接相连的两点必须染相同的颜色”是等价的。于是我们可以对每个点设一个变量，用x_i表示第i个点的颜色，x_i是$[1, m]$中的整数，那么一条无向边(i, j)就表示一个等式$x_i = x_j$。我们考虑Alice的得分$scoreA$和Bob的得分$scoreB$，令$F(C)$表示在

情况C下的染色数，用$[C]$表示一个情况C，则

$$\begin{aligned} scoreA &= \sum_{\emptyset \neq S \subseteq E, |S| \text{是奇数}} F\left(\bigcap_{(i,j)\in S} [x_i = x_j]\right) \\ scoreB &= \sum_{\emptyset \neq S \subseteq E, |S| \text{是偶数}} F\left(\bigcap_{(i,j)\in S} [x_i = x_j]\right) \end{aligned}$$

现在考虑$ans = scoreA - scoreB$，即$|S|$为奇数时贡献为正，$|S|$为偶数是贡献为负，容易想到加一个-1的幂将式子统一：

$$ans = \sum_{\emptyset \neq S \subseteq E} (-1)^{|S|-1} F\left(\bigcap_{(i,j)\in S} [x_i = x_j]\right)$$

我们把$[x_i = x_j]$这$\frac{n(n+1)}{2}$个情况用P_i代替，令$t = \frac{n(n+1)}{2}$，则P_i的i的取值范围是$1 \le i \le t$。令$Q = P_i$，那么再考虑上式：

$$\begin{aligned} ans &= \sum_{\emptyset \neq S \subseteq Q} (-1)^{|S|-1} F\left(\bigcap_{P_i \in S} P_i\right) \\ &= \sum_i F(P_i) - \sum_{i<j} F(P_i \cap P_j) + \cdots + (-1)^{t-1} F(P_1 \cap P_2 \cap \cdots \cap P_t) \end{aligned}$$

注意到这个形式与容斥原理极其相似！我们可以根据容斥原理，逆向分析出上式右边所求值的含义，即

$$ans = F\left(\bigcup_{i=1}^{t} P_i\right)$$

考虑上式右边的含义，即**至少有两个点颜色相同**的染色数！那么该问题中全集是点的染色方案集合，通过补集转化，我们就只需要求**点两两颜色不同**的染色数！而这个的计算方法是显然的，答案是$\prod_{i=1}^{n}(m-i+1)$。所以原问题答案就是$m^n - \prod_{i=1}^{n}(m-i+1)$。

细心的同学应该发现了，上面的式子中存在一个函数F，它对一个情况，即一些条件的交定义，其实我们考虑满足P_i的染色方案构成集合S_i，那么其实$F\left(\bigcap P_i\right) = \left|\bigcap S_i\right|$，这样就和之前的容斥原理形式一致了。

回顾我们的解题过程，我们首先直接写出了答案的数学形式，把一些文字条件转化为数学条件，再进行一些换元、代入，得到一个关于若干条件的交集

的式子，最终得到容斥原理的形式，**逆向分析**出问题的本质，找出算法并解决问题。如果说原来的容斥原理都是通过着眼局部，整合答案，在某种意义上进行了“微分”，那么这道题目中我们就是用的整体分析的方法，对答案的一个冗长的式子进行了“积分”，得到一个简洁的答案。这两个方向都体现了信息学中的**转化**思想。

3 容斥原理的推广

3.1 数论中的容斥原理

我们考虑一个经典的问题：给一个正整数n，求1到n中与n互质的数的个数$\varphi(n)$。

事实上我们要求的是$|\{x|1 \le x \le n, gcd(x,n) = 1\}|$，其中$gcd(a,b)$表示$a$和$b$的最大公约数。注意到这也是一个对某个集合计数的问题，但是$gcd(x,n) = 1$这个限制太“大”，因为gcd这个函数本身较“复杂”，所以，我们应该想到从最大公约数的性质，去把$gcd(x,n) = 1$这个限制拆成若干个小的限制。

我们考虑两个数a和b的质因数分解，若$a = p_1^{a_1}p_2^{a_2}\cdots p_k^{a_k}$，$b = p_1^{b_1}p_2^{b_2}\cdots p_k^{b_k}$，那么我们有$gcd(a,b) = p_1^{min(a_1,b_1)}p_2^{min(a_2,b_2)}\cdots p_k^{min(a_k,b_k)}$，其中$min(x,y)$表示$x$和$y$中的较小值。

注意到，若两个数a和b的最大公约数是1，那么它们的因数分解中一定没有相同的质数，而这是一个充要条件！所以，若n的不同的质因子有$p_1, p_2, \cdots, p_k$共k个，那么我们需要统计的x就要同时满足k个条件，即对于$1 \le i \le k$，都有x不是p_i的倍数。

现在我们可以把我们的结论写成数学的形式。设P_i表示x不是p_i的倍数这个性质，S_i表示1到n中满足P_i的数组成的集合，那么这里的全集U就是1到n的整数集合，我们需要求的就是：

$$\begin{aligned} |\{x|1 \le x \le n, gcd(i,n) = 1\}| &= \left|\bigcap_{i=1}^{k} S_i\right| \\ &= |U| - \left|\bigcup_{i=1}^{k} \overline{S_i}\right| \\ &= n - \left|\bigcup_{i=1}^{k} \overline{S_i}\right| \end{aligned}$$

这就是一个容斥原理的式子!

再考虑$\cap_i \overline{S_i}$的含义，它表示的是对于一些质数，我们统计$[1,n]$上有多少个数同时是这些数的倍数。这个的统计方法非常简单：设质数的积为m，那么答案就是$\frac{n}{m}$。

所以我们可以写出我们所求答案的表达式：

$$\begin{aligned}|\{x|1\le x\le n, gcd(i,n)=1\}| &= n-\sum_i \frac{n}{p_i}+\sum_{i<j}\frac{n}{p_ip_j}-\cdots+(-1)^k\frac{n}{p_1p_2\cdots p_k}\\ &= n\left(1-\frac{1}{p_1}\right)\left(1-\frac{1}{p_2}\right)\cdots\left(1-\frac{1}{p_k}\right)\\ &= n\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)\end{aligned}$$

这其实就是著名的欧拉公式。

从这个例子可以发现，我们容斥时考虑的是一个质数的集合，而我们取遍这个集合的子集时，得到的质数的乘积中所有质因子的次数都是1，我们称这样的数为**无平方因子数**。再看1到n中每个n的约数对答案的贡献，显然只有1和无平方因子数有贡献，而且无平方因子数所作贡献的正负与质因子的个数有关。定义一个函数$\mu(n)$，它定义在正整数集合上，且

$$\mu(n)=\begin{cases}1 & n=1\\ (-1)^k & n=p_1p_2\cdots p_k\\ 0 & \text{otherwise}\end{cases}$$

这其实就是著名的**莫比乌斯函数**。我们再重新考虑之前的问题，容斥过程中的表达式可以写成$\varphi(n)=\sum_{d|n}\mu(d)\frac{n}{d}$。

数论中的很多计数问题都可以用类似的方法解决：考察“最小元”即质数，计算“部分”即每个约数对答案的贡献，利用莫比乌斯函数进行容斥。在数论中，还有一种方法叫**莫比乌斯反演**，有兴趣的同学可以另行探究。

3.2 概率论中的容斥原理

在概率论中，对于一个概率空间内的n的事件$A_1,A_2,\cdots,A_n$，也存在着一个容斥原理：

$$\mathbb{P}\left(\bigcup_{i=1}^{n}A_i\right)=\sum_{i=1}\mathbb{P}(A_i)-\sum_{i<j}\mathbb{P}(A_i\cap A_j)+\cdots+(-1)^{n-1}\mathbb{P}(A_1\cap A_2\cap\cdots\cap A_n)$$

若事件的交集发生的概率只和事件的数量有关，且设k个事件的交集的概率为a_k，那么可以用组合数简化：

$$\mathbb{P}\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} (-1)^{k-1} C_n^k a_k$$

容斥原理在概率论中的实际应用比较少见，笔者也没有用容斥原理解决概率问题的经验。这个领域仍需更深一步的探究。

4 容斥原理的一般化

4.1 预备知识

由前面可知，容斥原理适用于对集合的计数问题，其实，对于两个关于集合的函数$f(S)$和$g(S)$，若

$$f(S) = \sum_{T \subseteq S} g(T)$$

那么就有

$$g(S) = \sum_{T \subseteq S} (-1)^{|S|-|T|} f(T)$$

这是一个更加一般的形式，而且对于之前讨论过的几种情形下的容斥原理都能找到$f(S)$和$g(S)$函数进行对应，其中S表示的是n个性质的集合。由于找到的$f(S)$和$g(S)$形式很复杂，在此略过，有兴趣的同学可以参考维基百科“容斥原理”词条。

另外，上面的式子也可以稍加变形写成这样：

$$\begin{aligned} f(S) &= \sum_{T \supseteq S} g(T) \\ g(S) &= \sum_{T \supseteq S} (-1)^{|S|-|T|} f(T) \end{aligned}$$

其实只用把之前式子中S和T替换成关于全集的补集，$\subseteq$号就换成$\supseteq$了。

下面我们通过一个例子来感知一下。

4.2 例题：有标号DAG计数

问题4.2.1. 给出n，对n个点的有标号有向无环图进行计数，输出答案模10^9+7的值。数据规模$n \leq 5 \times 10^3$。

这是一类图的计数的问题。我们考虑动态规划，因为有向无环图中有一类特殊的点，即0入度的点，所以记$dp(i,j)$表示i个点的有向无环图，其中恰有j个点的入度为0，的答案，那么我们考虑去掉这j个点后，还有k个点入度为0，写出转移

$$dp(i,j) = C_i^j \sum_{k=1}^{i-j} (2^j-1)^k 2^{j(i-j-k)} dp(i-j,k)$$

C_i^j表示从i个点中选出j个点的选法，而去掉j个点后的k个0入度点与这j个点间至少有1条边即$(2^j-1)^k$，然后这j个点还可以往除了这$j+k$个点之外的点随意连边即$2^{j(i-j-k)}$。这个算法时间复杂度$O(n^3)$。

注意我们在定义状态时，是“0入度点恰好为k”，因为限制过严，导致我们需要考虑的很多。一个常见的办法是，在状态定义中将“恰好”改成“不少于”以放宽限制。但在这个问题中，从i个点选不少于j个0度数点，选法很多，转移时重复计算的情况很复杂，我们可以考虑将这不少于j个的点特殊化，即

我们记$f(n,S)$表示n个点，只有S中的点的入度为0；类似地定义$g(n,S)$表示n个点，至少S中的点的入度为0。可以发现$g(n,S)$的转移比较简单：

$$g(n,S) = 2^{|S|(n-|S|)} g(n-|S|,\emptyset) \tag{17}$$

另一方面，我们再考虑$f(n,S)$和$g(n,S)$的关系，这也比较简单：

$$g(n,S) = \sum_{T \supseteq S} f(n,T) \tag{18}$$

注意式子(18)与之前提到的一般化的容斥原理相似，不妨将之应用：

$$f(n,S) = \sum_{T \supseteq S} (-1)^{|S|-|T|} g(n,T) \tag{19}$$

而我们的目的是求$g(n,\emptyset)$，先使用式子(18)进行推导：

$$\begin{aligned} g(n,\emptyset) &= \sum_{\emptyset \neq T} f(n,T) \\ &= \sum_{m=1}^{n} \sum_{|T|=m} f(n,T) \end{aligned}$$

再代入我们用容斥原理推出的式子(19)：

$$\begin{aligned} g(n,\emptyset) &= \sum_{m=1}^{n}\sum_{|T|=m}\sum_{S\supseteq T}(-1)^{|T|-|S|}g(n,S) \\ &= \sum_{m=1}^{n}\sum_{|T|=m}\sum_{S\supseteq T}(-1)^{|T|-|S|}2^{|S|(n-|S|)}g(n-|S|,\emptyset) \\ &= \sum_{m=1}^{n}\sum_{|T|=m}\sum_{k=m}^{n}C_{n-m}^{k-m}(-1)^{k-m}2^{k(n-k)}g(n-k,\emptyset) \\ &= \sum_{m=1}^{n}C_n^m\sum_{k=m}^{n}C_{n-m}^{k-m}(-1)^{k-m}2^{k(n-k)}g(n-k,\emptyset) \end{aligned}$$

利用一些组合数的性质可以继续进行化简。这里直接给出最后的化简结果：

$$g(n,\emptyset)=\sum_{k=1}^{n}(-1)^{k-1}C_n^k2^{k(n-k)}g(n-k,\emptyset)$$

注意到此时我们计算$g(n,\emptyset)$的时间复杂度降到了$O(n^2)$，容斥原理在中间起到了举足轻重的作用。

回顾我们的解题过程，首先我们在定义状态时放宽了状态的限制，这样可以认为新的状态是之前状态某种意义下的“前缀和”，列出等式后用容斥原理得到另一个式子，然后整合我们手中的等式推导答案的表达式，最后得到复杂度较低的算法。

5 总结

容斥原理是组合数学中一个重要的定理，在解决问题的时候，我们既可以使用“隔离法”，将所需求的解要满足的条件拆分，放宽限制，解决若干简单的子问题，再整合答案；也可以使用“整体法”，对所求的式子进行整体感知，逆向地合并条件，找出问题的本质。这里体现了**转化**的思想，当然在思考过程中也需要一些数学功底。

容斥原理同时并不是仅仅应用于组合计数，稍加变形后就可以解决一些数论或概率论的问题，其思想是一致的。而最后我们通过一些资料得知了容斥原理更为一般的形式，它适用于定义在集合上的函数，这使得容斥原理更加抽象，也让我们开阔了思路，即在一些情况下，我们用集合的形式描述我们的算

法，利用容斥原理得到另外的等式，这相当于增加了已知量，使得问题更容易入手。

在研究过程中，我从解决计数问题中体会到了一些信息学中的思维方法：转化、特殊化（放宽限制）、逆向分析，开阔了眼界；同时，在查阅容斥原理相关资料的过程中，意识到了平时学习的各种算法，其背后或许仍有继续研究的空间，所以我们应不断求知，将学习到的知识有机整合，并思考它们的本质，体会不同的算法后面的思想，形成自己的知识网络，增强自己的思维能力。

参考文献

1. http://en.wikipedia.org/ 维基百科

2. http://www.math.ust.hk/~mabfchen/Math232/Inclusion-Exclusion.pdf

3. 顾昱洲，《Graphical Enumeration》

感谢

- 感谢父母对我的养育
- 感谢我的教练成都七中的张君亮老师，以及其他所有给予我支持的老师
- 感谢罗雨屏、李凌霄、钟皓曦等同学的帮助
- 感谢CCF给我一个展示自己的机会