2015 年信息学奥林匹克
中国国家队候选队员论文集

教练：余林韵

中国计算机学会

目录

1 后缀自动机在字典树上的拓展
刘研绎 湖南省长沙市雅礼中学 **1**

2 浅谈启发式思想在信息学竞赛中的应用
任之洲 绍兴市第一中学 **17**

3 浅谈字符串匹配的几种方法
王鉴浩 绍兴市第一中学 **37**

4 后缀自动机及其应用
张天扬 湖南省长沙市第一中学 **67**

5 生成函数的运算与组合计数问题
金策 浙江省杭州学军中学 **81**

6 **ydc**的奖金命题报告
刘剑成 长沙市雅礼中学 **103**

7 浅谈分块在一类在线问题中的应用
邹逍遥 宁波市镇海中学 **113**

8 仙人掌相关算法及其应用
王逸松 浙江省杭州学军中学 **129**

9 浅谈图的匹配算法及其应用
陈胤伯 湖南省长沙市长郡中学 **149**

10 浅谈信息学竞赛中的物理问题
陈思禹 杭州第二中学 **175**

11 丢失的题面命题报告
于纪平 大连市第二十四中学 **199**

12 **DP** 的一些优化技巧
张恒捷 绍兴市第一中学 **205**

13 **Product** 命题报告
杜瑜皓 宁波市镇海中学 **217**

14 关于以源代码为输入的一类问题的初步探索
卢啸尘 宁波市镇海蛟川书院 **231**

15 集合幂级数的性质与应用及其快速算法
吕凯风 湖北省武汉市第二中学 **271**

后缀自动机在字典树上的拓展

湖南省长沙市雅礼中学 刘研绎

摘 要

本文详细分析了能够识别Trie树所有后缀的确定有限状态自动机。给出了状态数是线性、转移函数总数不超过Trie的节点数乘以字符集大小的结论及其证明，并给出了一种可行的、在允许离线时复杂度是线性的构造算法。同时对在线构造算法复杂度进行了分析，介绍了后缀自动机的相关运用，希望本文能够起到一个抛砖引玉的作用。

1 前言

在算法竞赛中，字符串处理问题是相当重要的一部分。一般有固定模式串的字符串处理问题和固定主串的字符串处理问题两大类问题。当固定模式串时，可能有不同的主串，如果模式串仅有一个，有著名的KMP算法[2,3]来处理这类问题，也可以用字符串哈希[13] 来解决模式串的出现次数问题，如果模式串较多，大家熟知的AC自动机算法[1]便可以胜任这类问题。如果主串固定，一般采用对主串构造后缀数组[11]、后缀树[4,5]、后缀自动机[7]来解决这一类问题，这一类问题已经在算法竞赛界得到了充分的研究，其中陈立杰在他的冬令营营员交流讲稿[8]提到了很多关于给串建立的后缀自动机的相关运用，罗穗骞在他的集训队论文[12]详细对后缀数组在信息学竞赛中的运用进行了讲解。当题目可以采用离线算法时，这两类问题往往可以互换，这也体现了字符串处理算法的灵活多变。

而在今年(2015)的浙江省省选中，出现了这么一个题目[1]：给出一棵$n(\leq 10^5)$个节点叶节点个数不超过20的树，树上的每条边上有一个字符，树上两点之间的路径形成了一个字符串，询问总共有多少互不相同的子串。这个题显然

[1]ZJOI2015 Day1 诸神眷顾的幻想乡，作者：陈立杰

属于固定主串的类型，但主串不再是一个串，而是一棵树。考虑直接枚举所有点对，再利用Trie树[1,13]判断字符串是否相等，复杂度达到了$O(n^3)$，即便使用字符串哈希算法[13]，时间复杂度也有$O(n^2)$，均无法通过此题。考虑将题目进行转化，因为只有20个叶节点，不妨以每个叶节点为根分别建立一棵Trie树，再将所有Trie树合并成一棵，这样一个出现在树上的串一定会出现在新的Trie树中，主串便是一棵Trie树。

为了解决上面的问题，本文提出了对能够识别Trie树所有后缀的确定有限状态自动机进行了仔细分析并提出一种构造方法，主要分为五个部分：1.将一些串上的概念定义到了Trie树上，并且对Trie树的后缀自动机进行了简单介绍；2.主要对后缀链接进行了分析，并构造出了Parent树这一结构；3.证明了状态数是线性的，也说明了自动机中转移函数总数的上界；4.详细描述了一个给Trie树构造后缀自动机的算法，当允许离线时，它的复杂度是线性的；5.阐述了Trie树后缀自动机的一些运用，加强了对后缀自动机的理解。

2 Trie树的后缀自动机

在本文中，所有字符串中的字符以及Trie树[1,13]边上的字符都属于一个有限的字符集A，由这个字符集生成的所有字符串记做A^*，一个空串将被表示为ϵ，不包含空串的集合$A^+ = A^* - \{\epsilon\}$，用$|s|$表示字符串s的串长，s_i 表示第i个字符，用$s_{i,j}$表示由字符串s第i个和第j个之间的字符组成的字符串。用$a \cdot b$表示字符串a, b 首尾相接得到的字符串（乘号可能省略）。

一个串s的所有后缀为：

$$S(s) = \{s_{i,|s|} \mid 1 \le i \le |s|\}$$

令$Minl(S), Maxl(S)$分别为一个字符串集合S中最短的和最长的串的长度。

用T表示一颗Trie树，用$T_x, T_y, T_z, \ldots$表示T中的一个节点。令T中某个节点T_x到其子树中某个节点T_y路径上经过的边上的字母顺次连接起来组成的字符串为$T_{x,y}$，下文中使用$T_{x,y}$在没有特殊声明的情况下T_y一定是T_x的子树中的节点。

令T的一个前缀为：

$$P_{T_x} = T_{root,x} \ (root \text{ is the root of } T, T_x \in T)$$

那么T的所有子串为：

$$F(T) = \{T_{x,y} \mid T_x, T_y \in T, T_y \text{ is in the subtree of } T_x\}$$

同理T的所有后缀为：

$$S(T) = \{T_{x,y} \mid T_x, T_y \in T, T_y \text{is a leaf node}\}$$

给Trie树建立后缀自动机，即建立一个有限状态自动机能够识别所有的$S(T)$，并尽可能的缩小自动机的状态数和转移数。类似给一个串建立后缀自动机，我们定义一个串在Trie树上的出现位置：

$$Right(s) = \{T_y \mid T_y \in T, \exists T_x \in T, T_y \text{ is in the subtree of } T_x, T_{x,y} = s\}$$

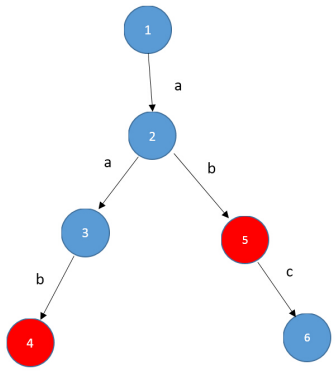
定义一种等价关系R_T（在没有歧义的时候记做R），

$$aRb \iff Right(a) = Right(b)\ (a, b \in A^*)$$

令$R(s)$表示与s等价的所有的串，即一个等价类。那么在自动机中一个状态就对应着一个等价类。令由T建成的后缀自动机为$\Phi(T)$，$\Phi(T)_\epsilon$为其初始状态，用$\Phi(T)_s(s \in F(T))$表示读入s后所处于的状态，那么转移函数被定义为$\Phi(T)_s.a(a \in A, sa \in F(T))$，有$\Phi(T)_s.a = \Phi(T)_{sa}(sa \in F(T), a \in A)$，同理$\Phi(T)_{s_1}.s_2 = \Phi(T)_{s_1 s_2}(s_1, s_2, s_1 s_2 \in F(T))$。根据定义，有：

$$\Phi(T)_a = \Phi(T)_b \iff aRb\ (a, b \in F(T))$$

并且称a是$\Phi(T)_b$这个状态对应的字符串，那么$\Phi(T)_b$对应的字符串集合就是$R(b)$。



*Picture a.*在图中所示的*trie*树中，bR等价于ab，红色节点为他们的*Right*集合。

为了分析出整个结构的状态数（即$F(T)$中R等价类的数量），我们需要对后缀自动机状态的性质进行一些分析。我们希望状态数能够尽可能的少。

引理2.1. 令$x, y, z, u \in F(T)$，有：

(i).$xRy \Rightarrow (x \in S(y)\ or\ y \in S(x))$

(ii).$x \in S(z) \Rightarrow Right(z) \subseteq Right(x)$

(iii).$(x \in S(z), z \in S(u), xRu) \Rightarrow zRuRx$

证明： (i) 不妨假设$|x| \leq |y|$，又$Right(x) = Right(y)$，考虑任意一个$T_v \in Right(x)$，有$x \in S(P_{T_v})$ and $y \in S(P_{T_v})$，因此$x \in S(y)$。

(ii) $\forall T_v \in Right(z)$，有$z \in S(P_{T_v})$，又$x \in S(z)$，因此$x \in S(P_{T_v})$，因此$T_v \in Right(x)$，原命题得证。

(iii) 由(ii)，$z \in S(u) \Rightarrow Right(u) \subseteq Right(z)$，同理$x \in S(z) \Rightarrow Right(z) \subseteq Right(x)$，又$xRu$，因此$Right(z) = Right(u) = Right(x)$即$zRuRx$。

上面个的(i)同时也说明了R等价类中长度相同的串必定唯一。

推论2.2. 对于一个等价类$R(a)$，令x为$R(a)$中长度最长的串，y为$R(a)$中长度最短的串，那么有：

$$R(a) = \{z \mid z \in S(x)\ and\ |y| \leq |z| \leq |x|\}$$

这个可以由引理2.1轻易推出。

3 后缀链接

在给串建立后缀自动机的构造算法中，后缀链接起了关键的作用[7,6]，而且整个算法与KMP算法[3]十分类似。而在给Trie建立后缀自动机时不妨保留后缀链接，并将构造过程与能够识别$A^*\{P_{T_x}\}$的确定最简状态自动机的构造算法，也就是我们熟知的AC自动机[1]的构造算法相类比。和AC自动机中的fail函数类似的，我们定义给Trie树构造的后缀自动机中后缀链接：

后缀链接是一个$A^* - \{\epsilon\}$到A^*的函数s_T（没有歧义的时候记做s），表示：

$$s_T(x) = \{z \mid z \in S(x), |z| = Minl(R_T(x)) - 1\}$$

即x的后缀中不满足xR_Ty的最长的y。

后缀链接也能帮助我们证明$\Phi(T)$的状态数是线性的，在此之前我们需要分析$s(x)$本身的性质。

引理3.1. 对于$x, y \in F(T)$，有：

(i) $xRy \Rightarrow s(x) = s(y)$

(ii) $Right(x) \subset Right(s(x))$

(iii) $Minl(R(x)) - 1 = |s(x)| = Maxl(R(s(x)))$

证明： (i) 不妨假设$|x| \leq |y|$。根据定义有$|s(x)| = Minl(R(x)) - 1, |s(y)| = Minl(R(y)) - 1$，又$R(x) = R(y)$，因此有$|s(x)| = |s(y)| < |x| \leq |y|$。由引理2.1(i)，$x \in S(y)$，又$s(x) \in S(x), s(y) \in S(y)$，因此$s(x) = s(y)$。

(ii) 由引理2.1(ii)，$Right(x) \subseteq Right(s(x))$，又$xRs(x)$为假，因此$Right(x) \neq Right(s(x))$，所以原命题得证。

(iii) 因为$s(x) \in R(s(x))$，有$s(x) \leq Maxl(R(s(x)))$，不妨令y为和$s(x)R$等价并且长度最长的串，那么有$Right(y) = Right(s(x))$且$s(x) \in S(y)$。不妨运用反证法，假设$|y| > |s(x)|$，考虑任意一个$T_v \in Right(x)$，由(ii)可知$T_v \in Right(s(x))$，那么$T_v \in Right(y)$，有$y \in S(P_{T_v})$并且$x \in S(P_{T_v})$。

若$|y| > |x|$，则$x \in S(y)$，根据引理2.1(ii)，势必有$Right(s(x)) = Right(y) \subseteq Right(x)$，与(ii) 矛盾；若$|y| \leq |x|$，因为$xRy$不成立，此时有$s(x) = y$，与$|y| > |s(x)|$矛盾，因此原命题得证。

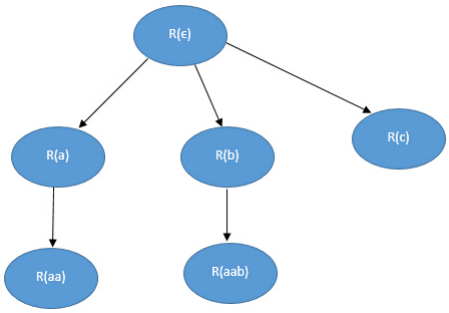
由上面的(i)可知，对于一个等价类$R(a)$，他们的后缀链接指向了相同的串，因此我们可以对后缀自动机的每个状态也建立后缀链接。不妨令$R(a)$的后缀链接指向$R(s(a))$。由(ii)或(iii)可知，这将形成一个树结构（如Picture c）。不妨称其为parent树[8]。因为有了树结构，定义$Fa(\Phi(T)_s)$为状态$\Phi(T)_s$的后缀链接，那么：

$$Fa(\Phi(T)_a) = \Phi(T)_b \iff s(a)Rb$$

引理3.2. 对于$x \in F(T)$，任取$T_v \in Right(x)$。

若$|P_{T_v}| > Maxl(R(x))$，不妨令$a = P_{T_v}$，那么$s(a_{|a|-Maxl(R(x)),|a|})Rx$。

证明： 不妨令$b = a_{|a|-Maxl(R(x)),|a|}$，根据已知可得$x \in S(b)$，根据推论2.2，因为$|b| > Maxl(R(x))$，所以$not\ bRx$，而$|b_{2,|b|}| = Maxl(R(x))$因此有$b_{2,|b|}Rx$，可知$not\ bRb_{2,|b|}$，根据后缀链接定义可知$s(b) = b_{2,|b|}$，因此$s(b)Rx$。



Picture b. 图中的树对应了*Picture a*中的*Trie*构建出的后缀自动机的*Parent*树

4 线性的状态数

在我们给一个串建立的后缀自动机中[7,6]，对x建立的后缀自动机大小是$O(|x|)$的。我们希望对于Trie建出的后缀自动机的状态规模是线性的。在定义了后缀链接后，我们需要对$\Phi(T)$进行更仔细的分析。

引理4.1. 令$x, y \in F(T)$，有：

(i) $\forall y, not\ xRs(y) \Rightarrow |Right(x)| = 1$。

(ii) $|Right(x)| = 1 \Rightarrow P_{T_v}Rx\ (Right(x) = \{T_v\})$

(iii) $\forall T_v \in Right(x), not\ xRP_{T_v} \Rightarrow \exists u, v, s(u)Rs(v)Rx\ and\ not\ uRv$

证明： (i) 反证法。不妨假设$Right(x) > 1$，那么任取$T_{v_1}, T_{v_2} \in Right(x)$，令$a = P_{T_{v_1}}, b = P_{T_{v_2}}$，那么有$|a|, |b| \geq Maxl(R(x))$（若$|a| < Maxl(R(x))$，则$R(x)$中长度为$Maxl(R(x))$的串不可能出现在$T_{v_1}$，矛盾），且$|a|, |b|$不能都等于$Maxl(R(x))$。

若$a \notin R(x)$，则$|a| > Maxl(R(x))$，那么引理3.2，$s(a_{|a|-Maxl(R(x)),|a|})Rx$；若$a \in R(x)$，又$T_{v_1} \neq T_{v_2}$，因此$a \neq b$，又$|b| \geq Maxl(R(x))$，那么可得$|b| > Maxl(R(x))$（否则两串会相等），同理可得$s(b_{|b|-Maxl(R(x)),|b|})Rx$。原命题得证。

(ii) 可知$x \in S(P_{T_v})$，那么$|Right(P_{T_v})| \leq |Right(x)|$，又$|Right(x)| = 1$，可得$Right(P_{T_v}) = \{T_v\} = Right(x)$，即$P_{T_v}Rx$。

(iii) 由(ii)的逆否命题可推得：$|Right(x)| \neq 1$，那么$|Right(x)| \geq 2$。根据已知可得$\forall T_v \in Right(x), |P_{T_v}| > Maxl(R(x))$，并且$\exists T_{v_1}, T_{v_2}$，使得$P_{T_{v_1}}, P_{T_{v_2}}$的最长公共后缀长度为$Maxl(R(x))$。

假设$\nexists T_{v_1}, T_{v_2}$，由已知可得$P_{T_v}, T_v \in Right(x)$这些串的最长公共后缀的长度至少为$Maxl(R(x))$，而任意两个$P_{T_v}$的公共后缀都不为$Maxl(R(x))$，又$|P_{T_v}| >$

$Maxl(R(x))$，因此他们的最长公共后缀的长度一定会大于$Maxl(R(x))$，根据引理2.1(ii)，可知他们的最长公共后缀一定与xR等价，而长度又大于$Maxl(R(x))$，因此假设不成立。即一定存在T_{v_1}, T_{v_2}，使得$P_{T_{v_1}}, P_{T_{v_2}}$的最长公共后缀长度为$Maxl(R(x))$。

不妨令$a = (P_{T_{v_1}})_{|P_{T_{v_1}}|-Maxl(R(x)),|P_{T_{v_1}}|}$，$b = (P_{T_{v_2}})_{|P_{T_{v_2}}|-Maxl(R(x)),|P_{T_{v_2}}|}$，有上面的结论可知$a \neq b$，因此$a \notin S(b)$and $b \notin S(a)$，根据引理2.1(i)可知$not\ aRb$。根据引理3.2可知$s(a)Rx, s(b)Rx$，至此原命题得证，a, b即为所求的u, v。

有了以上分析，我们可以对状态数的规模进行计算了。

定理4.2. *给一棵Trie树建立的后缀自动机，其状态数是线性的。*

证明： 令$T_v \in T$。

考虑由后缀链接构成的Parent树，由引理4.1(i)&(ii)可知对于所有Parent树上的叶节点$\Phi(T)_s$一定存在$P_{T_v}Rs$，由引理4.1(iii)可推知任意一个状态$\Phi(T)_s$只要满足对所有的T_v均有$not\ sRP_{T_v}$，那么$\Phi(T)_s$一定有两个以上的子节点。也就是说Parent树上叶节点和只有1个子节点的节点均对应着某个P_{T_v}，其他节点均有两个以上的子节点。而T_v的数量是$|T|$并且$\Phi(T)_1.P_{T_v}$所达到的状态是唯一的。那么Parent树上叶节点和只有1个子节点的节点数之和不会超过$|T|$，整棵树的节点数也就不会超过$2|T|$，因此是线性的。

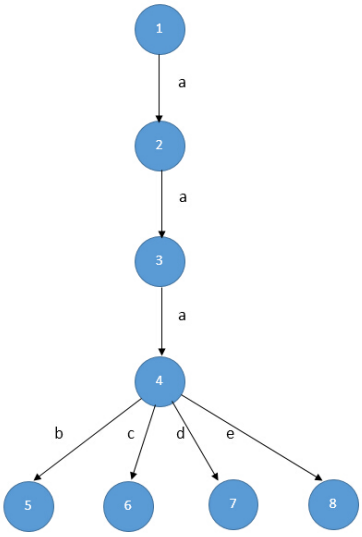
有了线性的状态数，如果转移函数（即边数）的总数也是线性的，那么我们就能推得整个自动机是线性的。但事与愿违，与给字符串建立后缀自动机不同，给Trie树所建立的后缀自动机转移函数的总数将是状态数乘以字符集大小级别。下面将证明不会超过这个数量级并且给出一个达到这个数量级的Trie树的构造方法。

定理4.3. *给一棵Trie树建立的后缀自动机，其转移函数的上界是 O((Trie的节点数) × (字符集大小)) 的。*

证明： 不妨令Trie树为T。

由定理4.2，$|\Phi(T)|$不会超过$2|T|$，再考虑对于每一个$\Phi(T)_s$，其转移函数最多有$|A|$个。因此转移函数的总数不会超过$2|T| \times |A|$。命题得证。

构造一个Trie树使得其后缀自动机转移函数规模达到上界。考虑Trie树T，显然对于A中没有出现在T的树边上的字符我们不需要将其考虑进来，那么$|A| \leq |T|$。T中的一部分是一条长度为$|T|/2$的边上的字母均为同一个字母的链，剩下的点全部以两两不同的字符连接这条链的最后一个点。不妨令链的边上的字母均为a，剩下的点与链的连边上的字符集合为B，那么$a^i b \in S(T)\ (0 \leq i \leq |T|/2, b \in B)$，又显然有$not\ a^i R a^j\ (i \neq j)$，因此对于每一个$\Phi(T)_{a^i}$，所有的$b \in B$都是其一个转移函数，因此$\Phi(T)$的转移函数总数达到了$|T|/2 \times |B|$也就是$|T|/2 \times |A|/2$，和定理4.3中的上界是同一个规模。



*Picture c.*一个能使转移函数总数达到上界的例子

虽然给Trie建立后缀自动机的转移数并不是线性的，但是在大多数题目以及实际问题中字符集都不会很大，一般字符集都是小写英文字母或者阿拉伯数字，在字符集是常数的情况下其转移数仍是线性的，仍不失为一个有效的处理Trie树有关字符串问题的方法。

5 构造算法

在给串建立后缀自动机的算法[7,8]中，我们使用的是一个在线的增量算法，即在建立好s的后缀自动机上建立出$sa(a \in A)$的后缀自动机，那么不妨在

给Trie建立的过程中也考虑在线的增量算法。有了上面对状态数和转移数的分析，我们知道构造算法的复杂度下界是$|T| \cdot |A|$。

令Trie树T在节点T_x加入一个字符为a的边连向子节点T_y得到的新的Trie树是TN，$s = P_{T_x}$。考虑$\Phi(TN)$在$\Phi(T)$的基础上有哪些需要修改的地方。

我们先对状态数进行分析。

5.1 状态的变化

对于任何$s \in F(T)$，因为s继续延伸可能是T的一个后缀，因此一定存在$\Phi(T)_s$这个状态。反之对于$s \notin F(T)$，s延伸不可能得到T的后缀，因此一定不存在$\Phi(T)_s$这个状态。

如果一个等价类在加入T_y之后没有改变，那么他们对应的后缀自动机上的状态也不会改变，有$\Phi(TN)_s = \Phi(T)_s$。

Case 1： 如果T_x已经存在一个儿子使得其与T_x的连边上的字符为a，那么我们不需要对$\Phi(T)$进行任何修改。

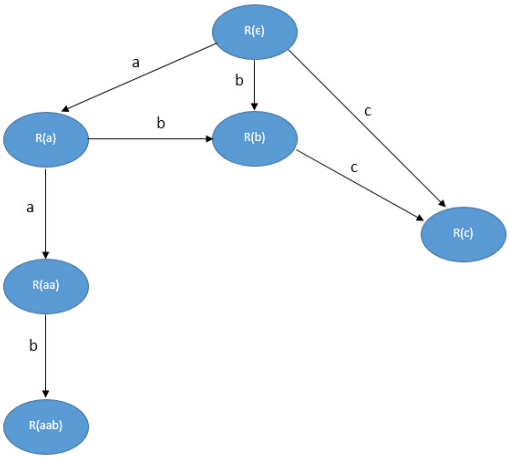
Case 2： 如果$sa \in F(T)$，那么有$\Phi(T)_s.a = \Phi(T)_{sa}$，我们可能把$R_T(sa)$分成两个等价类。

考虑如果$x, y \in F(T), xR_{TN}y$，那么如果x, y的$Right_{TN}$集合中同时出现T_y就一齐删除，因此肯定有xR_Ty。也就是说$\forall xR_{TN}sa, yR_{TN}sa$，均有$xR_Ty$，即与$saR_{TN}$等价的原本都是同一个$R_T$等价类$R_T(sa)$中的。当$\forall x \in R_T(sa), xR_{TN}sa$时，此时不会有新的状态出现。当$\exists x \in R_T(sa), not\ xR_{TN}sa$时，需要把$R_T(sa)$分成两个等价类，一类与$saR_{TN}$等价，一类不与其等价即可（如果不与$saR_{TN}$等价，那么他们的$Right_{TN}$集合中一定不存在$T_y$，即$Right_{TN}$和原来的集合$Right_T$相等）。

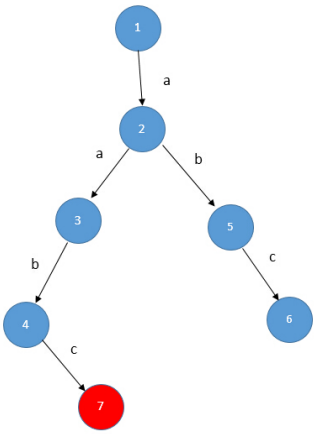
不妨考虑两个新状态的后缀链接。

令$u, v \in F(T), uR_{TN}sa, not\ vR_{TN}sa$且$uR_Tsa, vR_Tsa$，那么在$T_y$加入后，$u$将被分进第一类，$v$则是第二类。因为$sa = P_{T_y}$，又有$uR_{TN}sa$，可知$|u| \le |sa|$（如果$|u| > |sa|$则$u$不可能出现在$T_y$，$uR_{TN}sa$也就不成立了），因此对于任意$x \in S(u)$的$Right$集合中都新加入了$T_y$，其后缀链接不会改变，$s_{TN}(u) = s_T(u)$。再考虑$v$的情况，$vR_Tsa$却$not\ vR_{TN}sa$，考虑$Right_T(sa) \to Right_{TN}(sa)$只多出了$T_y$，那么必然有$T_y \notin Right_{TN}(v)$，根据引理2.1(i)，可推知$sa \in S(v)$，对所有$x \in S(v), |x| > |sa|$的串$x$，因为$vR_Tsa$，根据推论2.2，$xR_Tv$，又他们的$Right$集合在加入$T_y$前后没有变化（$Right_T(x) = Right_{TN}(x), Right_T(v) = Right_{TN}(v)$），因此$vR_{TN}x$，此时就能

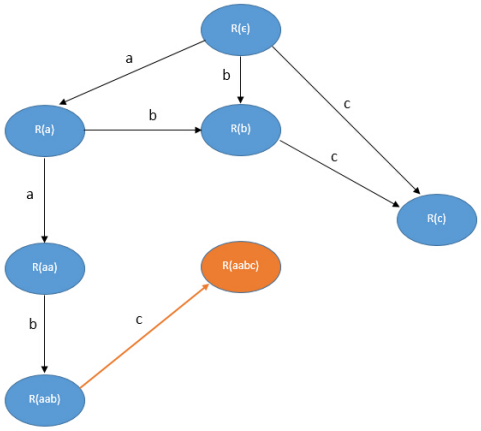
得出结论：$s_{TN}(v) = sa$。



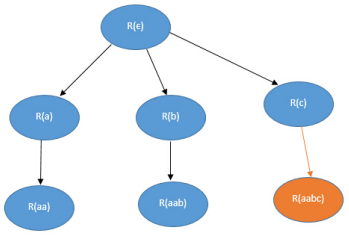
Picture d.为Picture a中的Trie建立的后缀自动机



Picture e.



Picture f.



Picture g.

Picture e&f&g.分别为给Picture a的Trie树加入一个节点的Trie，新Trie树的Parent树及后

缀自动机。红色的节点为新加入*Trie*的节点，橙色的节点和边表示*Parent*树和后缀自动机的变化。

既然已经知道sa所属的等价类$R_{TN}(sa)$，那么sa的所有后缀已经在这之前加入到$R(s_{TN}(sa)), R(s^2_{TN}(sa)), \ldots$中去了，而那些状态并不会改变，因此不需要再考虑。

Case 3： 如果$sa \notin F(T)$，我们将新建一个等价类$R_{TN}(sa)$，因为出现了新的后缀，因此有一个等价类可能分裂。

有一个比较显然的结论：$s_{TN}(sa)$等于$S(sa)$中一个出现在$F(T)$中的最长的串。证明也比较简单，若$x \in S(sa), x \notin F(T)$，那么$x$只会在$T_y$处出现，而$sa$亦只会在$T_y$出现，即$Right(sa) = Right(x) = \{T_y\}$，即$xRsa$；若$x \in S(sa) \& x \in F(T)$，那么$|Right(x)|$势必大于1，即$not\ xRsa$。

这个结论将给我们的构造过程带来很多方便。$\forall y \in S(sa), y \notin S(s_{TN}(sa))$将组成一个新的$R_{TN}$等价类，我们为其在自动机中新建一个状态，对于$y \in S(s_{TN}(sa))$，也就是剩下的那些串，我们可以将$s_{TN}(sa)$看做$sa$带入*Case 2*（虽然*Case 2*中有利用到$sa = P_{T_y}$的性质，但s_{TN}本身包含了极长的性质，因此不会造成错误），那么所有串都找到所属的状态。现在的问题就是怎么求$s_{TN}(sa)$了。根据上面的结论，我们要找sa中在$F(T)$出现至少一次的最长后缀，不妨先找到$\Phi(T)_s$，那么我们需要找到s中最长的后缀使得在其之后添上a仍然出现在$\Phi(T)$中，这时候就可以使用我们的转移函数了。根据推论2.2，我们只需在$\Phi(T)_s, \Phi(T)_{s_T(s)}, \Phi(T)_{s^2_T(s)}, \ldots$中找到第一个存在$a$转移的状态$\Phi(T)_{s^i_T(s)}$，那么$s^i_T(s)a$即为$sa$中最长的出现2 次的后缀，即$s_{TN}(sa) = s^i_T(s)a$。

从上面三种情况的分析可以发现，当Trie上新增了一个字符状态最多只会多出2个，这也证明了其状态是线性的，同时也讨论了给后缀链接带来的变化。但我们并没有讨论转移函数的修改情况，接下来将对转移函数的修改情况进行分析。

5.2 转移函数的变化

在*Case 2*中，如果没有状态新建就肯定不会有转移函数的变化。否则原来的等价类$R_T(sa)$将会拆成$R_{TN}(sa)$和$R_{TN}(v)$，考虑他们的转移函数，因为加入的T_y是叶节点，所以不会对原有的转移函数产生影响，$\Phi(TN)_v$和$\Phi(TN)_{sa}$的转移函数

都和$\Phi(T)_{sa}$的一致，将他们复制过来即可。由于之前的等价类$R_T sa$删除那么可能存在$\Phi(T)_x.a = \Phi(T)_{sa}$的状态，这些状态的转移函数肯定需要修改。根据*Case 2*中的分析，在新建的两个状态$\Phi(TN)_u$和$\Phi(TN)_v$中，有$Fa(\Phi(TN)_v) = \Phi(TN)_u$，而所以这些$\Phi(T)_x.a = \Phi(T)_{sa}$的状态，要么$\Phi(TN)_x.a = \Phi(TN)_u$，要么$\Phi(TN)_x.a = \Phi(TN)_v$，由*Case 3*，即所有$\Phi(T)_s, \Phi(T)_{s_T(s)}, \Phi(T)_{s_T^2(s)}, \ldots$直到$\Phi(T)_{s_T^i(s)}.a \neq \Phi(T)_{sa}$这些状态的$a$转移一定均为$\Phi(TN)_u$，同理可知其他$\Phi(T)_x.a = \Phi(T)_{sa}$的状态新的$a$转移都会到$\Phi(TN)_v$。因为其他的转移我们很难依依找到具体的状态是什么，我们保留$\Phi(TN)_v = \Phi(T)_{sa}$，新建一个状态代表$\Phi(TN)_u$，暴力修改所有$\Phi(T)_s, \Phi(T)_{s_T(s)}, \Phi(T)_{s_T^2(s)}, \ldots$直到$\Phi(T)_{s_T^i(s)}.a \neq \Phi(T)_{sa}$这些状态的$a$转移。由推论2.2和引理2.1可推知，$s_T^i(s)a = s_T(sa)$。

在*Case 3*中，新建了一个状态$R_{TN}(sa)$，其余部分和*Case 2*一致。因为T_y是叶节点，因此新建的状态的转移函数一定为空。由于新建了一个状态，可能有$\Phi(T)_s$等状态新增加了转移的边，由*Case 3*的分析，我们只需要将$\Phi(T)_{s_T^i(s)}$ $(i \geq 0, s_{TN}(sa) \neq s_T^i(s)a)$这些状态的转移函数新增转移$a$指向$\Phi(TN)_{sa}$即可。

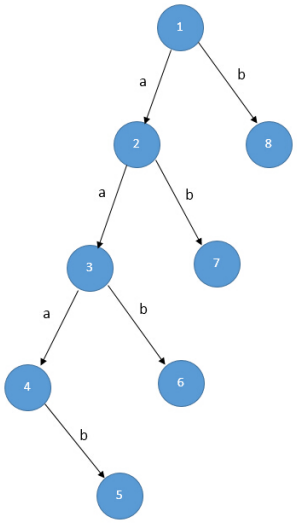
5.3 构造算法复杂度分析

和串类似，整个构造算法是一个在线的增量算法，对于T，他的执行次数是$|T|$次。每次只会新增两个状态，这里的复杂度是$O(|T|)$。每个新增的状态的后缀链接只会修改一次，这部分也是$O(|T|)$的。考虑转移函数的情况，在*Case 2*中可能存在一个转移函数的复制，一个状态的转移函数最多$|A|$个，因此这一步是$O(|T||A|)$的，比较复杂的是我们暴力修改其他转移函数的时间复杂度，可以证明他的时间复杂度不超过在trie树中叶节点深度之和。不妨令T中叶节点深度之和为$G(T)$。

证明的思路类似势能分析。不妨令$Dep(s), s \in F(T)$表示节点$\Phi(T)_s$在Parent树中的深度，考虑$\Phi(TN)_s$和$\Phi(TN)_{sa}$在Parent树中到根的路径，根据引理2.1(ii)，如果存在$R(ba)$就一定存在$R(b)$，因此$Dep(sa) \leq Dep(s) + 1$。令$c = s$，我们在暴力修改的时候会不断的让$c = s(c)$，同时修改c的转移函数，同时$Dep(c)$会减1。当$\Phi(T)_c.a$这个转移存在时，$Dep(c) \geq Dep(s(sa))$，因为$R(s(sa))$这个状态会被拆成两个，因此还有可能c有可能继续在Parent树上往上跳同时修改$\Phi(TN)_c.a$，直到不在往上时，有$\Phi(TN)_c.a = Fa(\Phi(TN)_{s(sa)}) = Fa(Fa(\Phi(TN)_{sa}))$，因此$Dep(sa) \leq Dep(s) + 3$并且每一次修改一定会造成深度的减少，可以将$Dep(s)$看做当前状态

的势能函数，不妨考虑对于每个T中的叶节点，单独做一次从根到该点的插入算法，一次插入树上的路径，每一次插入操作$Dep(s)$最多增加3，每一次暴力修改转移函数$Dep(s)$会减少1，那么我们暴力修改转移函数的次数实际上是深度级别的，将所有叶节点对应的操作次数加起来就能得到上面的结论。

上面这个复杂度并不优秀，但在线构造的复杂度是可以达到这个级别的。详见*Picture h*。



Picture h.如果按照节点的编号顺序进行加入，复杂度会达到$O(G(T))$。

当没有要求在线时，我们可以按照FIFO序，即宽度优先搜索顺序将Trie上的节点加入。这样在J. A. Blumer等[9,10]给出的对于串的构造算法复杂度的证明同时也对Trie树适用。只有固定的子串组合会使我们需要暴力修改转移函数[6]，这提示我们这一部分的复杂度不会超过$O(|T|)$。

综上所述，如果要求在线，那么构造算法的复杂度将是$O(G(T))$，如果允许离线，我们可以按照FIFO序构造得到一个复杂度是$O(|T||A|)$的算法，这已经是我们的复杂度下界了。

6 运用

6.1 ZJOI 2015 诸神眷顾的幻想乡

这道题的题面已经出现在前言之中，这里就不再赘述了。

题面已经被我们转化为：统计一个Trie树有多少本质不同的子串。利用上面介绍的给Trie树建立后缀自动机的方法，我们可以得到一个能够识别一个Trie树所有后缀的自动机。而Trie树的一个子串必定是某个后缀的前缀，这和给串建立后缀自动机有着相同的性质。同理我们可以与之相类比，对于一个状态$\Phi(T)_s$，它代表的本质不同的子串数为$|R(s)|$，那么我们将所有状态的$|R(s)|$加起来就能得到答案了。

当然还有另一种解决方案，在建立出自动机后，我们直接统计有多少不同的串不会被自动机拒绝，这就相当于统计不同的子串个数，也等价于从自动机初始状态开始有多少种不同的转移路径使得存在，把自动机看成一个有向无环图，在上面进行简单动态规划就能算出答案。

根据上面的复杂度分析，试题中并没有要求在线，因此我们可以采用离线构造算法，复杂度为$O(n|A|)$。实际上因为叶节点总数是常数这个条件非常强，可以证明按照上面的构造方法构造出来后缀自动机的时间复杂度是$O(n)$的，因为我们可以将根到每个叶子单独拆成字符串，之后与串的复杂度分析一致[7]，因为与本文内容不相关，因此省略详细证明过程。

至此我们圆满解决了前言中的问题。

由这个试题扩展开，我们可以知道对于大部分可以用后缀自动机处理的字符串问题均可以拓展到Trie上，比如求Trie树的第k小子串，多个Trie树的最长公共子串，求循环串在Trie树上的出现次数等等，还有一些串上很容易处理但推广到Trie上就变得复杂的题目，比如说求有多少本质不同的串的前缀是给定的串，这些问题都可以在给Trie树构建出后缀自动机后解决。

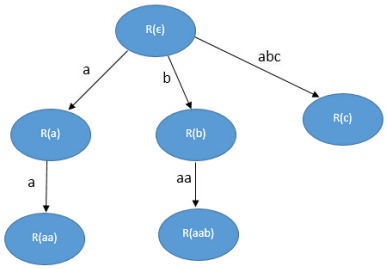
6.2 与AC自动机

我们熟悉的字符串算法：AC自动机，他将n个串建立成Trie树之后再建立好fail指针，然后可以识别对于输入的串每个模式串的出现次数。我们可以用给Trie建立后缀自动机来代替这个用法：将n个串建立成Trie之后再为这个Trie树建立后缀自动机，我们在读入识别串的时候，如果当前状态$\Phi(T)_s$不存在转移a，那么我们跳到$Fa(\Phi(T)_s)$继续进行识别，显然这样跳跃的次数不会超过识别串长。

考虑复杂度，我们用n个串建立出来的Trie树的T，其$G(T)$一定不会超过n个串的总串长，也就是说复杂度不会劣于AC自动机的复杂度。但我们的构造算法

是一个可以是在线算法，而AC自动机的构造算法只能是离线的。

6.3 Parent树



*Picture h.*将*Picture b*边上的字符补全。

这实际上是将所有Trie树中的前缀翻转过来再构成一棵Trie树，类似字符串的后缀树。我们给每个状态$\Phi(T)_s$在parent树中的儿子排序，即对于$Fa(\Phi(T)_x) = \Phi(T)_s$的$\Phi(T)_x$按照$x_{|x|-Maxl(R(s))}$的大小排序，我们将得到一个Trie树的前缀数组（即按照所有前缀翻转过来排序）。我们需要给$\Phi(T)_s$维护一个$T_v \in Right(s)$，配合树上倍增我们就能快速找到$x_{|x|-Maxl(R(s))}$了。

有了这个性质，我们可以快速找到在当前识别串之前加上一个字符$a \in A$，应该对应的状态位置。即假设当前位于状态$\Phi(T)_s$，我们在其Parent树的子节点中二分查找出$\Phi(T)_{as}$，或者判断出$\Phi(T)_{as} = \Phi(T)_s$。

7 致谢

感谢雅礼中学屈运华老师长期以来在学习生活上的指导和关心。
感谢父母对我的养育之恩。
感谢雅礼中学朱全民，汪星明老师的教导。
感谢CCF提供的交流平台和机会。
感谢国家集训队教练的辛勤付出。
感谢清华大学黄志翱，上海交通大学黄宇扬学长对我的帮助。
感谢雅礼中学杨定澄，匡正非，刘剑成同学为本文审稿。
感谢所有对我有过帮助的同学。

参考文献

[1] A.V. Aho and MJ. Corasick, Efficient string matching: An aid to bibliographic research

[2] R-S. Boyer and J.S. Moore, A fast string searching algorithm

[3] D.E. Knuth, J.H. Morris and V.I Pratt, Fast pattern-matching in strings

[4] P. Weiner, Linear pattern-matching algorithms

[5] E.M. McCreight, A space-economical suffix-tree construction algorithm

[6] M. Mohri, P. Moreno and E. Weinstein General Suffix Automaton Construction Algorithm and Space Bounds

[7] M. Crochemore, Transducers and repetitions

[8] 陈立杰，后缀自动机

[9] J. A. Blumer, Algorithms for the directed acyclic word graph and related structures

[10] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis

[11] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches

[12] 罗穗骞，后缀数组——处理字符串的有力工具

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

浅谈启发式思想在信息学竞赛中的应用

绍兴市第一中学 任之洲

摘 要

启发式算法是基于经验设计的算法，相较于传统的最优化算法，启发式往往通过牺牲精确性或稳定性以换取速度的提升。本文试图给出启发式算法在信息学竞赛中的进一步定义，并探讨一些常用的启发式算法和一些问题模型，其中包括一种最小斯坦纳树的$O(|S|\,|V|^2)$的近似解算法，并证明了该算法可以做到$2(1-\frac{1}{leaf})$近似。

1 引言

启发式算法是相对于最优化算法提出的一类算法，是基于直观或经验构造的算法，目的是在可接受的时间、空间开销下得到问题的一个较优解。这个概念非常宽泛，大多数启发式算法都十分复杂，并不能直接应用到OI竞赛中。

笔者对一些启发式算法进行简化和改进，使其能够适用于OI竞赛。并对于OI竞赛中的启发式算法，进一步给出了定义：

- 以直观感受为依据对算法进行调整优化，牺牲算法的稳定性换取速度提升的算法。
- 以直观感受为依据直接构造较优解，牺牲算法的正确性完成快速求解的算法。

在一些常用算法中也包含着启发式思想，这些算法经过优化后虽然没有降低最坏复杂度，但是算法效率都有了普遍的提升，比如：

- *k-d tree*最近点查询。
- 队列优化的*Bellman-Ford*算法（*SPFA*算法）及其贪心优化。
- 模拟退火算法、遗传算法等由自然科学衍生得到的随机搜索算法。

本文将对三个启发式算法进行探讨：

- A*算法及其变种，利用A*算法可以搭建起近似化算法和最优化求解的桥梁。
- 朴素贝叶斯分类，将直观判断和数学推导相结合的概率分类算法。
- 启发式合并，一种简单的合并类问题优化算法。

除了套用已有的经典模型，在很多时候还可以自己着手设计启发式算法，本文耗费了大量篇幅介绍了两个经典问题的启发式算法：

- 集合带权均分问题(2-*partition problem*)，得出了一个正确率很高的近似化算法，其设计流程可以给以启迪。
- 斯坦纳树问题(*steiner tree problem*)，耗费了较大篇幅介绍，得出了一个近似偏差程度可估计的高效算法，通过结合A*算法模型进一步得出了k小斯坦纳树近似解的高效算法。

对本文的每一部分，作者都给出了例题，希望能起到抛砖引玉的作用。

2 A*算法

在一些问题中，很难找到有效的多项式复杂度算法，只能采取搜索策略，但问题的状态数量往往非常的多。这时即可考虑设计估价函数进行剪枝，本文接下来将给出一些利用估价提高算法效率的方法。

2.1 启发式搜索

设当前状态为s，到状态s的实际代价为$g(s)$，从状态s到目标状态的估价为$h(s)$，实际最优代价为$h^*(s)$。

设$f(s) = g(s) + h(s)$，可以利用这个估价，每次选取$f(s)$最优[1]的状态扩展，并根据估价的性质进行剪枝。

2.2 $h(s) = h^*(s)$

当估价函数可以准确得出精确的代价时，每次选取$f(s)$最优的状态扩展，将可以快速得到最优解。

在得到最优解之后，继续扩展剩下的状态将可以进一步得到k优解，且避免了多余状态的计算。对于解法的一般化和特殊问题的进一步优化请参见俞鼎力的2014集训队论文。

[1]本文默认问题为最小化代价。

2.3 $h(s) \leqslant h^*(s)$

有时候为了方便计算，会舍去题目中的一些限制来进行估算，得到一个“优”于最优解的粗略解。

这时估价能够提供准确的剪枝依据，减少搜索状态的范围。

2.3.1 启发式迭代加深

启发式迭代加深算法（IDA*）是A*算法的一个变种，结合了迭代加深思想和估价函数，在一些状态量大、存储困难的问题中可以大量节省空间需求，书写代码也更加简洁。

在朴素的迭代加深算法中，每次会选择固定的步长进行搜索，在搜索过程中利用搜索深度限制进行剪枝。

在启发式迭代加深算法中，通过计算估价来进行剪枝。由于$h(s) \leq h^*(s)$，计算被剪掉状态最小的$g(s) + h(s)$可以估计下一个可能的答案取值，减少固定步长迭代的无用计算。

2.3.2 例题一

例1 (UVa 10181 15-Puzzle Problem)**.**

十五数码问题是八数码问题的一个扩展，在$4 * 4$的方格中有15个带编号格子和一个空格子，每次可以把空格子四周的一个格子移动到空格子位置。

A random puzzle position | The missing Tile moves to right. Denoted by `R` | The missing Tile moves upwards. Denoted by `U` | The missing Tile moves to the left. Denoted by `L`

要求在最小步数移动到目标状态，数据保证除了无解情况都能在45步内找到解。

对于无解的判断需要对数码问题进行进一步分析，本文不做深入讨论。

数码问题可以设计一种很简单的估价，求出所有非空格子到目标位置的曼哈顿距离之和作为$h(s)$，相当于忽略了只能和空格子交换的限制，也忽略了交换操作对另一个格子的影响。

由于忽略了一些限制，所以$h(s) \le h^*(s)$，对于保证45步内出解的数据剪枝效果和迭代次数的优化效果都是不错的。

2.4 Anytime algorithm

也称可中断算法(interruptible algorithm)。

并不是所有问题都容易找到高效的估价方式，有时候只能对答案进行粗略估计，无法控制$h(s) \le h^*(s)$，也就无法有效缩小搜索范围。在这种情况下，该问题则往往以提交答案题的方式出现，可以选用逐步变优的算法，在比赛时间内尽可能地得到较优的解。

2.4.1 例题二

例2 (CTSC 2009 Day2 N^2数码游戏)**.**

N^2数码游戏由N^2-1个滑块和一个空格组成，目标是通过上下左右移动空格使得N^2-1个滑块排成某一种特定的顺序。

提交答案题类型，$3 \le N \le 6$，共10组数据[2]

相较与例 1 ，这一题的数据范围大了许多，但是与之相对应的该题也不再要求输出精确解，只要能在比赛的许可时间内得到较优解就可以获得可观的分数。

直接套用例 1 中的估价用IDA*搜索，可以在短时间内得到其中5个测试点的最优解，但是这个估价计算出的值和精确的答案还是存在很大的差距的，对于剩下的测试点就无能为力了。

有一个简单的解决方法，设原估价函数为$h(s)$，得到新的估价$h_2(s) = ch(s)$，其中c为一个常数，根据具体测试点调整。

和$h(s)$相比，$h_2(s)$能更接近地估计真实步数，考虑将其套用在A*算法上。但是A*算法的内存开销非常大，这也是在例 1 中不选用A*的原因。

对于提交答案题，可以采用一种折中的手段，只保留估价最好的一部分状态，每次选择最佳状态扩展，如果新状态存储不下就舍弃最差的状态。由于该问题的解法方案很多，所以舍弃一些“差”的状态对最终答案的影响不大，除了一个特殊点都可以获得9 ~ 10分。

[2]具体分布：1组$N = 3$，4组$N = 4$，4组$N = 5$，1组$N = 6$。

3 启发式合并

启发式合并是一个简单的合并策略，可以有效优化合并算法的效率。

3.1 数据结构的启发式合并

假设有两个数据结构需要合并，这两个数据结构A、B维护的元素数量分别为a和b（$a \le b$），我们选择把数据结构A中维护的所有信息单独拆出，逐一插入到数据结构B 中。

定理3.1. *设经过若干次合并操作后得到了一个元素数量为n的数据结构，那么启发式合并策略所造成的插入操作次数是$O(n \log n)$的。*

Proof. 对数据结构中维护的每个元素单独考虑，当一个元素被插入到另一个数据结构中时，它所在数据结构中维护的元素量至少变为原来的2倍，经过$O(\log n)$次插入操作后，就必定已在最终的数据结构中了。 □

启发式合并策略可以完成一些结构复杂的数据结构的合并，也可以简单拓展到其他数据集的合并中。

3.1.1 例题三

例3 (HNOI2009 梦幻布丁)**.**

n个布丁摆成一行，进行m次操作。

每次将某个颜色的布丁全部变成另一种颜色的，然后再询问当前一共有多少段颜色。

数据范围：$n \le 10^5$

可以用链表或*vector*等数据结构维护每种颜色的布丁，修改时暴力拆解较小的链表或*vector*，逐个修改计算贡献。

根据定理 3.1 ，修改次数有$O(n \log n)$保证，暴力维护即可。

3.1.2 例题四

例4 (BZOJ3510 首都)**.**

维护一个n个点的森林，共三种操作：

- 连接点u和点v，保证连接后仍为森林。
- 询问点u所在树的重心（若有两个点可作为重心，选择编号较小的）。
- 询问所有树重心编号的异或和。

先考虑给一棵树增加一个叶子节点，重心只会往新叶子节点的方向移动，且最多移动一次。

当连接两棵树时，可以把点数较小的树暴力拆分，以叶子节点的形式加入，同时维护重心。根据定理 3.1 ，加入叶子的次数有$O(n\log n)$保证，只需另维护一个数据结构用于判断重心是否移动。

3.2 并查集的按秩合并

按秩合并是一个实用的并查集维护方式，每次选择把较小集合的根接到较大集合的根下，根据定理 3.1 易得树高为$O(\log n)$。具体实现时，也可以直接维护树深度按秩合并。

按秩合并可以在不进行路径压缩的情况下保证并查集的树深度，在一些问题中可以避免路径压缩时的信息缺失。

3.2.1 例题五

例5 (NOIP2013 货车运输)**.**

有n座城市，编号$1\sim n$，城市之间有m条双向道路，每一条道路对车辆都有重量限制z_i，简称限重。

现在有q辆货车在运输货物，司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

数据范围：$n\le 10000$，$m\le 50000$，$q\le 30000$，$z_i\le 100000$

考虑用*kruskal*求出最大生成树，最大生成树上的路径最小值就是这两个点连通的最宽限重，可以用倍增求解，时间复杂度$O(m\log m+q\log n)$。

如果用按秩合并代替路径压缩，可以记录并查集中每个点和它父亲连通时的边的限重，路径最小值仍可以表示两个点连通的最宽限重。暴力求解的时间复杂度为$O(m\log m+q\log n)$，比前一做法简洁很多。

由于转化后的问题仍是路径最小值，所以仍然可以采用倍增，询问部分理论上可以做到$O(q \log\log n)$。若选用线性的排序方法，并对并查集进行路径压缩（仍按原方法连边），时间复杂度理论上可以做到$O(z + m\alpha(n) + q \log\log n)$。

4 抽象问题与机器学习

当我们遇到一些抽象的分类问题时，直观的感觉并不能提供准确的判断依据，这时可以考虑直接采用机器学习代替人工观察。

本文接下来将通过一道GCJ题，简单介绍一种实用的分类算法。

4.1 朴素贝叶斯

4.1.1 贝叶斯公式

设$P(A \mid B)$表示事件B已经确定发生的前提下，事件A发生的概率，$P(A \cap B)$表示事件AB都发生的概率。

定理4.1 (贝叶斯公式)**.**

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

Proof.

$$P(A \cap B) = P(A)P(B \mid A) = P(B)P(A \mid B)$$

移项后即可得贝叶斯公式。 □

4.1.2 朴素的判断

设S为待分类事件，A和B为可选择的分类。

若$P(A \mid S) > P(B \mid S)$，则认为$S$属于$A$类，否则属于$B$类。

当有更多的分类时，则选择概率最高的一项。

4.1.3 朴素的假定

设待分类事件$S = \{S_1, S_2, ..., S_m\}$，$S_i$为特征属性。

假定这些特征属性是相互独立的，可以得到

$$P(A \mid S) \approx \prod_{i=1}^{m} P(A \mid S_i)$$

通过比较这样近似计算的概率来进行分类。

$P(A \mid S_i)$的计算可以通过随机抽样或设计其他算法来完成。

4.2 例题六

例6 (GCJ Round 1A 2014 Proper Shuffle)**.**

有两种生成排列的方式，并给出120个长度为1000的排列，每个排列都是由这两个算法之一生成的，且这些排列都是独立生成的。

现在要求识别这些排列是由哪个算法生成的，对120个排列中的109个及以上输出正确解则判定为*AC*。

两种算法的流程如下：

Algorithm 1 GOOD

for $k = 0$ to $n - 1$ **do**
  $a_k = k$
end for
for $k = 0$ to $n - 1$ **do**
  $p = random\_int(k..n - 1)$
  $swap(a_k, a_p)$
end for

Algorithm 2 BAD

for $k = 0$ to $n - 1$ **do**
  $a_k = k$
end for
for $k = 0$ to $n - 1$ **do**
  $p = random\_int(0..n - 1)$
  $swap(a_k, a_p)$
end for

其中*GOOD*算法可以等概率地生成排列，而*BAD*算法则不行。

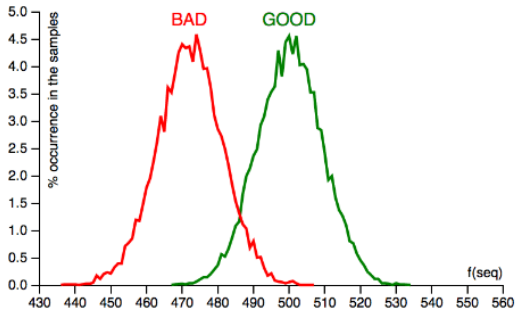
4.2.1 简单分析

对于$GOOD$算法，完成第k位的交换后，第k位上出现任何数的概率均为$\frac{1}{n}$，且之后的交换都不会影响这一位。

对与BAD算法，当决策第k位的交换时，$swap(a_k, a_p)$时$p < k$，这时原先在a_p的值就被交换到了更高的位上，并且此后会被交换到跟高的位上，这会对它出现在每位上的概率产生微妙的影响。

4.2.2 一种简单的特征判断

设S是输入的排列，$f(S)$为排列S中$S_i \le i$的位置数，GCJ的Contest Analysis中给出了以下统计数据：



通过设定阈值判断可以达到90%以上的正确率。

4.2.3 朴素贝叶斯分类

设S是输入的排列，我们想要求出$P(GOOD|S)$，那么根据定理 4.1 易得

$$\begin{aligned} P(GOOD \mid S) &= \frac{P(S \mid GOOD)P(GOOD)}{P(S)} \\ &= \frac{P(S \mid GOOD)P(GOOD)}{P(S \mid GOOD)P(GOOD) + P(S \mid BAD)P(BAD)} \end{aligned}$$

因为$P(GOOD) = P(BAD)$，所以式子可以简化

$$P(GOOD \mid S) = \frac{P(S \mid GOOD)}{P(S \mid GOOD) + P(S \mid BAD)}$$

同时也可以得到

$$P(BAD \mid S) = \frac{P(S \mid BAD)}{P(S \mid GOOD) + P(S \mid BAD)}$$

假如$P(GOOD \mid S) > P(BAD \mid S)$，那么$S$由$GOOD$算法生成的概率较大。容易发现$P(GOOD \mid S) > P(BAD \mid S)$的条件是$P(S \mid GOOD) > P(S \mid BAD)$。

由于$GOOD$算法能等概率生成所有排列，所以

$$P(S \mid GOOD) = \frac{1}{n!}$$

但是$P(S \mid BAD)$的计算则很难完成，于是选择近似化处理：

$$P(S \mid BAD) \approx \prod_{k=0}^{n-1} P(S_k \mid BAD)$$

为了使比较概率时更加公平，我们对能精确计算的$GOOD$算法也进行相同的近似化处理：

$$P(S \mid GOOD) \approx \frac{1}{n^n}$$

设$P_k[i][j]$为BAD算法完成了k次交换后$S_i = j$的概率，那么容易设计一个$O(n^3)$的DP计算$P_n[i][j]$的值，也就得到了所需的$P(S_k \mid BAD)$的值。

计算出$P(S \mid BAD)$后与$P(S \mid GOOD)$比较即可[3]。

5 带权均分问题

例7. 设S为一个由n个元素构成的多元集合*(multiset)*，$f(S)$为S集合中所有元素之和。将集合S划分为两个集合S_1、S_2，使$S_1 + S_2 = S$，求$|f(S_1) - f(S_2)|$的最小值，并求出划分方案。

5.1 简单举例

设$S = \{3, 1, 1, 2, 2, 1\}$，下面提供两种合法的划分方案使得$f(S_1) = f(S_2) = 5$，$|f(S_1) - f(S_2)| = 0$：

- $S_1 = \{1, 1, 1, 2\}$ $S_2 = \{2, 3\}$
- $S_1 = \{3, 1, 1\}$ $S_2 = \{2, 2, 1\}$

[3]直接计算$P(S \mid BAD)$涉及高精度运算，可以考虑计算$P(S \mid BAD)n^n$与1.0比较。由于经过了近似化处理，所以也可以用其他阈值代替1.0。

5.2 一种动态规划算法

可以设计一个动态规划，类似背包问题，设$f[i][j]$表示使用了前i个元素，$f(S_1) = j$的方案是否可行，时间复杂度为$O(nf(S))$。

当集合S中的元素都很小时，这确实是一个很优秀的算法，但这其实是一个伪多项式复杂度算法(pseudo-polynomial time algorithm)。

5.3 一种简单的贪心思想

当我们拿到这个问题时，很容易产生这样的想法：把元素读入后，依次把每个元素放入当前和较小的集合中。

为了使两个集合的大小逐渐逼近，所以把元素排成降序处理，时间复杂度$O(n \log n)$。

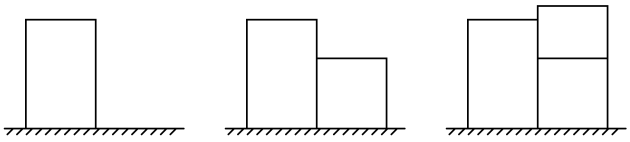
这个算法的思路十分自然，而且效果也不错，但是经过仔细分析后会发现是存在明显漏洞的。

5.4 差分算法

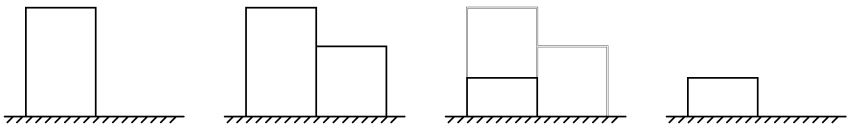
差分算法是对前一节中提到的贪心算法的改进，思路也很简洁，近似解效果有了很大的优化。

5.4.1 基本思想

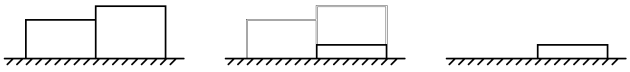
仔细考虑前一算法，该贪心算法的流程如下图：



考虑对算法进行修改，当我们把两个数X、Y放到不同集合时，对答案产生影响的部分只有两数的差的大小，相当于得到了一个新数$|X - Y|$，如图:



对于新得到的这个数，仍可以把它当作一个普通的数看待，继续进行原算法。



重新回顾前一节的贪心算法，考虑每一次决策时的状态。设新放入的元素为X，已完成决策的元素得到的差分值为Y。把新元素放入较小集合这个操作，相当于把X 和Y替换为$|X-Y|$，而我们设计这个算法的初衷是优先处理较大的元素，所以依次决策是不优的。

5.4.2 算法流程

根据差分思想，可以得出一个新的算法，需要用到堆和并查集[4]维护。

- 把集合S中的所有元素放入堆中，并查集中每个元素是一个独立集合。
- 从堆中取出最大的两个元素，设为X、Y，对应并查集中为集合S_X、S_Y。这时做出决策，把$|X-Y|$ 放入堆中，并查集中合并集合S_X、S_Y，连边时连接权值为1 的边。
- 当堆中只剩下一个元素时，进入下一步，否则重复前一步。
- 设并查集树结构的根在S_1集合，其它元素根据到根路径长度的奇偶性判断所在集合（偶$\rightarrow S_1$，奇$\rightarrow S_2$）。

[4]这里的并查集实现可以在路径压缩的时候维护路径长度的奇偶性，或者直接选用按秩合并保证$O(\log n)$树高。

5.4.3 近似效果

这个算法的时间复杂度为$O(n\log n)$，是非常高效的，所以主要需要关注它得出的解是否够优。

当n不过于小时，比如假设$n \geq 1000$，对于随机生成的数集$S(S_i \in [0, 2^{31}))$，该算法能以接近100% 的概率划分出集合S_1、S_2使得$|f(S_1) - f(S_2)| \leq 1$，也就是$f(S)$是奇数时$|f(S_1) - f(S_2)| = 1$，$f(S)$是偶数时$|f(S_1) - f(S_2)| = 0$。可以看出这个算法的正确率是相当高的。

在n较小的时候可以利用该算法设计估价函数，进行启发式搜索来获得更优的解。

5.5 例题八

例8 (POI2013 Polarization)**.**

给出一个n个点的树，把树上的每条边定向为单向边。若点u可以到达点v，则称(u, v)为合法点对。求出所有定向方案中合法点对数量的最小值和最大值。

数据范围：$n \leq 250000$

5.5.1 问题转化

树可以表示为二分图，所以最小值一定为$n - 1$。

对于最大值，最优解一定是存在一个点u使得剩下的点要么存在它到u的路径，要么存在u到它的路径。进一步分析易得u一定选取在树的重心位置。

以u为根划分出若干子树，剩下的问题就是把这些子树划分成两个集合使其大小的乘积最大。

5.5.2 传统最优化算法

注意到这里$f(S) = n$，所以说不同的元素种数只有$O(\sqrt{n})$种，转化为多重背包问题。

多重背包是经典问题，这里不详细介绍。

时间复杂度$O(n\sqrt{n})$。

5.5.3 近似化算法

假如直接使用差分算法计算解，能够通过80%的数据，主要原因在于当元素个数较少时，近似解的偏差率较大。

可以定一个阈值，当子树个数小于这个阈值时，采用暴力背包算法。

经过这样的拼接后将可以通过全部POI官方数据，时间复杂度$O(n\log n)$。

在传统最优化算法运行效率良好时并不提倡使用近似化算法，这里只是为了说明差分算法能够得出很优的近似解。

6 最小斯坦纳树问题

例9. 给出一张带权无向图$G=(V,E)$以及一个集合S，V表示图G的点集，E表示图G的边集，S是V的一个子集。求一个连通图$T=(V_T,E_T)$，满足$S\subseteq V_T\subseteq V$，$E_T\subseteq E$，$|E_T|=|V_T|-1$，并使得边集$E_T$的权值和最小。

6.1 一种最优化算法

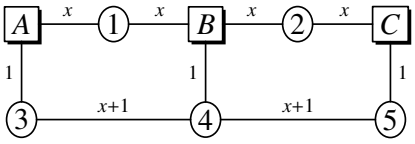
最小斯坦纳树有经典的状态压缩算法，转化为最短路问题模型，具体请参见姜碧野的2014集训队论文。

时间复杂度$O(|V|\,3^{|S|}+|E|\,2^{|S|})$，这是一个很优秀的算法，但是能适用的数据范围很小。

6.2 一种简单的贪心思想

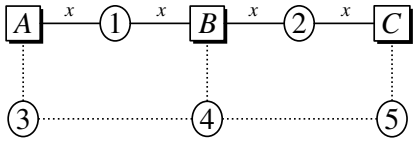
有一个简单的贪心：对原图求一棵最小生成树，再剔除最小生成树上多余的边，即在不影响点集S连通性的前提下尽可能地删边。

这个算法存在很大的漏洞，如下图：

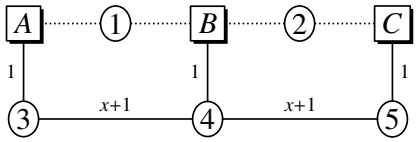


设$V=\{A,B,C,1,2,3,4,5\}$，$S=\{A,B,C\}$，$x\geq 2$，直接采用上述贪心得到的

解如下图：



容易发现最优解应为下图：

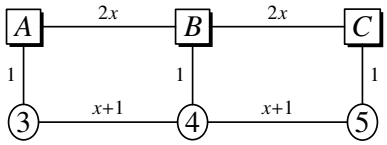


6.3 启发式算法

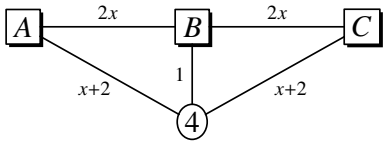
根据上面那个例子，可以对算法进行一些针对性的改进。

6.3.1 进一步分析

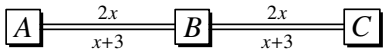
在这个例子中，假设忽视点1和点2可以得到下图：



对这张图做最小生成树已经可以得到最优解，但是为了使问题更一般化，考虑把剩下的非S集节点也从图上去除。

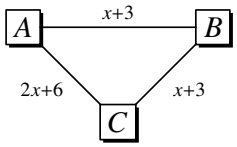


假如把点4也从图上去除，就得到下图：



容易发现一个问题，两条权为$x+3$的边有交，所以在最后做出的生成树中要除去重复统计的边。

为了将算法更一般化，可以构出一张完全图G'，图G'中只保留集合S中的点，边权为原图中的最短路长度。



6.3.2 算法流程

经过进一步整理后可以得到如下算法：

- 由原图$G = (V, E)$构造完全图$G_1 = (S, E_1)$，E_1边权为G中最短路长度。
- 得到G_1的最小生成树T_1（如果有多解则随意选择一个）。
- 构造G的一个子图$G_2 = (V_2, E_2)$，其中$V_2 = V$，E_2为T_1中的边在G中的路径（如果有多条最短路则随意选择一条）。
- 得到G_2的最小生成树T_2（如果有多解则随意选择一个）。
- 在T_2中去掉无用的边，使得所有叶子节点都是集合S中的点，得到斯坦纳树T。

分析算法每一步的时间复杂度：

- 构造完全图G_1需要完成$|S|$次单源最短路，复杂度$O(|S|\,|V|^2)$[5]。
- 求最小生成树T_1，完全图宜选用$Prim$算法，复杂度$O(|S|^2)$。
- 构造图G_2，每条边所对应的最短路可能会经过$O(|V|)$个点，但是所有路径边数相加是$O(|V|)$的。
- 求最小生成树T_2，复杂度$O(|V|^2)$或$O(|V|\log|V|)$。
- 删去无用边，复杂度$O(|V|)$。

可以看出这个算法的运行效率是十分高效的，而其近似化效果也十分优秀，下文将给出对其近似解的相对上界证明。

[5]可以选用高效的最短路算法进一步优化复杂度。

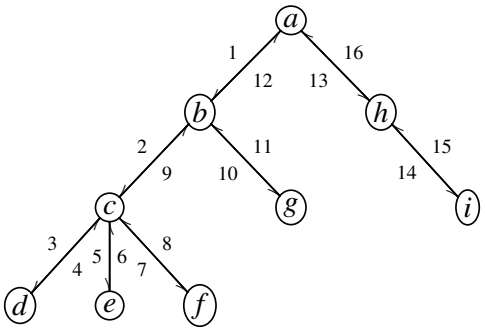
6.3.3 近似效果

设D_T为上述算法得到的斯坦纳树T的边权和，D_{MIN}为最小斯坦纳树T_{MIN}的边权和，下文将分析得出$\frac{D_T}{D_{MIN}}$的上界。

引理6.1. 对于一棵有$m(m \geq 1)$条边的树T，可以在树上找到一个环，$u_0, u_1, u_2, \ldots, u_{2m}$，其中$u_0 = u_{2m}$，每个点$u_i(0 \leq i \leq 2m)$都是$T$中的一个节点，点对$(u_{i-1}, u_i)(1 \leq i \leq 2m)$在$T$中有边，且这个环满足以下性质：

- T中的每条边在环中正好出现两次。
- T中的每个叶子节点在环中只出现一次（不考虑u_0）。
- 假设u_i和$u_j(0 < i < j)$是T中的两个叶子节点，且不存在叶子节点$u_k(i < k < j)$，那么$u_i, u_{i+1}, \ldots, u_j$是$T$上的一条简单路径。

Proof. 选定一个点作为根节点后，对树T进行欧拉遍历后得到的欧拉遍历序[6]满足上述性质。



□

定理6.1. 设$leaf$为最小斯坦纳树T_{MIN}中叶节点的个数，则

$$\frac{D_T}{D_{MIN}} \leq 2\left(1 - \frac{1}{leaf}\right) \leq 2\left(1 - \frac{1}{|S|}\right)$$

[6]把无向边视作两条方向相反的有向边后，以根为起点的欧拉回路。

Proof.

因为$leaf \le |S|$，所以显然$2\left(1-\frac{1}{leaf}\right) \le 2\left(1-\frac{1}{|S|}\right)$。

根据引理 6.1 ，可以在T_{MIN}中找出一个环L，设环L的边权和为D_L，则$D_L = 2D_{MIN}$。

在环L上选择最长的一条叶子节点间的简单路径删去后，得到一条路径P，设路径P的边权和为D_P，则易得$D_P \le \left(1-\frac{1}{leaf}\right)D_L$，即$D_P \le 2\left(1-\frac{1}{leaf}\right)D_{MIN}$。

由于T中两点间都选取了最短路径，所以$D_T \le D_P$，即$D_T \le 2\left(1-\frac{1}{leaf}\right)D_{MIN}$。

□

定理6.2. 设$leaf$为最小斯坦纳树T_{MIN}中叶节点的个数，在$leaf \ge 2$时，最坏情况下

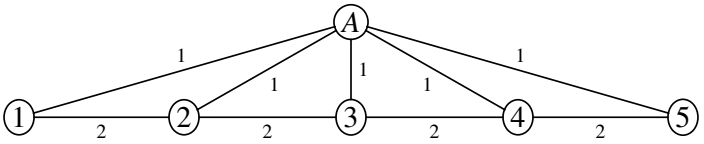
$$\frac{D_T}{D_{MIN}} = 2\left(1-\frac{1}{leaf}\right)$$

Proof.

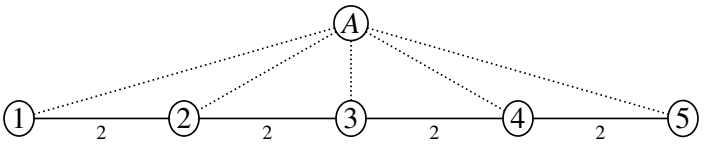
可以这样构造图$G=(V,E)$，$V=\{v_1,v_2,\ldots,v_{leaf+1}\}$，$S=\{v_1,v_2,\ldots,v_{leaf}\}$，边集定义如下：

$$d(v_i,v_j)=\begin{cases}2 & i+1=j \ \ and \ \ i<leaf\\ 1 & i\le leaf \ \ and \ \ j=leaf+1\\ \infty & otherwise\end{cases}$$

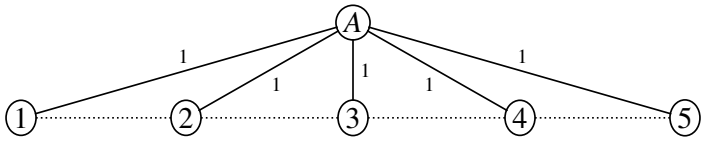
如下图，$V=\{A,1,2,3,4,5\}$，$S=\{1,2,3,4,5\}$：



在最坏情况下，T的构成会是这样：



然而，T_{MIN}的构成是这样：



对于这样构造的图，比值达到了上限

$$\frac{D_T}{D_{MIN}} = \frac{2(leaf-1)}{leaf} = 2\left(1-\frac{1}{leaf}\right)$$

□

6.4 例题十

例10 (2015集训队互测 Marketing network)**.**

给出图$G=(V,E)$，求前k小生成森林，要求特定点集S在生成森林中连通。

数据范围：$|V|,k\le 50$，$|E|\le 100$，$|S|\le 15$，图的边、边权以及点集S均随机生成。

6.4.1 最优化算法

对于第k优解问题，可以用A*算法求解。

可以用每条边是否选取来描述状态，搜索时确定下一条边是否选取。

选用斯坦纳树的状态压缩算法可以做到$h(s)=h^*(s)$，每次选择最优状态扩展，由于要求的是生成森林，所以每$O(n)$次有效决策就能找到一个当前最优解，求前k优解只需要进行$O(kn)$次估价计算。

这个算法能够保证正确性和复杂度，但效率较低，并不能胜任该题数据范围。

6.4.2 近似化算法

继续沿用A*思想，选用前文的启发式算法作为估价，但不幸的是$h(s)\ge h^*(s)$，虽然能估计出下界，但是不能有效剪枝。

考虑进行近似化处理，选定一个常数c，把$ch(s)$作为$h^*(s)$的下界参考，剪去估价不优的状态，通过调整常数c可以平衡运行效率和正确性。

7 感谢

感谢计算机协会提供学习和交流的平台。

感谢绍兴一中的陈合力老师、董烨华老师、邵红祥老师、游光辉老师多年来给予的关心和指导。

感谢国家集训队教练余林韵和陈许旻的指导。

感谢清华大学的俞鼎力、董宏华、何奇正学长对我的帮助。

感谢绍兴一中的张恒捷、王鉴浩、贾越凯同学对我的帮助和启发。

感谢其他对我有过帮助和启发的老师和同学。

参考文献

[1] Wikipedia, Heuristic (computer science).

[2] Wikipedia, Partition problem.

[3] Wikipedia, Steiner tree problem.

[4] L Kou, G Markowsky, L Berman, “A fast algorithm for Steiner trees”, Acta Informatica.

[5] 姜碧野, 迭代求解的利器——SPFA 算法的优化与应用, 2009集训队论文。

[6] 周而进, 浅谈估价函数在信息学竞赛中的应用, 2009集训队论文。

[7] 俞鼎力, 寻找第k优解的几种方法, 2014集训队论文。

浅谈字符串匹配的几种方法

绍兴市第一中学 王鉴浩

摘 要

字符串匹配问题是信息学竞赛中的经典问题。本文对于这类问题的解法进行了分类归纳。本文先简要总结了一些常用的结构和算法，然后着重介绍了一个新颖的字符串匹配结构 *Border Tree*，并推出了一类字符串匹配问题的通用解法。本文中每种算法和技巧都有例题，便于读者理解与分析。

1 引言

字符串匹配问题经常应用于文本编辑中。现在处于大数据时代的我们，在许多领域中需要解决一些操作多样、数据量大的问题。信息学竞赛中字符串匹配问题便是这类问题的模型，本文将会对这类问题的一些解法进行介绍。

在本文的第二节中，为了之后描述的方便，将会对一些字符串的定义进行回顾。

然后本文会对一些结构和算法进行简要总结。本文中把这些结构和算法分为了两类：前缀类解法和后缀类解法。

在本文第三节中作者将介绍一些前缀类结构和算法：*Trie*，*KMP*，*AC* 自动机。第四节中会介绍一些后缀类结构：后缀树、后缀数组、后缀仙人掌和后缀自动机。第五节中作者将会介绍一个新颖的结构：*Border Tree*，并进行详细分析。第六节中作者将会基于第五节中的 *Border Tree* 推出一类字符串匹配问题的通用解法。

其中第三节和第四节中的内容比较普及，有些算法以前的国家集训队论文已有介绍，所以作者只对其进行简要总结，并对于一些经典的问题进行分析讨论。

作者会对于第五节和第六节花大量篇幅介绍。其中 *Border Tree* 是比较新颖的结构，作者会提出一类应用抛砖引玉，有兴趣的读者可以继续研究。

2 定义

为了之后描述的方便，先给出如下定义：

定义 2.1 (子串，前缀，后缀，*Border*，*LBorder*)**.**

设一个长度为 n 的字符串 $s = s_1s_2s_3...s_n$。

对于 $1 \le i \le j \le n$，称 $s_is_{i+1}s_{i+2}...s_j$ 为 s 的一个子串，记成 $s[i:j]$。特别地，如果 $i > j$，则 $s[i:j]$ 表示空串，记作 $\emptyset$，空串也是 s 的一个子串。

对于 $1 \le i \le n$，称 $s[i:n]$ 为 s 的一个后缀，记成 $suf[i]$；称 $s[1:i]$ 为 s 的一个前缀，记成 $pre[i]$。

对于 $1 \le i < n$，如果 $s[1:i] = s[n-i+1:n]$，则称 $s[1:i]$ 为 s 的一个 *Border*，特别地，空串 $\emptyset$ 也称为是一个 *Border*；在 s 的 *Border* 中，称最长的 *Border* 为 s 的 *LBorder*。

我们又称 $pre[i]$ 的 *LBorder* 为 $LBorder_i$。

定义 2.2 (周期串，基，周期的长度)**.**

设一个长度为 n 的字符串 $s = s_1s_2s_3...s_n$。

如果串 s 的 *LBorder* 长度为 L，把 $s[L+1:n]$ 称作是 s 的基。

如果 $2L \ge n$，那么 s 串被称为是周期串。

如果 s 串是周期串，那么称 s 串的基的长度是周期的长度。

定义 2.3 (重复串)**.**

如果某个串 t 是由一个串 s 重复 k $(k > 1)$ 次后得到的，那么称 t 是重复串，记为 s^k。

显然，重复串是一类特殊的周期串。

3 前缀类解法

在信息学竞赛的字符串匹配解法中，有一些结构和算法是根据高效地利用前缀的信息来实现对字符串快速匹配。在本文中，我们称这类解法为前缀类解法。在本节中，本文会介绍 3 种前缀类解法及其变种。

3.1 Trie

Tire 是一棵单词查找树，在下文中我们称之为字母树。

字母树是能支持对多个模版串进行查找和匹配的数据结构。关于字母树的定义和构造方法请详见朱泽园 2004 年的集训队论文或董华星 2009 年的集训队论文。

字母树是一棵有根树，每条树边表示一个字符，把从根到点 i 路径上的边按遍历顺序排列，能构成一个串 s_i。所以，我们称字母树上每个点都表示一个串 s_i，其中根表示的串为 $\emptyset$。

通常地，对于多个模版串匹配，会把每个模版串加进字母树中，根据字母树的定义，可以发现每个模版串的前缀都能被字母树中一个点表示。那么，就可以利用树的性质高效地解决一些字符串匹配问题了。比如，快速地计算两个模版串的最长公共前缀等等。

接下来，让我们通过一个例子更好地理解字母树。

3.1.1 例题

例 1 (USACO 2012 Dec gold 2 *First*)**.**

给出 n 个由小写字母组成的字符串，第 i 个串表示为 s_i。

现在，26 个小写字母的字典顺序任意，问每个串是否能在某种字典顺序中成为字典序最小的串。

字符串保证不重复。

数据范围：$1 \le n \le 30000, 1 \le$ 字符串总长度 ≤ 300000

时限：$1s$

对于此题，可以先把这 n 个串建字母树。对于第 i 个串，设字母树中点 j 表示的串为 s_i。如果要使得第 i 个串在某种字典顺序中字典序最小，那么首先要满足在这 n 个串中没有一个别的串为 s_i 的前缀，其次要求在根到点 j 的路径上，每一个字符都要比同层的其他字符的字典顺序小。

那么对于每个串，可以把这些限制拿出来，然后判断这些限制是否会矛盾。具体实现的时候，可以把限制看作是有向边，然后判断这张有向图是否会有环。

此题的时间复杂度为 $O(26^2n + 26 \times$ 字符串总长度$)$。

3.2 KMP

信息学竞赛中会出现模式串和主串匹配的问题。在暴力字符串匹配过程中，会从第一位开始匹配，如果相等则匹配下一个字符，直到出现不相等的情况。此时人们会简单的丢弃前面的匹配信息，然后以主串的下一位和模式串的第一位开始重新匹配，循环进行，直到主串结束，或者出现匹配的情况。这种简单地丢弃前面的匹配信息的做法，造成了极大的浪费和低下的匹配效率。

可以使用 *KMP* 算法来加速模式串与主串匹配的速度，*KMP* 算法具体请参见朱泽园 2004 年的集训队论文。

下面给出模式串每一位失配的数组的定义：

定义 3.1 (失配数组)**.**

设一个长度为 n 的模式串 $s = s_1s_2s_3...s_n$。

令 $next[i] = |LBorder_i|$，并称 $next$ 数组为 s 的失配数组。

KMP 算法的核心思想就是先在模式串中预处理出失配数组。然后对于主串根据模式串的失配数组，计算出以每一位结束最多能与模式串匹配的长度。其中 *KMP* 算法的时间复杂度为线性。

3.2.1 KMP 自动机

当建出 *next* 数组后，可以发现如果把 $next[i]$ 向 i 连边后，整张图就构成了一颗 $n+1$ 个点的树。其中，0 号点为根，i 号点表示为 $pre[i]$。从根到点 i 路径上的点都为 $a[1:i]$ 的 *Border*。于是，就可以根据这棵树建出每个点的转移数组，表示每个串后加一个字符形成的新串最多能匹配模式串几位。在串匹配时，就可以根据转移数组进行快速计算了。我们把上述结构称为是 *KMP* 自动机。

KMP 自动机的具体树的形态可以见图 1 (图片在第 51 页)。

3.2.2 例题

例 2 (NOI 2014 动物园)**.**

有 T 组数据，每组数据给出一个长度为 n 的模式串 $s = s_1s_2s_3...s_n$。

在 $s[1:i]$ 的非空 *Border* 中，满足 $2|Border| \le i$ 的 *Border* 数量记为 $num[i]$。

要求求出 *num* 数组。

数据范围：$1 \le T \le 5, 1 \le n \le 1000000$

时间限制：1s

一种解决此题方法就是，先建出 *next* 数组后，再根据 *next* 数组，建出 KMP 自动机中的树。对于每个点，我们只需要知道根到这个点的路径上有多少标号不超过当前点标号一半的点。由于标号是单调的，所以直接二分就好了。又由于建 *next* 数组的时间是线性的。那么此题总共的时间复杂度为 $O(Tn\log n)$。

3.3 AC 自动机

AC 自动机可以看成是通过 $Tire$ 和 KMP 自动机相互结合得到的新结构。构建 *AC* 自动机是对于多个模式串先建 $Tire$，然后像 KMP 自动机一样建出 *next* 数组来构建树和转移数组。不过这个 *next* 数组是针对多个串的，不同于 KMP 自动机的 *next* 数组。我们称在 *AC* 自动机中，像 KMP 自动机那样用 *next* 数组构成的树叫 $fail$ 树。可以发现在 $fail$ 树中，每个点 i 都能表示一个串 v_i。而且对于点 i 子树中的点，每个点表示的串都有 v_i 这个前缀。于是，根据 $fail$ 树我们就可以构出每个点的转移数组。在串匹配时，就可以通过转移数组进行快速匹配。

AC 自动机具体性质和构造请参见朱泽园 2004 年的集训队论文。

如此，*AC* 自动机就结合了 $Tire$ 和 KMP 自动机的性质，能解决更复杂的多模式串匹配的问题。

接下来，本文给出一个例题让读者更好地理解 *AC* 自动机。

3.3.1 例题

例 3 (Coci 2015 *Divljak*)**.**

一开始给出 n 个字符串：$s_1, s_2, s_3, ..., s_n$。有一个字符串集合 T，T 一开始是空的。

有 q 个操作，有两种操作：

1. 向字符串集合 T 中添加一个字符串 p。
2. 选择一个串 s_x，问集合 T 中有多少个串，使得 s_x 是其子串。

设 sum_1 为 T 中的字符总数，$sum_2 = \sum_{i=1}^{n} |s_i|$。

字符为小写英文字母。

数据范围：$1 \le n, q \le 100000, 1 \le sum_1, sum_2 \le 2000000$

时限：$1s$

首先对于 $s_1, s_2, s_3, ..., s_n$，可以先建出 AC 自动机来得到 $fail$ 树和转移数组。设在 $fail$ 树中点 d_i 表示的串为 s_i。

接下来，对操作进行一些转化。对于操作一，可以认为是在 AC 自动机中，把那些能表示为 p 串的前缀的点染成一种颜色。对于操作二，可以认为答案就是在 $fail$ 树中点 d_i 的子树的颜色种数。对于这个转化，读者可以通过 AC 自动机的性质来自行理解。

实际实现中，可通过 $fail$ 树的 dfs 序统计答案。一个基本的思想是：由于树上的每个子树对应到 dfs 序都是一个区间，那么每次染色之后，要使得子树中有这个颜色的点，其 dfs 序的区间权值和会加一。那么对于操作二，可以把直接点 d_i 的子树对应到 dfs 序上得到一个区间，然后用树状数组统计区间权值和来得到答案。

对于操作一，根据上述的基本思想，具体的实现是：把那些需要染色的点拿出来，显然这个点数是 $O(|p|)$ 级别的。对于这些点，先把它们去重，然后根据 dfs 序进行排序；接下来，把每个点的权值加一，并在相邻两个点的树上最近公共祖先 (lca) 处把权值减一。经过上述操作，就可以满足：在树上的每个子树中，如果有染色的点的话，只有 dfs 序最小的那个点会有 1 的染色贡献。读者可以结合虚树的形态来更好地理解上述实现。

于是此题就可以在 $O(26sum_1 + sum_2 \log sum_1 + sum_2 \log sum_2)$ 的时间复杂度内解决。

4 后缀类解法

在信息学竞赛的字符串匹配解法中，后缀类解法有着重要的地位。与前缀类解法相比较，后缀类解法显得更加高端和复杂，能解决更加复杂的问题。在本节中，本文将介绍 4 种后缀类数据结构：后缀数组、后缀树、后缀仙人掌和后缀自动机。

4.1 后缀数组

后缀数组是字符串处理中非常优秀的数据结构，是一种处理字符串的有力

工具，在不同类型的字符串问题中有广泛的应用。

后缀数组的核心思想是对于一个模式串 b，把 b 串中的每个后缀按照字典序排序，用 sa 数组记录。其中 sa_i 表示：在 b 串的所有后缀中字典序第 i 小的后缀的位置。相对地，还能得到一个 $rank$ 数组，$rank_i$ 表示 b 串中的 $suf[i]$ 在 sa 数组中的位置。得到 sa 数组和 $rank$ 数组后，还能通过后缀数组的性质用线性复杂度得到 $height$ 数组。其中，$height_i$ 表示 sa_{i-1} 和 sa_i 这两个后缀的最长公共前缀 (LCP)的长度。运用 $height$ 数组，可更高效地计算一些问题。比如，求 b 串两个子串的 LCP 等等。

对于构建后缀数组，可以运用倍增算法或 $DC3$ 算法。但由于篇幅有限，本文就不详细介绍后缀数组具体性质和构造了，有兴趣的读者可以参见许智磊 2004 年的集训队论文或罗穗骞 2009 年的集训队论文。

接下来，本文将对于关于后缀数组的 3 个经典问题进行分析。

4.1.1 问题一

例 4 (最长公共前缀)**.**

给出一个长度为 m 的串 b。

有 q 个询问，每个询问在 b 串上给出两个子串 (b', b'')，问这两个子串的最长公共前缀的长度。

对于这个问题，可以先用倍增算法在 $O(m \log m)$ 的时间复杂度内把串 b 做后缀数组，得到数组 sa，$rank$ 和 $height$。然后对于每次询问两个子串的 LCP，可以把这两个子串加长成为两个后缀。先求出这两个后缀的 LCP，然后再对 $|b'|$ 和 $|b''|$ 取 min 就可以得到答案了。而对于询问两个后缀的 LCP，可以把这两个后缀对应到 sa 数组上，根据 $height$ 数组的性质，把问题转化为在 $height$ 数组上的 rmq 问题。对于在 $height$ 数组上的 rmq 问题，我们可以先在时间复杂度为 $O(m \log m)$ 内建出 ST 表，然后对于每个询问，在 $O(1)$ 的时间复杂度内计算答案。

对此问题，上述解法的时间复杂度是 $O(m \log m + q)$。

4.1.2 问题二

例 5 (子串拼合问题)**.**

给出一个长度为 m 的串 b。

有 q 个询问，每个询问在 b 串上选择两个子串 (b', b'')，令 $s = b'b''$。询问串 s 是否是 b 串的子串。如果是子串的话，给出一个合法的位置。

对此问题，可以先用倍增算法在 $O(m \log m)$ 的时间复杂度内把串 b 做后缀数组，得到数组 *sa*，*rank* 和 *height*。然后可以发现，如果串 s 是 b 串的子串的话，那么串 s 就可以表示为 b 串中某一个后缀的前缀。于是，就先把所有以 b' 为前缀的后缀都找出来。根据 *sa* 数组的定义，可以发现这些后缀对应到 *sa* 数组上是连续的一段，这一段可以通过二分得到。现在要在这个区间中找到一个以 s 串为前缀的后缀。由于 *sa* 数组是把那些后缀按照字典序排序，那么又可以根据字典序，在这个区间中继续二分，寻找以 s 串为前缀的后缀。在二分的过程中，可以结合 *rank* 数组和 LCP 来进行快速判断。由于在问题一中，我们已经可以支持在 $O(1)$ 的时间复杂度内进行 LCP 的计算，那么二分的时间复杂度还是 $O(\log m)$。

上述解法的时间复杂度为 $O(m \log m + q \log m)$。

4.1.3 问题三

例 6 (区间最大前缀问题)**.**

给出一个长度为 m 的串 b。

有 q 个询问，每个询问在 b 串上选择给一个区间 $[l, r]$，计算：

$$\max_{i=l}^{r} LCP(suf[i], b)$$

对此问题，可以先用倍增算法在 $O(m \log m)$ 的时间复杂度内把串 b 做后缀数组，得到数组 *sa*，*rank* 和 *height*。对于此题，我们可以先对每个 i 计算 $LCP(suf[i], b)$。于是，该题就转化成了 *rmq* 问题了。与问题一同理，可以先在时间复杂度为 $O(m \log m)$ 内建出 ST 表，然后对每个询问，在 $O(1)$ 的时间复杂度内计算答案。那么对于这个问题，上述解法的时间复杂度是 $O(m \log m + q)$。

此外，此题还有一个拓展：我们可以在计算最大值的同时统计最大值的个数。但是如果要统计最大值的个数，就不能使用 ST 表了。我们可以通过倍增来计算答案，在 $O(\log m)$ 的时间复杂度内完成对单次询问的计算。这个解法的时间复杂度是 $O(m \log m + q \log m)$。

4.2 后缀树

后缀树是处理字符串的经典数据结构。后缀树的本质是字母树。对于一个长度为 m 的字符串 b，如果把它的所有后缀建字母树，就得到了 b 串的后缀树了。但由于这样建字母树的话，字母树中的点会达到 $O(m^2)$ 级别，所以需要把这棵后缀树的路径进行压缩。原本的字母树一条边只表示一个字符，现在让每条边表示一个字符串。可以发现如果有 m 个后缀要加入后缀树中的话，那么这个字母树的边数就变成 $O(m)$ 级别了。简单来说，就是先在串 b 后加一个终止符，建出一棵点数是 $O(m^2)$ 的字母树后，把那些只有一个儿子的点和它的儿子合并，我们就得到一棵点数为 $O(m)$ 级别的后缀树了。

关于后缀树具体的性质分析和构造方法请参见朱泽园 2004 年的集训队论文。

当构建出后缀树之后，可以发现在上一节中的 sa 数组其实就是后缀树的 dfs 遍历数组。于是，就可以使用后缀树建后缀数组了，那么显然地，所有后缀数组能做的问题，后缀树都是可以支持的。而且在下一节中，本文就会介绍一种用后缀数组建后缀树的方法。由于这两个数据结构是可以互通的，在本节中，本文就不对后缀树进行举例分析了，有兴趣的读者可以根据后缀数组的经典问题自行理解分析。

4.3 后缀仙人掌

后缀仙人掌其实和后缀数组一样都是基于后缀树的。且后缀仙人掌和后缀树类似的，是一个树状结构。在后缀树中，我们是把那些只有一个儿子的点和它的儿子合并，但在后缀仙人掌中，我们是把每个非叶子结点都与一个儿子合并，所形成的连接体称为树枝。

对于构建后缀仙人掌，可以先建出后缀数组，然后根据 sa 数组和 $height$ 数组的定义，在 sa 数组升序枚举的同时维护一个 $height$ 数组的递减单调栈。对于每次枚举到的位置 i，把 sa_i 和目前栈顶元素连边。

由于篇幅有限，上述构造的正确性本文在这里就不再详述。关于后缀仙人掌的具体性质请参见王悦同，徐毅和徐子涵 2014 年的冬令营营员交流。

那么，根据后缀数组，即可得到后缀仙人掌。接下来，思考后缀仙人掌的定义，可以发现，如果把后缀仙人掌的每个树枝根据连边位置断开，就可得到后缀树。所以，只需要在后缀仙人掌连边的同时记录连边的位置，就可以根据

后缀数组得到后缀树了。而且这个根据后缀数组通过后缀仙人掌构建后缀树的方法的时间复杂度是线性的。

4.4 后缀自动机

后缀自动机的树是一个模式串逆序的后缀树。由于后缀树是字母树，所以后缀自动机也支持对多个模式串进行维护。于是，可以得到构建一个模式串后缀树的另一种方法：可以先建出后缀自动机，然后通过后缀自动机建后缀树。我们可以通过增量法构建后缀自动机。由于篇幅有限，关于后缀自动机的构造和性质分析请参见陈立杰 2012 年的冬令营营员交流。

接下来，本文给出一个例题让读者更好地理解后缀自动机。

4.4.1 例题

例 7 (ZJOI 2015 诸神眷顾的幻想乡)**.**

给一棵 n 个点的树。每个点上有一个字符。

每次选择树上两个点 x, y，把点 x 到点 y 路径上的字符按遍历顺序排列得到一个字符串。对树上每一对点都计算出字符串。问这 n^2 个字符串中有多少个不同的字符串。

树的叶子个数不超过 20 个，字符集大小不超过 10。

数据范围：$1 \le n \le 100000$

时限：$1s$

考虑暴力的做法。由于只有 20 个叶子，可以把每对叶子之间的字符串拿出来，可以发现，叶子之间的字符串的每个子串都能对应一条路径，而且每条路径都至少被一个子串对应。所以，可以把这 400 个字符串建后缀自动机，然后在后缀自动机上统计不同的字符串个数。由于每个字符串长度是 $O(n)$ 级别的，那么上述做法的时间复杂度为 $O(20^2 n \times 10)$。

更优地，可以考虑每个叶子当起点时的情况。把这个叶子当根的话，这个叶子到其他所有叶子之间的字符串我们可以理解为把整棵树遍历了一遍。那么，每个叶子当根得到的总字符串长度是 $O(n)$ 级别的。此题的时间复杂度为 $O(20n \times 10)$。

5 Border Tree

在本节中，本文将会介绍 *Border Tree* 这个新颖的数据结构。这个数据结构基于 *KMP* 自动机，能高效地处理一类单模版串匹配问题。

设模版串是长度为 m 的串 b。

如果把串 b 做 *KMP* 自动机，那么 *KMP* 自动机中树的深度是可以到达 $O(m)$ 级别的。*Border Tree* 是 *KMP* 自动机中树的变种，它能通过合并一些同样类型的前缀，使得把 *KMP* 自动机中树的深度降到 $O(\log m)$。也就是每个点表示的并非是一个前缀而是一组前缀了。这一组前缀称为一个 *Border Group*。

Border Tree 具体形态见图 2 (图片在第 52 页)。

5.1 Border Group

考虑 $b[1:i]$ $(1 \le i \le n)$ 的 *Border*。*Border* 的数量是可以达到 $O(i)$ 级别的，例如：$b[1:i] = \underbrace{111...1}_{i\text{ 个 }1}$。

我们需要压缩这些信息。基本的思想是我们把这些 *Border* 分类，把 $b[1:i]$ 的 *Border* 的种类数控制在 $O(\log i)$ 的级别。

定义 5.1 (*Border Group*)**.**

$b[1:i]$ 的 *Border* 可以分类得到序列 g：

$$g = \left[B_1, B_2, B_3, ..., B_{|g|}\right] \ (1 \le |g| \le O(\log i))$$

把 $b[1:i]$ 的 *Border* 按照长度递减排序，令 B_i 是保存连续的一段 *Border* 的序列。并且要满足：在 B_i $(1 \le i < |g|)$ 内的 *Border* 要比 B_{i+1} 内的 *Border* 长。

其中，B_i 内的 *Border* 序列是：

$$\left[\pi'_i\pi_i^{k_i}, ..., \pi'_i\pi_i^3, \pi'_i\pi_i^2\right] \ (k_i \ge 2)$$

或是：

$$\left[\pi'_i\pi_i^{k_i}, ..., \pi'_i\pi_i^3, \pi'_i\pi_i^2, \pi'_i\pi_i\right] \ (k_i \ge 1)$$

满足：π'_i 是 $b[1:i]$ 的某一个前缀，π_i 是 $b[1:i]$ 的基。

那么则称 B_i 是 *Border Group*，称 g 为 *BG* 序列。

下面给出一个例子来更好地理解 *Border Group*。

例 8.

设串 b = bababbababcbababbabab。

那么可以发现有 4 个 *Border*：[bababbabab, babab, bab, b]。

其中可以把这 4 个 *Border* 分成 3 类：[bababbabab], [babab, bab], [b]。

其中第一类为：π = babab, π' = $\emptyset$，第二类为：π = ab, π' = b，第三类为：π = b, π' = $\emptyset$。

现在来构造 *BG* 序列 g，其主要思想是：用 *Border Group* 把所有的 *Border* 分配进不同的序列中。每个 *Border Group* 中的 *Border* 含有相同的 π' 和 π。具体方法如下：

1. 令 $\pi'\pi^k = LBorder_i$，把序列 $\left[\pi'\pi^k, ..., \pi'\pi^3, \pi'\pi^2\right]$ 加入 B_1。
2. 令 $s = \pi'\pi$。若 s 不是周期串，那么把 s 加入 B_1，并且把 s 的 *LBorder* 作为 B_2 的第一个元素，否则把 s 作为 B_2 的第一个元素。
3. 当 π' 和 π 的 *LBorder* 同时为 $\emptyset$ 的时候结束构造。

可以发现这样构造可以保证 $1 \le |g| \le O(\log m)$：

Proof.

在第 2 步中，如果 s 是周期串，那么 s 是下一个 *Border Group* 的第一个元素。根据定义，我们可以得到 $|\pi'| < |\pi|$，于是 $|s| \le \frac{2}{k+1}\left|\pi'\pi^k\right|$ $(k \ge 2)$。由于当 $k = 2$ 时，$\frac{2}{k+1}\left|\pi'\pi^k\right|$ 最大，所以对于任意的 k，都有 $|s| \le \frac{2}{3}\left|\pi'\pi^k\right|$。

否则 s 不是周期串，那么 $LBorder_{|s|}$ 是下一个 *Border Group* 的第一个元素。根据 *LBorder* 的定义，$2\left|LBorder_{|s|}\right| < |s|$，又因为 $|s| = |\pi'\pi| \le \left|\pi'\pi^k\right|$，所以 $2\left|LBorder_{|s|}\right| < \left|\pi'\pi^k\right|$。

于是 $\left|\pi'_{i+1}\pi_{i+1}^{k_{i+1}}\right| \le \frac{2}{3}\left|\pi'_i\pi_i^{k_i}\right|$，所以这样构造可以保证 $1 \le |g| \le O(\log m)$。 □

5.2 构建 Border Tree

对于构建 *Border Tree*，首先要构建出 *KMP* 自动机，然后把自动机中的树转化成 *Border Tree*。

构建 *KMP* 自动机请参见本文的第三节。

对于 *KMP* 自动机的每个点 i，定义一个三元组 (x_i, y_i, z_i)：

1. 如果 $b[1:i]$ 是周期串，即 $b[1:i] = \pi'\pi^k$ $(k > 1)$，那么 $(x_i, y_i, z_i) = (|\pi'|, |\pi|, k)$。
2. 如果 $b[1:i]$ 不是周期串，即 $b[1:i] = \pi'\pi$，那么 $(x_i, y_i, z_i) = (|\pi'|, |\pi|, 1)$。
3. 如果 $b[1:i]$ 不是周期串，但 i 有一个儿子 j，满足 $z_j = 2$，那么 $(x_i, y_i, z_i) = (x_j, y_j, 1)$。

在情况三中，有如下定理，保证 i 最多只会有一个儿子 j，满足 $z_j = 2$：

定理 5.1. *每个非周期串 $s = b[1:i]$ 一定不会有两个及以上的儿子是周期串。*

Proof.

我们使用反证法。

假设 s 有两个及以上的儿子是周期串，则 s 可以根据它的某两个周期串儿子，表示成：$s = \pi'_1\pi_1 = \pi'_2\pi_2$，满足：$|\pi_1| < |\pi_2|, |\pi_2| \bmod |\pi_1| \neq 0$。

因为当 $|\pi_2| \bmod |\pi_1| = 0$ 时，s 一定能表示为 $\pi'_2\pi_1^x$ $(x > 1)$ 的形式，于是 s 就是周期串了，和我们的定理矛盾。

设 $d = \gcd(|\pi_1|, |\pi_2|)$，那么 s 一定存在一种表示方式：$s = \pi'_3\pi_3$，其中 $|\pi_3| = d$。

那么将导致 $|\pi_2| \bmod |\pi_3| = 0$，于是结论和我们的条件矛盾，所以假设不成立，原命题正确。 □

计算 (x_i, y_i, z_i) 是可以经过两次遍历做到线性复杂度的：

1. 第一次遍历时，对满足情况一和情况二的点，根据 *Border Group* 中的定义，令：

$$x_i = |\pi| = i - |LBorder_i|$$
$$y_i = |\pi'| = i \bmod |\pi|$$
$$z_i = \lfloor \frac{i}{|\pi|} \rfloor$$

2. 第二次遍历时，对满足情况三的点修改 (x_i, y_i, z_i)。

其中，情况三存在的意义在于维护 *Border Group* 中的 $\pi'\pi$。下面给出一个例子来理解情况三。

例 9.

设串 b = abaaba。

本来 $(x_3, y_3, z_3) = (1, 2, 1)$，但由于 $(x_6, y_6, z_6) = (0, 3, 2)$，于是 (x_3, y_3, z_3) 被修改成了 $(0, 3, 1)$。

于是就把 (x_i, y_i, z_i) 维护好了。

现在已经给了 *KMP* 自动机的每个点一个三元组。接下来，我们要通过修改自动机中的树来构造 *Border Tree*。

对于 x_i, y_i 都相同的点，根据 *Border Group* 的定义，它们属于同一类，就只保留 z_i 最小的那个点，并在那个点上记录这些点的信息。比如，如果一类点中有能表示为 $\pi'\pi$ 的点，那么保留这个点，否则保留能表示为 $\pi'\pi^2$ 的点。

最后再经过一次遍历，就可以把 *Border Tree* 构出来了。通过图 2 (图片在第 52 页)，我们可以更好地理解 *Border Tree* 。

5.3 分析性质

对于 *Border Tree* 中每个点 i 都对应了一个 *Border Group*。

这个 *Border Group* 可以表示为：

$$\left[\pi'_i\pi_i^{l_i}, \pi'_i\pi_i^{l_i+1}, \pi'_i\pi_i^{l_i+2}, ..., \pi'_i\pi_i^{r_i}\right]$$

但实际上，对于点 i，根到其路径上每个点对应的 *Border Group*，并不能准确地表示 $b[1:i]$ 的 *Border*。因为已经把边缩起来了。

例如图 1 中，$b[1:7]$ 并非是 $b[1:13]$ 的 *Border*，但在图 2 的 *Border Tree* 中，$b[1:7]$ 是被分在了点 3 的类型中。而点 3 是点 13 的父亲，处于根到点 13 的路径中。

设 $fa[i]$ 表示在 *Border Tree* 中点 i 的父亲。在 $fa[i]$ 这个 *Border Group* 中，$a[1:i]$ 的 *Border* 表示为：

$$\left[\pi_i'\pi_i^{l_i}, \pi_i'\pi_i^{l_i+1}, \pi_i'\pi_i^{l_i+2}, ..., \pi_i'\pi_i^{x}\right]$$

其中 $x = \min(z_{next[i]}, r_i)$。($z_{next[i]}$ 指的是在 *KMP* 自动机中，点 i 的父亲 $next[i]$ 所对应的三元组中的 z)

所以当需要查找 $b[1:i]$ 所有的 *Border* 时，需要对从根到 i 的路径上的点实时维护 x 值。

5.4 例子

设串 b = bababababcbabab。串 b 的长度是 13。下图为串 b 的 *KMP* 自动机中的树和 *Border Tree* 的形态。本来自动机中的树有 14 个点，深度为 5，经过把 *Border* 分类后建成 *Border Tree* 后变成有 10 个点，深度为 3 的树了。构造 *Border Tree* 时，图 1 中方形的点会缩起来，读者可以通过图 2 来理解构造。

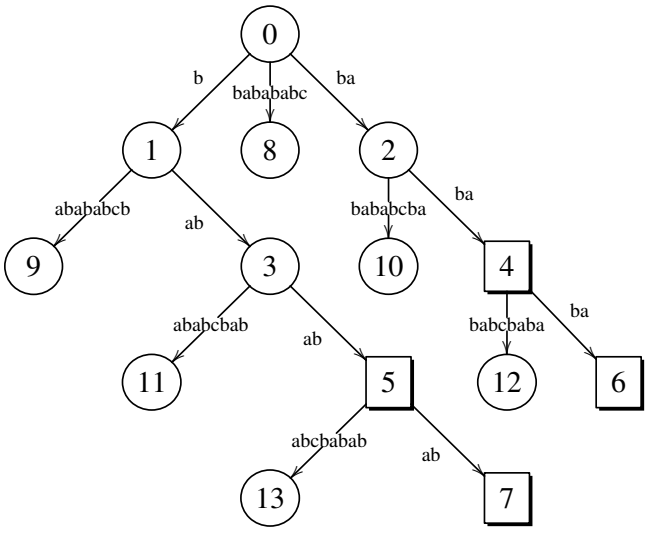


图1：*KMP* 自动机

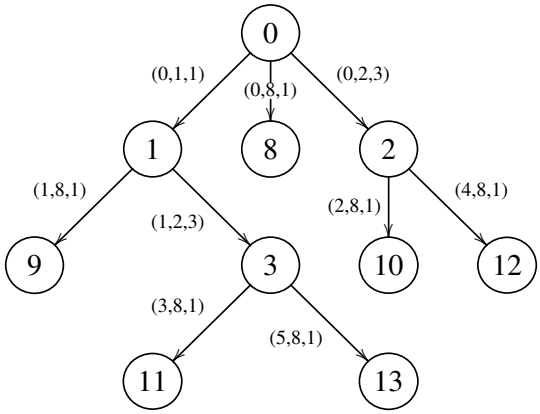


图2：*Border Tree*

读者可以根据下表来更好地对 *KMP* 自动机和 *Border Tree* 进行比较。

KMP 自动机和 *Border Tree* 中各个点的状态

点编号/	*KMP* 自动机中的 (x_i, y_i, z_i)			*Border Tree* 中的 (x_i, y_i, z_i)
前缀长度	π'	π	k	
1		b	1	(0, 1, 1)
2		ba	1	(0, 2, 3)
3	b	ab	1	(1, 2, 3)
4		ba	2	
5	b	ab	2	
6		ba	3	
7	b	ab	3	
8		babababc	1	(0, 8, 1)
9	b	abababcb	1	(1, 8, 1)
10	ba	bababcba	1	(2, 8, 1)
11	bab	ababcbab	1	(3, 8, 1)
12	baba	babcbaba	1	(4, 8, 1)
13	babab	abcbabab	1	(5, 8, 1)

6 一类文本匹配问题

众所周知，动态文本串与静态模式串的匹配问题是字符串匹配的经典问题。本节将对这类文本匹配提出一种基于 *Border Tree* 来实现的高效通用解法。其中，关于 *Border Tree* 的相关内容请详见上一节。

6.1 子问题

设模式串是长度为 m 的 b 串。

处理这一类问题之前，需要先解决 3 个在 b 串上的子问题：

1. 最长公共前缀 (LCP)：在 b 串上给两个子串 (b', b'')，问这两个子串的最长公共前缀长度。

   解法详见例 4[1]，预处理时间复杂度为 $O(m \log m)$。单次询问时间复杂度为 $O(1)$。

2. 子串拼合问题：在 b 串上两个子串 (b', b'')，令 $s = b'b''$，问串 s 是否是 b 串的子串。如果是子串的话，给出一个合法的位置。

   解法详见例 5，预处理时间复杂度为 $O(m \log m)$。单次询问时间复杂度为 $O(\log m)$。

3. 区间最大前缀问题：在 b 串上给一个区间 $[l, r]$。询问：

$$\max_{i=l}^{r} LCP(suf[i], b)$$

   解法详见例 6，预处理时间复杂度为 $O(m \log m)$。单次询问时间复杂度为 $O(1)$。如果还需要计算答案最大化时 i 合法的个数，那么单次询问时间复杂度为 $O(\log m)$。

6.2 模型转换

首先，把动态文本串与静态模式串的匹配问题简化为一个基础问题：给一个文本串 a 和一个模式串 b，每次修改 a 串后都与 b 串匹配一遍。修改操作可以是 a 串中修改某一位或是插入或删除一位。

[1]例题在第 43 页

然后转化模型，把 a 串与 b 串匹配问题变成求如下函数的值：

定义一个关于 a 串和 b 串的函数 $F(a,b)$，$F(a,b)$ 的值是一个二元组 (x,y)，x,y 的含义如下：

- $x = \max_{i=1}^{|a|} LCP(a[i:|a|], b)$
- y 表示当 x 最大化时，合法的 i 的个数。

问题就转化成了：给出 a 串和 b 串，求 $F(a,b)$，其中 a 串为动态文本串，b 串为模式串。

6.3 覆盖

在本小节中，将提出覆盖这个概念。

设模式串为长度为 m 的 b 串，文本串为长度为 n 的 a 串。

覆盖的基本思想为：用模式串 b 中的子串去精确覆盖 a 串。

下面给出覆盖的定义：

定义 6.1.

一个关于 b 串和 a 串的覆盖 $\mathcal{G}$，是一个字符串的序列 $[v_1, v_2, v_3...v_t]$，而 v_i 被称为是覆盖的一个元素。

覆盖需满足 $v_1v_2v_3...v_t = a$，且对于每个元素 v_i $(1 \le i \le t)$，均满足以下两条性质：

- 子串性质：v_i 是 b 串的子串。
- 极大化性质：若 $i < t$，则 v_iv_{i+1} 不是 b 串的子串。

特别地，对于 a 串的第 x 位，若 a_x 这个字符没有出现在 b 串中，我们也把 a_x 作为一个 v_i，不过这个 v_i 在 b 串中对应的是空串。

由于覆盖方式可能有多种，所以合法的 $\mathcal{G}$ 也可能有多种，具体实现时，只需取任意一种合法的 $\mathcal{G}$ 即可。

对于 $\mathcal{G}$ 可以简单地理解为 v_i 把 a 串分裂成了 t 个小块。

现在对于每个 v_i 用一个三元组 (s_i, l_i, r_i) 来表示：

- l_i, r_i 表示 v_i 在 b 串中的位置：$v_i = b[l_i : r_i]$。

  特殊地，若 v_i 在 b 串中表示的是空串，那么令 $l_i = r_i = 0$。

- s_i 表示 v_i 在 a 串中的位置，令 $s_i = 1 + \sum_{j=1}^{i-1} |v_j|$。

那么先来考虑如何初始化 $\mathcal{G}$。具体操作如下：

1. 新建一个数组 c，大小为字符集大小。对于第 i 这个字符，如果在串 b 中的第 x 位出现了，那么 $c_i = x$，否则 $c_i = 0$。

2. 对于 $1 \le i \le |a|$，我们认为 a_i 在 b 串中出现的位置是 c_{a_i}。如果 $c_{a_i} = 0$ 表示 a_i 并没有在串 b 中出现。我们令 $v_i = (i, c_{a_i}, c_{a_i})$。

3. 这样构造 v_i 并不满足极大化性质。那么要通过合并相邻的元素来满足极大化性质。根据 s_i 递增的顺序依次判断 $v_i v_{i+1}$ 是否是 b 串的子串。如果是的话，那么合并，否则不合并。对于这个判断，可以使用子问题 2[2] 来解决。

现在已经初始化 $\mathcal{G}$ 了。接下来考虑如何动态维护 $\mathcal{G}$。

修改操作可以是在 a 串中插入、删除或是修改某一位，在下文分析中只考虑修改某一位，插入、删除一位的做法类似。

对于修改某一位，现在需要动态维护 $\mathcal{G}$，具体做法如下：

1. 对于把第 x 位改为 y，可以先找到覆盖 x 的 v_i。由于是精确覆盖，所以 v_i 有且只有一个。

2. 根据第 x 位，把 v_i 分裂成左中右 3 个子元素：$v_{i'}, v_{i''}, v_{i'''}$，并用三元组来表示它们：

$$v_{i'} = (s_i, l_i, l_i + (x - 1 - s_i))$$
$$v_{i''} = (x, l_i + (x - s_i), l_i + (x - s_i))$$
$$v_{i'''} = (x + 1, l_i + (x + 1 - l_i), r_i)$$

   那么现在，第 x 位恰好被 $v_{i''}$ 精确覆盖。根据修改，令 $v_{i''} = (x, c_y, c_y)$，再在这 3 个子元素中，把在 a 串中表示为 $\emptyset$ 的元素删掉。

[2]子问题 2 在第 53 页

3. 根据 v_i 的极大化性质，当 v_i 分裂后，只有 v_{i-1} 和 v_{i+1} 是有可能改变的。那么也把这两个元素提出来。

4. 现在，最多有 5 个元素未满足极大化性质，根据 s_i 递增的顺序依次用子问题 2 判断相邻两个元素是否能合并。于是就可以重构这些元素使得其满足极大化性质了。

于是，就可以动态维护 $\mathcal{G}$ 了。接下来通过一个例子更好地理解 $\mathcal{G}$ 的维护：

例 10.

设串 a = cbabbabbbacb，串 b = abbacb，初始化后 $\mathcal{G}$ 为 [cb, abba, bb, bacb]。现在要把 a 串中的第 4 位修改成 c，做法是：

1. 找到覆盖这一位的 v_2 = abba；

2. 把 v_2 分裂：$v_{2'}$ = a, $v_{2''}$ = b, $v_{2'''}$ = ba；

3. 修改 $v_{2''}$ = c。

4. $v_{2'}$ 和 $v_{2''}$ 是可以合并的，将其合并。

于是此时的 $\mathcal{G}$ 为 [cb, ac, ba, bb, bacb]，满足定义中的性质，完成了重构。

6.3.1 时间复杂度分析

在本小节中，我对上述做法的时间复杂度做一些分析：

在子问题 2 中，需要进行时间复杂度为 $O(m \log m)$ 的预处理，然后使得询问单次时间复杂度为 $O(\log m)$。

在初始化阶段，会进行 n 次子问题 2 的判断，时间复杂度为 $O(n \log m)$

设修改次数为 q。那么每进行一次动态维护 $\mathcal{G}$，需要进行常数次的子问题 2 的判断，时间复杂度为 $O(q \log m)$。

此时，发现需要一个能支持如下操作的数据结构：插入、删除、查找前驱、后继和下标不小于 k 的第一个值的位置。如果使用二叉搜索树来维护 $\mathcal{G}$ 的话，就可以做到单次操作的时间复杂度为 $\log n$。由于每次修改，二叉搜索树的操作数是常数级别的，于是就可以得到动态维护 $\mathcal{G}$ 的时间复杂度是 $O(q \log n + q \log m)$ 的。

在预处理中，二叉搜索树建树是时间复杂度是 $O(n)$ 的，由于这个复杂度并非是瓶颈，可以忽略不记。

于是综合上述分析，可以在 $O((n+m+q)\log m + q\log n)$ 的时间复杂度内维护 $\mathcal{G}$。

同时，可以直接记录 s_i 作为下标，于是就可以运用线段树代替二叉搜索树，虽然时间复杂度不变，但常数可以减小。

更优地，可以使用 *van Emde Boas tree* 来代替二叉搜索树。由于在 *vEB Tree* 上单次操作时间复杂度为 $O(\log\log m)$，我们可以继续优化这个算法。

6.4 计算技巧

在上一小节中，已经可以高效地动态维护 $\mathcal{G}$ 了。

在本节中，作者会通过简化原问题，来介绍一种基于 *Border Tree* 和 $\mathcal{G}$ 的高效计算 $F(a,b)$ 的技巧。

6.4.1 简化问题

由于 $F(a,b)$ 函数的本质是把 $LCP(a[i:|a|],b)$ $(1\le i\le|a|)$ 的最大值及最大值的个数表示成一个二元组，$\mathcal{G}$ 函数也可以简单地理解为把 a 串分裂成了 t 个小块 v_j $(1\le j\le t)$。对于每个 v_j，我们令 $s_j = v_jv_{j+1}v_{j+2}..v_t$，并把 $LCP\left(s_j\left[i:|s_j|\right],b\right)$ $\left(1\le i\le|v_j|\right)$ 的最大值及最大值的个数也表示成一个二元组w_j。于是我们就可以通过合并部分的最大值及最大值的个数 w_j，来得到整体的最大值及最大值的个数 $F(a,b)$。

而且根据 v_j 的极大化性质，可以直观地发现当 $s_j = v_jv_{j+1}v_{j+2}$ 时，w_j 的值并不会变。于是 w_j 只和 v_j, v_{j+1}, v_{j+2} 有关。在上一节中，已经可以用 $O(\log n)$ 的时间复杂度找到 v_j 的前驱和后继，于是，每当需要计算 w_j 时，只需要耗费 $O(\log n)$ 的时间就能找到 v_{j+1} 和 v_{j+2}。

小结一下，我们把计算 $F(a,b)$ 的问题简化成了计算 w_j。而对于计算 w_j，我们又可以把问题转化为：

给出 b 串的 3 个子串 s_1, s_2, s_3，令 $s = s_1s_2s_3$，把 $LCP(s[i:|s|],b)$ $(1\le i\le|s_1|)$ 的最大值及最大值的个数合并。

6.4.2 新问题

设 s_1 串长度为 d，$s'' = s_2 s_3$。

现在我们来求 $LCP(s[i:|s|], b)$ $(1 \le i \le d)$ 的最大值及最大值的个数。令最大值为 Max，最大值的个数为 Num。

对于 $1 \le i \le d$，不妨令 $j = i + LCP(s[i:|s|], b) - 1$，则 $s[i:j]$ 便是 b 串的一个前缀。

特殊地，如果 s_1 是一个字符且这个字符没有在 b 串中出现，那么 $Max = 0$，$Num = 1$。

否则，可以分两种情况分别计算答案：当 $j < d$ 时和当 $j \ge d$ 时。

6.4.3 情况一 ($j < d$)

对于这种情况 $j < d$，可以把问题转化为询问：

$$\max_{i=1}^{d} LCP(s_1[i:d], b)$$

由于 s_1 是 b 串的一个子串，那么我们可以在串 b 中找到一个区间 $[L, R]$，使得 $s_1 = b[L:R]$。又因为有 $j < d$ 限制的存在，我们把串 $s_1[i:d]$ $(1 \le i \le d)$ 加长成为 $b[L+i-1:|b|]$，然后计算答案，答案是不变的。

于是，这个问题就转化成了子问题 3[3]，根据子问题 3，就可以得到 Max 和 Num 了。

但是，问题来了：该如何控制 $j < d$ 呢？

有一个简单的想法：在计算子问题 3 时，保证在 b 串中，后缀匹配的最远位置是严格小于 R 的。

考虑 $b[1:R]$ 的所有 $Border$，要求一个最大的 x，使得 $b[1:x]$ 是 $b[1:R]$ 的一个 $Border$，且满足 $0 \le x < R - L + 1$。

于是，只要对区间 $[L, R-x]$ 做子问题 3，就可以保证在 b 串中，后缀匹配的最远位置是严格小于 R 的。而对于区间 $[R-x+1, R]$，根据 $j < d$，在做子问题 3 时，答案显然是小于 x 的，所以可以忽略这种情况。

对于最大化 x，可以先建出 KMP 自动机，然后从 R 这个点开始向根用倍增算法寻找 x。关于 KMP 自动机，本文已在第二节中介绍，具体形态见图 ref1。

[3]子问题 3 在第 53 页

显然地，也可以建出 *Border Tree* 来计算答案。由于 *Border Tree* 的深度是 $O(\log m)$ 的，所以直接枚举 $b[1:R]$ 所在的点到根路径上的点计算 x 也是能保证时间复杂度的。关于 *Border Tree* 的性质请参见第六节。

现在我们给出一个例子来理解情况一：

例 11.

设串 a = babaca，串 b = abcabacabc。

现在 $\mathcal{G}$ 为：[b, abaca]，其中 v_1 = b, v_2 = abaca，v_2 对应的三元组为 $(2,4,8)$。

如果现在需要计算 w_2，可以发现 $LCP(v_2,b)$ 为 2，所对应的前缀为 ab。此时属于情况一。

分析 b 串中的子串 $[4,8]$。

如果直接对于区间 $[4,8]$ 做子问题 3 的话，答案会是 3，所对应的前缀为 abc，匹配的区间为 $[8,10]$。这个是错误的。

需要先找到 x，关于 x 的定义见上文*(在子串 $[4,8]$ 中 $x=1$)*。现在对于区间 $[4,8-x]$ 做子问题 3，可以得到答案为 2，所对应的前缀为 ab，匹配的区间为 $[3,5]$，匹配的最远距离是小于 8 的。

于是对于计算 w_2，就可以得到 $Max = 2, Num = 1$。

6.4.4 情况二 ($j \geq d$)

由于 s_1 是 b 串的一个子串，那么我们可以在串 b 中找到一个区间 $[L,R]$，使得 $s_1 = b[L:R]$。

因为 $j \geq d$，所以 $s_1[i:d]$ 是 $s[i:j]$ 的前缀。可以显然地发现，由于 $s_1[i:d] = b[L+i-1:R]$，所以 $s_1[i:d]$ 一定是 $b[1:R]$ 的一个 *Border*。

考虑一种暴力的方法：先建出 *KMP* 自动机，然后在根到点 R 的路径上暴力枚举点 R 的 *Border*，对于一个长度合法的 *Border*，可以把这个 *Border* 与 s'' 拼接起来得到串 s'，再计算 $LCP(s',b)$。由于串 s' 是由 3 个 b 串的子串拼起来的，所以只需计算 3 次子问题 1[4] 就可以得到 *Max* 了，且 $Num = 1$。

但由于 *KMP* 自动机深度是可以达到 $O(m)$ 的，所以单次暴力枚举的时间复杂度是可以达到 $O(m)$ 的。现在，让我们来优化上述暴力做法。可以很自然地想

[4]子问题 1 在第 53 页

到，我们可以根据 *Border Tree* 深度是 $O(\log m)$ 这一性质来优化暴力。如果能在 $O(1)$ 的时间复杂度内完成对一整个 *Border Group* 的询问，那么就可以把单次暴力枚举的时间复杂度降到 $O(\log m)$ 了。

考虑如何在 $O(1)$ 的时间复杂度内完成对一整个 *Border Group* 的询问。首先来回顾一下我们暴力枚举的 *Border Group* 的表现形式，设这个 *Border Group* 为 v：

$$\left[\pi'_v\pi_v^{l_v}, \pi'_v\pi_v^{l_v+1}, \pi'_v\pi_v^{l_v+2}, ..., \pi'_v\pi_v^{x}\right]$$

具体详见 5.3 中的性质。

这个转化的本质是：在 *KMP* 自动机中，只需考虑一个 i $(1 \le i \le d)$，而现在要考虑的是一组具有相同性质的 i。

特殊地，如果这个 *Border Group* 中有 $\pi'_v\pi_v$ 这个串，那么用暴力判断，并且令 $l_v = 2$。然后，根据周期串和基的定义，这个 *Border Group* 中剩下的串都是周期串，它们周期的长度为 $|\pi_v|$。

现在对 b 串进行如下的分类讨论：

1. 当串 b 是周期串且周期的长度为 $|\pi_v|$ 时，先令

$$len = LCP\left(s'', b\left[\left|\pi'_v\right| + 1 : |b|\right]\right)$$

对于这一个 *Border Group*，可以得到匹配长度为：

$$\min\left(|b|, \min_{k=l_v}^{x}\left\{\left|\pi'_v\right| + k\,|\pi_v| + len\right\}\right)$$

可以这样理解：当 *LCP* 长度不超过 $|b|$ 时，由于串 b 和 *Border Group* 中的串都是周期串且周期的长度为 $|\pi_v|$，那么可以发现 *len* 的值是固定的，与 k 的取值无关。其中，*len* 的意义为：选择一个 *Border* 后，s'' 拼在后面能继续匹配的长度。

于是，可以先判断出最大的匹配长度是否为 $|b|$，如果是的话，$Max = |b|$，*Num* 为合法的 k 的数量。否则，*Max* 为最大匹配长度，$Num = 1$。

2. 否则，一定存在一个 q，满足：

$$\left|\pi'_v\pi^{r_v}\right| < q \le \left|\pi'_v\pi^{r_v+1}\right|, b_q \ne b_{q-|\pi_v|}$$

目前，已知 r_v，那么可以通过计算 LCP 来求出最小的 q。

由于已知 *Border Group* 中的串为周期串，直观地，可以发现 $b[1:q-1]$ 也是周期串且周期的长度为 $|\pi_v|$。

设 d_1 为这个 *Border Group* 中最长串长度：$\left|\pi'_v\pi_v^x\right|$，$d_2$ 为这个 *Border Group* 中原来最长串长度：$\left|\pi'_v\pi_v^{r_v}\right|$ (具体详见 5.3 中的性质)。再设 k 为使得 π_v^k 为串 s'' 的前缀的最大整数 (可以通过多次计算子问题 1 来得到 k)，并令

$$
\begin{aligned}
len_1 &= LCP\left(s'', b\left[|\pi_v|+1:|b|\right]\right)\\
len_2 &= LCP\left(s''\left[k|\pi_v|+1:|s''|\right], b\left[d_2+1:|b|\right]\right)\\
rem &= \left(q-1-\left|\pi'_v\right|\right) \bmod |\pi_v|
\end{aligned}
$$

现在会有 3 种情况：

(a) 当 $x+k<r_v$ 时，那么在这个 *Border Group* 中的 $\pi'_v\pi_v^x$ 后面拼接 s'' 能使得匹配的长度最长。所以 $Max=d_1+len_1$，$Num=1$。

(b) 当 $l_v+k>r_v$ 时，那么可以把 b 串简化为 $b[1:q-1]$，然后通过分类 1 来解决 (其实这时 $Max=q-1$)。

(c) 当 $l_v+k\le r_v$ 且 $x+k\ge r_v$ 时，可以发现 b 串的前 d_1 位是一定能匹配的 (即 $Max\ge d_1$)。此时还是有两种情况：

如果 $len_2>rem$，那么 $Max=d_2+len_2$，$Num=1$。

如果 $len_2\le rem$，那么又可以把 b 串简化为 $b[1:q-1]$，然后通过分类 1 来解决 (这时 $Max\le q-1$)。

现在本文给出一个例子来更好地理解分类 1。

例 12.

设串 $b=$ abababababa，子串 $s_1=$ ababab, $s_2=$ ab, $s_3=$ ba，那么 $s''=$ abba。

由于 s_1 是 b 串的一个子串，那么可以在串 b 中找到一个区间 $[L,R]$，使得 $s_1=b[L:R]$。*(在这个例子中令 $[L,R]=[3,8]$)*

如果在 *KMP* 自动机中计算答案，枚举 $b[1:8]$ 的 *Border*，可以发现 ababab, abab, ab 这三个 *Border* 的长度都是符合条件的 *(没有超过 s_1 的长度)*。那么可以在这个两个 *Border* 后接上 s'' 成为 abababababba, abababba, ababba。其中，

第一个串匹配的长度为 8，第二个串匹配的长度为 6，第三串匹配的长度为 4，那么 $Max = 8, Num = 1$。

而在 *Border Tree* 中计算答案时，可以发现 $b[1:8]$ 的所有 *Border* 是属于一个 *Border Group* 的，这个 *Border Group* 的三元组为 $(0, 2, 4)$。由于 b 串是周期串且周期的长度为 2，所以这个例子是属于分类 1 的。根据上文的定义，这个 *Border Group* 中包含了 3 个 *Border*：ababab, abab, ab。

对于 ab 这个 *Border*，据上文所述，通过暴力计算，可以得到匹配长度为 5。接下来，由上文公式，就能算出 $len = 2$。对于 ababab, abab 这两个 *Border*，就可以直接算出匹配长度：8, 6。于是，我们可以得到 $Max = 8, Num = 1$。

分类 2 中有 4 种情况要讨论，但由于篇幅有限，所以本文在这里只根据一种情况进行举例分析：

例 13.

设串 b = abababababacde，子串 s_1 = ababab, s_2 = ababac, s_3 = de，那么 s'' = ababacde。

由于 s_1 是 b 串的一个子串，那么可以在串 b 中找到一个区间 $[L, R]$，使得 $s_1 = b[L:R]$。*(在这个例子中令 $[L, R] = [3, 8]$)*

在 *Border Tree* 中计算答案时，可以发现 $b[1:8]$ 的所有 *Border* 是属于一个 *Border Group* 的，这个 *Border Group* 的三元组为 $(0, 2, 4)$。由于 b 串并不是周期串，所以这个例子是属于分类 2 的。根据上文的定义，这个 *Border Group* 中包含了 3 个 *Border*：ababab, abab, ab。

对于 ab 这个 *Border*，据上文所述，通过暴力计算，可以得到匹配长度为 4。然后根据上文定义，我们可以得到：$q = 10, l = 2, x = 3, r = 4$。接下来，由上文公式，又可以得到：$d_1 = 6, d_2 = 8, k = 2, len_1 = 5, len_2 = 4, rem = 1$。

于是，可以发现 $l+k \le r$ 且 $x+l \ge r$，是属于情况 2c。而且又满足 $len_2 > rem$，那么在这个例子中 $Max = 12, Num = 1$。*(使用 abab 这个 Border)*

6.4.5 时间复杂度分析

根据上文中对于情况一和情况二的分析，可以发现，如果运用 *Border Tree* 来实现计算的话，暴力枚举 *Border Group* 的个数是 $O(\log m)$ 级别的。对于每个

Border Group 的计算，都可以把问题分类讨论，根据具体情况进行不同的转化。从时间复杂度角度来说，计算一个 *Border Group* 的时间复杂度实际上等于进行常数次子问题 1 的运算的时间复杂度。由于子问题 1 单次是可以 $O(1)$ 询问的，所以计算一个 w_j 的时间复杂度是 $O(\log m)$。

那么联系上一节的时间复杂度分析，对于动态修改文本串 a 并对静态模式串 b 求 $F(a,b)$ 这个问题，是可以在时间复杂度为 $O((n+m+q)\log m+q\log n)$ 内解决的。

当解决这个动态文本串与静态模式串匹配的基础问题之后，就可以把这个问题推广，从而解决以这个问题为基础的一类问题了。在下节中，我们将对这一类问题进行举例讨论。

6.5 例题

现在本文给出这类问题的一个例题：

例 14 (2015 国家集训队互测 *Sone2*)**.**

给一个长度为 n 的主串 a 和一个长度为 m 的模式串 b。

有 q 个操作，操作有 4 种：

1. 修改 a 串的一位，求 $F(a,b)$。
2. 选取 a 串中的一个区间 $[L,R]$，求 $F(a[L:R],b)$。
3. 求子问题 1。
4. 求子问题 2。

数据范围：$1\le n,m,q,$ 字符集大小 ≤ 100000

时限：$2s$

在这道题目中，除了第二种操作，其他操作的计算方法都已在上文中提及。对于第二种操作，一个简单的想法就是维护 $a[L:R]$ 这个串的覆盖 $\mathcal{G}'$，然后进行计算。由于 $a[L:R]$ 是 a 串的子串，可以发现，如果把 $\mathcal{G}'$ 序列的头尾两个元素去掉，剩下的序列就是 $\mathcal{G}$ 序列的子序列了。于是对于第二种操作，只需要对于 $\mathcal{G}'$ 的头尾两个元素进行特殊计算，而中间的部分可以运用数据结构直接从 $\mathcal{G}$

序列中提取。那么就能在单次 $O(\log n + \log m)$ 的时间复杂度内进行构造 $\mathcal{G}'$ 了。然后在维护 v'_j 的同时维护 w'_j，就可以直接对 $\mathcal{G}'$ 计算来得到答案。

于是，此题就可以在 $O((n+m+q)\log m + q\log n)$ 的时间复杂度内解决了。

此题还有一个部分分，在这类数据中 b 串是随机的。由于 b 串随机，那么就可以发现如果用 b 串建 *KMP* 自动机，*KMP* 自动机的深度就是 $O(\log m)$ 级别的。于是，在上述算法中，就不需要运用 *Border Tree* 了，可以直接用 *KMP* 自动机代替。在这类数据中，这个解法并没有能降低时间复杂度，但大大降低了实现此题的难度。

此外，此题可以加强一下：建一棵 n 个点的树，树上每个点都有一个字符。对于树上的一条链，把链上点的字符按顺序拼接起来能得到一个字符串 a。再给一个长度为 m 的模式串 b。有 q 个操作，每个操作可以改变树上一个点的字符，或是给定树上的一条链，得到串 a，求 $F(a,b)$。

再加强的话，还可以把这棵树变成动态树。

对于解法的话，可以用 *Link-Cut Tree* 来维护覆盖 $\mathcal{G}$。每个 *splay* 内维护这一条重链构成字符串的覆盖 $\mathcal{G}'$。每次 *access* 操作把维护重链的 *splay* 拼接或分裂，只会造成常数个 v_j 和 w_j 重构。所以可以在 $O((n+m)\log m + q\log n\log m)$ 的时间复杂度内解决这个问题。

7 感谢

感谢中国计算机学会提供学习和交流的平台。

感谢绍兴一中的陈合力老师、董烨华老师、游光辉老师、邵红祥多年来给予的关心和指导。

感谢国家集训队教练余林韵和陈许旻的指导。

感谢清华大学的俞鼎力、董宏华、何奇正学长对我的帮助。

感谢绍兴一中的张恒捷、任之洲、贾越凯对我的帮助和启发。

感谢绍兴一中的杨嘉诚、郭雨、邵杰晶、翁天东、陶渊政、洪华敦等同学对我的帮助和启发。

感谢绍兴一中的贾越凯同学为本文审稿。

感谢其他对我有过帮助和启发的老师和同学。

感谢父母对我的关心和照顾。

8 参考文献

[1] 许智磊：《后缀数组》

[2] 朱泽园：《多串匹配算法及其启示》

[3] 杨弋：《Hash在信息学竞赛中的一类应用》

[4] 罗穗骞：《后缀数组——处理字符串的有力工具》

[5] 董华星：《浅析字母树在信息学竞赛中的应用》

[6] 陈立杰：《后缀自动机Suffix Automaton》

[7] 王悦同, 徐毅, 徐子涵：《Suffix Cactus》

[8] Amihood Amir,Gad M. Landauy,Moshe Lewensteinz,Dina Sokolx,“Dynamic Text and Static Pattern Matching”

[9] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali, “Linear Programming and Network Flows (4th ed.)”, John Wiley & Sons Inc.

[10] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[11] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

后缀自动机及其应用

长沙市一中 张天扬

摘 要

后缀自动机作为一种OI中新兴的字符串处理工具，越来越受到广大出题人的欢迎。本文旨在对后缀自动机的性质以及一些应用做一个归纳，并通过一些例题来深入对后缀自动机性质的研究。

1 后缀自动机的定义及构造

1.1 后缀自动机的定义

一个串S的后缀自动机（SAM）是一个有限状态自动机（DFA），它能且只能接受所有S的后缀，并且拥有最少的状态与转移。

我们将在下面证明：一个串S的后缀自动机的状态数和转移数都是$O(|S|)$的。

1.2 后缀自动机的构造

几个定义

定义后缀自动机的母串为S。令$S[l,r]$表示S中第l个字符到第r 个字符组成的字串。令从第i 个位置开始的后缀为suf_i，到第i个位置为止的前缀为$pref_i$。

令SAM_{str}表示从初始状态读入字符串str后的状态（或者也可以叫节点）。

对于一个S的一个子串str，令$right_{str}$表示str在S中每一次出现的右端点位置组成的集合。

如果没有特殊说明，我们认为字符集为所有小写字母。并且把字符集大小视为常数，在计算复杂度的时候不予考虑。

复杂度证明

考虑一个串str。如果str是S的一个子串，那么SAM_{str}应该是一个合法状态。这是因为我们可以在str后添加若干字符来变成S 的一个后缀，而这个后缀应该是一个可接受的状态，那么SAM_{str} 就必须是一个合法状态。反之，如果str 不是S的子串，那么SAM_{str} 应该是一个不合法的状态（或者说空状态），因为无论怎么添加字符它都不会变为S的一个后缀。

也就是说，对于S的每一个子串，后缀自动机中都应该有对应的状态。但是显然我们不能对于每个子串单独建立一个状态，因为子串个数是$O(|S|^2)$ 的。

考虑S的一个子串str。如果$right_{str}$和$right_{a+str}$（其中a 是一个字符串）完全一致的话，那么我们可以把它们合并为同一个状态。因为我们从这两个字符串开始添加字符，能够得到的后缀是一样的（或者说除了前面的a字符串以外一样）。换言之，我们从它们的状态开始，能够转移到的接受状态也是一样的。因此，这两个串在后缀自动机中的本质是一样的，它们可以合并为同一个状态。

接下来我们考虑S的两个子串A和B。如果$right_A$ 和$right_B$ 有交，那么显然有一个串是另一个串的后缀，不妨设A是B的后缀，那么容易看出$right_A \subset right_B$。也就是说，对于S 任意两个子串，它们的$right$ 集合要么没有交集，要么一个包含另一个。我们令一个状态s 的父状态（fa_s）为满足$right_s \subset right_{fa_s}$ 且$|right_{fa_s}|$最小的状态。那么所有状态会形成一个树结构，我们把它叫做$parent$树。

显然初始态的$right$集合是$1 - |S|$的所有整数。而对每一个叶节点，它的$right$集合大小为1。也就是说叶节点个数是$O(|S|)$的。而一个非叶节点至少有两个子节点（如果只有一个，那么可以把它和它的子节点合并），那么非叶节点至多有$O(|S|)$个，因此**后缀自动机的状态数是$O(|S|)$的**。

考虑转移。注意到状态是$O(|S|)$的，不妨考虑一个从初始状态开始的自动机的树形图：树形图上边的数量显然是$O(|S|)$的。那么只需要考虑非树边。

考虑一条非树边$u \to v$。我们从初始状态沿树边走到u，然后经过这一条非树边走到v，然后继续向下走一定能走到至少一个后缀。也就是说，考虑一个后缀把它经过的第一条非树边覆盖，那么所有非树边一定都会被覆盖。因此，非树边的数量不会超过后缀的数量。所以我们证明了**后缀自动机的转移数是$O(|S|)$的**。

构造方法

后缀自动机有一种非常简单的构造方法：增量法。

我们对每个状态s，记它代表的最长子串的长度为len_s。

考虑我们当前已经有了前$|S|-1$个字符的后缀自动机，且现在的自动机中$[1,|S|-1]$位于状态$last$。现在加入第$|S|$个字符（不妨设为c），那么我们新建一个状态np，显然$len_{np}=|S|$。

之后考虑转移：显然我们应该加入一个$last \to np$的转移，我们也应该加入一个$fa_{last} \to np$的转移……直到我们发现，这个状态已经有了一个字符c的转移。不妨设这个状态是p，设它经过字符c 的转移后的状态为q。

此时，$right_q$会多出一个：右端点为$|S|$的串，且最长的长度是len_p+1。那么有两种情况：

1. $len_q = len_p + 1$。此时q代表的所有串的$right$仍然相同，那么我们只需要令$fa_{np}=q$ 即可。

2. $len_q > len_p + 1$。这种情况下q代表的串中，长度不超过len_p+1的串的$right$集合会多出一个值$|S|$，而长度超过它的串则不会。那么为了维护一个状态中所有串的$right$相同这一性质，我们需要新建一个状态nq，nq代表的是原来q代表的串中所有长度不超过len_p+1 的串，因此$len_{nq}=len_p+1$，nq的其他属性（fa 和转移）和原来的q 点一致。同时容易发现$fa_q=fa_{np}=nq$。

   然后我们再次从p开始：本来p的c字符转移到的是点q，现在它应该转移到nq。同理，如果fa_p的c字符转移到的是点q，那么它也应该转移到nq……直到发现当前点的c字符转移到的不是q为止。

至此，我们完成了后缀自动机的构造。

1.3 后缀自动机的几个基本性质

上文中提到了后缀自动机的几个基本性质。将它们归纳总结如下：

性质一：每个状态s代表的串的长度是区间$(len_{fa_s}, len_s]$。

性质二：每个状态s代表的所有串在原串中的出现次数及每次出现的右端点相同。

性质三：在*parent*树中，每个状态的*right* 集合是它的父状态*right* 集合的子集。

1.4 *right*集合的求法

首先，在$pref_i$所在状态的*right*集合中加入i。

之后，我们按*parent*树中的深度从大到小，依次将每个状态的*right*集合并入它父状态的*right* 集合。

这样就可以求出每个状态的*right*集合了。但是这样做是$O(|S|^2)$的。

根据实际的问题，我们可能需要求出*right*集合的大小或者最大值等数值，可以在$O(|S|)$的时间内求出。如果确实需要每个状态的*right*集合，可以使用平衡树来维护，在合并的时候使用启发式合并[1]，可以做到$O(|S|\log|S|)$ 的复杂度。如果需要在线，则可以使用可持久化平衡树来维护。

1.5 使用后缀自动机求后缀树和后缀数组

除了后缀自动机之外，后缀树和后缀数组是我们通常使用的后缀数据结构。但是求它们一般不太方便：后缀树的线性构造可以说相当麻烦，而后缀数组的线性构造常数比较大。因为后缀自动机的构造常数小而且比较简洁，我们可以考虑使用后缀自动机来求后缀树和后缀数组。

性质四：后缀自动机的*parent*树是原串的反向前缀树。

首先，反向前缀树[2]的定义是：把每一个前缀的反串插入到一个trie中，并把没有分支的链合并（就像后缀树那样）。

这个性质是容易发现的。考虑一个前缀在后缀自动机上的状态，我们一直沿fa 指针走，每次会变成当前串的一个后缀，直到空串。把这个过程反过来看，就是在反向前缀树上从上向下走。

那么，我们求出初始串的反串的后缀自动机。它的*parent* 树就是原串的后缀树！

[1]关于平衡树的启发式合并的问题，因为与本文无关在此不作讨论，有兴趣可以参见《小树苗与集合》解题报告算法八

[2]其实就是反串的后缀树

为了还原后缀树，我们可以求一下后缀自动机上每个状态*right*集合内的任意一个值（比如最大值）。然后就容易找出每个点在后缀树上父边上的字符串了。

最后只需要对后缀树DFS一遍就可以求出后缀数组了。

2 后缀自动机的应用

2.1 一个解释

我们描述“将串放到后缀自动机上运行”，可以类比KMP和AC自动机的运行，也就是：从初始状态开始，每次如果存在对应的转移就转移，否则沿fa指针回跳，直到存在对应的转移或者跳出了后缀自动机为止。

2.2 几个经典问题

首先我们来讨论几个经典问题的解法。

2.2.1 最长公共子串

我们对其中一个串建立后缀自动机。将另一个串放到后缀自动机上运行，并时刻维护当前子串的长度即可。

对于多个（大于2）串的最长公共子串，我们对其中一个建立后缀自动机，将其它的串分别放到后缀自动机上运行，对每个状态维护它对每个串的最长匹配长度。然后每个状态把所有串在这个状态的最长匹配长度取min，所有状态取max即可。

2.2.2 字典序k小子串

我们对后缀自动机做一遍拓扑，就可以求出从每个点开始可以到达多少个子串。然后就可以从初始状态开始DFS，如果当前状态能到的子串不少于k就继续DFS，否则就把k减去当前状态能到的子串个数然后回溯。

如果求不同子串的k小则不用求*right*集合大小。否则需要求出每个点的*right*大小。

2.2.3 字典序最小后缀（最小循环表示）

把原串复制一遍接到后面，然后构造后缀自动机。

从初始状态开始，每次走字典序最小的转移，走$|S|$次之后得到的就是最小循环表示。

如果求的是最小后缀，就在原串后加入一个比字符集中最小的字符更小的字符作为终止，然后再复制一遍即可。

下面我们以一些简单的例题来探讨后缀自动机的基础应用。

例题一：字符串[3]

题目大意

给定n个字符串，询问每个字符串有多少非空子串是所有n个字符串中至少k个的子串。

$1 \leqslant n, k \leqslant 10^5, \sum |S_i| \leqslant 10^5$

题目分析

首先把n个字符串连接起来，中间用一个不在原字符集中的字符隔开。然后构造它的后缀自动机。对后缀自动机中的每个节点，我们需要计算它是原来n个串中多少个串的子串。

基本的想法是：将每个串放到后缀自动机上运行，对于运行到的每个节点，把它沿fa指针走到根的路径上的每个节点的出现次数+1。但是这样做会导致重复。我们可以对每个状态记录一下最后一次被哪一个串到达，那么每当到一个点时，只需要向上走到第一个当前串已经到达过的点就可以了。如果直接暴力的话，复杂度是$O(n\sqrt{n})$的[4]。使用树链剖分可以达到$O(n\log^2 n)$。

之后只要递推一遍求出每个节点开始沿fa走到根的路径上有多少个串是出现了至少k次的，然后再把每个串放到后缀自动机上运行，把每个运行到的节点的值相加即可。

[3]**题目来源：Adera 1 杯冬令营模拟赛**

[4]需要非常特殊的串才会达到这个复杂度，而且常数非常小

例题二：回文串[5]

题目大意

给一个长度为n的字符串，求它的所有回文子串中出现次数乘以长度的最大值。

$1 \leqslant n \leqslant 3 \times 10^5$

题目分析

构造原串的后缀自动机，并求出每个节点$right$集合的最大值$rmax$。

之后把反串放到后缀自动机上运行，如果当前反串中的匹配串$[l, r]$[6]覆盖了当前节点的$rmax$，那么$[l, rmax]$是一个回文串。

详见参考文献[4]。

例题三：识别子串

题目大意

给一个长度为n的字符串，定义一个位置i的识别子串为包含这个位置且在原串中只出现一次的字符串。求每个位置的最短识别子串长度。

$1 \leqslant n \leqslant 10^5$

题目分析

显然所有的识别子串在后缀自动机中都在那些$right$集合大小为1的节点上。那么我们首先求出所有这样的节点。对于每一个集合大小为1的节点，我们容易知道它唯一一次出现的右端点。

而显然，如果一个串的出现次数为1，那么把它向左扩展一段之后的出现次数也为1。所以我们对每个$right$ 集合大小为1的节点，算出这个右端点向左最少扩展多长的出现次数是1。那么每一个$right$ 对应的就是一段区间内和一个值

[5]**题目来源：** APIO2014

[6]这里指在原串中的位置

取min，然后一个前缀和一个公差是−1的等差数列取min。这可以排个序之后简单的模拟实现。

复杂度为$O(n \log n)$。

例题四：差异[7]

题目大意

给一个长度为n的字符串，求它的所有后缀两两的最长公共前缀长度之和[8]。

$1 \leqslant n \leqslant 5 \times 10^5$

题目分析

首先我们把这个字符串反过来，那么我们要求的就是：所有前缀两两最长公共后缀长度之和。

性质五：两个串的最长公共后缀，位于这两个串对应的状态在*parent* 树上的最近公共祖先状态

这是因为，一个串对应节点的祖先节点都是它的后缀，且深度越大则长度越长。

那么我们把每个前缀所在的节点染黑，然后问题就变成了每个节点是多少个黑色节点的LCA。这是一个简单的问题，在*parent*树上从下往上递推一遍即可。

后缀自动机也可以与其它数据结构和字符串工具相结合，从而解决一些较难的问题。

[7]**题目来源：**AHOI2013

[8]这里给出的是原题经过简单的转化后的题意

例题五：珠宝商[9]

题目大意

给定一棵n个点组成的树，每个点有一个字符。再给定一个长度为m的字符串S。对每两个点组成的点对，它们之间路径上的点组成了一个字符串，求所有这样的字符串在S中出现次数的和。

$1 \leqslant n, m \leqslant 5 \times 10^4$

题目分析

考虑使用点分治。由于n, m同阶，一下计算复杂度时统一使用n。

1. 分治块大小小于$\sqrt{n}$。此时我们直接DFS 每一条路径，并在S的后缀自动机中统计出现次数。因为这样的块是互不相交的，计算一个块的复杂度是$O(size^2)$。那么复杂度最多是$O(n\sqrt{n})$。

2. 分治块大小大于$\sqrt{n}$。考虑如何计算经过当前分治块重心的路径，对于其它路径直接递归分治即可。

   考虑重心x上的字符c在S中的出现位置。如果一条路径形如$a \to x \to b$，那么$a \to x$这一段在S中的出现位置的$right$必然是c在S中的出现位置的子集。同理，我们把$x \to b$这一段反过来，那么它在S 的反串中出现位置的$right$必然是c在S 中出现位置的子集。

   那么考虑从x开始DFS分治块。每次维护当前串在原串和反串的后缀树上的位置。然后我们将两个后缀树从上向下递推一遍就可以求出每一个后缀（也就是每一个位置）的匹配数量了。最后把两个后缀树上对应位置的匹配数相乘，就可以得到经过重心的路径在原串中的出现次数之和了。

   注意这样做会多算那些a, b在重心的同一棵子树的情况，那么把重心的每一棵子树分别使用相同的方法计算一遍就可以了。

   计算一个分治块的复杂度是$O(size + n)$。注意分治块大小大于$\sqrt{n}$的至多只有$O(\sqrt{n})$ 个[10]，那么复杂度是$O(n\sqrt{n})$。

[9]**题目来源：**CTSC2010

[10]证明的思路是：每次分治子树大小至少缩小一半，那么进行k次分治之后最大块不超过$\frac{n}{2^k}$，此时块数是2^k，然后令最大块大小不超过$\sqrt{n}$即可得出结论。

综上，我们得到了一种复杂度是$O(n\sqrt{n})$的优秀算法。

例题六：str[11]

题目大意

给定两个长度分别为n, m的字符串。定义两个字符串匹配为它们至多有一位不同。求这两个字符串的最长公共子串。

$1 \leqslant n, m \leqslant 10^5$

题目分析

我们考虑最后能匹配出来的串：它中间有一位不匹配，而这一位左边和右边分别匹配。假如我们已经知道了左边的串，那么我们就容易知道它在两个串中的每一次出现位置，然后把相应的右边的串两两取一个LCP[12]就可以了。

那么我们将两个串连接起来，中间加一个不在原字符集中的字符，求出这个串的后缀自动机。考虑后缀自动机中的一个节点：它的*right*集合中，有一部分是出现在第一个串内部的，另一部分是出现在第二个串内部的。比如有一个*right*是a，它在第一个串内部；另一个*right*是b，它在第二个串内部，那么我们求$a+2$和$b+2$的LCP，就是相应的最长匹配长度。如果我们把所有这样的(a, b)对的相应LCP 都求出来，取其最大值加上当前点的*len*，得到的就是这一个节点的答案。把所有节点的答案取最大值就是最后的答案。

但是求所有(a, b)对的LCP时间效率上肯定是不行的。我们考虑把所有后缀排序，那么我们对于一个节点，如果把所有的a和b混合起来按照对应的后缀顺序来排序，那么我们就只要计算所有排序后相邻的出现在两个不同的串内部的LCP。对于这样一个问题，我们按*parent*树自底向上依次考虑每个节点，使用平衡树来维护整个序列就可以了。在平衡树合并的时候维护一下信息，就可以做到$O((n+m)\log(n+m))$的复杂度了。

[11]**题目来源：Feyat Cup 1.5 题目作者：黄志翱**

[12]最长公共前缀

3 字母树的后缀自动机

我们将KMP算法推广到字母树上，形成了AC自动机。同样的，我们也可以把后缀自动机推广到字母树上。

我们先来看一道例题：

例题七：pty的字符串[13]

题目大意

给一棵n个节点的有根树，每条边上有一个字符。定义一条路径是从某个节点开始，向下走到某个节点结束，边上的字符组成的字符串。定义两个字符串匹配为它们完全相同。再给一个长为m的字符串，求它的子串和树的路径共有多少对匹配。

$1 \leqslant n \leqslant 8 \times 10^5, 1 \leqslant m \leqslant 8 \times 10^6$

题目分析

基础的做法是，对字符串建立后缀自动机，并求出*right*集合的大小。然后DFS 字母树即可。

但是，在本题中，因为串的长度太大，对串建立后缀自动机的做法空间上无法接受[14]，需要另辟蹊径。

考虑对所给的字母树建立后缀自动机。我们插入一个节点的时候，将它的父节点的前缀[15]在后缀自动机中的状态作为*last* 状态来插入。但是会有一个问题：一个节点有可能有两个子节点的边上的字符是一样的。

其实解决方法很简单：我们把同一个节点下字符相同的子节点合并起来，合并后的节点*right*初始大小是合并的节点个数。

最后将所给字符串放到后缀自动机上运行一遍统计一下答案就可以了。

再来看另一道例题：

[13]**题目作者：彭天翼**

[14]也许将边使用hash来保存的方法可以。

[15]这里的前缀指从根节点到它的路径。

例题八：诸神眷顾的幻想乡[16]

题目大意

给定一棵n个节点的树，每个节点上有一个字符。定义路径为某两点间的节点上的字符组成字符串。求树上有多少条互不相同的路径。

$1 \leqslant n \leqslant 10^5$，字符集大小为10，树至多有20个叶节点。

题目分析

注意到题目的特点：叶节点很少。那么我们依次把每个叶节点当做根，然后把它们合在一起形成一棵字母树。

接下来的问题就是这棵字母树上有多少条互不相同的从上至下的路径。我们对它建立后缀自动机后，对每个节点将$len - len_{fa}$计入答案即可。

4 总结

后缀自动机作为OI中新兴的字符串处理工具，具有功能全面，代码简洁，复杂度低，常数小等优点。相应的，它需要我们在应用过程中进行更多的思考与研究，才能理解后缀自动机的优美之处。

5 鸣谢

- 感谢CCF提供的学习与交流的平台。
- 感谢引进后缀自动机的陈立杰。
- 感谢教会我后缀自动机的石文斌同学。
- 感谢陈胤伯、吕凯风同学在论文写作过程中的帮助。
- 感谢教练周祖松老师的支持。

[16]题目来源：ZJOI2015 题目作者：陈立杰

参考文献

[1] 陈立杰，《后缀自动机》，2012冬令营营员交流。

[2] 范浩强，《后缀自动机与线性构造后缀树》。

[3] 许昊然，《CTSC2010珠宝商 新解》，2013国家集训队作业。

[4] 张天扬，《APIO2014回文串 解题报告》，2015国家集训队作业。

[5] 陈立杰，《ZJOI2015 Day1 题解》。

[6] 黄志翱，《Feyat Cup 1.5 题解》。

生成函数的运算与组合计数问题

杭州学军中学 金策

摘 要

本文介绍了处理形式幂级数的一些高效算法，并在生成函数的运算过程中加以应用，从而解决一系列组合计数问题。

目 录

1 引言 **83**

2 多项式与形式幂级数 **83**

2.1 多项式 83

2.2 多项式的基本运算 83

2.3 形式幂级数 84

3 组合计数问题 **85**

3.1 组合对象 85

3.2 普通生成函数 86

3.3 指数生成函数 86

4 乘法逆元 **87**

5 乘法逆元的应用 **88**

6 对数与指数运算 **90**

6.1 复合运算 90

6.2 形式导数 90

6.3 对数函数与指数函数 91

6.3.1 对数函数的计算 91
6.3.2 指数函数的计算 91
6.4 牛顿迭代法 92
6.5 k次幂的计算 93

7 集合的计数 **93**
7.1 有标号集合的计数 93
7.2 无标号集合的计数 94

8 环的计数 **95**
8.1 有标号环的计数 95
8.2 无标号环的计数 96

9 复合运算 **97**
9.1 复合与复合逆 97
9.2 Lagrange反演 98

10 二元生成函数 **100**

11 结语 **102**

1 引言

组合计数问题是信息学竞赛中常见的一类问题，而生成函数往往是解决这类问题的重要工具。近年来，信息学竞赛中出现了这样一类计数问题，不仅需要选手根据题意对生成函数进行分析与推导，还要求使用高效的算法完成各类多项式运算，才能优化求解的时间复杂度。本文将对这一类问题进行进一步分析与总结。

本文第2,3节回顾了一些需要用到的基本知识。其中第2节提到了几种熟知的多项式乘法算法，以及算法实现时需要注意的地方；第3节引入了组合对象的生成函数的概念，并分析了生成函数加法、乘法的组合意义。

第4节介绍了求解形式幂级数乘法逆元的牛顿迭代法，这一算法是后文许多算法的基础。第5节中的例题对该算法进行了简单应用。

第6节介绍了形式幂级数的对数、指数函数的求解算法。

第7,8节分析了集合、环这两类常见组合模型的计数方法，并对第6节中的算法进行了应用。

第9节简单介绍与复合运算相关的算法和定理。

第10节介绍了运用二元生成函数解决问题的技巧。

2 多项式与形式幂级数

2.1 多项式

多项式是我们熟知的数学概念。一个关于x的多项式可以写成

$$A(x)=\sum_{i=0}^{n-1}a_ix^i \tag{2.1}$$

的形式，其中系数a_i均为某个环R中[1]的元素。这些多项式组成多项式环$R[x]$。

非零多项式$A(x)$的次数定义为其最高次项的次数，记作$\deg A(x)$。

2.2 多项式的基本运算

设参与运算的多项式的最高次数为n。那么多项式的加法、减法显然可以在$O(n)$时间内计算。

[1]一般指可交换环，可以是复数域$\mathbb{C}$、实数域$\mathbb{R}$、整数环$\mathbb{Z}$、剩余类环$\mathbb{Z}/n\mathbb{Z}$等

我们关心的是两个多项式的乘积。朴素的计算方法需要$O(n^2)$时间，并不够优秀。

一种优化方法是分治乘法[2]，它的原理是利用

$$\begin{aligned}&(Ax^m+B)(Cx^m+D)\\=&ACx^{2m}+\big((A+B)(C+D)-AC-BD\big)x^m+BD\end{aligned}$$

减少乘法次数进行递归。复杂度为$O(n^{\log_2 3})=O(n^{1.585})$。

分治乘法的劣势在于复杂度较高，但它不涉及除法，所以对环R没有特别的要求。

此外，当运算在模2意义下进行时，我们也可以利用位运算加速，使得算法常数减少到原来的1/32。

另一种做法是使用FFT[3]。它利用单位根的性质，实现了多项式的系数表示与点值表示之间的快速转化。时间复杂度是$O(n\log n)$。

为了使用FFT，R需具有2^k次单位根($2^k\geq 2n$)，且存在2的乘法逆元。当系数在复数域内时，单位根总能找到，但计算时容易出现精度问题。

需要注意到的是，信息学竞赛中涉及的计数问题往往要求答案模一个大质数p(如10^9+7，或$998244353=7\times 17\times 2^{23}+1$等等)后输出。这样的好处包括：每次算术运算的时间可以视作$O(1)$，不存在精度问题，且大多数情况下可以支持除法(只要在问题规模范围内，除数都远小于p，存在乘法逆元)。因此本文中假设所有运算都在$\mathbb{F}_p$下进行。

在模p意义下进行FFT时，若满足$2^k\big|\varphi(p)=p-1$，则可取p的原根g，并用$g^{\frac{p-1}{2^k}}$作为单位根；否则，将系数视作$\mathbb{Z}$的元素进行运算后再对p取模。这时，相乘得到的系数大小不超过np^2，只要取若干个便于FFT的大质数分别进行运算，再用中国剩余定理还原系数即可。

2.3 形式幂级数

一个多项式仅有有限项的系数是非零的。若去掉这一限制，可将其推广为形式幂级数

$$A(x)=\sum_{i\geq 0}a_ix^i, \tag{2.2}$$

[2]详见http://en.wikipedia.org/wiki/Karatsuba%5falgorithm

[3]详见http://en.wikipedia.org/wiki/Discrete%5fFourier%5ftransform

它们组成了形式幂级数环$R[[x]]$。

此定义中，x仅作为一个符号，而不用具体的数值代入运算，故不必考虑与幂级数敛散性有关的问题。

我们用$[x^n]A(x)$表示$A(x)$的n次项系数a_n。

形式幂级数的加减法与乘法也可与多项式运算类似定义:

$$A(x)=\sum_{i\geq 0}a_ix^i, B(x)=\sum_{i\geq 0}b_ix^i, \tag{2.3}$$

$$A(x)\pm B(x)=\sum_{i\geq 0}(a_i\pm b_i)x^i, \tag{2.4}$$

$$A(x)B(x)=\sum_{k\geq 0}\Big(\sum_{i+j=k}a_ib_j\Big)x^k. \tag{2.5}$$

实际运算时，通常只需保留次数不超过$n-1$的项进行计算，并将多余的项舍去(即在模x^n意义下计算)。因此加减法的复杂度为$O(n)$，乘法的复杂度为$O(n\log n)$。

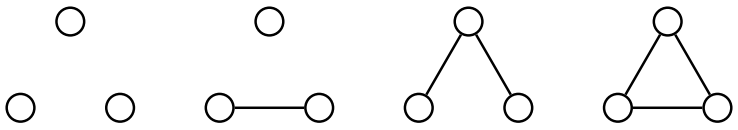
3 组合计数问题

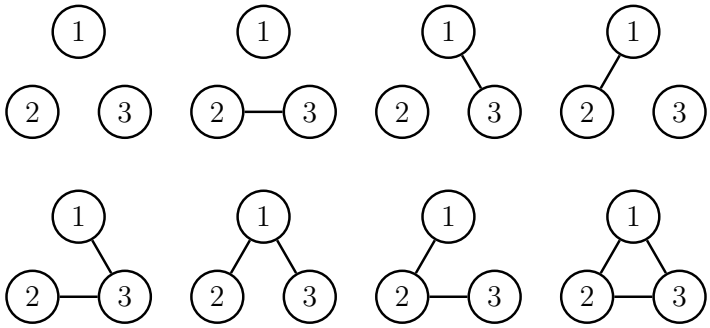
3.1 组合对象

组合计数问题是一类常见的问题。这类问题中一般定义了一类组合对象A，它可能是满足某一性质的树、图、串等对象的集合；其中每个对象$a\in A$都被定义了大小$size(a)\in\mathbb{N}$，它可能代表结点数量、序列长度等。对于某个固定的n，满足$size(a)=n$的对象a的数量是有限的，记作A_n。我们的任务通常为求出A_n的数值。

根据不同的问题要求，组合对象可以分成无标号和有标号两类。下面简单以无向图为例解释它们的区别：

n个点的标号图中，每个顶点都被赋予了$1,2,\cdots,n$中的唯一标号；然而在无标号图中，每个顶点的地位是没有区别的。如图所示，$n=3$时，无标号的简单无向图共有4种，而有标号的简单无向图共有8种。





3.2 普通生成函数

数列$A_0, A_1, \cdots$的普通生成函数(Ordinary Generating Function, OGF)定义为形式幂级数

$$A(x) = \sum_{i \geq 0} A_i x^i. \tag{3.1}$$

A是一类无标号对象，则A的普通生成函数$A(x)$定义为数列$A_0, A_1, \cdots$的普通生成函数，其中A_n为满足$size(a) = n$的对象$a \in A$的数量。

对于两类无标号对象A, B，定义一类新的对象$C = A \cup B$，若A, B交集为空，则C的生成函数即为

$$C(x) = A(x) + B(x). \tag{3.2}$$

再来考虑A, B的笛卡尔积$D = A \times B$，其中D的每个元素d都是一个二元组(a, b)，其中$a \in A, b \in B$，并定义$size(d) = size(a) + size(b)$。则有

$$D_k = \sum_{i+j=k} A_i B_j, \tag{3.3}$$

于是D的OGF即为

$$D(x) = A(x)B(x). \tag{3.4}$$

这一操作实现了A中元素和B中元素的拼接。

3.3 指数生成函数

数列$A_0, A_1, \cdots$的指数生成函数(Exponential Generating Function, EGF)定义为形式幂级数

$$A(x) = \sum_{i \geq 0} A_i \frac{x^i}{i!}. \tag{3.5}$$

指数生成函数在处理有标号问题时更加便捷。对于一类有标号对象A，A的指数生成函数$A(x)$定义为数列$A_0, A_1, \cdots$的指数生成函数，其中A_n为满足$size(a) = n$的对象$a \in A$的数量。

给定两类有标号对象A, B，对于它们的并集$C = A \cup B$，同样有

$$C(x) = A(x) + B(x). \tag{3.6}$$

现在考虑有标号对象的拼接。给定两个对象a, b，设$size(a) = n, size(b) = m$，它们分别带有$1, 2, \cdots, n$和$1, 2, \cdots, m$的标号。为将a, b拼接得到c，需给c分配$1, 2, \cdots, n + m$的标号。规定重新分配时需要保持标号的原有相对顺序，则有

$$\binom{n+m}{n} = \frac{(n+m)!}{n!\,m!} \tag{3.7}$$

种方法。

因此，若将两类带标号对象A, B拼接得到D，则有

$$D_k = \sum_{i+j=k} A_i B_j \frac{k!}{i!j!}, \tag{3.8}$$

从而D的EGF也具有

$$D(x) = A(x)B(x) \tag{3.9}$$

的形式。

4 乘法逆元

当$A(x)B(x) = 1$时，称$A(x), B(x)$互为乘法逆元，可以写作$A(x) = B(x)^{-1} = 1/B(x)$。除以一个形式幂级数，就相当于乘上它的乘法逆元。

例如，

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots. \tag{4.1}$$

根据这一等式可得，若数列$a_0, a_1, \cdots$的普通生成函数为$A(x)$，令$s_n = \sum_{i=0}^{n} a_i$，那么$s_0, s_1, \cdots$的普通生成函数为$A(x)/(1-x)$。

再结合二项式系数的递推性质，可得

$$\frac{1}{(1-x)^{n+1}} = \sum_{i \geq 0} \binom{n+i}{n} x^i. \tag{4.2}$$

接下来介绍，在给定$A(x)$时，如何求出$A(x)$的乘法逆元$B(x)$。

可以证明，$A(x)$存在乘法逆元的充要条件是$A(x)$的常数项存在乘法逆元。必要性由$\big([x^0]A(x)\big)\big([x^0]B(x)\big)=1$可得。而当$A(x)$的常数项可逆时，即可根据乘法的定义(2.5)按顺序求出$[x^1]B(x),\ [x^2]B(x),\ \cdots$的值，于是此时乘法逆元存在且唯一。同时，我们得到了一个求出乘法逆元前n项的$O(n^2)$朴素算法。

下面介绍一个用$O(n\log n)$时间计算乘法逆元的算法，它的本质是牛顿迭代法。

首先求出$A(x)$的常数项的逆元b，并令$B(x)$的初始值为b。

假设已求得满足

$$A(x)B(x)\equiv 1 \pmod{x^n} \tag{4.3}$$

的$B(x)$，则

$$\begin{aligned} A(x)B(x)-1&\equiv 0 \pmod{x^n},\\ \big(A(x)B(x)-1\big)^2&\equiv 0 \pmod{x^{2n}},\\ A(x)\big(2B(x)-B(x)^2A(x)\big)&\equiv 1 \pmod{x^{2n}}. \end{aligned} \tag{4.4}$$

我们用$O(n\log n)$时间计算出$2B(x)-B(x)^2A(x)$，并将它赋值给$B(x)$进行下一轮迭代。每迭代一次，$B(x)$的有效项数n都会增加一倍。于是该算法的时间复杂度为

$$T(n)=T(n/2)+O(n\log n)=O(n\log n).$$

5 乘法逆元的应用

例题1 (序列计数). 你有若干种颜色不同的骨牌，其中大小为$1\times i$的骨牌共有a_i种。每种骨牌都可以无限量使用。用骨牌不重叠地铺满一排$1\times n$的方格，共有几种方法?$(a_i,n\le 10^5)$

我们枚举使用的骨牌数量k。设$A(x)=\sum_{i\ge 0}a_ix^i$，则根据生成函数的乘法的意义，容易知道此时的答案为$[x^n]A(x)^k$。所以总方法数目为

$$[x^n]\sum_{k\ge 0}A(x)^k=[x^n]\frac{1}{1-A(x)}. \tag{5.1}$$

于是只需计算$1-A(x)$的乘法逆元即可，时间复杂度是$O(n\log n)$。

一般地，对于一类组合对象A，由A的元素组成的序列定义一类新的组合对象B，则B的生成函数为

$$B(x)=\frac{1}{1-A(x)}. \tag{5.2}$$

这一结论对于无标号(OGF)、有标号(EGF)的情况都成立。

例题2 (预处理Bernoulli数[4]). 对于所有$0\le i\le n-1$求出B_i。($n\le 10^5$)

Bernoulli数$B_0,B_1,\cdots$的指数生成函数为

$$\begin{aligned}B(x)&=\sum_{i\ge 0}B_i\frac{x^i}{i!}\\&=\frac{x}{e^x-1}\\&=\Big(\sum_{i\ge 0}\frac{x^i}{(i+1)!}\Big)^{-1}.\end{aligned} \tag{5.3}$$

这样，只要求一次乘法逆元即可。

例题3. 字符集大小为m。给定一个长为k的字符串s，求出所有长为n的串中，不包含子串s的共有几个。($n,m,k\le 10^5$)

假定串的下标都从1开始。

考虑DP，用f_i表示有多少种方法填入前i个字符，使得第一次匹配上串s的位置是$[i-k+1,i]$。当$i<k$时，$f_i=0$；当$i\ge k$时，

$$f_i=m^{i-k}-\sum_{0\le j\le i-k}f_jm^{i-k-j}-\sum_{d>0,d\in C}f_{i-d}, \tag{5.4}$$

其中集合C定义为$\big\{d\ge 0\big|s[d+1,k]=s[1,k-d]\big\}$。

这个DP方程的含义是，末k位字符与串s相同，前$i-k$位字符可以任意确定，但为了保证$i-k+1$是第一次匹配上的位置，需要从中减去之前已经匹配过的情况。减去的第一项是这次匹配与上一次没有重叠的情况，第二项是与上次出现重叠的情况，为此需要先用KMP算法求出C集合。

为了优化这个DP，考虑f的生成函数$f(x)=\sum_{i\ge 0}f_ix^i$。令$C(x)=\sum_{d\in C}x^d$，则可由DP方程写出

$$f(x)=\frac{x^k}{1-mx}-\frac{x^kf(x)}{1-mx}-f(x)\big(c(x)-1\big),$$

[4]定义见http://en.wikipedia.org/wiki/Bernoulli%5fnumber

即

$$f(x)=\frac{x^k}{x^k+(1-mx)c(x)}.$$

最后的答案即为$g_n=m^n-\sum_{i\geq 0}f_im^{i-j}$，而相应的生成函数为

$$g(x)=\frac{c(x)}{x^k+(1-mx)c(x)}. \tag{5.5}$$

时间复杂度是$O(n\log n)$。

6 对数与指数运算

6.1 复合运算

首先引入形式幂级数的复合运算。

设$A(w)=\sum_{i\geq 0}a_iw^i, B(x)=\sum_{i\geq 1}b_ix^i$，则$B(x)$与$A(w)$的复合为

$$C(x)=A\circ B(x)=A\big(B(x)\big)=\sum_{i\geq 0}a_i\big(B(x)\big)^i, \tag{6.1}$$

可以将其整理为$C(x)=\sum_{i\geq 0}c_ix^i$ 的形式。

由于已假定$B(x)$没有常数项，因此即使A有无限多个非零项，$B(x)$与$A(w)$的复合仍然可以定义。

6.2 形式导数

对于$A(x)=\sum_{i\geq 0}a_ix^i$，定义$A(x)$的形式导数为

$$A'(x)=\sum_{i\geq 1}ia_ix^{i-1}. \tag{6.2}$$

容易验证

$$\big(cA(x)\big)'=cA'(x), \tag{6.3}$$

$$\big(A(x)\pm B(x)\big)'=A'(x)\pm B'(x), \tag{6.4}$$

$$\big(A(x)B(x)\big)'=A'(x)B(x)+A(x)B'(x), \tag{6.5}$$

$$\left(\frac{1}{A(x)}\right)'=-\frac{A'(x)}{A(x)^2}, \tag{6.6}$$

$$\Big(A\big(B(x)\big)\Big)'=A'\big(B(x)\big)B'(x) \tag{6.7}$$

这些基本求导法则对于形式导数依然成立。

6.3 对数函数与指数函数

我们可以定义形式幂级数的对数与指数函数，这相当于将给定的级数$A(x)=\sum_{i\geq 1}a_ix^i$与对应的麦克劳林级数复合，即

$$\ln\big(1-A(x)\big)=-\sum_{i\geq 1}\frac{A(x)^i}{i}, \tag{6.8}$$

$$\exp\big(A(x)\big)=\sum_{i\geq 0}\frac{A(x)^i}{i!}. \tag{6.9}$$

容易验证，对数和指数函数的大多数性质在这里仍然成立。

6.3.1 对数函数的计算

给定$A(x)=1+\sum_{i\geq 1}a_ix^i$，令

$$B(x)=\ln\big(A(x)\big),$$

则

$$B'(x)=\frac{A'(x)}{A(x)}, \tag{6.10}$$

于是只需求出$A(x)$的乘法逆元，就可求出$\ln\big(A(x)\big)$，时间复杂度是$O(n\log n)$。

6.3.2 指数函数的计算

给定$A(x)=\sum_{i\geq 1}a_ix^i$，令

$$B(x)=\exp\big(A(x)\big)=\sum_{i\geq 0}b_ix^i,$$

则

$$B'(x)=B(x)A'(x), \tag{6.11}$$

比较系数得

$$\begin{aligned} b_0&=1,\\ b_i&=\frac{1}{i}\sum_{1\leq k\leq i}ka_kb_{i-k}\quad(i\geq 1). \end{aligned} \tag{6.12}$$

用分治法计算所有b_i，每次用FFT计算左半边的b_i对右半边所产生的贡献，时间复杂度是$O(n\log^2 n)$。

下一节的牛顿迭代法可以在$O(n\log n)$时间计算$\exp\big(A(x)\big)$。

6.4 牛顿迭代法

令$f(x)=e^{A(x)}$，可得到一个关于$f(x)$的方程

$$g\big(f(x)\big)=\ln\big(f(x)\big)-A(x)=0. \tag{6.13}$$

考虑用牛顿迭代法解这一方程[5]。首先$f(x)$的常数项是容易确定的。设已求得$f(x)$的前n项$f_0(x)$，即

$$f(x)\equiv f_0(x)\pmod{x^n}, \tag{6.14}$$

作泰勒展开，得

$$\begin{aligned}0&=g\big(f(x)\big)\\&=g\big(f_0(x)\big)+g'\big(f_0(x)\big)\big(f(x)-f_0(x)\big)+\frac{g''\big(f_0(x)\big)}{2}\big(f(x)-f_0(x)\big)^2+\cdots\\&\equiv g\big(f_0(x)\big)+g'\big(f_0(x)\big)\big(f(x)-f_0(x)\big)\pmod{x^{2n}},\end{aligned}$$

即

$$f(x)\equiv f_0(x)-\frac{g\big(f_0(x)\big)}{g'\big(f_0(x)\big)}\pmod{x^{2n}}, \tag{6.15}$$

将(6.13)代入，得

$$\begin{aligned}f(x)&=f_0(x)-\frac{\ln\big(f_0(x)\big)-A(x)}{1/f_0(x)}\\&=f_0(x)\Big(1-\ln\big(f_0(x)\big)+A(x)\Big).\end{aligned} \tag{6.16}$$

按此迭代，复杂度$T(n)=T(n/2)+O(n\log n)=O(n\log n)$。

[5]一般情况下，方程$g\big(f(x)\big)$有解需要g满足一定条件。但在这里并没有问题，因此不深入讨论。

6.5 k次幂的计算

例题4 (k次幂的计算). 给定多项式$f(x)$和正整数k，求出$f(x)^k$的前n项系数。$(n, k \leq 10^5)$

普通的做法是利用倍增快速幂。每次用FFT将两个多项式相乘，并舍去多余项，时间是$O(n \log n)$。共做$O(\log k)$次，总时间$O(n \log n \log k)$。

根据指数函数和对数函数的性质，当$f(x)$的常数项为1时，有

$$f(x)^k = \exp\big(k \ln f(x)\big). \tag{6.17}$$

按此式计算，时间复杂度是$O(n \log n)$。

如果$f(x)$的常数项不为1，设$f(x)$的最低次项为ax^d，则

$$f(x)^k = a^k x^{kd} \left(\frac{f(x)}{ax^d}\right)^k,$$

于是转化成了常数项为1的情况，可以按照(6.17)计算。

我们也可以计算$f(x)$的k次方根。这时若$f(x)$的最低次项为ax^d，则需要保证$k|d$，且a存在k次方根。然后即可将常数项调整为1后按照上面的方法计算。

7 集合的计数

7.1 有标号集合的计数

例题1中已讨论了无标号对象所组成的序列的计数，且得到的结论对有标号对象也成立。下面考虑有标号对象组成的集合的计数。

集合与序列的区别在于集合内的元素没有顺序关系，需要将重复的统计去除。因此，A的元素组成的集合的EGF为

$$B(x) = \sum_{i \geq 0} \frac{A(x)^i}{i!} = e^{A(x)}. \tag{7.1}$$

例题5 (连通图计数). 求出n个顶点的连通无向图的个数。顶点有标号，不允许重边和自环。$(n \leq 10^5)$

设G是所有简单无向图，则G的EGF为

$$G(x) = \sum_{n \geq 0} 2^{\binom{n}{2}} \frac{x^n}{n!}. \tag{7.2}$$

令C为所有简单连通图。由于一个简单无向图可以看做若干个连通分量组成的集合，于是

$$G(x) = e^{C(x)},$$

从而

$$C(x) = \ln G(x). \tag{7.3}$$

只要求出$G(x)$的对数即可求得答案。时间复杂度为$O(n \log n)$。

例题6 (有限制的置换计数). 给定集合S和正整数n，计算有多少个n阶置换p，满足p分解后的每一个轮换的大小都在S内。($n \leq 10^5$)

一个置换是若干个轮换组成的集合，每个轮换相当于一个带标号的环。根据圆排列的计数，大小为k的轮换的数量有$(k-1)!$个，对应的EGF为

$$\frac{(k-1)!}{k!}x^k = \frac{x^k}{k}, \tag{7.4}$$

因此答案的EGF即为

$$\exp\Big(\sum_{k\in S}\frac{x^k}{k}\Big). \tag{7.5}$$

7.2 无标号集合的计数

例题7 (完全背包计数). 你有若干种不同的物品，其中体积为i的物品有a_i种。每种物品有无限个。求选取物品恰好装满总体积为n的背包的方案数。($a_i, n \leq 10^5$)

在本题中，无标号集合里可能存在重复元素，因此不能简单地除以$k!$来去重。

令$A(x) = \sum_{i\geq 1} a_i x^i$，考虑对每一种体积$i$写出生成函数，并相乘，进一步

用对数函数化乘为加，由此得到OGF为

$$\begin{aligned}
&\prod_{1\le i\le n}(1+x^i+x^{2i}+\cdots)^{a_i}\\
=&\prod_{1\le i\le n}(\frac{1}{1-x^i})^{a_i}\\
=&\exp\Big(-\sum_{1\le i\le n}a_i\ln(1-x^i)\Big)\\
=&\exp\Big(\sum_{1\le i\le n}a_i\sum_{j\ge 1}\frac{x^{ij}}{j}\Big)\\
=&\exp\Big(\sum_{j\ge 1}\frac{1}{j}A(x^j)\Big),
\end{aligned}\tag{7.6}$$

注意到$A(x^j)$中只有n/j项是有用的，故可用

$$n+\frac{n}{2}+\frac{n}{3}+\cdots+1=O(n\log n)$$

的时间对所有$A(x^j)$求和。

最后再求一次exp，总复杂度为$O(n\log n)$。[6]

例题8 (01背包计数). *你有若干种不同的物品，其中体积为i的物品有a_i种。每种物品仅有1个。求选取物品恰好装满总体积为n的背包的方案数。*$(a_i, n\le 10^5)$

本题和上题的处理方法相同，这里不再赘述。

8 环的计数

这一节中考虑环的计数。环的特别之处在于它可以旋转，两个旋转后相同的环即被认为是同一个环。

8.1 有标号环的计数

比较容易处理的是有标号环的计数。根据圆排列公式，k元环的数量有$(k-1)!$个。因此，对于一类组合对象A，它的元素所组成的环(可旋转而不可

[6]当a_i均为1时，本题即为整数无序拆分问题，可以使用欧拉五边形数定理更方便地解决，详见http://en.wikipedia.org/wiki/Pentagonal%5fnumber%5ftheorem

翻转)的EGF为

$$\sum_{k\geq 1}\frac{A(x)^k}{k}=-\ln\big(1-A(x)\big). \tag{8.1}$$

例题9. *求包含n个顶点，n条边的连通无向图有几个。顶点有标号。不允许重边和自环。($n\leq 10^5$)*

由基本图论知识可知，这样的图包含一个长度不小于3的环，且环上的每个结点是一棵有根树的根。

先来解决树的问题。这里考虑的树是带标号有根树，且结点的儿子没有顺序。设有根树的EGF为$T(x)$。由Cayley定理[7] 可知，n个结点的有根树数量为n^{n-1}，因此[8]

$$T(x)=\sum_{n\geq 1}\frac{n^{n-1}x^n}{n!}. \tag{8.2}$$

再来考虑环。需要注意本题中的环是没有方向的(可以翻转)，于是由树组成的不小于3的环的EGF 为

$$\begin{aligned}&\frac{1}{2}\sum_{k\geq 3}\frac{T(x)^k}{k}\\ =&-\frac{1}{2}\ln\big(1-T(x)\big)-\frac{T(x)}{2}-\frac{T(x)^2}{4}.\end{aligned} \tag{8.3}$$

8.2 无标号环的计数

例题10 (无标号环的计数). *你有若干种颜色不同的珠子，其中重量为i的珠子共有a_i种。每种珠子都可以无限量使用。用珠子串成一串重量为n的项链，共有几种方法？项链允许旋转，但不允许翻转。($a_i, n\leq 10^5$)*

无标号环的处理稍微复杂一些，因为环中允许出现相同元素，可能产生循环节。我们容易想到利用Pólya计数公式[9] 来解决这一问题。

[7]详见http://en.wikipedia.org/wiki/Cayley%27s%5fformula

[8]也可以根据树的定义得到$T(x)=xe^{T(x)}$，再由拉格朗日反演推出这一结论

[9]详见http://en.wikipedia.org/wiki/P%C3%B3lya%5fenumeration%5ftheorem

给定一类组合对象A，由k个A中元素组成的环的OGF应当是

$$\begin{aligned}&\frac{1}{k}\sum_{0\le i<k}A(x^{k/\gcd(i,k)})^{\gcd(i,k)}\\=&\frac{1}{k}\sum_{d|k}\varphi(d)A(x^d)^{k/d},\end{aligned}$$

于是A中元素组成的环的OGF为

$$\begin{aligned}&\sum_{k\ge1}\frac{1}{k}\sum_{d|k}\varphi(d)A(x^d)^{k/d}\\=&\sum_{d\ge1}\varphi(d)\sum_{k\ge1}\frac{1}{kd}A(x^d)^k\\=&-\sum_{d\ge1}\frac{\varphi(d)}{d}\ln\big(1-A(x^d)\big).\end{aligned}\tag{8.4}$$

与例题7类似，$\ln\big(1-A(x^d)\big)$中只有n/d项是有用的，因此只需求出$\ln\big(1-A(x)\big)$后，再用$O(n\log n)$时间求和即可。总时间为$O(n\log n)$。

9 复合运算

9.1 复合与复合逆

在6.1节中，我们介绍了两个形式幂级数的复合运算。

设$f(w)=\sum_{i\ge1}f_iw^i,g(x)=\sum_{i\ge1}g_ix^i$，如果$f\big(g(x)\big)=x$，那么称$f(w),g(x)$互为复合逆(反函数)。同时我们可以得到$f_1g_1=1,g\big(f(w)\big)=w$。

设$f(w),g(x)$互为复合逆。对于$f(w)$，如果存在一个算法，对任何给定的$p(x)$都高效地求出$f\big(p(x)\big)$，则可利用6.4节中的牛顿迭代法，对于任何给定的$q(w)$求出$g\big(q(w)\big)$。若取$q(w)=w$，即可求出$g(x)$。

给定$f(w),g(x)$，根据定义直接利用FFT计算$f\big(g(x)\big)$的前n项系数，时间复杂度为$O(n^2\log n)$。下面介绍一个用$O(n^2)$时间计算$f\big(g(x)\big)$的算法，它的核心是小步大步思想：

令$d=\lceil\sqrt{n}\rceil$，先求出$g(x)^2,g(x)^3,\cdots,g(x)^d$，复杂度为$O(dn\log n)$。再利用已求出的$g(x)^d$，算出$g(x)^{2d},\cdots,g(x)^{(d-1)d}$，复杂度也是$O(dn\log n)$。

接下来，根据

$$f\big(g(x)\big) = \sum_{0\le i<d} g(x)^{id} \sum_{0\le j<d} f_{id+j} g(x)^j, \tag{9.1}$$

并代入之前求得的结果，即可计算。总时间为$O(n\sqrt{n}\log n + n^2) = O(n^2)$。

参考文献[1]中给出了一个时间复杂度为$O\left((n\log n)^{3/2}\right)$的算法计算$f\big(g(x)\big)$，感兴趣的读者可自行参看。

9.2 Lagrange反演

Lagrange反演公式[10]：若$f(w)$是$g(x)$的复合逆，则

$$[x^n]g(x) = \frac{1}{n}[w^{n-1}]\Big(\frac{w}{f(w)}\Big)^n. \tag{9.2}$$

它具有一个推广形式

$$[x^n]h\big(g(x)\big) = \frac{1}{n}[w^{n-1}]h'(w)\Big(\frac{w}{f(w)}\Big)^n, \tag{9.3}$$

其中取$h(w) = w$即为(9.2)式。

求出$g(x)$的复合逆的复杂度较高，但根据(9.2)式与例题4中的方法，可用$O(n\log n)$时间求出复合逆中的某一项系数。

例题11. 给定整数n和集合$S(1 \notin S)$，求出含有n个结点，且每个非叶结点u的儿子数量$\deg(u) \in S$的有根多叉树的数量。树的结点无标号，结点的孩子有顺序。$(n \le 10^5)$

令这些树的OGF为$T(x)$。根据定义，一棵树可以是单个叶子结点，或者是一个非叶节点拼上$s(s \in S)$棵子树组成的序列，于是

$$T(x) = x + \sum_{s\in S} T(x)^s, \tag{9.4}$$

可以将其写成$x = f\big(T(x)\big)$的形式，其中$f(w)$是一个一次项系数为1的多项式。

令$g(x)$为$f(w)$的复合逆，则有$T(x) = g(x)$，于是用Lagrange反演求出其n次项系数即可。

[10]证明可参考http://www.math.msu.edu/%7Emagyar/Math880/Lagrange.pdf

例题12 (带标号的边双连通图计数). 求出n阶边双连通图[11]的个数。顶点有标号，不允许重边和自环。($n \le 10^5$)

令$C(x)$为所有连通无向图的EGF，$B(x)$为其中所有边双连通图的EGF。例题5中已求出了$C(x)$，现在的任务是计算$B(x)$的x^n项系数。

考虑这样的二元组(G, v)，其中G是一个图，v是G中的某个具有特殊地位的顶点，不妨将此二元组称作有根图。一个图G_0共可产生$|G_0|$个有根图，从而所有连通图产生的有根图的EGF为$xC'(x)$。

给定一个有根图(G_C, v)，可将G_C缩成一棵BCC树T，树上每个结点代表原图中的一个BCC(边双连通分量)。记根结点v所在的BCC为B_v，将B_v定为树T的根。若B_v包含的顶点数量为n，则有根图(B_v, v)的EGF为$nb_nx^n/n!$。

在T中，根B_v可能有若干个(或0个)子树。设T_1是其中某个子树，这个子树也是某个连通图G_1缩成的BCC树。连通图G_1中的某点u与根B_v中的某一点w有连边。有根图(G_1, u)的EGF为$xC'(x)$，又由于点w有$|B_v| = n$种选择，所以一个子树的EGF是$nxC'(x)$，从而若干个子树的EGF就是$e^{nxC'(x)}$。

枚举B_v的大小n，有

$$xC'(x) = \sum_{n\ge 1} \frac{b_n}{(n-1)!} x^n e^{nxC'(x)}. \tag{9.5}$$

简记$C_1(x) = xC'(x), B_1(w) = wB'(w)$，则有

$$C_1(x) = B_1(xe^{C_1(x)}). \tag{9.6}$$

其中，$C_1(x), xe^{C_1(x)}$都是容易计算的。需要计算的是$B_1(w)$的某一项系数。于是问题转化为，已知$P(x), Q(x)$，求出满足$P(x) = R\big(Q(x)\big)$的$R(w)$的n次项系数$[w^n]R(w)$。用$Q^{(-1)}(w)$表示$Q(x)$的复合逆，则有$R(w) = P\big(Q^{(-1)}(w)\big)$。根据Lagrange反演公式的推广形式，得

$$[w^n]R(w) = \frac{1}{n}[x^{n-1}]P'(x)\Big(\frac{x}{Q(x)}\Big)^n,$$

故可用$O(n \log n)$时间计算答案。

[11]删去任意一条边后仍然连通的无向图

10 二元生成函数

有时，我们会在生成函数中引入第二个变量，并用它做一些附加标记。

例题13 (连通图计数II). 求出含有n个顶点，m条边的连通无向图的个数。顶点有标号，不允许重边和自环。$(n, m \leq 1000)$

引入字母y来标记边，即用$x^n y^m/n!$表示一个n个点，m条边的图的EGF，则所有无向图的EGF为

$$G(x,y) = \sum_{n\geq 0} \frac{x^n}{n!}(1+y)^{\binom{n}{2}}. \tag{10.1}$$

与例题5类似，需求出$G(x,y)$的对数。这里涉及到了二元多项式的乘法和乘法逆元。

为了将两个二元多项式相乘，可将系数排成矩阵后，对每一行做DFT，再对每一列做DFT，这样便能得到二元多项式的点值表示，然后将点值相乘再转回系数即可。求乘法逆元的过程与一元情况大同小异。总复杂度是$O\big(nm(\log n + \log m)\big)$。

例题14. 给定一个多项式$f(x)$，和一个数k。对于所有$1 \leq i \leq n$，求出$[x^k]f(x)^i$。$(n, k \leq 3000)$

假设$k = O(n)$。利用一个与9.1节中类似的小步大步算法，可以做到$O(n^2)$的复杂度。

现从另一个角度考虑。引入字母u，则所求的答案序列为下式中的x^k项系数：

$$1 + uf(x) + u^2 f(x)^2 + \cdots = \frac{1}{1-uf(x)}, \tag{10.2}$$

这是一个关于u的幂级数。

为了提取x^k项系数，考虑Lagrange反演。最简单的情况是当$[x^0]f(x) = 0, [x^1]f(x) \neq 0$时，$f(x)$存在复合逆$g(w)$，于是

$$[x^k]\frac{1}{1-uf(x)} = \frac{1}{k}[w^{k-1}]\frac{u}{(1-uw)^2}\Big(\frac{w}{g(w)}\Big)^k. \tag{10.3}$$

分母可以用(4.2)展开。由此提示了这个问题与求解$f(x)$的复合逆可在$O(n\log n)$时间内相互转化。

另外，如果$f(x)$不存在复合逆，则还需做一些处理：

- 如果$f(x)$常数项为0，但最低次项$f_t x^t$的次数$t \geq 2$，那么对$f(x)/f_t$开t次方根表示成$f(x) = f_t \cdot \big(p(x)\big)^t$，那么$p(x)$的最低次项次数为1，可以使用上述方法。

- 如果$f(x)$的常数项是$c \neq 0$，则可将其写成$f(x) = c + q(x)$。由此可先求出所有$[x^k]q(x)^i$，再根据

$$\begin{aligned}[x^k]\big(c+q(x)\big)^i &= [x^k]\sum_{0\leq j\leq i}\binom{i}{j}c^j q(x)^{i-j}\\ &= i!\sum_{0\leq j\leq i}\frac{c^j}{j!}\frac{[x^k]q(x)^{i-j}}{(i-j)!},\end{aligned}$$

  用FFT求一次卷积即可。

只要能求出$f(x)$的复合逆，就可以再用$O(n\log n)$时间解决本题。一般情况下，求复合逆较难达到优秀的复杂度。但在某些特殊情况下，$f(x)$的复合逆较易求得，此时便可使用这一方法。

例题15 (预处理第一类Stirling数[12]). 给定n，求出所有$\begin{bmatrix}n\\k\end{bmatrix}, 0 \leq k \leq n$。($n \leq 10^5$)

传统做法是利用

$$\sum_{k=0}^{n}\begin{bmatrix}n\\k\end{bmatrix}x^k = x(x+1)(x+2)\cdots(x+n-1) \tag{10.4}$$

进行分治，每次将因式均等的分成两半分别计算后用FFT乘起来，复杂度是$O(n\log^2 n)$。

另外一个做法利用了

$$\begin{bmatrix}n\\k\end{bmatrix} = \frac{n!}{k!}[x^n]\big(-\ln(1-x)\big)^k. \tag{10.5}$$

根据此式套用刚才的方法即可，复杂度$O(n\log n)$。然而该算法的常数略大。

[12] $\begin{bmatrix}n\\k\end{bmatrix}$表示将$n$个元素划分成$k$个环的方案数量。

11 结语

本文介绍了形式幂级数的乘法逆元、对数、指数等运算的$O(n\log n)$算法，并以串、树、图、集合等各类组合模型为例，在计数问题中加以应用。对于复合逆的求解，本文也介绍了用$O(n\log n)$时间计算其某一项系数的算法。一些原本需要$O(n^2)$或$O(n\log^2 n)$解决的计数问题，可以用本文中的方法优化至$O(n\log n)$，但有时算法的常数较大，在实现时需要注意。

可以预见到，这一类组合计数问题会在今后的信息学竞赛中继续出现。但仍有许多有趣的问题并未在本文中涉及到，还有待我们进一步研究和探索。

感谢

感谢中国计算机学会提供学习和交流的平台。
感谢教练徐先友老师对我的指导。
感谢彭雨翔、吕凯风、毛啸等同学给我的启发。
感谢徐寅展同学为本文审稿。

参考文献

[1] Brent R P, Kung H T. Fast algorithms for manipulating formal power series[J]. Journal of the ACM (JACM), 1978, 25(4): 581-595.

[2] Flajolet P, Sedgewick R. Analytic combinatorics[M]. Cambridge University Press, 2009.

[3] 彭雨翔, Inverse Element of Polynomial.

[4] 彭雨翔, Fast Fourier Transform Modulo Prime.

[5] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

ydc的奖金命题报告

雅礼中学 刘剑成

1 试题大意

有一场比赛，如果你能在这场比赛中获得第i名，那么你可以得到p_i的奖金。

现在你已经提前了解了其他$n-1$名选手与你自己的各项能力，并且根据他们的能力分别计算出了他们得分的概率。这场考试的满分为1分，最低为0分，分数可以为任意小数。对于第i个人，他得x分的概率，与t_i次函数$f_i(x)$成正比。

如果一个人的函数为$f(x)=2$，那么他获得任意分数的概率都是相等的；如果一个人的函数为$f(x)=2-x$，那么他获得越高的分的概率就越低，且他获得0分的概率是获得1分的概率两倍。

现在你需要计算你能在这场比赛中期望能得到多少的奖金。由于分数可以为小数，所以无需考虑排名相等的情况。

答案对998244353（$7\times17\times2^{23}+1$，一个质数）取模。

时间限制：清澄 8sec/UOJ 6sec

空间限制：64MB

2 输入格式

输入共$n+2$行。

第一行包含一个整数n。

在第二行有n个整数，第i个数代表p_i。

接下来$n-1$行，每行第一个数为t_i，代表第i个人所对应的函数是一个t_i-1次函数，接下来t_i个实数，第j个数代表该函数中x^{j-1}项的系数。

最后一行，第一个数为t_n，代表你所对应的函数是一个t_n-1次函数，接下来t_n个实数，第j个数代表该函数中x^{j-1}项的系数。

3 输出格式

输出1行，包含一个整数，表示你期望能获得的奖金。

4 数据范围

<table>
<tr><th>编号</th><th>n</th><th>$S = \sum t_i$</th><th>特殊限制</th></tr>
<tr><td>1</td><td rowspan="2">2</td><td>5</td><td rowspan="10">无</td></tr>
<tr><td>2</td><td>10</td></tr>
<tr><td>3</td><td rowspan="4">10</td><td>40</td></tr>
<tr><td>4</td><td>60</td></tr>
<tr><td>5</td><td>80</td></tr>
<tr><td>6</td><td>100</td></tr>
<tr><td>7</td><td rowspan="4">100</td><td>400</td></tr>
<tr><td>8</td><td>600</td></tr>
<tr><td>9</td><td>800</td></tr>
<tr><td>10</td><td>1000</td></tr>
<tr><td>11</td><td rowspan="10">500</td><td rowspan="4">4000</td><td>所有名次的奖金相等</td></tr>
<tr><td>12</td><td>所有人的函数相同</td></tr>
<tr><td>13</td><td rowspan="2">除了第一名与最后一名
其他人的奖金都相等</td></tr>
<tr><td>14</td></tr>
<tr><td>15</td><td>1500</td><td rowspan="6">无</td></tr>
<tr><td>16</td><td>2000</td></tr>
<tr><td>17</td><td>2500</td></tr>
<tr><td>18</td><td>3000</td></tr>
<tr><td>19</td><td>3500</td></tr>
<tr><td>20</td><td>4000</td></tr>
</table>

保证所有人的函数在$[0, 1]$的范围以内大于等于0，输入的所有实数仅有2位小数，且除了常数项以外的系数绝对值均小于5，常数项的绝对值小于30。

所有数据均为随机生成。

p_i的范围在0到10000之间。

5 算法介绍

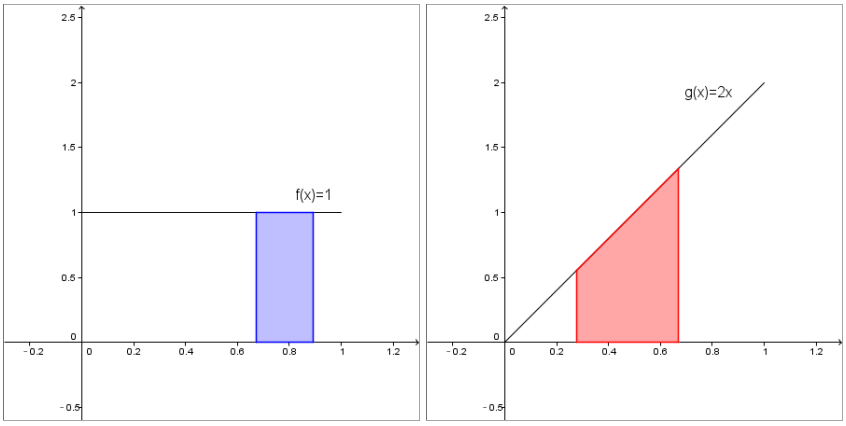
5.1 算法1

在10%的数据中，只有2个人，即我们只需要求出第一个人有多大概率在第二个人以下，再乘上获得的奖金即可。

对于第i个人，他的分数落在$[l,r]$以内的概率$P_i(l,r)$为：

$$P_i(l,r)=\frac{\int_l^r f_i(x)\,dx}{\int_0^1 f_i(x)\,dx} \tag{1}$$

直观上看，该结果就是函数在$[l,r]$这一段中与x轴所构成区域的面积，除以函数在$[0,1]$中与x轴所构成区域的面积。即函数在$[l,r]$这一段的定积分除以在$[0,1]$这一段的定积分，如下图所示：



染色区域的面积除以函数在$[0,1]$的总面积即为分数落在左右边界内部的概率

由于函数乘上一个常数对答案并没有任何影响，则我们可以将所有的函数除以它在$[0,1]$中与x轴所构成区域的面积，则有：

$$P_i(l,r)=\int_l^r f_i(x)\,dx \tag{2}$$

假设答案不需要对998244353取模，考虑一种并不完美的做法：

确定一个很小的距离ϵ，将每ϵ的距离分为一段，枚举你所获得的分数落在哪一段内，通过计算另一个人的分数在你之下的概率，求和即为你超过另一个

人的概率P：

$$P \approx \sum_{k\in Z^+}^{k\epsilon\le 1} P_1(0,k\epsilon-\epsilon)P_2(k\epsilon-\epsilon,k\epsilon) \tag{3}$$

$$= \sum_{k\in Z^+}^{k\epsilon\le 1} \int_0^{k\epsilon-\epsilon} f_1(x)\,dx \int_{k\epsilon-\epsilon}^{k\epsilon} f_2(x)\,dx \tag{4}$$

可以发现，当d取到无限接近于0的时候，$\int_{k\epsilon-\epsilon}^{k\epsilon} f_2(x)\,dx$等于$f_2(x)$的不定积分在$k\epsilon$处的导数，即为$f_2(k\epsilon)$。

记$f_i(x)$的原函数中，满足$F_i(0)=0$的原函数为$F_i(x)$，则有：

$$P = \int_0^1 F_1(x)f_2(x)dx \tag{5}$$

由于本题中函数均为多项式函数，则对于函数：

$$f(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_Nx^N$$

它的原函数为：

$$F(x) = a_0x + \frac{a_1}{2}x^2 + \frac{a_2}{3}x^3 + ... + \frac{a_N}{N+1}x^{N+1}$$

所以我们只需要求出$F_1(x)f_2(x)$的值，再求一次定积分即可。

时间复杂度：$O(S^2)$

空间复杂度：$O(S)$

5.2 算法2

对于接下来20%的数据，有$n=10$，那么考虑枚举哪些人分数比你高，哪些人分数比你低。

定义$U=\{1,2,3,...,n-1\}$，假设所有分数比你高的人构成了集合A，则分数比你低的人构成了集合$\complement_U A$，套用**算法1**的计算方法，则有：

$$P = \int_0^1 f_n(x)\prod_i^{i\in A} F_i(x) \prod_i^{i\in \complement_U A} [1-F_i(x)]\,dx \tag{6}$$

接下来，只需统计当前超过了多少人，计入答案即可。

时间复杂度：$O(2^nS^2)$

空间复杂度：$O(S)$

5.3 算法3

我们观察获得第i名的总概率：

$$P_i = \sum_{A\subseteq\{1,2,\ldots,n-1\}}^{|A|=i} \int_0^1 \prod_i^{i\in A} F_i(x) \prod_i^{i\in \complement_U A} [1-F_i(x)] f_n(x)\,dx \tag{7}$$

$$P_i = \int_0^1 f_n(x) \sum_{A\subseteq\{1,2,\ldots,n-1\}}^{|A|=i} \prod_i^{i\in A} F_i(x) \prod_i^{i\in \complement_U A} [1-F_i(x)]\,dx \tag{8}$$

即我们可以先求出：

$$\sum_{A\subseteq\{1,2,\ldots,n-1\}}^{|A|=i} \prod_i^{i\in A} F_i(x) \prod_i^{i\in \complement_U A} [1-F_i(x)]$$

接下来乘上$f_n(x)$再积分即为获得第i名的概率。

考虑动态规划，我们依次枚举第i个函数，判断它是乘上$F_i(x)$还是$1-F_i(x)$。可以设出状态$g_{i,j}(x)$代表已经到了第i个人，且有j个人的分数比你低的时候，当前的多项式为$g_{i,j}(x)$，

则有递推式：

$$g_{i,j}(x) = F_i(x)g_{i-1,j-1}(x) + [1-F_i(x)]g_{i-1,j}(x) \tag{9}$$

接下来，我们只需要暴力进行多项式乘法，递推即可。使用滚动数组可以将空间优化至$O(nS)$。

时间复杂度：$O(nS^2)$

空间复杂度：$O(nS)$

5.4 Bonus数据

5.4.1 数据11

当所有名次的奖金相等时，无论获得任意名次都必然获得p的奖金，所以可以直接输出p。

时间复杂度：$O(1)$

空间复杂度：$O(1)$

5.4.2 数据12

当所有人的概率函数相等时，每个人获得第i名的概率是相同的，所以我们只需输出$\sum_{i=1}^{n}\frac{p_i}{n}$即可。

时间复杂度：$O(n)$
空间复杂度：$O(1)$

5.4.3 数据13-14

对于数据点13和14，我们只需要计算出获得第一名与最后一名的概率，再乘上他们奖金与2到$n-1$名奖金的差值即为答案。

获得第一名的概率为：

$$P_1 = \int_0^1 \prod_{i=1}^{n-1} F_i(x)\, f_n(x)\, dx \tag{10}$$

获得最后一名的概率为：

$$P_n = \int_0^1 \prod_{i=1}^{n-1} [1 - F_i(x)]\, f_n(x)\, dx \tag{11}$$

我们只需要直接暴力计算多项式乘法即可。

时间复杂度：$O(S^2)$
空间复杂度：$O(S)$

5.4.4 一个更优的解法

可以发现，在求第一名与最后一名的概率时，所需要做的即为求n个多项式的乘积。由于本题中模数的特殊性，很容易想到使用快速傅里叶变换（FFT）优化乘法。

对于所有的i满足$1 \le i < n$，将$F_i(x)$进行离散傅里叶变换（DFT），即用一个数组记录当x分别为$\{\omega_L^0, \omega_L^1, \omega_L^2, ..., \omega_L^{L-1}\}$[1]时，函数的值分别为多少。

如：新的A_k等于原A函数$A(\omega_L^k)$的值；新的$F_{i,k}$等于原F_i函数$F_i(\omega_L^k)$的值。

[1] ω_L^k指第k个L次单位复根；L为2的若干次幂

对于多项式$C(x) = A(x) + B(x)$，在进行DFT之后有：

$$C_i = A_i + B_i$$

同理，对于$C(x) = A(x) \times B(x)$，在进行DFT之后有：

$$C_i = A_i \times B_i$$

所以在计算多项式的运算时，我们可以将两个多项式进行DFT，在进行若干运算过后通过逆DFT转换回来，即为多个多项式的运算结果。

转换的时间复杂度为$O(L \log L)$，单次计算的时间复杂度为$O(L)$。[2]

如果我们直接使用FFT，需要转换$2n$次，计算n次，是$O(nS \log S + nS)$的，这样的时间复杂度反而更劣了。

注意到如果直接一个一个多项式乘过去，到最后会出现长度为$O(S)$的多项式乘上长度为$O(1)$的多项式，这样并不是最优的方法。所以我们可以每次将n个多项式分为两半，将两边的多项式分别乘起来之后再相乘，这样的复杂度为：

$$T(S) = 2T\left(\frac{S}{2}\right) + O(S \log S) \tag{12}$$

可以算出总复杂度为$O(S \log^2 S)$。

时间复杂度：$O(S \log^2 S)$
空间复杂度：$O(S)$

5.5 算法4

同样的，我们在递推求$g_{i,j}$的时候，可以考虑用FFT进行优化。

在我们将F_i进行DFT之后，转移方程9变为：

$$g_{i,j,k} = F_{i,k} g_{i-1,j-1,k} + (1 - F_{i,k}) g_{i-1,j,k} \tag{13}$$

接下来，我们确定i与k，即将整个g的第二维与第三维交换一下，则转移方程变为：

$$g_{i,k,j} = F_{i,k} g_{i-1,k,j-1} + (1 - F_{i,k}) g_{i-1,k,j} \tag{14}$$

[2]FFT的具体实现过程可以戳这里http://en.wikipedia.org/wiki/Fast_Fourier_transform

可以发现，数列$g_{i,k}$的生成函数$G_{i,k}(x)$与数列$g_{i-1,k}$的生成函数$G_{i-1,k}(x)$的关系为：

$$G_{i,k}(x) = [F_{i,k}x + (1 - F_{i,k})]G_{i-1,k}(x) \tag{15}$$

由于$G_{0,k}(x) = 1$，则可推导出：

$$G_{n-1,k}(x) = \prod_{i=1}^{n-1} [F_{i,k}x + (1 - F_{i,k})] \tag{16}$$

观察16，对于每一个k，如果我们已经知道了$F_{i,k}$的值，都可以使用如5.4.4所说的分治套FFT算法，在$O(n \log^2 n)$的时间内求出$G_{n,k}(x)$。而$F_{i,k}$可以直接利用FFT在$O(nS \log S)$的时间内求出。

在求出所有的$g_{n-1,k,j}$之后，最终使用逆DFT转化回来即为函数$g_{n-1,j}(x)$。

时间复杂度：$O(nS \log^2 n)$

空间复杂度：$O(nS)$

5.6 算法5

换个角度，假设对于转移方程9我们直接套用生成函数，则对于$g_{i,j}(x)$的二元生成函数$G_i(x, y)$，有：

$$G_i(x, y) = \Big\{F_i(x)y + [1 - F_i(x)]\Big\}G_{i-1}(x, y) \tag{17}$$

又由于$G_0 = 1$，则可以推导出：

$$G_{n-1}(x, y) = \prod_{i=1}^{n-1} \Big\{F_i(x)y + [1 - F_i(x)]\Big\} \tag{18}$$

则最终我们需要求的即为若干个二元多项式乘起来之后所得的多项式。

对于二元多项式的DFT，相当于分别枚举x与y在取不同的值时，函数的值为多少，所以我们只需要先固定y，对于y的若干次幂之前关于x多项式进行FFT，再对于每个x不同的取值，对关于y的多项式进行FFT即可，对于一个$N \times M$的二元多项式进行FFT的时间复杂度为$O(NM \log NM)$。

有了二元多项式的DFT，接下来我们可以沿用之前的思路——分治套FFT来解决当前问题。

需要注意的是，假设我们的问题规模是$n \times S$的，由于数据是随机的，当我们将n个多项式等量的分成两份时，它们乘起来的项数也大约是差不多的，则此时的时间复杂度为：

$$T(n,S) = 2T\left(\frac{n}{2}, \frac{S}{2}\right) + O(nS \log nS) \tag{19}$$

可以发现，此时分治套FFT的时间复杂度并没有$\log^2$项了，时间复杂度仅为$O(nS \log nS)$。

在求出G_{n-1}之后，我们只需要用$g_{n-1,j}(x)$乘上$f_n(x)$再积分即为答案。

至此，问题完美解决。

时间复杂度：$O(nS \log nS)$

空间复杂度：$O(nS)$

6 感谢

感谢屈运华老师在学习生活上的关心和照顾。

感谢父母对我的养育之恩。

感谢朱全民老师、廖晓刚老师和汪星明老师的教导。

感谢CCF提供的平台和机会。

感谢国家集训队教练的辛勤付出。

感谢清华大学黄志翱学长为我提供的帮助。

感谢匡正非、杨定澄、刘研绎同学为本文审稿。

感谢于纪平、吕凯风、杜瑜皓同学在出题的过程中提出的建议。

感谢所有帮助过我的同学。

浅谈分块在一类在线问题中的应用

宁波市镇海中学 邹逍遥

摘 要

分块是一种适用性高，易于实现且思维难度低的方法，能够用于解决一些信息不能快速合并的题目。本文介绍了几种常见的分块方法，并通过一些例题简单介绍了分块在一些在线维护（或询问）问题中的应用。

引言

在竞赛和实践中，往往会出现一类要求维护序列或树形结构的问题，而且一般带有修改操作。有许许多多的数据结构可以在log级别的复杂度中维护一些比较简单的情况，但是遇到一些比较复杂的情况这些数据结构往往会遇到一些困难，需要使用结构嵌套或挖掘需要维护的信息的深层性质甚至根本无法解决。

在遇到这类题目时分块方法的优势就体现出来了：不需要对问题做过多的分析，一种方法可以维护许许多多不同的要求。虽然分块方法的复杂度往往是根号级别甚至更高，但是分块方法的思维复杂度和代码复杂度往往都远低于复杂度更低的数据结构，在实践中实用性很高。

当然这类题目也有一些优秀的离线算法可以解决，比如莫队算法，cdq分治等等。不过假如遇到强制在线的题目，这些算法就无法发挥了。

本文简单介绍了几种常用的在线维护分块结构或预处理分块结构以便在线回答询问的方法。第1节主要介绍了直接将序列或树分割成许多块进行维护的方法；第2节主要介绍了通过分类将信息分开维护或将询问分类回答的方法；第3节主要介绍了定期重建思想的运用。

以下题目如未特殊说明均要求在线算法。

1 直接分块

1.1 序列分块

序列分块是算法竞赛中最常见也最容易实现的一类分块题目。

基本思想是在序列上每隔$H(N)$个点选择一个关键点，这样任意一段区间询问都能被拆成$O(N/H(N))$个区间和$O(H(N))$个单点，询问时就不需要遍历整个数组，同时在修改的时候也不需要修改整个数组。

1.1.1 最简单的例子

【例1】弹飞绵羊[1]

给你N个数a_i满足$i < a_i \leq N+1$，有N个操作，每次可以

- 修改一个数（修改后仍然满足上述条件）
- 查询从$x = i$开始每次将x变为a_x直到$x = N+1$一共需要几次变换。

这题可以使用Link-cut trees在$O(N \log N)$的时间内解决，不过这和本文关系不大故略去。

首先考虑两种暴力做法：修改直接修改那个位置，查询的时候$O(N)$模拟。或者修改的时候$O(N)$将每个位置的答案算出来，查询的时候$O(1)$回答。

那么我们可以使用分块来平衡两种做法的复杂度：首先将序列每连续的$H(N)$个分成一块，例如当$N = 18, H(N) = 4$时（相同颜色的表示一块）：

那么对于每一块使用第二种暴力算出每个位置的答案，在修改的时候只需要将这一块整个重新计算，时间复杂度$O(H(N))$。在查询的时候使用第一种暴力，由于利用预处理信息每次至少可以跳过一段，时间复杂度$O(N/H(N))$。

注意到只需要在预处理中算出某个数跳出所在的那一块需要多少步和跳出去后落在哪个位置就可以使在查询的时候每次都能从当前点$O(1)$跳出这一段。

由于要平衡修改和询问的复杂度，所以当$H(N) = \sqrt{N}$时最优，这样这题就可以在$O(N\sqrt{N})$的时间内解决了。

[1]题目来源:HNOI2010

1.1.2 一个稍微复杂一点的例子

【例2】Chef and Churu[2]

给定N个数a_i，N个函数f_i，每个函数表示一段区间的a_i的和（即$f_i = \Sigma_{x=l_i}^{r_i} a_x$，$f_i$的值会根据$a_i$的值的改变而改变）。有$N$ 个操作，每个操作可以：

- 修改一个数（a_i）
- 询问$\Sigma_{x=l}^{r} f_x$

注意到维护一个树状数组就可以$O(\log n)$求出某一个f_i的值，所以可以考虑对f_i分块。

和上一题一样先将f_i每隔$H(N)$个分成$N/H(N)$块，对于每一块都预处理出这一块中每个a_i出现了几次，以及初始状态下这一块中f_i的和，时间复杂度$O(N \times H(N))$。

对于每个修改，将每一段的f_i的和更新一下即可（由于预处理出了a_i的出现次数所以这很容易实现），时间复杂度$O(N/H(N))$。

对于每个询问，将询问区间拆分成$O(H(N))$个单点和$O(N/H(N))$个区间，其中每个单点可以在$O(\log N)$的时间内解决（通过维护树状数组即可），每个区间可以在$O(1)$的时间内解决。

不过注意到维护树状数组是$O(\log N)$-$O(\log N)$的，而修改的时候完全可以做到更高的复杂度而不影响整体复杂度。所以可以使用$O(\sqrt{N})$-$O(1)$ 支持单点修改和区间和查询的数据结构维护a_i：

由于只需要查询区间和，满足可减性质，于是可以将区间和转化为前缀和相减。由于改了一个点会影响到的前缀查询是一个后缀，所以单点修改前缀查询可以变成后缀修改单点查询。那么可以把a_i每隔$O(\sqrt{N})$个分为$O(\sqrt{N})$块，每次修改只需要修改$O(\sqrt{N})$块和$O(\sqrt{N})$ 个单点，查询只需要把块内答案和单点答案相加即可。

这样询问和修改就可以做到$O(N/H(N) + H(N))$，当$H(N) = \sqrt{N}$的时候时间复杂度为$O(N\sqrt{N})$，空间复杂度为$O(N\sqrt{N})$。

[2]题目来源:Codechef Nov 14 Challenge

1.1.3 需要维护块的区间信息的情况

【例3】Chef and Problems[3]

给定N个数a_i，有N个询问，每次给出l, r，求满足$a_i = a_j$和$l \le i, j \le r$的$j-i$的最大值。

首先考虑对于每个询问$O(n)$的暴力做法：扫一遍这个区间，并用每个数和这个区间内这个数第一次出现的位置更新答案。使用时间戳进行更新就可以避免每次遍历整个数组而只需要遍历询问区间的长度。

那么可以和上一题一样将a_i每$H(N)$个分成$N/H(N)$块，用暴力算法算出每一块内部的答案。时间复杂度$O(N)$。

但是注意到这题不能像上一题一样直接合并块内信息，还需要计算块的区间内的答案。首先需要预处理出和a_i相同的满足$j > kH(N)$的最小的j，同理算出和a_i相同的满足$j \le kH(N)$的最大的j（时间复杂度$O(N \times N/H(N))$）。这样就可以算出i在第x块，j在第$x \sim y$块时的最大解，记为$calc(x, y)$（时间复杂度$O(N \times H(N))$）。

同时第x块到第y块的答案还包含端点不在第x块或第y块上的情况，所以需要使用一个区间dp来算出这部分的答案：用$f(i, j)$表示第i块到第j块之间的答案。那么$f(i, j) = \max(f(i+1, j), f(i, j-1), calc(i, j))$。

接下来处理询问的时候就方便多了，每个区间都可以分成一个已经计算出答案的区间和不超过$2H(N)$个点。可以扫一遍多出来的点在$O(H(N))$的时间内算出两边端点都在外面的答案。然后就利用预处理信息算出每个外部的点和区间内部的点的最大答案。将三部分答案取最大值即可。时间复杂度$O(H(N))$。

总的时间复杂度为$O(N \times (H(N) + N/H(N)))$-$O(H(N))$，空间$O(N \times N/H(N))$。当$H(N) = \sqrt{N}$的时候时间复杂度为$O(N \times \sqrt{N})$。但是为了节省空间可以把$H(N)$适当开大。

像例3这样的题目不能很方便地加入修改操作，因为每次修改可能造成$O((N/H(N))^2)$个预处理值变动。不过加入修改以后可以通过让$H(N) = O(N^{\frac{2}{3}})$来达到$O(N^{\frac{5}{3}})$的复杂度。

[3]题目来源:Codechef March 15 Challenge

1.2 树上分块

有许多题目在链上分块时很容易实现，但是在树上使用分块就会比较麻烦。本节介绍了一种非常容易实现的对树分块的方法，可以解决一系列的只询问链上信息或祖先关系信息的题目。

1.2.1 解决链上信息询问

【例4】Children Trips[4]

给定一棵N个点的树，每条边长度是1或2。有N个询问，每次给出x,y,z，要求找出尽量少的点$a_1 \cdots a_n$使得$dis(x,a_1), dis(a_1,a_2), \cdots, dis(a_n,y)$ 都不超过z（这里$dis(i,j)$表示树上i点和j点的距离）。

这道题的信息合并非常困难，因为合并不支持结合律，只能从前向后合并过去。但是可以观察到有以下两个性质：

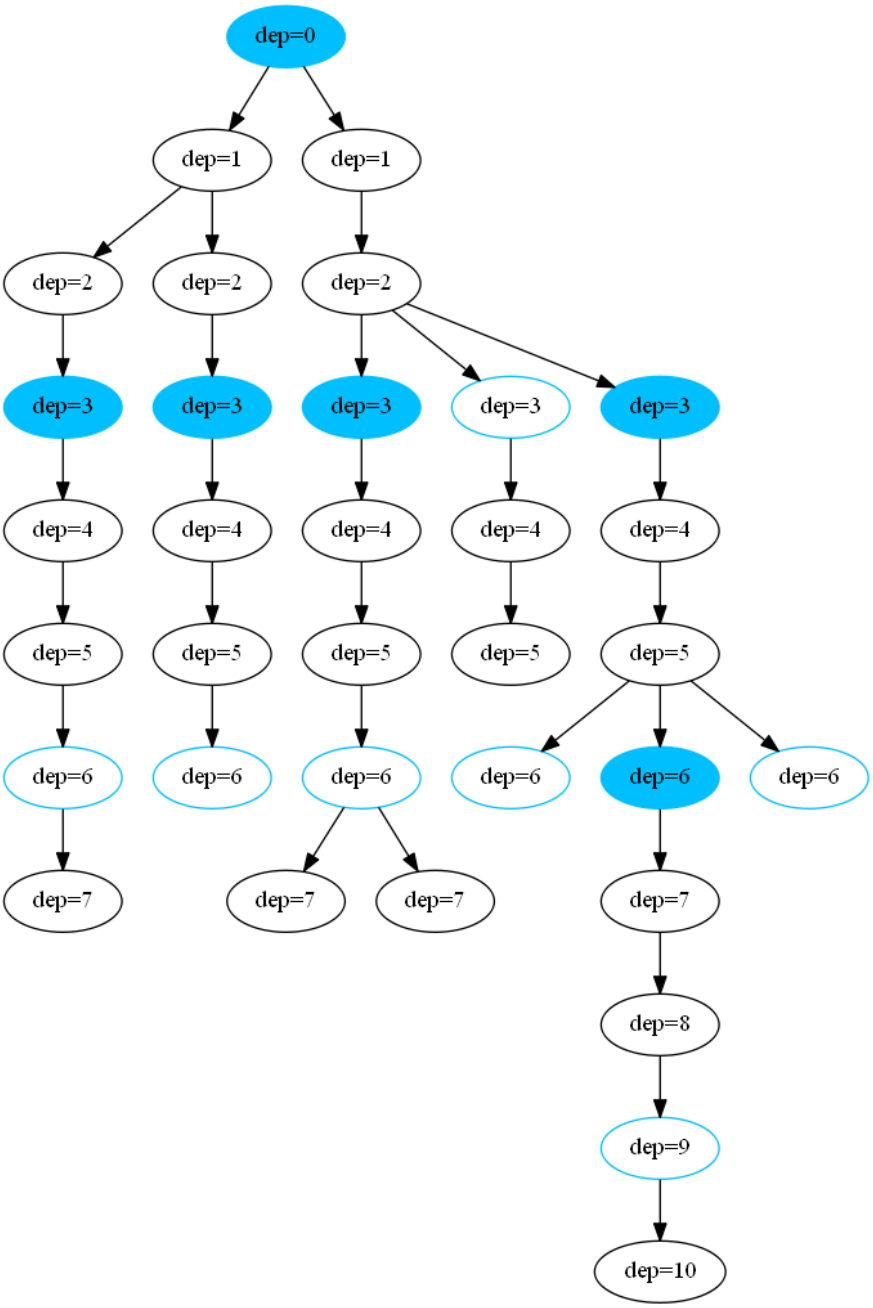
性质1:假如按i递增的顺序选择a_i，每次选a_{i-1}（设x为a_0,y为a_{n+1}）到y路径上最远的那个满足条件的点当a_i，这样选出来的点一定是满足最小的条件的（即每次走的步长尽量大答案不会变差）。

性质2:假如前一些点按i递增选最远的直到a_x，后面一些点按i递减选最远的直到a_x，这样选出来的点也是满足最小的条件的（即两端向内同时走答案不会变差）。

考虑像序列分块一样在树上选出一些关键点。首先求出每个点的深度dep_i（根结点为0）和子树大小$size_i$，并把那些满足$dep_i \mod H(N) = 0$的点选出来。

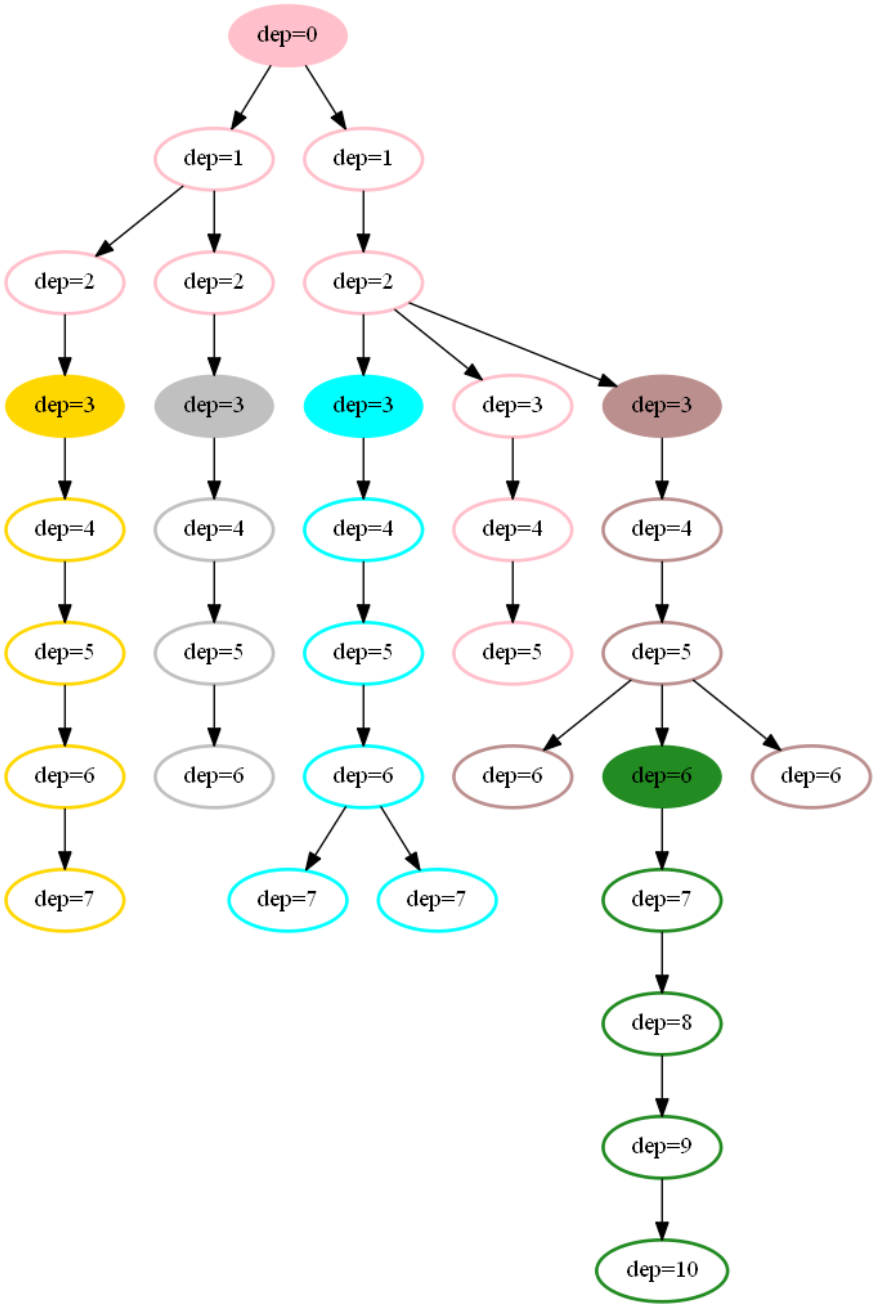
但是注意到这样做可能会选出非常多的点（比如一条长为$H(N)-1$的链，在链的一段接出去$O(N)$个点），所以需要把那些子树大小不超过$H(N)$的点去除。最后选出来的点如下图所示：

[4]题目来源:Codechef Oct 14 Challenge



然后我们把所有的关键点拿出来建成一棵树，称为“关键树”，在关键树上每个节点的父节点是原树上向上走的第一个关键点，每条边对应原树中这两个点之间的链。同时每个关键点代表一个点集，点集包含所有向根走第一个遇到的关键点是它的点。点集分类如下图所示，实心点表示关键点，相同颜色的点

属于同一个集合。



那么我们分析一下这个结构有哪些性质：

性质3:任意两个关键树上的相邻关键点在原树中距离为$H(N)$。

性质4:每个关键点代表的点集的大小不小于$H(N)$。

证明:假如这个关键点在关键树中是叶子，那么它所有原树中的子孙都属于这个点集，由定义显然成立。假如这个关键点在关键树中不是叶子，那么它到它的某一个儿子节点的那条链有$H(N)$个点，而这些点显然从属于它代表的集合。

性质5:最多选出$O(N/H(N))$个关键点。

性质6:每个点至多向上走$2H(N)$步就能遇到一个关键点。

证明:假设走了$2H(N)$步还没有遇到关键点，那么容易看出向上走$2H(N)$步经过的点至少有两个点的深度被$H(N)$整除，那么其中更靠近根的那个点的子树大小显然超过了$H(N)$，所以那个点一定能成为关键点，所以假设不成立。

性质7:任意两个点之间的路径可以被拆分成$O(H(N))$条原树上的边和$O(N/H(N))$条关键树上的边。

证明:由于不可能出现一个点连续向父亲走了$2H(N)$步的情况，所以最多只有$4H(N)$条原树上的边。而关键树上的边总共只有$O(N/H(N))$条，所以不可能多于$O(N/H(N))$条。

假如已经预处理出关键树上相邻两个点之间的答案，询问就很好处理了。

注意到这里只需要维护z为$1 \sim 2H(N)$时的答案即可。因为假如$z \geq 2H(N)$，那么这一段会被直接跳过去所以不需要计算。那么首先需要计算出每个点向上以z为步长跳到的第一个点是什么，跳到最近的关键点需要几步，跳到之后最后一步还剩多长。预处理复杂度是$O(N \times H(N))$。

询问的时候只需要利用性质2，类似倍增求LCA的算法将两个点不停向根移动就可以了。不过注意LCA并不一定是关键点，所以在再跳一步就要高于LCA的情况下要使用暴力把最后的一段跳完。在移动的过程中维护已经移动了几步和最后一步还剩多长。移动一大步的时候首先把最后剩余的长度移动完，然后利用预处理信息一步移动到上一个关键点。时间复杂度$O(N/H(N))$。

当$H(N) = \sqrt{N}$的时候时间复杂度为$O(N\sqrt{N})$，空间复杂度为$O(N\sqrt{N})$。

1.2.2 解决祖先关系询问

【例5】Regions[5]

[5]题目来源:IOI2009

给定一棵N个结点的树，每个结点有一个颜色，每次给出x,y查询有多少对数i,j满足i的颜色是x，j的颜色是y且i是j的祖先。

我们像上一题一样建出一棵关键树，但是注意到这样的关键树并没有一些很容易划分成祖先关系的性质，于是我们将它进行一些扩展。

首先和上一题一样选出“初始”关键点（$dep_i \mod H(N) = 0$且$size_i \geq H(N)$的点），然后把所有满足是两个初始关键点的LCA（最近公共祖先）的点也选出来并加入关键点集合形成最后的关键点集。

性质1:关键点最多增加一倍。

证明:将初始关键点按DFS序排列，按顺序枚举初始关键点，对于每个初始关键点，不停地向根节点走直到走到另一个初始关键点或一个已经访问过的点。假如停下来的时候所在的位置是初始关键点，那么再向上走的情况和那个点向上走的情况完全相同，没有必要再走下去。假如走到的是已经访问过的点，就把这个点加入关键点集合，那么再向上走的情况和第一次访问这个点的点遇到的情况完全相同，也没有必要再走下去。所以每个点至多只会贡献一个新点。

其实这么做就是建出了一棵“虚树”，在实现的时候可以直接加入所有DFS序相邻的两个点的LCA并去重即可。关于虚树的详细讨论可以参考2014年徐寅展的集训队论文。

扩充了关键点集合后关键树的定义仍然和之前相同（每个点的父亲是原树中的最近关键祖先），关键树中的边代表原树中的一条链中的点组成的点集。

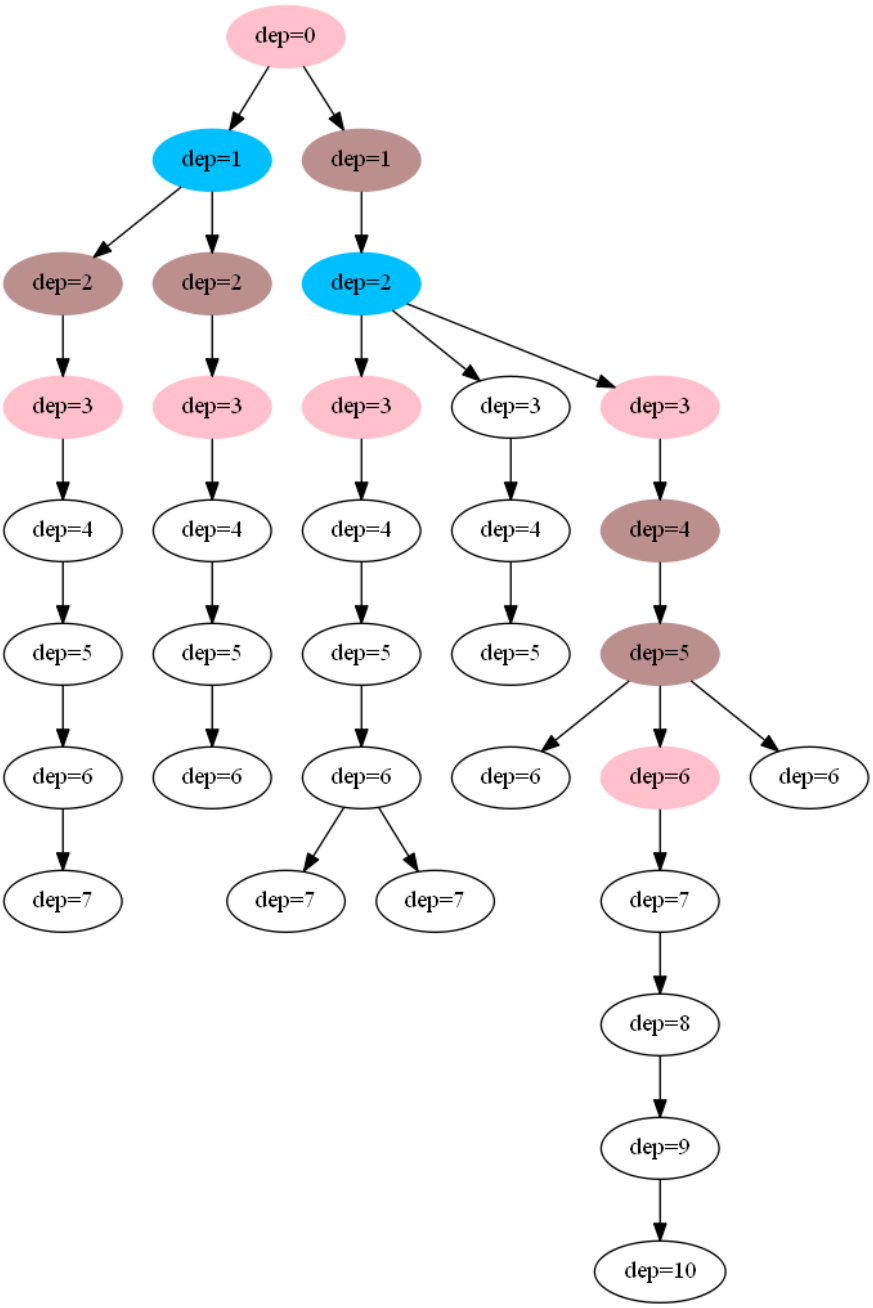
现在这棵树就又满足了更多性质；

性质2:任意两条关键树中的边在原树中代表的路径只会在端点处相交。

性质3:树中任意两点之间的路径都可以被划分为$O(H(N))$条边和$O(N/H(N))$条相邻的关键点之间的路径。

性质4:任意两个相邻的关键点之间的路径长度不超过$H(N)$且关键点数量不超过$2N/H(N)$。

新选出的关键点如下图所示（粉色表示初始关键点，蓝色表示新加入的关键点，棕色表示位于关键树的边上的点）：



那么有了这些性质就容易解决这道题了。

把所有点分成两类：把位于关键树的边上的点（包括关键点，比如上图中的所有有颜色的点）称为“内部点”，把剩下的点称为“外部点”。那么我们把所有需要统计进去的情况分成三类：

- 两个点都是内部点。
- 祖先结点是内部点，子孙结点是外部点，但是祖先结点不在子孙结点向上走到第一个关键点所经过的路径上。
- 祖先结点是内部点，子孙结点是外部点，但是祖先结点在子孙结点向上走到第一个关键点所经过的路径上。

第三种情况很容易解决，由于每个点至多向上走$2H(N)$步就会遇到一个关键点，所以只需要暴力枚举就可以预处理出第三种情况的答案。

在关键树上把每个关键点和它到它父亲结点的路径上的点分成一块（根结点只包含他自己），那么对于每一块都预处理出块内部每种颜色的数量，这个可以在$O(N \times N/H(N))$的时间内解决。

容易证明假如关键点x是关键点y的祖先，那么x所在块中的每一个点都是y所在块中的点的祖先。所以在询问的时候只需要遍历一遍关键树就可以解决第一种情况。

最后考虑第二种情况：预处理出每个外部点向上走遇到的第一个关键点，并将这个外部点归入关键点的集合。那么可以和第一种情况类似地解决：只需要预处理出每个关键点所代表的外部点中每种颜色的出现次数，查询时利用DFS到这个点时这个点到根路径上的x颜色的数量和这个点的集合内的y颜色数量更新答案即可。

当$H(N) = \sqrt{N}$的时候本题的时间复杂度为$O(N\sqrt{N})$，空间复杂度为$O(N\sqrt{N})$。

2 按大小分类讨论

按大小分类讨论的基本思想是将询问（或修改或度数等等）分成大于$H(N)$的和不大于$H(N)$的两类分开解决。也就是将两种$O(size)$（或$O(size \log size$等）和$O(N/size)$的暴力结合在了一起，所以思维复杂度低，而且结合的是两个暴力做法，在考场上容易实现。

2.1 【例4】Children Trips[6]

给定一棵N个点的树，每条边长度是1或2。有N个询问，每次给出x,y,z，要求找出尽量少的点$a_1 \cdots a_n$使得$dis(x,a_1), dis(a_1,a_2), \cdots, dis(a_n,y)$ 都不超过z（这里$dis(i,j)$表示树上i点和j点的距离）。

考虑暴力做法：每次二分下一个a_i的位置，这样是$O(N \log N)$的，因为a_i最多会有$O(N)$个。但是注意到当$z > H(N)$的时候a_i最多只有$O(N/H(N))$ 个可以直接暴力，而$z <= H(N)$的情况只有不超过$H(N)$种，假如把这些z的答案预处理出来，那么问题就解决了。

那么可以对每一个比较小的的z，都预处理出每个点向根走z长度（假如没有距离为z的点那么就选取距离为$z-1$的点）到的是哪个点，并倍增出每个点向上走2^k 步（每步长度为z）到的是那个点。这样需要$O(N \times H(N) \log N)$ 的时间和空间。询问的时候可以像正常的倍增算法一样向上走到再向上走就超过LCA为止，最后多出来的那一段可以暴力在$O(H(N))$的时间内走完。

取$H(N) = \sqrt{N}$就可以在$O(N\sqrt{N}\log N)$的时间和$O(N\sqrt{N}\log N)$的空间下解决这道题。

这是官方给出的本题解法，思维难度比较低，容易想到。但是在树上两点间二分比较麻烦不易实现。使用第一章给出的做法能更容易地解决这道题。

2.2 【例5】Regions[7]

给定一棵N个结点的树，每个结点有一个颜色，每次给出x,y查询有多少对数i,j满足i的颜色是x，j的颜色是y且i是j的祖先。

首先来看一种特殊情况：所有的询问中x都相同。那么可以通过一个简单的DFS来算出每个y的答案，只需要在DFS的过程中记录下当前这个点到根的路径上有多少个颜色为x的结点即可。

[6]**题目来源**:Codechef Oct 14 Challenge
[7]**题目来源**:IOI2009

预处理出DFS序，那么每个子树所代表的点就是左端点为它本身的一个区间。并同时预处理出每种颜色的点代表的区间的左右端点排好序的结果。

对于每个询问，设两种颜色的点分别有A个和B个，那么算出每个点在多少个区间内，然后把答案相加即可。由于已经排好了序，那么可以在$O(A+B)$的时间内将排序结果归并起来，然后扫描一遍就能得到答案。

那么再考虑这样一种针对颜色数较少的情况的暴力：对于每个x都进行一次DFS，可以在$O(NC)$的时间内（C为颜色数）预处理出每个询问的答案，只需要在DFS时记录下当前点到根一共有多少种x颜色即可。类似地固定y时也可以通过扫一遍所有点算出答案，只需要预处理出前缀和就能$O(1)$查询区间了。

考虑将这两种暴力结合起来：对于出现次数超过$H(N)$的颜色，预处理出这些颜色作为其中一种（x或y）时的答案。对于剩下的在询问时使用第一种暴力直接回答。

当$H(N)=\sqrt{N}$时，时间复杂度为$O(N\sqrt{N})$，空间复杂度为$O(N\sqrt{N})$。

当然官方题解中还给出了另外一种分类讨论的方法可以在$O(N\sqrt{N\log N})$的时间和$O(N)$的空间之内解决。这种做法在2014年王悦同的论文中已经提及，这里不再赘述。

3 定期重建

定期重建的思想是将操作分块，每$H(N)$个操作后就重建一次状态，查询的时候根据上一次重建和期间的所有修改计算答案。

注意对操作分块并不需要离线，只需要每隔一定时间重新预处理一遍即可。

3.1 【例2】Chef and Churu[8]

给定N个数a_i，N个函数f_i，每个函数表示一段区间的a_i的和（即$f_i=\Sigma_{x=l_i}^{r_i}a_x$，$f_i$的值会根据$a_i$的值的改变而改变）。有$N$ 个操作，每个操作可以：

- 修改一个数（a_i）
- 询问$\Sigma_{x=l}^{r}f_x$

[8]**题目来源**:Codechef Nov 14 Challenge

先将操作分块，即每$H(N)$次操作重建一次，每次重建的时候$O(N)$算出原数组的前缀和并$O(N)$求出每个f_i的值，复杂度$O(N \times N/H(N))$。

接下来只需要计算每个操作对块内询问的贡献。由于询问是可减的，可以把每个询问拆成前缀询问，然后把每个前缀询问按长度排序。

排完序后就可以按右端点递增扫过来并维护一个支持$O(\sqrt{N})$区间加$O(1)$单点查询的数据结构，这个数据结构在之前提到过：把a_i每隔$O(H(N))$个分为$O(N/H(N))$块，每次修改只需要修改$O(N/H(N))$块和$O(H(N))$个单点，查询只需要把块内答案和单点答案相加即可。然后对于每个询问枚举和它在同一块中并且在它前面的所有$O(H(N))$个修改，由于可以$O(1)$查询某个位置对那个询问的影响，就可以在$O(N \times H(N))$ 的时间内算出这一部分的贡献。

总时间复杂度$O(N\sqrt{N})$，空间复杂度$O(N)$。

当然假如需要强制在线的话可以把排序并扫过来的那个操作改成预处理并记录在一棵可持久化线段树中。这样可以通过增加一个log来完成在线做法，时间复杂度为$O(N\sqrt{N}\log N)$，不过内存也需要$O(N\sqrt{N}\log N)$。

3.2 【例6】支持link,cut的树上第k大问题[9]

给定一个N个点的森林，有N个操作，每次可以加一条边/删一条边（保证处理完后还是森林），可以询问一条链上的第k大。

首先考虑没有link,cut操作的情况。这可以使用可持久化线段树来解决：每个点维护一棵在它的父亲的基础上加上这个点的点权的可持久化线段树，这样在查询的时候每条链可以拆分成两条从某个点开始指向根的路径，利用这两条链的和在可持久化线段树上二分即可。

假如维护了K次操作以前的可持久化线段树，那么可以通过Link-cut trees将现在树上的链拆分成最多$O(K)$条K时刻以前的树上的链。通过这些链的和在可持久化线段树上二分即可。

那么可以每进行$\sqrt{N}$次操作就将可持久化线段树重建，时间复杂度为

[9]经典问题

$O(N\sqrt{N}\log N)$。

总结

分块方法的优势在于通用性高，可以支持维护很多种类的信息，前几章的例题几乎都用了两种不同的想法来做。不过时间复杂度高是分块方法的致命缺陷，但是在想不出更优的解法的情况下试一试未尝不可。

致谢

感谢中国计算机学会为我提供学习和交流的平台。
感谢李建老师的多年来给予的关心和指导。
感谢在本文写作过程中提供帮助的岑若虚，张煜皓，杜瑜皓等同学。

参考文献

[1] Mugurel Ionut Andreica, “Novel O(H(N)+N/H(N)) Algorithmic Techniques for Several Types of Queries and Updates on Rooted Trees and Lists ”

[2] “21-st International Olympiad In Informatics TASKS AND SOLUTIONS” from www.ioi2009.org

[3] 《根号算法——不只是分块》2014年国家集训队论文，南京外国语学校-王悦同

[4] 《线段树在一类分治问题上的应用》2014年国家集训队论文，杭州学军中学-徐寅展

仙人掌相关算法及其应用

杭州学军中学 王逸松

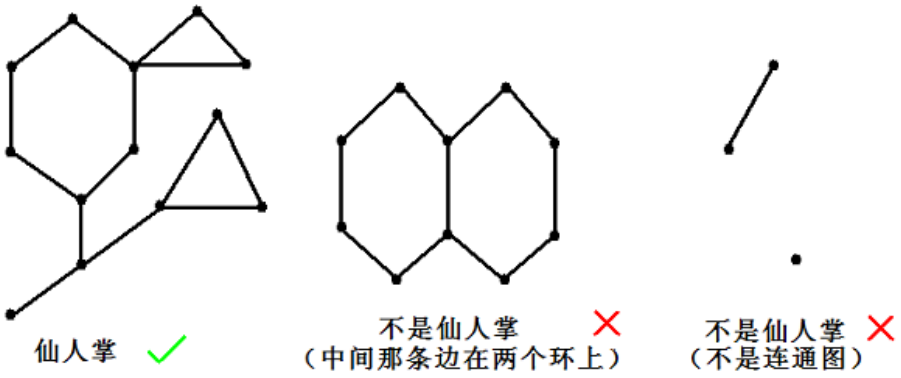
摘 要

近几年来，信息学竞赛中出现了不少仙人掌相关的问题，如动态仙人掌问题。本文将对仙人掌相关的常见算法进行介绍，并举出一些例题，希望能起到抛砖引玉的效果。

1 仙人掌的定义和结构

1.1 仙人掌的定义

如果一个无向连通图的任意一条边属于最多一个简单环，且没有自环，我们就称之为仙人掌。

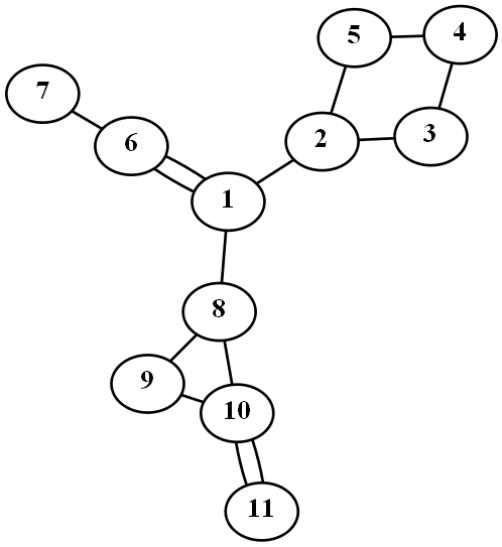


1.2 仙人掌的边数

对于一棵n个结点的仙人掌，我们考虑它的一棵生成树，有$n-1$条边。剩下的边均为非树边，每条非树边对应了生成树上的一条路径，并形成了一个简单

环。由于每一条边最多属于一个简单环，任意两条非树边对应的路径的边的交集为空，即这些路径不重叠。因此最多有$n-1$条这样的路径，所以n个结点的仙人掌最多有$2n-2$条边，最少有$n-1$条边。

1.3 仙人掌的结构



如图是一棵根为1的仙人掌

1.3.1 父亲

类比树上结点的父亲，我们定义仙人掌上结点的父亲和环的父亲。

如果一个结点u到仙人掌的根之间的所有简单路径的第一条边相同，那么就类似树上的情况，u的父亲为u到根的一条简单路径上第二个点，否则u的父亲为u到根的一条简单路径上第一条边所在的环。

比如说，8号结点的父亲是1号结点，10号结点的父亲是环8-9-10，11号结点的父亲是环10-11。

一个环的父亲为环上离根最近的结点。

比如说，环8-9-10的父亲为8号结点，环10-11的父亲为10号结点。

1.3.2 儿子

类比树上结点的儿子，我们定义仙人掌上结点的儿子和环的儿子。

一个结点的儿子可能是个结点，也可能是个环。

比如说，1号结点的儿子有2号结点、8号结点和环1-6。

一个环的儿子为环上除去环的父亲以外的所有结点。

比如说，环8-9-10的儿子为9号结点和10号结点。

1.3.3 父亲结点和母亲结点

对于一个在环上的结点，我们定义它的**父亲结点**和**母亲结点**分别为环上与它相邻的两个结点。（请注意区分**父亲**和**父亲结点**）

比如说，在环2-3-4-5中，3号结点的父亲结点为2号结点，母亲结点为4号结点，4号结点的父亲结点为3号结点，母亲节点为5号结点。

从环的任意一个儿子出发，沿着父亲结点和母亲结点就能遍历整个环。

1.4 如何遍历一棵仙人掌

类似遍历一棵树的过程，我们从根开始dfs，假设当前在结点x，接下来要访问结点y。

如果y还没访问过，就跟树上的情况一样处理，只要将y的父亲设置为x即可。

如果y已经访问过，且第一次访问y的时间早于第一次访问x的时间，那么说明我们发现了一个环，这个环的父亲为y，儿子为x到y路径上除去y的所有点。我们遍历这个环，并设置环上所有点的父亲结点和母亲结点就行了。

如果y已经访问过，且第一次访问y的时间晚于第一次访问x的时间，那么说明y在一个已经访问过的环上，可以无视这种情况。

2 仙人掌上的DP

类比树上的DP，我们可以在仙人掌上进行DP。

从仙人掌的根开始DP，假设当前DP到结点u，我们先处理结点u上的信息，然后枚举u的每一个儿子和以u为父亲的每一个环的每个儿子进行DP。

2.1 【例题一】一个经典问题

给一棵仙人掌，每条边有个长度，求1号结点到每个结点的距离。

其中两个点的距离为它们之间的最短路径的长度。

$n \leq 10^5$

仙人掌上的DP

先dfs一遍仙人掌搞清楚仙人掌的结构。

然后从1号结点开始DP。假设当前已经求出了1号结点到结点u的距离，我们枚举u的每一个儿子，如果该儿子是一个结点，那么就能求出1号结点到这个结点的距离，并从这个结点开始继续DP。否则该儿子是一个环，我们枚举环上的每个儿子x，求出结点x到1号结点的距离，然后从x开始DP。

时间复杂度是$O(n)$的。

2.2 【例题二】另一个经典问题

给一棵仙人掌，每条边有个长度，求直径。

直径的定义是，距离最远的两个点的距离。其中两个点的距离为它们之间的最短路径的长度。

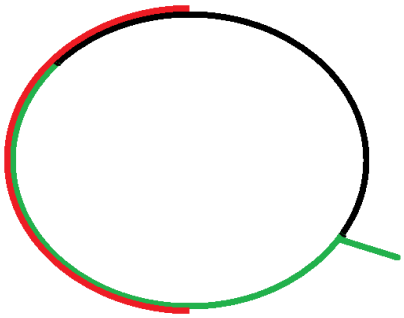
$n \leq 10^5$

2.2.1 树上的情况

由于树也是仙人掌，我们先来考虑树上的情况。

一个经典的做法是，随便选一个点，然后找到离它最远的点，再从这个点开始，再找一次最远的点，这两个点之间的距离即为答案。

这个做法在仙人掌上是错的。



如图，考虑一个有$2k$个结点的环加上一条边，环上每条边的长度都是1，那么答案为$k+1$，但是只有当一开始选到$O(1)$个点时，才能得到这个答案。

2.2.2 另一种做法

对于树上的情况，考虑树形DP。我们可以对每个x求出以x为根的子树的最大深度，然后用每个结点的不同儿子的最大深度+次大深度更新答案即可。

对于仙人掌上的情况，可以类比树形DP。我们先dfs一遍弄清仙人掌的结构，然后对于每个结点x，求出子仙人掌x的最大深度。（子仙人掌x的定义是，删掉根到x的所有简单路径上的边之后，x所在的连通块）

对于每个结点，用它的不同儿子的最大深度+次大深度来更新答案。

对于每个环，还要用环上的每对结点的最大深度的和加上这对结点的最短路长度来更新答案。不妨枚举其中一个结点，那么最短路是顺时针走的另一个结点，一定是环上的一个区间，而且这个区间的另一个端点随着枚举的结点顺时针移动而顺时针移动。这是个两个端点都单调移动的RMQ问题，可以用单调队列来维护。

也可以不用单调队列。按照从环的父亲往下走哪边距离近，把环分成两部分。每一部分内部的点对的最短路一定是在内部走。对于每个结点，它在另一部分最短路是顺时针走的一定是一个前缀，最短路是逆时针走的是一个后缀，直接处理前缀max和后缀max即可。

时间复杂度$O(n)$。

2.3 【例题三】mx的仙人掌[1]

给一棵n个点的仙人掌和q个询问，每个询问给定一个点集，要求输出这个点集中距离最远的点对的距离。两个结点之间的距离为它们的最短路径长度。

$n \le 3 \times 10^5, q \le 3 \times 10^5$, 点集中点的个数之和tot $\le 3 \times 10^5$

时间限制5s，内存限制512MB

暴力算法

对每个询问暴力。

对于树上的情况，每个询问只要做一次树形DP即可。

对于仙人掌上的情况，每个询问用上一道例题中的仙人掌上的DP即可。

时间复杂度$O(nq)$。

一个性质

在树形DP过程中，如果一个询问给了cnt个点，那么有效的更新答案的次数为$cnt - 1$次，因为一开始有cnt个点上有答案，每次更新答案都会减少一个“有答案”的点。

对于仙人掌上的DP也有类似性质，即有效的更新答案的次数为$O(cnt)$次。

所以对于所有询问，有效的更新答案的次数之和是$O(n)$的。

树上的情况

我们可以用$f[x][i]$表示在子树x中第i个询问的点集中的点到结点x的最大距离，然后树形DP的过程相当于合并这些数组，并更新相应询问的答案。

现在问题转化成，对于每个结点，合并它的每个儿子的DP数组，并快速找到需要被更新答案的询问编号。

这相当于，维护很多数的集合，支持把其中两个合并，并且在合并的时候返回它们的交集。

可以直接用数组来存集合中的数，然后启发式合并，即每次合并时，将小的集合暴力插入到大的集合中。由于要求交集，可以对每个集合开一个哈希表（或直接在全局开哈希表），然后合并的时候判一下即可。

时间复杂度$O(n \log n)$，空间复杂度$O(n)$。

[1]Source: UOJ #87. mx的仙人掌

仙人掌上的情况

对于除了环以外的部分，跟树上的情况一样处理。

对于每个环，先将环的每个儿子的DP数组处理出来，然后用每对DP数组的交集来更新答案。

由于没法从一个集合中删除另一个集合的元素，所以不能使用带删除操作的单调队列。

可以按照从环的父亲往下走哪边近，将环分成左右两部分，对环的每一个部分，把DP数组按顺序合并起来，并与另一部分的DP数组求交更新答案即可。

在环上更新完答案后还要撤销所有合并操作，这可以在合并的过程中记录下合并时进行的操作来实现。

最后要把环上所有儿子的DP数组全部合并。

复杂度分析：

不考虑环上更新答案时DP数组的合并，总时间复杂度是$O(n\log n)$的，因为每个询问点在合并时作为“较小的集合”最多$O(\log n)$次。

考虑环上更新答案的过程，本质上跟把环上所有DP数组合并是一样的，所以不会影响复杂度，所以总时间复杂度是$O(n\log n)$。

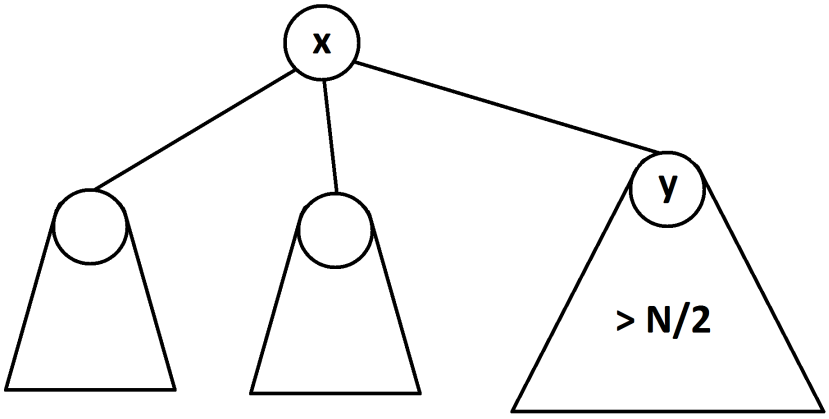
假设环上有若干个集合，总的大小为N，那么可以证明，把这些集合按顺序一个一个合并的复杂度不会超过$O(N)$，所以合并时记录下的操作个数也是$O(N)$，所以总的空间复杂度也是$O(n)$。

3 仙人掌上的点分治

类比树上的点分治，我们可以对仙人掌进行点分治。

每次找一个结点（我们称它为“重心”），使删掉这个结点和这个结点所在的所有环上的边之后，最大的连通块大小最小，然后统计与重心和重心所在的所有环上的信息，再递归分治每个剩下的连通块。

复杂度



考虑当前连通块大小为N，如果选择了结点x，并且存在一个结点y，使以x为根，子仙人掌y的大小大于$\frac{N}{2}$，那么选择y一定比选择x要优。所以每次分治后，最大的连通块大小至少减少了一半。所以分治的层数是$O(\log n)$的。

3.1 【例题一】跳蚤国王下江南[2]

给一棵仙人掌，求对于$l = 1, 2, \ldots, n$，从1号点出发经过l条边的简单路径条数，答案模998244353（$7 \times 17 \times 2^{23} + 1$，一个质数）。

$n \leq 10^5$

暴力算法

记$f[x][l]$为从结点u开始，只能向远离根的方向走，长度为l的路径条数。

那么对于x的每个儿子y，如果y是个结点，就把$f[y][l-1]$加到$f[x][l]$里。如果是个环，那么枚举这个环上的每个儿子z，把$f[z][l-d_1]+f[z][l-d_2]$加到$f[x][l]$里（其中d_1, d_2表示x到z的两条路径的长度）。

仙人掌分治

首先我们发现，$f[u]$可以看作一个多项式，每次从u的一个儿子v转移过来时，相当于是把$f[v]$乘上x或乘上$x^{d_1} + x^{d_2}$并加到$f[u]$中。这启发我们用分治的算法。

对于每次分治，找到重心u以后，先递归分治每个连通块，然后将根到重

[2]Source: UOJ #23. 【UR #1】跳蚤国王下江南

心的多项式求出来，记为$f(x)$，再将u的所有儿子的答案求出来相加，记为$g(x)$。我们只要将$f(x) \cdot g(x)$和每个连通块的答案加起来即可。

对于求$f(x)$，如果将重心到根的多项式用分治+FFT乘起来，复杂度为$O(n \log^2 n)$。其实我们可以把递归分治过程中根到重心的多项式记录下来，然后一个一个乘回去，复杂度就是$T(n) = T(n/2) + O(n \log n) = O(n \log n)$的了。

总复杂度为$O(n \log^2 n)$。

3.2 【例题二】Tree and Sets[3]

有一棵n 个结点的仙人掌，每条边有一个长度l。（不同的边的长度不一定相同）

有q 个点集，每个点集可以用两个整数u, d 来描述（$1 \le u \le n$），一个结点v在这个点集中当且仅当结点v 与结点u 的距离不超过d。两个结点之间的距离为它们之间的最短路径的长度。

现在要求构造一个有向无环图（DAG），满足：

- 这个DAG 至少有$n + q$ 个结点，至多有1200000 个结点和2400000 条边。
- 对于每一条边，如果是从u 连向v 的，那么$u > n$ 且$u \neq v$。
- 对于结点编号在第i 个点集（$1 \le i \le q$）的每一个结点x，第$n + i$ 个结点到第x 个结点有且仅有一条路径。
- 对于结点编号在$\{1, 2, \ldots, n\}$ 中但不在第i 个点集（$1 \le i \le q$）的每一个结点x，不存在第$n + i$ 个结点到第x 个结点的路径。

$1 \le n, q \le 10000$

树上的情况

先给原图加虚点，使每个点的度数都不超过3，并且对于每个点，要满足最多只有一个点集的u为该点。

然后进行树的点分治，每次分治将重心的每一个子树中的所有结点按照到重心的距离排序，然后预处理前缀和，并对于每个点集连相应的边即可。

由于点分治的层数为$O(\log n)$，所以连边的总数为$O(n \log n)$的。（假设$q = O(n)$）

[3]Source: 2015年集训队互测

仙人掌上的情况

还是可以先给原图加虚点，使每个点的度数不超过3。于是这棵仙人掌的每个结点最短属于一个环。

我们可以每次分治找一个结点，或找一个环，把它作为“重心”，容易证明这样分治的层数也是$O(\log n)$的。

重心为结点的时候，跟树上的情况一样处理即可。

重心为环的时候，类似仙人掌的DP，将这个环分成两部分，每一部分之内和两部分之间都是一个前缀和/后缀和的问题。以每一部分之内的前缀和问题为例，我们把这一部分当作一个序列，记$size[i]$为序列中下标为i的子仙人掌的大小。每次分治$[l,r]$时，记$\sum_{i=l}^{r} size[i] = N$，找一个$mid$，使$\sum_{i=l}^{mid-1} size[i] \le N/2$，且$\sum_{i=mid+1}^{r} size[i] \le N/2$，先递归分治$[l,mid)$和$(mid,r]$，然后建出$[l,mid)$、$mid$、$(mid,r]$这三部分之间的边。在这个环上的分治中，对于每个$i$，第$i$棵子仙人掌在$O(\log \frac{N}{size[i]}) + O(1)$次分治中出现。于是对于每个结点，它在每次点分治和每个环上分治的出现次数之和为$O(\log n)$。

所以连边的总数还是$O(n \log n)$的。

4 link-cut cactus

4.1 动态仙人掌问题

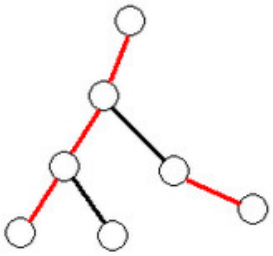
动态仙人掌问题指的是一类在仙人掌上动态维护信息的问题，动态维护指的可以是修改形态，也可以是修改相关信息。

很多动态树问题可以用link-cut tree解决。类比link-cut tree，我们能得到link-cut cactus，能用来解决不少动态仙人掌问题。

4.1.1 link-cut cactus的结构

link-cut tree维护的是树的一个链剖分，于是我们可以维护仙人掌的一个链剖分。

如果没有环，就跟link-cut tree一样，每个结点有个preferred-child，用实边连起来，其他儿子用虚边连起来。

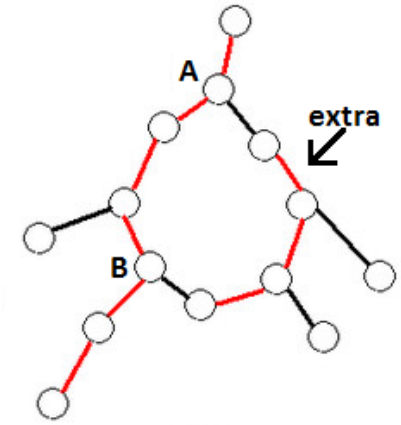


（红色的边为实边，黑色的边为虚边）

对于一个环，我们定义它的父亲为环上离仙人掌的根最近的结点，记为A。我们定义环的preferred-child为环上最后一次access到的结点，记为B。

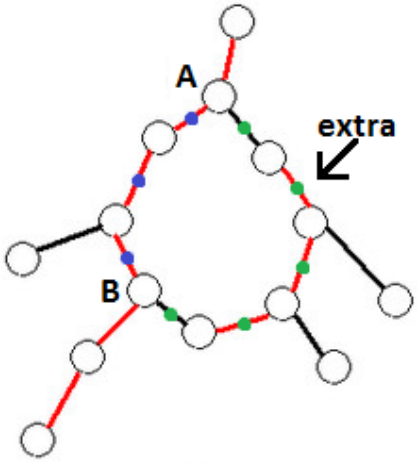
我们将A和B的最短路用实边连起来。

对于环上其他的结点构成的链，也用实边连起来，我们称这条链为**额外链**（即图中的extra链）。



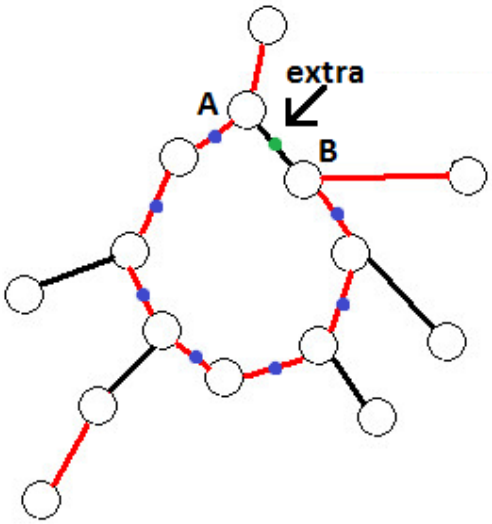
4.1.2 如何在splay上维护边的信息

在每条边上都加个点，如图所示：



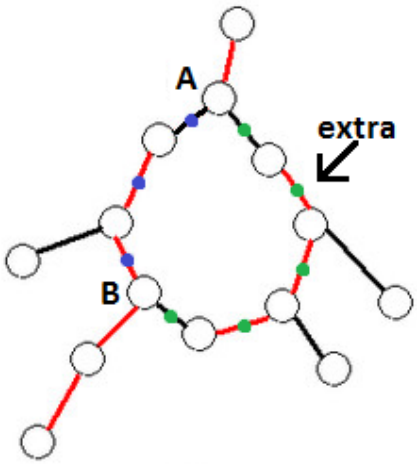
注意环上额外链两段与A和B之间的边（黑色边），我们将它直接接在额外链的两端来维护。

4.1.3 边界情况



如图，环上所有的结点都在AB链上，于是额外链上只有一条边。但是我们是在这条边上加了一个点来维护的，所以实际上不需要讨论这种情况。

4.1.4 一种特殊情况



如图，上一次access了结点A，这导致了AB链上与A相连的边变为虚边，但是表示这条边的splay结点还在AB链的splay上。

要注意特判这种情况。

4.1.5 access操作

类比link-cut tree的access操作。

假设当前access到结点x，我们先找到x在splay上的前一条边e。

如果e不在环上，就跟link-cut tree的一样处理。

否则先把这个环的AB链的左右两端断开（要注意考虑特殊情况），然后把AB链和额外链连接起来。

然后把x提到splay的根，选择较短的一边连回去作为新的AB链，并将另一边设置为额外链。

4.1.6 换根操作

类比link-cut tree的换根操作，在access后对根到该结点的路径打翻转标记来换根。

要注意这会导致这条路径经过的环的A和B结点互换，而且额外链方向不正确。

解决方法是对于access到的每一个环，检查A和B在splay中的前后顺序，如果反了，就交换A和B，并给额外链打翻转标记。

4.1.7 如何使用link-cut cactus

以查询u到v最短路上边权最小值为例：

先将根换成u，然后access结点v，然后返回这棵splay上的边权最小值即可。

要注意u到v可能有多条最短路，这个在access到环的时候在splay上打个标记维护即可。

4.1.8 时间复杂度

类比link-cut tree，对于n个结点的仙人掌和$O(n)$次access操作，结点和环的preferred-child的切换次数是$O(n \log n)$的，由于每次preferred-child的切换用到了splay，所以总时间复杂度为$O(n \log^2 n)$。

4.2 【例题一】Push the Flow! [4]

给一棵n个点的仙人掌，保证没有重边或自环，每条边有个容量。

有q个操作，每个操作是修改一条边的容量，或询问两个结点之间的最大流。

$1 \le n \le 100000$

$0 \le q \le 200000$

link-cut cactus的做法 [5]

对于没有环的情况，两个结点之间的最大流就是这两个结点之间路径上边的容量的最小值。

对于每个环，A和B结点之间的最大流为AB链上的容量最小值加上额外链上的容量最小值，所以可以将额外链的容量最小值加到AB链的每一条边的容量上。

询问时只要换根再access就行了。

[4]Source: CODECHEF PUSHFLOW

[5]这题也可以用link-cut tree来解决，由于不是重点略去不讲。

修改一条边的容量时，将根换成这条边的一个端点，然后access这条边的另一个端点，此时需要修改的边一定不会在某个环的额外链上，所以直接修改即可。

时间复杂度$O((n+q)\log^2 n)$，空间复杂度$O(n)$。

4.3 【例题二】一个经典问题

给一棵仙人掌，要求支持加边、删边、对某两点之间最短路或某个子仙人掌进行修改或询问。

根为x时，子仙人掌y的定义是，删掉x到y的所有简单路径上的边后，y所在的连通块。

修改是将一条链或一个子仙人掌的点权加上一个数或修改为同一个数。

询问的是一条链或一个子仙人掌的点权最大值、最小值、和。

4.3.1 一个子问题

给一棵树，要求支持加边、删边、对某条路径或某个子树进行修改或询问。

可以用Self-adjusting top trees来做，时间复杂度为$O(n\log n)$。这个做法在[1]和[2]中有详细介绍。

4.3.2 算法一

我们用link-cut cactus来维护仙人掌的形态。

对于每个环，我们将环上的B点与额外链相连的边断开。（如果这个环上只有两个点，就不把这条边断开）

这样我们就得到了这棵仙人掌的一棵生成树，可以用Self-adjusting top trees（以下简称为“top tree”）维护。

对链进行操作时，在link-cut cactus上换根再access，然后在top tree上进行链操作即可。

对子仙人掌进行操作时，换根再access，然后这棵子仙人掌在top tree上就是一棵子树，直接进行top tree的子树操作即可。

在link-cut cactus的access操作时，会影响到$O(\log n)$个环的preferred-child，我们要更改那些环上被断开的边，于是要进行$O(\log n)$次top tree的link和cut操作，复杂度是均摊$O(\log^2 n)$的。

由于使用的是top tree，常数有点大。

4.3.3 算法二

考虑直接用link-cut cactus来维护。

类比top tree，可以在link-cut cactus的每个结点上维护子仙人掌的信息。我们将这个做法称为“top cactus”。

一个小技巧

我们将需要维护的信息记为`Info`，标记记为`Tag`，那么只要实现`Info+Info`，`Info*Tag`，`Tag*Tag`即可。注意`Tag`可以用a*x+b的形式来表示，可以简化代码。

在top cactus的每一个结点上，我们维护以下几个值：

- `x`
  表示该结点本身的信息
- `sum`
  表示以该结点为根的`splay`表示的链上的所有结点的信息和
  （不包含子仙人掌信息）
- `sub`
  表示以该结点为根的`splay`表示的链上的所有结点的子仙人掌信息和
  （不包含链上的信息）
- `ex`
  对于每个环的`AB`链的第一条边，用`ex`表示该环的额外链的`all`的值
- `all`
  `all = sum + sub + ex`
- `chain_tag`
  表示对以该结点为根的`splay`表示的链上的所有结点的标记
  （不包含子仙人掌标记）

- sub_tag
  表示对以该结点为根的splay表示的链上的所有结点的子仙人掌的标记（不包含链上的标记）
- ex_tag_sum
  表示该环从上次access到以来，AB链上打上的sub_tag的和

其中sub的值需要枚举每个该结点的每个子仙人掌来求出，由于一个结点的度数可能为$O(n)$，所以不能暴力求sub，应该对每个结点用一棵平衡树（如splay）来维护sub。

标记的下传

chain_tag只在splay内部进行下传。

sub_tag下传到子仙人掌时，被拆成chain_tag + sub_tag的形式。

ex_tag_sum维护的是环的*AB*链上已有而额外链上没有的标记，在access到每一个环的时候下传到该环的额外链上。

access操作

每次access到结点x时，

如果x不在环上：

- 将x原来的preferred-child与x相连的边改为虚边，并将该preferred-child加入维护子仙人掌的平衡树中。
- 将x的新preferred-child与x相连的边改为实边，并将该preferred-child从维护子仙人掌的平衡树中删除，并将子仙人掌标记传给它。

如果x在环上：

- 将环的上下两端断开，找到*AB*链上的ex_tag_sum，将其传给额外链。
- 将x提到splay的根，选择较短的一边接回去，并更新相应的信息和标记。

时间复杂度

link-cut cactus本身的复杂度为$O(n \log^2 n)$。

access循环的总次数为$O(n \log n)$，所以维护每个结点的子仙人掌信息的复杂度也是$O(n \log^2 n)$。

总时间复杂度为$O(n \log^2 n)$，但是常数比上一个算法要小。

4.4 【例题三】一个经典问题 EXT

给一棵仙人掌，要求支持加边、删边、对某两点之间最短路或某个子仙人掌进行修改或询问。

根为x时，子仙人掌y的定义是，删掉x到y的所有简单路径上的边后，y所在的连通块。

修改是将一条链或一个子仙人掌的点权加上一个数或修改为同一个数。

询问的是一条链或一个子仙人掌的**第k大**点权。

4.4.1 一个子问题

给一棵树，要求支持加边、删边、对某条路径或某个子树进行修改或询问第k大点权

top tree并不能用来维护第k大。

我们考虑维护链和子树信息的另一种经典做法：dfs序

由于有链修改和询问第k大，维护dfs序的数据结构可以是块状链表。

假设块状链表的块大小为T，那么在块状链表上一次分裂或合并操作需要$O(\log n + T)$的时间，一次询问操作需要$O(\frac{n}{T}\log^2 n)$的时间。（注意可以用平衡树来维护块的序列，就能做到$O(\log n + T)$的修改复杂度）

如果我们能将原问题的每个操作转化成$f(n)$个dfs序的分裂、合并操作和$O(1)$个询问操作，那么将块状链表的每块大小设为$O(\sqrt{\frac{n\log^2 n}{f(n)}})$，每个操作的时间复杂度就是$O(\sqrt{nf(n)\log^2 n})$。

树的形态不变，且树根固定

我们用link-cut tree来维护这棵树的一个链剖分，然后维护一个对应的dfs序列。

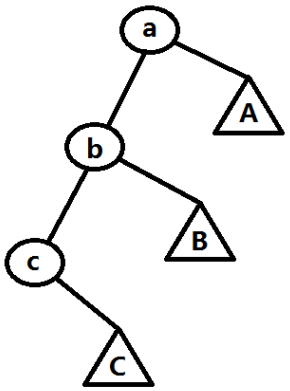
在这个dfs序列上，所有链剖分里的链在dfs序上都是一段区间。

我们在link-cut tree的access过程中对dfs序列进行相应的调整。

假设当前要将结点x的preferred-child改为y，就只要把子树y所对应的dfs序列的区间取出来，放到结点x后面。

这样在access了结点x之后，根到x这条链在dfs序上是个区间，子树x在dfs序上也是个区间，于是就能进行链和子树操作了。

树的形态不变，有换根操作



图中椭圆表示结点，三角形表示子树。

根为a时，这棵树的dfs序列为$a\ b\ c\ C\ B\ A$。（小写字母表示结点，大写字母表示子树）

根为c时，这棵树的dfs序列为$c\ b\ a\ A\ B\ C$。

假设我们要把根从a换到c。

首先access新根，进行了$O(f(n))$次dfs序操作。

然后只能用O(树高)次dfs序操作来换根？

不妨强行对原根到新根这条链所在的dfs序区间和dfs序剩下的部分打翻转标记。

这样dfs序列就变成$c\ b\ a\ r(A)\ r(B)\ r(C)$了。（$r(X)$表示将$X$这棵子树的dfs序翻转后得到的序列）

注意某些子树的整个dfs序列都反了。

我们在access的每一次循环中，检查一下是否发生了这种情况，如果有的话，就用$O(1)$次dfs序的操作把这个子树的dfs序列翻转回去。

这样就能做到$f(n) = O(\log n)$了。

树的形态可以改变

对于加边和删边操作，我们在换根、access相应的结点后，直接将整棵树的dfs序列接上去或分离出来即可。

仍然满足$f(n) = O(\log n)$。

于是对于这个子问题，我们能做到每个操作$O(\sqrt{n \log^3 n})$的复杂度。

4.4.2 仙人掌上的情况

我们还是用link-cut cactus来维护仙人掌的形态，一次操作可以转化成$O(\log n)$次树上的问题的操作。于是我们得到了$f(n)$的上界，为$O(\log^2 n)$。但是写个程序后会发现，$f(n)$更接近$O(\log n)$。

下面给出$f(n) = O(\log n)$的证明：

因为link-cut tree维护的是仙人掌的生成树，且维护时access到的点一样，于是在LCT上的链剖分与link-cut cactus维护的链剖分类似，那么在一次link-cut cactus的access中，除了LCT的第一次access以外，每一次LCT的access都只改变了1个环的链剖分，经过了$O(1)$条虚边，这就说明了，LCT的access循环总次数是$O(\log n)$，所以$f(n) = O(\log n)$，

于是这题就在每个操作$O(\sqrt{n \log^3 n})$的复杂度内解决了。

5 感谢

感谢CCF提供的学习交流的机会。

感谢教练徐先友对我的培养。

感谢帮助我完成论文的同学们。

参考文献

[1] 黄志翱, 浅谈动态树的相关问题及简单拓展, 2014年国家集训队论文

[2] Tarjan, Robert E., and Renato F. Werneck. “Self-adjusting top trees.”

浅谈图的匹配算法及其应用

长郡中学 陈胤伯

摘 要

图的匹配算法是信息学竞赛中常用的算法，有着广泛的应用。本文对图的匹配算法进行了总结，总体分为最大匹配和最大权匹配两部分，每部分先介绍二分图上的做法，再进一步提出一般图上的做法，并提出了一些拓展与应用。

1 匹配的有关定义

在图论中，设图 $G = (V, E)$，其中 V 是点集，E 是边集。

一组两两没有公共点的边集 $M(M \in E)$ 称为这张图的一个**匹配**。

定义一个匹配的大小为其中边的数量，边数最多的匹配是**最大匹配**。

当图中边带权时，边权和最大的匹配是**最大权匹配**。

匹配中的边称为**匹配边**，不在匹配中的边称为**非匹配边**。

我们称一个点在匹配中，当且仅当它是某条匹配边的端点。在匹配中的点称为**匹配点**，不在匹配中的点称为**未盖点**。

特别地，所有点都是匹配点的匹配，称为**完备匹配**。

对于匹配中的一条边 (u, v)，我们称 u 和 v 互为配偶。

如未特殊说明，接下来均用 n 来表示图的点数，m 来表示图的边数。

2 增广路

设 P 为图中一条简单路径，如果满足：

1. P 的起点、终点都是未盖点。

2. P 经过的边中匹配边、非匹配边交错出现。

则称 P 是一条**增广路**。

特别地，我们把满足条件 2 的简单路径称为**交替路**。

增广路上非匹配边比匹配边数量多 1，如果把这条路径取反[1]，则当前匹配大小 +1 且依然合法。我们称把增广路取反这个过程为增广。

沿增广路增广，路径上的所有点都将称为匹配点，也就是增广的这个过程会不断扩充匹配点的集合。

定理 1.

一个匹配 M 是图 G 的最大匹配，充要条件是 G 中不存在增广路。

证明 1.

由于取反增广路匹配大小 +1，必要性得证。

考虑 M 和一个更大的匹配 M' 的对称差 $D = M \oplus M' = (M \cup M') \setminus (M' \cap M)$。

根据匹配的定义可以推出 D 中每个点的度数至多为 2，且度为 2 的点两条邻边分别来自 M 和 M'。

这说明 D 由若干环、链组成，且每个环、链中 M, M' 的边交替出现。每个交替环长度一定为偶数，其中 M, M' 的边的数量相同。

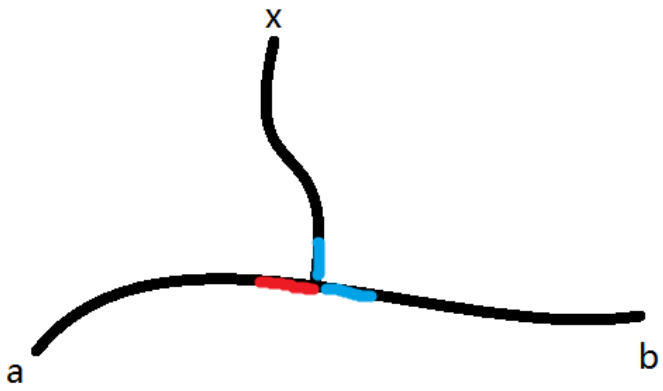
除去所有交替环，此时应该仍有 $|M'| > |M|$，则 D 中一定存在一条 M' 中的边出现得更多的交替链，该链对于 M 来说就是一条增广路。

因此，只要存在更大的匹配，就会有增广路，充分性得证。

由上述定理我们可以得出一个求最大匹配的算法框架：每次枚举图中所有未盖点，找增广路并增广，直到找不到增广路。

其实，如果某一次找不到从未盖点 x 出发的增广路，则增广多次后，依然没有从 x 出发的增广路。

[1]把路径上的匹配边变成非匹配边、非匹配边变成匹配边。为了描述方便，后文中的“把路径取反”、“把路径异或”均指这个意思。



假设某一轮沿增广路 $a-b$ 增广后，出现了以 x 为起点的增广路 P_x，则 P_x 一定是和路径 $a-b$ 有交。考虑 P_x 第一次碰上路径 $a-b$，由于 $a-b$ 是交替路，则触点两端的边类型不同，这意味着在增广前 x 就已经能走到 a,b 中的某个点，即某个未盖点，说明在此之前就已经存在从 x 出发的增广路，矛盾。

用这个结论可以对上述算法框架进一步优化：枚举每个未盖点，各找一次从它出发的增广路。

3 二分图最大匹配

3.1 基本算法

3.1.1 增广路算法

为了描述方便，我们把二分图的点分成左、右两部分，所有边都是横跨左右的。

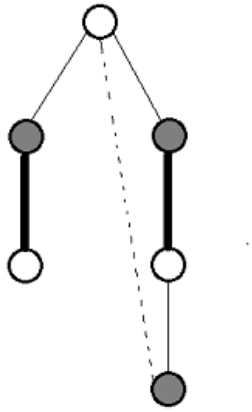
因为增广路长度为奇数，路径的起点终点一定分居左右两侧，不妨从左边的每个未盖点找增广路。

注意到增广路上第奇数次走非匹配边、第偶数次走匹配边，这说明左到右走的都是非匹配边、右到左走的都是匹配边。

因此我们可以给二分图按上述方式定向，问题变成了：有向图中，从给定起点找一条简单路径走到某个未盖点。不难发现有向图中“能通过简单路径走到”等价于“能够走到”，因此只需关心可达性问题。

给二分图定向后，枚举左边一个未盖点 u，从 u 开始 DFS，每个点记录一个访问标记，以保证至多访问一次，当找到某个未盖点时结束遍历。

因此找一次增广路的复杂度为 $O(m)$。

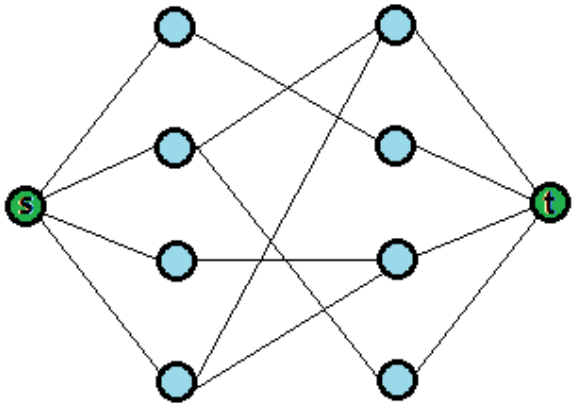


未找到增广路时，我们拓展出的 DFS 树也叫 **交错树**。

因为要枚举 $O(n)$ 个未盖点，上述算法的总复杂度是 $O(nm)$ 的。

3.1.2 Dinic 求最大匹配

二分图最大匹配可以转化为网络流的模型。



新建源汇，源向左边每个点连容量为 1 的弧，右边每个点向汇连容量为 1 的弧，原来的每条边从左往右连容量为 1 的弧。

最大流即为最大匹配，中间的满流边对应的是匹配边。

我们使用 Dinic 算法[1]，可以在 $O(m\sqrt{n})$ 的时间复杂度内求得最大匹配。

这个复杂度是如何得出的呢？

Dinic 算法的每一轮分为两步。第一步是用 $O(m)$ 的时间 BFS 建立层次图，第二步是用 $O(nm)$ 的时间 DFS 进行多路增广。

对于二分图模型，由于每条边容量都是 1，第二步 DFS 的复杂度可以做到 $O(m)$。

对于前 $\sqrt{n}$ 轮，总复杂度为 $O(m\sqrt{n})$。

$\sqrt{n}$ 轮后，因为 Dinic 算法每一轮重建层次图后 S 到 T 的最短增广路距离是严格递增的[2]，那么每条增广路的长度都不短于 $\sqrt{n}$。

换个视角，考虑最大匹配和当前匹配的对称差，即若干个互不相交的交替环、交替链。每一条交替链对应一条增广路，长度至少为 $\sqrt{n}$，因此这样的链的数量不超过 $\sqrt{n}$ 条。

这意味着当前匹配还剩 $O(\sqrt{n})$ 条增广路，也就是接下来 Dinic 的只会最多进行 $O(\sqrt{n})$ 轮，总复杂度为 $O(m\sqrt{n})$。

[1]一种网络流算法，考虑到大多数读者都会该算法，限于篇幅不在此展开详细介绍，不了解的读者可以参考其他资料自行学习。

[2]用反证法可以证明沿最短路增广一次不会产生更短的路，而 Dinic 一轮完毕意味着当前长度的增广路已全部增广完，接下来就只会有更长的了。

3.2 一些拓展

任意图中，设 M, M' 为两个不同的最大匹配，令 D 为 M, M' 的对称差。

则 D 由若干交替环、交替链组成，且由于 M, M' 都是最大匹配，环、链长度都为偶数（长为奇数的交替链代表有增广路）。

这说明，任意两个最大匹配，一个都可以通过异或上若干不相交的偶链、偶环得到另一个。

特别地，两个完备匹配的对称差只有若干不相交的交替环。因为某个匹配异或长度为偶数的交替链后，链的某端点将成为未盖点，说明不是完备匹配。

3.2.1 二分图最大匹配关键点

关键点是指的一定在最大匹配中的点。

由于二分图左右两侧是对称的，我们只考虑找左侧的关键点。

先求任意一个最大匹配 M，要找的关键点此时一定都是匹配点。考虑 M 中的一个匹配点 p，设 M' 为某个不包含 p 的最大匹配，对称差 $D = M \oplus M'$，则 D 中一定存在一条以 p 为端点的偶交替链，这条链另一端不在 M 中。

那么一个匹配点 s 能变成非匹配点，当且仅当从这个点出发能找一条以匹配边出发的交替链，使得终点是某个未盖点 t。由于链长为偶数，t 和 s 属于同一侧（左侧）。

我们倒过来考虑，先给二分图定向：匹配边从右到左、非匹配边从左到右，从左侧每个未盖点出发 DFS，给到达那些点打上标记。最终左侧每个没有标记的匹配点即为关键点。因为只关心可达性，显然每个点只需访问至多一次，复杂度 $O(n + m)$。

3.2.2 二分图最大匹配关键边

我们需要找到哪些边一定在最大匹配中。

同样地，先求任意一个最大匹配 M。

关键边一定是匹配 M 中的边，对于一条边 $e(e \in M)$，假设存在另一个不包含 e 的最大匹配 M'。

M, M' 的对称差中，e 一定存在，则要么属于一个偶交替链、要么属于一个偶交替环。

对于偶交替链的情况，链的某一端一定是未盖点，说明 e 的某个端点能通过交替路走到未盖点，即某个端点是非关键点。因此只需判定 e 两端是否存在非关键点。

对于偶交替环的情况，我们给二分图定向：匹配边从左到右、非匹配边从右到左，再检测 e 是否在某个环中（因为不存在奇环）。用 Tarjan 算法求强连通分量即可。

考虑复杂度，需要先 $O(n+m)$ 求关键点，再 $O(n+m)$ 用 Tarjan 算法求强连通分量。

3.2.3 二分图最大独立集

最大独立集是说，选最多的点，满足两两之间没有边相连。

我们求出二分图的最大匹配 M，根据匹配的定义，M 中的边两两没有公共点，而每条边两端点至少有一个不在独立集中，因此最大独立集的大小至多为 $n-|M|$。

接下来介绍一种构造出大小为 $n-|M|$ 的最大独立集的方法。

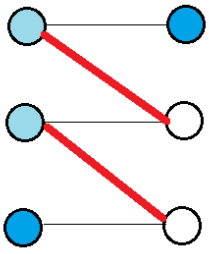
假设已经求出了最大匹配 M，先把所有未盖点加入最大独立集。

考虑一条匹配边 $e=(u,v)$，为了让独立集大小达到上界，e 的两端点 u,v 有且仅有一个要加入到最终的独立集中，不妨把这个“取左还是取右”的决策看做一个变量 x_e，通过枚举 u,v 的出边我们可以得到 x_e 对其他匹配边 e' 的 $x_{e'}$ 的决定关系。

把 $x_{1\sim|M|}$ 之间的决定关系图建出来后，再枚举每个未盖点 t，由于 t 已经被钦定到最大独立集中了，那么与 t 有边相连的所有匹配点所在的匹配边 e 的变量 x_e 的值将被确定。

这样建出来的推导图是否会产生矛盾呢？

如果产生矛盾，那么一定是如图所示的情况：



红色的表示匹配边，蓝色的表示被钦定到最大独立集中的未盖点，他们的值发生矛盾，是通过中间的匹配边们的决定关系来传递的，浅蓝色表示选择下面那个蓝点推导出来匹配边的 x 的取值，于上面发生矛盾。

这意味着找到了一条增广路，与 M 是最大匹配这一条件不符。

因此，我们建立的 $x_{1\sim|M|}$ 的推导图不存在矛盾。这样就可以赋值了，从每个被未盖点决定了 x_e 的匹配边 e 开始 DFS，把能确定的 x 都确定了，最后剩下的还未确定的 x_e 自由取值。

我们的推导图中点数是 $O(|M|) = O(n)$ 的；推导边是通过枚举原图中的边建出来的，所以是 $O(m)$ 的。

因此上述求二分图最大独立集的时间复杂度为 $O(n + m)$。

其实最大独立集也可以直接用网络流建模后最小割求，左边的点在 S 集表示选、右边的点在 T 集表示选，那么中间有边相连的自然就需要割掉了。

这也说明：二分图中，最大独立集 = n - 最大匹配。

3.2.4 二分图最小点覆盖

最小点覆盖是说，选最少的点，满足每条边都至少有一个端点被选。

根据定义不难发现：任意点覆盖的补集都是独立集、任意独立集的补集都是点覆盖。

那么在所有点中去掉最大独立集，得到的就是最小点覆盖。

这也说明：二分图中，最小点覆盖 = n - 最大独立集。

3.2.5 二分图最小字典序完备匹配

最小字典序是指：令 $match_i$ 表示与左边第 i 个点匹配的右边的点的编号，最小化 $match_{1\sim n_L}$ 的字典序。

之前提到过，任意两个完备匹配的对称差一定是若干个交替环。那么我们先求任意一个最大匹配，即一个完备匹配。

根据字典序贪心的思想，按编号从小到大枚举左边每个点 u 来决定 $match_u$。

我们从小到大枚举右边每个和 u 有边相连的点 v，每次需要检验边 (u, v) 能否成为匹配边。若其为匹配边则显然能；否则，给二分图定向后等价于求边

(u, v) 是否在一个环里，即询问 u, v 是否属于同一强连通分量，如果是则说明能、否说明不能。

对于上述每个点 u，找到最小的能匹配的 v 即可确定匹配关系。决定了 $match_u = v$ 后，为了防止再次被修改，我们不妨把 u、v 从二分图中删去。

匹配关系只会变动 $O(n)$ 次，因此只需求 $O(n)$ 次强连通分量。对于某个 u，找到合法的 v 后，找交替环用 DFS 是 $O(m)$ 的。

因此时间复杂度为 $O(nm)$。

3.3 应用

我们知道了在二分图中，最大匹配 = 最小点覆盖 = n - 最大独立集。

那么二分图模型的上述问题都能解决。

常见的二分图模型有棋盘的行列、人为的黑白染色，或者能证明出无奇环的图等。

这一类题目十分常见，相信读者已经见过很多，在此只进行总结而不加赘述。

下面介绍一些思路较为巧妙的二分图匹配的题目。

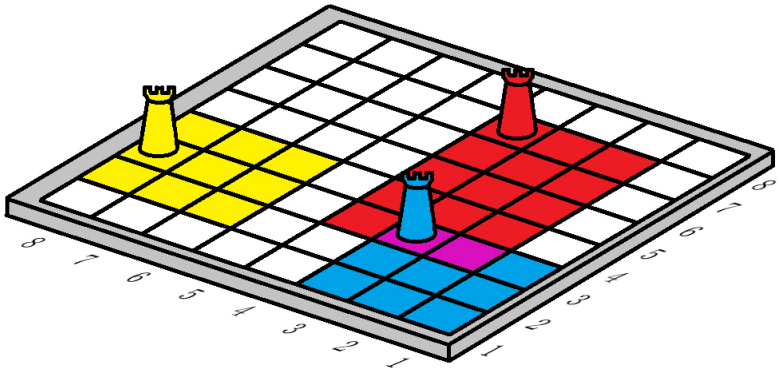
3.3.1 Problem 1. Chessboard

题意 [1]

有一个 $n \times n$ 的棋盘和 n 个车，你要将 n 个车放进棋盘中使得每行每列都至多有一个车。$n \leq 10^5$。

出于某些原因，每个车的坐标范围限制在一个矩形区域内。

[1] 来源：POJ 挑战赛 Round 2 - by sumix173



请你求出哪些车的摆放位置是唯一确定的。

题解

首先题中的 x, y 坐标是独立的，我们只看一维。

问题变成一个排列 P，数字 x 出现的位置限定在 $[L_x, R_x]$，求哪些数字出现位置固定。

我们建立二分图，左边对应数字，右边对应排列位置，左边每个点向右边一个区间的点连边。我们等于询问哪些边一定在最大匹配中。

这是之前提到过的“关键边”问题。

注意为了保证有解，这个二分图一定有完备匹配。也就是说“求关键点”这一步可以省略。那么我们只需要像之前说的，给二分图定向（匹配边向左、非匹配边向右），然后判定每条边是否在环里。

我们用 Tarjan 求强连通分量，如果一个表示数字的点所在的强连通分量大小大于 1，说明这个点的出边在环内，说明这条出边不是“关键边”，即这个点所代表的数字的位置不确定。

由于是区间连边，又只关心连通性，可以用对右边的点建立线段树结构来优化连边，这样边数是 $O(n \log n)$ 的。

至于本题的求完备匹配，这是一个经典的贪心问题，即所有数字按 L 排序然后从左到右枚举每个位置，每次填入 R 最小的数字，即得到一组方案。

时间复杂度 $O(n \log n)$。

3.3.2 Problem 2. Minimum Diameter

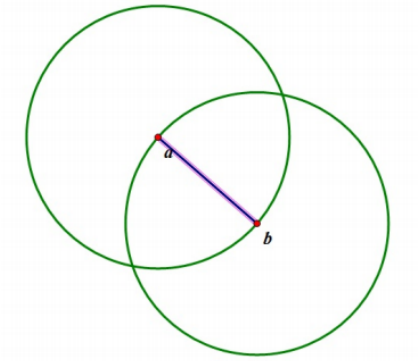
题意 [1]

给平面上 $n(n \le 1000)$ 个点，删除其中 $k(k < n, k \le 30)$ 个点，使得剩下的点中相距最远的点对最近。

题解

枚举剩下来的点里相距最远的两个点 a, b。

以 $|ab|$ 为半径，a, b 为圆心画圆。



那么剩下的点 x 需要满足 $|xb|, |xa| \le |ab|$，也就是一定在两圆的交集内部。

接下来，我们在两圆交集内的点中找到所有距离 $> |ab|$ 的点对，连上边，表示不能同时选。最大独立集即为内部最多能保留的点，检查一下删除的点总数是否 $\le k$ 即可。

注意到每条边一定是跨越了两圆交集的两瓣的中线 ab，因为同一瓣内不存在距离 $> |ab|$ 的点对。所以这实际上是一个二分图最大独立集问题。

由于最大独立集 = n - 最大匹配，当发现最大匹配 $> k$ 时，说明最大独立集 $< n - k$，说明需要拿走超过 k 个点，这是不可行的，因此可以直接终止这轮匹配。

找一次增广路是 $O(m) = O(n^2)$ 的，至多找 k 次增广路就会终止算法。再考虑之前枚举 a, b 的复杂度，总时间复杂度 $O(kn^4)$。

尽管这个复杂度不能通过此题，但也是一个多项式级别的不错的算法，思想十分巧妙。

[1] 来源：CodeForces VK Cup 2012 Round 3 D

完整做法由于和主题关系不大且限于篇幅在此不深入讨论，有兴趣的读者可以参考陈老师 2012 年自选题解题报告（见参考文献）。

3.3.3 Problem 3. The Square Div One

题意 [1]

一个 $n \times m(n, m \leq 50)$ 的矩阵，你需要给每个格子填上 $1 \sim \max(n, m)$ 的整数，使得每行每列不存在重复数字。

求字典序最小的方案。（字典序比较按先行后列的顺序）

题解

根据字典序的定义，一个直接的想法是：从上到下一行行确定数字。

考虑这一行每个位置，根据所在列已出现的数字，将每个位置与可以填的数字之间连边，求字典序最小的匹配。

遗憾的是，这样做可能导致之后几行无解。因为一个数字必须出现 $\max(n, m)$ 次，如果前几行填得太少，后几行可能行数不足以让它填完所需次数。

如何修复这个问题呢？

考虑正在填某一行，令：$P = \max(n, m)$（数字种数）、r_x 表示数字 x 还需要出现多少次、h 表示还剩多少行未确定。

则有解当且仅当：

1. $\sum_{i=1}^{P} r_x = h \times m$
2. 对于任意 x 满足 $r_x \leq h$

必要性是显然的。充分性的证明：当 $h = 1$ 时显然成立；当 $h > 1$ 时，我们强制填 $r_x = h$ 的那些 x，由于 $\sum_{i=1}^{P} r_x = h \times m$，那么 $r_x = h$ 的 x 一定不超过 m 个，也就是说这一行位置是充足的。既然如此，填完到下一行时，发现条件依然成立，递归证明即可。

所以每次只要保证把 $r_x = h$ 的 x 填一次即可。

考虑二分图模型。右边 P 个点表示数字，左边 m 个点表示位置。

一个数字能填在某个位置就连一条边。

[1] 来源：Topcoder - SRM 459

若 $P > m$ 则在左边补一些点表示选择不填，右边满足 $r_x < h$ 的 x 向新建的点连边。

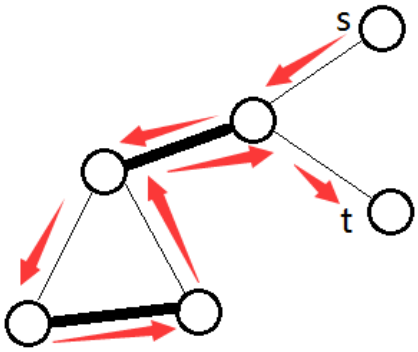
用之前说的做法求这个二分图的最小字典序完备匹配。

4 一般图最大匹配

4.1 带花树算法

一般图和二分图的唯一区别在于可能存在奇环，这导致我们不能转换成有向图可达问题。

能不能还是只关心走交替边的可达性呢？



很遗憾并不行，如上图所示，虽然 s 到 t 可达，但由于不是简单路径，取反后将使得某些点在两条匹配边上，这是非法的。

我们发现问题出在奇环。如果上图的环是偶环，即使只关心可达性我们也不会绕回同一个点，因为绕偶数步回到同一个点不改变下一步走的边的类型。

下面考虑一般图的增广路算法。

不妨先当做二分图来做，每次枚举一个未匹配点 s，从 s 开始按交替的规则 DFS 找增广路。

我们对走出的这棵交错树中的点奇偶标号，其中根为偶点。不难发现偶点是被奇点直接用匹配边“拉进来”的，每次我们枚举的是偶点的出边。

设当前所在点为 u，枚举的下一个点为 v。

如果 v 未访问过：若 v 是未盖点则找到增广路；否则从 v 的配偶开始 DFS。

如果 v 已被访问：则当前枚举的出边是交错树的非树边，找到了一个环。

如果找到的是偶环，直接忽略。因为只有偶环的图就是二分图，算法正确性之前已经证明过了。

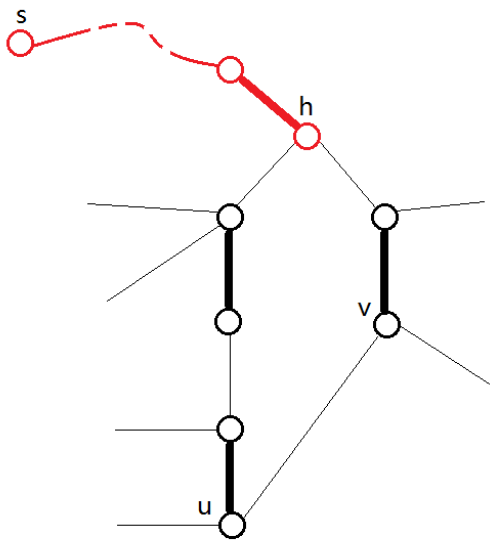
如果找到的是奇环，我们把环上所有边缩掉，即把整个环缩成一朵花（一个点）。然后在新图中重新找增广路。

下面来分析这样做的理由。

设原图为 G，缩点后得到的新图为 G'，我们只需要证明以下两点：

1. 若 G 存在增广路，则在 G' 中也存在。
2. 若 G' 存在增广路，则在 G 中也存在。

在证明之前，我们先画出交错树和一条非树边，即找到的那个奇环。



设非树边为 (u,v)，定义花根 $h = LCA(u,v)$。

奇环一定是交替的，有且仅有 h 的两条邻边类型相同，都是非匹配边。

那么进入 h 的树边一定是条匹配边，环上除了 h 外其他点往环外的边都是非匹配边。

观察可知，沿匹配边进入 h 可以到环上其他任意点沿非匹配边出去（顺时针、逆时针总有一种走法可行），反之亦然。

我们称缩成的点为新点，被缩的环为旧环。

若 G 存在增广路，则在 G' 中也存在。

设之前图中 $s-h$ 这一段红色路径为 S。

令 G_S, G'_S 分别表示 G, G' 中把路径 S 取反后得到的新图，显然 G_S, G'_S 相对于自己的原图来说匹配合法且大小不变。

既然 G 存在增广路，说明 G 中的匹配不是最大的，也就说明了 G_S 中的匹配不是最大的。根据增广路定理，G_S 中也应该存在增广路。

设 G_S 中的增广路为 P，若 P 没有经过旧环，则 G'_S 中也存在 P。

否则，让 P 刚走上旧环就径直沿着环走到 h，在 G_S 中 h 是未盖点，那么这样走也是一条合法的增广路，把这条增广路对应到 G'_S 中，相当于直接走到了新点，也是合法的增广路。

又因为 G'_S 和 G' 的匹配大小相同，既然 G'_S 存在增广路，那么 G' 也应该存在增广路。

若 G' 存在增广路，则在 G 中也存在。

G' 中的增广路，如果没走新点，那么 G 中显然存在一条一模一样的。

如果走了新点，无非是一进一出，进出分别是一条匹配边和非匹配边。

考虑把这两条边对应到 G 中去，匹配边一定是和 h 相连的那一条。

我们之前已经分析过了 h 可以到旧环上任意一点，反之亦然。那么 G' 中对新点的一进一出，就可以根据进出边在旧环上的位置，还原出在 G 中旧环上的走法，得到 G 中的增广路了。

进一步地可以发现：如果当前是从 s 出发找增广路碰到了奇环，那么如果存在从 s 出发的增广路，缩环之后一定还存在从 s 出发的增广路。因此缩点之后，找增广路的起点不变。

在找到奇环的时候，为了实现方便，我们不“显式”地缩点，而是用并查集来记录每个点现在在以哪个点为根的花中。在记录匹配关系的同时，用一个 *next* 数组记录最终增广的时候每个点的后继，每朵花上的 *next* 实际组成了一个双向链表来描述花内部的走法。主程序用一段 BFS 来拓展交错树，队列中存所有偶点，在碰到奇环时，用并查集完成缩点并将环上所有奇点标记为偶点加入 BFS 的队列。这样可以做到一次找增广路的时间为 $O(n^2)$。

一共要找 $O(n)$ 次增广路，因此总复杂度为 $O(n^3)$。

4.2 应用

4.2.1 Problem 1. 无向图游戏

题意

Alice 和 Bob 在一张 $n(n \le 200)$ 个点的简单无向图上玩游戏，他们轮流操作，其中 Alice 先手。

最初，有一个小人在某个点 s 上。每次，设小人在点 u，玩家可以让小人走到和 u 相邻的某个点 v 上去。小人走过的点会被炸掉，从图中永久删去。不能操作者输。

已知 Alice 和 Bob 足够聪明，求有多少个 s 是先手必胜。

题解

结论是：若存在一种最大匹配，满足 s 不是匹配点，则先手必败；否则先手必胜。

首先，若存在一种最大匹配满足 s 不是匹配点，那么先手每走一步（设走到了 v），后手只需走到 v 的匹配点 u 即可。这样后手总是有路可走，除非走到某个未盖点，但注意如果这种情况发生，我们实际上找到了一条从未盖点到未盖点的交替路，即增广路，这与之前所说的最大匹配的条件矛盾。

若不存在，即 s 一定在最大匹配中，那么任意找一个最大匹配，然后先手使用上一段描述中后手的策略。这样若先手无路可走，说明到了某个未盖点，走出了一条只有一端是未盖点的交替路，但这条路一旦取反，s 将不再在匹配中而匹配大小不变，说明 s 并非一定在最大匹配中，与之前的条件矛盾。

现在问题变成了：在一般图里，判定哪些点一定在最大匹配中。

我们任意找一个最大匹配，枚举每个匹配点，把它从图中删去，看看是否存在增广路，如果是则说明该点不一定在最大匹配中。

注意删去一个点 u 后，如果存在增广路，u 的配偶 v 一定是增广路的一端。否则，考虑删点时匹配数 -1，增广后匹配数 $+1$，此时 u,v 还能匹配起来匹配数再 $+1$，得到了一个比之前求出的最大匹配更大的匹配，矛盾。

因此对于枚举的 u，我们只需从 v 开始找增广路即可。

首先求一次最大匹配是 $O(n^3)$ 的。之后枚举一个匹配点 $O(n)$，删去后用带花树可以在 $O(n^2)$ 的时间里判定是否存在从某个点出发的增广路。

时间复杂度 $O(n^3)$。

4.2.2 Problem 2. 拓扑图拿点

题意

有一张 $n(n \le 200)$ 个点的拓扑图，每轮你可以同时拿走至多两个入度为 0 的点（入度在每一轮结束后才重新结算，拿走的两个点不能有拓扑关系），求至少多少轮能拿走所有点。

题解

考虑拓扑图上任意两个有拓扑关系的点 u,v，即 u,v 中一个可以到达另一个，它们显然不能在同一轮被拿走。那么我们建立新图 G'，在所有没有拓扑关系的点对之间无向边，表示这条边两端点可以在一轮被同时拿走。

由于点数一定，要最小化拿完全部的轮数，等价于最大化一轮拿两个的轮数 R。

考虑 G' 中的最大匹配，显然是 R 的上界。如果这个上界可以达到，也就求出了最大的 R。

下面我们证明，这个上界是一定可以达到的。

首先，考虑原拓扑图中所有入度为 0 的点，再考虑 G' 中的匹配关系。

如果某个点没有配偶，直接拿掉。

如果某个点的配偶入度也为 0，一起拿掉。

现在情况只剩下：所有入度为 0 的点配偶都被“压着”了。考虑这些配偶们之间的拓扑序，这些配偶们中处于拓扑序顶点的（即不被其他配偶压着）配偶一定有至少两个。

否则，设那个唯一处于顶点的为 u，由于 u 在原拓扑图入度不为 0，一定存在一个入度为 0 的点 v 能到达 u，既然 v 压着 u，那么 v 一定也压着了 v 自己的配偶，而根据定义 v 是不能到达 v 的配偶的，因此矛盾。

既然至少有两个，设她们为 x,y，和她们匹配的入度为 0 的点为 X,Y，原来匹配关系是 $x-X, y-Y$，我们改成 $X-Y, x-y$，其中 X,Y 都入度为 0 所以可以一并拿去，x,y 之间不存在拓扑关系所以可以匹配起来。

这样每次可以调整最大匹配使得大小不变且合法，并且总能调整出一条匹配了两个入度为 0 的点的匹配边，那么就可以做到刚好 R 轮拿走两个了。

上述证明事实上也提供了一种找合法解的方法。

时间复杂度 $O(n^3)$。

5 二分图最大权匹配

最大权匹配，是指边权和最大的匹配。

如果我们在二分图左右两边点数较少的一边补点，使得左右两边点数一样，再将不存在的边看做权值为 0，那么最大权匹配问题可以转化为最大权完备匹配问题。

接下来我们讨论最大权完备匹配问题，即找一个边权和最大的匹配，满足每个点都在匹配中。

5.1 KM 算法

KM 算法又叫匈牙利算法，可以在 $O(n^3)$ 的时间求出最大权完备匹配。

我们先介绍两个概念。

可行顶标是指给每个节点 i 分配一个权值 $l(i)$，对于所有边 (u,v) 满足

$$w(u,v) \le l(u) + l(v)$$

相等子图是指在一组可行顶标下原图的生成子图，包含所有点，只保留满足 $w(u,v) = l(u) + l(v)$ 的边。

定理 2.

对于某一组可行顶标，如果其相等子图存在完备匹配，那么该匹配一定是原二分图的最大权完备匹配。

证明 2.

考虑原二分图任意一组完备匹配 M，其边权和

$$val(M) = \sum_{(u,v)\in M} w(u,v) \le \sum_{(u,v)\in M} l(u) + l(v) \le \sum_{i=1}^{n} l(i)$$

任意一组可行顶标下的相等子图的完备匹配 M' 的边权和

$$val(M') = \sum_{(u,v)\in M'} l(u) + l(v) = \sum_{i=1}^{n} l(i)$$

即任意一组完备匹配的边权和都不会比 M' 的大，那么 M' 就是最大权完备匹配。

有了上述定理，我们的目标就是通过不断调整顶标，使得相等子图中存在完备匹配。

为了方便描述，我们作如下定义。

由于左右两边点数相等，令 n 表示一边的点数。令 lx_i 表示左边第 i 个点的顶标，rx_i 表示右边第 i 个点的顶标，$w(u,v)$ 表示左边第 u 个点和右边第 v 个点之间边的权值。

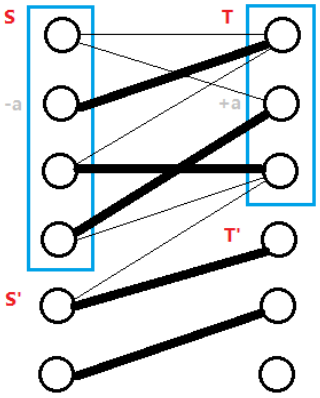
先初始化一组可行顶标，比如

$$lx_i = \max\{\, w(i,j) \mid 1 \le j \le n \,\}, rx_i = 0$$

然后每次选一个未盖点，像做最大匹配那样试着找一条增广路。

如果找到了增广路就直接增广。

否则，我们将得到一棵之前所说的交错树。令 S、T 表示二分图左边、右边在交错树中的点，S'、T' 表示二分图左边、右边不在交错树中的点。



在相等子图中：

- S 到 T' 一定没有边。因为若存在非匹配边，交错树应该会继续生长；S 中所有点要么是起点 s、要么和 T 中匹配，所以匹配边显然没有。
- S' 到 T 一定都是非匹配边，否则 T 中对应点会从匹配边遍历到 S'。

如果我们给 S 中所有点顶标 $-a$，T 中所有点顶标 $+a$，那么原来的匹配边、$S-T$ 的交错树中的边一定都还在相等子图，$S-T'$ 中可能有新的边加入相等子图。

不难推出：

$$a = \min\{\ lx_u + ly_v - w(u,v) \mid u \in S, v \in T'\ \}$$

如果 a 更大，顶标将非法；如果 a 更小，相等子图将不会变化。

当一条新的边 (u,v) 加入相等子图后，有两种情况：

1. v 是未盖点，则找到增广路。

2. v 和 S' 中某个点已经匹配，则将 v 和它的配偶都加入交错树。

每次修改顶标后，要么找到增广路，要么交错树中节点 +2。这说明至多修改 $O(n)$ 次顶标就能找到增广路。

每次修改顶标时，交错树中的边并不会离开相等子图，因此我们实质上是在扩展这棵交错树，考虑直接维护这棵树。

我们对 T' 中每个点 v，维护

$$slack(v) = \min\{\ lx_u + ly_v - w(u,v) \mid u \in S\ \}$$

以便每次 $O(n)$ 算出顶标修改值

$$a = \min\{\ slack(v) \mid v \in T'\ \}$$

在交错树新加入一个 S 节点的时候，$O(n)$ 更新一下 $slack(v)$。

在修改顶标的时候，$O(n)$ 给每个 $slack(v)$ 减去 a。

只要交错树扩展到某一个未盖点，那么就找到了增广路。

回顾一下，最初枚举一个未盖点是 $O(n)$ 的，每次为了找增广路要扩展 $O(n)$ 次交错树，每次扩展需要 $O(n)$ 维护，总复杂度 $O(n^3)$。

5.2 应用

首先可以解决最大权匹配、最大权完备匹配的问题。

通过把边权取负，也可以解决最小权匹配问题。

KM 算法的顶标模型

之前的二分图模型中，找一组 lx, ly 满足：

$$lx_i + ly_j \geq w(i,j)$$

最小化：

$$\sum_{i=1}^{n} lx_i + \sum_{i=1}^{n} ly_i$$

KM 算法求出的顶标即为答案，这是因为任意合法顶标的顶标和都会大于等于最大权完备匹配的边权和，而 KM 求出的顶标刚好是这个下限。

6 一般图最大权匹配

6.1 带权带花树

类似二分图中 KM 算法的顶标，我们给每个节点 i 定义一个非负顶标 z_i、给每个奇数点集 B 定义一个非负顶标 z_B。

对于一条边 $e = (u, v)$，令 $f_e = z_u + z_v + \sum_{B \mid u,v \in B} z_B$，$w_e$ 表示 e 的边权。

如果对于某个匹配 M，满足：

1. $f_e \geq w_e$ for $e \in E$
2. $f_e = w_e$ for $e \in M$
3. $z_i = 0$ 若 i 是未盖点, $z_B = 0$ 若 B 中匹配边不足 $\frac{|B|-1}{2}$

那么 M 一定是最大权匹配。

因为对于任意匹配 M'，其边权和一定不超过：

$$\text{LIMIT} = \sum_{i=1}^{n} z_i + \sum_{B} z_B \times \frac{|B| - 1}{2}$$

我们把边权 w_e 放大成 f_e 求和，上式中前者是节点顶标贡献的上界，后者是点集顶标贡献的上界。

而注意到我们的匹配 M 的边权和刚好等于 LIMIT，因此 M 一定是最大权匹配。

（这里看起来 B 的定义十分古怪，后面会看到 z_B 实际上是对每个缩起来的花维护的一个权值。这里只需理解其正确性。）

现在看来我们的目标在于，同时找到一组顶标和一个匹配 M，满足上述条件。

称满足条件 1, 2 的一组顶标是合法的。

称所有 $f_e = w_e$ 的边 e 组成的图为**相等子图**。

初始化当前匹配 $M = \emptyset$，$z_i = \max w_e/2$。一开始所有 $z_B = 0$，我们只记录和维护那些 $z_B > 0$ 的点集。

注意到初始的状态只违背了条件 3。

我们的算法过程将在满足条件 1、2 的前提下，不断减少违背条件 3 的点的数量。

算法和求最大匹配时一样，分为 n 个阶段，每个阶段我们在相等子图中找增广路，和求最大匹配时一样，不过我们从所有未匹配点一起开始 BFS，把点奇偶染色（所有根均为偶点），用一个队列 Q 存所有偶点，缩花后花内所有点标记为偶点并把新的偶点加入队列，如果找到了增广路则直接增广，否则调整 z。

为了保证条件 3，我们在一个阶段结束后需要保护那些 $z_B > 0$ 的缩起来的花 B，也就是不展开。因此我们在 BFS 交错树的时候，把花看做点，除了偶花还可能会碰到奇花。

如果相等子图中没有找到增广路，像二分图 KM 算法中那样调整 z 的值，即：

- $z_i -= a$，如果 i 是偶点。
- $z_i += a$，如果 i 是奇点。

然而一个花内部所有点类型相同，花内的边两端都 $+a$ 或都 $-a$，势必会造成这条边离开相等子图，甚至顶标不合法，这时就要用到之前定义的 z_B 了。

- $z_B += 2a$，如果 B 是偶花。
- $z_B -= 2a$，如果 B 是奇花。

这里说的花均指最外层的花（因为花可能里面套着花），这样就可以保持花内的边依然留在相等子图了。

下面我们需要考虑 a 的取值，不难发现是下面四种取 min：

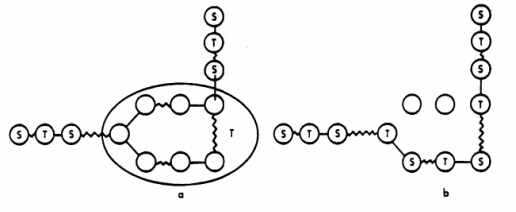
1. z_i，i 是偶点。
2. $f_e - w_e$，$e = (u, v)$、u 是偶点、v 是未访问点。
3. $(f_e - w_e)/2$，$e = (u, v)$、u, v 是不在同一个花里的偶点。
4. $z_B/2$，B 是奇花。

容易发现这样的 a 不会使花中的边、交错树上的边、匹配边从相等子图中消失，也不会使顶标非法。

我们按 a 最终是在哪种情况中取到 min 的来分类。

如果是情况 1，那么这次改变后条件 3 将可以得到满足。

如果是情况 4，那么会有若干奇花的权值变成 0。奇花肯定不是当前这一轮形成的，一定是前几轮留下的，之前保留的理由是为了满足条件 3，现在它权值为 0 了，我们就可以把它展开了。



其中 a, b 分别表示展开前、展开后。展开后这朵奇花的子花就成了当前交错树中的花。把所有新的偶点加入队列 Q。

如果是情况 2 或者 3，都将有至少一条新的边加入相等子图中。如果某两个不在同一棵树中的偶点之间加入了一条边，则找到增广路。

这样我们总会找到增广路。这一阶段结束后，再把交错树上所有权值为 0 的偶花展开，进入下一阶段。

时间复杂度 $O(mn^2)$。

用类似二分图中的维护 *slack* 变量的方法可以做到 $O(n^3)$。

6.2 应用

6.2.1 无向图中国邮递员问题

题意

给定一张 $n(n \le 200)$ 个点的简单无向图，边有边权。找一条最短的封闭路径，经过所有边至少一次。

题解

我们相当于要加最少的边，使得原图存在欧拉回路（即每个点度数都是偶数），其中新加的边的边权为两点间的最短路。

用 Floyd 算法 $O(n^3)$ 求出所有点的两两最短路，问题转化为度数为奇数的点之间的最小权完备匹配，把边权都用一个大数减去后转化为最大权匹配。

7 总结

我们讨论了四种类型的最大匹配问题，从最大匹配到最大权匹配，从二分图到一般图，每一类问题都介绍了一些多项式算法。

我们发现，匹配算法的核心在于找增广路，最大权匹配通过顶标模型一定程度可以转化为最大匹配问题从而得以解决。

事实上，图的匹配的内容远不止这些，还有很多更为高效的算法以及一些对匹配问题的研究历程本文中没有介绍，有兴趣的读者可以自行研究，希望本文能起到抛砖引玉的作用。

8 感谢

感谢中国计算机学会提供学习与交流的平台。

感谢吕凯风、黄志翱、郭晓旭、罗雨屏、余行江、胡家琛、张天扬同学在论文写作上对我的帮助。

感谢向期中教练、徐光明老师对我的栽培。

感谢长郡中学的同学们对我的帮助与支持。

感谢我的亲人对我的关心和照顾。

参考文献

[1] Wikipedia，Berge’s lemma

[2] Wikipedia，Hopcroft－Karp algorithm

[3] Matthew Kusner，《Edmonds’s Blossom Algorithm》

[4] Zvi Galil，《Efficient Algorithms for Finding Maximal Matching in Graphs》

[5] 刘汝佳，《算法竞赛入门经典》，清华大学出版社

[6] 陈立杰，《CodeForces VK Cup 2012 Round 3 D. Minimum Diameter》

[7] 黄志翱，《最大权匹配》

[8] 范浩强，《无向图匹配的带花树算法》

浅谈信息学竞赛中的物理问题

杭州第二中学 陈思禹

摘 要

随着近年来学科竞赛的发展，各个学科的关系变得更为密切。一些物理问题也开始出现在信息学竞赛的比赛中。信息学竞赛中的物理问题所用到的物理知识不会太复杂，但给出的数据往往带有信息学色彩，即数据量大而复杂，不太可能用人脑计算。不过经过一些物理推导往往可以使问题大大简化。本文将分别分析信息学竞赛中的运动学、动力学、静力学和电磁学问题，通过一些例题分析其中蕴含的物理知识以及从物理问题到信息学问题的转化关系。

1 信息学与运动学

“运动学”这一名称和后面要介绍的“动力学”很相似，但是并不是同一类问题。运动学问题并不涉及物体的受力，主要侧重于位移、速度、加速度、时间之间的关系。信息学中的运动学问题一般需要先从物理上对运动进行分析，而且数据比较复杂，需要用计算机辅助计算。其中需要的许多知识在中学的物理教材中已有涉及，这里就先简单介绍一下。

位移 速度 加速度

物体在运动过程中的位置$\boldsymbol{r}$（加粗表示矢量）随时间t的变化关系可表述为：$\boldsymbol{r}=\boldsymbol{r}(t)$。从$\boldsymbol{r}(t)$到$\boldsymbol{r}(t+\Delta t)$ 引一矢量称为位移矢量，简称位移。即：

$$\Delta\boldsymbol{r}=\boldsymbol{r}(t+\Delta t)-\boldsymbol{r}(t)$$

$\Delta\boldsymbol{r}$给出的是一段时间内质点运动的总效果，可以引入平均速度$\bar{\boldsymbol{v}}=\dfrac{\Delta\boldsymbol{r}}{\Delta t}$。为了更细致地描述运动，取$\Delta t\to 0$，可以得到瞬时速度，即速度：

$$\boldsymbol{v}=\frac{d\boldsymbol{r}}{dt} \tag{1}$$

同样的，我们可以引入瞬时加速度，即加速度：

$$\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt} = \frac{d^2\boldsymbol{r}}{dt^2} \tag{2}$$

平面曲线运动

质点在一个平面上运动往往涉及到坐标系的分解，下面将介绍常见的两种方法。

直角坐标系分解

对于一个固定的直角坐标系Oxy，质点的位置矢量$\boldsymbol{r}$可以分解为：$\boldsymbol{r} = x\boldsymbol{i} + y\boldsymbol{j}$，因此这一运动方程有两个分量式:

$$x = x(t),\ y = y(t) \tag{3}$$

这就意味着平面运动正交分解为了两个直线运动。同样地我们有：

$$v_x = \frac{dx}{dt},\ v_y = \frac{dy}{dt}, \tag{4}$$

$$a_x = \frac{dv_x}{dt},\ a_y = \frac{dv_y}{dt} \tag{5}$$

当然速度也可以用两个分速度来合成，即：

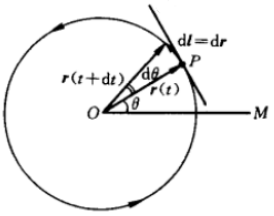
$$v = \sqrt{v_x^2 + v_y^2} \tag{6}$$

圆周运动

圆约束是一种常见的轨道约束。建立极坐标系，质点位置可以由相对圆心转过的角θ和圆的半径R唯一确定。这样我们即可引入角速度ω和角加速度β的概念，即：

$$\omega = \frac{d\theta}{dt}, \tag{7}$$

$$\beta = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2} \tag{8}$$



现在我们再来讨论位移。圆运动中位移亦为的$d\boldsymbol{r}$，常常改为$d\boldsymbol{l}$，可以分解为切向和法向（法向会引起半径的变化，但是这里讨论的是无穷小量，可以忽略），其沿切线方向的投影设为dl_τ。利用几何关系$dl_\tau = Rd\theta$，并结合$\boldsymbol{v} = \dfrac{d\boldsymbol{l}}{dt}$，其沿切线方向的投影设为$v_\tau$，可得：

$$v_\tau = \frac{dl_\tau}{dt} = \frac{Rd\theta}{dt} = R\omega \tag{9}$$

利用$\boldsymbol{v} = \boldsymbol{v_\tau} + \boldsymbol{v_r}$即可得到法向分量$v_r$。至于加速度这里就不作详细讨论。

下面就将利用以上的公式分析几个比较经典的运动学问题。

1.1 Just a Joke [1]

1.1.1 题目大意

在一个半径为R的圆上G点在以v_1的速率匀速圆周运动。B点从圆心O出发以恒定速率v_2（$v_2 \geq v_1$）追赶，且任意时刻G、B、O始终保持在同一直线上。求B点追上G点所需经过的路程d。

1.1.2 分析

设B点运动的时间为T。由于B点作匀速运动，$d = v_2 \cdot T$。

虽然这并不是一个圆周运动，但G点作圆周运动且B点角速度与G点相同。不妨考虑也将B点的运动分解为径向运动和切向运动。G点转动的角速度$\omega = \dfrac{v_1}{R}$。所以B点位于距离圆心r处时，B点的切向速度$v_\tau = \omega \cdot r = \dfrac{v_1 \cdot r}{R}$。

由于B点速率为v_2，其径向速度$v_r = \sqrt{v_2^2 - v_\tau^2} = v_2 \cdot \sqrt{1 - \dfrac{v_1^2 r^2}{v_2^2 R^2}}$。利用$dt = \dfrac{dr}{v_r}$，

[1] hdu 4969, 2014 Multi-University Training Contest 9

B点的运动时间可以通过下式得出：

$$T = \int_0^R \frac{dr}{v_r} = \left.\frac{R \cdot arcsin\left(\frac{v_1 r}{v_2 R}\right)}{v_1}\right|_0^R = \frac{R \cdot arcsin\left(\frac{v_1}{v_2}\right)}{v_1} \tag{10}$$

因此$d = v_2 \cdot T = \dfrac{R \cdot v_2 \cdot arcsin\left(\frac{v_1}{v_2}\right)}{v_1}$。

至此，物理部分的分析已经结束。此题信息学的部分较为容易，只需使用推出的公式代入计算即可，单组测试数据时间复杂度$O(1)$。

1.2 Magic [2]

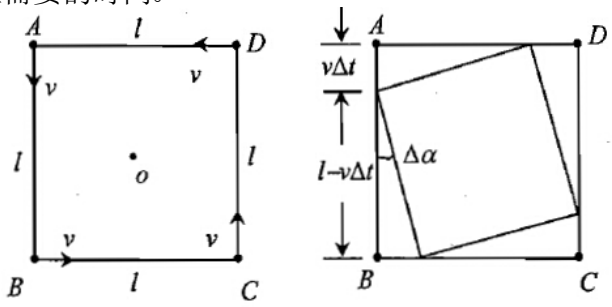
1.2.1 题目大意

有n个人初始位于一个边长为l的正n边形的n个顶点，以相同的不变速率v作追逐游戏，且每个人始终对准他顺时针方向下一个人运动。求追上需要的时间。

在分析之前，不妨先来看一个与此题相关的经典物理问题。

1.2.2 经典问题

如下左图所示，有4个人A、B、C、D初始位于一个边长为l的正方形的四个顶点，以相同的不变速率v作追逐游戏，且每个人始终对准他逆时针方向下一个人运动。求追上需要的时间。



[2]POJ Challenge Round 5 A

1.2.3 经典问题解法1

观察容易发现，4个人始终会处在某个正方形的4个顶点上，只是正方形在变小。最终他们将会相遇于正方形的中心。

这样就只需考虑一个人与中心连线方向的运动。显然这是一个匀速运动，分速度$v_r = v \cdot sin\frac{\pi}{4}$。而与中心的初始距离$d = \frac{l}{2sin\frac{\pi}{4}}$。因此，

$$t = \frac{d}{v_r} = \frac{l}{v}$$

1.2.4 经典问题解法2

另一种思路是只考虑相邻的两个人，他们之间的相对速度即为两个人连线方向分速度之差，即$\Delta v = v - v \cdot cos\frac{2\pi}{4} = v$。因此，

$$t = \frac{l}{\Delta v} = \frac{l}{v}$$

1.2.5 再分析

下面就再回到我们要探讨的例题。

容易看出，只要将上面2种解法角度中的4替换为n即可，从而得出以下两种解法。至于信息学的部分，也只需套用公式即可，时间复杂度$O(1)$。

1.2.6 解法1

$$v_r = v \cdot sin\frac{\pi}{n}, \tag{11}$$

$$d = \frac{l}{2sin\frac{\pi}{n}}, \tag{12}$$

$$t = \frac{d}{v_r} = \frac{l}{2v \cdot sin^2\left(\frac{\pi}{n}\right)} = \frac{l}{v\left(1 - cos\frac{2\pi}{n}\right)} \tag{13}$$

1.2.7 解法2

$$\Delta v = v - v \cdot cos\frac{2\pi}{n}, \tag{14}$$

$$t = \frac{l}{\Delta v} = \frac{l}{v\left(1 - cos\frac{2\pi}{n}\right)} \tag{15}$$

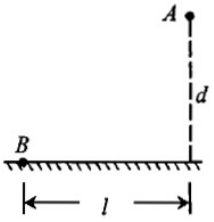
1.3 滑行[3]

1.3.1 题目大意

有一个小滑块要从$(0,0)$出发滑到(X,Y)。但这一块地面被划分为n个区域，分界线都与x轴平行，第i个区域限速v_i。求到达目标点最少需要用的时间。

同样，不妨也先来看一个物理中的经典问题。

1.3.2 经典问题



如左图所示，湖中小岛A与湖岸的距离为d，湖岸边有一点B，B湖岸方向与A点距离为l。一人自A点出发，要到达B点。已知他在水中游泳的速度为v_1，在岸上行走的速度为v_2（$v_1 < v_2$）。要求他由A到B所用的时间最短，问此人应当如何选择其运动路线。

1.3.3 经典问题分析

此题显然需要在岸上寻找一个转折点C，使得先从A游到C再从C走到B用时最短。设C湖岸方向与A的距离为x，则用时$T(x) = \frac{\sqrt{d^2 + x^2}}{v_1} + \frac{l - x}{v_2}$。最短用时应该是该函数在$[0, l]$上的最小值，因此求出此函数的极值点：

$$\frac{dT}{dx} = \frac{x}{v_1\sqrt{d^2 + x^2}} - \frac{1}{v_2} = 0 \tag{16}$$

解得：$x = \frac{v_1 \cdot d}{\sqrt{v_2^2 - v_1^2}}$，这是一个极小值点。所以此题答案就出来了：

[3]bzoj3695

①若$l \le \frac{v_1 \cdot d}{\sqrt{v_2^2 - v_1^2}}$，则只需直接从A游泳至B；

②若$l > \frac{v_1 \cdot d}{\sqrt{v_2^2 - v_1^2}}$，则需先游泳至湖岸方向距离A$\frac{v_1 \cdot d}{\sqrt{v_2^2 - v_1^2}}$处再走至B。

当然这是利用导数进行的分析，其实还可以利用一些变换使得结果变得更为简洁。设AC与垂直岸方向的夹角为θ，则(16) 可以改写为：$sin\theta = \frac{v_1}{v_2}$。利用$sin\frac{\pi}{2} = 1$ 可得：

$$\frac{sin\theta}{sin\frac{\pi}{2}} = \frac{v_1}{v_2} \tag{17}$$

注意到(17)与光的折射定律类似。事实上，人的游泳和行走就相当于光在两种介质中的运动，而之前的分析过程就类似于光传播的费马原理，因此这道题的结果也就相当于“折射定律”。

1.3.4 再分析

下面就回到要探讨的例题。

根据“折射定律”，为了用时最短，从第i个区域穿到第$i+1$个区域时，入射角和折射角应满足$\frac{sin\theta_i}{sin\theta_{i+1}} = \frac{v_i}{v_{i+1}}$。

至此此题只剩下信息编程的部分。容易看出，初始入射角越大，设出第n层介质时的横坐标x_n越大。因此只需二分初始入射角，找到一个合适的角度使得射出第n层介质时$x_n = X$即可。当然注意初始入射角不能过大，要使得对于任意的$1 \le i \le n$满足$sin\theta_i < 1$，不然在光学中光将会全反射或射至x 轴方向无穷远处。

时间复杂度$O(kn)$，其中k为二分次数。

2 信息学与动力学

不同于运动学，动力学的题目主要分析的是运动和力的关系。信息学竞赛中的动力学的题目一般运动物体数量较多，且常常出现碰撞的问题，看似比较复杂无从下手。下面不妨先来回顾一下碰撞所涉及的一些定律，从而找到解决问题的途径。

动量定理与动量守恒定律

质量为m的质点，在某惯性系的速度为$\boldsymbol{v}$，相对此惯性系的动量定义为

$$\boldsymbol{p} = m\boldsymbol{v} \tag{18}$$

质点系动量定理即（$\boldsymbol{F}$表示合外力，$\boldsymbol{p} = \sum \boldsymbol{p}_i$）：

$$\boldsymbol{F} = \frac{d\boldsymbol{p}}{dt} \tag{19}$$

于是便有如下的动量守恒定律：

Theorem 1. 若过程中$\boldsymbol{F}$恒为零，则过程中$\boldsymbol{p}$为守恒量。

动量守恒定律也有分量单独守恒的情况，即：

Theorem 2. 若过程中F_x恒为零，则过程中p_x为守恒量。

动能定理与动能守恒

相对某惯性系定义质点的动能为：

$$E_k = \frac{1}{2}mv^2 \tag{20}$$

这里仅写出不考虑内力做功的质点系动能定理。将所有外力做功之和记为W，质点系动能增量记为ΔE_k，则：

Theorem 3.

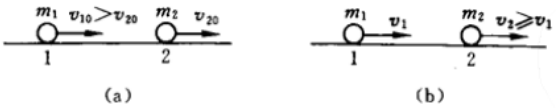
$$dW = dE_k,\ W = \Delta E_k \tag{21}$$

显然，若过程中dW恒为零，则过程中E_k为守恒量。

碰撞

现在就来考虑如何利用以上两个守恒分析两个物体的弹性碰撞，即满足动能守恒的碰撞。

一维弹性碰撞



质点1和2碰撞前后的状态分别在上图(a)(b)中示出。根据动量守恒和能量守恒可列出下列方程组：

$$m_1v_1 + m_2v_2 = m_1v_{10} + m_2v_{20}$$

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1v_{10}^2 + \frac{1}{2}m_2v_{20}^2$$

数学上可得两组解，但物理上其中一组对应碰前运动状态，另一组为碰后的状态，即：

$$v_1 = \frac{(m_1 - m_2)v_{10} + 2m_2v_{20}}{m_1 + m_2}, \tag{22}$$

$$v_2 = \frac{(m_2 - m_1)v_{20} + 2m_1v_{10}}{m_1 + m_2} \tag{23}$$

下面来看一下特殊情况：

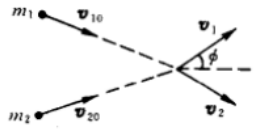
①若$m_1 = m_2$，则$v_1 = v_{20}$，$v_2 = v_{10}$；

②若$m_2 \gg m_1$，且$v_{20} = 0$，则$v_1 = -v_{10}$，$v_2 = 0$。

可以发现这些特殊情况十分简洁优美，下面的例题中就将有所应用。

二维弹性斜碰撞

二维斜碰撞只需将一维的动量守恒方程中的速度改为矢量即可。但是速度矢量$\boldsymbol{v_1}$和$\boldsymbol{v_2}$的解将具有不定性，原因就在于物体的质点化。



若能根据物体的几何形状确定其碰撞方向，则垂直碰撞方向的速度不变，碰撞方向上看作一个一维弹性碰撞即可。

有了以上方法，下面就来分析几道例题。

2.1 DZY Loves Physics I [4]

2.1.1 题目大意

有一条无限长的光滑水平轨道，初始有N（$N \le 10^5$）个相同的小球在其中运动，每个小球都有初始位置x、初始速率v和运动方向（正方向或反方向）。设小球的加速度为a，则任意时刻所有小球的运动都满足$a \cdot v = C$（C为给定常数），且小球间的碰撞均为弹性碰撞。共有Q（$Q \le 10^5$）次操作，分为两类：①再加入一个小球；②询问时刻T第K小的速率值。

2.1.2 分析

质量相同的小球一维弹性碰撞属于特殊情况①，只会交换速度，速率值不发生变化，而这道题询问的又恰恰是第K小的速率，因此碰撞不影响答案，可以忽略。这显然是一个对于问题的巨大简化，这其实就变成了一个运动学的问题。

对于一个小球，$a \cdot v = C$，即$\frac{dv}{dt} \cdot v = C$。由此可得：

$$\frac{d(v^2)}{dt} = 2C \tag{24}$$

解得$v_T^2 = v_0^2 + 2CT$，也就是说小球速率的平方都是均匀增加的，小球间的速率大小关系其实不会改变。所以要询问的其实是初始速率第K小的小球。这个用许多经典的数据结构都可以求出，这里就不详述。

时间复杂度$O((N+Q)log\ N)$。

2.2 Billiards [5]

2.2.1 题目大意

给出一个$W \times H$的矩形球台以及n（$n \le 22$）个半径为r、质量相等的台球的坐标，一开始它们都静止在球台上，并且不会有球和球、球和台边贴合的情况。现在给1号球一个初速度，问在第m（$m \le 50$）次碰撞发生时，所有球的坐标。

[4] bzoj3570

[5] POJ Challenge Round 5 C

碰撞包括球和球碰撞、球和任意一条台边碰撞。保证不会有两次碰撞同时发生，且球的质量远小于球台的质量。

2.2.2 分析

由于数据范围较小，不妨考虑暴力模拟。直接找出所有可能的碰撞发生的时间，选取最小的一个作为下一次碰撞即可。

剩下的问题就是碰撞时速度的改变。球与球台的碰撞属于特殊情况②。碰撞方向显然垂直于台边，于是垂直台边速度反向，平行台边速度不变即可。球与球的碰撞属于特殊情况①，碰撞方向为球心连线方向，因此碰撞方向交换速度，垂直碰撞方向速度不变。可以看出，物理的推导也极大地简化了此题的分析。

由于每次可能的碰撞数是n^2的，时间复杂度$O(n^2m)$。

3 信息学与静力学

静力学主要分析的是物体的平衡，一般分析方式较为套路，侧重于对物体受力、重心位置等的分析。下面不妨先来看一下物体的平衡条件。

物体平衡条件

这里我们探讨容易一些的情况，即共点力系作用下物体的平衡条件。这在中学物理教材中也已涉及，即物体所受合外力等于零。解题时往往考虑直角坐标系下的分量形式，即：

$$\sum F_x = 0,\ \sum F_y = 0,\ \sum F_z = 0 \tag{25}$$

一般物体的平衡还需合力矩为零，此处就不详述。

重心

重心一般即质心。质点系所受重力可一等效于作用于质心上的一个力。

设N个质点系统各质点的位置矢量为$\boldsymbol{r_1}, \boldsymbol{r_2}, ..., \boldsymbol{r_N}$，第$i$个质点质量为$m_i$，则定义该质点系的质心位置$\boldsymbol{r_C}$为：

$$\boldsymbol{r_C} = \frac{1}{\sum_{i=1}^{N} m_i} \sum_{i=1}^{N} m_i \boldsymbol{r_i} \tag{26}$$

这其实相当于以质量为权重的加权位置平均值。且质心的定义是一个矢量表达式，自然可以写出其分量式，这里就不列出。

值得注意的是，均匀物体的质心就是其几何中心。若是复杂物体的质心，可以考虑将其分解为比较易于求出质心的几部分然后再利用定义求得总体的质心。

下面就利用以上知识来分析两道例题。

3.1 书堆[6]

3.1.1 题目大意

N（$N \le 10^{18}$）本相同的长为M的书叠放在桌沿，求平衡时最上面的那一本书能伸出桌面的最长长度L（书的厚度不计，上方书的重心必须位于下方书或桌面边缘以内）。

3.1.2 分析

此题书受力比较简单，平衡主要涉及重心位置的分析。N本书的重心恰好落在桌面边缘显然是最优的（但根据题目要求答案可能需要减1）。把L 看作是一个关于N 的函数$L(N)$，$L(1) = \frac{M}{2}$。

假设已经求出了$L(N-1)$。不妨把第N本书放在最底层，设其伸出边缘l，则为了最优，上方$N-1$本书的重心应恰好落在第N 本书的右边缘，即$L(N) = L(N-1) + l$。为了使当前N本书重心位于桌面边缘，可列出以下方程：（式中m表示一本书的质量）

$$\frac{m \cdot \left(l - \frac{M}{2}\right) + (N-1)m \cdot l}{N \cdot m} = 0 \tag{27}$$

解得：$l = \frac{M}{2N}$。所以，

$$L(N) = \frac{M}{2} \sum_{i=1}^{N} \frac{1}{i} \tag{28}$$

至此此题物理的部分已经结束，剩下的就是设法编程求出$H(N) = \sum_{i=1}^{N} \frac{1}{i}$。对于较小的$N$ 可以暴力，时间复杂度为$O(N)$；对于较大的N，可以利用欧拉常数$\gamma \approx 0.5772156649$ 进行近似，即$H(N) \approx ln\ N + \gamma$，时间复杂度为$O(1)$。

[6]bzoj2048, 2009年集训队作业

3.2 Water Tanks [7]

3.2.1 题目大意

有N个底面积均为1的圆柱容器自左向右排成一列，高度已知，且除第一个容器开口外剩余的容器均密封。相邻两个容器间有体积可忽略的细管相连，且细管从左到右高度递增。已知外界气压为一个大气压，水面以下D处的水压为$0.097D$ 个大气压，封闭气体的气压与体积的乘积恒定。求1 号容器被注满时共注入的水量。

3.2.2 分析

不妨设第i个容器高H_i，其中水位为h_i，连接容器i和$i+1$的细管高度为a_i（由题意$a_i < a_{i+1}$），则：

可以发现，当$h_i < a_i$ 时第$i+1$ 个容器中一定没有水；当$0 < h_{i+1} < a_i$时$h_i = a_i$；当$h_{i+1} \geq a_i$（为了保证容器$i+2$中没有水，须满足$a_i < a_{i+1}$）时，第i 个容器中的气体就可以看作是密封的，与后面容器中的气体不再有关系，也可以不再考虑。

因此，可以考虑依次枚举i，看$h_{i+1} = a_i$时1号容器是否已满。判断方式可以利用压强的平衡，也就是之前所述的某一点处受力平衡：

若1号容器盛满时在高度a_i处的水压大于$h_{i+1} = a_i$时$i+1$号容器及之后容器中密封的气体的压强（这些气体压强设为p，体积设为V），即：

$$0.097(H_1 - a_i) + 1 > p$$

则1号容器还未满，可以枚举下一个i，p和V利用乘积恒定的条件相应修改即可；否则1号容器已满，需判断h_i是否能达到a_i，从而求出最终的h_i和h_{i+1}。具体判断和求解方法与上面类似，这里就不详述。

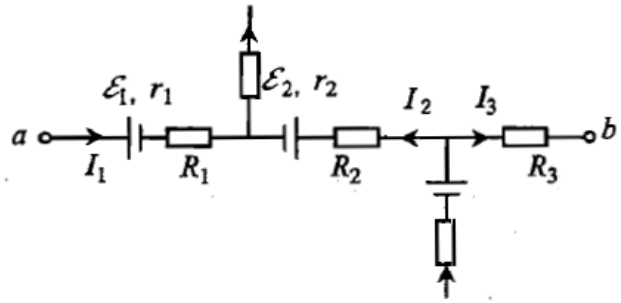
这样只需枚举一遍，时间复杂度$O(N)$。

[7] 2007 ACM-ICPC World Finals I

4 信息学与电磁学

电磁学中恒定电流的题目很适合利用编程解决，也已经数次出现在信息学的比赛中。当然信息学竞赛中的恒定电流题目电路更为复杂，数据也不太具有规律，不容易直接解决。下面也先回顾一下物理解题中常用的一些方法，从而寻找解决问题的途径。

数电压法



电路中任意两点间电势下降等于连接这两点的任一支路上各电路元件电势下降的和。例如如上图所示的电路中，取出连接a、b两点的一条支路。设a点电压为U_a，b点电压为U_b，则：

$$U_a - U_b = I_1(r_1 + R_1) + \mathscr{E}_1 - I_2(r_2 + R_2) - \mathscr{E}_2 + I_3R_3 \tag{29}$$

事实上这对于封闭回路也是正确的，也就是后面要提到的基尔霍夫回路电压方程。

基尔霍夫定律

基尔霍夫第一定律

Theorem 4. 汇合于任一节点处的各电流的代数和等于零，即

$$\sum I = \sum I_{in} + \sum I_{out} = 0 \tag{30}$$

以上称为基尔霍夫第一定律，又称节点电流方程。对于一个具有n个节点的多回路电路，便可写出$n-1$个独立的节点电流方程。

基尔霍夫第二定律

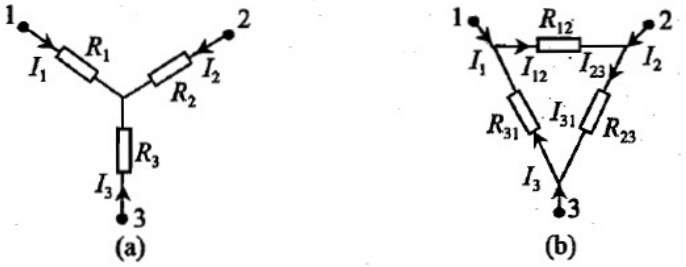
Theorem 5. 电路中的任一闭合回路的全部支路上的电压的代数和等于零，即

$$\sum U = \sum (\pm \mathscr{E} \pm Ir \pm IR) = 0 \tag{31}$$

以上称为基尔霍夫第二定律，又称回路电压方程。式中$\mathscr{E}$和I前面的正负号根据回路绕行方向决定。独立方程的个数等于电路的独立回路的个数。

$Y-\Delta$电路的等效代换

复杂电路中常常会遇到电阻连成Y形或Δ形，如下图所示。若能够将两种连接等效变换，就可能变换出简单的串并联，从而简化计算。



所谓“等效代换”，就应保持电路中其余部分在三个端点的电势U_1、U_2、U_3以及流经三端点的电流I_1、I_2、I_3全部相同。

图(a)中利用数电压法和节点电流方程可得：

$$\begin{aligned} I_1R_1 - I_2R_2 &= U_1 - U_2, \\ I_1R_1 - I_3R_3 &= U_1 - U_3, \\ I_1 + I_2 + I_3 &= 0 \end{aligned}$$

可解得：

$$I_1 = \frac{(R_2+R_3)U_1 - R_3U_2 - R_2U_3}{R_1R_2 + R_2R_3 + R_3R_1}$$

图(b)中：

$$I_{12} = \frac{1}{R_{12}}(U_1 - U_2),\ I_{31} = \frac{1}{R_{31}}(U_3 - U_1)$$

$$I_1 = I_{12} - I_{31} = \left(\frac{1}{R_{12}} + \frac{1}{R_{31}}\right)U_1 - \frac{U_2}{R_{12}} - \frac{U_3}{R_{31}}$$

为了使I_1相同，独立参量U_1、U_2、U_3的系数相等即可。因此有结论：

$$R_{12} = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_3}, \tag{32}$$

$$R_{31} = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_2} \tag{33}$$

用类似方法还可以解得：

$$R_{23} = \frac{R_1R_2 + R_2R_3 + R_3R_1}{R_1} \tag{34}$$

反变换也容易由此推出，这里就不详述。但是值得注意的是，可以把该变换的结果用电导（电阻的倒数）g表示：

$$g_{ij} = \frac{g_ig_j}{\sum_{k=1}^{3} g_k} \ (1 \le i, j \le 3,\ i \ne j)$$

这里就可以感受到物理学的优美之处。模仿上式的形式可以推广得到更易于使用的n阶的“$Y-\Delta$变换”，即：

$$g_{ij} = \frac{g_ig_j}{\sum_{k=1}^{n} g_k} \ (1 \le i, j \le n,\ i \ne j) \tag{35}$$

后文就将应用到这种变换。

有了以上方法，下面就来分析几道例题。

4.1 电阻[8]

4.1.1 题目大意

一块电路板有N（$N \le 100$）个接点和M个两端都接在接点上的电阻（电阻为非负整数），求接点1和接点N间的等效电阻。

4.1.2 分析

求等效电阻可以使用注流法，即在1号点通入1A电流，在N号点流出，这样即可将问题转化为求U_{1N}。令$U_N = 0$，则$U_{1N} = U_1$。

[8]poj3532

首先应合并阻值为0的电阻连接的接点，直接并联的电阻也可合并。然后考虑列出方程。鉴于列回路电压方程较为复杂，还是考虑结合数电压法和节点电流方程。为了表示简便不妨使用电导。

设连接i、j接点的电阻上流过的电流为I_{ij}，则根据数电压法，有$I_{ij} = (U_i - U_j)g_{ij}$。考虑接点$i$的节点电流方程可得：

$$\sum_{j\neq i} U_j g_{ij} - U_i \sum_{j\neq i} g_{ij} = -I_i \ (1 \le i < N) \tag{36}$$

其中$I_1 = 1A$，对于其它的i，$I_i = 0$。这样共有$N-1$个未知数和$N-1$个方程，用高斯消元即可解得U_1。时间复杂度$O(M + N^3)$。

4.2 电阻网络[9]

4.2.1 题目大意

在一个有N（$N \le 50000$）个节点的树中，每条边上都有一个阻值为$R = 10000\Omega$的电阻，所有叶子节点都通过一个阻值同样为R的电阻接地。已知树的最长链长度不超过50。共有M（$M \le 50000$）次操作，分为两种：

①在u和v的连边上串联一个大小为w伏且负极指向u的电源；

②询问点u当前的电压（设接地处电压为0）。

4.2.2 分析

此题最主要的区别是加入了电源，但依旧可以用类似的方法列式分析。

首先由于叶子节点接地电势已知，不妨将根换为任意一个叶子节点，设为$root$。设此时点i的孩子集合为son_i，父亲为fa_i，度数为deg_i，电势为U_i，连接fa_i与i的边上电源电势为W_i，负极指向i，由fa_i流向i的电流为I_i。不妨先考虑$son_i \neq \varnothing$的情况，边界情况放到最后考虑。由数电压法和节点电流方程：

$$\begin{aligned} I_i R + W_i &= U_{fa_i} - U_i, \\ I_x R + W_x &= U_i - U_x (x \in son_i), \\ \sum_{x\in son_i} I_x &= I_i \end{aligned}$$

[9] bzoj2805, CTSC2012 day1

消去I解得：

$$U_i = \frac{1}{deg_i}\left(U_{fa_i} - W_i + \sum_{x \in son_i}(U_x + W_x)\right) \tag{37}$$

可以注意到U_i同时与父亲的U_{fa_i}和孩子的U_x相关，询问的时候不容易直接求出。于是转而考虑能否推出一个U_{fa_i}到U_i的递推关系式，因为那样询问的时候就可以直接从根节点$root$开始暴力递推了。容易看出U_i和U_{fa_i}应是线性的关系，设$U_i = K_i U_{fa_i} + B_i$，其中K_i和B_i是只与i和i的孩子相关的可预先求出的系数。显然$U_x = K_x U_i + B_x$（$x \in son_i$），那么代入(37) 可得：

$$U_i = \frac{1}{deg_i}\left(U_{fa_i} - W_i + \sum_{x \in son_i}(K_x U_i + B_x + W_x)\right)$$

解得：

$$U_i = \frac{1}{deg_i - \sum_{x \in son_i} K_x} U_{fa_i} + \frac{\sum_{x \in son_i}(B_x + W_x) - W_i}{deg_i - \sum_{x \in son_i} K_x}$$

$$\Rightarrow\ K_i = \frac{1}{deg_i - \sum_{x \in son_i} K_x},\ B_i = \frac{\sum_{x \in son_i}(B_x + W_x) - W_i}{deg_i - \sum_{x \in son_i} K_x} \tag{38}$$

最后还有边界的问题，即对于以$root$为根的树的叶子节点i，其实i的下方相当于接了一个电势为0的节点，即$deg_i = 2$，$\sum_{x \in son_i}(U_x + W_x) = 0$，所以有：

$$U_i = \frac{1}{2}U_{fa_i} - \frac{W_i}{2}$$

$$\Rightarrow\ K_i = \frac{1}{2},\ B_i = -\frac{W_i}{2} \tag{39}$$

这样所有K和B都可以事先递推出来，询问的时候可以每次$O(L)$（L为树的最长链长度）递推U_i，而修改W_i的时候只需在修改点u到根的路径上暴力重新递推K和B即可，单次递推也可以做到$O(L)$。

总时间复杂度$O(N + QL)$。

4.3 插线板[10]

4.3.1 题目大意

有一棵N（$N \leq 10000$）个节点且每个节点都是一个插线板的树，每个插线板由三条可忽略电阻的导线（或小节点）组成，树的每条边就是父亲的三条导线与孩子的三条导线两两连电阻组成的3×3的电阻网络。共有Q（$Q \leq 10000$）次操作，分为两种：

①修改某个电阻的阻值；

②询问某两条导线之间的等效电阻。

其中阻值均以电导g（$0 < g \leq 10$）的形式给出。

4.3.2 解法1

可以解方程暴力求解等效电阻，方法同例4.1。时间复杂度$O(Q \cdot 27N^3)$。

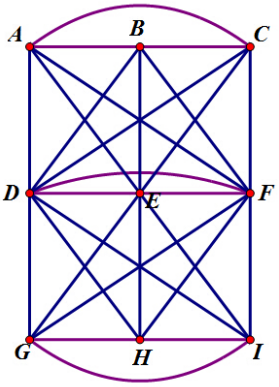
4.3.3 解法2

解法1的问题在于电路图点数过多。现考虑能否通过n阶“$Y-\Delta$变换”使得询问的两个插线板直接相连。

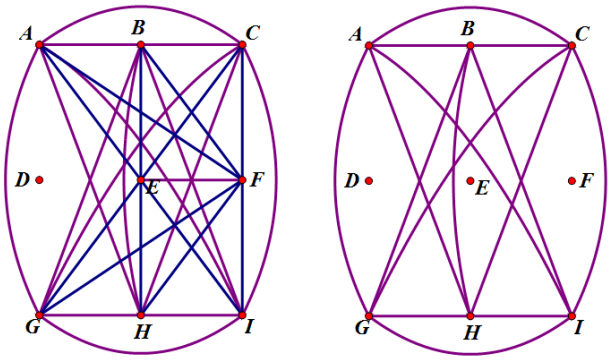
定义与导线x相连的导线集合为$S(x)$，$Transform(x)$表示以x为中心进行一次“$Y-\Delta$变换”，使得x不与任何导线相连且$S(x)$中的导线两两连电阻构成一张完全图。具体变换方式见(35)。根据电导的性质，两个不连通的点可以看作接有电导为0的电阻，两点间并联多个电阻只需将电导相加。利用这两点，这个“$Y-\Delta$变换”可以很容易地用程序实现。

为了方便，先只考虑树是一条链并且询问的两个插线板在这条链的两端的情况。

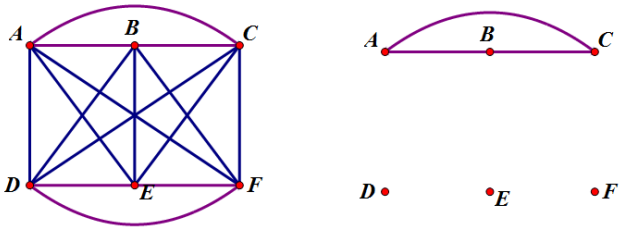
[10]bzoj3556, CTSC2014 day1



如右图所示为一棵只有三个插线板的树。点A、B、C、D、E、F、G、H、I均为插线板的导线，蓝色的连边是本来的电阻，而紫色的连边是新增的电阻（可以看作初始电导为0的电阻）。当$Transform(D)$时，容易发现电路会变为下左图所示（当然连边电导会有增加）。此时再$Tranform(E)$和$Transform(F)$，就会发现会变为下右图所示，也正是理想的效果——A、B、C三点直接与G、H、I三点相连。所以对于一条插线板数大于3 的链，也只需多次进行上述变换过程即可使得询问的两个插线板直接相连。此时只要解一个5×5的方程组即可得到等效电阻。



于是现在问题变为如何将一棵树变为一条询问的两个插线板位于两端的链。其实这与上面的过程也是类似的。如下左图所示，从叶子节点（设导线为D、E、F）开始依次$Transform(D)$、$Transform(E)$、$Transform(F)$，就会发现叶子节点类似于“收缩”到了父亲节点上，结果如下右图所示。因此只要把询问的链连出的子树全部“收缩”到询问的链上即可。



由于$Transform$过程有一定的常数k（这里要用到最高为8阶的“$Y-\Delta$变换”，而n阶的时间复杂度为$O(n^2)$），整个算法的时间复杂度为$O(Q\cdot(kN+5^3))$。

4.3.4 解法3

其实通过解法2，已经可以发现这个问题就是一个修改边、维护子树信息和链信息的题目。不妨考虑使用$Link-Cut\ Tree$来进行优化。

为了方便，可以在$Link-Cut\ Tree$中加入$N-1$个表示边的点。表示边的点维护3×3的电阻边，表示插线板的点维护3条导线之间连接的3 个电阻即可。另外对于每个点还需要维护虚边连的点“收缩”后产生的等效电阻，这样就可以维护子树信息了。询问的时候与其它题目$Link-Cut\ Tree$的链询问也没有太大的区别，只是链上的信息较为复杂，合并信息代码较长。最后依旧用高斯消元解一下方程组即可。

时间复杂度$O(k(N+Q)log\ N+5^3Q)$，可以通过全部测试数据。

5 总结

由以上几部分的例题可以看出，信息学中的物理问题看上去确实较为复杂，但是经过一些物理（可能用到一定的高等数学）推导就可以使问题直接用一个公式解决或者转化为信息学中的经典问题，从而使原问题得以快速地解决。这既体现了物理学的优美，也体现了信息学的准确高效。

这样看来，学科与学科之间的界限其实并没有那么分明，不同学科的结合也许会使更多新思路、新方法产生。这或许将给我们此后的学习带来一些启示。

参考文献

[1] 刘汝佳，《算法竞赛入门经典》，清华大学出版社。

[2] 程稼夫，《中学奥林匹克竞赛物理教程·力学篇》，中国科学技术大学出版社。

[3] 舒幼生，《力学（物理类）》，北京大学出版社。

[4] 程稼夫，《中学奥林匹克竞赛物理教程·电磁学篇》，中国科学技术大学出版社。

例题网址

[1] 例1.1，http://acm.hdu.edu.cn/showproblem.php?pid=4969

[2] 例1.2，http://poj.openjudge.cn/challenge5/A/

[3] 例1.3，http://www.lydsy.com/JudgeOnline/problem.php?id=3695

[4] 例2.1，http://www.lydsy.com/JudgeOnline/problem.php?id=3570

[5] 例2.2，http://poj.openjudge.cn/challenge5/C/

[6] 例3.1，http://www.lydsy.com/JudgeOnline/problem.php?id=2048

[7] 例3.2，http://icpc.baylor.edu/download/worldfinals/problems/2007WorldFinalProblemSet.pdf

[8] 例4.1，http://poj.org/problem?id=3532

[9] 例4.2，http://www.lydsy.com/JudgeOnline/problem.php?id=2805

[10] 例4.3，http://www.lydsy.com/JudgeOnline/problem.php?id=3556

丢失的题面 试题讨论

大连市第二十四中学 于纪平

摘 要

2015年集训队互测第五场中，一道非传统题的命题思路和题目分析。

1 题目大意

传统题，不给出题面，只给出10组输入输出数据。

时间限制1秒，内存限制128M，没有Special Judge。

2 总体分析

2.1 命题契机

近年来，信息学竞赛的题目类型变得多种多样，被选手们称为“传统题十合一”的逐测试点分析的非传统题也连续在两年的冬令营中出现（WC2014非确定机，和WC2015未来程序），引发了大家很大的关注。

分析这两道题目的本质，“非确定机”是给出程序和输出，求任意一个可能的输入，“未来程序”是给出程序和输入，求它的输出。两道题也有各自的难点，例如“非确定机”的程序是通过黑箱给出的，需要选手手动分析；“未来程序”的输入会使程序运行的非常慢，需要选手做很大的分析优化。

注意我们的三个要素：输入、程序和输出，给出任意两个都可以让选手求出剩下的一个。“非确定机”求的是输入，“未来程序”求的是输出，这道“丢失的题面”就是给出输入和输出求程序。

2.2 得分估计

这道题目虽然以传统题的形式评测，其本质却是需要对每组输入输出数据分别分析，应该称之为非传统题。

在这道题上花费的时间越多，得到的分数就会越多，平均每10分钟应该就能得到10分。

2.3 集训队互测得分情况

1人90分，3人70分，2人60分，2人50分，1人20分，2人10分，略低于预期情况。

2.4 UOJ镜像得分情况

互测结束以后，在Universal Online Judge上进行了镜像比赛，向所有人开放，得分情况如下：2人90分，4人80分，3人70分，2人60分，1人50分，1人40分，4人30分，1人20分，5人10分。

2.5 得分总结

对于实际的情况，平均每15～30分钟应该就能得到10分。这也能看出出题人对于分数的估计很可能高于实际情况。

3 题目解法及命题思路

3.1 测试点1,2,3

这三个测试点的共同点是，输入文件只有一个整数，输出文件是一个二进制或三进制串。

这些测试点的主要命题思路是考察选手的找规律能力和找到正确的分析方法（例如，观察输出串的长度和含有各数字的比例等）。

3.1.1 测试点1

输入是22，输出是长度为2^{22}的二进制串，其中0和1各占一半。

从头开始读容易发现明显的规律，从而得出输出串的构造方式：初始串是”0”，每次把串取反接在尾部。

3.1.2 测试点2

输入是33，输出是长度为3524578的二进制串，其中0和1的比例大约是黄金分割比。

同时注意到3524578是第33个斐波那契数，可知这是斐波那契字符串：初始串是”0”，每次扫一遍串，把”0”变成”1”，把”1”变成”01”。

3.1.3 测试点3

输入是12，输出是长度为531452的三进制串，三种数出现的次数大致相同，但是0稍微多一点。

注意到$531452 = 3^{12} + 11$，经观察发现输出串包含了所有的长度为12的三进制串作为子串且是字典序最小的。

按照这个规则暴搜+剪枝就可以通过了。

3.2 测试点4,5,6

这三个测试点的共同点是，输入文件和输出文件的格式相同，都是第一行一个数n，接下来$n + 1$行每行一个数。其中测试点4和5的输入输出文件还具有对称性。

这些测试点的主要命题思路是考察选手分析多项式的能力和对多项式的计算。

大部分的数看起来很随机而且分布在某个数以内，很可能是模某个数的运算。容易得出模数是104857601。

而$104857601 = 25 \times 2^{22} + 1$，这几个测试点很可能与FFT有关。

3.2.1 测试点4

给出一个多项式，要给它平方。

我们可以套用经典的多项式乘法计算。

3.2.2 测试点5

给出一个多项式，要给它开方。

我们可以套用经典的多项式开方计算。

3.2.3 测试点6

给出一个多项式，要给它开立方。

由于模数104857601的特殊性质，这个是可以直接开三次方的，但是如果不会FFT怎么办呢？

……

注意到测试点4的输出实际上是$(x+1)^{262144}$，直接算组合数就可以了。

测试点5的输出实际上是$(x-1)^{131072}$，也是算组合数。

测试点6的输入和输出看起来非常随机，但是我们可以猜测输出文件是$(ax+b)^{177147}$。

根据第一项和最后一项，可列方程

$$a^{177147} \equiv 22131490, b^{177147} \equiv 43392819 \pmod{104857601}$$

暴力解得

$$a = 23333333, b = 33333333$$

这个答案一看就很靠谱，代入验证成立。

这样就可以通过测试点4,5,6了。

3.3 测试点7,8,9

这三个测试点的共同点是，输入输出格式非常像图上的数据结构题，其中测试点7没有边权，测试点8是一棵树。

这些测试点的主要命题思路是考察选手通过数据猜测题面的能力以及简单数据结构的应用。

3.3.1 测试点7

点数和边数都是10^5，没有边权，看起来非常随机。每个询问的答案只可能是0或$+\infty$。

观察发现，如果两个点连通，答案就是0，否则答案是+∞。用并查集或Floodfill就可以做了。

3.3.2 测试点8

是10^5个点的随机生成的树，边上带权，权也是随机生成的。每个询问的答案看起来都比较大。

观察发现，答案就是两点间树上路径边权最大值。由于数据随机所以可以不用高级数据结构维护。

3.3.3 测试点9

格式与测试点8相同，但不是树而是一般的图。答案看起来也没有那么大，而且出现了不连通的情况。

询问的依然是路径问题，但是不连通时答案是+∞，可知求的是某个量的最小值。结合测试点8分析可知求的是两点间路径边权最大值的最小值。

将所有边从小到大排序后依次加入按秩合并不路径压缩的并查集，对于每次询问二分答案在并查集中查询即可。

（当然，也可以用一些小的技巧去掉二分，不过这道题不卡时间，所以怎么做都可以）

3.4 测试点10

测试点10要求程序能输出自己。这个任务在完成前9个测试点的基础上是很难的。

但是由于题目的评测方式是传统评测方式，所以直接无视输入输出文件里面奇怪的英文，把输出文件原样打出来就可以了。

4 总结

在互测当中出了这样一道题，是想引发大家对于非传统题命题方向的思考，希望以后会有更多更新颖的非传统题涌现出来。

5 致谢

感谢中国计算机学会提供的交流平台。

感谢国家集训队教练余林韵和陈许旻的热情帮助。

感谢我的教练张新钢老师多年来的培养。

感谢参加互测的集训队员和参加镜像的全国各地的OIer。

DP 的一些优化技巧

绍兴市第一中学 张恒捷

摘 要

动态规划是信息学竞赛中十分常见的题目类型。本文列举了若干针对 DP 的优化方法，在一些题目上大显身手，获得了不错的效果。

1 引言

DP (Dynamic Programming)，即动态规划。是一种通过把原问题分解为相对简单的子问题来求解复杂问题的方法。近几年来，DP 一直在信息学竞赛中，尤其是各大竞赛网站上频繁出现，而且出现形式复杂，种类繁多。

一些诸如单调队列，斜率优化，以及四边形不等式的传统方法在前几年的论文中已经有详细介绍，本文将不再一一赘述。作者研究整合出了一系列 DP 优化技巧，并给出了一些例题帮助读者深入理解，希望对广大参加信息学竞赛的同学带来或多或少的帮助。

注：本文着重介绍优化技巧，故例题中的转移方程是如何得到的将不作为重点介绍。

2 方法整合

2.1 矩阵乘法加速

相信用矩阵乘法来优化 *DP* 转移的做法大家都已经熟知，故在此只是简单介绍一下。假如把 $f(i)$ 看成一个向量，而 $f(i+1)$ 可以用 $f(i)$ 乘上一个转移系数矩阵 A 得到，那么所求的 $f(n)$ 就可以写成 $f(0)\times A^n$ 的形式。由于矩阵乘法满足结合律，因此对于后者 A^n 只需要用快速幂优化即可。

因此对于一类转移单一并且转移次数又非常多的时候，可以考虑使用矩阵乘法来优化该DP。

2.2 位运算加速

对于一些 *DP* 其键值只会是0 或者1，这时就可以考虑使用位运算来打包处理信息了。0/1 背包就是一个很好的例子。用了位运算优化后程序速度将会极大地提升，可以过掉一些数据规模较大的数据了。

2.3 优化状态的数量

例1. *(codeforces Round 265,Div 1,D)*

现在有一个人玩一款游戏，游戏中有 k 种装备。每种装备初始的时候都是等级1，现在开始打怪，每打一个怪系统就会随机掉出一件装备，随机方式如下：先等概率随机装备的种类，假设该种类的装备当前的等级为 t，则系统会在 $[1 \sim t+1]$ 中等概率随机出该装备的等级。爆出装备后，如果该装备比玩家当前穿戴的等级高，则玩家会卖掉原装备，换上新装备，否则玩家会直接卖掉生成出的装备。等级为 i 的装备价值 i 金币。求打了 n 只怪后获得金币的期望。

数据范围： $n \le 10^5, k \le 100$

由于是求期望，而期望有线性性。所以可以针对一种装备算出它的期望，然后答案乘 k 就能得到最终的答案。

设 $f(i,j)$ 为当前已经有等级为 j 的装备了，再打 i 只怪期望获得多少金币。可以得到DP 方程：

$$f(0,j) = 0$$

$$\begin{aligned} f(i,j) &= \frac{1}{k}\sum_{x=1}^{j+1}\frac{1}{j+1}(f(i-1,max(j,x)) + min(j,x)) + \frac{k-1}{k}f(i-1,j) \\ &= \frac{1}{k}\left(\frac{j}{j+1}\cdot(f(i-1,j) + \frac{j+1}{2}) + \frac{1}{j+1}\cdot(f(i-1,j+1)+j)\right) + \frac{k-1}{k}\cdot f(i-1,j) \end{aligned}$$

这个DP 方程的转移已经做到 $O(1)$ 了，因此我们可以使用标题所写的方法，尝试着删去那些非常小的概率才发生的状态？在本题中，如果已经拥有了等级为 p 的装备，下一次得到等级为 $p+1$ 的概率为 $\frac{1}{kp}$。因此，如果最终得到了等

级为 T 的装备，那么期望打败的怪物数为 $2k+3k+\cdots+Tk\approx\frac{kT^2}{2}$，可以得出打败了 n 只怪物装备能升级到 $\sqrt{\frac{2n}{k}}$ 左右。根据这个思路，我们在 $\sqrt{2n}$ 左右寻找那条界线，最后在700 左右发现当这个界线增大的话，答案只会有非常细微的变化。设界线为B，事实上 $B\sim O(\sqrt{n})$。因此我们最终得到了一个 $O(n\sqrt{n})$ 的做法。

本题使用了优化状态的方法，在保证了答案精度的情况下，直接优化了复杂度，不失为一种很好的方法。

2.4 找规律

找规律是OI解题中的一大特色，但是似乎并不能直接和严谨的DP算法联系起来，下文将会以一道非常具有代表性的趣题为例，分析找规律在一类分层DP中的应用。

例2. *(topcoder SRM 526 Div 1 - Level 3)*

一个包含 n 个串的集合，一个模式串 S。每次从集合中随机选一个串（选完后仍在集合中），选 K 次后全部拼起来变成 T，求 T 中 S 出现的期望次数。

数据范围：$n\le 50, |S|\le 500, K\le 10^{12}$

我们来分析一下这道题吧。首先由于题目要求期望，故设 $f(i,j)$ 表示：若当前已经匹配到了串 S 的 j 位置，再随机选 i 个串后 S 在之后期望出现了几次。很容易列出 DP 方程：

$$f(i,j)=\frac{1}{n}\sum_{x=1}^{n}(f(i-1,trans(x,j))+val(x,j))$$

其中$trans(i,j)$ 表示从 S 的第 j 位开始匹配，在加入了第 i 个串后匹配到了哪里，$val(i,j)$ 表示对应的匹配次数。最后$f(K,0)$ 就是要求的值。

相信不少同学看见这个转移方程就会想用矩阵乘法来优化。虽然这很正确，但是由于复杂度较高（为 $O(|S|^3 logK)$），是无法通过给定的时限的。

如果打出一张 f 函数的表，就能够发现在 i 较大的时候（事实上是 $i>|S|$ 时），对于任意的 j，有 $f(i,j)-f(i-1,j)$ 是一个定值！这里可以对这个规律给出简要的说明：

假设当前已经选了 x 个串了，考虑选出第 $x+1$ 个串后所贡献的答案，换种说法就是以第 $x+1$ 个串中的某个位置作为结束位置的匹配个数。假设 S 的长度为 L，那么除了第 $x+1$ 个串自身所拥有的匹配个数以外，还与它前面 $L-1$ 个字符有关。而当 x 的值远大于 L 的时候，前 $L-1$ 个字符都是随机出来的了。也就是说，当 x 大于 L 时，第 x 个串与第 $x+1$ 个串所带来的贡献的计算方式一模一样。所以会出现上述的神奇性质。使用这个算法后只需要做出 $i \le |S|$ 的解即可。复杂度为 $O(|S|^2 n)$，足以通过此题。

2.5 分离状态中的变量

在 *DP* 状态中，可能会有一些在转移中互不影响的变量，这时就可以将它们分别进行运算来达到一起运算时的效果。下面看一道例题深入了解一下。

例3. *(topcoder SRM 639 Div 1 - Level 2)*

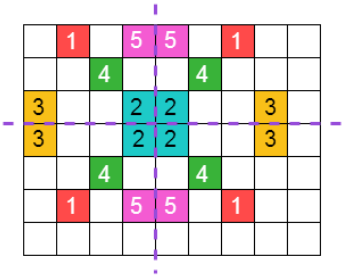
一张纸被等分成 n 行 m 列个小格子。每个格子是黑或者白色。整张纸可以用一个01 矩阵来表示。每次操作可以找一条竖直或者水平的不跨越小格子的线，然后将纸折叠一下。折叠时要保证贴合的两面每个格子的颜色都相同，而且不能把大的一面覆盖在小的上面。在任意次操作下最后有多少种结局。结局不同当且仅当剩下的纸在原矩阵中的位置不同。

$n, m \le 250$

一个很暴力的想法就是：设 $f(u,d,l,r)$ 表示原来的纸是否能折到上到 u，下到 d，左到 l，右到 r 的子矩阵。转移时枚举接下来的操作使用了哪条对称轴。最终答案就是所有状态中为 *true* 的个数。

但是这个方法复杂度较高，需要优化。事实上对于这道题来说横着折叠与竖着折叠是是可以分开做的。也就是说 (u,d) 与 (l,r) 其实是互不影响的。让我们分析一下是怎么得到这个性质的。

考虑任意一张带颜色的纸，如果我们连续使用了一次横向折叠与一次纵向折叠，如图：

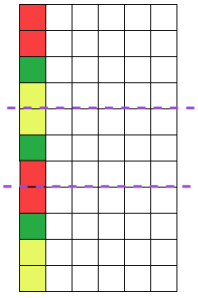


同样的颜色就代表了同样的数字，可以发现先横向后纵向与先纵向后横向是等价的。

考虑任意一个操作序列，一定是由一些横向，一些纵向的操作组成的，比如说 $VHVVHV$ 。由于相邻两个 HV 交换后是等价的，所以可以通过一系列交换使得所有的 H 都排在 V 前面，或者把所有的 V 都排在 H 的前面。如：$VHVVHV \to HVVHVV \to ... \to HHVVVV$ 。因此 $VHVVHV$ 合法等价于 HH 与 $VVVV$ 合法。所以我们成功的将横着折与竖着折分开了，最后的答案就相当于横着折的方案乘上竖着折的方案。

这样就可以很轻松的设计出状态了，考虑横着折的情况（竖着只要旋转90度），用 $f(u,d)$ 表示原来的纸是否能折出上到 u ，下到 d 的子矩形，转移时枚举折了那条线，这个用回文串的方法预处理一下就可以做到 $O(n^3)$ 了，足以过掉此题。

但是优化并没有结束，事实上上述算法中 $f(u,d)$ 中的 u 和 d 也是可以分开做的。与之前的想法类似，考虑相邻两次往上折与往下折。如果翻过去的纸没有碰到折痕，那么这两次操作顺序交换一下是等价的。如果碰到了折痕，由于折痕两侧的颜色是相同的，所以超过折痕的部分可以预先折进去，这样就到了没有碰到折痕的情况。因此我们总可以把向上折的操作与先下折的操作分离。



在只考虑向下折的情况下（向上折的话只要上下翻转），我们只要用 $f(u)$ 表示是否原来的纸是否能折到上到 u 的情况。这样复杂度就成功降到了 $O(n^2)$。

在上面的例子中，我们发现在变量分离过后，原先的4维状态被逐个击破，最后变成了4个无关的1 维状态，程序的复杂度大大降低。在特定的题目中使用往往能获得不错的效果。

2.6 将键值函数化

将键值函数化，顾名思义，就是把 $(x, f(x))$ 绘制在二维平面上，然后用一些函数来拟合它们，这样一个DP 数组就可以使用一些分段函数来表示了。这种方法可以做定义在实数域上的动态规划问题，或者定义域非常大的情况。而且将其绘制成函数后也会方便研究。

例4. *(topcoder SRM 610 Div 1 - Level 3)*

一个 $n \times m$ 的矿区，一共有 K 秒，每秒钟在 (a_i, b_i) 处会出现一个矿。假设你在 (x, y) 处，会获得 $n+m-|x-a_i|-|y-b_i|$ 的钱数。第 i 秒钟你横坐标不能移动超过 dx_i，纵坐标移动不能超过 dy_i，一开始你可以在任意位置，求最大钱数。

$K \le 1000$

$n, m \le 10^6$

我们发现 x 与 y 在计算最优值时不影响，所以可以用本文“分离状态中的变量”的方法将 x 与 y 分开考虑。以横坐标为例，设 $f(i, j)$ 为第 i 秒时，人在 j 处的与横坐标有关的最大收益。很显然有转移：

$$f(i, j) = \left(\max_{|j-k| \le dx_i} f(i-1, k) \right) + n - |j - a_i|$$

不过第二维状态的定义域过于大，所以不妨使用本节提到的方法，将 $f(i)$ 看成一个函数。我们可以发现 $f(i)$ 的导数[1]是单调不增的。这个可以使用归纳法证明。根据这个性质，我们可以得出如下做法：设 $f(i-1)$ 的最高点在 $x = u$ 处，那么：

[1]不考虑断处无导数的情况

$$f(i,j) = f(i-1, j+dx_i) + n - |j - a_i| \qquad (j < u - dx_i)$$
$$f(i,j) = f(i-1, u) + n - |j - a_i| \qquad (u - dx_i \le j \le u + dx_i)$$
$$f(i,j) = f(i-1, j-dx_i) + n - |j - a_i| \qquad (j > u + dx_i)$$

我们可以发现 $f(i-1)$ 到 $f(i)$ 的过程只是将函数在最高点分割开，平移一下，再整体加上一个 $n + |j - a_i|$ 的函数。所以我们可以直接维护构成函数的线段即可。

复杂度也优化到了 $O(K^2)$。

例5. *(codeforces Round 172,Div 1,E)*

给出一个序列 $x[1..n]$ 与三个数 $q, a, b (a \le b, a \times (n-1) < q)$，满足 $1 \le x_1 \le x_2 \le \cdots \le x_n \le q)$。现在要求一个序列 $y[1..n]$，满足 $1 \le y_i \le q; a \le y_{i+1} - y_i \le b$。你需要做的是最小化 $\sum_{i=1}^{n}(y_i - x_i)^2$

数据范围：

$n \le 6000$

$1 \le q, a, b \le 10^9$

我们先列出*DP*方程：$f(i,j)$ 表示确定了 $y[1..i]$，并且 $y_i = j$ 时，$\sum_{k=1}^{i}(y_k - x_k)^2$ 最小是多少。有转移：

$$f(i,j) = \left(\min_{k=j-b}^{j-a} f(i-1,k)\right) + (j - x_i)^2$$

只不过在这道题中所求的 y 可以是实数，这让普通的*DP*变得很难求解。因此我们仍然使用与上一题类似的做法，将 $f(i)$ 用分段函数来表示。经过分析我们仍然可以得出其导数单调的性质（分析过程与上一题类似），因此我们可以使用与上一题同样的方法来维护这些分段函数，只不过在此题中 $f(i)$ 是由一些二次函数组成的。

复杂度：$O(n^2)$

由于二次函数太复杂了，所以我们可以直接维护它们的导数！因为这两者的本质是相同的。这样就回到了维护一次函数时的情况。我们又发现所有的操作都是可以用平衡树来实现的，所以我们成功把复杂度降为 $O(n \log n)$。

2.6.1 小结

其实包括斜率优化在内的一些其他方法都可以属于本节提到的“将键值函数化”的方法。该方法往往能解决定义域非常大或者无穷的情况，而且具有非常强的拓展性。在例 5上使用了该方法后成功使用了数据结构优化了复杂度。

2.7 启发式合并DP 数组

现在考虑如下问题：有 m 个操作， n 个变量。每一个操作只能作用在某一个变量上，并且会给这个变量带来一些变化。求使用不超过 k 次操作的前提下最大化所有变量的和。假设我们已经求出了 $f(i,j)$ 表示如果在变量 i 上使用了 j 个操作，最大能到多少。我们可以两两合并这些*DP*数组。比如说在合并 a 变量与 b 变量后，可以得出在 a,b 上使用了 j 个操作， $a+b$ 最大能到多少。如果 a 变量用了 x 次操作， b 变量用了 y 次操作，那么合并后就用了 $x+y$ 次操作。所以很显然，合并一个长为 x 的数组与一个长为 y 的数组将会得到长为 $x+y$ 的数组。合并完所有的数组就可以得到最终的答案了。但是我们应该以什么顺序合并它们呢？

我们可以把这个问题扩展到一类*DP*问题上：将一个长为 x 的数组看做一个权重为 x 的点，所有点权重之和为 m 。合并权重为 x 与 y 的点需要花费 $O(T(x,y))$ 的复杂度，得到一个权重为 $x+y$ 的点。

我们不妨设第 i 个点的权重为 s_i ，接下来根据 T 来考虑其复杂度。

若 $T(x,y)=min(x,y)$

这个情况很简单，只要按顺序依次合并既可。

考虑复杂度： $\sum_{i=2}^{n} O(min(x,s_i)) \le \sum_{i=2}^{n} O(s_i) = O(m)$

所以该情况的复杂度为： $O(m)$

若 $T(x,y)=x\times y$

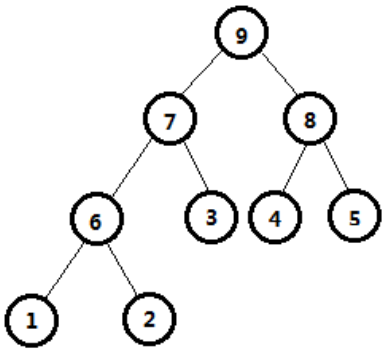
这种情况多数在树上问题出现。其实无论你按什么顺序合并，总复杂度都是 $O(m^2)$ 而非 $O(m^3)$ 的。关于这一点，我们可以把合成的点向合成它的点连边，整个结构就是一棵树， $T(x,y)$ 相当与枚举左右子树内所有点的匹配方案。这是你只需要注意到每一对点只会在它们 *LCA* 处枚举到就行了。

所以该情况的复杂度为： $O(m^2)$

若 $T(x,y)=x+y$

这个时候我们应该每次选取两个权值最小的点合并。

现在来考虑其复杂度。我们可以把每一个合成出来的点向合成它的点连边。这整个结构事实上就是一棵哈夫曼树。



考虑上图，其意义为：第一次选了权值最小的两条点1,2，合成了点6，第二次将点6,3 合成了点7，第三次将点4,5合成了点8，最后将点7,8合成了最终的点9。

由于 $T(x,y)=x+y$ ，因此整个算法的复杂度等于所有点的权值之和。

$$complexity=\sum O(s_i)=O(\sum s_i)=O(\sum_{i\ is\ a\ leaf} s_i\times dep_i)$$

其中 dep_i 为点 i 的深度。

现在我们来求叶子节点深度的上界，以1号点为例，可以推出：

$$\begin{aligned} s_9 &= s_7+s_8 && (1)\\ &\geq 2s_6+s_3 && (2)\\ &\geq 3s_1+2s_2 && (3)\end{aligned}$$

(1)式推到(2)是因为6在8之前合并，所以可以得到 $s_8\geq s_6$ ，(2)到(3)同理。

从推导中可以得知， s_x 前的系数是斐波拉契数列。因此推广到任意点 i 有：

$$s_{root} \geq Fib(dep_i) \times s_i + Fib(dep_i - 1) \times s_x$$

其中 $Fib(x)$ 为斐波拉契数列第 x 项。 s_{root} 即为所有点权重之和 m。因此可以得到 $dep_i \leq O(log\ n)$

所以该情况的复杂度为： $O(m\ log\ m)$

其实这种做法非常具有拓展性。下面来看一个小问题，希望可以起到抛砖引玉的作用。

例6. *(使用了2015集训互测Robot(原创)的主要方法)*

假如有 n 个变量，它们的DP数组都已经通过本文的另一个方法“将键值函数化”化为了一条条折线函数， n 条折线的总线段数为 m 条，现在有 Q 次询问，每次询问所有变量在某点处的最小值。

$n, m \leq 10^5, Q \leq 5 \times 10^5$

定理2.1. 假如一条折线 l 是由段数分别为 $a_1, a_2, \ldots, a_m$ 的折线取 min 而成的。则 l 的段数是 $O(\sum a_i)$ 的。

关于定理 2.1 的证明，由于篇幅有限所以可以详见 *Robot* 解题报告。

根据定理 2.1 ，我们在合并两条段数分别为 x 与 y 的折线时就可以认为合出来的是一条段数少于 $x + y$ 的折线了。由于两条段数分别为 x 与 y 的折线取 min 要使用 $O(x + y)$ 的复杂度，所以直接使用 $T(x, y) = x + y$ 时的做法，每次选两条交点数少的线段求一下 min 。由于求出了最终的折线，所以复杂度为 $O((m + Q)log\ m)$ 。假如询问已经排好序了，那么复杂度可以降至 $O(m\ log\ m\ + Q)$

3 总结

*DP*的优化方法种类繁多，变幻莫测。我们应当细心分析题目性质，选择合适的优化方法。

4 感谢

感谢CCF提供学习和交流的平台。

感谢绍兴一中的陈合力老师、游光辉老师、董烨华老师多年来给予的关心和指导。

感谢清华大学的俞鼎力，董宏华学长对我的帮助。

感谢绍兴一中的王鉴浩、任之洲、贾越凯等同学对我的帮助和启发。

感谢其他对我有过帮助和启发的老师和同学。

感谢我的父母二十年如一日无微不至的关心和照顾。

5 参考文献

[1] topcoder SRM526 Algorithm Problem Set Analysis

[2] topcoder SRM639 Algorithm Problem Set Analysis

[3] Codeforces Round #265 Editorial

[4] Codeforces Round #172 Editorial

[5] Robot解题报告, by 张恒捷

Product 命题报告

宁波市镇海中学 杜瑜皓

1 试题

1.1 题目描述

令$N = \prod_{i=1}^{n} p_i^{a_i}, M = \prod_{i=1}^{n} p_i^{b_i}$，其中$p_i$是两两不同的素数。

求将N表示成k个数的乘积的方案数，也就是$N = \prod_{i=1}^{k} x_i$解的个数，答案对$10^9 + 21$取模。

对于子问题1，要求对于所有整数i满足$1 \le i < k$，都有$x_i < x_{i+1}$。

对于子问题2，要求对于所有整数i满足$1 \le i < k$，都有$x_i \le x_{i+1}$。

对两个子问题都要求对于所有整数i满足$1 \le i \le k$，都有$x_i \nmid M$。

只有子问题1答案正确将获得3分，只有子问题2答案正确将获得6分，两问都正确将获得10 分。

1.2 输入格式

第一行两个正整数n, k。

接下来一行n个非负整数，第i个表示a_i。

接下来一行n个非负整数，第i个表示b_i。

1.3 输出格式

一行两个整数，表示子问题1和2的答案。

1.4 数据范围

数据编号	n	a_i	b_i	k
1	≤ 5	≤ 5	≤ 5	≤ 3
2	≤ 10	≤ 20	≤ 20	≤ 5
3	$= 1$	$\leq 10^{18}$	$\leq 10^{18}$	≤ 25
4	≤ 50	$\leq 10^3$	$= 0$	≤ 20
5	≤ 50	$\leq 10^{18}$	$= 0$	≤ 25
6	≤ 50	$\leq 10^3$	$\leq 10^3$	≤ 10
7	≤ 50	$\leq 10^{18}$	$\leq 10^{18}$	≤ 10
8	≤ 50	$\leq 10^{18}$	$\leq 10^{18}$	≤ 15
9	≤ 50	$\leq 10^{18}$	$\leq 10^{18}$	≤ 20
10	≤ 50	$\leq 10^{18}$	$\leq 10^{18}$	≤ 25

时限为6秒，内存256MB。

2 一个计数的方法

考虑一个一般化的问题，对k个计数对象计数，这k个计数对象之间是等价的，同时这些计数对象总的有一个限制，求这样k个对象的集合或者可重集的个数。这里的顺序问题处理起来十分困难，所以我们要通过一些手段转化成求着k个对象的序列，或者转化成只有等价关系没有大小比较的序关系。

2.1 关于集合的做法

首先考虑集合的计数问题，由于所有数字都不同，所以只要求出两两不同的方案然后除掉$k!$即可。

所有数字两两不同，等价于$\frac{k(k-1)}{2}$个限制，$s_i \neq s_j$，于是可以考虑容斥。

也就是没有限制的方案数减去违反了某个限制的方案数加上违反了某两个限制的方案数等等的结果。

对于某个限制$s_i \neq s_j$，如果违反了也就代表着$s_i = s_j$。

如果容斥中枚举了方案的子集，也就是子集中每个限制对应的两个数必须相同，也就是所有数字划分成了若干个等价类。

如果暴力枚举这些限制是$O(2^{\frac{k(k-1)}{2}})$的。

注意只要关注每个等价类的大小就行了，具体的顺序和对应哪些数并不影响答案。因为这k个计数对象是不可区分的。

上述做法中枚举限制关系是瓶颈。我们可以换一个思路，枚举等价类的划分，所以也就是k的拆分数，记作P_k。然后对每一类拆分计算它在答案中会被统计到几次。

将每个数当成一个点，等价关系当成边，那么一个等价类相当于一个连通块。首先枚举连通块和点的对应关系，也就是点的编号。

假设这个拆分中有a_i个i，其中有$\sum_{i=1}^{k} a_i * i = k$，那么对应关系就相当于把$k$个数划分成若干个无序集合的方案数。

首先假设集合有标号，那么由多项式系数得方案为$\frac{k!}{\prod_{i=1}^{k}(i!)^{a_i}}$，然后大小相同的集合之间又是无序的，所以还要除掉$a_i!$。

所以方案数为$\frac{k!}{\prod_{i=1}^{k}(i!)^{a_i}a_i!}$。

点的对应方案确定，接下来要算边的系数。

令S_n表示n个点的连通图集合，$e(G)$表示G的边数。

假设这个划分为$p_1, p_2, \cdots, p_m$，那么系数为$\sum_{G_1 \in S_{p_1}} \cdots \sum_{G_m \in S_{pm}} (-1)^{\sum_{i=1}^{m} e(G_i)}$。

因为每个求和之间独立，也就是$\prod_{i=1}^{m} \sum_{G_i \in S_{p_i}} (-1)^{e(G_i)}$。

记$g_i = \sum_{G \in S_i} (-1)^{e(G)}$，所以系数也就是$\prod_{i=1}^{m} g_{p_i}$。

接下来考虑如何求g_n，也就是对所有n个点的连通图求和，对于图G，权值为$(-1)^{e(G)}$。

类比求连通图的做法，使用容斥，也就是首先考虑全集的权值，也就是n个点图的集合，减去不连通图的权值，也就是枚举点1所在的连通块。

首先假设有x条边，如果$x = 0$，那么全集的权值和为1，否则由二项式定理$\sum_{i=0}^{x} (-1)^i \binom{x}{i} = 0$。

对于不连通图，考虑1所在的连通块，假设有k个点，那么就有$\binom{n-1}{k-1}$个集合，然后对剩下$n-k$个点的所有图权值求和。

如果$n - k > 1$，那么和为0，所以当且仅当$k = n - 1$时，对g_n有贡献，也就是$g_n = -(n-1)g_{n-1}, g_1 = 1$，所以有$g_i = (-1)^{i-1}(i-1)!$。

结合这两部分，就能知道每个拆分对答案的贡献。

这样做是以$O(P_k)$和额外的限制若干个等价类转化到序列问题，也就是没有了序关系。等价关系比序关系好处理得多。

2.1.1 关于g_i取值的另一个证明

关于g_i的取值有一个不基于数学归纳法的证明，在这里补充。

对一个集合S，令g_S表示这个集合内点形成连通图的权值和。如果$|S| = 1$，那么$g_S = 1$。

否则考虑编号最小的两个点v_1, v_2。如果一个连通图不存在(v_1, v_2)这条边，那么将这条边加入到图中，还是个连通图，并且这两个图边数差1，所以恰好配对，权值和为0。考虑不能配对的连通图，也就是多出来的，这样的图中一定存在(v_1, v_2)这条边，并且去掉这条边整个图不连通，分成两个子集。

接着递归考虑这两个子集，考虑两个子集编号最小两个点连成的边。在不能配对的情况中，相当于去掉这条边这个子集不连通，接着分成两个更小的子集。

这样最后形成了一棵树的结构，对于每个点$v_i(2 \le i \le n)$，向$v_j(1 \le j < i)$连边，表示这条边是某个子集割边。同时也表示只有这样的图无法配对。

所以$g_i = (-1)^{i-1}(i-1)!$

2.2 关于可重集的容斥原理做法

类比上面的算法，可以这么想，首先可以枚举每种等价类的拆分S，假设大小分别为$p_1, p_2, \cdots, p_m$，然后计算出恰好有p_1 个数相同，p_2个数相同等等，然后每类数之间两两不同的方案数。每种拆分的重复计算的次数都是相同的。

首先考虑每个等价类重复计算的次数，然后将其除去得出有序的方案。对于每个等价类，求出的方案相当于按某个顺序排序了，只有若干个大小相同的等价类不可区分。如果一个拆分有a_i个大小为i的等价类，其中有$\sum_{i=1}^{k} a_i * i = k$，那么这$a_i$个是不可区分的，所以被重复计算了$a_i!$次。所以这个方案被重复计算了$\prod_{i=1}^{k} a_i!$，在总方案中除掉即可。

接下来要计算的等价类恰为某种划分的方案数。集合也就是两两不同为这个问题的一个特例。

如果使用容斥，对于一个划分，限制为每两个等价类之间的数字两两不同。所以可以暴力枚举这些限制，然后容斥。

这样做的代价为$O(2^{\frac{k(k-1)}{2}} P_k)$。

剩下要做的也就是快速计算对于拆分之间的容斥系数。

拆分之间构成了偏序集，如果拆分S通过合并其中某些等价类能得到T，那么称为$S \leq T$。

令$f(S)$表示所有等价类T满足$S \leq T$的划分的方案数的和，$g(S)$表示等价类恰为S的集合。

那么$f(S)$就是对于每个划分dp出来的方案，$f(S) = \sum_{S \leq T} g(T) * w(S,T)$，其中$w(S,T)$ 表示重复的方案数，也就是T 这个划分可能在S中被计数多次。

而要计算的是$g(S) = \sum_{S \leq T} f(T) * \mu(S,T)$，也就是要计算$\mu(S,T)$。

考虑状压dp，首先枚举集合S，然后每次选出一个子集，将其合并成为一个数字。类似前文的论证可得将i个等价类合并的权值为$(-1)^{i-1}(i-1)!$，而一个方案最后的权值为所有等价类合并权值的积。

为了防止重复计数，可以对子集进行编号，也就是让每次选出的子集编号递增，所有子集个数为$\sum_{i=1}^{k} P_i$。

那么状态可以表示为$dp_{S_1,S_2,p}$，即现在已经选完编号在$1 \leq p$之间的子集，之前选出的子集合并后的数集合为S_1，剩下没合并的数集合为S_2 的权值总和。

转移的时候假设当前考虑合并子集T，那么转移到$S_1 \cap \{\sum T\}, S_2 \backslash T$这个状态，其中$\sum T$表示$T$中数的和，同时乘上将$T$这个集合内等价类合并的权值和在$S_2$中选出$T$这个子集的方案。如果$T$这个子集被删去了$k$次，那么要在最后除掉$k!$这个系数，因为这$k$ 个子集不可区分。

这样总状态数为$O((\sum_{i=1}^{k} P_i)^3)$，然而对某个$S$的集合做的时候，$S_1$一定是$S$的子集能合并出来的状态，$S_2$一定是$S$的子集，$p$也只要对$S$的子集编号即可，所以远远达不到上面的复杂度。

考虑最后的答案，将$g(S)$全部表示成$f(S)$的线性组合，那么答案一定能被表示成$f(S)$的线性组合，也就是$\sum f(S) * w(S)$。

那么这样只要枚举将S划分成多少个子集，然后统计对应的系数。只要爆搜每个数字的划分方案，然后记忆化即可，这样做更加方便。

2.3 关于可重集的Burnside引理做法

换个角度考虑问题，那么可重集相当于就是问在在所有k的置换也就是对称群S_k作用下不同构的元素个数，即在对称群S_k 下轨道的个数。所以可以用Burnside引理解决。

对于某个置换P，将其拆成若干个轮换，如果一个方案在这个置换下不变就

相当于在同一个轮换中的元素相同。

所以相当于一个置换将一些元素划分成了若干个等价类。枚举等价类的划分，也就是k的拆分，然后计算多少个置换对应的等价类是这样的。

和前面相同，首先考虑将元素划分到若干个等价类的方案。假设这个拆分中有a_i个i，其中有$\sum_{i=1}^{k} a_i * i = k$，方案数为$\frac{k!}{\prod_{i=1}^{k}(i!)^{a_i}a_i!}$。

然后对于一个轮换长度为i，那么对应着$(i-1)!$个圆排列，所以方案为$\frac{k!}{\prod_{i=1}^{k} i^{a_i}a_i!}$。

2.4 从群论的角度考虑集合的计数

我们注意到一个这样的事实，对于一个等价类的划分，计算可重集时Burnside引理得到的系数和计算集合时从容斥原理得到的系数除了符号全部相同，更一般的有假设有m个等价类，那么容斥系数为$(-1)^{k-m}$倍。

如果一个置换σ由m个轮换[1]组成，那么$(-1)^{k-m} = \mathrm{sgn}(\sigma)$。

类似Burnside引理，有对每个σ属于S_k令X^σ表示X中在σ作用下的不动元素。由这个容斥得到的式子为：

$$\frac{1}{k!}\sum_{\sigma\in S_k}|X^\sigma|\mathrm{sgn}(\sigma)$$

如果一个序列中的元素两两不同，那么它只在恒等置换作用下不动。由于这个序列的所有置换都被算入，最后又除掉了$k!$，所以这样的序列恰好被算了一次。

假设一个序列中等价类的大小为$p_1, p_2, \cdots, p_m$，那么在一个置换作用下这个序列不动，必有这个置换中的每个轮换的元素属于同一个等价类。

如果存在一个等价类大小超过1，记作t，因为交错群$|A_t| = t!/2$，所以奇置换和偶置换个数相同，也就是对应的和为0，所以如果有两个元素相同，那么在上面的式子中被算入0次。

不过对于一般的置换群，关于只在恒等置换作用下不动的元素计数，并没有上面这样简单的式子，并且我也没有找到相关的资料。

关于这个问题，我也在思考中，也欢迎读者来交流。

[1]这里一个置换的轮换个数默认为完全轮换分解中轮换的个数，下同。

2.5 第一类Stirling数

对于这些计数对象，如果加的限制为只与等价类的大小有关，那么可以有多项式的做法。

我们只要知道对称群S_k中轮换个数为m的置换个数，也就是将k个元素的分作m个圆排列的方法数，也就是$(-1)^{m+k}s(k,m)$，其中$s(k,m)$为第一类Stirling数。

其中$\prod_{i=0}^{k-1}(x-i)=\sum_{i=1}^{k}s(k,i)x^i$。

如果是集合的计数，那么对应的系数恰为$s(k,m)$。使用这个结论，可以很方便的解决一些问题。

2.5.1 例题 CountTables[2]

求$n*m$的数表个数，要求每个数在$1\sim c$之间，并且要求任意两行不能完全相同，任意两列不能完全相同。

分析一下这个题的计数对象是大小为n的向量，并且每一维在$1\sim c$之间，额外的限制为不存在某两维完全相同。

假设有i个等价类，那么每一维有c^i个取值，两两不同的方案数为$\binom{c^i}{n}*n!$。

于是答案为$\sum_{i=1}^{m}s(m,i)\binom{c^i}{n}*n!$。

这个题通过直接dp和Stirling反演能得到同样的式子，但是通过这个方法就能很轻松地得到这个问题显示的解。

3 一个不定方程的非负解数

接下来再考虑一个方程$\sum_{i=1}^{m}p_ix_i=a$解数的算法。

3.1 基于动态规划的做法

可以看成一个简单的完全背包计数。那么可以在$O(ma)$的代价用背包解决。

令$dp_{i,s}$表示选了前i个数，总和为s，那么$dp_{i,s}=dp_{i,s-p_i}+dp_{i-1,s}$。

[2]来源: TCO 2014 Wildcard 750分题

3.2 基于循环节的做法

引理： $\sum_{i=1}^{m} p_i x_i = a$的非负整数解个数是一个以$a$模$\mathrm{lcm}_{i=1}^{m} p_i$为周期的关于$a$的次数不超过$m-1$的多项式。

简单的证明，这相当于$\frac{1}{\prod_{i=1}^{m}(1-x^{p_i})}$的第$a$项，可以将分母的所有根求出来，表示成若干个$(1-\omega_j^i x)$的乘积。然后一定可以拆成若干个部分分式，也就是$\frac{p(x)}{(1-\omega_j^i x)^k}$的和，其中$p(x)$为一个关于$x$的次数不超过$k-1$的多项式。

对于每项，分母可以用二项式定理展开$\frac{1}{(1-\omega x)^k} = \sum_{i\geq 0}\binom{i+k-1}{k-1}\omega^i x^i$，其中$\binom{i+k-1}{k-1}$是一个次数不超过$k-1$的多项式，假设$\omega$是$j$次单位根，那么这一部分是模$j$为周期的多项式。易证将这个幂级数乘上$p(x)$还有同样的性质。

所以若干个加起来就有引理的结论了，并且次数不超过重数最多的根也就是1的重数。

如果知道了前$m * \mathrm{lcm}_{i=1}^{m} p_i$项的取值，那么可以在$O(m)$的时间复杂度内求出某项的值。

首先可以得到模周期的$m+1$个值$f(0), f(1), \cdots, f(m)$，假设要求的是$f(n)$，那么由拉格朗日插值得$f(n) = \sum_{j=0}^{m}(-1)^{m-j} f(j)\frac{\prod_{i=0}^{m}(n-i)}{(m-j)!\,j!\,(n-j)}$。这个可以在$O(m)$内算出。

3.2.1 引理的一个简单证明

令$M = \mathrm{lcm}_{i=1}^{m} p_i$，令$x_i = \frac{M}{p_i} t_i + r_i (0 \leq r_i < \frac{M}{p_i})$。如果枚举了$r_1, r_2, \cdots, r_m$，那么方程变成$M\sum_{i=1}^{m} t_i + \sum_{i=1}^{m} p_i * r_i = a$。令$c = \sum_{i=1}^{m} p_i * r_i$，如果$a \equiv c \pmod{M}$，那么就变成$\sum_{i=1}^{m} t_i = \frac{a-c}{M}$。因为$c \leq mM$，所以$\frac{a-c}{M} \geq -m+1$，那么解数为$\binom{\frac{a-c}{M}+m-1}{m-1}$。

对于模M同余的某个数，都是有限个次数不超过$m-1$的多项式相加，所以也是个多项式。

3.3 基于生成函数的做法

所以解的个数为幂级数$\prod_{i=1}^{m}\sum_{j\geq 0} x^{jp_i}$中第$a$项的系数。

$$\prod_{i=1}^{m}\sum_{j\geq 0} x^{jp_i} = \prod_{i=1}^{m}\frac{1}{1-x^{p_i}} = \frac{1}{\prod_{i=1}^{m}(1-x^{p_i})}$$

也就是$\frac{1}{\prod_{i=1}^{m}(1-x^{p_i})}$中第$a$项前的系数。

将其看成一个数列的生成函数，这是一个常系数线性递推数列。将分母展开就能得到得到递推的系数，而通过动态规划能得出这个数列初始值。

令$s = \sum_{i=1}^{m} p_i$，通过矩阵快速幂可以做到$O(s^3 \log a)$，通过倍增可以做到$O(s^2 \log a)$，可以使用FFT加速做到$O(s \log s \log a)$。

3.4 基于数位dp的做法

令$x_i = \sum_{j\geq 0} 2^j x_{i,j}, (0 \leq x_{i,j} \leq 1)$，于是相当于物品为$p_i * 2^j$的01背包。令$dp_{i,j}$表示只考虑$2^j$倍数的物品，剩余的体积为$i * 2^j$的方案数。剩余的物品体积不会超过$s * 2^j$，所以当$i \geq s$时不可能再填满。

所以状态数不会超过$s \log a$，然后对于每个物品使用01背包转移，时间复杂度为$O(ms \log a)$。

3.5 基于部分分式的做法

类似引理的第一个证明方法，可以将它拆成若干个部分分式，然后分别计算。当p_i两两互质时，可以使用单位根的技巧做到$O(ms)$的复杂度。不互质时也可以暴力拆成部分分式。然后在$O(m)$的复杂度解决。

由于过于复杂，并且与这个题关系不大，所以略去。有兴趣的读者可以阅读参考文献[2]。

4 关于前三个点的算法

在这个问题里面计数对象为存在一个j满足$s_{i,j} > b_j$的n维向量，限制为对于某一维j满足$\sum_{i=1}^{k} s_{i,j} = a_j$。

令$x_i = \prod_{j=1}^{n} p_j^{s_{i,j}}$，也就是每个数的因子分解。

满足$\sum_{i=1}^{k} s_{i,j} = a_j$。因为$x_i \nmid M$，所以对于所有$i$，存在$j$使得$s_{i,j} > b_j$。

记e为a_i, b_i中的最大值。

4.1 算法1

爆搜出所有可行方案，然后验证是否合法。

4.2 算法2

第二个点中$k \le 5$。考虑使用动态规划一个一个因子确定，同时保证s_i按字典序不降。

当两个前缀$s_{i,1..j}, s_{i+1,1..j}$确定时，如果$s_{i,1..j} < s_{i+1,1..j}$，那么$s_{i,j+1}$和$s_{i+1,j+1}$之间没有限制。如果$s_{i,1..j} = s_{i+1,1..j}$，那么要有$s_{i,j+1} \le s_{i+1,j+1}$。

这个时候要用一个状态S表示相邻两个前缀相同的集合和对于每个前缀i是否已近存在j满足$s_{i,j} > b_j$。令$f_{i,S}$表示处理完前i个因子状态为S 的方案数。同组内转移记$g_{S',i,j,k}$ 表示当前集合为S'，选了i个数，总和为j，上一个数为k，枚举每个数的取值，然后更新当前状态。

同组内状态数为$O(e^2k4^k)$，转移复杂度为$O(e)$，共有$O(n)$组。

时间复杂度为$O(ne^3k4^k)$。

4.3 算法3

第三个点中$n = 1$，令$t_i = s_{i,1}$，所以问题等价于求$\sum_{i=1}^{k} t_i = a_1$，然后$M \nmid x_i$，也就是$t_i > b_1$。

因为$t_1 \le t_2 \le \cdots \le t_n$，所以就是$t_1 > b_1$。

记$\delta_i = t_{k+1-i} - t_{k-i}$，令$t_0 = 0$，那么$\sum_{i=1}^{k} t_i = \sum_{i=1}^{k} \delta_i * i = a_1$。

求这个方程的非负整数解，并且对于δ_i有$\delta_i \ge c_i$这样的限制。对于子问题1，有$c_1 = b_1 + 1$，对于所有$i \ne 1$，$c_i = 1$。对于子问题2，有$c_1 = b_1 + 1$，对于所有$i \ne 1$，$c_i = 0$。

就相当于$\sum_{i=1}^{k} x_i * i = a_1 - \sum_{i=1}^{k} c_i * i$的非负整数解。

套用前面的生成函数或者数位dp做法就可以通过。

5 后面几个点的做法

5.1 算法1

上面的结论还不能直接解决这个问题，因为它只是转化到了有若干个等价类的方案数。对于这个问题中特殊的计数对象还要特别考虑。

首先考虑一个简单的问题即没有限制的情况。在上一步转化下就变成某些变量是相等的，然后每一维加起来和要等于这一维对应的值。

那么可知每一维都是独立的，也就是算出每一维分解成若干个等价类，然后将所有答案乘起来的方案数。

假设等价类的大小为$p_1, p_2, \cdots, p_m$，这一维总和为a，那么就是将$\sum_{i=1}^{m} p_i x_i = a$的非负整数解的个数。

那么可以在$O(ma)$的代价用背包解决，$dp_{i,s}$表示做了前i个等价类，总和为s，那么$dp_{i,s} = dp_{i,s-p_i} + dp_{i-1,s}$。

所以这个问题可以在$O((n+ke)P_k)$内解决。

记$f(N,k)$表示将N分解成k个严格递增的数的方案数，$g(N,k)$表示将N分解成k个不降的数的方案数。

回到原问题，对于$b_i = 0$的情况就是$x_i \neq 1$。

对于子问题2，答案就是$g(N,k) - g(N,k-1)$，也就是去掉$x_1 = 1$的方案数。

对于子问题1，令$h(N,k)$表示将N分解成k个严格递增的数且第一个数超过1的方案数，那么$h(N,k) = f(N,k) - h(N,k-1)$。也就是如果$x_1 = 1$，那么一定有$x_2 \geq 2$。所以最后的答案为$\sum_{i=0}^{k} f(N,i)(-1)^{k-i}$。

时间复杂度为$O((n+ke)\sum_{i=1}^{k} P_i)$。

5.2 算法2

接下来考虑如何处理存在一个j满足$s_{i,j} > b_j$的n维向量的计数问题。

考虑使用容斥，如果不存在j满足$s_{i,j} > b_j$，也就是$s_{i,j} \leq b_j$。

对于每个等价类，枚举对应的变量是否违反限制也就是M的约数。如果一个变量违反了限制，也就是每一维不能超过b_j。

假设等价类的大小为$p_1, p_2, \cdots, p_m$，令$dp_{i,s}$表示做了前i个等价类，总和为s，那么$dp_{i,s} = dp_{i,s-p_i} + dp_{i-1,s} - dp_{i,s-(b_j+1)*p_i}$。即减去这一维超过$b_j$的方案。

时间复杂度$O(P_k 2^k (ke+n))$。

5.3 算法3

首先考虑加快容斥的情况，如果有a_i个大小为i的等价类，那么这a_i个等价类是不可区分的，也就是如果其中x个违反了限制，对应的情况数为$\binom{a_i}{x}(-1)^x$。所以对于一个划分，只要做$\prod_{i=1}^{k}(a_i+1)$种情况就好了。

当$k = 25$时所有划分的情况总和为129512，为了方便描述，将这个记为k的一个函数Q_k。

有前面的做法可知，如果知道了前$m*\mathrm{lcm}_{i=1}^{m} p_i$项的取值，那么可以在$O(m)$的时间复杂度内求出某项的值。

为了方便描述，令所有拆分中对应要知道取值的项记为R_k。当$k = 25$时，R_k[3]为5040。

在枚举这些拆分的时候可以用dp顺便算出这R_k个初值。

接着考虑这些限制。对于第j维，如果一个大小为p_i的等价类内数只能取$0\cdots b_j$，那么对应的生成函数为$\frac{1-x^{(b_j+1)p_i}}{1-x^{p_i}}$。所以分母为$\prod_{i=1}^{m}(1-x^{p_i})$，分子中因为指数都为$(b_j+1)$的倍数，所以可以当成$x^{b_j+1}$的$m$次的多项式，在$O(m^2)$ 复杂度内展开，并且对于n维形式都相同。

此时分子至多有m项，那么可以对每项分别算对答案的贡献。假设第i项为wx^c，那么对应的答案为$\frac{1}{\prod_{i=1}^{m}(1-x^{p_i})}$的第$a-c$ 项乘上w，所以可以在$O(nk^2)$的时间复杂度内算出答案。

注意对于每个拆分，每一维要求的值为$a_j, a_j-(b_j+1), \cdots, a_j-m*(b_j+1)$，这些项可以在$O(nk^2)$内时间预处理出来，可以将上面每个容斥的复杂度降到$O(nk+k^2)$。

其中$Q_k = \sum_{i=0}^{k} P_i P_{k-i}$，因为$Q_k$表示将数字分成两部分，一部分选入，一部分不选入，两边分别划分的方案总和。

所以总时间复杂度为$O(R_k \sum_{i=1}^{k} P_i + nk^2 P_k + (nk+k^2)(\sum_{i=0}^{k} P_i P_{k-i}))$。

6 总结

这个题的本意是对于一个特殊的计数对象求它的集合和可重集个数。在第二节中叙述了怎么将这个问题转化，通过了容斥原理和Burnside引理，最后引入了群的作用的观点，直观地统一地解释了这两个方法。这个做法对一般的计数对象都是可行的，并且给出了它方便地解决算法竞赛题的一个例子。

对于可重集的计数使用容斥原理，是一个不太成功的尝试，因为它打破了问题的对称性，所以失去了很多优美的性质。但是它能得到一些更微观具体的结果，比如某个特定结构的方案数。

由于本题的特殊性，第三节介绍了求一类不定方程解数的一些算法，要结合实际实际情况，选择最简便并且效率可以接受的做法。第四第五节介绍了这

[3]这个可以近似看成S_k中阶最大的元素的阶乘上k，关于这个的渐进复杂度可以查阅参考文献[1], [3]

个问题几个特殊点的做法和一般的做法。

本题是一道难题，主要考察选手的计数水平和数学功底，包括Burnside引理，容斥原理，生成函数，多项式等方面的知识和灵活的运用。

对于一般置换群轨道的计数已经由Burnside引理可以解决，关于只在恒等置换作用下不动的元素计数，或者某种特定结构的元素的计数还没有什么好的方法。

虽然这并不是一个简单的问题，不过也希望大家包括我自己能提出一些更有效的算法。

致谢

感谢中国计算机学会提供学习和交流的平台。

感谢李建老师的关心和指导。

感谢上海交通大学郭晓旭同学和杭州学军中学徐寅展同学的交流和帮助。

参考文献

[1] E. Landau, “Uber die Maximalordnung der Permutationen gegebenen Grades”

[2] 黄志翱, “Codechef CHANGE解题报告”

[3] Jean-Pierre Massias, Jean-Louis Nicolas, and Guy Robin, “Effective Bounds for the Maximal Order of an Element in the Symmetric Group”

[4] Hansraj Gupta, “On the Partition of J-partite Numbers”

关于以源代码为输入的一类问题的初步探索

宁波镇海蛟川书院 卢啸尘

摘 要

在ACM-ICPC竞赛中，时常会有以源代码为输入的问题作为考察选手综合代码能力的问题而出现。本文介绍了解决此类问题时的一些实现技巧，并将此类问题与OI竞赛中常用的数据结构相结合，提出了数据结构题的一种新表现形式。

1 引言

在信息学竞赛中，大部分试题都可以被归为想法题、算法题和代码题这三类问题之一。想法题考察选手的创新能力，算法题考察选手的知识面，而代码题考察选手的基本功，包括代码能力和后续的调试能力。OI中比较经典的代码题除了高级数据结构题以外，还有山东省选的《猪国杀》，浙江省选的《杀蚂蚁》等。作者认为，这类以游戏作为背景的问题，存在着部分没有接触过有关游戏的选手无法准确理解题面的弊端。以源代码为输入的问题，其模型贴近选手的日常训练，就不存在这样的弊端，是代码题命题的一种良好选择。

作者整理了一些以源代码为输入的问题，将其分为三类，并对其进行了一些总结。本文的框架如下：

第二节中，介绍了一些预备知识：编译器的结构、程序设计语言的实现方式，以及符号约定。

第三节中，给出了一个基础问题。该问题的特点是不需要对于源代码的具体语义进行分析，而只需要分析代码结构。此节中介绍了词法分析器的实现。

第四节中，给出了几个实现难度较大的问题。这类问题要求对给出的程序代码进行模拟。此节中介绍了程序设计语言的混合实现系统的一种比较方便的代码实现。

第五节是本文的重点，在此节中，作者给出了几个变种问题。这类问题以消除不可达代码为模型，将数据结构和代码分析结合起来。

第五节中的所有题目具有同一个性质：“有修改，有回退修改”，这是一个弱于”可持久化”和”有修改、有取消修改”而强于”有修改”的性质。以这类问题为启发，我们可以设计一些试题（甚至是并非编译类问题的问题），它们具有明显强于单纯”有修改”的要求，又没有现成的优秀解法。而在降低要求到这一要求后就可以设计出高效的算法。

2 预备知识

2.1 编译器的结构

一个编译器的工作是，将一个高级语言的源代码映射到另一种高级语言或机器语言的代码。因此编译器被分为前端和后端两个部分。

前端被称为分析(analysis)部分。这个部分的工作是，将源代码解析成基本单元，并根据语法创建一个中间表示，中间表示通常不是另一种高级语言，因为高级语言难以被直接处理，也不是一种机器码，因为机器码和代码差的太远。常见的中间表示有“三地址码”等。

后端被称为综合(synthesis)部分。这个部分的工作是，根据前端传回的各要素和中间表示，创建目标语言的代码。根据编译器目标语言的不同，这里可能对中间表示进行进一步简化而形成机器语言，也可能生成一个高级语言的源代码。

每个部分又可以被分为若干步骤。前端包括词法分析、语法分析、语义分析、中间代码生成四个步骤。后端包括代码生成这一个步骤。有的编译器会在前端和后端之间设置一个机器无关代码优化步骤，或在后端代码生成步骤后增加一个机器有关代码优化步骤。

至于预处理器、汇编器和链接器，一般不认为其是编译器的一部分。

下面将说明这五个步骤的大致工作。以下图语句为例，其中A是double类型，B是int类型。

```
A = A + B * 123
```

2.1.1 词法分析

词法分析器接受一个字符串（即源代码）作为其输入，输出一个词法单元（词素）列表。

在上面的例子中，源代码被分为以下词素：$\{A, =, A, +, B, *, 123\}$。词素$A$与$B$被识别为变量标识符，词素=，+和*被识别为算符，123被识别为整数字面值。词法分析器的输出如下:

$$
\begin{array}{ll}
Identifier & id = 1 \\
Assign & \\
Identifier & id = 1 \\
Operator & ”+” \\
Identifier & id = 2 \\
Operator & ”*” \\
Literal & 123
\end{array}
$$

在编译器发展的早期阶段，词法分析器是作为一个独立的部分工作，它一次性地处理整个源文件。但是现在词法分析器通常是作为语法分析器的子程序而出现：语法分析器在工作时不时调用词法分析器，词法分析器从输入流中读取一系列字符直到构成了一个词素，词法分析器每次向它的调用者——语法分析器返回接下来的一个词素。也就是说，以上表为例，第一次调用词法分析器返回上表第一行的内容，第二次是第二行，以此类推。

2.1.2 语法分析

语法分析器接受一系列词素，分析其是否在语法上是正确的。然后如果是正确的，语法分析器返回一棵语法生成树。

```
STATEMENT
  LVALUE
    IDENTIFIER: Identifier [1]
  ASSIGN: Assign
  EXPRESSION
    EXPRESSION
      IDENTIFIER: Identifier [1]
    OPERATOR: Operator "+"
    EXPRESSION
      EXPRESSION
        IDENTIFIER: Identifier [2]
      OPERATOR: Operator "*"
      EXPRESSION
        LITERAL: Literal(Integer) 123
```

（上表并没有明确地指出语法分析器是如何处理不同运算符的优先级的，

这是因为此表只是用来说明语法分析器的大致任务）

虽然在实践中这棵语法生成树并不显式地建出，但是一个语法分析器需要强到可以建出一棵语法生成树，这就是说，至少可以追踪这棵语法生成树的结构。

大多数语言现象都可以使用上下文无关文法描述，因此我们有一个普适的三次方的算法，即CYK算法。它的效率不高。但是计算机科学家们牺牲了普适性换取了效率：对于上下文无关文法的某些特定子集，存在线性的算法。我们将在后面的章节中介绍其细节。

2.1.3 语义分析

语义分析器在语法分析树上对程序的语义进行审查。如类型检查。在这一阶段，一些语义方面的属性被附加到文法符号上。

```
STATEMENT
  LVALUE(double)
    IDENTIFIER: Identifier [1](double)
  ASSIGN: Assign
  EXPRESSION(double)
    EXPRESSION(double)
      IDENTIFIER: Identifier [1](double)
    OPERATOR: double operator + (double, double)
    INT_TO_DOUBLE(double)
      EXPRESSION(int)
        EXPRESSION(int)
          IDENTIFIER: Identifier [2](int)
        OPERATOR: int operator * (int, int)
        EXPRESSION(int)
          LITERAL: Literal(Integer) 123(int)
```

考虑表达式计算：由于存在运算符重载等特性，在语法分析阶段通常无法确定某一运算符具体对应哪一函数（特指那些在语法分析树高处的运算符）。在语义分析阶段，由于已经确定了每个子树（子表达式）的类型，也就确定了每一运算符对应的函数。因此，在语义分析之后，这个程序变得可执行。

2.1.4 中间代码生成

我们希望编译器的前端和后端相互独立。这就是说，如果中间代码的格式被约定了，一旦有N个前端（同一源语言或不同源语言的）和M个后端（同一目标语言或不同目标语言的），就能得到NM个不同的编译器，每一个把某种语言的程序翻译成另一种语言。

然而，虽然语义分析后的程序是可解释的，但是语法分析树仍然带有源语言的色彩——如语法结构等。这明显不符合我们对中间代码的要求。因此，我们需要一个中间代码生成器。

常见的中间表示形式是三地址码。在三地址码之前有时还有一种表示形式——语法DAG，它在优化上有重要作用，但是本文不涉及优化所以略去语法DAG的内容。下图是示例的三地址码：

```
MULTIPLY(t3, id2, 123) [t3 = id2 * 123]
INT_TO_DOUBLE(t2, t3) [t2 = int_to_double(t3)]
ADD(t1, id1, t2) [t1 = id1 + t2]
ASSIGN(id1, t1) [id1 = t1]
```

2.1.5 代码生成

代码生成器将中间表示转化为目标代码。

我们注意到，三地址码涉及的都是一些基本操作。所以如果不考虑效率的话，很容易从中间表示生成一个目标代码，只要为三地址码的每一指令设计一个目标代码框架就可以了。但这样通常伴随着大量的冗余代码。本文不涉及优化，所以这里不再赘述了，有兴趣的读者可以查阅参考文献[1]。

2.2 程序设计语言的三种实现方式

程序设计语言是来用描述程序的一段代码。这段代码无法被机器直接识别运行。因此，我们需要通过某种方法，将语言所描述的程序转变为机器可识别运行的形式，这被称为程序设计语言的实现方式。常见的三种实现方式是：完全编译、完全解释和混合实现。

2.2.1 完全编译

在这种方式下，编译器一次性地将整个程序编译成可执行的机器码。其特点是编译阶段和运行阶段分离。使用这种模式的语言一般运行效率较好。C++就是使用完全编译方式的语言。

2.2.2 完全解释

这种方式类似于部署了一个该语言的虚拟机：解释器每次找到源代码中的一个语句并翻译它。注意到这样一来，同一个语句可能被翻译数次，这也就说明了为什么使用完全解释模式的语言一般运行效率较差。然而完全解释也有优点：首先，可移植性好；其次，在调试时，很容易定位错误发生的位置。SHELL是一种完全解释方式的语言，有很多脚本语言都是完全解释的。

2.2.3 混合实现

这一方式是编译和解释的折中。在这种方式下，编译器一次性地处理整个程序，但只使用前端——这样就得到了中间表示，然后对中间表示进行解释。由于中间表示已经经过了一定的格式化，它比源代码更加易解释，因此多次解释同一语句的代价也就小多了。这类语言有着略低于完全编译语言的效率，同时又有完全解释语言的高可移植性。Java就是一种混合性语言，Java代码经过编译生成“字节码”，然后在Java虚拟机上解释运行字节码。

2.3 巴克斯-诺尔范式

巴克斯-诺尔范式被用来描述上下文无关语言。在下面的章节中描述程序文法时将大量使用巴克斯-诺尔范式。

其格式为A ::=< $String$ >。其中A是非终结符号，< $String$ >是一个符号串，可以为空串。

比如以下文法就描述了整数四则运算的语法：

```
EXPR ::= TERM
EXPR ::= TERM + TERM
EXPR ::= TERM - TERM
TERM ::= FACTOR
```

```
TERM ::= FACTOR * FACTOR
TERM ::= FACTOR / FACTOR
FACTOR ::= (EXPR)
FACTOR ::= integer_value
```

其中*integer_value*是整数字面值，是终结符号。在编译器中，整数字面值由词法分析器，而非语法分析器定义。

3 基本问题: 不涉及语义的问题

3.1 代码长度统计问题[1]

统计一段Pascal代码的代码长度。代码长度用“单元”的数量定义，每个保留字、每个标识符、每个字面值、每对花括号、每对括号、每个运算符被作为一个”单元”。注释不参与统计。

```
{THIS IS COMMENT}
‘THIS IS A STRING’
(A * A) shr 1
```

以上文为例：第一行只有一个单元，即左花括号；第二行也只有一个单元，即字符串字面值；第三行有6个单元，包括一个保留字，两个标识符，一对括号，一个运算符，一个整数字面值。

很明显，这个问题只需要选手实现一个词法分析器。

3.1.1 词法分析器

词法分析器的实现方法类似于读入优化的写法。首先读入若干个空白字符直到遇到一个非空白字符。根据这个非空白字符的类型判断这个词素的类型。在这个问题中，我们可以认为只有三种非空白字符类型：数字类型（0 – 9），标识符头类型（_、A-Z、a-z），其他。

```
const int TEMPTY = -1;
const int TDIG = 0;
const int TSHEAD = 1;
const int TOTHER = 2;

int charType(char x)
{
    if (x == ' ') return TEMPTY;
    if (x == '\n') return TEMPTY;
    if (x == '\t') return TEMPTY;
    if (x == '\r') return TEMPTY;
```

[1]UVA 189. Pascal Program Lengths, ACM-ICPC 南加利福尼亚区域赛(1989/1990赛年)

```
    if (x == '_') return TSHEAD;
    if ((x >= '0') && (x <= '9')) return TDIG;
    if ((x >= 'A') && (x <= 'Z')) return TSHEAD;
    if ((x >= 'a') && (x <= 'z')) return TSHEAD;
    return TOTHER;
}
```

为了方便在“向前看”的同时不改变输入流中的内容，我们另外写一个输入流管理器来代替标准输入流。输入流管理器有一个缓冲区，这个管理器要支持：返回输入流头的字符，从输入中取得一个字符，因此缓冲区的大小为1。

```
namespace Stream
{
    char buffer;
    void Init()
    {
        buffer = 0;
    }
    char nxtChar()
    {
        if (buffer == 0)
            buffer = getchar();
        return buffer;
    }
    char getChar()
    {
        char ret = nxtChar();
        buffer = 0;
        return ret;
    }
}
```

接下来词法分析器将只调用新的输入流管理器，而不调用标准输入。

```
namespace Lex
{
  string buffer;
    // 缓冲区
  void Init()
  {
```

```
    buffer = "";
  }
  string nxtLexeme()
  {
    if (buffer.size()) return buffer;
    while (charType(Stream::nxtChar()) == TEMPTY) Stream::getChar();
    // 跳过空白字符
        int TYPE = charType(Stream::nxtChar());
    if (TYPE == TDIG)
      {
        while (charType(Stream::nxtChar()) == TDIG)
          buffer += Stream::getChar();
        return buffer;
      }
    if (TYPE == TSHEAD)
      {
        int t;
        while (((t = charType(Stream::nxtChar())) == TDIG) || (t ==
       TSHEAD))
           buffer += Stream::getChar();
        return buffer;
      }
    // TOTHER
    switch (Stream::nxtChar())
    {
      case '+': case '-': case ',':
      case '*': case '/': case '=':
      case '^': case '@': case ';':
      case '(': case ')': case '.':
      case '~': case '[': case ']':
        return buffer += Stream::getChar();
      break;
      case '>': case ':'
        buffer += Stream::getChar();
        if (Stream::nxtChar() == '=')
          buffer += Stream::getChar();
        return buffer;
      break;
      case '<':
        buffer += Stream::getChar();
        if ((Stream::nxtChar() == '=') || (Stream::nxtChar() == '>'))
          buffer += Stream::getChar();
        return buffer;
      break;
    };
  }
    string getLexeme()
    {
```

```
54          string ret = nxtLexeme();
55          buffer = "";
56          return ret;
57      }
58  }
```

这里我们注意到，这段代码中使用string来储存词素。我们知道，除了注释内容和字符串字面值以外，每个词素的类型都可以通过首字符类型判断出：TSHEAD类型的首字符说明一个标识符或关键字（注意在这道题中这两者等价），TDIG类型的首字符说明一个整数字面值开始。

3.1.2 引入科学计数法

上面的那个做法对于原来的这道题是不够的：在原题中数可能以科学计数法读入，并且因此，可以是实数，自然也可能以小数形式出现。

没有科学计数法的词法分析器已经适用于之后的大部分题目。没有兴趣的读者可以认为本题没有这一要素，略过本节。

在这之前先考察本题中的科学计数法是怎么定义的：”The most general form of a numeric constant is illustrated by the constant 10.56E-15. The 10 is the integral part (1 or more digits) and is always present. The .56 is the decimal part and is optional. The E-15 is the exponent and it is also optional. It begins with an upper or lower case E, which is followed by a sign (+ or -). The sign is optional.”

小数点前至少有一个数字。小数点到E前的部分是可选的，E之后的部分也是可选的。

很容易写出这样的代码：

```
1  if (TYPE == TDIG)
2    {
3      int Statu = 0;
4      do
5        {
6          while (charType(Stream::nxtChar()) == TDIG)
7            buffer += Stream::getChar();
8          if ((Statu <= 1) && (Stream::nxtChar() == '.'))
9            {
10             Statu = 1;
11             buffer += Stream::getChar();
12           }
13         else
```

```
14          if ((Statu <= 2) && (Stream::nxtChar() == 'E'))
15            {
16              Statu = 2;
17              buffer += Stream::getChar();
18              buffer += Stream::getChar();
19            }
20          else
21            Statu = 3;
22      }
23    while (Statu <= 2);
24    return buffer;
25  }
```

3.1.3 引入注释

Pascal中注释的策略是：每次找到第一个未被标记为注释或字符串一部分的左花括号，找到这之后的第一个右花括号，它们之间的部分构成注释。读者可以通过在IDE中输入代码{{}}来确认这一点，应当会发现第四个字符不呈现注释的颜色。

那么代码就很简单了。

```
case '{': {
  while (Stream::nxtChar() != '}')
        buffer += Stream::getChar();
  return buffer += Stream::getChar();
}
```

3.1.4 引入字符串

Pascal中识别字符串的策略是：每次找到第一个未被标记为注释或字符串一部分的引号，找到这以后的第一个后面紧跟的不是引号的引号。读者可以通过试着编译以下代码来验证这一点，应当会发现编译错误。

```
begin
  writeln('' '');
end.
```

这部分的代码也很容易写出。

```
case '\'': {
  buffer += Stream::getChar();
  while (1)
    if (Stream::nxtChar() == '\'')
      {
        buffer += Stream::getChar();
        if (Stream::nxtChar() == '\'')
          buffer += Stream::getChar();
        else
          return buffer;
      }
    else
      buffer += Stream::getChar();
}
```

3.1.5 其他实现细节

如果希望AC此题，选手应当注意到以上处理科学计数法的程序段无法处理定义数组时的区间，如[1..2]。这将要求输入流管理器的缓冲区大小扩张到2，并且对处理数的部分进行相应的修改。

4 进阶问题: 模拟源代码的问题

4.1 一个简易python解释器[2]

编写一个解释器。

支持两种语句：

1. 赋值语句：变量=表达式。变量只有整数型，且无需事先声明。表达式中只有以下要素：四则运算符（加、减；乘、整除），变量名，常量。没有括号。

2. 输出语句：print(变量名或常量[, 变量名或常量]*)

这里的常量皆为可以用16位带符号整数型存下的自然数。所有变量在被调用前至少被赋值过一次。输入数据符合Python3语法。

4.1.1 区分两种语句

输出语句是容易模拟的。用一个STL::map将变量名映射到值即可。

关键在于赋值语句，之后会介绍多种方法来处理赋值语句。

首先主程序的结构可以写成一个while循环，每个循环体处理一个语句。

从词法分析器读入一个语句的首词素。这个词素要么是print，要么是变量名。注意：通过这个词素并不能判断出语句的类型——在Python3中print并非是一个关键字，而是一个函数对象。对这个对象进行赋值是合法的——只不过在此之后就不能使用print函数进行输出了。因此正确的区分两个语句的做法是，在读入首词素后检查（这样就只需要大小为1的缓冲区了）下一词素。如果是等于号，则为赋值语句，是左小括号，则为输出语句。

现在我们已经能区分哪个是赋值语句了。现在有三种方法来处理赋值语句右端的表达式。这些方法分别列于下面。

现在本题还剩余的部分就是支持高精度了。不过那就与本文内容无关，也就不再赘述了。

4.1.2 第一种: 扫描，维护栈的经典表达式计算

具有四则运算的表达式计算是一个经典问题。

[2]水题(py), 某不知名NOI模拟赛

维护一个符号栈和一个值栈，从左到右扫描，遇到加减法压入两个栈，遇到乘除法直接应用到值栈的栈顶元素即可。

在这种做法下，如果要支持括号的话，就要为每个元素设置时间戳。遇到乘除法时并非直接应用，而是同样压入。只不过每轮循环体结束前检查：若符号栈头是乘除法，并且其时间比值栈头来得早的话，把符号栈头元素应用在值栈的最上方两个元素上。

在这种做法下，如果要支持更多优先级的运算，应当这么修改：每次遇到运算符的时候，检查栈顶，应用所有优先级更高的运算。将符号栈做成一个单调栈。

4.1.3 第二种: 一次性读入，按运算符优先级拆解表达式再计算

一次性读入所有词素直到遇到回车（这需要对词法分析器进行一些另外的修改），然后找出其中所有的最低优先级符号——加法和减法。用这些符号将表达式拆分成数个内含运算符优先级皆严格高于加法和减法的“项”（这里借用了“多项式”中的“项”这个术语，并没有什么特别的意思）。然后将每个“项”用次低优先级符号——乘法和除法拆分成多个“系数”即可。求值的时候先计算出所有“项”的值，再计算整个表达式的值即可。

在这种做法下，如果要支持括号的话，只需要在寻找目前最低优先级时忽略掉那些在括号中的符号，并把表达式计算写成一个函数，进行递归调用即可。

在这种做法下，如果要支持更多优先级的运算，简单地增加步骤数即可。

4.1.4 第三种: 扫描，伴随着递归下降的语法分析的表达式计算

这种方法和以上两种相比，在实现时，来得更复杂。虽然如此，之后的题目中并非只有表达式计算，将增加更多的要素。因此，掌握递归下降语法分析将是很重要的。

递归下降分析方法由若干个函数构成，每一个函数对应了描述此语言的上下文无关文法中的一个非终止符号。

当描述这一语言的上下文无关文法具备LL(1)性时，实现将会变得非常简单。

以本题为例。以下是本题中表达式满足的上下文无关文法。其中EXPR为初始符号。

```
EXPR ::= TERM
EXPR ::= EXPR + TERM
EXPR ::= EXPR - TERM
TERM ::= FACTOR
TERM ::= TERM * FACTOR
TERM ::= TERM / FACTOR
FACTOR ::= variable
FACTOR ::= integer
```

与其等价的一个LL(1)文法如下：

```
EXPR ::= TERM REXPR
REXPR ::= ε
REXPR ::= + TERM REXPR
REXPR ::= - TERM REXPR
TERM ::= FACTOR RTERM
RTERM ::= ε
RTERM ::= * FACTOR RTERM
RTERM ::= / FACTOR RTERM
FACTOR ::= variable
FACTOR ::= integer
```

LL(1)性说明，在这一文法的递归下降分析器中，对于每个非终结符号，都只需要从词法分析器中读取一个词素就可以判断它适用的变换规则。借助这个性质就可以写出这样的分析程序：

```
1 void EXPR();
2 void REXPR();
3 void TERM();
4 void RTERM();
5 void FACTOR();
6
```

```
7  void EXPR()
8  {
9    TERM(); REXPR();
10 }
11
12 void REXPR()
13 {
14   if (Lex::nxtLexeme() == "+")
15     {
16       AssertNext("+"); TERM(); REXPR();
17       return;
18     }
19   if (Lex::nxtLexeme() == "-")
20     {
21       AssertNext("-"); TERM(); REXPR();
22       return;
23     }
24   return;
25 }
26
27 void TERM()
28 {
29   FACTOR(); RTERM();
30 }
31
32 void RTERM()
33 {
34   if (Lex::nxtLexeme() == "*")
35     {
36       AssertNext("*"); FACTOR(); RTERM();
37       return;
38     }
39   if (Lex::nxtLexeme() == "/")
40     {
41       AssertNext("/"); FACTOR(); RTERM();
42       return;
43     }
44   return;
```

```
45 }
46
47 void FACTOR()
48 {
49   if (isDigit(Lex::nxtLexeme()[0]))
50     {
51       AssertInteger();
52       return;
53     }
54   if (isSHead(Lex::nxtLexeme()[0]))
55     {
56       AssertVariableName();
57       return;
58     }
59   throw "Syntax Error";
60 }
```

这些过程之间的调用结构就构成了该语句的具体语法树。

```
// 样例词素流
integer + integer * variable - integer

// 调用结构
EXPR(); // nxtLexeme is [integer]
  TERM();
    FACTOR();
      AssertInteger(); // nxtLexeme -> ["+"]
    RTERM();
  REXPR();
    AssertNext("+"); // nxtLexeme -> [integer]
    TERM();
      FACTOR();
        AssertInteger(); // nxtLexeme -> ["*"]
      RTERM();
        AssertNext("*"); // nxtLexeme -> [variable]
        FACTOR();
          AssertVariableName(); // nxtLexeme -> ["-"]
        RTERM();
    REXPR();
```

```
      AssertNext("-"); // nxtLexeme -> [integer]
      TERM();
        FACTOR();
          AssertInteger(); // nxtLexeme -> [EOF]
        RTERM();
      REXPR();

// 具体语法树
EXPR ::= TERM REXPR
  TERM ::= FACTOR RTERM
    FACTOR ::= integer
      [integer]
    RTERM ::= ε
  REXPR ::= + TERM REXPR
    ["+"]
    TERM ::= FACTOR RTERM
      FACTOR ::= integer
        [integer]
      RTERM ::= * FACTOR RTERM
        ["*"]
        FACTOR ::= variable
          [variable]
        RTERM ::= ε
    REXPR ::= - TERM REXPR
      ["-"]
      TERM ::= FACTOR RTERM
        FACTOR ::= integer
          [integer]
        RTERM ::= ε
      REXPR ::= ε
```

而有了语法树以后就可以利用它来进行计算了。以之前的分析程序衍生出的求值程序是这个样子的：

```
1 valtype EXPR();
2 valtype REXPR(valtype lhs);
3 valtype TERM();
4 valtype RTERM(valtype lhs);
```

```
valtype FACTOR();

valtype EXPR()
{
  return REXPR(TERM());
}

valtype REXPR(valtype lhs)
{
  if (Lex::nxtLexeme() == "+")
    {
      AssertNext("+");
      return REXPR(lhs + TERM());
    }
  if (Lex::nxtLexeme() == "-")
    {
      AssertNext("-");
      return REXPR(lhs - TERM());
    }
  return lhs;
}

valtype TERM()
{
  return RTERM(FACTOR());
}

valtype RTERM(valtype lhs)
{
  if (Lex::nxtLexeme() == "*")
    {
      AssertNext("*");
      return RTERM(lhs * FACTOR());
    }
  if (Lex::nxtLexeme() == "/")
    {
      AssertNext("/");
      return RTERM(lhs / FACTOR());
```

```
    }
  return lhs;
}

valtype FACTOR()
{
  if (isDigit(Lex::nxtLexeme()[0]))
    return GetIntegerVal();
  if (isSHead(Lex::nxtLexeme()[0]))
    return GetVariableVal();
  throw "Syntax Error";
}
```

问题到此解决。不过，如果对于向分析程序内部传入参数感到不适的话，可以活用while循环简化这个程序。

```
valtype EXPR();
valtype TERM();
valtype FACTOR();

valtype EXPR()
{
  valtype ret = TERM();
  do
    {
      if (Lex::nxtLexeme() == "+")
        {
          AssertNext("+");
          ret = ret + TERM();
          continue;
        }
      if (Lex::nxtLexeme() == "-")
        {
          AssertNext("-");
          ret = ret - TERM();
          continue;
        }
      break;
```

```
    }
  while (1);
  return ret;
}

valtype TERM()
{
  valtype ret = FACTOR();
  do
    {
      if (Lex::nxtLexeme() == "*")
        {
          AssertNext("*");
          ret = ret * FACTOR();
          continue;
        }
      if (Lex::nxtLexeme() == "/")
        {
          AssertNext("/");
          ret = ret / FACTOR();
          continue;
        }
      break;
    }
  while (1);
  return ret;
}

valtype FACTOR()
{
  if (isDigit(Lex::nxtLexeme()[0]))
    return GetIntegerVal();
  if (isSHead(Lex::nxtLexeme()[0]))
    return GetVariableVal();
  throw "Syntax Error";
}
```

这个程序实际上是基于以下文法的，它的表达方式是BNF的一个引入

了Kleene星号的扩展版本。

```
EXPR ::= TERM ((+ TERM) | (- TERM))*
TERM ::= FACTOR ((+ FACTOR) | (- FACTOR))*
FACTOR ::= integer | variable
```

现在我们得到了两个扫描并伴随着递归下降语法分析的解决本题表达式计算问题的程序。下面我们考虑它的扩展。

增加更多优先级运算是容易的，只需要如法炮制，增加几个分析过程就可以了。

增加对括号的支持也是容易的。这相当于在FACTOR的变换规则中加上一条FACTOR::=(EXPR)。对FACTOR()稍作修改即可。

如果常数中有负数，就需要增加对单目运算“负”的支持。在TERM和FACTOR之间加上一层分析过程就可以了。这层分析过程大概会是这样的：

```
valtype NEWLAYER()
{
  if (Lex::nxtLexeme() == "-")
    {
      AssertNext("-");
      return - FACTOR();
    }
  return FACTOR();
}
```

从而我们可以方便地将这一分析程序加入更多的特性。这将极大地有助于我们解决接下来的问题。

4.2 一个简易的C++混合实现系统[3]

你需要编写一个C++解释器。它需要支持以下特性：

int型变量，以及int数组（含高维数组）变量；int型变量上的若干运算；使用cin和cout进行int的输入和输出；全局变量和局部变量（不含动态分配内存）；使用int putchar(int)函数输出一个字符；以零或一或多个int为参数的，int型自定义函数（不含提前定义），直接递归；if语句，for循环和while前置的while循环。

这道题目的部分分基本上是按照编写解释器的步骤分步给出的。因此摘抄部分分分布如下：

第一个测试点是送分点。

第二个测试点是只含加减乘除和自然数常数的表达式计算。

第三个和第四个测试点是表达式计算。

第五个测试点中只有main函数，且不含流程控制语句。

第六个和第七个测试点中只有main函数。

4.2.1 测试点2

第二个测试点的做法参见前一节“简易python解释器”。

4.2.2 测试点3和4

这两个测试点就是在第二个测试点的分析程序上加入几级新的运算，加入括号，加入单目运算，以及一个有语义动作且返回自身的函数(即putchar函数)。

同样参见前一节。

4.2.3 测试点3和4, 准备转化到测试点5

在之前的程序中，我们抽取了cout和endl之间的内容并对其进行分析。

为了处理一个cout后跟多个表达式，或cin后跟多个地址的场合，我们应当对整个语句进行分析，这就是说，将输入输出运算符也作为运算而加入表达式处理模块。

[3]tsinsen P7003 / uoj 98. 未来程序・改, 集训队互测'15 第四场

这两个运算是具有动作的。在这里我们使用以下策略来实现这两个运算符：当运算符<<以后的那个更低级非终结符号对应的表达式被计算出来以后，就直接输出它。（因为在本题中可以确定此运算符前的表达式的结果是cout。）cin同理。

写成代码大概是这样的。

```
valtype NEWLAYER()
{
  valtype ret = NEXTLAYER();
  do
    {
      if (Lex::nxtLexeme() == "<<")
        {
          AssertNext("<<");
          putvalue(NEXTLAYER());
          continue;
        }
      if (Lex::nxtLexeme() == ">>")
        {
          AssertNext(">>");
          getvalue(NEXTLAYER());
          continue;
        }
      break;
    }
  while (1);
  return ret;
}
```

其中NEWLAYER处理输入输出运算以及比它优先级更低的运算（其实就是一切运算，NEWLAYER就是语句），NEXTLAYER处理低于输入输出运算，也就是通常意义上所说的表达式。putvalue和getvalue是valtype有关的两个函数，关于getvalue的实现将在之后引入变量概念时具体介绍。

4.2.4 测试点5, 新特性一: 变量

测试点5引入了变量。

在本题中，有以下对象存在：int型变量，int&型变量（赋值号左边的那个东西），cin，cout，endl。

我们先考虑如何处理前面那两个东西。我们开一个大数组来模拟内存，然后用一个$pair < bool, int >$（下简写为，PBI）来表示对象。当PBI的bool是真时，代表这个PBI表示的对象在内存中有确定的地址（这个变量在大数组中的位置保存在PBI的int中），是一个int&；否则它只是一个值，具体的值保存在PBI的int中。

接下来要加入cin，cout，endl这三个对象。只需要将某个PBI的bool设为真，然后在int中写一个不可能出现在地址中的整数，比如说，负数，就可以表示像cin这样的特殊对象了。

下面是putvalue和getvalue的一种实现。

```
const constENDL = make_pair(true, -1);

void putvalue(PBI x)
{
  if (x == constENDL)
    cout << endl;
  else
    if (x.first)
      cout << MEMORY[x.second];
    else
      cout << x.second;
}

void getvalue(PBI x)
{
  if (!x.first) throw "Syntax Error";
  if (x.second < 0) throw "Syntax Error";
  cin >> MEMORY[x.second];
}
```

当两个PBI参与除赋值和输入输出以外的运算时，建立一个新的PBI作为其结果。其bool为false，其int为运算结果。示例如下。

```
int to_val(PBI x)
```

```
{
  if ((x.first) && (x.second < 0))
    throw "Runtime Error";
  if (x.first)
    return MEMORY[x.second];
  else
    return x.second;
}

PBI operator + (PBI lhs, PBI rhs)
{
  return make_pair(false, to_val(lhs) + to_val(rhs));
}
```

当两个PBI参与赋值时则根据左手侧的PBI的int来把右侧的值放到内存数组中适当的位置。

4.2.5 测试点5, 新特性二: 数组

对于数组的处理有多种方式。有一种方法是用恰好等于数组中元素个数的连续内存来保存数组，每次用公式计算需要的元素在那个位置。但由于本题中的变量占用内存数量不多，所以还有一种空间利用率在50%以上，100%以下的实现方法，这种方法与已有的表达式计算模块更加相容。

对一个数组建立一棵树。每个i维数组的孩子就是它下面的那数个$i-1$维数组。这种实现就为这棵树上的所有节点都设置一个内存中的位置。在叶子节点保存这个变量的竖直，在非叶子节点保存它第一个子节点的在内存中的位置，并将每个非叶子节点的孩子放置在连续的内存位置中。数组A[3][2]的一种安排方式是这个样子的：

Address	*Content*	*Value*
x	A	$x+1$
$x+1$	$A[0]$	$x+4$
$x+2$	$A[1]$	$x+6$
$x+3$	$A[2]$	$x+8$
$x+4$	$A[0][0]$	$A[0][0]$
$x+5$	$A[0][1]$	$A[0][1]$
$x+6$	$A[1][0]$	$A[1][0]$
$x+7$	$A[1][1]$	$A[1][1]$
$x+8$	$A[2][0]$	$A[2][0]$
$x+9$	$A[2][1]$	$A[2][1]$

在这种安排下，容易将取数组元素”[]”写成一个PBI与int的运算。

4.2.6 测试点5, 新特性三: 内存管理

虽然测试点5中只有顺序结构，但是，并没有保证只存在int main后的那对花括号。

这就是说，需要考虑变量作用域的问题。

当一个函数中调用了某一变量时，通常使用以下原则确定具体是哪个变量：从这个变量调用语句所属的花括号向上，最低一层在该位置前包含声明语句的花括号中的那个声明语句所声明的变量就是要调用的变量。

在只有一个函数时，我们考虑在遇到声明语句时记录这个语句，在块尾撤销本块中所有的声明语句，那么执行调用语句的时候调用的恰好就是叫这个变量名的最后一个声明语句所声明的变量。

使用一个$map<string, stack<int>>$和一个$stack<stack<string>>$来处理。前者用来调用，后者用来撤销块中所有声明。

4.2.7 测试点5, 新特性四: 多个语句

用一个递归调用的分析过程来分析一个块。

每次通过首词素来判断语句类型。

如果首词素是右花括号说明块结束。

否则：

int指示一个声明语句；

左花括号指示一个块语句；

return指示一个返回值语句，当只有一个函数的时候这可以用分析程序中的一个exit(0)解决；

其他情况下交由之前已经增强过的表达式处理模块处理。

4.2.8 测试点6和7

这两个测试点中引入了循环语句。这可能会导致同一源代码段被反复处理。因此，每次对其进行扫描和语法分析，在效率上变得不佳。

为了解决这一问题，需要在第一遍扫描的时候为其显式地建出抽象语法树，然后之前的扫描过程变成扫描语法树上某一节点的孩子。

抽象语法树有很多实现方法。笔者认为它至少要有以下要素：

一个用来保存有关信息的*vector* < *string* >。一个变量声明语句的*vector* < *string* >中可能保存了这个变量的变量名，以及各维度的大小；代表函数的节点的*vector* < *string* >中可能含有了它的参数的名称；代表只含有加法和减法及其以下运算的子表达式的节点中可能保存了其第二个分量开始的各分量前是加号还是减号。

一个用来保存孩子的*vector* < *Tree*∗ >。根节点的孩子可能指向几个变量声明语句节点和几个函数节点。

在建出这棵语法树后，扫描执行就变成了语法树上的某一个节点。整个模拟过程就是执行根节点下的名称为main的function节点。

下面还是使用表达式计算的例子来说明大致的思想。

4.2.9 测试点6和7: 一个示例

这个示例是一个只包含加法、减法和乘法两级的表达式。

上下文无关文法：

```
EXPR = TERM ((+ TERM) | (- TERM))*
TERM = FACTOR (* FACTOR)*
FACTOR = integer
```

EXPR节点的配置是：*vector* < *string* >含有若干个"+"或"-"，指示了第二个TERM开始的符号，*vector* < *Tree** >含有若干个TERM节点。

TERM节点的配置是：*vector* < *string* >为空，*vector* < *Tree** >含有若干个FACTOR节点。

FACTOR节点的配置是：*vector* < *string* >含有一个元素，为integer的具体数值，*vector* < *Tree** >为空。

它的语法树节点定义部分是这样的：

```
struct Tree
{
  vector<string> VS;
  vector<Tree*> VTp;
  Tree()
  {
    VS.clear();
    VTp.clear();
  }
};
```

那么它对应的分析程序是这样的：

```
Tree* analyseEXPR();
Tree* analyseTERM();
Tree* analyseFACTOR();

Tree* analyseEXPR()
{
  Tree* ret = new Tree;
  ret->VTp.push_back(analyseTERM());
  while ((Lex::nxtLexeme() == "+") || (Lex::nxtLexeme()
       == "-"))
    {
      ret->VS.push_back(Lex::getLexeme());
      ret->VTp.push_back(analyseTERM());
    }
  return ret;
}
```

```
16
17 Tree* analyseTERM()
18 {
19   Tree* ret = new Tree;
20   ret->VTp.push_back(analyseFACTOR());
21   while (Lex::nxtLexeme() == "*")
22     {
23       Lex::getLexeme();
24       ret->VTp.push_back(analyseFACTOR());
25     }
26   return ret;
27 }
28
29 Tree* analyseFACTOR()
30 {
31   Tree* ret = new Tree;
32   ret->VS.push_back(Lex::getLexeme());
33   return ret;
34 }
```

它的解释程序也相应的使用数个函数。

```
1  int executeEXPR(Tree* cur);
2  int executeTERM(Tree* cur);
3  int executeFACTOR(Tree* cur);
4
5  int executeEXPR(Tree* cur)
6  {
7    int ret = executeTERM(cur->VTp[0]);
8    int SZ = cur->VS.size();
9    for (int i = 0; i < SZ; i++)
10     if (cur->VS[i] == "+")
11       ret = ret + executeTERM(cur->VTp[i + 1]);
12     else
13       ret = ret - executeTERM(cur->VTp[i + 1]);
14   return ret;
15 }
16
17 int executeTERM(Tree* cur)
```

```
18 {
19   int ret = 1;
20   for (vector<Tree*>::iterator it = cur->VTp.begin();
         it != cur->VTp.end(); it++)
21     ret = ret * executeFACTOR(*it);
22   return ret;
23 }
24
25 int executeFACTOR(Tree* cur)
26 {
27   int ret = 0, SZ = cur->VS[0].size();
28   for (int i = 0; i < SZ; i++)
29     ret = ret * 10 + cur->VS[0][i] - '0';
30   return ret;
31 }
```

读者可以参考这个示例写出适用于本题的更复杂的建立语法树和解释语法树过程。

4.2.10 测试点8, 9, 10

这三个测试点引入了函数。这要求分析程序的数个模块都要进行改进。

4.2.11 改进一: 表达式计算模块

本来，表达式计算模块中对putchar函数进行了特判。在引入了自定义函数以后，应当有可以处理所有函数的机制。

具体的来说是这样的：当表达式运算的最底层（即之前示例中的FACTOR层）遇到一个标识符时，判断它是不是一个函数。函数表可以通过预先检索语法树中根节点的儿子得到。如果不是函数，那么就是一个变量，向内存管理模块请求对应变量的值。如果是函数，检查是否是预置的putchar，如果是的话执行语义动作，否则，这里需要调用函数，从预先检索得到的$map < string, Tree* >$中检查这个函数对应的语法树子树，下面就是要执行这个子树。在这个时刻向内存管理模块把函数的参数当做变量进行声明和赋值。

4.2.12 改进二: 内存管理模块

4.2.6中的内存管理机制在这三个点将出现错误。

以以下程序为例。

```
int x;

int func()
{
  x;
}

int main()
{
  int x;
  func();
}
```

根据4.2.6的机制，在开始执行main之前，全局变量x被声明，入栈。开始执行main以后，局部变量x被声明，入栈。调用func以后，需要申请x的地址时返回的是x这个名字对应的栈顶，即main函数的局部变量x。这是不符合常理的，在函数func中的x应当指的是一层大括号以外的全局变量x。

于是要引入新数据结构$stack < multiset < string >>$。这个数据结构用来保存每层函数中的变量声明情况。在询问变量地址，首先检查该层函数中是否声明过此变量标识符，若是，返回$map < string, stack < int >>$的栈顶，否则返回那个全局变量的地址，当然这个地址就在$map < string, stack < int >>$的栈底。

4.2.13 一个实现细节: return

return关键字将中断一个函数的调用。

而在执行语法树的过程中中断一个子程序并不等于中断了一个函数调用。这带来一个如何处理return语句的问题。

在之前的测试点中，由于只有一个函数main，当遇到return的时候可以直接中断整个程序（如使用exit(0)）。

我们将一个return的执行过程划分成几个阶段。

执行前阶段：分析器遇到了return关键字，正在计算后面的表达式。

执行时阶段：分析器进行一系列回退，直到回到这个函数的调用语句。

执行后阶段：进行退出一层函数后的打扫工作。

我们注意到，虽然可以有很多个函数中的return语句同时处于执行前阶段，但每时每刻处于执行时阶段的return语句只有一个！这就是说，可以使用全局变量来管理return语句。实现略。

5 变种问题: 结合了数据结构的问题

在4.1所在的那场模拟赛中，4.1被称为“编程难度比较大”的“超级细节题”，需要“熟练的选手的1个小时到1.5个小时”或是“薄弱的选手的3个小时”来完成。而如同4.2一样的强模拟题，即使是出题人在阅读过有关材料（见参考文献）之后，也花了6个小时来写出其长度超千行的标程，也就难怪4.2被用来测试12名候选队员时，最高分只有40分，也就是“复杂表达式计算”的程度了。

第五章的题目在代码强度上远低于4.2（代码长度上是四分之一强的程度），略低于4.1（注意如果要AC4.1还需要一个高精度实现），这一强度是可以接受的，是适合被放置在一场五个小时的OI/ACM比赛中的。

在5.1中将讲解一个基础向问题。它最初出现在一场ACM比赛中，是该场比赛中最难的问题，在逾400个参赛队伍/选手中只有1个队伍/选手解出此题。5.2中的几个问题则是将5.1与一些低阶的高级数据结构相结合得到的产物。

5.1 Unreachable Statement[4]

找出一段Cb语言源代码中所有不可能被执行到的语句。

Cb语言是C语言的一个子集。

一个Cb程序由若干个function组成。

每个function的格式是function 函数名() {语句*}

语句有三种，“print 变量”，“if (表达式) 语句[else 语句]”和”{语句*}”

print没有实际意义。

表达式只有三种，true，false和“变量名二元算符非负常量”。二元算符包括四种不等号。

所有变量为int类型。没有变量声明。

5.1.1 分析程序框架

分析这个问题的分析程序可以被分成以下模块。

1. 输入流管理器、词法分析器（见3.1）。
2. 语法分析器（见4.2）。
3. 可行性判定器。

[4]ZOJ 3786. Unreachable Statement, 第十一届浙江省大学生程序设计竞赛

可行性判定器是本章新引入的一个构件，5.2中的与数据结构进行的结合就发生在可行性判定器。

下面将分模块讨论针对这道题的实现措施。

5.1.2 输入流管理器的修改

在本题中，需要输出具体在哪些坐标，代码从可执行到变成了不可执行到。对于具体坐标的要求使得有必要在输入流管理器中加入一个当前坐标信息。这个信息在每次调用Stream::getChar()函数时更新。

5.1.3 词法分析器的修改

一个根据本题中可能出现符号而设计的普通的词法分析器就可以达到本题的要求。但是同样是由于上一小节中提到的原因，需要记录一个缓冲区词素坐标位置信息。这个信息在每次调用Lex::nxtLexeme()时，在跳过所有空白字符后更新。

5.1.4 语法分析器的修改

本题中不存在循环语句。从而一个初级的，如同4.1.4的递归下降语法分析器就可以胜任这份工作。

在扫描到一个条件分歧语句，并准备进入其一个分支时，语法分析器就需要收集有关表达式的信息，将这一信息传入可行性判定器，若可行性判定器在这一信息传入前后返回的结果从“可行”变成“不可行”，则语法分析器需要从词法分析器中取得坐标信息并把它输出来。在这一个分支结束之后语法分析器还应当告诉判定器删除这一信息。

由于复杂非终止符号的数目比较少，因此这个语法分析器可以用非递归的形式实现。在5.2.1和5.2.2中要求使用非递归形式的语法分析器，具体实现留给读者。

注意到有if就存在else匹配问题。一个这样的上下文无关文法难以被改造成LL(1)形式。然而根据常理，一个else和它之前最近一个尚未匹配的if匹配。因此在IF分析程序中只需要贪心往前取else即可，并不一定要把整个文法改造成LL(1)文法。

5.1.5 可行性判定器

可行性判定器是一个数据结构。它处理一系列命题，并实时地返回这一系列命题是否矛盾。这些命题对应了源代码中套着当前位置的那些条件分歧语句。当可行性判定器返回矛盾时，就代表语法分析器当前扫描到的位置是不可访问到的。

可行性判定器需要支持以下操作：

1. 增加一个命题。

2. 删除最后一个增加的尚未删除的命题。

3. 返回是否矛盾。

在本题中，这个可行性判定器包含一个$map < string, stack < pair < int, int >>>$和一个$int$。前者保存某个变量当前的取值范围的上界和下界，后者保存发现的冲突数，后者的值等于$false$的数量加上前者中那些上界小于下界的变量的个数。使用$stack$是为了便于撤销最后一个命题。当判定器中的int为0时，说明没有冲突，否则说明发现了冲突。

5.2 变体

从5.1.1中可以看到，可行性判定器是一个独立的模块。通过更换这个模块，可以实现一些其他功能。

5.2.1 运用线段树的变体[5]

在5.1的基础上：

if的表达式的格式变更为“a[下标] >= 值”，这里a是一个长为N的int数组，其每个元素被要求大于等于0，并且有公理：如果a[i..j]皆非0，则a[i..j]的元素和小于等于给定的常数K。

在本题中，可行性判定器的工作就是在判定取值范围冲突以外判断一个数列的最大非零子段和大小。这里只需要加上一棵线段树即可。

[5]HDU 4874. ZCC Loves RPG, 多校'14 第二场

5.2.2 运用并查集的变体[6]

在5.1的基础上：

变量的类型从int替换成bool。if的表达式变更为两个变量是否相等。

在本题中，可行性判定器是一个并查集。由于其撤销操作只针对最后的操作，所以只需要用一个栈记录每次连了哪条边即可。使用按秩合并来保证复杂度。

5.2.3 运用凸包的变体

在5.1的基础上：

变量的类型从int替换成浮点数且只有两个（x和y）。if的表达式变更为“ax+by+c >= 0”。并且有公理$|x|, |y| \leq 100$。

在本题中，每时每刻的(x,y)范围在一个凸多边形中。用两棵伸展树来保存这个凸包。每个命题代表的直线将一个凸包分成两个，讨论一下与两个凸壳的相交的各种情形即可。

5.3 小结

本类问题的解答程序，就是在固定的输入流管理器+词法分析器+语法分析器的基础上，添加不同的可行性判定器模块。

这类问题自带的“撤销最后一个未被撤销的操作”的性质，使得一些难以可持久化的问题在这一性质之下变得容易实现——同时还没有现成的算法。

从而，这类问题不仅在设计试题上，还在深入分析数据结构类问题的方面上有其特别的意义。

还有哪些数据结构问题是难以可持久化，但是容易进行回退操作的？直到现在笔者只想到了5.2.2和5.2.3两个例子。这里权当抛砖引玉，将这个问题留给我们的读者思考了。

[6]ZCC Loves AVG, 校内模拟赛

6 感谢

感谢CCF提供这样一个学习和交流的机会；

感谢教练李建老师的指导；

感谢在本文写作过程中给予我帮助的同学们；

感谢浙江省大学生程序设计竞赛的出题人员，他们的工作让我发现了这个对我而言全新的领域。

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman “Compilers: Principles, Techniques & Tools (2nd ed.)” （《编译原理(第2版)》，机械工业出版社）

[2] Robert W. Sebesta “Concepts of Programming Language (10th ed.)” （《编程语言原理(第10版)》，清华大学出版社）

集合幂级数的性质与应用及其快速算法

武汉市第二中学 吕凯风

摘 要

为了研究以集合为状态的递推，本文提出了集合幂级数，并分析了集合幂级数的性质，总结归纳了三种常见的乘法及相关快速算法。最后本文给出了几个利用集合幂级数加速递推的实际例子。

1 引言

在信息学竞赛中，经常会遇到一些有关集合的动态规划问题。比如是某种关于集合的计数问题，通常你会设一个 f_S 表示对于集合 S 的某某方案数，然后写一个递推式。等你写完了递推式，看了看题目规定的时间限制——这个算法太慢了。这个时候，怎么办？你想要优化递推式，但是手上却没有称手的工具，那你或许就只能选择放弃了。

回忆我们较为成熟的数列理论，我们可以把一个数列 $f_0, f_1, \ldots$ 和一个形式幂级数 $f(x) = \sum_{k\geq 0} f_k x^k$ 相对应，并称之为该数列的生成函数。由于左有数学上对形式幂级数的缜密分析，右有计算上对形式幂级数的快速算法，所以生成函数一马平川，解决了一个又一个数列的计算问题。

不禁想问，我们是否能把 f_S 这样的下标为集合的数列也跟一个幂级数挂钩并作为其生成函数，从而产生优美而高效的算法呢？本文作者做了一番研究，得到了这篇论文要展示的内容。

在本文第 2 节，本文作者给出了集合幂级数的定义以及加法运算，讨论了应该如何定义乘法。

接下来第 3, 4, 5 节兵分三路，从三种途径定义乘法并介绍相关快速算法，进一步分析性质。

在第 6 节，本文作者给出了几个集合幂级数的实际应用。

为了方便，我们规定如下几个本文中要用到的记号。

记号 1.1. 我们设全集为有限集 $U = \{1, 2, \ldots, n\}$，其中 $n = |U|$。本文中我们假设所有我们关心的 f_S 中的集合 S 都是 U 的子集。

记号 1.2. 设 X 是一个集合，用 2^X 表示 X 的幂集，即 X 的所有子集组成的集合。

记号 1.3. 用 $[P]$ 表示表达式 P 成立时为 1 否则为 0。例如 $[x = 3]$ 表示 $x = 3$ 时值为 1 否则为 0。

2 定义

现在我们引入集合幂级数。

定义 2.1. 设 F 是一个域，则称函数 $f : 2^U \to F$ 为 F 上的一个**集合幂级数**。对每个 $S \in 2^U$，记 f_S 表示把 S 代入函数 f 后的函数值，并称 f_S 为该集合幂级数的第 S 项的**系数**。显然如果确定了每一个系数的值，那么 f 也就确定了。

记号 2.1. 常见的域有 $\mathbb{C}, \mathbb{R}, \mathbb{Q}, \mathbb{F}_p$ 等等，域具有基本相同的性质。本文中设我们要研究的是 F 上的集合幂级数，而不是一个个具体分析每个域上的集合幂级数的情况。所以本文中不明确指出时，我们都用 F 表示要研究的集合幂级数的系数所属的域。

集合幂级数之间的加法运算是容易定义的。

定义 2.2. 设 f, g 为集合幂级数，定义 $f + g = h$，其中 h 是一个集合幂级数，且 h_S 满足：

$$h_S = f_S + g_S \tag{1}$$

即对应系数相加。

减法只要把 $f_S + g_S$ 改为 $f_S - g_S$ 就行了。显见，我们可以在 $O(2^n)$ 时间内求两个集合幂级数的加减法。

容易看出集合幂级数形成了一个加法阿贝尔群，其中零元为系数均为 0 的集合幂级数。

记号 2.2. 对于任意 $c \in F$，$S \in 2^U$，我们用符号 $f = cx^S$ 来表示一个第 S 项系数为 c，其余项均为 0 的集合幂级数 f。这里的 cx^S 是一个整体的符号，并不是某种乘法。按照习惯，对于 $1x^S$ 我们可以省略为 x^S。

容易看出一个集合幂级数一定可以写成若干个 cx^S 相加的形式。于是我们可以用符号：

$$f = \sum_{S \subseteq 2^U} f_S x^S \tag{2}$$

来表示一个集合幂级数。

例如取 $U = \{1, 2\}$ 以及 $F = \mathbb{R}$，我们用 $f(x) = 5x^{\varnothing} + 9x^{\{1\}} + 3x^{\{1,2\}}$ 可以表示一个 $f_{\varnothing} = 5, f_{\{1\}} = 9, f_{\{2\}} = 0, f_{\{1,2\}} = 3$ 的集合幂级数。

那么现在问题来了，乘法应该如何定义？

记乘法为 $h = f \cdot g$，显然我们希望乘法对加法有分配律，那么：

$$\sum_{S \in 2^U} h_S x^S = \left(\sum_{L \in 2^U} f_L x^L\right) \cdot \left(\sum_{R \in 2^U} g_R x^R\right) = \sum_{L \in 2^U} \sum_{R \in 2^U} \left(f_L x^L\right) \cdot \left(g_R x^R\right) \tag{3}$$

自然我们希望 $\left(f_L x^L\right) \cdot \left(g_R x^R\right)$ 是以某种集合运算乘起来的，我们设一个 2^U 里的二元运算 $*$，满足：

- $\forall L, R \in 2^U$：$L * R = R * L$（交换律）
- $\forall L, M, R \in 2^U$：$(L * M) * R = L * (M * R)$（结合律）
- $\forall S \in 2^U$：$S * \varnothing = S$（单位元为 $\varnothing$）

那么我们就可以定义 $\left(f_L x^L\right) \cdot \left(g_R x^R\right) = (f_L g_R) x^{L*R}$，易证这样定义集合幂级数的乘法满足交换律、结合律、对加法的分配律。我们可以把所有 $c \in F$ 的元素看作 $cx^{\varnothing}$，这样能使得对于任意集合 $S \in 2^U$ 有 $cx^S = c \cdot x^S$。于是集合幂级数形成了一个交换环，并且包含了整个 F 作为子环。

把 $*$ 取成不同的运算可以得到不同的效果。下面三节我们将分别看到由不同的 $*$ 定义产生的集合并卷积、集合对称差卷积、子集卷积三种乘法。

3 集合并卷积

我们取 $L * R = L \cup R$ 就可以得到集合并卷积。

定义 3.1. 我们定义两个集合幂级数的乘积为**集合并卷积**。设 $f = \sum_{S \subseteq 2^U} f_S x^S$，$g = \sum_{S \subseteq 2^U} g_S x^S$，定义 $f \cdot g = h$，其中 h 是一个集合幂级数，且 h_S 满足：

$$h_S = \sum_{L \subseteq 2^U} \sum_{R \subseteq 2^U} [L \cup R = S] f_L g_R \tag{4}$$

为了方便，$f \cdot g$ 可以简记为 fg。

3.1 快速算法

给出两个集合幂级数，暴力枚举 L 和 R 把 $f_L g_R$ 加到 $h_{L\cup R}$ 上去，这样进行集合并卷积的计算的时间复杂度是 $O(4^n)$，通常是无法接受的。

我们需要更快的算法。

3.1.1 分治乘法

我们考虑 U 中的元素 n。我们可以把含 n 的集合单独提一个 $\{n\}$ 出来，那么就可以写成 $f = f^- + x^{\{n\}} f^+, g(x) = g^- + x^{\{n\}} g^+$ 的形式。于是：

$$\begin{aligned} fg &= \left(f^- + x^{\{n\}} f^+\right)\left(g^- + x^{\{n\}} g^+\right) && (5)\\ &= f^- g^- + x^{\{n\}}(f^- g^+ + f^+ g^- + f^+ g^+) && (6)\\ &= f^- g^- + x^{\{n\}}((f^- + f^+)(g^- + g^+) - f^- g^-) && (7) \end{aligned}$$

所以我们把问题转化成了求 $f^- g^-$ 和 $(f^- + f^+)(g^- + g^+)$，而这里的集合幂级数的集合中都不含 n 这个元素了，所以可以令 $U = \{1, 2, \ldots, n-1\}$ 然后递归进行乘法即可。

设时间复杂度为 $T(n)$ 那么 $T(n) = 2T(n-1) + O(2^n)$，解得 $T(n) = O(n2^n)$。

3.1.2 快速莫比乌斯变换

我们来看一个更加强大的计算乘法的算法。

定义 3.2. 对于一个集合幂级数 f 我们定义 f 的**莫比乌斯变换**[1]为集合幂级数 $\hat{f}$，其中：

$$\hat{f}_S = \sum_{T\subseteq S} f_T \tag{8}$$

反过来，我们定义 $\hat{f}$ 的**莫比乌斯反演**[2]为 f。

由容斥原理，我们可以得到：

$$f_S = \sum_{T\subseteq S} (-1)^{|S|-|T|} \hat{f}_T \tag{9}$$

[1]Möbius Transform
[2]Möbius Inversion

所以对于任意一个集合幂级数，莫比乌斯反演是存在且唯一的。

我们对 (4) 等式两端同时做莫比乌斯变换，那么就可以得到：

$$\hat{h}_S = \sum_{L\subseteq 2^U}\sum_{R\subseteq 2^U}[L\cup R\subseteq S]f_L g_R \tag{10}$$

我们知道，由于 $[L\cup R\subseteq S]=[L\subseteq S][R\subseteq S]$ 所以：

$$\begin{aligned}\hat{h}_S &= \sum_{L\subseteq S}\sum_{R\subseteq S} f_L g_R &(11)\\ &= \left(\sum_{L\subseteq S} f_L\right)\left(\sum_{R\subseteq S} g_R\right) &(12)\\ &= \hat{f}_S\hat{g}_S &(13)\end{aligned}$$

所以，如果我们想求 $h=fg$，我们可以先求出 $\hat{f}$ 和 $\hat{g}$，对应系数乘起来得到 $\hat{h}$，再通过 $\hat{h}$ 求 h。那么现在关键问题在于，怎样快速求出一个集合幂级数的莫比乌斯变换以及莫比乌斯反演。

我们可以使用递推。设 $\hat{f}_S^{(i)}$ 为 $S\setminus T\subseteq\{1,\dots,i\}$ 的 S 的子集 T 的系数之和。令 $\hat{f}_S^{(0)}=f_S$，容易得到对于每个不包含 i 的 S 有 $\hat{f}_S^{(i)}=\hat{f}_S^{(i-1)}$，$\hat{f}_{S\cup\{i\}}^{(i)}=\hat{f}_{S\cup\{i\}}^{(i-1)}+\hat{f}_S^{(i-1)}$。

我们可以用同样的方法计算莫比乌斯反演。两个算法的时间复杂度都是 $O(n2^n)$。下面给出了代码示例：

Algorithm 1 快速莫比乌斯变换

输入： 集合幂级数 f

输出： f 的莫比乌斯变换

```
for i ← 1 to n do
    for all S ∈ 2^{U\{i}} do
        f_{S∪{i}} ← f_{S∪{i}} + f_S
    end for
end for
return f
```

Algorithm 2 快速莫比乌斯反演

输入： 集合幂级数 f

输出： f 的莫比乌斯反演

```
for i ← 1 to n do
    for all S ∈ 2^{U\{i}} do
        f_{S∪{i}} ← f_{S∪{i}} − f_S
    end for
end for
return f
```

这样，我们就能在 $O(n2^n)$ 时间内计算两个集合幂级数的乘法。

3.1.3 比较

我们比较一下这两种算乘法的算法。两个算法时间复杂度相同。仔细观察发现分治乘法其实是把莫比乌斯变换与莫比乌斯反演一起进行了——递归下去的时候是在做莫比乌斯变换，回溯上来的时候是在做莫比乌斯反演。这两个算法的本质是一样的。

值得一提的是，这两种算法都能拓展到任意交换环 R 上计算 R 上的集合幂级数的乘法。

3.2 由变换引出的性质

当我们把乘法拆成做莫比乌斯变换，对应系数相乘，做莫比乌斯反演这三个过程的时候，如果我们要把三个集合幂级数乘起来，只要分别求莫比乌斯变换，把对应系数乘起来之后再做莫比乌斯反演就行了，而不用分别做两次乘法。

这个结论启发我们把某些对集合幂级数的操作转化为对莫比乌斯变换的操作。由于 $f+g$ 的莫比乌斯变换的系数等于 $\hat{f}$ 和 $\hat{g}$ 的对应系数相加，fg 的莫比乌斯变换的系数等于 $\hat{f}$ 和 $\hat{g}$ 的对应系数相乘，所以对于一个集合幂级数 f，如果有一个多项式 $T(x)$，我们想求 $T(f)$，那么我们只用算出 $\hat{f}$，然后对每个系数 $\hat{f}_S$ 求出 $T(\hat{f}_S)$，这样就得到了 $T(f)$ 的莫比乌斯变换，对其做莫比乌斯反演就能得到 $T(f)$。

很自然地，我们会想知道能否做除法。作为乘法的逆运算，容易看出做除法就是对莫比乌斯变换的系数做除法。所以，一个集合幂级数有唯一的乘法逆元当且仅当莫比乌斯变换的所有系数都不为 0。

上述内容看起来似曾相识。我们常用快速离散傅里叶变换在 $O(n\log n)$ 的时间内计算两个多项式的乘法，而这个算法既可以像上文那样抽象地推导出来，也可以形象地理解成把所有 n 次单位根代入多项式求值，把函数值乘起来，再用插值求出多项式的系数。

我们可以尝试得出一个快速莫比乌斯变换的另一个版本的理解。

定义 3.3. 对于一个 n 维向量 $\mathbf{x}\in F^n$ 和一个集合 $S\in 2^U$，我们定义：

$$\mathbf{x}^S=\prod_{i\in S}\mathbf{x}_i \tag{14}$$

定义 3.4. 对于一个 n 维向量 $\mathbf{x}\in F^n$ 和一个集合幂级数 f，我们定义：

$$f(\mathbf{x})=\sum_{S\subseteq 2^U} f_S\cdot\mathbf{x}^S \tag{15}$$

可以看出 $f(\mathbf{x})$ 是一个关于 $\mathbf{x}_1,\ldots,\mathbf{x}_n$ 的 n 元多项式函数。

定义 3.5. 对于一个集合 $S\in 2^U$，定义 S 的**特征向量**为 $\mathbf{S}\in F^n$，其中：

$$\mathbf{S}_i=[i\in S] \tag{16}$$

那么我们可以发现，$\hat{f_S}=f(\mathbf{S})$，所以莫比乌斯变换就是把所有集合的特征向量代入集合幂级数求值，莫比乌斯反演就是插值。注意到 0 和 1 这两个数的任意正整数次幂都等于它本身，所以对于一个 01 向量 $\mathbf{x}$ 必有 $\mathbf{x}^L\cdot\mathbf{x}^R=\mathbf{x}^{L\cup R}$。所以对于 $h=fg$ 我们就自然而然地有 $h(\mathbf{S})=f(\mathbf{S})g(\mathbf{S})$。这样我们就从另一个视角解释了这个计算乘法的算法。

4 集合对称差卷积

接下来我们换一种方式定义乘法。

定义 4.1. 定义两个集合 A,B 的**对称差**为：

$$A\oplus B=\{x\mid (x\in A)\oplus(x\in B)\} \tag{17}$$

其中右侧的 $\oplus$ 为逻辑异或，即两个表达式真假性不同时为真，否则为假。

我们取 $L * R = L \oplus R$ 就可以得到集合对称差卷积。

定义 4.2. 我们定义两个集合幂级数的乘积为**集合对称差卷积**。设 $f = \sum_{S \subseteq 2^U} f_S x^S$，$g = \sum_{S \subseteq 2^U} g_S x^S$，定义 $f \cdot g = h$，其中 h 是一个集合幂级数，且 h_S 满足：

$$h_S = \sum_{L \subseteq 2^U} \sum_{R \subseteq 2^U} [L \oplus R = S] f_L g_R \tag{18}$$

为了方便，$f \cdot g$ 可以简记为 fg。

4.1 快速算法

给出两个集合幂级数，暴力枚举 L 和 R 把 $f_L g_R$ 加到 $h_{L \oplus R}$ 上去，这样进行集合对称差卷积的计算的时间复杂度是 $O(4^n)$，通常是无法接受的。

我们需要更快的算法。

4.1.1 分治乘法

跟集合并卷积的时候一样，我们还是可以使用分治乘法。我们考虑 U 中的元素 n。我们可以把含 n 的集合单独提一个 $\{n\}$ 出来，那么就可以写成 $f = f^- + x^{\{n\}} f^+, g(x) = g^- + x^{\{n\}} g^+$ 的形式。于是：

$$\begin{aligned} fg &= \left(f^- + x^{\{n\}} f^+\right)\left(g^- + x^{\{n\}} g^+\right) && (19) \\ &= (f^- g^- + f^+ g^+) + x^{\{n\}}(f^- g^+ + f^+ g^-) && (20) \end{aligned}$$

我们可以转而求 $(f^- + f^+)(g^- + g^+)$ 和 $(f^- - f^+)(g^- - g^+)$，然后两式相加除以 2 得到 $f^- g^- + f^+ g^+$，两式相减除以 2 得到 $f^- g^+ + f^+ g^-$。我们可以令 $U = \{1, 2, \ldots, n-1\}$ 然后递归进行乘法即可。

由于涉及除以 2，该算法不能对特征为 2 的 F 使用。

设时间复杂度为 $T(n)$ 那么 $T(n) = 2T(n-1) + O(2^n)$，解得 $T(n) = O(n2^n)$。

4.1.2 快速沃尔什变换

我们来看一个更加强大的计算乘法的算法——快速沃尔什变换。快速沃尔什变换其实也就是每一维大小为 2 的高维快速傅里叶变换。

原算法是用矩阵的语言描述的，而下面我们用集合的语言来描述。

首先注意到对于一个集合 S 有：

$$\frac{1}{2^n}\sum_{T\subseteq 2^U}(-1)^{|S\cap T|}=[S=\varnothing] \tag{21}$$

这只用注意到，当 S 为空集时，$(-1)^{|S\cap T|}$ 恒为 1。当 S 不为空时，设 $v\in S$，那么考虑一个 $T\in 2^U$，由于 $(-1)^{|S\cap(T\oplus\{v\})|}=(-1)^{|(S\cap T)\oplus\{v\}|}=-(-1)^{|S\cap T|}$，所以 $(-1)^{|S\cap T|}+(-1)^{|S\cap(T\oplus\{v\})|}=0$。由于 $T\oplus\{v\}\oplus\{v\}=T$，所以我们可以把每个 T 跟 $T\oplus\{v\}$ 配成一对，所以当 S 不为空时，原式左边为 0。

利用这个性质，我们可以化简 (18)：

$$\begin{aligned}
h_S &= \sum_{L\subseteq 2^U}\sum_{R\subseteq 2^U}[L\oplus R\oplus S=\varnothing]f_Lg_R &(22)\\
&= \sum_{L\subseteq 2^U}\sum_{R\subseteq 2^U}\frac{1}{2^n}\sum_{T\subseteq 2^U}(-1)^{|T\cap(L\oplus R\oplus S)|}f_Lg_R &(23)\\
&= \sum_{L\subseteq 2^U}\sum_{R\subseteq 2^U}\frac{1}{2^n}\sum_{T\subseteq 2^U}(-1)^{|T\cap L|}(-1)^{|T\cap R|}(-1)^{|T\cap S|}f_Lg_R &(24)\\
&= \frac{1}{2^n}\sum_{T\subseteq 2^U}(-1)^{|T\cap S|}\left(\sum_{L\subseteq 2^U}(-1)^{|T\cap L|}f_L\right)\left(\sum_{R\subseteq 2^U}(-1)^{|T\cap R|}g_R\right) &(25)
\end{aligned}$$

这就启发我们定义沃尔什变换：

定义 4.3. 对于一个集合幂级数 f 我们定义 f 的**沃尔什变换**[1]为集合幂级数 $\hat{f}$，其中：

$$\hat{f}_S=\sum_{T\subseteq 2^U}f_T(-1)^{|S\cap T|} \tag{26}$$

反过来，我们定义 $\hat{f}$ 的**沃尔什逆变换**[2]为 f。

容易用和上面类似的推理发现沃尔什逆变换是唯一的（事实上只用取 $g=x^{\varnothing}$ 即可），且为：

$$f_S=\frac{1}{2^n}\sum_{T\subseteq 2^U}\hat{f}_T(-1)^{|S\cap T|} \tag{27}$$

我们再看集合对称差卷积，就有：

$$\hat{h}_S=\hat{f}_S\hat{g}_S \tag{28}$$

[1]Walsh Transform

[2]Inverse Walsh Transform

于是如果想求 $h = fg$，我们可以先对 f, g 分别做沃尔什变换，对应系数乘起来得到 $\hat{h}$，再做沃尔什逆变换。所以现在问题变成了，怎样快速求一个集合幂级数的沃尔什变换以及沃尔什逆变换。

我们可以使用递推。设 $\hat{f}_S^{(i)}$ 为只考虑那些与 S 的对称差是 $\{1, \ldots, i\}$ 的子集的集合时的沃尔什变换第 S 项，即那些 $S \oplus T \subseteq \{1, \ldots, i\}$ 的集合 T 的 $f_T(-1)^{|S \cap T|}$ 之和。令 $\hat{f}_S^{(0)} = f_S$，容易得到对于每个不包含 i 的 S 有 $\hat{f}_S^{(i)} = \hat{f}_S^{(i-1)} + \hat{f}_{S \cup \{i\}}^{(i-1)}$ 且 $\hat{f}_{S \cup \{i\}}^{(i)} = \hat{f}_S^{(i-1)} - \hat{f}_{S \cup \{i\}}^{(i-1)}$。

我们可以做沃尔什变换后乘以 $\frac{1}{2^n}$ 得到沃尔什逆变换。于是做沃尔什变换和沃尔什逆变换的时间复杂度都是 $O(n2^n)$。下面给出了代码示例：

Algorithm 3 快速沃尔什变换

输入： 集合幂级数 f

输出： f 的沃尔什变换

```
for i ← 1 to n do
    for all S ∈ 2^{U\{i}} do
        l ← f_S, r ← f_{S∪{i}}
        f_S = l + r, f_{S∪{i}} = l − r
    end for
end for
return f
```

这样，我们就能在 $O(n2^n)$ 时间内计算两个集合幂级数的乘法。

不过遗憾的是，由于涉及乘以 $\frac{1}{2^n}$，该算法不能对特征为 2 的 F 使用。

4.1.3 比较

我们比较一下这两种算乘法的算法。仔细观察发现分治乘法其实是把沃尔什变换与沃尔什逆变换一起进行了——递归下去的时候是在做沃尔什变换，回溯上来的时候是在做沃尔什逆变换。这两个算法的本质是一样的。

然而两个算法都无法对特征为 2 的 F 使用，这是一个小小的遗憾。实际上两个算法都能对任意 2 有乘法逆元的交换环上的集合幂级数使用。

4.2 由变换引出的性质

跟集合并卷积一样，当我们把乘法拆成做沃尔什变换，对应系数相乘，做沃尔什逆变换这三个过程的时候，如果我们要把三个集合幂级数乘起来，只要分别求沃尔什变换，把对应系数乘起来之后再做沃尔什逆变换就行了，而不用分别做两次乘法。

这个结论又启发我们把某些对集合幂级数的操作转化为对沃尔什变换的操作。由于 $f+g$ 的沃尔什的系数等于 $\hat{f}$ 和 $\hat{g}$ 的系数相加，fg 的沃尔什变换的系数等于 $\hat{f}$ 和 $\hat{g}$ 的系数相乘，所以对于一个集合幂级数 f，如果有一个多项式 $T(x)$，我们想求 $T(f)$，那么我们只用算出 $\hat{f}$，然后对每个系数 $\hat{f}_S$ 求出 $T(\hat{f}_S)$ 得到 $T(f)$ 的沃尔什变换，对其做沃尔什逆变换就能得到 $T(f)$。

很自然地，我们又会想知道能否做除法。作为乘法的逆运算，容易看出做除法就是对沃尔什变换的系数做除法。所以，一个集合幂级数有唯一的乘法逆元当且仅当沃尔什变换的所有系数都不为 0。

我们可以尝试得出一个快速沃尔什变换的另一个版本的理解。

我们沿用集合并卷积一节中 $\mathbf{x}^S$ 和 $f(\mathbf{x})$ 的定义，重新定义集合的特征向量。

定义 4.4. 对于一个集合 $S \in 2^U$，定义 S 的**特征向量**为 $\mathbf{S} \in F^n$，其中：

$$\mathbf{S}_i = (-1)^{[i \in S]} \tag{29}$$

那么我们可以发现，$\hat{f}_S = f(\mathbf{S})$，所以沃尔什变换就是把所有集合的特征向量代入集合幂级数求值，沃尔什逆变换就是插值。注意到 ± 1 的平方都等于 1，所以对于一个每一维均为 ± 1 的向量 $\mathbf{x}$ 必有 $\mathbf{x}^L \cdot \mathbf{x}^R = \mathbf{x}^{L \oplus R}$。所以对于 $h = fg$ 我们就自然而然地有 $h(\mathbf{S}) = f(\mathbf{S})g(\mathbf{S})$。这样我们就从另一个视角解释了这个计算乘法的算法。

5 子集卷积

接下来我们再换一种方式定义乘法。这次我们定义 $*$ 为不相交集合并，即如果 $A \cap B \neq \varnothing$ 那么 $A * B$ 未定义（严谨地说，$x^A \cdot x^B = 0$），否则为 $A \cup B$。下面是正式的定义。

定义 5.1. 我们定义两个集合幂级数的乘积为**子集卷积**[1]。设 $f = \sum_{S \subseteq 2^U} f_S x^S$，

[1] subset convolution

$g = \sum_{S\subseteq 2^U} g_S x^S$，定义 $f \cdot g = h$，其中 h 是一个集合幂级数，且 h_S 满足：

$$h_S = \sum_{L\subseteq S}\sum_{R\subseteq S}[L \cup R = S][L \cap R = \varnothing] f_L g_R \tag{30}$$

为了方便，$f \cdot g$ 可以简记为 fg。

5.1 快速算法

给出两个集合幂级数，暴力枚举集合 S 和子集 L，可以得到 $R = S \setminus L$，把 $f_L g_R$ 加到 h_S 上去，这样进行子集卷积的计算的时间复杂度是 $O(3^n)$ 的。这是因为 $\sum_{i=0}^{n}\binom{n}{i}2^i = 3^n$。

下面我们考虑更快的算法。

5.1.1 转化为集合并卷积

注意到 $[L \cup R = S][L \cap R = \varnothing] = [L \cup R = S][|L| + |R| = |S|]$，这就启发我们对 F 上的形式幂级数环 $F[[z]]$ 定义集合幂级数。通过增加未知数 z，我们可以用 z 的次数表示集合大小来限制 $|L| + |R| = |S|$。

对于一个 $F[[z]]$ 上的集合幂级数 σ，我们仍旧用符号 σ_S 表示 σ 的第 S 项的系数，但此时这个系数是个 F 上的形式幂级数。我们用 $\sigma_{S,i}$ 表示 σ_S 的第 i 项的系数。

定义 5.2. 我们称一个 $F[[z]]$ 上的集合幂级数 σ 为集合幂级数 f 的**集合占位幂级数**当且仅当对于任意一个集合 $S \in 2^U$，对于 $0 \le i < |S|$ 满足 $\sigma_{S,i} = 0$，且 $\sigma_{S,|S|} = f_S$。即：

$$\sigma = \sum_{S\in 2^U} f_S z^{|S|} x^S + \sum_{S\in 2^U}\sum_{k=|S|+1}^{\infty} \sigma_{S,i} z^i x^S \tag{31}$$

现在我们回到我们要研究的子集卷积。现在事情就很简单了，要把 f, g 给乘起来，我们就分别取它们的集合占位幂级数 σ, τ，然后求 σ, τ 的集合并卷积得到 δ，那么必有 $h_S = \delta_{S,|S|}$。由于形式幂级数加减是 $O(n)$ 的，相乘是 $O(n^2)$ 的，所以算法的时间复杂度为 $O(n^2 2^n)$。

把中间的集合并卷积的快速算法展开来看，算法就变成了：对集合占位幂级数 σ, τ 做莫比乌斯变换，对应系数乘起来，此处是个形式幂级数的乘法，然后再做莫比乌斯反演得到 δ。

5.1.2 转化为集合对称差卷积

注意到 $[L \cup R = S][L \cap R = \varnothing] = [L \oplus R = S][|L| + |R| = |S|]$，所以我们也可以取 f, g 的集合占位幂级数 σ, τ 然后求 σ, τ 的集合对称差卷积得到 δ，那么必有 $h_S = \delta_{S,|S|}$。算法的时间复杂度也是 $O(n^2 2^n)$。

5.1.3 比较

比较上述两种算法，由于集合对称差卷积的快速算法需要 2 有逆元，略有局限性，而实际上任意交换环上的集合幂级数都可以快速求集合并卷积，所以本文作者更推荐使用集合并卷积计算子集卷积。

5.2 由变换引出的性质

无论是用集合并卷积还是用集合对称差卷积，我们都把乘法变成了变换、系数相乘、逆变换，我们就可以把对某些集合幂级数的操作转化为对集合占位幂级数的变换的系数的操作，再做逆变换。

我们再来考虑除法。显然一个形式幂级数有乘法逆元当且仅当常数项不为 0，所以一个集合幂级数有乘法逆元当且仅当第 $\varnothing$ 项系数不为 0。我们可以用递推在 $O(n^2)$ 时间内求出形式幂级数的逆元的 0 到 n 次项，所以也就能在 $O(n^2 2^n)$ 时间内求出集合幂级数的乘法逆元。

我们还可以定义集合幂级数的导数，$(cx^S)' = \sum_{v \in S} cx^{S \setminus \{v\}}$。可以证明这样定义满足导数的性质。但是另一个方面，定义积分是很困难的。

在后面的应用中我们可以看得出，子集卷积比集合并卷积、集合对称差卷积更有组合意义。

6 应用

接下来我们来看看集合幂级数的一些应用。

6.1 随机集合并为全集的期望集合数

6.2 描述

设 $U = \{1, \ldots, n\}$。有一个人在玩游戏机。每回合，游戏机会随机产生一个 U 的子集，其中产生子集 S 的概率为 p_S。当游戏机产生的集合的并集为 U 时游戏结束。

给你 n 和每个集合 S 产生的概率 p_S，问期望多少回合游戏结束。

数据范围：$n \leq 20$。

6.2.1 解法

我们把 p 看成集合幂级数，那么第 k 回合结束后并集为 S 的概率为集合幂级数 p^k 的第 S 项。我们用集合并卷积来定义乘法。

那么游戏在第 k 回合结束的概率为 $p^k - p^{k-1}$ 的第 U 项。所以期望回合数就是下面这个集合幂级数的第 U 项：

$$\sum_{k=1}^{\infty} k(p^k - p^{k-1})$$

设这个集合幂级数为 f。

做莫比乌斯变换，那么：

$$\hat{f}_S = \sum_{k=1}^{\infty} k(\hat{p}_S^k - \hat{p}_S^{k-1})$$

由于概率是 $[0, 1]$ 中的实数且和为 1，所以 $0 \leq \hat{p}_S \leq 1$。所以：

$$\hat{f}_S = \begin{cases} -\frac{1}{1-\hat{p}_S} & \hat{p}_S < 1 \\ 0 & \hat{p}_S = 1 \end{cases}$$

然后对 $\hat{f}$ 做莫比乌斯反演得到 f 即可。

如果并集永远不可能为 U 那么 p^k 的第 U 项必为 0，所以这样得到的 f_U 也是 0。

所以我们就得到了时间复杂度 $O(n2^n)$ 的优秀算法。

6.3 Topcoder SRM 518 Nim

6.4 描述

两个人在玩 Nim 游戏，有 m 堆石子，每次可以选一堆石子拿走若干个石子，不能不拿。不能拿的人输。

求每堆石子数为不超过 l 的素数且先手必败的方案数对 10^9+7 取模后的结果。

数据范围：$m \le 10^9, l \le 10^5$。

6.5 解法

由斯普莱格 - 格隆第定理[1]知，先手必败当且仅当石子数的异或和为 0。

集合可以跟二进制数相对应，对称差就是异或。用集合对称差卷积来定义乘法，用 set(k) 表示 k 这个二进制数对应的集合，那么这个问题就是求：

$$\left(\sum_{p=1}^{l}[p\text{是素数}]x^{\text{set}(p)}\right)^m \tag{32}$$

的第 $\varnothing$ 项系数。

这里需要对一个集合幂级数求 m 次方。我们可以先对这个集合幂级数做沃尔什变换，然后用快速幂求每一个系数的 m 次方，然后再做沃尔什逆变换即可。

所以我们就得到了时间复杂度 $O(l\log l + l\log m)$ 的优秀算法。

6.6 连通生成子图计数

6.7 描述

给你一个 n 个结点的无向图 G，求连通生成子图的个数对 10^9+7 取模后的结果。

G 的生成子图即包含 G 中所有结点的子图。

数据范围：$n \le 20$。

[1] Sprague - Grundy theorem

6.8 解法

我们设 f_S 为 G 由结点集 S 导出的子图的连通生成子图的个数，g_S 为由结点集 S 导出的子图的生成子图的个数。特别地，令 $f_\varnothing = g_\varnothing = 0$。

把 f, g 看成集合幂级数，用子集卷积来定义乘法，则：

$$1 + g = \sum_{k\geq 0} \frac{f^k}{k!}$$

右边就是 e^f 的幂级数形式，所以我们可以写成：

$$1 + g = e^f$$

于是可以解得：

$$f = \ln(1 + g) = \sum_{k\geq 1} \frac{(-1)^{k+1}}{k} g^k$$

g 是好求的，只要求出每个导出子图的边数 m_S 然后对于 $S \neq \varnothing$ 就有 $g_S = 2^{m_S}$。所以我们可以求出 g 的一个集合占位幂级数 σ 然后做莫比乌斯变换得到 $\hat{\sigma}$，然后对于每个 $\hat{\sigma}_S$ 求 $\ln(1 + \hat{\sigma}_S)$，这里是对形式幂级数取自然对数，最后再做莫比乌斯反演得到 f 的一个集合占位幂级数。

对于形式幂级数 $a = \sum_{k\geq 1} a_k z^k$ 求 $b = \ln(1 + a)$ 的方法：

$$\begin{aligned}
b' &= \frac{a'}{1 + a} \\
b'(1 + a) &= a' \\
b'_k + \sum_{i=0}^{k-1} b'_i a_{k-i} &= a'_k \\
(k + 1)b_{k+1} &= (k + 1)a_{k+1} - \sum_{i=0}^{k-1} (i + 1)b_{i+1}a_{k-i} \\
b_{k+1} &= a_{k+1} - \frac{1}{k + 1} \sum_{i=0}^{k-1} (i + 1)b_{i+1}a_{k-i}
\end{aligned}$$

当然，$b_0 = 0$。这样就可以 $O(n^2)$ 递推出来 0 到 n 次项了。

所以我们就可以在 $O(n^2 2^n)$ 的时间内求出连通生成子图的个数。

7 总结

集合幂级数给我们打开了一扇门，对于以集合为状态的递推把集合幂级数作为其生成函数，我们可以从整体上来分析问题再用快速算法解决。

有关集合幂级数仍有许多有趣的问题，有待我们进一步研究和探索。愿此文对读者有所启发，创造出更加炫酷的应用。

8 感谢

感谢彭雨翔同学，他的那种学到了一点新科技就出成 OI 题给大家分享的精神令人感动，我从他那里学到了许多。本文也是从他的一篇博客延伸而来的。

感谢上海交通大学的郭晓旭学长，他就像移动信息学百科全书一样，平时帮助了我许多。子集卷积的快速算法我最初是从他那里听说的。

感谢金策和杨定澄同学为本文审稿。

感谢中国计算机学会提供学习和交流的平台。

参考文献

[1] Andreas Björklund and Petteri Kaski and Thore Husfeldt, “Fourier meets Möbius: fast subset convolution”. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing, pp. 67–74. ACM (2007)

[2] 彭雨翔，“Fast Walsh-Hadamard Transform”，http://picks.logdown.com/posts/179290-fast-walsh-hadamard-transform