全国青少年信息学奥林匹克竞赛

IOI2017中国国家候选队论文集

2017年4月

目 录

关于数列递归式的一些研究

雅礼中学 毛啸

摘 要

本文是作者2017年的中国信息学国家集训队论文。在这篇论文中，作者首先介绍了递归多项式，接着介绍了Berlekamp-Massey算法，接着讲述了它们在信息学竞赛中的应用前景，包括递归式的计数，以及求某些稀疏矩阵的特征多项式。

引言

递归多项式是一个作者原创的词。在信息学竞赛中，递归式通常是显式的，而选手的任务通常是给定一个数列的前若干项以及它的某个递归式，求这个数列的某一项。然而，本文研究的却是隐式的递归式，因此作者提出了一个全新的概念——递归多项式，这是一个十分强有力的概念，它不仅有着例如递归式计数这样的应用，而且还可以帮助我们学习和理解Berlekamp-Massey算法。

Berlekamp-Massey算法是一个被冷落的优秀算法。在信息学竞赛中，Berlekamp-Massey算法不仅不怎么为人所知，而且即使是知道它的人，也仅仅是把这个算法作为一个终南捷径。即使在全世界范围内，我也没有听说过这个算法作为任何一道题的标准算法出现。然而这个算法拥有着不少的应用。

因此，本文会将这两个内容呈现给大家，并介绍他们的一些应用。

1 递归多项式

在这一节中，作者会给出很多全新的定义，同时也会介绍很多有用的结论。

1.1 最小次数

定义 1.1. 对于数列$\{s_0, s_1, \cdots, s_{l-1}\}$，我们设这个数列对应的多项式为$S(x) = \sum_{k=0}^{l-1} s_k x^k$。

定义 1.2. 对于一个非零多项式A，它的**次数**定义为它的最高项次数，记为$\deg(A)$，特别的，零多项式次数为无穷小。

定义 1.3. 在某个域下，对于数列$\{r_0, r_1, \cdots, r_{m-1}\}$，以及数列$\{a_0, a_1, \cdots, a_{n-1}\}$，如果$r$不全为零，且对于任意$0 \le p \le n-m$，有

$$\sum_{k=0}^{m-1} a_{p+k} \times r_{m-k-1} = 0 \tag{1.1}$$

则称r为数列a的**递归式**[1]。

如果$r_0 = 1$，那么r被称为**递推式**，如果r是数列a的所有递推式中长度最短的，那么称r为数列a的**最短递推式**。

定义 1.4. 对于数列a和非零多项式$R(x)$，一定存在唯一的最短的递归式r使得对应的多项式是$R(x)$，我们称这个r为$R(x)$的对应的**最短递归式**。r的长度减一被称为多项式$R(x)$的**最小次数**，记为d_R或$d_{R(x)}$，多项式$R(x)$被称为a的d_R次**递归多项式**。

证明. 给定一个非零多项式$R(x)$和一个长度为n的数列a，它的零次项到$\max(\deg(R), n)$次项组成的数列一定是a的递归式且对应的多项式是$R(x)$，所以r存在。

如果非零多项式$R(x)$的最小次数是d，那么r一定是它的零次项到d次项组成的数列，所以r唯一。 □

1.2 递归多项式的运算

有了递归多项式，我们可以得到一些有用的结论，从这些结论出发我们可以做很多事情，而且对Berlekamp-Massey算法也有帮助。

引理 1.1. 多项式$R(x)$关于数列a的最小次数为d当且仅当d不小于$R(x)$的次数，且$R(x)\times A(x)$的d次项到$n-1$次项均为零。

证明. 我们发现1.1式是一个卷积的形式，据此容易推出该引理。 □

定理 1.1. 假设$F(x), G(x)$是关于数列a的最小次数不超过d的两个多项式，那么$F(x)+G(x)$关于数列a的最小次数也不超过d。

证明. 根据引理1.1，我们知道$F(x)\times A(x)$，$G(x)\times A(x)$的d次项到$n-1$次项均为零。因此$(F(x)+G(x))\times A(x)$的d次项到$n-1$次项均为零，且$F(x)+G(x)$的次数显然不超过d，于是$F(x)+G(x)$关于数列a的最小次数不超过d。 □

引理 1.2. 假设$F(x)$关于数列a的最小次数为d，那么对于$k \in N$且$k \ge 0$，$x^k F(x)$关于数列a的最小次数不超过为$d+k$。

[1]有些地方把这个叫做递推式，作者没有找到这两个词的区别，因此本文选择了一个词更少用的词进行了特殊的定义，而另一个词则保留了常见的定义。

证明. 根据引理1.1，我们知道$F(x)\times A(x)$的d次项到$n-1$次项均为零。因此$x^kF(x)\times A(x)$的$d+k$次项到$n+k-1$次项均为零，且$x^kF(x)\times A(x)$的次数显然不超过$d+k$，于是$x^kF(x)$关于数列a的最小次数不超过$d+k$。 □

定理 1.2. *假设$F(x)$关于数列a的最小次数为d_F，那么$F(x)\times G(x)$关于数列a的最小次数不超过$d_F+\deg(G)$。*

证明. 根据定理1.1和引理1.2可推出。 □

1.3 基

定义 1.5. *对于数列a，以及多项式序列$\{\mathrm{Base}_0(x),\mathrm{Base}_1(x),\mathrm{Base}_2(x),\cdots\}$，如果对于任意$k$，满足$\mathrm{Base}_k(x)$为最低项为$x^k$的所有多项式中最小次数最小的多项式，那么称多项式序列Base为数列a的基。*

我们发现，一个数列的一个基实际上就是它所张成的线性空间的基，我们实际上可以发现一个多项式的在一组基下的坐标和它的最小次数密切相关：

定理 1.3. *如果Base是数列a的基，则任意一个最小次数为d的多项式都可以唯一的被若干个最小次数不超过d的$\mathrm{Base}_k(x)$线性表出，且这个表示中一定至少存在一个$\mathrm{Base}_k(x)$它的最小次数恰好是d。*

证明. 假设$R(x)$是一个最小次数为d的多项式，我们考虑如何分解它。

假如它为零不需要分解，否则假如它的最低项是x^k，系数为C_r，$\mathrm{Base}_k(x)$的对应项系数为C_{Base}，那么$R(x)-C_{\mathrm{Base}}^{-1}C_r\mathrm{Base}_k(x)$的最低项至少是$x^{k+1}$，由于$\mathrm{Base}_k(x)$是最低项为$x^k$的多项式中最小次数最小的一个，它的最小次数显然不超过d，根据定理1.1，$R(x)-C_p^{-1}C_r\mathrm{Base}_k(x)$的最小次数也不超过$d$，我们对这个式子递归地进行分解即可。最后我们就得到了一个满足条件的线性表出方法。

由于$\mathrm{Base}_k(x)$是它所张成的线性空间的基，所以表示方法显然是唯一的。

如果表示方法中所有的$\mathrm{Base}_k(x)$的最小次数均小于d，那么要么这个多项式为零多项式，要么根据引理1.1它的最小次数会小于d，矛盾。 □

2 Berlekamp-Massey算法

本节将介绍Berlekamp-Massey算法[2]，虽然维基百科[1]等文献中对Berlekamp-Massey算法流程均有介绍，但是作者没有找到任何中文版本的算法流程。同时，作者也希望能用本论文中的语言重新阐述一下该算法的流程。

[2]维基百科中文翻译为伯利坎普－梅西算法。

2.1 算法简介

Berlekamp-Massey算法可以对任意域下长度为n的数列a在$O(n^2)$次运算内无显著额外空间的求出它的一组基Base。

2.2 算法流程

根据定义，所有最低项为$x^k(k \geq n)$的多项式最小次数都恰好是它的次数，因此对于$\text{Base}_k(x)$我们只要选任何一个$Cx^k(C \neq 0)$即可，我们不妨假设求出的每一个多项式的最低项均为1，那么这里我们把C设为1即可。

假设我们已经求出了$\text{Base}_{k+1}(x)$到$\text{Base}_n(x)$的多项式中最小次数的一个。

考虑如何求$\text{Base}_k(x)$。

我们考虑$x\text{Base}_k(x)$，根据1.3，我们只要使得它的线性表示中，最小次数最大的一个多项式的最小次数尽量小即可。

首先，由于$x\text{Base}_k(x)$的最低项为x^{k+1}，于是表示中一定有$\text{Base}_{k+1}(x)$这一项。

考虑剩下的项，显然它们当中最大的最小次数会尽量小。

设$\text{res}(x) = x\text{Base}_k(x) - \text{Base}_{k+1}(x)$。我们求出$\text{pro}(x) = \text{res}(x) \times A(x)$，根据引理1.1易知$\text{pro}(x)$的$k+1$到$n-1$次项均为零。

考虑n次项，如果它为零，那么我们已经知道$x\text{Base}_k(x)$，否则假设它为u。

考虑$\text{Base}_l(x)(l > k+1)$，如果$\text{Base}_l(x) \times A(x)$的$n$次项不为零，那么假设$n$次项为$v$，易知$\text{res}(x) - uv^{-1}\text{Base}_l(x)$的$k+1$次项到$n$次项均为零，由于最大的最小次数会尽量小，于是如果存在满足条件的多项式，那么$\text{Base}_l(x)$一定是所有满足条件的多项式中最小次数最小的，取这样的$P_l(x)$，我们便可以求出$x\text{Base}_k(x)$。假如这样的$\text{Base}_l(x)$不存在，那么只可能是$x\text{Base}_k(x)$的最小次数超过了n，这样的话我们只需要任选一个最低项为x^{k+1}次项且系数为1的多项式即可。

知道了$x\text{Base}_k(x)$，我们显然也就知道了$\text{Base}_k(x)$。

利用这个方法，每一个$P_k(x)$均可在不超过$O(n)$次运算内求出，因此总运算次数为$O(n)$，而需要运算的Base只有$O(n)$个。于是总运算次数为$O(n^2)$。

显然这个算法除了需要存储$A(x)$和Base外不需要显著的额外空间。

2.3 伪代码

根据上面的描述，这个算法可以用如下方法实现：

算法 1 Berlekamp-Massey Algorithm I

Require: $a, n = |a|$

Ensure: Base

1: $\text{Base}_k = x^k (k \geq n)$
2: $l \leftarrow n + 1$
3: $v \leftarrow 1$
4: **for** $k \leftarrow n - 1$ to 1 **do**
5: $\quad u \leftarrow [x^n]\text{Base}_{k+1}(x) \times A(x)$
6: $\quad \text{Base}_k(x) \leftarrow (\text{Base}_{k+1}(x) - uv^{-1}\text{Base}_l(x)) \div x$
7: $\quad$ **if** $(u \neq 0) \wedge (d_{\text{Base}_{k+1}} < d_{\text{Base}_l})$ **then**
8: $\quad\quad l \leftarrow k + 1$
9: $\quad\quad v \leftarrow u$
10: $\quad$ **end if**
11: **end for**=0

根据算法描述，l和v的初值并不重要，只要满足$l \geq n + 1$的初值均可以保证答案正确。然而，$l = n + 1, v = 1$是维英文维基百科上的参考初值，故作者将其作为一种约定俗成的方案。

然而，由于多项式运算比较麻烦，所以真正实现的时候，我们一般不会显式的使用多项式。

我们记r_k为$\text{Base}_{n-k+1} \div x^{n-k+1}$对应的最短递归式，那么我们容易看出$r_k$是数列$a$长度为$k$的前缀的最短递推式。那么，上述过程很容易改成计算$r$的过程。

我们把数列看成一个变长数组，下标从零开始，那么新的算法如下：

算法 2 Berlekamp-Massey Algorithm II

Require: $a, n = |a|$

Ensure: $r_k (0 \le k \le n)$

$r_0 = \{1\}$
last $\leftarrow \{1\}$
$v \leftarrow 1$
$s \leftarrow 1$
for $k \leftarrow 1$ to n **do**
  $r_k \leftarrow r_{k-1}$
  $u \leftarrow \sum_{i=0}^{|r_k|-1} a_{k-i-1} \times r_i$
  if $(u \neq 0)$ **then**
    if $|r_k| <$ last $+ s$ **then**
      r_k.append($\{0\} \times (|\text{last}| + s - |r_k|)$)
      for $i \leftarrow 0$ to $|\text{last}| - 1$ **do**
        $r_k[i + s] \leftarrow r_k[i + s] - uv^{-1}\text{last}[i]$
      end for
      last $\leftarrow r_{k+1}$
      $v \leftarrow u$
      $s \leftarrow 0$
    else
      for $i \leftarrow 0$ to $|\text{last}| - 1$ **do**
        $r_k[i + s] \leftarrow r_k[i + s] - uv^{-1}\text{last}[i]$
      end for
    end if
  end if
  $s \leftarrow s + 1$
end for=0

另外，假如我们只需要求整个序列的最短递推式，那么我们实际上任意时刻只需要存储一个r_k和一个last，于是不考虑每个元素的大小，空间复杂度可以降为$O(n)$。

3 递归式的计数

3.1 最小递归多项式与非平凡次小递归多项式

定义 3.1. 在所有已经求出的$\mathrm{Base}_k(x)$中，最小次数最小的一个被称为**最小递归多项式**，记为$M(x)$，若有多个任选一个，它的最小次数记为d_M。

在所有已经求出的$\mathrm{Base}_k(x)$中若存在一个若干多项式不能被表示为$x^k M(x), k \in N, k \geq 0$的形式，那么其中最小次数最小的一个被称为**非平凡次小递归多项式**，记为$S(x)$，若有多个任选一个，若不存在，则$S(x) = \mathrm{Base}_{n+1}(x)$，它的最小次数记为$d_S$。

3.2 线性表示

定义 3.2. 对于多项式$A(x), B(x), C(x)$，如果存在唯一的次数不超过$d_A - d_B$的多项式$F_B(x)$和次数不超过$d_A - d_C$的多项式$F_C(x)$，使得$A(x) = F_B(x)B(x) + F_C C(x)$，那么称$A(x)$能被$B(x), C(x)$**线性表示**。

引理 3.1. 对于多项式$A(x), B(x), C(x), D(x), E(x)$，如果$A(x)$能被$B(x), C(x)$线性表示，且$B(x), C(x)$均能被$D(x), E(x)$线性表出，那么$A(x)$也能$D(x), E(x)$线性表示。

证明. 设：

$$\begin{aligned} A(x) &= F_B(x)B(x) + F_C(x)C(x) \\ B(x) &= F_B D(x)D(x) + F_B E(x)E(x) \\ C(x) &= F_C D(x)D(x) + F_C E(x)E(x) \end{aligned}$$

则易知

$A(x) = (F_B(x)F_B D(x) + F_C(x)F_C D(x))D(x) + (F_D(x)F_B E(x) + F_C(x)F_C E(x))E(x)$

由于：

$$\begin{aligned} \deg(F_B(x)) &\leq d_A - d_B \\ \deg(F_B D(x)) &\leq d_B - d_D \end{aligned}$$

故有：

$\deg(F_B(x)F_B D(x)) = \deg(F_B(x)) + \deg(F_B D(x)) \leq d_A - d_B + d_B - d_D = d_A - d_D$

同理：

$\deg(F_C(x)F_C D(x)) = \deg(F_C(x)) + \deg(F_C D(x)) \leq d_A - d_C + d_C - d_D = d_A - d_D$

故：

$\deg((F_B(x)F_BD(x)+F_C(x)F_CD(x)))\le d_A-d_D$

同理：

$\deg((F_B(x)F_BE(x)+F_C(x)F_CE(x)))\le d_A-d_E$

这个过程保证了唯一性，定理得证。 □

定理 3.1. *在计算完*$\mathrm{Base}_k(x)$*后，它是最小的或者非平凡次小的。*

我们暂时不对这个定理进行证明，而是先介绍一个新的引理，再一起证明。

引理 3.2. *任意*$\mathrm{Base}_k(x)$*均可以被*$S(x)$*和*$M(x)$*线性表示。*

证明. 考虑Berlekamp-Massey算法的流程。

当$k=n$时，显然定理3.1以及引理3.2均成立。

假如在计算完$\mathrm{Base}_{k+1}(x)$时，它们均成立。

考虑计算$x\mathrm{Base}_k(x)$的过程，它最终一定是$\mathrm{Base}_{k+1}(x)$加上某一个$\mathrm{Base}_l(x)$乘上一个常数，根据算法描述我们容易发现$\mathrm{Base}_{k+1}(x)$和剩下的那一项均是$M(x)$和$S(x)$中的某一个。

于是$x\mathrm{Base}_k(x)$的最小次数等于$\max(d_M,d_S)$，而$\mathrm{Base}_k(x)$的最小次数还需要减一，因此它一定是最小的或者平凡次小的，这样我们证明了定理3.1，

显然对于它自己，引理3.2成立，然而由于$M(x),S(x)$发生了变化，我们还要考虑$\mathrm{Base}_l(x)(l>k)$。

考虑$M(x)$和$S(x)$的变化。

$M(x)$要么不变，要么成了新的$S(x)$。

考虑$S(x)$，根据算法流程，我们有$x\mathrm{Base}_k(x)=C_MM(x)+C_SS(x)$，于是我们有$S(x)=x\mathrm{Base}_k(x)-C_MM(x)$，由于$d_S>d_{\mathrm{Base}_k}$，且$d_S\ge d_M$，于是变化前的$S(x)$可以被变化后的$M(x)$和$S(x)$线性表示。

这样，变化前的$M(x)$和$S(x)$均可以被变化后的$M(x)$和$S(x)$线性表示，从而根据引理3.1，$\mathrm{Base}_l(x)(l>k)$仍可以被$S(x)$和$M(x)$线性表示。

同时，对于$k>n$的情况，由于$\mathrm{Base}_k(x)=x^{k-n}\mathrm{Base}_n(x)$，显然引理成立。

同样，这个过程保持了唯一性，引理得证。 □

定理 3.2. *任意多项式均可以被*$S(x)$*和*$M(x)$*线性表示。*

证明. 根据定理1.3和引理3.2容易得证。 □

3.3 递归式计数定理

定理 3.3 (递归多项式计数定理)**.** *假设域的大小为*q*，那么一个数列的*k*次递归多项式个数为：*

$$\mathrm{num}_k=q^{\max(k-d_M-1,0)+\max(k-d_S-1,0)}(q^{[k\ge d_M]+[k\ge d_S]}-1)$$

在这个式子中$[x]$*当*x*为真时为*1*，否则为*0*。*

证明. 根据定理3.2容易得证。 □

定理 3.4 (递归式计数定理)**.** 一个数列的递归式的个数为：

$\sum_{k=1}^{n}(n-k+1)\mathrm{num}_k$

证明. 一个x次递归多项式对应着恰好一个长度为$x, x+1, \cdots, n$的递归式，共有$n-x+1$个，故有此式。 □

4 特殊稀疏矩阵的特征多项式

Berlekamp-Massey算法不仅在递归式这个领域有着很强有力的作用，它在矩阵相关的问题中也能帮上忙。

于是，本章将介绍如何使用该算法计算特殊稀疏矩阵的特征多项式。

4.1 重要条件

当每个特征值仅对应一个Jordan块[3]的时候，特征多项式就是最小多项式，这是能使这个算法工作的重要条件。

4.2 实现过程

根据Cayley-Hamilton定理，一个$n\times n$的矩阵A的特征多项式就是零化多项式，也就是说假如特征多项式是p，那么$p(A)=0$。

回忆一下递归式的定义，我们发现如果我们随机一个向量v，首先显然有$p(Av)=0$，假如数域的大小足够大，有很大的概率$\{v, Av, Av^2, \cdots\}$这个向量序列存在一个唯一的最短递推式r，满足它的长度为n，那么这个时候$R(x)$就是特征多项式。

假如A是稀疏矩阵，元素个数为e，那么可以在$O(e)$次运算内根据Av^k计算出Av^{k+1}。

考虑使用Berlekamp-Massey算法，我们有两个难题。

第一个难题是如何快速处理向量数列。我们可以设计一个哈希函数，这个哈希函数是一个线性变换，将所有向量变成一个数，这样我们就可以处理向量数列了。

第二个难题是如何处理无限长的数列。虽然这个数列有无限长，但是我们可以发现在随机情况下长度为$2n$的数列的最短递推式长度大约是n，于是我们可以取一个足够长的前缀进行计算。一个更好的办法是考虑算法2，我们实际上可以不断增长前缀，直到最短递推式长度达到n为止。

于是，我们得到了可以在$O(n(n+e))$次运算、关于矩阵大小是线性的空间内计算特殊稀疏矩阵的特征多项式的蒙特卡罗算法。

[3]换句话说就是每个向量的高度都等于它的重数。

4.3 稀疏矩阵的行列式

考虑如何计算行列式$\det(A)$，显然如果这个矩阵满足4.1小节中的重要条件，我们就可以求特征多项式，然后取常数项再乘上$(-1)^n$即可。

假如这个矩阵不满足该条件，我们可以随机一个对角矩阵B，这个的运算次数是$O(ne)$。假如数域足够大，那么AB有很大的可能满足这个条件。由于$\det(AB) = \det(A)\det(B)$，我们只需要计算$\det(AB)$，并除以$\det(B)$即可。由于$B$是对角矩阵，$\det(B)$就是$B$矩阵对角线上的元素的乘积。

于是，我们得到了可以在$O(n(n+e))$次运算、关于矩阵大小是线性的空间内计算稀疏矩阵的行列式的蒙特卡罗算法。

4.4 稀疏图生成树计数

显然，生成树的个数可以利用基尔霍夫矩阵树定理计算。假如这个图足够稀疏，那么拉普拉斯矩阵的行列式显然可以用4.3小节中的方法计算。

这个算法的时间复杂度是$O(n(n+m))$，空间复杂度是$O(n+m)$，十分优秀。

5 总结

本文全文均围绕递归式展开，第一部分讲述了递归多项式，第二部分研究了Berlekamp-Massey算法，第三、四部分主要介绍了它们的一些应用。

我希望这篇文章能起到抛砖引玉的作用。本文仍然是对递归式的不成熟研究，介绍的应用也都比较基础，而科技在不断进步，一定还有更多有趣的应用等着我们去发现。因此，愿此文能对读者有所启发，引出一些更加炫酷的应用。

鸣谢

1. 感谢杜瑜皓同学，我最早是从他口中听说的这个算法，且本文中部分内容的出现离不开他的帮助。

2. 感谢杜瑜皓同学，杨家齐同学与翁文涛同学的纠错。

3. 感谢中国计算机学会提供学习和交流的平台。

参考文献

[1] 维基百科

https://en.wikipedia.org/wiki/Berlekamp%E2%80%93Massey_algorithm

基于线性代数的一般图匹配

中山市中山纪念中学 杨家齐

摘 要

图的匹配问题是图论中的经典问题，无论在学术界还是在算法竞赛界都有着重要的地位。本文主要介绍了一种基于线性代数的一般图匹配算法，并给出了它的一些简单应用。这个算法与传统的解决这一问题的组合方法有很大不同，希望这个算法本身以及在得到这个算法的过程中所用到的思想能够给读者带来启发。

引言

图的匹配问题按照图的性质可以分为二分图匹配和一般图匹配两类。其中二分图匹配已经在算法竞赛界中出现很多年了，是算法竞赛的一个常见考点。由于它的算法相对简单，因此它得到了很好的普及：如今大部分选手都已经掌握、并且能够熟练应用二分图匹配的算法。

而一般图匹配则是一个新兴的考点，近几年的集训队作业、冬令营考试以及集训队互测中都有它的身影。但这些比赛都是 NOI 级别以上的，在 NOI 以下的比赛中罕见考察一般图匹配的题目，并且目前能够掌握一般图匹配算法的选手也相对较少。笔者认为，一般图匹配算法普及度不够的一个重要原因是目前解决这类问题常见的带花树算法较为复杂，学习门槛较高。因此本文将要介绍一个便于记忆，易于实现的匹配算法，希望这个算法能够促进一般图匹配算法的普及。

正如本文的题目所述，本文将要介绍的算法是一个代数方法。尽管在算法竞赛中组合方法更为常见，但代数方法同样是解决图论问题的强有力工具。这其中最广为人知的例子恐怕就是生成树计数的 Matrix-Tree 定理了。代数方法是解决问题的一个全新的视角，希望本文能够让读者感受到代数方法在解决问题中的独特魅力。

本文将首先从完美匹配问题入手，解决它对应的判定性问题，然后得到一个构造完美匹配方案的算法，并通过一些线性代数知识来逐步优化它，接着将最大匹配转化为完美匹配问题解决，最后再叙述这一算法的一些应用与拓展。

1 预备知识

1.1 图论

若无特殊说明，文中涉及的图均指无向图。

我们用 $G=(V,E)$ 来表示一个图，其中 V 是点集，E 是边集。我们一般用 n 来表示点集的大小 $|V|$，用 m 来表示边集的大小 $|E|$，并且我们一般认为 $V=\{v_1,v_2,\dots,v_n\}$。

在不致产生歧义的情况下，我们用 uv 表示边 (u,v)。

图 $G=(V,E)$ 的**匹配** M 是指边集 E 的一个子集，满足 M 中任意两条边都没有公共点。如果结点 v 是 M 中某条边的端点，那么我们称 v 在匹配 M 中。图 G 的所有匹配的大小的最大值记作 $\nu(G)$，并且如果 $|M|=\nu(G)$，那么我们称 M 是图 G 的一个**最大匹配**。特别地，如果图 G 中的每个点都在匹配 M 中，那么我们称 M 是图 G 的一个**完美匹配**。显然，只有当 $|V|$ 为偶数时图 G 才可能有完美匹配。

对于一个图 $G=(V,E)$ 以及一个点集 $U\subseteq V$，我们把从图 G 中删去 U 中所有点得到的图记作 $G-U$，即 $G-U=(V\backslash U,E\cap\{uv|u,v\in V\backslash U\})$；把图 G 关于 U 的导出子图记作 $G[U]$，即 $G[U]=G-(V\backslash U)$。

1.2 线性代数

对于一个 m 行 n 列的矩阵 A，我们设它的行号集合为 $\{1,\dots,m\}$，列号集合为 $\{1,\dots,n\}$。矩阵 A 第 i 行第 j 列的元素记作 $A_{i,j}$。

对于一个 $m\times n$ 的矩阵 A 以及任意 $I\subseteq\{1,\dots,m\},J\subseteq\{1,\dots,n\}$，我们记从 A 中保留 I 中所有行以及 J 中所有列得到的子矩阵为 $A_{I,J}$。我们将 $A_{I,\{1,\dots,n\}}$ 简记为 $A_{I,\cdot}$，将 $A_{\{1,\dots,m\},J}$ 简记为 $A_{\cdot,J}$。

类似地，我们记 $A^{I,J}$ 为从 A 中删去 I 中所有行及 J 中所有列得到的子矩阵。

对于一个 $n\times n$ 的方阵 A，我们将它的**行列式**记作 $\det A$，并且

$$\det A=\sum_{\pi\in\Sigma_n}(-1)^{\operatorname{sgn}\pi}\prod_{k=1}^{n}A_{k,\pi(k)}$$

其中 Σ_n 表示 $\{1,\dots,n\}$ 的所有置换构成的集合，$\operatorname{sgn}\pi$ 表示置换 π 的符号。如果序列 $\{\pi(1),\dots,\pi(n)\}$ 的逆序对数为 x，那么 $\operatorname{sgn}\pi=(-1)^x$。

给定 m 个 n 维向量 $\{v_1,\dots,v_m\}$，如果存在 $\lambda_1,\lambda_2,\dots,\lambda_m$，使得 $v=\sum_{i=1}^{m}\lambda_i v_i$，则称 v 是 $\{v_1,\dots,v_m\}$ 的**线性组合**。如果这 m 个向量中的任意一个都不能够表示为其它向量的线性组合，那么我们称这 m 个向量**线性无关**，否则称为**线性相关**。

对于一个矩阵 A，如果我们将它的每一行看作一个向量，那么从中选出线性无关向量的极大数目称为行秩；如果我们将每一列看作一个向量，得到的极大线性无关向量数目称为列秩。矩阵的行秩和列秩总是相等的，因此统称为矩阵 A 的**秩**，记作 $\operatorname{rank}A$。

最后，我们不加证明地给出一个结论

定理 1.1. *以下四个问题的复杂度相等*

1. *计算矩阵的行列式；*
2. *将两个矩阵相乘；*
3. *求矩阵的逆；*
4. *对矩阵做高斯消元（LU 分解）。*

我们设这四者的复杂度都是 $O(n^\omega)$[4]。

2 完美匹配的存在性

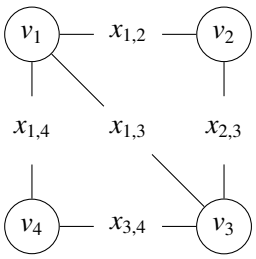
2.1 Tutte 定理

定义 2.1 (Tutte 矩阵)**.** 对于一个无向图 $G = (V, E)$, 定义 G 的 Tutte 矩阵为一个 $n \times n$ 的矩阵 $\tilde{A}(G)$, 其中

$$\tilde{A}(G)_{i,j} = \begin{cases} x_{i,j} & \text{若 } v_i v_j \in E \text{ 并且 } i < j \\ -x_{j,i} & \text{若 } v_i v_j \in E \text{ 并且 } i > j \\ 0 & \text{其它情况} \end{cases}$$

其中，$x_{i,j}$ 是一个与边 $v_i v_j$ 相关联的独一无二的变量。

下面我们来看一个例子。



$$\begin{pmatrix} 0 & x_{1,2} & x_{1,3} & x_{1,4} \\ -x_{1,2} & 0 & x_{2,3} & 0 \\ -x_{1,3} & -x_{2,3} & 0 & x_{3,4} \\ -x_{1,4} & 0 & -x_{3,4} & 0 \end{pmatrix}$$

图 1: 图 G 与它的 Tutte 矩阵

图 1 的左半部分给出了一个图 G，右半部分给出了图 G 所对应的 Tutte 矩阵 $\tilde{A}(G)$。可以看到，图 G 的每条边都对应着一个变量，因此 $\tilde{A}(G)$ 中总共会用到 $|E|$ 个变量。

[4]学术界一般认为 $2 \le \omega < 2.373$。

2.2 Tutte 定理

Tutte 矩阵与图的完美匹配的存在性之间有着密切的连系。事实上，我们有如下定理

定理 2.1 (Tutte)**.** 图 G 有完美匹配当且仅当 $\det \tilde{A}(G) \neq 0$。

为了证明这个定理，我们首先需要用“偶环覆盖”的概念来刻画一个具有完美匹配的图。

定义 2.2. 图 $G = (V, E)$ 的一个环覆盖是指用若干个环去覆盖图 G 的所有结点，使得图 G 的任意一个结点恰好在一个环中。

形式化地说，图 G 的每个环覆盖对应于 1 到 n 的一个排列 π，满足 $v_i v_{\pi(i)} \in E$。

如果这个环覆盖中的所有环的长度都是偶数，那么我们称这是一个偶环覆盖，否则我们称它是一个奇环覆盖。

注意，定义 2.2 并没有要求 $v_i v_{\pi(i)}$ 两两不同，也就是说环覆盖中可以出现重复的边。

引理 2.2. 图 $G = (V, E)$ 有完美匹配当且仅当图 G 有一个偶环覆盖。

证明. 若 G 有一个完美匹配，那么我们直接用匹配中每一对点构成的二元环就可以覆盖图 G 了。

若 G 有一个偶环覆盖，那么我们在每一个偶环中隔一条边取一条边，就能够得到 G 的一个完美匹配了。 □

下图 2 将说明如何从图 G 的一个偶环覆盖得到一个完美匹配。

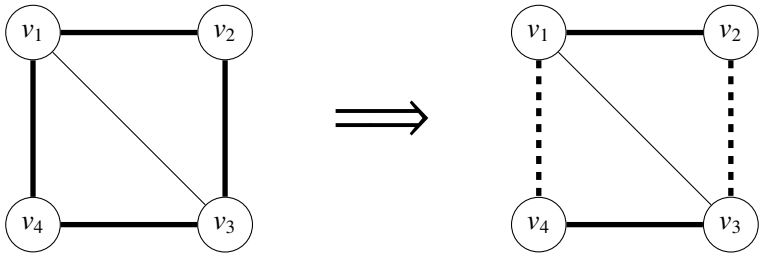


图 2: 偶环覆盖与完美匹配

有了引理 2.2，我们就可以证明 Tutte 定理（定理2.1）了。

证明(Tutte 定理). 我们有

$$\det \tilde{A}(G) = \sum_{\pi \in \Sigma_n} (-1)^{\mathrm{sgn}(\pi)} \prod_{k=1}^{n} \tilde{A}(G)_{k,\pi(k)} \tag{2.1}$$

等式右边的非 0 项形如 $(-1)^{\mathrm{sgn}(\pi)} \prod_{k=1}^{n} \pm x_{k,\pi(k)}$，其中 $v_1 v_{\pi(1)}, \ldots, v_n v_{\pi(n)}$ 是 G 中的边。这些边构成了图 G 的一个环覆盖（与 π 的轮换分解相对应）。

接下来，如果我们将 π 中任意一个长度大于 2 的环反向，那么我们会得到一个相同的环覆盖。现在我们来观察它的符号将如何变化。考虑一个通过将 π 中的某个环反向后得到的置换 π'。将单个环反向并没有改变置换 π 的符号（即 $\operatorname{sgn}(\pi) = \operatorname{sgn}(\pi')$），但改变了环中所有变量 $x_{i,\pi(i)}$ 的符号。

因此，如果我们将 π 中的一个偶环反向，那么得到的项符号与 π 所对应的项是相同的，否则是相反的。

于是所有存在奇环的环覆盖都会被一正一负相抵消掉。由于 $\det \tilde{A}(G) \neq 0$，这就意味着图 G 存在一个偶环覆盖，因此图 G 有完美匹配。 □

为了更加直观地理解这个证明，我们下面以一个三元环为例，来说明为什么奇环覆盖不会出现在 $\det \tilde{A}(G)$ 中。

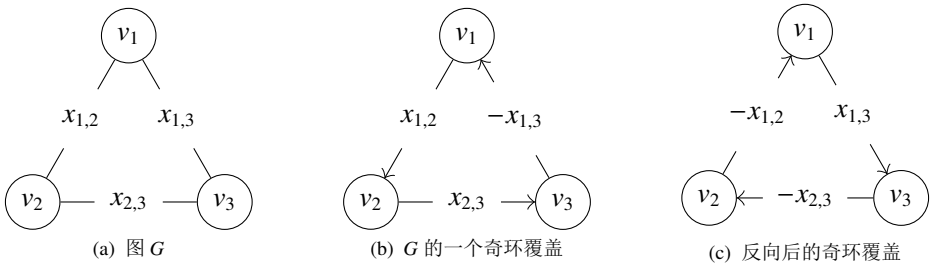


图 3: 三元环 G

我们取图 G 的一个奇环覆盖来进行分析。如图 3(b)，不妨假设我们取的环覆盖所对应的排列为 $\pi = (2\ 3\ 1)$，那么这一项对 $\det \tilde{A}(G)$ 的贡献为 $\operatorname{sgn}(\pi) x_{1,2} x_{2,3} (-x_{1,3})$。现在我们将 π 反向，那么我们将会得到图 3(c)，它对应的排列为 $\pi' = \pi^{-1} = (3\ 1\ 2)$，它对 $\det \tilde{A}(G)$ 的贡献为 $\operatorname{sgn}(\pi')(-x_{1,2})(-x_{2,3})x_{1,3}$。

显然 $\operatorname{sgn}(\pi) = \operatorname{sgn}(\pi')$，并且我们可以看到，$\pi'$ 的贡献是在 π 的贡献的基础上将环上每条边对应的变量都乘以 -1 得到的。因此这两项的符号恰好相反，它们的总贡献是 0。

2.3 随机化

Tutte 定理（定理 2.1）给了我们一个非常好的判定图 G 是否存在完美匹配的算法：我们只需要判断 $\det \tilde{A}(G)$ 是否为零就可以了。但这个做法有一个显著的问题：由于 Tutte 矩阵的每一项都是一个变量，并且变量的总数高达 $|E|$ 个，因此直接计算行列式是非常困难的，所需要的时间甚至是指数级的。

然而，我们并不一定需要计算出 $\det \tilde{A}(G)$，我们所需要做的仅仅是判断 $\det \tilde{A}(G)$ 是否**恒为** 0。

判定一个多项式是否恒为 0 是一个经典问题。考虑我们现在有一个 n 元多项式 $P(x_1, x_2, \ldots, x_n)$，现在我们想判定 $P(x_1, x_2, \ldots, x_n)$ 是否恒为 0，我们该怎么做。

我们首先可以观察到，如果 $P(x_1, x_2, \ldots, x_n) = 0$，那么不论我们选取什么值代入多项式 P 中，得到的结果都一定是 0。那么我们是否能够选取 n 个随机数 $r_1, r_2, \ldots, r_n$，然后通过 $P(r_1, r_2, \ldots, r_n)$ 是否为 0 来推断出 $P(x_1, x_2, \ldots, x_n)$ 是否恒为 0 呢？下面将要叙述的 Schwartz-Zippel 引理将会告诉我们，这个看起来不太靠谱的做法实际上**有很大概率**能得到正确的结果。

引理 2.3 (Schwartz, Zippel)**.** *对于域 $\mathbb{F}$ 上的一个不恒为 0 的 n 元 d 度多项式 $P(x_1, x_2, \ldots, x_n)$，设 $r_1, r_2, \ldots, r_n$ 为 n 个 $\mathbb{F}$ 中的独立选取的随机数，则*

$$\Pr[P(r_1, r_2, \ldots, r_n) = 0] \leq \frac{d}{|\mathbb{F}|}$$

证明. 我们尝试归纳证明这个定理。

考虑单变量的情况：此时 P 至多有 d 个根，因此我们恰好选到其中一个的概率不超过 $\frac{d}{|\mathbb{F}|}$。

现在假设定理对于 $(n-1)$ 的情况成立，我们考虑将 P 中的所有项按照 x_1 的次数分类，设

$$P(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x_1^i P_i(x_2, \ldots, x_n)$$

由于 $P \not\equiv 0$，因此存在一个 i 使得 $P_i \not\equiv 0$，我们考虑所有满足条件的 i 的最大者，则 $\deg P_i \leq d - i$。

由归纳假设，$\Pr[P_i(r_2, \ldots, r_n) = 0] \leq \frac{d-i}{|\mathbb{F}|}$。若 $P_i(r_2, \ldots, r_n) \neq 0$，那么 $P(x_1, r_2, \ldots, r_n)$ 为 i 次单项式，因此

$$\Pr[P(r_1, r_2, \ldots, r_n) = 0 | P_i(r_2, \ldots, r_n) \neq 0] \leq \frac{i}{|\mathbb{F}|}$$

设事件 $P(r_1, r_2, \ldots, r_n) = 0$ 为 A，$P_i(r_2, \ldots, r_n) = 0$ 为 B，那么

$$\begin{aligned}
\Pr[A] &= \Pr[A \cap B] + \Pr[A \cap B^c] \\
&= \Pr[B] \Pr[A|B] + \Pr[B^c] \Pr[A|B^c] \\
&\leq \Pr[B] + \Pr[A|B^c] \\
&\leq \frac{d-i}{|\mathbb{F}|} + \frac{i}{|\mathbb{F}|} = \frac{d}{|\mathbb{F}|}
\end{aligned}$$

□

我们知道，对于一个给定的图 G，多项式 $\det \tilde{A}(G)$ 的度是 $O(n)$ 级别的。因此我们不妨选取一个 n^2 级别的质数 p，并在模 p 域 $\mathbb{F}_p$ 下为每条边 $v_i v_j$ 所对应的变量 $x_{i,j}$ 随机赋一个

值，然后通过计算[5] $\det \tilde{A}(G)$ 是否为 0 来判断图 G 是否存在完美匹配。由 Schwartz-Zippel 引理，我们判断错误的概率不超过 $\frac{n}{p} = O(\frac{1}{n})$，这个概率对于我们来说是可以接受的。并且我们可以通过调整 p 的大小，或者多次随机来降低错误率。

从而，我们有如下判定算法

算法 1 Lowasz's testing algorithm

```
if det Ã(G) ≠ 0 then
    print "YES"
else
    print "NO"
end if=0
```

定理 2.4. *算法 1 的时间复杂度为* $O(n^\omega)$。

3 构造完美匹配方案

3.1 一个暴力算法

根据算法 1，我们现在已经能够判定图 G 是否存在完美匹配了。那么我们可以立刻得到一个构造匹配的暴力算法：枚举每条边，删去它的两个端点，并判断剩下的图是否存在完美匹配。

算法 2 A naive matching algorithm

```
M ← ∅
for i ← 1 to n do
    for j ← 1 to n do
        if v_i v_j ∈ E(G)and det Ã(G − {v_i, v_j}) ≠ 0 then
            G ← G − {v_i, v_j}
            M ← M ∪ {v_i v_j}
        end if
    end for
end for
return M =0
```

算法 2 需要进行 $O(n^2)$ 次循环，每次循环需要计算一个 $O(n)$ 阶的方阵的行列式，因此

[5]从此，当我们需要计算 $\tilde{A}(G)$ 的行列式、逆矩阵，或是需要对 $\tilde{A}(G)$ 消元时，我们指的是将变量随机赋值后在 $\mathbb{F}_p$ 下进行相应的操作。

定理 3.1. *算法 2 的时间复杂度为 $O(n^{\omega+2})$。*

3.2 一些线性代数知识

算法 2 的瓶颈在于需要判断每条边是否在完美匹配中。那么如果我们能加速这一判断过程，就可以降低这一算法的复杂度了。

为了达到这一目的，我们需要先了解一些线性代数的知识。

定义 3.1. 矩阵 A 关于第 i 行和第 j 列的余子式（记做 $M_{i,j}$）为 $\det A^{i,j}$。

定义 3.2. 矩阵 A 关于第 i 行和第 j 列的代数余子式（记做 $C_{i,j}$）为 $(-1)^{i+j}M_{i,j}$。

矩阵 A 的余子矩阵是一个 $n \times n$ 的矩阵 C，满足其第 i 行第 j 列的元素是 A 关于第 i 行和第 j 列的代数余子式。

定义 3.3. 矩阵 A 的伴随矩阵 $\operatorname{adj} A$ 是 A 的余子矩阵的转置矩阵，即 $\operatorname{adj} A = C^T$。

定理 3.2 (拉普拉斯展开)**.** *设矩阵 A 为一个 $n \times n$ 的矩阵，那么对于任意一行 $i \in \{1, \ldots, n\}$，有*

$$\begin{aligned}\det A &= \sum_{j=1}^{n} A_{i,j}C_{i,j} \\ &= \sum_{j=1}^{n} A_{i,j}(\operatorname{adj} A)_{j,i}\end{aligned}$$

类似地，对于任意一列 $j \in \{1, \ldots, n\}$，有

$$\det A = \sum_{i=1}^{n} A_{i,j}(\operatorname{adj} A)_{j,i}$$

定理 3.3. *如果矩阵 A 可逆，那么我们有*

$$A^{-1} = \operatorname{adj} A / \det A$$

证明. 由定义 3.2 及定义 3.3，$A \times \operatorname{adj} A$ 的第 i 行第 i 列的系数是 $\sum_{j=1}^{n} A_{i,j}C_{i,j}$。根据定理 3.2，这个系数等于 $\det A$。

如果 $i \neq j$, 那么 $A \times \operatorname{adj} A$ 的第 i 行第 j 列的系数是 $\sum_{k=1}^{n} A_{i,k}C_{j,k}$。实际上，这就相当于把 A 的第 j 行元素换成第 i 行元素后求行列式。由于有两行相同，因此行列式为 0。

于是我们可以得到

$$A \times \operatorname{adj} A = \operatorname{adj} A \times A = \det A \times I_n$$

即

$$A^{-1} = \operatorname{adj} A / \det A$$

□

3.3 斜对称矩阵的性质

我们先定义斜对称矩阵的概念。

定义 3.4. 对于一个 $n \times n$ 的矩阵 A，如果 $A^T = -A$，即 $A_{i,j} = -A_{j,i}$，那么我们称 A 为斜对称矩阵。

定义 2.1 中的 Tutte 矩阵就是一个斜对称矩阵。可以看到，我们前面的算法都是在围绕计算 Tutte 矩阵的行列式展开，这正是我们研究斜对称矩阵性质的原因。

引理 3.4. *对于一个 $n \times n$ 的斜对称矩阵 A，如果 n 是奇数，那么 $\det A = 0$。*

证明. 由行列式的相关知识可知，$\det A = \det A^T$。又因为 $A^T = -A$，所以 $\det A = \det(-A) = (-1)^n \det A$。

当 n 为奇数时 $(-1)^n = -1$，即 $\det A = -\det A$，于是 $\det A = 0$。 □

定理 3.5. *对于一个 $n \times n$ 的斜对称矩阵 A 和一个行号的子集 $I = \{i_1, i_2, \ldots, i_k\}$，使得 $A_{i_1,\cdot}, A_{i_2,\cdot}, \ldots, A_{i_k,\cdot}$ 是 A 的一组关于行的极大线性无关组，那么 $\det A_{I,I} \neq 0$。*

证明. 不失一般性，设 $I = \{1, 2, \ldots, k\}$。

因为 $A_{1,\cdot}, A_{2,\cdot}, \ldots, A_{k,\cdot}$ 是 A 关于行的一组极大线性无关组，由 A 的斜对称性我们可以得到 $A_{\cdot,1}, A_{\cdot,2}, \ldots, A_{\cdot,k}$ 是 A 关于列的一组极大线性无关组。

因此，对于所有 $k < i \leq n$，$A_{\cdot,i}$ 都可以被 $A_{\cdot,1}, A_{\cdot,2}, \ldots, A_{\cdot,k}$ 线性表出。于是 $\operatorname{rank} A_{I,I} = \operatorname{rank} A_{I,\cdot} = k$，即 $\det A_{I,I} \neq 0$。 □

推论 3.6. *对于一个斜对称矩阵 A，存在一个行号集合 $I = \{i_1, i_2, \ldots, i_k\}$，使得 $\operatorname{rank} A = \operatorname{rank} A_{I,I} = k$。*

定理 3.7. *一个斜对称矩阵的秩为偶数。*

证明. 由推论 3.6 可知，斜对称矩阵存在一个秩与它本身相等的满秩主子式。由于斜对称矩阵的主子式也是斜对称的，结合引理 3.4 可知，这个主子式的秩为偶数。因此一个斜对称矩阵的秩为偶数。 □

引理 3.8. *对于一个 $n \times n$ 的可逆斜对称矩阵 A，以及任意的 $i, j \in \{1, \ldots, n\}, i \neq j$，我们有 $\det A^{i,j} \neq 0$ 当且仅当 $\det A^{\{i,j\},\{i,j\}} \neq 0$。*

证明. 如果 $\det A^{i,j} \neq 0$，那么 $\operatorname{rank} A^{i,j} = n - 1$。由于 $A^{\{i,j\},\{i,j\}}$ 是在 $A^{i,j}$ 的基础上删去一行一列得到，因此 $\operatorname{rank} A^{\{i,j\},\{i,j\}} \geq n - 3$。又因为 $A^{\{i,j\},\{i,j\}}$ 是斜对称矩阵，由定理 3.7 它的秩为偶数，因此 $\operatorname{rank} A^{\{i,j\},\{i,j\}}$ 一定为 $n - 2$。于是 $\det A^{\{i,j\},\{i,j\}} \neq 0$。

反过来, 如果 $\det A^{\{i,j\},\{i,j\}} \neq 0$，那么 $\operatorname{rank} A^{\{i,j\},\{i,j\}} = n - 2$，同时 $\operatorname{rank} A^{i,\{i,j\}} = n - 2$，根据定理 3.7，$\operatorname{rank} A^{i,\{i,j\}} = \operatorname{rank} A^{i,i} = n - 2$。这表明 $A^{i,\emptyset}$ 的第 j 列是其它列的线性组合。于是 $\operatorname{rank} A^{i,j} = \operatorname{rank} A^{i,\emptyset} = n - 1$，$\det A^{i,j} \neq 0$。 □

3.4 Rabin-Vazirani 算法

现在我们尝试得到一个比算法 2 更优的构造匹配方案的算法。

定理 3.9 (Rabin, Vazirani)**.** *设 $G = (V, E)$ 是一个有完美匹配的图，$\tilde{A} = \tilde{A}(G)$ 是 G 所对应的 Tutte 矩阵。那么，$(\tilde{A}^{-1})_{i,j} \neq 0$ 当且仅当 $G - \{v_i, v_j\}$ 有完美匹配。*

证明. 这个定理可以由定理 3.3，引理 3.8 以及 Tutte 定理（定理 2.1）得到。 □

基于定理 3.9，我们可以得到如下算法

算法 3 Matching algorithm of Rabin and Vazirani

```
M ← ∅
for i ← 1 to n do
    if v_i is not yet matched then
        compute Ã(G)^{-1}
        find j, such that v_i v_j ∈ E(G) and (Ã(G)^{-1})_{j,i} ≠ 0
        G ← G − {v_i, v_j}
        M ← M ∪ {v_i v_j}
    end if
end for
return M =0
```

算法 3 共需要进行 $O(n)$ 次循环，每次循环需要对一个 $O(n)$ 阶的方阵求一次逆，因此

定理 3.10. *算法 3 的复杂度为 $O(n^{\omega+1})$。*

3.5 基于高斯消元的构造方法

算法 3 的瓶颈在于计算 Tutte 矩阵 $\tilde{A}(G)$ 的逆。事实上，在每次迭代过程中，我们仅仅删除了 $\tilde{A}(G)$ 的两行两列，但却需要从零开始重新计算矩阵的逆。如果我们每次不再需要重新计算矩阵的逆，而是能够用一些高效的方法来维护矩阵的逆，那么我们就可以降低这个算法的复杂度。

我们的算法基于如下定理

定理 3.11. *如果*

$$A = \begin{pmatrix} a_{1,1} & v^T \\ u & B \end{pmatrix} \quad A^{-1} = \begin{pmatrix} \hat{a}_{1,1} & \hat{v}^T \\ \hat{u} & \hat{B} \end{pmatrix}$$

其中 $\hat{a}_{1,1} \neq 0$。那么 $B^{-1} = \hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}$。

证明. 由于 $AA^{-1} = I$, 于是我们有

$$\begin{pmatrix} a_{1,1}\hat{a}_{1,1} + v^T\hat{u} & a_{1,1}\hat{v}^T + v^T\hat{B} \\ u\hat{a}_{1,1} + B\hat{u} & u\hat{v}^T + B\hat{B} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & I_{n-1} \end{pmatrix}$$

从而

$$\begin{aligned} B(\hat{B} - \hat{u}\hat{v}^T/\hat{a}_{1,1}) &= I_{n-1} - u\hat{v}^T - B\hat{u}\hat{v}^T/\hat{a}_{1,1} \\ &= I_{n-1} - u\hat{v}^T + u\hat{a}_{1,1}\hat{v}^T/\hat{a}_{1,1} \\ &= I_{n-1} - u\hat{v}^T + u\hat{v}^T \\ &= I_{n-1} \end{aligned}$$

□

注意到，在上面这个定理中，我们对 $\hat{B}$ 的修改操作恰好是高斯消元中的一次消去操作。在定理中我们是以消去第一行第一列为例来证明的。

作为定理 3.11 的一个直接应用，我们可以得到如下算法

算法 4 Simple matching algorithm

```
M ← ∅
compute Ã(G)^{-1}
for i ← 1 to n do
    if the i-th row is not yet eliminated then
        find j such that v_i v_j ∈ E(G) and (Ã(G)^{-1})_{j,i} ≠ 0
        G ← G − {v_i, v_j}
        M ← M ∪ {v_i v_j}
        update Ã(G)^{-1} by
        eliminating the i-th row and the j-th column
          and then the j-th row and the i-th column
    end if
end for
return M =0
```

算法 4 共需要循环 $O(n)$ 次，每次循环需要进行两次消去操作，单次消去操作的时间复杂度为 $O(n^2)$，因此

定理 3.12. *算法 4 的时间复杂度为 $O(n^3)$。*

4 最大匹配

4.1 最大匹配的大小

到目前为止，我们的所有算法都是用来求解完美匹配的，但在实际应用中我们要求解的往往是最大匹配。

与之前解决完美匹配问题的思路类似，我们先从最简单的问题入手：对于一个给定的图 $G = (V, E)$，如何求出图 G 的最大匹配的大小 $\nu(G)$？对于这个问题，我们有如下定理

定理 4.1. 图 G 的最大匹配的大小 $\nu(G)$ 是 $\frac{1}{2}\operatorname{rank}\tilde{A}(G)$。

证明. 设 $k = \operatorname{rank}\tilde{A}(G)$。由推论 3.6，我们可以选出 $\tilde{A}(G)$ 的一个行号集合 $I = \{i_1, i_2, \ldots, i_k\}$，使得 $\operatorname{rank}\tilde{A}(G) = \operatorname{rank}\tilde{A}(G)_{I,I} = k$。设 $U = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$，那么 $\tilde{A}(G)_{I,I} = \tilde{A}(G[U])$。由于 $\operatorname{rank}\tilde{A}(G[U]) = |U|$，因此图 $G[U]$ 存在完美匹配，这说明 $\nu(G) \geq \frac{k}{2}$。

如果 $\nu(G) > \frac{k}{2}$，我们考虑取出所有在最大匹配中的点，那么一定存在一个行号集合 I，使得 $|I| > k$ 并且 $\det\tilde{A}(G)_{I,I} \neq 0$，这意味着 $\tilde{A}(G)$ 的子矩阵的秩大于它本身的秩，显然这是不可能的。

综上，$\nu(G) = \frac{k}{2}$。 □

4.2 转化一

接下来我们尝试把最大匹配转化为我们所熟悉的完美匹配问题。如果对于一个图 G，我们已经求出它的最大匹配的大小 $\nu(G)$ 了，那么我们可以在图 G 的基础上新增 $n - 2\nu(G)$ 个点，并从新加进来的点往之前 n 个点的每一个连边，这样得到的新图就是一个有完美匹配的图了，并且这个新图的完美匹配与原图的最大匹配相对应。

这样我们就解决了最大匹配问题。这个转化的通用性非常好，但它有一个缺点，在最坏情况下我们可能要新增 n 个点，这将使我们的数据规模（点数）翻倍，因此将会显著增大算法的常数。

4.3 转化二

我们考虑另一个做法，同样是将最大匹配问题转化为完美匹配问题，不过这次我们将通过删点来实现。对于一个图 $G = (V, E)$ 和它的一个最大匹配 M，我们考虑所有在匹配 M 中的点构成的点集 U，显然 $G[U]$ 有完美匹配，并且 $G[U]$ 的完美匹配就是 G 的最大匹配。现在我们的思路就是设法找出这样的点集 U，然后对 $G[U]$ 应用我们之前求解完美匹配的算法。

实际上定理 4.1 的证明过程已经为我们指明了方向。由引理 3.5 和推论 3.6，我们只需要找到 $\tilde{A}(G)$ 的一组关于行的极大线性无关组就可以了。这可以通过一遍高斯消元求出来。

于是我们就可以在 $O(n^\omega)$ 的复杂度内把最大匹配问题转化为完美匹配问题，并且它不会增大我们算法的常数。

5 应用

本文所讲述的是解决一般图最大匹配问题的通用算法，因此它自然可以解决所有用到了一般图匹配的题目。

除此之外，它还有一些较为有特色的应用。下面我们来看两个例子。

5.1 结点接龙[6]

5.1.1 问题描述

Alice 和 Bob 在一个无向图 $G = (V, E)$ 上面玩游戏。游戏由双方交替操作，其中 Alice 先手。

一开始有一个棋子放在某个结点上，接下来每一回合玩家可以将这个棋子移动到相邻并且之前没有到达过（含起点）的结点上，无法操作的一方输。问哪些结点先手必胜。

5.1.2 算法介绍

首先我们有一个结论：一个点 v_i 是必胜态当且仅当它一定在图 G 的最大匹配上。[7]

于是问题转化为判定某个点是否一定在图 G 的最大匹配上。我们按照第 4.2 节的做法新建 $k = n - 2\nu(G)$ 个点并连上相应的边。设这样得到的新图为 G'，新建的点的编号分别为 $n+1, n+2, \ldots, n+k$。然后我们任选一个新点 $v_j (n+1 \le j \le n+k)$，那么对于原图中的每个点 $v_i (1 \le i \le n)$，如果 $v_i v_j$ 能够出现在图 G' 的完美匹配中，那么 v_i 就不一定在图 G 的最大匹配上。

由定理 3.9，我们只需要检查 $(\tilde{A}(G')^{-1})_{i,j}$ 是否为零就可以了。

可以看到，本文介绍的匹配算法在判断 “一个点是否一定在最大匹配上” 以及 “一条边是否一定在最大匹配上” 这类问题上具有优势。因为这个算法可以避免对增广路的分析，而只需要检查逆矩阵的某个位置是否为零。

[6]经典问题，英文为 Undirected Vertex Geography (UVG)。

[7]这个结论的证明不是本文的重点，这里略去，感兴趣的读者可以参考相关资料。

5.2 最大权匹配

5.2.1 问题描述

所谓最大权匹配是指让图 G 的每条边 v_iv_j 带上一个权值 $w_{i,j}$，然后现在我们的目标是求一个边权和最大的匹配 M。最大匹配相当于边权均为 1 的最大权匹配。

5.2.2 算法介绍

我们不妨在每对点之间都连上权值为 0 的边，当 n 是奇数时我们再新增一个点，并让它向所有点连边。因此原图的每一个带权匹配都会对应新图的至少一个带权完美匹配，并且新图的一个带权完美匹配对应了原图的一个带权匹配。

现在我们转化为求新图 G' 的带权完美匹配。我们新增一个变量 y。设 $\tilde{A}'(G')_{i,j} = \tilde{A}(G')_{i,j} \times y^{w_{i,j}}$。这样我们相当于要求 $\det \tilde{A}'(G')$ 中，y 的最高次数（这个值除以 2 就是最大权匹配）。

我们先把所有的 $x_{i,j}$ 赋上一个随机的值。然后设所有边的权值和不超过 W，那么我们可以将 0 到 W 中的每个值代入 y，算一遍行列式，并用这 $(W+1)$ 个结果插值得到一个关于 y 的多项式。那么这个多项式的最高项次数就是我们要求的东西。

这个算法的时间复杂度为 $O(Wn^\omega + W^2)$，在 W 和 n 均较小的时候有一定的价值。

6 总结

至此，我们已经能够在 $O(n^\omega)$ 的复杂度内求出一般图最大匹配的大小，并且在 $O(n^3)$ 的复杂度内构造出一组匹配方案。

本文所介绍的算法仍旧有着巨大的潜力等待我们去探究，并且本文所介绍的应用还较为基础。因此我希望本文能够起到抛砖引玉的作用，希望有读者能够拓展本文中的算法，将这一算法发扬光大，得到更加有趣的应用。

致谢

- 感谢中国计算机学会提供学习和交流的平台；
- 感谢宋新波老师，卓明聪老师多年以来的关心和指导；
- 感谢国家集训队教练张瑞喆和余林韵的指导；
- 感谢周子鑫同学与我分享这个算法；
- 感谢高闻远同学、武弘勋同学为本文审稿；

- 感谢父母对我的关心和照顾。

参考文献

[1] Mucha M, Sankowski P. Maximum matchings via Gaussian elimination[C]//Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on. IEEE, 2004: 248-255.

[2] Ivan I, Virza M, Yuen H. Algebraic Algorithms for Matching[J]. 2011.

[3] Cheung H Y. Algebraic algorithms in combinatorial optimization[D]. The Chinese University of Hong Kong, 2011.

[4] Huang C C, Kavitha T. Efficient algorithms for maximum weight matchings in general graphs with small edge weights[C]//Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2012: 1400-1412.

[5] 陈胤伯. 浅谈图的匹配算法及其应用[C]//2015 年信息学奥林匹克中国国家队候选队员论文集. 2015.

多项式求和 命题报告

长沙市雅礼中学 袁宇韬

1 试题大意

给定n，m，a，b，c。定义$f_0(x) = 1$，对于$k > 0$，$f_k(x) = \sum_{i=0}^{x}(ai^2 + bi + c)f_{k-1}(i)$。对于$0 \le i < n$的整数$i$，求出$f_i(m)$模1004535809的值。

1.1 数据范围

数据编号	n	m	a	b	c
0	2000	≤ 2000	0	0	1
1	2000	≤ 2000	$\le 10^9$	$\le 10^9$	$\le 10^9$
2	300	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$
3	1000	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$
4	2000	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$
5	3000	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$
6	50000	$\le 10^9$	0	0	$\le 10^9$
7	250000	$\le 10^9$	0	0	$\le 10^9$
8	50000	≤ 2000	$\le 10^9$	$\le 10^9$	$\le 10^9$
9	250000	≤ 2000	$\le 10^9$	$\le 10^9$	$\le 10^9$
10	50000	≤ 50000	$\le 10^9$	$\le 10^9$	$\le 10^9$
11	250000	≤ 50000	$\le 10^9$	$\le 10^9$	$\le 10^9$
12	50000	$\le 10^9$	0	1	0
13	50000	$\le 10^9$	0	$\le 10^9$	$\le 10^9$
14	250000	$\le 10^9$	0	1	0
15	250000	$\le 10^9$	0	$\le 10^9$	$\le 10^9$
16	50000	$\le 10^9$	1	0	0
17	50000	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$
18	250000	$\le 10^9$	1	0	0
19	250000	$\le 10^9$	$\le 10^9$	$\le 10^9$	$\le 10^9$

2 算法介绍

2.1 算法一

根据定义递推。

时间复杂度$O(nm)$。期望得分10–15分。

2.2 算法二

多项式$f(x)$的前缀和是一个多项式，可以在$O(n^2)$或$O(n\log n)$时间内求出。依次求出所有多项式$f_i(x)$。

时间复杂度$O(n^3)$或$O(n^2\log n)$。与算法一同时使用，期望得分15–30分。

2.3 算法三

多项式$f_k(x)$的次数不超过$3k$，因此只需要递推求出$0 \le x \le 3k$的 $f_k(x)$，再插值求出答案。

时间复杂度$O(n^2)$。与算法一同时使用，期望得分30–35分。

2.4 算法四

当$a = b = 0$时$f_k(x) = \binom{x+k}{k}c^k$。可以递推求出。

时间复杂度$O(n)$。与算法三同时使用，期望得分40–45分。

2.5 算法五

令

$$g_k(x) = \sum_{i=0}^{\infty} f_i(k)$$

则由定义得到当$k > 0$时

$$g_k(x) = (ak^2 + bk + c)xg_k(x) + g_{k-1}(x)$$

即

$$g_k(x) = \frac{g_{k-1}(x)}{1 - (ak^2 + bk + c)x}$$

同时

$$g_0(x) = \frac{1}{1 - cx}$$

因此

$$g_k(x) = \prod_{i=0}^{k} \frac{1}{1-(ai^2+bi+c)x}$$

暴力或分治求出分母，时间复杂度为$O(m^2)$或$O(m\log^2 m)$。再求出多项式逆元得到答案。

时间复杂度$O(m^2+n\log n)$或$O(m\log^2 m+n\log n)$。期望得分45–60分。

2.6 算法六

令

$$E(x) = \prod_{i=0}^{k}\left(1-(ai^2+bi+c)x\right) = \sum_{i=0}^{k+1}(-1)^{k-i}e_{k-i}x^k$$

令

$$p_i = \sum_{j=0}^{k}(aj^2+bj+c)^i$$

由牛顿恒等式得到

$$e_i = \frac{1}{i}\sum_{j=1}^{i}(-1)^{j-1}e_{i-j}p_j$$

令

$$P(x) = \sum_{i=0}^{\infty}(-1)^i p_{i+1}$$

则得到$E'(x) = E(x)P(x)$。

可以分治FFT求出$E(x)$，每次用前一半的系数与$P(x)$卷积更新后一半的系数。

接下来考虑求出p_i。分为两种情况考虑。

当$a=0$，$b\neq 0$时，令$u=\frac{c}{b}$，则

$$p_i = b^i\sum_{j=0}^{k}(j+u)^i = b^i\left(\sum_{j=0}^{k+u}j^i - \sum_{j=0}^{u-1}j^i\right)$$

可以用Bernoulli数求出。

当$a\neq 0$时，令$u=\frac{b}{2a}$，$v=\frac{c}{a}$，则

$$\begin{aligned}
p_i &= a^i\sum_{j=0}^{k}(j^2+2uj+v)^i \\
&= a^i\sum_{j=0}^{k}\left((j+u)^2+(v-u^2)\right)^i \\
&= a^i\sum_{t=0}^{i}\binom{i}{t}(v-u^2)^{i-t}\sum_{j=0}^{k}(j+u)^{2t}
\end{aligned}$$

可以类似前一种情况求出$\sum_{j=0}^{k}(j+u)^{2t}$，然后再卷积得到答案。

时间复杂度$O(n\log^2 n)$。期望得分80分。

2.7 算法七

由$E'(x)=E(x)P(x)$，$E(0)=1$解得$E(x)=\exp\left(\int_0^x P(t)\,\mathrm{d}t\right)$。因此用一次多项式exp可以求出$E(x)$。

时间复杂度$O(n\log n)$。期望得分90–100分。

3 命题思路

求多项式序列的前缀和是一个很经典的问题。连续多次对一个多项式序列求前缀和可以用组合数求出通项公式，于是一个自然的扩展是在两次前缀和之间进行其它操作，在本题中即为与另一个多项式序列相乘。由于我没有找到高效的求出通项公式的算法，就改变了询问的内容，得到了本题。

4 总结

本题主要考察选手对生成函数的了解和多项式相关算法的运用，以及牛顿恒等式与基础微积分等数学知识。

5 感谢

感谢中国计算机学会提供学习和交流的机会。

感谢汪星明老师、朱全民老师对我的教导。

感谢帮助过我的同学们。

参考文献

[1] 金策, 《生成函数的运算与组合计数问题》, 2015年国家集训队论文。

[2] 彭雨翔, “Inverse Element of Polynomial”, http://picks.logdown.com/posts/189620-inverse-element-of-polynomial

[3] 彭雨翔, “Newton’s Method of Polynomial”, http://picks.logdown.com/posts/209226-newtons-method-of-polynomial

[4] 刘汝佳,《算法竞赛入门经典》, 清华大学出版社。

浅谈信息学竞赛中的独立集问题

福建省福州第一中学 钟知闲

寻求NP-Hard问题的较优算法是信息学的重要内容之一，其中，与独立集相关的问题较为常见，在信息学竞赛中也时常出现此类问题。本文将从独立集的性质入手，介绍基于多种思想的几类独立集算法。对于一般图，本文在搜索算法的基础上，提出了动态规划的优化算法，同时分析并测试了几种算法在随机数据下的运行效率；对于特殊图，本文介绍了两种针对特殊图的算法思想，提出了求解“k-仙人图”的独立集问题的算法思想及其扩展应用。

1 前言

不少与独立集有关的问题（最大独立集、最大带权独立集、独立集计数等问题）都是图论中经典的NP-Hard问题，在信息学竞赛中广泛出现，然而解决独立集问题的算法效率通常不高。由此，本人对此类问题进行了更深入的研究，希望能用更加高效的方法解决此类问题。

本文的研究分为两部分，第一部分介绍了两种求解一般图的独立集问题的算法：基于极大独立集搜索的独立集算法和基于动态规划的独立集算法，第二部分介绍了两类特殊图的独立集算法，分别基于图匹配思想和图上的阶段划分思想。

2 定义及约定

独立集问题有多种形式。为了方便描述，以下给出一些定义。

定义 2.1. 对于无向图 $G = (V, E)$ 和点 $u, v \in V$，若 $(u, v) \in E$，则称 u, v **相邻**（*adjacent*）；定义点 $v \in V$ 的**邻域**（*neighborhood*）为 V 中与 v 相邻的结点集合，记为 $N(v)$；另外，$N_G(v)$ 表示 v 在图 G 中的邻域。

定义 2.2. 点 v 的**度**（*degree*）$\deg(v)$ 定义为 $N(v)$ 的大小，即 $\deg(v) = |N(v)|$；另外，$\deg_G(v)$ 表示 v 在图 G 中的度。

定义 2.3. 无向图 $G = (V, E)$ 的一个**独立集**（*independent set*）定义为 V 的一个子集，满足子集中的结点两两不相邻。形式化地，I 是 G 的一个独立集，当且仅当 $I \subseteq V$ 且 $\forall u, v \in I$，$(u, v) \notin E$。

定义 2.4. 无向图 $G = (V, E)$ 的一个**最大独立集**（*maximum independent set*）是指 G 中所含结点数 $|I|$ 最多的独立集 I。

定义 2.5. 无向图 $G = (V, E)$ 的**独立数**（*independence number*）定义为 G 的最大独立集 I 所含的结点数 $|I|$，记为 $\alpha(G)$。

定义 2.6. 无向图 $G = (V, E)$ 在 $S \subseteq V$ 上的**导出子图**（*induced subgraph*）[8]定义为以 S 为点集，两端点都在 S 内的边为边集构成的图，记为 $G[S]$。

本文中的所有问题均以**最大独立集问题**为例，即对于给定的无向图 $G = (V, E)$，找出 G 的一个最大独立集 I。

为了方便起见，约定 n 为图 G 的阶数（即顶点数），m 为图 G 的边数，即 $n = |V|, m = |E|$。

3 一般图的独立集问题

目前，解决一般图的大多数独立集相关的问题都不存在多项式时间的算法，只能用复杂度较优的指数级算法。

事实上，已有不少理论复杂度十分优秀的求图的最大独立集的算法，能够快速计算出上百阶的无向图的最大独立集[9]，但这些算法实现往往过于复杂，难以应用到信息学竞赛中。笔者选择了一些相对高效又较易于实现的算法进行了研究。

3.1 基于极大独立集搜索的独立集算法

3.1.1 朴素的搜索算法

最朴素的搜索算法非常简单：用深度优先搜索枚举 V 的子集 $I \subseteq V$，即按一定顺序枚举每个点 $v \in V$ 是否属于 I，一旦存在 $(u, v) \in E$ 使得 $u, v \in I$，就回溯。输出枚举的所有独立集 I 中，$|I|$ 最大的一个。该算法的复杂度为 $O(2^n m)$，效率太低。

针对最大独立集这一问题，可以进行一些剪枝。例如：

1. 若 $\deg(v) = 0$，则不存在与 v 关联的边，故总可以令 $v \in I$。
2. 若 $\deg(v) = 1$，考虑唯一的与 v 关联的结点 u，若 $u \notin I$，则总可以令 $v \in I$；否则，从 I 中删去 u 并加入 v，I 的大小不变。因此总可以令 $v \in I$。

[8]导出子图有**点导出子图**和**边导出子图**两种，本文中均指前者。

[9]Robson在2001年提出了一种复杂度约 $O(1.1888^n)$ 的最大独立集算法，见参考文献[2]。

3. 搜索时记录当前搜到的独立集的大小的最大值 a，记 P 为 $V-I$ 中不与 I 中结点相邻的点集，当 $|I|+|P| \le a$ 时可进行最优性剪枝。

然而加入这些剪枝之后，复杂度仍然很高，难以接受。

3.1.2 极大独立集与Bron-Kerbosch算法

朴素的搜索算法效率太低，有没什么好的方法来优化呢？我们提出极大独立集的概念：

定义 3.1. *无向图 $G=(V,E)$ 的一个**极大独立集**（maximal independent set）是指 G 的一个独立集 I，满足对于任意的结点 $v \in V-I$，点集 $I+\{v\}$ 不是独立集。*

通常情况下，一个图的极大独立集个数比独立集个数少得多。例如 50 个结点构成的链有 $32,951,280,099$ 个独立集，却只有 $1,221,537$ 个极大独立集。另外，不少有关独立集的组合优化问题都可以只考虑极大独立集，最大独立集问题就是这样一个例子：

定理 3.1. *每个最大独立集都是极大独立集。*

证明. 设 I 是一个最大独立集。对于任意的 $v \in V-I$，假如 $I+\{v\}$ 是独立集，因为 $|I+\{v\}| = |I|+1 > |I|$，所以 $I+\{v\}$ 是一个比 I 更大的独立集，也就是说，I 不是最大独立集，与假设矛盾。所以 I 一定是一个极大独立集。

因此，如果能找出 G 的所有极大独立集，就能找出 G 的最大独立集。

求极大独立集的算法有很多，其中Bron-Kerbosch算法是实现较为简洁的一种。下面介绍求极大独立集的Bron-Kerbosch算法。

Bron-Kerbosch算法可以对任意的无向图 G 求出其所有的极大独立集，其基本思想仍然是搜索优化。伪代码如下[10]：

调用 BronKerbosch(R,P,X) 函数，将输出 G 的所有包含 R 中的所有结点、P 中的任意多个结点且不包含 X 中的结点的所有极大独立集。调用 BronKerbosch$(\varnothing,V,\varnothing)$ 即可得到 G 的所有极大独立集。

实现时，集合可以用压位的方法存储，即用一个大小为 $\lceil \frac{n}{64} \rceil$ 的 64 位整数数组 A 存一个大小为 n 的集合 A'，$x \in A'$ 当且仅当 $A[\lfloor \frac{x}{64} \rfloor]$ AND $2^{x \bmod 64} > 0$（A 数组下标从 0 开始）。因为独立集问题中，图的阶数 n 不会很大，所以压位的数组大小可以近似认为是一个常数。

上述算法的最关键之处在于Pivoting。算法过程中，有一步是选择结点 $u \in P \cup X$，使得 $|P \cap (\{u\} \cup N(u))|$ 最小，u 称为Pivot结点。之后枚举 $\{u\} \cup N(u)$ 中，属于独立集 R 的第一个结点 v。这就是Pivoting的过程。

为什么要这么做？

[10]Bron-Kerbosch算法有多种实现方式，本文介绍其中的一种。

算法 1 BronKerbosch(R, P, X)

```
if P = X = ∅ then
    print R
end if
选择结点 u ∈ P ∪ X，使得 |P ∩ ({u} ∪ N(u))| 最小
for all v ∈ P ∩ ({u} ∪ N(u)) do
    BronKerbosch(R ∪ {v}, P − ({v} ∪ N(v)), X − ({v} ∪ N(v)))
    P ← P − {v}
    X ← X ∪ {v}
end for=0
```

如果直接搜索极大独立集的话，效率是很低的，因为会搜到很多不是极大的独立集。例如当 G 为 n 阶零图[11]时，显然 V 是 G 的唯一的极大独立集，然而朴素的搜索枚举了某个结点不属于极大独立集时，尽管不可能搜出极大独立集，但算法还会继续搜索下去，浪费了大量时间。Pivoting的正确性基于以下定理：

定理 3.2. *对于无向图 $G = (V, E)$ 和 $v \in V$，G 的任意极大独立集 I 满足 $I \cap (\{v\} \cup N(v)) \neq \varnothing$。*

证明. 假设存在极大独立集 I，满足 $I \cap (\{v\} \cup N(v)) = \varnothing$，则对于任意 $u \in I$，$u \notin \{v\} \cup N(v)$，即 $u \neq v$ 且 $(u, v) \neq E$。

因此 $I \cup \{v\}$ 也是一个独立集，且 $I \subsetneq I \cup \{v\}$，这说明 I 不是极大独立集，矛盾。

这样，我们就证明了该定理的正确性。尽管这样仍然会搜到一些不是极大的独立集，但这样的集合显然少了很多。

3.1.3 极大独立集的个数

之前我们只是感性地认识了极大独立集比较少，这里将给出Bron-Kerbosch算法的递归次数上界：

定理 3.3. *Bron-Kerbosch算法的递归调用次数为 $O(3^{\frac{n}{3}})$。*

该定理的证明较为复杂，本文略去[12]。

由此可以得到推论：

定理 3.4. *n 阶无向图的极大独立集个数为 $O(3^{\frac{n}{3}})$。*

[11]**零图**定义为没有边的图，即 $G = (V, E)$ 为零图当且仅当 $E = \varnothing$。

[12]有兴趣的读者可以阅读参考文献[3]。

这个上界是很容易达到的，构造 $\lfloor \frac{n}{3} \rfloor$ 个相互独立的三元环即可。但在图随机生成的情况下，这个上界是很不满的。为了说明这一点，笔者对随机图的极大独立集个数进行了研究。

从边数为 m 的 n 阶简单无向图中随机生成一个图 $G = (V, E)$，记 G 的极大独立集个数 x，即

$$x = \sum_{S \subseteq V} [S \text{ is a maximal independent set}]$$

考虑计算 x 的期望值 $E(x)$。S 是 G 的极大独立集的条件为：

- S 是独立集，即对于任意的 $u, v \in S$，$(u, v) \notin E$；
- 对于任意的 $v \in V - S$，$V + \{v\}$ 不是独立集，即 $V - S$ 中的每个点至少与 S 中的一个点相邻。

用容斥原理，枚举 k 个 $V - S$ 中的点不与 S 中的点相邻。记 $i = |S|$，则剩下 $n - i - k$ 个点可以和 S 中的点连边，以及 $V - S$ 中任意两点（一共 $\frac{1}{2}(n-i)(n-i-1)$ 对点）可以连边。则满足 S 是极大独立集的图 G 个数为

$$\sum_{k=0}^{n-i} (-1)^k \binom{n-i}{k} \binom{(n-i-k)i + \frac{(n-i)(n-i-1)}{2}}{m}$$

由于边数为 m 的 n 阶简单图共有 $\binom{\frac{n(n-1)}{2}}{m}$ 个，故有

$$E(x) = \sum_{S \subseteq V} P([S \text{ is a maximal independent set}])$$

$$= \sum_{i=0}^{n} \sum_{S \subseteq V, |S| = i} \binom{\frac{n(n-1)}{2}}{m}^{-1} \sum_{k=0}^{n-i} (-1)^k \binom{n-i}{k} \binom{(n-i-k)i + \frac{(n-i)(n-i-1)}{2}}{m}$$

$$= \binom{\frac{n(n-1)}{2}}{m}^{-1} \sum_{i=0}^{n} \binom{n}{i} \sum_{k=0}^{n-i} (-1)^k \binom{n-i}{k} \binom{(n-i-k)i + \frac{(n-i)(n-i-1)}{2}}{m}$$

下面给出了一些计算结果（四舍五入）：

$E(x)$	$m=n$	$m=\lfloor\sqrt{3}n\rfloor$	$m=2n$	$m=3n$	$m=\frac{n^2}{4}$
$n=20$	84	101	99	81	49
$n=30$	706	933	909	691	157
$n=40$	5.95×10^3	8.67×10^3	8.40×10^3	5.88×10^3	403
$n=50$	5.02×10^4	8.07×10^4	7.76×10^4	5.01×10^4	891
$n=60$	4.23×10^5	7.51×10^5	7.18×10^5	4.27×10^5	1779
$n=70$	3.57×10^6	6.99×10^6	6.64×10^6	3.63×10^6	3291
$n=80$	3.01×10^7	6.51×10^7	6.14×10^7	3.10×10^7	5730
$n=90$	2.54×10^8	6.06×10^8	5.68×10^8	2.64×10^8	9506
$n=100$	2.15×10^9	5.64×10^9	5.25×10^9	2.25×10^9	15154

可见随机情况下，极大独立集的个数远少于 $3^{\frac{n}{3}}$。另外，当 m 接近 $\sqrt{3}n$ 时，独立集个数 x 的期望值 $E(x)$ 最大，而过于稠密的图，独立集个数相当少。

根据该结论，还可以进一步得出，$x\geq k\cdot E(x)$ 的概率不超过 $\frac{1}{k}$，所以在大多数情况下随机图的极大独立集个数不会比期望值大太多[13]。

值得注意的是，Bron-Kerbosch算法复杂度不和极大独立集个数直接相关，所以用极大独立集个数的期望值分析Bron-Kerbosch算法的期望运行时间并不准确；事实上，存在复杂度和极大独立集个数直接相关的极大独立集搜索算法，但超出了本文的范围，有兴趣的读者可以自行了解。

3.1.4 应用

介绍了极大独立集的性质及算法之后，我们来看看它有哪些应用。

例 1. (图的3-染色问题)给定 n 阶简单无向图 $G=(V,E)$，用三种颜色对 V 中的结点进行染色，使得每条边 $(u,v)\in E$ 的两端点 u,v 颜色不同。$n\leq 40$。

图的3-染色问题也是著名的NP-Hard问题。朴素的算法是每次枚举一个与已确定颜色的结点相邻的结点颜色，需要枚举 $O(2^n)$ 种情况，无法通过 $n=40$ 的数据。如何才能更加高效地求解？

先给出一个定理：

定理 3.5. *无向图 $G=(V,E)$ 能够3-染色的充要条件是 G 存在一个极大独立集 I，使得图 $G-I$ 是二分图*[14]。

证明. (充分性)设 I 为 G 的一个极大独立集，且 $G-I$ 为二分图，根据二分图的性质，存在点集 $X\subseteq V-I$，记 $Y=V-I-X$，使得对任意 $u,v\in X$ 或 $u,v\in Y$，有 $(u,v)\notin E$。

[13]由于笔者并没有对 x 的方差进行研究，无法得到更准确的对 x 分布的描述。

[14]无向图 $G=(V,E)$ 是**二分图**（bipartite graph）定义为可以将 V 划分为两个集合 S 和 $V-S$，使得每条边的两个端点不在同一个集合内，即 $\forall(u,v)\in E$，$u\in S,v\in V-S$ 或 $u\in V-S,v\in S$。

因为 I 是 G 的独立集，所以 $\forall u,v \in I$，有 $(u,v) \notin E$。

因此将 I 中的点染色为1，X 中的点染色为2，Y 中的点染色为3是一种合法方案。

(必要性)设 $G=(V,E)$ 能够3-染色，记 X,Y,Z 分别为染颜色1,2,3的点集。

由定义，对任意 $(u,v) \in E$，结点 u,v 不属于这三个集合中的同一个集合，因此 X,Y,Z 都是独立集。

如果 X 不是极大独立集，则存在 $v \in V-X$，使 $X+\{v\}$ 是独立集。将 v 加入点集 X，同时，若 $v \in Y$，则将 v 从 Y 中删去；否则 $v \in Z$，将 v 从 Z 中删去。重复此过程直至 X 是极大独立集为止。

显然此时 Y,Z 仍然是 G 的独立集，即对于 $u,v \in Y$ 或 $u,v \in Z$，有 $(u,v) \notin E$，故 $G[Y \cup Z]$ 是二分图。问题得证。

判断图是否为二分图以及将其进行染色可以在 $O(m)$ 的时间内解决。因此只需用枚举图 G 的所有极大独立集 I，然后判断图 $G-I$ 是否为二分图：

1、若对所有的极大独立集 I，图 $G-I$ 都不是二分图，则图 G 不能3-染色。

2、若存在一个极大独立集 I，使得图 $G-I$ 是二分图，则图 G 能3-染色：将 I 中的结点用颜色1染色，$G-I$ 用颜色2和3进行二分图染色即可。

本题中，由于 G 是简单图，$m=O(n^2)$，所以该算法时间复杂度为 $O(3^{\frac{n}{3}}n^2)$。

例 2. (小Q运动季 测试点10)给定一个 n 元一次同余方程组

$$\begin{cases} a_{0,0}x_0+a_{0,1}x_1+...+a_{0,n-1}x_{n-1} \equiv c_0 \pmod{b_0} \\ a_{1,0}x_0+a_{1,1}x_1+...+a_{1,n-1}x_{n-1} \equiv c_1 \pmod{b_1} \\ ... \\ a_{m-1,0}x_0+a_{m-1,1}x_1+...+a_{m-1,n-1}x_{n-1} \equiv c_{m-1} \pmod{b_{m-1}} \end{cases}$$

求一组解 $(x_1,x_2,...,x_n)$ 满足尽量多的方程。提交答案题。[15]

本例中仅讨论测试点10。该测试点中，通过建立图论模型，将每个方程看成一个点，相互冲突的方程间连一条边，可以转化为点数 $n=90$，边数 $m=223$ 的无向图的最大独立集问题。由于具体转化过程超出了本文的范围，故略去。

用Bron-Kerbosch算法搜出所有极大独立集，输出其中最大的一个即可。这样做的效率如何呢？

笔者将朴素搜索算法Simple-Search和基于极大独立集的搜索算法Maximal-Search进行比较，两个算法仅使用了最基本的剪枝：将剩余的点全部加入 I 都不大于当前搜到的点集，得到的点集大小都不超过当前搜到的最大的独立集，则剪枝。由于仅仅测试的是Maximal-Search是否有比Simple-Search更优的运行效率，这里并没有加入更多依赖问题性质的剪枝。

对于Simple-Search，笔者的程序经过运行若干小时，仍然只能得到大小为 33 的独立集，并且程序未能结束。然而对于Maximal-Search，笔者的程序仅用不到1min就得到了一

[15]题目来源：WC2013 第3题

组大小为 34 的独立集，仅用3min就证明了图的独立数确实为 34。可见用极大独立集进行搜索确实能大幅提高运行效率。

通过加入更多的剪枝优化，可以进一步缩短算法运行时间，有兴趣的读者可以尝试。

3.2 基于动态规划的独立集算法

动态规划是一种高效、灵活的处理问题的方法。在独立集问题中，动态规划不仅能求解最优化类问题（如最大独立集、最大权独立集），还能求解计数类问题（如独立集计数）。下面仍以最大独立集问题为例，但为了体现动态规划的通用性，接下来的讨论将不加入最优性剪枝等仅针对最优化问题的剪枝。

3.2.1 算法

如果要使用动态规划求解独立集问题，就需要将问题化为规模更小的子问题。对于独立集，我们有以下两个定理：

定理 3.6. *对于无向图 $G=(V,E)$ 和 $V'\subseteq V$，则对于任意 $I\subseteq V'$，I 是 G 的独立集当且仅当 I 是 $G[V']$ 的独立集。*

证明. (充分性)当 I 是 $G[V']$ 的独立集时，对于任意 $(u,v)\in E$，若 $u,v\in V'$，显然 u,v 不同时属于 I；若 u,v 有一个不属于 V'，不妨设 $u\notin V'$，那么 $u\notin I$。因此 I 是 G 的独立集。

(必要性)当 I 是 G 的独立集时，由于 $G[V']$ 的每条边都属于 G，故 $G[V']$ 的每条边至少有一个端点不属于 I。因此 I 是 $G[V']$ 的独立集。

定理 3.7. *对于无向图 $G=(V,E)$ 和 $v\in V$，若 $I\subseteq V$ 且 $v\in I$，则 I 是 G 的独立集当且仅当 $I-\{v\}$ 是 $G[V-\{v\}-N(v)]$ 的独立集。*

证明. 记 $T=V-\{v\}-N(v)$。

(充分性)当 I 是 G 的独立集时，$N(v)\cap I=\varnothing$，所以 $I-\{v\}\subseteq T$，因为 I 在 G 中任意两点不相邻，且 $I-\{v\}\subseteq I$，$G[T]\subseteq G$，所以 $I-\{v\}$ 是 $G[T]$ 的独立集；

(必要性)当 $I-\{v\}$ 是 $G[T]$ 的独立集时，因为 $I-\{v\}\subseteq V-\{v\}-N(v)$，所以 $N(v)\cap I=\varnothing$，又因为 G 比 $G[T]$ 多的边均与 v 或 $N(v)$ 中的结点相邻，所以 I 是 G 的独立集。

根据以上两个定理，我们可以用状态压缩的动态规划（DP）对于任意的无向图 $G=(V,E)$ 求出 G 的独立数 $\alpha(G)$。

对点集 $S\subseteq V$，定义 $f(S)$ 为 S 在 G 上的导出子图的独立数，即 $f(S)=\alpha(G[S])$，显然 $f(\varnothing)=0$。

考虑 $S\neq\varnothing$ 的情况。任取 $v\in S$，考虑一个点集 $I\subseteq S$。若 $v\notin I$，则 I 是 $G[S]$ 的独立集当且仅当 I 是 $G[S-\{v\}]$ 的独立集；若 $v\in I$，则 I 是 $G[S]$ 的独立集当且仅当 $I-\{v\}$ 是 $G[S-\{v\}-N(v)]$ 的独立集。由此可得：

$$f(S)=\begin{cases}0, & S=\varnothing\\ \max\{f(S-\{v\}), f(S-\{v\}-N(v))+1\}, \forall v\in S, & S\neq\varnothing\end{cases} \tag{3.1}$$

实现时，将图中结点编号为$0,1,\cdots,n-1$，结点v可以选取S中编号最大的点。同样可以使用压位技巧来存集合S。另外，计算f可以使用记忆化搜索，状态可以用Hash Table来存储。

该算法不仅能求出独立数，还能求出一个最大独立集，见以下伪代码（f函数为依照(3.1)式定义的记忆化搜索函数）：

算法 2 Subset-Dynamic-Programming

```
S = V
I = ∅
while S ≠ ∅ do
    令 v 为 S 中编号最大的点
    if f(S − {v}) > f(S − {v} − N(v)) + 1 then
        S ← S − {v}
    else
        S ← S − {v} − N(v)
        I ← I + {v}
    end if
end while
return I =0
```

如果直接实现，复杂度为$O(2^{\frac{n}{2}})$（设Hash Table的的单次操作时间为$O(1)$），因为在前$\frac{n}{2}$层递归中，每层递归最多2个分支，而递归超过$\frac{n}{2}$层之后，S中只包含编号前$\frac{n}{2}$的结点，从而总复杂度为$O(2^{\frac{n}{2}})$。

对于任意的n，都能构造出使得该算法复杂度为$\Theta(2^{\frac{n}{2}})$的图G，方法是：将结点 0 和$\lfloor\frac{n}{2}\rfloor$连边，结点 1 和$\lfloor\frac{n}{2}\rfloor+1$连边，……，结点$\lfloor\frac{n}{2}\rfloor-1$和$2\lfloor\frac{n}{2}\rfloor-1$连边。这样递归的前$\lfloor\frac{n}{2}\rfloor$层每层都有2个分支，且所有的分支都不同。

例 3. (团的计数)给定无向简单图$G=(V,E)$，求G有多少个团。一个团定义为一个点集$S\subseteq V$，满足S中任意两点都有边相连。$n\leq 50$。

记$\overline{G}$为G的补图，不难发现，S是G的团当且仅当S是$\overline{G}$的独立集。

这是因为，如果S是G的团，那么S中的点在G中两两相邻，故在$\overline{G}$中两两不相邻，即S是$\overline{G}$的独立集。反之，如果S不是G的团，即存在两点u,v在G中不相邻，则u,v在$\overline{G}$中相邻，也就是说，S不是$\overline{G}$的独立集。

因此，问题转化为独立集计数问题——求 G 的独立集个数。显然这类计数问题无法用搜索优化的策略，不过，使用上述的Subset-Dynamic-Programming算法即可在 $O(2^{\frac{n}{2}})$ 时间内解决问题。

3.2.2 效率优化

经过实际测试，Subset-Dynamic-Programming算法运行效率不如优化的搜索算法。为什么？考虑最大独立集问题，该算法的最坏复杂度为 $O(2^{\frac{n}{2}})$，但搜索剪枝时可以直接处理度为1的结点，从而只需要 $O(n)$ 的时间。是否可以类比优化搜索算法，用一些“剪枝”来优化上述DP呢？

答案是肯定的。不过笔者并不打算重复之前的优化——既然用了基于DP的算法，就应当研究更加通用的优化。例如在求最大权独立集时，直接处理度为1的结点是错误的。经过笔者研究，以下优化可以大大提高DP的效率：

1、在状态转移方程中，结点 v 不取 S 中编号最大的点，而取 $G[S]$（注意是导出子图，而不是原图）中度数最大的点；

2、当图 $G[S]$ 不连通时，记每个连通块的点集分别为 $S_1, S_2, \cdots, S_k$，由于每个连通块是独立的，可以转化为规模更小的子问题解决：

$$f(S) = \sum_{i=1}^{k} f(S_i)$$

3、当图 $G[S]$ 不含环时，可以改用树形DP求解。

优化后的DP算法记作Optimized-Subset-Dynamic-Programming。笔者无法分析该算法在最坏情况下的复杂度是多少，但通过对随机图的测试，Optimized-Subset-Dynamic-Programming比之前的Subset-Dynamic-Programming快得多，其中优化1、2效果明显，尤其对于较稀疏的图。这是因为较稀疏的图在不断删去度数大的点时，导出子图 $G[S]$ 很容易不连通。

例 4. (小Q运动季 测试点10)题意见例题2。

上文已经提到了用Maximal-Search求得该问题的最优解所需的时间。现在我们尝试使用基于动态规划的独立集算法Subset-Dynamic-Programming，遗憾的是，测试表明，这个算法运行效率并不高，并且由于状态数过多，空间消耗都无法接受。

我们再试一试Optimized-Subset-Dynamic-Programming。令人惊讶的是，这个算法的运行效率非常高——经过笔者测试，该算法只用了 1s 就得到了最优解（大小为 34 的独立集）！并且这个算法运行过程中没有使用依赖任何问题的特殊性的优化（如最优性剪枝）。可见在稀疏图上，Optimized-Subset-Dynamic-Programming 的确是一个优秀的算法。

3.2.3 与搜索算法的联系

事实上，如果把这个算法的记忆化去掉（即不用Hash Table存储 f 值），就是一个带优化的搜索算法。这样的搜索算法（记为Optimized-Search）仍然比朴素的搜索算法Simple-Search快，但慢于Optimized-Subset-Dynamic-Programming。

Optimized-Subset-Dynamic-Programming结合了搜索和动态规划的优化思想，不仅有比较强的通用性，实现难度也很小。

然而Optimized-Subset-Dynamic-Programming有一定的缺陷：空间复杂度比较大。而搜索算法Optimized-Search的空间是多项式级别的，支持运行较长时间。因此可以只记忆化较小的 S 的 $f(S)$ 值，剩余部分采用搜索的方法，这样就能在较低的空间需求下解决问题了。

3.2.4 测试与对比

笔者在研究出上述动态规划算法及优化之后，将该算法（及优化后的算法）和之前的基于搜索的算法进行了实现，并且用随机图测试了这些算法的期望运行效率。下表的第一行中，n,m 代表 n 阶 m 边随机图，最后一列 $90,223$ 代表WC2013《小Q运动季》的测试点10对应的图。表格内的时间代表算法在对应的图上的期望运行时间估计值，“-”表示运行时间过长，未测出。

算法	$40,60$	$50,85$	$60,120$	$90,223$
Simple-Search	2s	–	–	–
Maximal-Search	<0.01s	<0.1s	1s	–
Subset-Dynamic-Programming	<0.01s	<0.1s	1s	–
Optimized-Subset-Dynamic-Programming-1	<0.01s	<0.01s	<0.01s	1s
Optimized-Subset-Dynamic-Programming-2	<0.01s	<0.01s	<0.01s	1s

Maximal-Search和Subset-Dynamic-Programming分别采用了“搜索剪枝”和“记忆化”的优化，其效果比较接近，Optimized-Subset-Dynamic-Programming则结合了两者的优势，效率严格高于这两个算法。值得一提的是，Optimized-Subset-Dynamic-Programming-1和Optimized-Subset-Dynamic-Programming-2的区别在于后者加入了优化3（转为树形DP），尽管优化3看起来很高效——把指数级的问题用线性时间解决，但经过测试，两者运行效率几乎无差别。

4 特殊图的独立集问题

上文介绍了解决一般图的独立集的基本思想（搜索优化以及动态规划），这些方法复杂度均为指数级。本节中，我们将进一步探讨特殊图的独立集问题——当图本身具有一定特殊性质时，能否用多项式复杂度解决同样的问题？

4.1 基于图匹配思想的最大独立集算法

4.1.1 二分图的最大独立集

二分图的最大独立集是一个经典问题。我们有以下定理：

定理 4.1. 对于 n 阶二分图 G，$\alpha(G) = n - \nu(G)$，其中 $\alpha(G), \nu(G)$ 分别为图 G 的独立数和匹配数。

该定理的证明可以在很多材料中找到，故证明略。

用匈牙利算法或网络流求出二分图 $G = (X, Y, E)$ 的一个匹配数 $\nu(G)$，即可得到 G 的独立数 $\alpha(G)$。另外，用网络流建图后求最小割可以得到一个最大独立集 I。

这个算法只能解决最优化类的独立集问题，不能解决更加复杂的问题（如计数或带有其它限制等）。

4.1.2 无爪图的最大独立集

二分图的最大独立集给了我们一个思路——求图的最大匹配时，可以通过找增广路不断增加匹配大小，那么求其它图的最大独立集能否也采用增广的方式？遗憾的是，在任意图上，两个独立集的对称差[16]的导出子图不一定是若干条路径或环，所以并不能用找增广路的方法求最大独立集。

不过，在一种特殊的图上，这种算法是可行的。

定义 4.1. 无爪图（*claw-free graph*）定义为所有导出子图都不是 $K_{1,3}$ 的无向图。其中 $K_{1,3}$ 称为**爪**（*claw*），即两部分别含有 1 个点和 3 个点的完全二分图。

无爪图的最大独立集可以用类似一般图匹配的算法来求解。该算法依赖于以下定理：

定理 4.2. 设 I_1, I_2 为无爪图 G 的两个独立集，则 $G[I_1 \Delta I_2]$ 的每一个连通块都是一条简单路径或简单环。

证明. 设 $v \in I_1$，则 $N(v) \cap I_1 = \varnothing$，在 $G[I_1 \Delta I_2]$ 中，v 的度数为 $|N(v) \cap I_2|$。

假设存在三个不同的点 $v_1, v_2, v_3 \in N(v) \cap I_2$，因为 I_2 是独立集，所以 v_1, v_2, v_3 两两不相邻，因此 $G[\{v, v_1, v_2, v_3\}]$ 是一个爪，矛盾。

因此 $|N(v) \cap I_2| \le 2$，即 $G[I_1 \Delta I_2]$ 的所有点度数均不超过 2，定理得证。

注意到如果 $|I_1| < |I_2|$，那么 $G[I_1 \Delta I_2]$ 必然存在一个连通块 C，满足连通块中属于 I_2 的结点比属于 I_1 的结点多。由于 C 中属于 I_1, I_2 的结点交替出现，当 C 为简单环时，C 中属于 I_1, I_2 的结点一样多，而当 C 为简单路径时，C 中属于 I_1, I_2 的结点个数相差 1，故必然存

[16]集合 A, B 的对称差 $A \Delta B = \{x | [x \in A] \neq [x \in B]\}$

在一条路径满足属于 I_2 的点数比属于 I_1 的点数多 1。我们把 C 称为 I_1 的**增广路**（augment path）。

这样，就可以类比一般图最大匹配的算法，用增广路算法求无爪图 $G = (V, E)$ 的最大独立集：初始时令 $I = \varnothing$，每次从一个点出发找一条增广路，然后将增广路上的点状态取反，即：原来不属于独立集的点加入独立集，原来属于独立集的点从独立集中删去。该算法的实现类似Edmonds的带花树算法，实现细节及正确性证明见参考文献[4]，这里不再赘述。

4.2 基于图上阶段划分思想的最大独立集算法

4.3 分层图上的动态规划

对于图 $G = (V, E)$，将点集 V 划分为 k 个不相交的集合 $V_1, V_2, \cdots, V_k$，使得对任意 $u \in V_i, v \in V_j$，若 $|i - j| > 1$，则 $(u, v) \notin E$，则称集合序列 $< V_1, V_2, \cdots, V_k >$ 是 G 的一个分层。如果每个 V_i 中的结点都不多，那么可以按 $V_1, V_2, \cdots, V_k$ 顺序进行决策，在每个阶段只需状压一个层的选取情况即可，效率远高于一般图中的对整个图状压DP。

记 $f(i, S)$ 为图 $G[V_1 \cup V_2 \cup \cdots \cup V_i]$ 中包含 S 为子集的最大的独立集，状态转移方程如下：

$$f(i,S) = \begin{cases} -\infty, & S \text{ is not independent,} \\ 0, & i = 0, \\ \max\{f(i-1,S') | S' \subseteq V_{i-1}, S \cup S' \text{ is independent}\} + |S'|, & \text{otherwise} \end{cases}$$

G 的独立数为所有 $G[V_k]$ 的独立集 I 中 $f(k, I)$ 的最大值。该过程同样能求出一个 G 的最大独立集，方法和之前类似，这里不再赘述。

这个算法的时间复杂度为 $O(\sum_{i=1}^{k-1} 2^{|V_i|+|V_{i+1}|})$，不过在大多数情况下，每个 V_i 内的独立集个数并不多，实际的时间效率远高于理论上界。

4.4 “k-仙人图”上的动态规划

例 5. 给定简单无向图 $G = (V, E)$，保证每条边属于且仅属于一个简单环，求 G 的独立数。$|V| \leq 50,000$，$|E| \leq 60,000$。[17]

如果 G 不连通，那么求出 G 的每个连通块的独立数并求和即可。下文假设 G 是连通图。每条边最多属于一个简单环的简单连通图称为“仙人掌”，它有什么特殊的性质呢？

[17]题目来源：LYDSY 4316

我们不妨大胆尝试一下——任选一个 $r \in V$，以 r 为根对图 G 进行**深度优先搜索**（depth-first search, DFS），得到一个深度优先搜索树（DFS树）$T = (V, E_T)$。显然 T 是 G 的一个生成树。定义树边为属于 E_T 的边，非树边为不属于 E_T 的边（即属于 $E - E_T$ 的边）。

DFS树有一个很重要的性质：

定理 4.3. *对任意非树边 $e = (u, v)$，在 T 中或者 u 是 v 的祖先，或者 v 是 u 的祖先。*

证明. 设 $e = (u, v)$ 为非树边，且 u 比 v 先访问到，则访问到 u 时，假如 v 不在以 u 为根的子树内，那么枚举到 u 的出边 e 时，v 未被访问，因此下一步将沿着边 e 访问到 v，从而 $e \in E_T$，与假设矛盾，从而 u 是 v 的祖先。同理，若 v 比 u 先访问到，则 v 是 u 的祖先。

对于非树边 $e = (u, v)$，若树边 e' 在树 T 中从 u 到 v 的简单路径上，则称树边 e' 被非树边 e **覆盖**（cover）。对于仙人掌，我们有：

定理 4.4. *每条树边最多被一条非树边覆盖。*

证明. 假设一条树边 e 被两条非树边 $(u_1, v_1), (u_2, v_2)$ 覆盖，则 (u_1, v_1) 和 T 中 v_1 到 u_1 的简单路径构成一个简单环 C_1，(u_2, v_2) 和 T 中 v_2 到 u_2 的简单路径构成一个简单环 C_2，而 e 同时属于 C_1 和 C_2，与仙人掌的定义矛盾。因此 e 最多被一条非树边覆盖。

这个性质使得我们可以对树 T 进行树形动态规划。初步的想法是：记 $f(i, 0)$ 为 T 中以结点 $i \in V$ 为根的子树内最大的独立集大小，$f(i, 1)$ 为 i 的父结点属于独立集的情况下，i 子树内最大的独立集大小，然而这样不能保证非树边的两端点不同时属于独立集。

考虑添加一维状态，记 $f(i, j, k)$ 为 i 的父结点属于（$j = 1$）或不属于（$j = 0$）独立集，覆盖 i 与其父结点 e_i 连边的非树边 $e'_i = (u_i, v_i)$ 顶端结点 u_i 属于（$j = 1$）或不属于（$j = 0$）独立集的情况下，i 子树内最大的独立集大小。当 e_i 不属于环时，$k = 0$。转移时枚举 i 是否属于独立集即可，注意当 i 是 e'_i 的底端结点 v_i 且 $k = 1$ 时，不能选 i。状态转移方程如下（Ch_i 表示 i 的子结点集合）：

- 当 $j = 1$ 或 $k = 1 \wedge i = v_i$ 时：

$$f(i, j, k) = \sum_{c \in \mathrm{Ch}_i} f(c, 0, [e'_c = e'_i]k)$$

- 否则：

$$f(i, j, k) = \max\{\sum_{c \in \mathrm{Ch}_i} f(c, 0, [e'_c = e'_i]k), 1 + \sum_{c \in \mathrm{Ch}_i} f(c, 1, [e'_c = e'_i]k + [u_c = i])\}$$

用 $O(m)$ 时间预处理所有 u_i, v_i 之后就可以用上述 $O(n)$ 的动态规划求出最大独立集了，时间复杂度 $O(n + m)$。

笔者在研究上述算法之后，思考这种算法能否进行扩展。经过分析，笔者发现，将该算法做一些简单的修改之后，不仅能处理仙人掌，还能处理每条边所属的简单环个数不多的图。

我们把这样的图称为“k-仙人图”[18]，即每条边最多属于 k 个简单环的图，其中 k 的值比较小。

求解k-仙人图 $G = (V, E)$ 的独立集问题时，同样先取一个点为根对图进行DFS，得到DFS树 T。接下来对每个点 i，记 C_i 为覆盖 i 与其父结点的连边 e_i 的非树边集合，则有一个性质：

定理 4.5. *在k-仙人图中，对于任意的 $i \in V$，$|C_i| \le k$。*

证明. 对于每个 $(u, v) \in C_i$，(u, v) 和 T 中从 u 到 v 的路径都对应了一个包含 e_i 的简单环，因此必然存在 $|C_i|$ 个包含 e_i 的简单环。

由于包含 e_i 的简单环个数不超过 k，故 $|C_i| \le k$。

接下来，我们利用这个性质设计动态规划。在状态中，需要记录 C_i 中所有边的顶端是否属于独立集，转移方式类似。具体地，记 $f(i, j, S)$ 为 i 的父结点属于（$j = 1$）或不属于（$j = 0$）独立集，且 C_i 内的边的顶端结点构成的集合 U_i 中属于独立集的结点集合为 S 的情况下，T 中以 i 为根的子树内最大的独立集大小，则状态转移方程如下：

- 当 $j = 1$ 或 $S \cup \{i\}$ 不是独立集时：

$$f(i, j, S) = \sum_{c \in \mathrm{Ch}_i} f(c, 0, S \cap U_i)$$

- 否则：

$$f(i, j, S) = \max\{\sum_{c \in \mathrm{Ch}_i} f(c, 0, S \cap U_i), 1 + \sum_{c \in \mathrm{Ch}_i} f(c, 1, (S \cup \{i\}) \cap U_i)\}$$

和之前的算法类似，当 k 视为常数时，该算法的复杂度为 $O(n)$。

事实上，“k-仙人图”这个条件过于宽松，只要满足覆盖每条树边的非树边数目均不超过 k（甚至只要 $\sum_{v \in V} 2^{|C_v|}$ 不大），这个算法的效率都是很高的。

该算法可以拓展到更复杂的问题，如独立集计数。

例 6. (子集计数问题测试点7,8)给定无向图 $G = (V, E)$，$|V| = n$，$|E| = m$。求有多少个子集 $V' \subseteq V$ 满足 $|V'| = k$ 且 $\forall (u, v) \in E, u \notin V' \vee v \notin V'$。由于答案可能很大，只需输出答案对 p 取模的结果。提交答案题。[19]

本例中仅讨论测试点7,8。这两个测试点满足 G 是连通图，数据规模如下表：

[18]命名来源：LYDSY 4863

[19]题目来源：原创，UOJ #259

测试点编号	$n=$	$m=$	$k=$	$p=$
7	4998	5002	666	1000000009
8	11986	12011	1098	1000000007

问题求的是 G 中大小为 k 的独立集个数。由于 G 是连通图，且 $m-n$ 很小，可以发现 G 可以通过往一个树中加入 $m-n+1$ 条边得到。

如果构出 G 的DFS树之后，暴力枚举每条非树边的一端点是否选取，然后用树形动态规划统计独立集个数，可以较快通过测试点7，但测试点8需要运行的时间太长，需要优化。根据本节中提到的思想，每条边所属的简单环的个数不多，因此可以采用上述算法解决。

记

- C_i 为满足 $e\in E'$，u_e 为 i 的祖先（不含 i），v_e 在以 i 为根的子树内的 u_e 集合
- $f(i,j,S,s)$ 为满足 C_i 中属于独立集的结点集合为 S 的前提下，i 的前 j 个子树中，大小为 s 的独立集个数
- $g(i,j,S,s)$ 为满足 C_i 中属于独立集的结点集合为 S 的前提下，i 以及 i 的前 j 个子树中，大小为 s 的独立集个数
- $F(i,S,s)=f(i,t_i,S,s)$，$G(i,S,s)=g(i,t_i,S,s)$，这里 t_i 为 i 的子节点数

对于 $f(i,j,S,s)$，设 i 和 i 的前 $j-1$ 个子树内共有 s_1 个点，第 j 个子树内有 s_2 个点，第 j 个子节点为 c_j，则

$$f(i,j,S,s)=\sum_{x=\max\{s-s_1,0\}}^{\min\{s,s_2\}} f(i,j-1,S,s-x)G(c_j,S\cap C_{c_j},x),0\le s\le s_1+s_2$$

对于 $g(i,j,S,s)$，如果存在边 $e\in E'$ 使得 $u_e\in S$ 且 $v_e=i$，那么 i 不能属于独立集，有

$$g(i,j,S,s)=f(i,j,S,s)$$

否则，存在包含 i 的独立集，这样的独立集个数记为 $g'(i,j,S,s)$，则

$$g'(i,j,S,s)=\sum_{x=\max\{s-s_1,0\}}^{\min\{s-1,s_2\}} g'(i,j-1,S,s-x)F(c_j,(S\cup\{i\})\cap C_{c_j},x),0\le s\le s_1+s_2$$

所以

$$g(i,j,S,s)=g'(i,j,S,s)+f(i,j,S,s),0\le s\le s_1+s_2$$

边界为 $f(i,0,S,0)=g'(i,0,S,1)=1$，未定义的状态值均为 0。答案就是 $G(r,\varnothing,k)$。

可以只计算满足 $s\le k$ 的状态，复杂度 $O(k\sum_{v\in V}2^{|C_v|})$，其中 $|C_v|$ 通常在 12 以内。整个计算都在模 p 意义下进行即可，内存可以动态分配。

值得注意的是，选取不同的点 $r \in V$ 当根，以及用不同的顺序进行DFS，运行效率是不同的。可以选择一个根 r 进行DFS，使得

$$\sum_{v \in V} 2^{|C_v|}$$

最小，然后再执行上述算法，以降低时间复杂度。经过实测，测试点7的状态数约为 5×10^5，可以在1s内通过该测试点；测试点8的状态数约为 10^8，可以在2min内通过该测试点。

5 总结

NP-Hard问题的算法优化方法数不胜数，本文仅仅提到了若干种独立集问题的优化算法，这些方法解决的问题相类似，但思想各有区别——针对普通的最优化问题（如最大独立集），可以用带最优性剪枝的搜索算法减少枚举量；针对计数或有额外约束的问题（如独立集计数），可以用状态压缩动态规划，通过优化状态数来提高运行效率；针对可“增广”的图以及具有明显阶段性的图，又可以用多项式复杂度的算法来高效完成。

同时，本文对几种算法在随机情况下的运行时间进行了分析和比较，让大家对独立集问题求解的效率有更进一步的认识。

6 感谢

感谢中国计算机学会提供学习和交流的平台。

感谢福州一中的陈颖老师的关心与指导。

感谢福州一中的各位学长带给我的启发和指导。

感谢OI教练组带给我的帮助。

感谢陈俊锟同学为我提供了不少优化独立集算法的技巧，对我研究DFS树上的DP有很大启发。

感谢其他所有本文所参考过的资料的提供者。

感谢各位在百忙之中抽出宝贵时间阅读。

参考文献

[1] Wikipedia, Bron-Kerbosch algorithm

[2] Robson, J. M. (2001), Finding a maximum independent set in time $O(2^{\frac{n}{4}})$

[3] Tomita, Etsuji; Tanaka, Akira; Takahashi, Haruhisa (2006), The worst-case time complexity for generating all maximal cliques and computational experiments

[4] Minty, George J. (1980), ”On maximal independent sets of vertices in claw-free graphs”, Journal of Combinatorial Theory. Series B, 28 (3): 284 – 304, doi:10.1016/0095-8956(80)90074-X, MR 579076.

《神奇的子图》命题报告及其拓展

福建省福州第一中学 陈俊锟

摘 要

《神奇的子图》是我在集训队互测 2017 中命制的一道传统题。通过弱化问题的不同条件，可以将问题转化为满足不同的特定条件的图的问题，从而使用多种算法解决。本文重点对于一种特定条件下的图的一类带修改 DP 问题进行研究，提出“圆方树”和基于链分治思想的算法，并对两个算法进行了一些扩展应用。我希望借助《神奇的子图》一题，与大家分享一类点双连通分量的通用处理方法和树上带修改 DP 问题的解决方法。

1 试题

1.1 题目简述

给定一张无向简单图 $G = (V, E)$，设 $n = |V|, m = |E|$，每个点都有权值，分别为 $v_1, v_2, ..., v_n$。G 满足一个特殊条件：任取 V 中的 7 个点构成集合 V'，都存在三个不同的点 p、q、x 满足 $p, q \in V', x \in V - \{p, q\}$，使得从图中删去点 x 后， p 和 q 不连通。

现在，给出常数 k，你需要从 G 中找出一个连通的子图 $G' = (V', E')$，使得对于任意 $p \in V'$ 满足 p 在 G' 中的点度不超过 k。要求最大化子图中每个点的权值之和 $\sum_{p \in V'} v_p$（只需要输出该式子的最大值），并求出能最大化该式子的不同子图个数模 64123。两个子图 $G_1 = (V_1, E_1)$ 和 $G_2 = (V_2, E_2)$ 不同，当且仅当 $V_1 \neq V_2$ 或 $E_1 \neq E_2$。

还需要支持 q 次修改，每次修改一个点 p 的 v_p，请在所有修改之前及每次修改之后都求出上述两个问题的答案。

1.2 输入格式

第一行包含 3 个整数 n, m, k，表示图 G 的节点个数、边数及点度限制常数 k；

第二行包含 n 个整数 $v_1, v_2, , , v_n$，表示每个节点的初始权值；

接下来 m 行，其中第 i 行包含 2 个整数 $x_i, y_i \in [1, n]$ ，表示图中的第 i 条边连接节点 x_i 和节点 y_i；

接下来一行包含 1 个整数 q，表示修改节点权值的次数；

接下来 q 行，其中第 i 行包含两个整数 p_i, w_i，表示在第 i 次修改中，将 v_{p_i} 修改为 w_i。

1.3 输出格式

在刚开始及每次修改过后输出 1 行答案，因此总共应该输出 $q+1$ 行；输出的每一行包含两个数，分别表示“权值和最大的那个子图的权值之和”和“有多少不同的满足条件的子图的权值和达到最大值（除以 64123 的余数）”这两个问题的答案。

1.4 样例输入

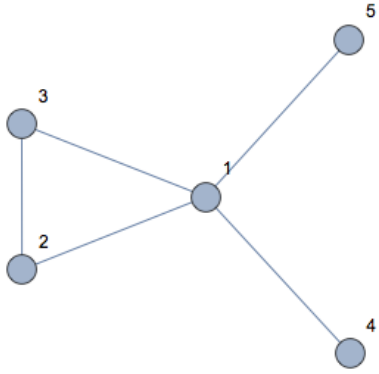
```
5 5 2
1 2 1 2 2
2 3
2 1
1 4
1 3
1 5
1
2 1
```

1.5 样例输出

```
6 4
5 5
```

1.6 样例说明

样例中，图 G 如下图所示：



在最开始，存在以下 4 种点权和为最大值 6 的方案：

编号	V.	E.
1	$\{1,2,3,4\}$	$\{(2,3),(1,2),(1,4)\}$
2	$\{1,2,3,4\}$	$\{(2,3),(1,3),(1,4)\}$
3	$\{1,2,3,5\}$	$\{(2,3),(1,2),(1,5)\}$
4	$\{1,2,3,5\}$	$\{(2,3),(1,3),(1,5)\}$

当修改节点 2 的点权为 1 后，存在以下 5 种点权和为最大值 5 的方案：

编号	V.	E.
1	$\{1,2,3,4\}$	$\{(2,3),(1,2),(1,4)\}$
2	$\{1,2,3,4\}$	$\{(2,3),(1,3),(1,4)\}$
3	$\{1,2,3,5\}$	$\{(2,3),(1,2),(1,5)\}$
4	$\{1,2,3,5\}$	$\{(2,3),(1,3),(1,5)\}$
5	$\{1,4,5\}$	$\{(1,4),(1,5)\}$

1.7 数据范围与约定

对于所有数据，保证 $n \geq 2, m, q \geq 0, 2 \leq k \leq 3, 1 \leq v_i, w_i \leq 5000$。

每个子任务的详细信息如下表（采用捆绑测试）：

子任务	总分值	测试点	n	m	k	q	特殊性质
1	7	1	≤ 10	≤ 20	$= 3$	≤ 100	无
2	18	2,3	≤ 10000	$= n-1$	$= 2$	≤ 20000	1
3	7	4,5	≤ 50000	$= n-1$	$= 2$	≤ 50000	1
4	15	6,7,8	≤ 100000	$= n-1$	$= 2$	≤ 200000	1
5	12	9,10	≤ 100	≤ 300	$= 2$	$= 0$	2,3
6	9	11,12	≤ 1000	≤ 3000	$= 3$	$= 0$	3
7	5	13	≤ 30000	≤ 100000	$= 3$	$= 0$	无
8	14	14,15	≤ 100000	≤ 300000	$= 3$	$= 0$	无
9	3	16	≤ 30000	≤ 55000	$= 3$	≤ 10000	无
10	10	17 20	≤ 30000	≤ 100000	$= 3$	≤ 10000	无

特殊性质 1：保证 G 是一棵 n 个节点的无根树。

特殊性质 2：保证所有的 v_i, w_i 均为 1。

特殊性质 3：任取 V 中的 5 个点构成集合 V'，都存在三个不同的点 p、q、x 满足 $p, q \in V', x \in V - \{p, q\}$，使得从图中删去点 x 后，p 和 q 不连通。

对于每个测试点，如果选手输出的答案中，每行的第一个数都是对的，但某些行的第二个数出错了，仍然可以获得该测试点 60% 的分数（四舍五入到整数）。

1.8 限制

时间限制：TUOJ 上的 4 秒；

空间限制：1GB。

2 初步分析

2.1 记号与约定

使用记号 $[x]$，其定义为：若命题 x 为真命题，则 $[x] = 1$，否则 $[x] = 0$。

使用记号 $O(A) + O(B) + O(C)$ 来描述一个算法的复杂度为预处理复杂度 $O(A)$，单次修改的复杂度为 $O(B)$，空间复杂度为 $O(C)$。则算法的总时间复杂度应为 $O(A + qB)$。

使用通配符记号 $*$。若一个出现包含 $*$ 的变量，则表示所有符合条件的变量的集合。例如，对每个 $p \in V$ 定义了 $f(p)$，那么 $f(*)$ 指代集合 $\{f(p)|p \in V\}$；对每个 $p \in V, 1 \le d \le k$ 定义了 $f(p,d)$，则 $f(a,*)$ 指代集合 $\{f(a,b)|1 \le b \le k\}$。

原图可能不连通。若原图不连通，可以对每个连通块分别求出最优解。因此之后的分析默认是在连通的图上进行。

本题需要求最优解的方案数。统计最优解的方案数可以使用以下通用算法：定义信息 (v,c) 表示最优值为 v、方案数取模后为 c；定义加法运算

$$(v_1,c_1) + (v_2,c_2) = (\max(v_1,v_2), c_1[v_1 \ge v_2] + c_2[v_2 \ge v_1] \bmod 64123)$$

定义乘法运算

$$(v_1,c_1) \times (v_2,c_2) = (v_1 + v_2, c_1 \times c_2 \bmod 64123)$$

使用该信息即可同时维护最优解和方案数，在后文中不再进行说明。

2.2 算法一 暴力算法

子任务 1 中，数据范围非常小，m 不超过 20，因此可以考虑枚举所有子图。

注意到最优化的目标和修改只和 V' 有关，而 E' 只会影响方案数，不难想到对每个 V' 预处理其 E' 的方案数。

预处理时，枚举边集 E'，判断其生成子图是否符合点度限制及是否连通即可，若满足条件则更新该生成子图中点集的方案数。注意处理单点的情况。

每次修改后只需要重新计算每个 V' 的权值和，将达到最优的 V' 的方案数相加即可。

复杂度 $O(m2^m) + O(2^n) + O(2^n + m)$，可以通过子任务 1，期望得分 7 分。这个算法并没有用到题目的任何性质，因而复杂度非常糟糕。

考虑首先对一类特殊的数据——树的数据进行思考。

3 树上 $k = 2$ 的数据

子任务 2 至 4 的这 40 分数据满足 G 是一棵树，并且 $k = 2$，不难看出这些测试点求的是支持单点修改权值的树上权值和最大的链。下文中，一律用“最长链”来指代权值和最大的链。

3.1 算法二 树上的暴力

子任务 2 中，数据范围支持每次修改 $O(n)$ 的算法。

这个问题满足最优子结构的性质，因此可以考虑使用动态规划算法。随意选取根节点 r 将树转为有根树，设 $Ch(i)$ 为点 i 的孩子节点集合。设 $f(i)$ 表示 i 子树中距离 i 最远的点的**距离**[20]，$g(i)$ 表示完全在 i 子树中的最长链的权值，则

$$f(i) = v_i + \max_{p \in Ch(i)} f(p)$$

$$g(i) = \max\left(f(i), \max_{p \in Ch(i)} g(p), v_i + \max_{p,q \in Ch(i) | p \neq q} (f(p) + f(q))\right)$$

只需要在 $f(i)$ 和 $g(i)$ 中同时记录方案数即可求出最优解的方案数，答案就是根节点 r 的 $f(r)$ 及其方案数。

在每次修改的时候，我们只需要更新 p 到根节点 r 的路径上的 DP 值即可，单次修改的时间复杂度为 $O(n)$。

复杂度 $O(n) + O(n) + O(n)$，可以通过子任务 2，可能可通过子任务 3，期望得分 18 至 25 分。

3.2 算法三 点分治

子任务 3、4 的修改次数和树的规模较大，考虑优化每次修改的复杂度。

这里的问题是一类路径统计问题，因此可以考虑使用点分治，其具体内容及实现方式可以参考 [1]。在分治的过程中，只需要考虑过当前重心 g 的所有路径的最长链及其方案数即可。设重心 g 的分支集合为 Br（其实就是当前连通块中 g 连出的出边集合），设 $f(i)$ 表示当前连通块中，以 g 为根的 i 子树中，**距离 g 最远**的点的距离，则过重心 g 的最长链为

$$\max\left(f(g), \max_{p,q \in Br | p \neq q} (f(p) + f(q)) - v_g\right)$$

在每次修改的时候，考虑包括被修改的点 i 所在的所有分治中的连通块，而这样的连通块只有 $O(\log n)$ 个，因此可以一个个暴力修改。在每个连通块中，被修改的是以当前 g 为根、在 i 子树中的所有点到 g 的距离，并且修改为加上修改后 v_i 和修改前的 v_i 的差值。

[20]这里的距离定义为两点之间唯一简单路径上所有点的点权和

不难想到求出以 g 为根时当前连通块的一个 DFS 序后，用线段树维护区间到 g 的距离的最小值，那么每次修改就是一个区间修改。

修改完深度后，我们还要维护过重心的距离，其中 $f(g)$ 只需要查询线段树的根节点，为了维护

$$\max_{p,q\in Br|p\neq q} f(p)+f(q)$$

可以用堆来维护每一个分支的 $f(p)$ 值，这样只需要用堆取出最大值和次大值即可。每个分支的 $f(p)$ 同样可以在线段树上区间查询。

复杂度为 $O(n\log n)+O(\log^2 n)+O(n\log n)$，可以通过子任务 2、3、4，期望得分 40 分。

4 树上 $k=3$ 的数据

考虑一类特殊的数据：树上 $k=3$ 的数据。这种数据虽然没在子任务中出现，但它是一个很好的接近正解的方向。我们对这一类问题进行思考，并尝试将其解决方案推广到一般图的问题上来。

4.1 算法四 树形 DP

同样可以设计出一个动态规划算法。记录 $f(i,d)$ 为所有节点都在 i 中，包括点 i 且 i 的点度恰好为 d 的子图的最大权值，并记录 $g(i)=\max f(i,*)$。那么 $f(i,*)$ 的转移就是对每一个 $p\in Ch(i)$ 进行一个重量为 1 价值为 $g(p)$ 的背包 DP 即可，最后再加上 a_i。

这个算法的复杂度为 $O(nk^2)+O(nk^2)+O(nk)$，期望得分与可通过的测试点与算法二相同。

4.2 优化的方向？

在算法三中，我们采用了点分治，将树划分成 $O(\log n)$ 个分治结构，那么修改的时候就只需要分别修改这些结构，然后再使用数据结构（线段树）维护每个分治结构，从而支持了高效修改。

然而在 $k=3$ 的问题中，我们要求的不再是最大权值和的链，而是一个连通块。在点分治中，不仅难以处理“经过重心的满足条件的连通块”，更难以支持高效地修改。

下面，我们介绍这样一道例题，来启发我们对原问题的思考。

例题一 树上最大权独立集带修改版[21]

给出一棵 n 个节点的无根树，每个点有点权 a_i。

[21]题目来源：经典问题

现在有 q 次操作，每次修改一个点的 a_i。要求在最开始和每次修改之后，都输出树上最大权值和的独立集的权值和。

数据范围 $n, q \leq 1000000, 1 \leq a_i \leq 10^9$

时间限制 3 秒

空间限制 512 MB

对于没有修改的问题，可以设计出这样的 DP：随意找一个点当根，将树转化为有根树。设 $f(i)$ 表示所有节点都在 i 子树中、且 i 在该独立集中的最大权独立集的权值和，$g(i)$ 表示所有节点都在 i 子树中、且 i 不在该独立集中的最大权独立集的权值和，则转移为

$$g(i) = \sum_{p \in Ch(i)} \max\left(f(p), g(p)\right)$$

$$f(i) = a_i + \sum_{p \in Ch(i)} g(p)$$

然而这种 DP 并不能在修改输入的时候，快速求出最优解。

同样考虑一类特殊的数据：树的形态是一条链。链上的问题容易用线段树维护。在线段树中的每个节点记录信息 $ans(a, b)$，其中 $a, b \in \{0, 1\}$。设该节点所维护的区间为 $[l, r]$，则 $ans(a, b)$ 表示所有点都在 $[l, r]$ 中，且 a = [l 在该独立集中]、b = [r 在该独立集中] 的最大权独立集的权值和。则区间信息的合并为（设下标 l、r 表示左右子节点的信息，如 $ans_l(a, b)$ 表示左子节点的 $ans(a, b)$）

$$ans(a, b) = \max_{c+d<2} \left(ans_l(a, c) + ans_r(d, b)\right)$$

线段树中，包含某个位置的节点只有 $O(\log n)$ 个，因此在修改后也只需要重新计算这 $O(\log n)$ 个节点的信息即可。因此这种算法支持在 $O(\log n)$ 的时间内求出修改过后的最优解。

但是将这一算法扩展到树上时，我们遇到了不小的麻烦。算法三中，我们使用了动态点分治来解决。然而这题的线段树算法相当于是记录了当前连通块（在链上为区间）中，所有与外界（其他分治结构的连通块）相连的点的信息（是否在独立集中）。在链上一个连通块和外界相连的点最多只有 2 个，可以记录下来；但如果是在树上，进行点分治，一个连通块和其他分治结构的连通块相连的点可达到 $O(\log n)$ 个，维护起来比较复杂，更难以支持高效修改。因此动态点分治不可取。

线段树和点分治能高效的问题的关键在于使用了分治思想，使得每次影响的分治结构为 $O(\log n)$ 个，从而保证了复杂度。在这里，我们可以考虑使用**基于链的分治**（简称链分治）。

基于链的分治是对链进行轻重链剖分后，先取出一条重链，然后从图中删去这条链，剩下每一棵子树（这些子树的根都是重链上某个点的轻儿子）都这样递归进去做；递归做

完之后，再考虑这条链的贡献。[22]

使用基于链的分治，考虑**直接对前面的 DP 本身进行优化**。不难发现由于每条重链的顶端要么是根节点，要么顶端的父亲属于另一条重链，因此重链与重链之间同样形成了**有根树**的关系，即若重链 A 是重链 B 的父亲，则 B 顶端的父亲在 A 中，B 的顶端是 A 中某个节点的轻儿子。称这棵树为**重链树**，那么我们将 DP 的顺序进行修改，按照**重链树的顺序**进行 DP，即原本是**先处理子树中的所有节点**，现在是先处理**当前重链的重链树子树中的所有重链**。

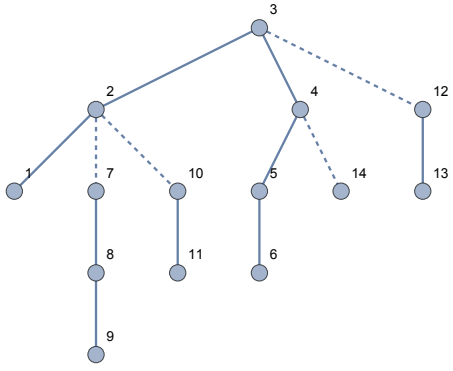
对于每一条重链，当我们递归处理完每个点的轻边子树中所有重链，并考虑轻边子树的 DP 值对答案的影响（具体的考虑方法在下一段中介绍）后，在这条重链上的问题**就和序列上的动态独立集相差无几**了，因此考虑用序列上的线段树进行维护。这里可以沿用序列上的 $ans(a,b)$ 的状态设计，并进行一些修改。设当前重链上的点按到根的距离从小到大的顺序分别是 $p_1, p_2, ..., p_c$，当前线段树节点维护的区间是 $[l,r]$，则记录信息 $ans(a,b)$ 表示所有点都在 $p_l, p_{l+1}, ..., p_{r-1}, p_r$ 这些点以及它们的**轻儿子的子树中**，且 $a = [\ p_l$ 在该独立集中 $]$，$b = [\ p_r$ 在该独立集中 $]$ 的最大权独立集的权值和。这里的信息合并和序列上的方式相同。

现在还差一个问题：如何求出一个点的所有轻边子树对它的影响，即如何对于链上长度为 1（只包含一个点）的区间（简称单点）求出其 $ans(a,b)$。不难看出只有 $ans(0,0)$ 和 $ans(1,1)$ 是有意义的。回忆算法四的 DP，设当前的单点为 i，不难得出转移（其中 $Ch'(i)$ 表示节点 i 的轻儿子集合）

$$ans(0,0) = \sum_{p \in Ch'(p_i)} \max\left(f(p), g(p)\right)$$

$$ans(1,1) = a_i + \sum_{p \in Ch'(p_i)} g(p)$$

这里只需要统计两个和式 $\sum_{p \in Ch'(p_i)} \max\left(f(p), g(p)\right)$ 和 $\sum_{p \in Ch'(p_i)} g(p)$ 的值。这看起来难以下手，但观察下图:



[22]关于链分治的具体实现及详细证明可以参考 [1]。

别忘了到**一个点的轻儿子就是该轻儿子所在重链的** p_1！那么可以借用轻儿子重链的信息求出 DP 值。

对于轻儿子 p，其 $f(p) = \max(ans(1,0), ans(1,1))$，$g(p) = \max(ans(0,0), ans(0,1))$。因此只需要对每个节点开两个变量分别维护所有轻儿子的两个和式的值即可。

每次修改的时候，同样先修改被修改点所在的重链，然后跳到这条重链顶端节点的父亲所在的重链进行单点更新，这样一路跳到根即可完成更新。注意在重链与重链之间跳跃的时候，不能 $O(|Ch'(p_i)|)$ 地重新求出两个和式的值，而应该直接对两个维护和式的值的变量进行加减。

这样这个问题就被以 $O(n) + O(\log^2 n) + O(n)$ 的复杂度解决了。

简单地审视一下这个算法。这个算法是通过修改 DP 的顺序为重链的树的顺序，将问题转化为链上的问题，再用序列上的算法高效解决。由于这个算法并不是直接利用树上问题的性质，而是**将树分成若干条链**后，依次解决序列上的问题;而序列上的相同问题**往往比树上的问题具有更多性质、更加方便维护**，因此它具有非常强的扩展性。考虑将这个算法扩展到树上 $k = 3$ 的数据中。

4.3 算法五 基于链的分治

考虑用链分治解决树上的 $k = 3$ 的带修改的问题。同样考虑直接对算法四的 DP 进行优化。首先处理完子树中的所有重链后，对于每一条重链，需要考虑的就是最高点在这条重链上的所有连通子树。这棵子树和这条重链的交一定是重链上的一段区间，因此当考虑了轻边子树的贡献后，每条重链上的问题就类似一个**带点度限制的最大子段和问题**了，同样可以使用线段树维护。

同样设重链上的每个点按到根的距离从小到大的顺序分别是 $p_1, p_2, ..., p_c$。对于每一个线段树节点（设其区间为 $[l, r]$），记录以下信息（前提均为“所有节点都在 p_l 的子树中”）：

- $cro(a, b)$：包括 p_l 和 p_r 、且 p_l 的点度为 a、p_r 的点度为 b 的连通子图的最大权值和
- $lef(a)$：包括 p_l 但不包括 p_r、且 p_l 的点度为 a 的连通子图的最大权值和
- $rig(b)$：包括 p_r 但不包括 p_l、且 p_r 的点度为 b 的连通子图的最大权值和
- mid：既不包括 p_l 也不包括 p_r 的连通子图的最大权值和

这里的信息维护与最大子段和问题类似。在合并两个区间的信息的时候，枚举左右节点的状态，判断中间的点度能否合并（合并需要在这两个点之间加一条边，因此需要判断这两个点的点度是否均小于 k），若可以合并则更新合并后的状态。合并信息的复杂度为 $O(k^4)$，可以使用前缀和优化至 $O(k^2)$。

由于这里要统计方案数，信息不能有冗余，因此必须严格地分类，每个子图必须属于且仅属于其中某一类。需要注意的细节有：左子节点的 $cro(a,*)$ 可以更新自己的 $lef(a)$、右子节点的的 $cro(*,b)$ 可以更新自己的 $rig(b)$、左子节点的 $rig(*)$ 和右子节点的 $lef(*)$ 均可以更新自己的 mid。在合并后的状态中，如果两个子节点的区间都不是单点（即该节点的区间长度为 1），那么合并后 p_l 的点度就是合并前左子节点状态中 p_l 的点度、合并后 p_r 的点度就是合并前右子节点状态中 p_r 的点度；但如果某一个子节点是单点（例如左子节点是单点），那么合并后 p_l 的点度是会改变的（在这里的问题中只会 +1），还需要维护。

现在，信息已经合并完成了，最后的问题就是如何对链上每个点求出 $cro(*,*)$、$lef(*)$、$rig(*)$、mid 这些信息了。利用轻重链剖分后，一个点的轻儿子就是该轻儿子所在重链的左端点这一性质，同样可以借助轻儿子所在重链的线段树的信息。对于一个单点，只有 mid 和 $cro(a,a)$ 这样的信息是有意义的，考虑如何维护。

为了方便，我们对每个重链记录一个“重链信息”：$val(d)$ 表示完全在这个重链顶端的的子树中、包含重链顶端并且重链顶端点度为 d 的子图的最大权值和；ans 表示完全在这个重链顶端的子树中、且不包含重链顶端的最大权值和。$val(d)$ 可以用重链线段树根节点的 $lef(d)$ 和 $cro(d,*)$ 求得，ans 则可以用重链线段树根节点的 mid 和 $rig(*)$ 求得。

单点的 mid 就是每个轻儿子的重链信息的 ans 的最大值，单点的 $cro(a,a)$ 就是对每个轻儿子重链信息的 $val(*)$ 数组进行一个背包 DP。为了支持修改，要对每一个点使用一个线段树或平衡树来维护其所有轻儿子重链信息中 $val(*)$ 的背包和最大的 ans。这样就能求出链上每个单点的信息了并支持修改了。

每次修改的时候，同样先修改被修改点所在的重链，然后跳到这条重链顶父亲所在的重链进行单点更新，这样一路跳到根即可完成更新。同样注意在重链与重链之间跳跃的时候，需要在维护轻儿子重链信息的线段树或平衡树中进行修改。

复杂度 $O(n)+O(\log^2 nk^2)+O(n)$，通过的子任务和期望得分均与算法三相同。

至此，我们已经成功地解决了所有的树上数据，接下来考虑一般图上的数据。

5 一般图的数据

子任务 5 至 8 这 40 分的数据中，没有修改，只需要进行一次求解。

5.1 特殊性质

注意到本题中特殊条件：任取 V 中的 $t\in\{5,7\}$ 个点构成集合 V'，都存在三个不同的点 p、q、x 满足 $p,q\in V',x\in V-\{p,q\}$，使得从图中删去点 x 后，p 和 q 不连通。考虑分析这一条件的意义。

回忆点双连通分量的知识。[23]我们知道，两个点 $p,q\in V$ 是点双连通的，当且仅当不存

[23]关于点双连通分量的具体的实现方式及相关定理的证明详见 [2]

在点 $x \in V-\{p,q\}$ 使得从图中删去 x 后，p 和 q 不连通。在一张无向图中，可以利用 Tarjan 算法求出每个点双连通分量，如果两个点 p,q 满足存在点双连通分量 C 使得 $p,q \in C$，那么 p,q 点双连通，否则 p,q 不点双连通。两个点双连通分量有不超过一个公共点，若点双连通分量 A,B 存在公共点 x，那么对于任意的 $p \in A-\{x\}, q \in B-\{x\}$，删去 x 后 p,q 不连通，我们称 x 为（A 和 B 之间的）**割点**。

从上述性质，猜想特殊性质等价于以下命题：**图中不存在大于或等于 t 的点双连通分量（即图中所有点双连通分量都小于 t）**。尝试证明：

证明.

(充分性) 考虑反证法。若存在不小于 t 的点双连通分量，那么只需要任取一个大小不小于 t 的点双连通分量中的任意 t 个点构成 V'，则这 V' 个点两两都点双连通，即不存在 $x \in V-\{p,q\}$，使得从图中删去点 x 后，p 和 q 不连通，与题设矛盾。

(必要性) 同样使用反证法。若存在 t 个点构成集合 V'，不存在三个不同的点 $p,q \in V', x \in V-\{p,q\}$，使得从图中删去点 x 后，p 和 q 不连通，那么由定义得对于任意 $p,q \in V'$ 满足 p,q 点双连通，则 V' 为一个点双连通分量或一个点双连通分量的子集，因此一定存在一个点双连通分量的大小不小于 t，与题设矛盾。 □

题目中的特殊条件满足 $t=7$，因此每个点双连通分量的大小不超过 6，我们可能可以根据这个性质设计算法。接下来的算法介绍中，若无歧义则将点双连通分量简称为“点双”。

5.2 算法六 点双连通分量上的暴力

子任务 5、6 满足特殊性质 3，每个点双的大小不超过 4。

Tarjan 求点双的过程是这样的：在 DFS 的过程中维护一个栈，每次 DFS 到点 i 的时候，将点 i 压入栈中，然后令计时器加一，并设置 dfn(i) 和 low(i) 为当前计时器。对于 i 的每一条边 (i,p) 进行考虑，如果这条边是返祖边，则用 dfn(p) 更新 low(i)；否则 DFS 进入 p，返回后先用 low(p) 更新 low(i)，然后若 low(p) $\geq$ dfn(i)，那么我们找到了一个点双（设为 C），从栈中弹栈直到弹出 p 为止。设弹出的点集为 S，则我们找到的点双的点集为 $V_C = S \cup \{i\}$，设 i 为 C 的**根**。

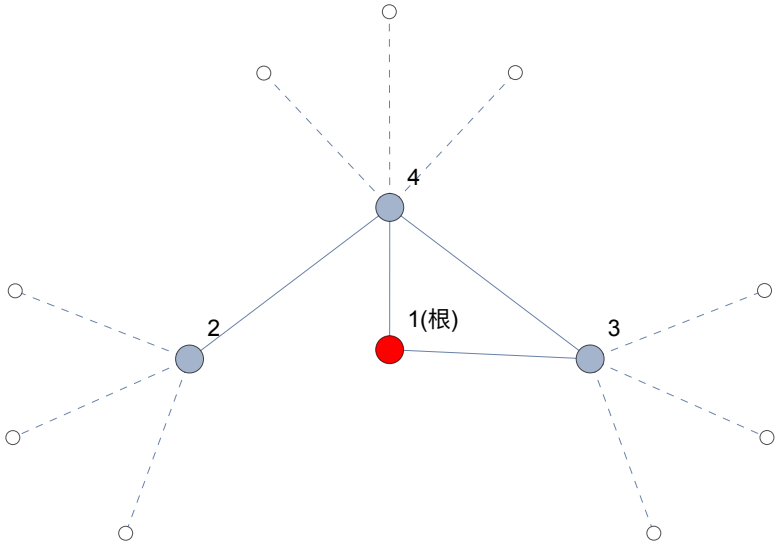
如果一个点 p 在一个点双 C 中，且不是 C 的根，那么我们就称 p **真实属于点双** C，这样（除了最开始调用 Tarjan 的点，即 DFS 树的根之外）每个点都真实属于一个唯一的点双。每一个点双都有一个唯一的根，而它的根真实属于另一个点双，因此点双与点双之间**构成了一个有根树森林的结构**，其中每个点双的父亲为它的根所在点双（如果存在）。我们称这个有根树森林为**点双树**。结合 Tarjan 的过程，可以看出 Tarjan 相当于是**按照 DFS 序遍历点双树**。

现在可以考虑在树形结构上进行 DP。设计状态 $f(i,d)$ 表示所有点都在以点 i 为根的点双（以及这些点双的子树内的所有点双）中，且满足子图中 i 的点度为 d 时的最大点权和

连通子图的点权和。由于 Tarjan 相当于 DFS 遍历点双树，那么对于任意 $p \in S$，以 p 为根的所有点的点双都已经全部考虑过了（当找出根为 i 的点双时 i 一定在栈中），因此 $f(p,*)$ 也都已经被求出。

现在考虑在 C 中进行处理以更新答案及 $f(i,*)$。由于 $V_C \le 4$，其边数不超过 $\binom{4}{2} = 6$，因此可以类似算法一那样，通过暴力枚举边集来枚举 C 的一个子图，即枚举所求子图 G' 在点双 C 中的连边情况。

接下来，我们还要考虑这个子图在点双树子树中的连边情况。如下图是一个枚举到的子图（设 $k = 3$）：



如点 2 在当前点双的连边情况中的点度为 1，还可以再添加上 2 的点度，因此这个子图在子树中的连边情况只需要满足点 2 的点度不超过 2，从而可以将当前子图拼接[24]一个“所有点都在以点 i 为根的点双（以及这些点双的子树内所有点双）中，且满足点 2 的点度不超过 2”的子图，因此当前子图的最大权值可以加上 $\max(f(2,0), f(2,1), f(2,2))$。

对于当前枚举的子图 G_0 中的每一个点都进行一个这样的拼接操作，那么我们就求出了在 C 中的子图为 G_0 时，且考虑了所有在 C 的点双树子树中的所有点双中的子图时的最大点权和，设这个点权和为 $g(G_0)$。如果 G_0 包含根 i，就用 $g(G_0)$ 来更新 $f(i,*)$（这里的更新是一种背包问题），否则用 $g(G_0)$ 取更新全局的答案 ans（因为在这种情况下当前子图不再和 i 及其上方的点连通，因此需要在这里更新答案）。在这里需要注意的细节是：计算 G_0 的最大权值和的时候，要加上在 G_0 点集中所有非根节点的权值，而根节点的权值在根节点所在点双中考虑，避免错误计算子图的权值和[25]。在 Tarjan 算法运行完之后，我们就得到了答案。方案数可以在这个过程中同时维护。

[24]这里的“拼接”两个子图指将两个子图的点集和边集分别合并

[25]如果累加了根节点的权值，那么在以这个根节点为根的所有点双中都会累加一次，导致 $f(i,*)$ 中计算了多次根的权值

设图中最大的点双大小为 c，则最坏复杂度为 $O\left(c2^{\binom{c}{2}}n+m\right)+O\left(c2^{\binom{c}{2}}n+m\right)+O(n+m)$，可以通过子任务 1、2、5、6，可能可以通过子任务 3，期望得分 46 至 53 分[26]。

5.3 算法七 预处理子图

子任务 7、8 不再满足特殊性质 3，一个点双的边数可能为 $\binom{6}{2}=15$，且总点数 n 非常大，算法六的复杂度无法通过。

算法六中的瓶颈在于枚举点双的所有子图，考虑对枚举进行优化。算法一中，我们将等价的、不变的状态合并（对每个点集预处理连边方案数），从而优化了复杂度，那么这里也可以这么做。但这里**还需要考虑点度限制**，因此需要记录状态：一个 6 位四进制数[27] deg 为**每个点的点度的状态压缩（简称“点度状压”）**，其中 deg 在四进制下从低到高的第 i 位表示图中第 i 个点的点度。deg 相同的两个状态的权值和转移方式都是相同的，因此只需要对每一个 deg 状态预处理连边方案数即可。这里有一个方便之处：对于一个至少有一条边的图，所有的点的点度大于或等于 1，因此记录 deg 同时还可以表示点集的信息。那么现在设 $f(deg)$ 表示所有点度状压为 deg 的不同子图个数，这样只需要求出每个 $f(deg)$ 即可统计最优解和方案数，不需要枚举每一个子图。

考虑如何求出 $f(deg)$。可以在一开始预处理所有点数不超过 6 的满足点度限制的图（不会超过 20000 个），按照点数排序。这样在处理每个点双的时候，只需要枚举所有点数不超过当前点双大小的图，用位运算快速判断这个图的边集是否是当前点双的子集即可。这样对每个点双就只需要不到 20000 次运算，而不同的状态数理论上只有 4^6 个，实际上只有不到 1600 个，这样就可以通过子任务 7、8 的数据了。

为了加快程序运行速度以通过子任务 3，可以对程序进行一些优化。不难想到这样的优化：由于这个算法相当于是以子树为阶段的树形 DP，每次修改点 p 后，只有 p 到根的路径上的点的 DP 值可能被修改。因此每次修改也只需要改这条路径上的点即可。注意到原本的 DP 中有“更新全局的答案”这一操作，可以不记录全局变量 ans 而是记录 $g(i)$ 表示考虑了所有以 i 为根的点双及它们在点双树的子树中的所有点后最大的点权和，这样每次也只需要更新所修改的点到根的 $g(i)$ 即可。事实上加了这些优化后，这个算法可以通过子任务 10 的部分测试点。

设图中最大的点双大小为 c；$S(s,k)$ 表示点数不超过 s 且所有点点度不超过 k 的子图个数[28]；$G(s,k)$ 表示点数不超过 s 且所有点点度不超过 k 的子图的**点度状压**个数[29]。则本算法的复杂度为 $O\left(c2^{\binom{c}{2}}+(S(c,k)+G(c,k)c)n+m\right)+O(G(c,k)cn)+O(S(c,k)+G(c,k)n+m)$，可以通过子任务 1、2、5、6、7、8，可能可通过子任务 3，期望得分 65 至 72 分。

[26]对于树的数据 $c=2$

[27]2 的整数次幂进制可以用位运算快速查询、修改某一位，因此无论 k 的取值如何一律使用四进制

[28]根据前文 $S(6,3)\le 20000$

[29]根据前文 $G(6,3)\le 1600$

6 更好的转化方式

最后的两个子任务修改较多，需要支持更加高效的修改。

6.1 圆方树——点双树的改进

前面我们提到，树上 $k = 3$ 的数据已经有了优秀的算法——基于链的分治。在算法六中，我们提出了点双树这一树形结构。然而这一结构相当于是将点双中所有点全部缩在一起，给处理带来了很多不便——在转移中既需要枚举子图又需要统计点权，还要特判各种细节。因此这个树形结构难以直接用链分治进行维护。

现在，我们考虑对点双树进行一些修改，使得这个 DP 更加简洁、更加容易优化。

例题二 商人[30]

给出一张 n 个点 m 条边的无向连通图，每个点有点权 a_i。

给出 q 个询问，每次给出两个点 x, y，要求找一条从 x 到 y 的点不重复的路径，使得路径上点权最小的点的点权最小。

数据范围 $n, m, q \le 5000000, 1 \le a_i \le 10^9$

时间限制 2 秒

空间限制 512 MB

如果 $x = y$，则答案为 a_x。后文默认 $x \neq y$。

首先，如果存在一个点双 C 同时包含询问的 x, y，那么答案即为该点双中的最小权值。考虑证明：

证明.

先证明**对于任意的 $q \in C$，存在一条从 x 到 y 的点不重复的路径经过 q**。新建附加节点 p，连接 $(p, x), (p, y)$，则由于 x, y 连通，一定存在一条 x 到 y 的不经过 p 的路径 $y \overset{a}{\leftrightarrow} x$。这时从 p 到 x 存在两条路径：$p \leftrightarrow x$ 和 $p \leftrightarrow y \overset{a}{\leftrightarrow} x$，因此 p 和 x 点双连通；同理 p 和 y 点双连通，因此 x, p, y 同属一个连通分量。由于两个点双最多只有一个公共点，p 一定在点双连通分量 $C \cup \{p\}$ 中。因此对于任意 $q \in C$，都存在 p, q 之间两条点不相交的路径，其中必然一条经过 x，一条经过 y；设它们为 $q \overset{b}{\leftrightarrow} x \leftrightarrow p$ 和 $q \overset{c}{\leftrightarrow} y \leftrightarrow p$，则存在点不相交的路径 $x \overset{b}{\leftrightarrow} q \overset{c}{\leftrightarrow} y$。因此点双中每一个点 q 都可以在从 x 到 y 的点不重复的路径上。

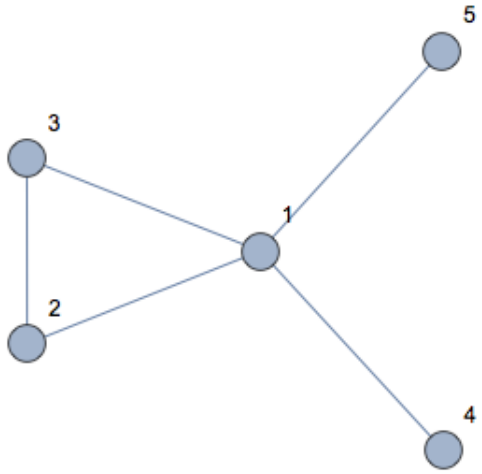
然后证明**对于任意的 $q \notin C$，不存在一条从 x 到 y 的点不重复的路径经过 q**。对于点双连通外的点 q，设其所在的任意一个点双为 D，设点双树上从 C 到 D 的路径上的点双序列按顺序为 $C, P_1, P_2, ..., P_c, D$，则从 x 到 q 需要经过 C 和 P_1 的公共点，从 q 到 y 同样也需要

[30]题目来源：简化自 **UOJ 30**，并扩大数据范围

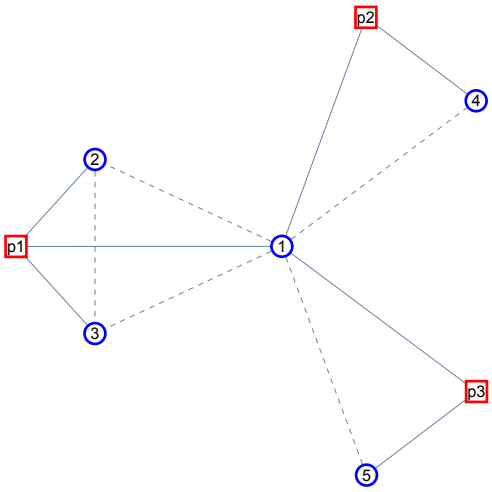
经过 C 和 P_1 的公共点，因此这个点一定会经过两次，从而 q 一定不在任意一条从 x 到 y 的点不重复的路径上。 □

继续在点双树上考虑。猜想性质：对于任意询问 x,y，答案为 x 所在点双到 y 所在点双的点双树路径上所有点双中的点的最小值。然而这个性质描述的不够具体：x、y 可能在多个点双中，那么哪一对点双才是正确的呢？任意选择不是正确的，一个反例就是三个三角形拼接而成的图。一种正确的描述是：距离最近的一对。然而这种描述还是不够具体，也难以实现。点双树的局限性显露了出来，我们需要更好的转化为树的方式。

考虑进行这样的转化：对于每一个点双 C，建立附加节点 p_C，向 C 中的每一个节点连边，并删除原点双中的所有边（由于两个点双最多只有一个公共点，因此任意两个点双没有公共边，所以这样的构造算法中不会有边被重复删除）。称原图中的点为**圆点**，新建的附加点为**方点**。如下面这张图



将其转化为圆方树后为下图（一条边也是一个点双）：



图中的每一个方点对应了一个点双。不难看出，现在转化后的图相当于对点双树进行了这些修改：1、点双树中每个点双是直接向父亲点双连边，而转化后每个点双的方点是先连接到该点双的根的圆点，再从根的圆点连接到父亲点双的方点；2、对于没有成为某个点双的根的节点，在转化后作为一个叶子节点保留在树上。从修改可以看出转化后的图同样是一棵树。我们称这一棵树为原图的**圆方树**。

由于每个附加点连出的是“菊花树”，因此圆方树的形态不受点双树的形态的影响。也就是说，无论从哪一个点开始执行 Tarjan 算法，构造出的圆方树都是相同的！并且由于保留了原图中的所有点，圆方树能更加方便地处理点双和点双之间的关系：若原图中两个点双 A, B 有公共点 q，那么在圆方树中则存在边 (p_A, q) 和 (q, p_B)。与其将圆方树视为对点双树进行修改的结果，不如将其视为一种全新的、组织点双与点双之间信息的方式。

考虑在圆方树上解决这一问题。设每个方点的权值为该点双所有点的最小值，则我们惊讶地发现：**答案就是圆方树上 x 到 y 的路径上的点权最小值！**考虑证明这一性质。任意一个在 x 到 y 的圆方树链上的点双都可以从一个圆点进入、另一个圆点离开，从而可以经过点双中任意一个点；而想要到达不在链上的点双都需要重复经过某一个圆点（两个方点之间的链上一定存在至少一个圆点），从而必须重复经过某个点。

现在我们要解决的问题就是**圆方树上 x 到 y 的路径上的点权最小值**了。这个问题和 NOIP2013 的一道经典题《货车运输》非常相似，使用 [4] 中 3.2.1 节的算法可以做到 $O(m + n\alpha(n)) + O(q \log(\log n)) + O(n \log(\log n) + m)$。由于这个算法不需要处理特殊情况，实现起来代码比较优美。

这道例题体现了圆方树的强大与便捷。考虑将圆方树应用在本题上。

6.2 算法八 使用圆方树解决子任务 7、8

现在，我们已经有了圆方树这一有力工具。尝试用它来解决子任务 7、8，看看是否能找出一些更简便的算法。

首先求出原图的圆方树。在圆方树中，原图中所有的边都没有保留，因此子图中所有的连边情况都需要在对应的方点中处理。这里的问题和树上 $k = 3$ 的问题较为类似，因此可以借鉴算法四。同样记录状态 $f(i, d)$：若 i 为圆点，表示所有在 i 的圆方树子树中、包含 i 且子图中 i 的点度恰好为 d 的子图的最大点权和；若 i 为方点，则表示所有在 i 的圆方树子树中、包含 i 对应点双的根 r_i 且 r_i 在子图中的点度恰好为 d 的子图的最大点权和。

考虑转移。对树进行 DFS。若 i 为圆点，则其所有子节点均为方点，且它们对应点双的根均为 i。只需要将所有子节点 $p \in Ch(i)$ 的 $f(p, *)$ 进行一个背包 DP，最后加上 v_i 即可。

若 i 为方点，则需要枚举 i 对应点双的连边情况（使用算法七中的方法优化枚举即可）。对于枚举的每一个子图，我们同样需要考虑子图中每个非根的点 p 可以在下方添加多少的点度，若可以添加 d 的点度，那么当前子图的权值就可以加上 $\max_{k=0}^{d} f(p, k)$——在圆方树中，每一个方点的子节点就是该方点对应点双中的所有非根节点，因此它们的 $f(p, *)$ 同样都已经求出了。最后，如果这个子图包括 i 对应点双的根 r_i，设子图中 r_i 的为 d，则可以用

这个子图的权值更新 $f(i,d)$；否则用这个子图的权值更新全局答案 ans。

这个算法看起来与算法七相差无几。但可以注意到一个特点：所有的连边情况在方点中考虑，所有的权值计算和点度限制在圆点中进行。**圆点和方点分工明确**，不仅使得算法的描述更加清晰，使程序的代码复杂度降低，还给后续算法的优化带来了莫大的方便。

复杂度、通过的子任务和期望得分均与算法七相同。

6.3 算法九 满分算法

现在万事俱备，只欠东风。结合基于链的分治和圆方树，满分算法近在咫尺。在圆方树上的 DP 比之前在点双树上的 DP 更加简洁，因此更容易使用链分治进行优化。在这里，每条重链上的问题同样是一个**带点度限制的最大子段和问题**，因而解决方法和前面类似。

在重链上的线段树中，同样维护 $cro(*,*)$、$lef(*)$、$rig(*)$、mid 这四个信息，这里信息的定义和之前相似但略有区别。对于每个区间，我们定义**“左真实点”** p'_l **和“右真实点”** p'_r 这两个点：如果 p_l 是圆点，则 $p'_l = p_l$，否则 p'_l 为 p_l 在圆方树上的父亲即 p_l 对应点双的根（当 $l \neq 1$ 时这个点就是 p_{l-1}）；如果 p_r 是圆点，则 $p'_r = p_r$，否则 p'_r 为 p_{r+1}（这里的 p_{r+1} 一定存在，因为重链末端一定是叶子）。也就是说，这里的方点除了要在状态中记录点双的根（即父亲圆点）的点度信息，还要记录**重链中下一个圆点**的信息。那么在这里，状态 $cro(a,b)$ 的意义就变为“p'_l 的点度恰好为 a、p'_r 的点度恰好为 b 时的最大权值子图”，其他状态的意义也进行相应的修改，即记录的点度改为 p'_l 和 p'_r 的点度。

每一次信息合并的时候，在中点区间中点 $m = \lfloor \frac{l+r}{2} \rfloor$ 处考虑的是**同一个点**的点度，即边 (p_m, p_{m+1}) 中的那一个圆点（每条边的两个端点都是一个圆点一个方点），它既是左子节点的右真实点，又是右子节点的左真实点，因此合并判断合法的条件是**这个点在左右子节点的状态中点度之和是否小于或等于** k。在这里同样要注意算法五中所讲述的细节，如某个子节点是单点的情况。

现在，只剩下最后一个问题——如何求出重链上每个单点的信息。如果这个点是圆点，同样可以使用算法五中用线段树维护出边的重链信息的方式维护，最后所有的 $cro(a,a)$ 也都要加上 v_i。如果这个点是方点，则需要枚举这个方点中所有的子图状态（即前面所说的 (set, deg) 这样的状态）。设这个线段树单点为 i，在这个状态中，对于既不是左真实点又不是右真实点的圆点，一定是一个轻儿子，因此使用它们的重链信息进行拼接操作。最后求出这个状态的最优解之后，根据这个状态是否包含左右真实点、及它们的点度来决定更新的是这个单点的四种信息中的哪一种，方案数要乘上预处理的 (set, deg) 的不同子图个数。一种方便的实现方式是在状压中，设这个单点的左真实点为第 0 个点，右真实点为第 1 个点，这样能减少许多特判。别忘了轻儿子的重链信息的 ans 也是要更新 mid 的。

每次修改的时候，在树链剖分中往上跳跃的过程中，需要更新所有经过的重链顶端的父亲节点的单点信息。注意如果这个点是方点，同样需要进行 1600×4（4 是除掉 p_{i-1} 和 p_{i+1} 后点双中的点数）次运算枚举状态暴力维护它的信息（初始化的时候先预处理并保存每个子图状态压及其方案数，无需每次都遍历所有子图）。

复杂度[31] $O\left(c2^{\binom{c}{2}}+(S(c,k)+G(c,k)c)n+m\right)+O\left(\left(G(c,k)c+k^4\log n\right)\log n\right)+O(S(c,k)+G(c,k)n+m)$，可以通过所有子任务，期望得分 100 分。

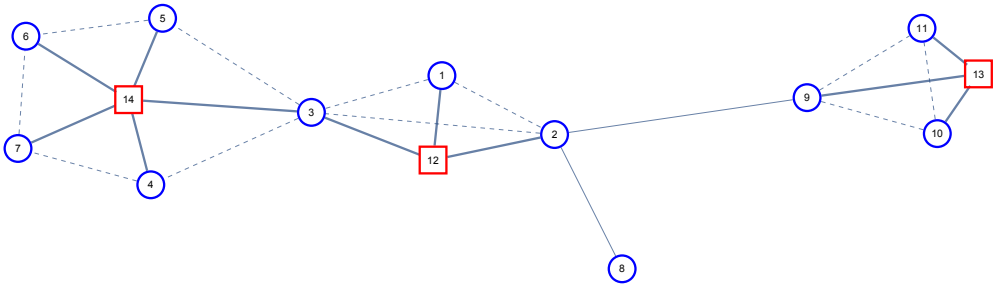
7 拓展

在本题中，我们通过构造原图的圆方树，从而将问题转化到树上，并使用基于链的分治进行解决。事实上，本题中的圆方树和链分治都具有很大的可扩展性。接下来，我将会介绍它们的一些应用。

7.1 仙人掌上的问题

一个仙人掌定义为一张无自环的无向连通图，其中每一条边都在不超过一个简单环中。

不难发现，仙人掌中的每个环都是一个点双，那么自然也可以用圆方树来解决。由于需要处理环上的信息，我们还需要维护每个方点所连出圆点的顺序。同时为了区分二元环（重边）和普通树边，我们可以不对普通树边建立方点，只对环建立方点。以下是一棵仙人掌及其圆方树：



事实上，圆方树这一算法的提出的初衷，就是为了能够更加高效、便捷地解决各种静态的仙人掌上的问题。而点双连通分量的处理方式，是从仙人掌的方式改进而来的。

经过研究，几乎所有的静态仙人掌问题，如仙人掌 DP[32]、仙人掌最短路[33]、仙人掌链剖分[34]、虚仙人掌[35]以及仙人掌点分治[36]都可以直接使用圆方树来解决。而且经过其他同学的研究[37]，甚至连动态仙人掌和动态 k 仙人掌[38]都可以使用类似圆方树的思想来解决。

[31]沿用算法七所定义的 $c, S(s,k), P(s,k)$ 记号

[32]http://www.lydsy.com/JudgeOnline/problem.php?id=4316

[33]http://www.lydsy.com/JudgeOnline/problem.php?id=2125

[34]http://uoj.ac/problem/158

[35]http://uoj.ac/problem/189

[36]http://uoj.ac/problem/23

[37]http://awd.blog.uoj.ac/blog/2319

[38]每一条边最多在 k 个简单环上的无向连通图。一般所说的仙人掌是 1 仙人掌。

关于圆方树在仙人掌上的应用，在本文中与主题无关，且篇幅过长，因此略去不讲。具体算法在作者的 UOJ 博客[39]及我在 WC2017 的营员交流课件[40]中有介绍。

7.2 支持修改输入的子树动态规划

在本题中，我们用基于链的分治高效地支持了单点修改输入、维护全局 DP 值。事实上，这个算法可以扩展到一类更加通用的问题上：支持单点修改输入、询问某一个有根树子树的 DP 值。

例题三 疫情控制问题[41]

给出一棵 n 个节点的有根树，每个节点都有点权 a_i。对于每个点，你可以选择花费 a_i 的代价将 i 的子树中所有叶子节点选中。现在有 m 个操作，每个操作为以下两者中的一种：

1. 修改某个点的 a_i；

2. 输入 i，询问如果要将 i 子树中所有叶子节点都选中，最少的花费是多少？

数据范围 $n, m \leq 1000000$。

时间限制 1 秒

空间限制 256MB。

这题的原题有特殊性质，保证修改后 a_i 不降。原题的正解从这个性质出发，设计出了一种基于决策点变动的均摊算法。但这里没有这个保证，因此原来的算法不再适用了。

如果每次询问都可以单独做一次暴力 DP，那么可以记录 $f(i)$ 表示把 i 子树中所有叶子选中的最小费用，则若 i 为叶子则 $f(i) = a_i$，否则

$$f(i) = \min\left(a_i, \sum_{p \in Ch(i)} f(p)\right)$$

时间复杂度为线性。

现在考虑有修改的数据。这题中的树是有根树，其 DP 值的计算有一个严格的顺序，如果用点分治难以维护计算 DP 值的顺序，更难以支持询问。

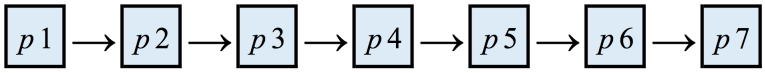
继续考虑链分治。借鉴前面的思想，考虑按照重链树的顺序计算 DP 值。由于重链树同样是有根树，按照重链树的顺序计算 DP 值只是相当于对每一个节点选择了子节点的转移顺序，并不影响转移的正确性，因此这里采用链分治是完全可行的。

[39] http://immortalco.blog.uoj.ac/blog/1955

[40] 参考文献 [3]，可以在我的 UOJ 博客中下载

[41] **题目来源：改编自 BZOJ 4712，并扩大数据范围**

要解决的第一个问题是如何在一条链上转移。设当前重链的链长为 m，我们记录 p_i 表示这条链上从上往下的第 i 个点，c_i 表示节点 i 的所有轻儿子 x 的 $f(x)$ 之和（为了方便，若 i 为叶子，则记 $c_i = \infty$）



由于是按照重链树的顺序进行 DP，同样也只需要用一个变量维护 c_i，和树上最大权独立集的算法类似。修改的时候也只需要将 c_i 加上该轻儿子的 $f(x)$ 的差值。

为了求出这条重链上每个节点的 $f(i)$，可以这样 DP：首先 $f(p_m) = a_{p_m}$，接下来对于任意 $1 \le i < m$ 都有

$$f(p_i) = \min\left(a_{p_i}, f(p_{i+1}) + c_{p_i}\right)$$

分析这个式子。可以把一个位置的 (a_{p_i}, c_{p_i}) 看做一个变换 (a, b) 满足

$$trans_{(a,b)}(x) = \min(a, x + b)$$

则对于当前重链，$f(p_i)$ 就是 $(a_{p_m}, c_{p_m}), (a_{p_{m-1}}, cp_{m-1}), ..., (a_{p_{i+1}}, c_{p_{i+1}}), (a_{p_i}, c_{p_i})$ 这些变换依次作用在 0 上的结果。而两个这样的变换合并之后，仍是原来的形式：

$$\begin{aligned} & trans_{(c,d)}(trans_{(a,b)}(x)) \\ = \ & trans_{(c,d)}(\min(a, b + x)) \\ = \ & \min(c, d + \min(a, b + x)) \\ = \ & \min(\min(c, a + d), b + d + x) \end{aligned}$$

即对一个数 x 先执行变换 (a, b) 再执行变换 (c, d) 相当于执行了变换 $(\min(c, a + d), b + d)$。

那么就可以直接用线段树来维护区间的变换和。每次询问的时候，查询该重链所在的后缀和即可，复杂度 $O(\log n)$。修改 a_p 的时候，先将 p 所在重链的线段树中 p 所在这一位进行修改，接下来求出新的 $f_{top(p)}$，然后更新 $c_{fa_{top(p)}}$ 并进行对应线段树上的修改，再继续往上更新直至根即可，复杂度 $O(\log^2 n)$。

这样，这道题被基于链的分治以初始化 $O(n \log n)$、每次修改 $O(\log^2 n)$、每次询问 $O(\log n)$ 的时间复杂度和 $O(n)$ 的空间复杂度解决了。

这道题目同样是直接使用链分治对 DP 本身进行优化，并且十分巧妙地利用了链上转移的性质，最终得出了一个非常高效且实现起来代码十分优美的算法。因此，链分治确实是解决一类树上带修改 DP 问题的利器。

7.3 支持 Link/Cut 和修改输入的子树动态规划

不难发现前面这些例题中，链分治在具体的实现上，和普通的树链剖分并没有很大的差异，都是使用线段树维护一些可以高效合并的信息。那么，这些问题是否同样可以推广到动态树链剖分——LCT 上，以同时支持 Link 和 Cut 和修改输入呢？

答案是肯定的。以例题一为例，只需要对每个节点使用两个变量维护所有轻边的两个和式的值即可，剩下的操作就和链分治中的完全相同了。不仅如此，由于 LCT 支持换根，使用 LCT 可以支持**对任意节点为根时的任意子树**求出 DP 值。而且 LCT 的复杂度为每次操作均摊 $O(\log n)$，甚至优于链分治的算法。

当然，使用 LCT 的链分治也有一定的局限性。一是 LCT 有较大的常数因子，实际运行时间不一定会优于有极小常数因子的链分治算法；二是如果轻边维护的信息不可减（如维护所有轻边的信息的最小值），则需要使用 Splay 树来保证复杂度为 $O(\log n)$，增大了代码难度；三是某一些这类 DP 使用 LCT 维护会给实现带来诸多细节，如若例题三需要支持 Link/Cut 和任意子树询问，那么就需要处理叶子节点的变化，为了支持换根时的翻转还需要维护一正一反两种标记和。

7.4 支持修改的简单路径询问

在一般图上，也可能会有一种问题，要求支持修改及询问两点的某条点不重复的路径的信息。这时，圆方树和树链剖分就同时派上用场了。

例题四 Tourists[42]

给出一张 n 个点 m 条边的无向连通图，每个点有点权 a_i。

给出 q 个操作，操作有以下两种：

1、给出两个点 x, y，要求找一条从 x 到 y 的点不重复的路径，使路径上点权最小的点的点权最小。

2、输入 p, v，将 a_p 修改为 v。

数据范围 $n, m, q \le 100000, 1 \le a_i \le 10^9$

时间限制 1 秒

空间限制 512 MB

这道题是例题二的一个强化版本，要求支持修改。如果和例题二一样，设每个方点的权值为该点双中所有点的权值的最小值，用树链剖分维护链上最小值的话，那么每次修改一个点的权值就需要遍历该点所在的所有点双，复杂度不可接受。

别忘了《神奇的子图》的算法五！在算法五中，圆点和方点分工明确：圆点只维护权值、方点只维护连边情况。这使得圆点和方点解耦，这样修改圆点时就无需遍历与其相连的所有方点。这里采用的实际上是这样一种思路：不要让可能会改变的信息被太多的结构同时维护。在这题中，让每个方点只维护**其所有子节点的圆点的信息**，而不维护其父节点

[42]题目来源：**UOJ 30**

的信息。这样每个节点的权值被修改的时候，只需要更新其父亲方点的权值即可。这样修改操作是 $O(\log n)$ 的。

考虑询问。将 x 到 y 拆成 $x \to z \to y$ 两条链，其中 z 为 x,y 在圆方树上的 LCA。不难发现所有完全包含在 $x \to z$ 和 $z \to y$ 且为 z 的方点，其父亲圆点同样也会在 $x \to z \to y$ 这条链上，因此不用单独考虑。但如果 z 是方点，那么它的父亲圆点没有被考虑到，这里暴力考虑即可。这里的操作就都是经典的树链剖分套线段树能维护的了。复杂度 $O(n+m)+O(\log^2 n)+O(n+m)$。

事实上，这道题也存在不使用圆方树的做法，但那个做法的复杂度优化不太容易想到。而从圆方树的角度入手思考，更容易想到如何进行优化，降低了一定的思维难度。

例题五 神奇的简单路径[43]

给出一张 n 个点 m 条边的无向连通图，每个点有点权 a_i。保证图中每个点双连通分量的大小都不超过 $c=8$。

给出 q 个操作，操作有以下三种：

1、输入 x,y，找一条从 x 到 y 的点不重复的路径，使路径上所有点点权和最大。

2、输入 p,v，将 a_p 修改为 v。

3、询问整张图的最长简单路径。

数据范围 $n \le 100000, m \le 500000, q \le 2000000, -10^9 \le a_i \le 10^9$，保证操作 2 的个数不超过 20000。

时间限制 6 秒

空间限制 2 GB

这个问题和《神奇的子图》有类似的限制，保证了点双连通分量的大小，因此可以考虑类似的解决方案。

如果只有操作 2 和 3，那么问题和《神奇的子图》相同（由于这里不需要计数，而点集相同的简单路径和简单环的答案相同，因此两个问题等价），使用《神奇的子图》算法九即可。这里在每个方点中预处理出所有简单路径，要记录简单路径经过的点集和起点终点，因此总状态数是 $O(2^{c-2}\binom{c}{2})$。也可以像《神奇的子图》算法五中那样记录点度，如果不保存环的话，状态数相同。操作 2 的复杂度为单次 $O\left(\left(2^{c-2}\binom{c}{2}+\log n\right)\log n\right)$，操作 3 的复杂度为单次 $O(1)$。

现在考虑支持操作 1。注意到询问点 x 到 y 在圆方树上的路径上的每个圆点都是必须经过的，因此答案可以直接加上这些点的权值；而每个方点是相对独立的，只需要在每个方点中分别取路径长度的最大值，再相加即可。即设 x 到 y 在圆方树上的路径上的点分别为 $p_0, s_1, p_1, s_2, \ldots, p_{m-1}, s_m, p_m$，其中 $p_0 = x, p_c = y$，那么对于任意的 $1 \le i \le m$，s_i 对答案的

[43]题目来源：原创

贡献就是从 p_{i-1} 到 p_i 的最大权值简单路径（这一路径一定完全包含在 s_i 对应的点双中，和其它的点双都独立）。

对每一条重链，设重链上的点按顺序为 $p_0, s_1, p_1, s_2, ..., p_{m-1}, s_m, p_m$（若重链顶端是方点，则不存在 p_0，无需维护 s_1 的信息），只需要设每个 p_i 的贡献为 a_{p_i}，设每个 s_i 的贡献为从 p_{i-1} 到 p_i 的最长简单路径的长度减去 $a_{p_i} + ap_i$，那么在这条重链上只需要询问区间贡献之和即可完成询问。事实上这里维护的就是区间的 $cro(1,1)$ 信息。

一个询问可能会跨过多条重链，因此还需要考虑跨越处的方点的点双中，走的是哪一条路径，这里的路径的起点 s 和终点 t 都是已知的。可以在初始化或修改的时候，直接对于每一对起点和终点预处理出权值和最大的路径，那么这里的查询就是 $O(1)$ 的，这样操作 1 的单次复杂度为 $O(\log^2 n)$。

这样，这道题被以预处理复杂度 $O(2^{c-2}\binom{c}{2}n + m)$、操作 1 单次复杂度 $O(\log^2 n)$、操作 2 单次复杂度 $O\left(\left(2^{c-2}\binom{c}{2} + \log n\right)\log n\right)$、操作 3 单次复杂度 $O(1)$、空间复杂度 $O(2^{c-2}\binom{c}{2}n + m)$ 地解决了。

这道题将《神奇的子图》的一类特殊情况进行了扩展，同时利用到了轻重链剖分的两种用途：分治和链上询问，成功地解决了“维护全局信息”和“支持两点之间询问”这两种问题，体现了圆方树和轻重链剖分配合的强大性。

8 命题思路与总结

8.1 命题思路

近年来，在一些在线比赛或在线评测网站中出现了一些需要支持在单点修改后询问全局或某个子结构的 DP 值的题目，如 BZOJ 4712、Codeforces 480E、Codeforces 750E 等。我对这类题目进行了一些深入的研究，从而推导出了一些基于分治思想的高效算法。

其中，如 BZOJ 4712 的这种需要修改单点输入、以子树为阶段的树形 DP 最为有趣，然而 BZOJ 4712 的出题人给出的算法是一种基于修改的特殊性质的均摊算法。我对这道题进行研究，最终成功地设计出了一种使用链分治的算法，这个算法不仅运行速度更快，而且适用范围更广，因此我将其扩展到更广的范围，改编为例题三。这一解决方案十分优美，适用面也比较广，因而激发了我极大的兴趣，因此我想把它推广到更加通用的应用中。

对于点双连通分量这类经典问题，我之前研究出了一种能够高效、简洁地处理点双之间关系的结构——圆方树。它能够将图转化为树，从而使用树上问题的解决方法。结合圆方树和链分治，我命制了这样一道题目。

为了防止被暴力算法或错误算法水过，我设计了所求的信息——最优解和最优解的方案数。该信息既不可减也不可重复贡献或遗漏贡献，因此更加容易构造出强度较大的数据。

命制本题的另一个目的，是希望能与大家分享自己提出的适用圆方树和链分治解决一

类问题的研究成果，希望能够引起更多关于树上动态 DP 问题及点双连通分量问题的一些思考，从而起到抛砖引玉的作用。

《神奇的子图》是一道有一定难度的题目。其子任务分为两块：树的数据和 $q = 0$ 的数据，它们分别弱化了题目的一个条件，分别启发了正解的一个思路：树分治和圆方树。选手可以从不同的角度思考问题，选择不同的算法解决问题。正确实现本题的正解需要足够严谨的思维和处理细节的能力。除此之外本题的正解有一定代码量和实现细节，因此本题属于一道难度较大的题目。

8.2 拓展总结

圆方树可以非常高效、简洁、通用地处理仙人掌上的问题，同样也可以解决一般图上的一些询问简单路径的题目；链分治可以解决一类支持修改的、以子树为状态的 DP 问题，还能结合 LCT 以支持 Link/Cut 及任意子树询问。使用轻重链剖分来维护圆方树，可以高效地解决了一类支持单点修改的两点之间简单路径询问的问题。

如今，点双连通分量和支持修改的 DP 问题，对很多选手来说都具有一定的难度。我希望这道题能引发选手对链分治在树上带修改 DP 问题中的应用及圆方树的一些思考，能够有更多的扩展，解决更加广泛的问题。如果有同学在此方面的研究有所收获，欢迎来找我讨论。

9 致谢

1. 感谢中国计算机学会提供学习与交流的平台。

2. 感谢父母的养育之恩。

3. 感谢福州一中的陈颖老师的关心与教导。

4. 感谢福州一中的各位学长带给我的启发和指导。

5. 感谢国家集训队教练们带给我的帮助。

6. 感谢福州一中的刘一凡同学在我研究圆方树给予我巨大的帮助，并为圆方树命名。

7. 感谢钟知闲同学为本题验题、为本文审稿。

8. 感谢其他所有本文所参考过的资料的提供者。

9. 感谢各位在百忙之中抽出宝贵的时间阅读。

参考文献

[1] 漆子超,《分治算法在树的路径问题中的应用》

[2] Tarjan, R. E. (1972), ”Depth-first search and linear graph algorithms”, SIAM Journal on Computing, 1 (2): 146 – 160, doi:10.1137/0201010

[3] 陈俊锟, 钟知闲, ”Making Graphs into Trees”, WC2017 营员交流

[4] 任之洲,《浅谈启发式思想在信息学竞赛中的应用》

[5] 吴作凡,《〈火车司机出秦川〉命题报告》

[6] 徐寅展,《线段树在一类分治问题上的应用》

动态传递闭包问题的探究

绍兴市第一中学 孙耀峰

摘 要

本文对动态传递闭包问题进行了探究：对静态传递闭包问题、部分动态传递闭包问题及完全动态传递闭包问题分别给出相应的算法；并通过原创例题讲解动态传递闭包问题在信息学竞赛中的应用。

1 前言

“传递闭包”是图论中非常经典的问题，用于求解有向图中所有点对间的可达关系。目前传递闭包问题在算法竞赛中仍停留于静态层面，无法支持加删边操作，命题空间狭窄，题目格局不高。为顺应数据结构类算法由静态向动态发展的趋势，拓宽算法竞赛的命题选择，我对这方面问题展开了学习和探究。

本文将详细地介绍解决动态传递闭包问题的一些方法，包括最短路算法的拓展、树形图合并算法，以及基于线性代数的随机算法等等。同时，还将通过例题来讲解动态传递闭包问题在信息学竞赛中的应用。希望本文能将该问题带入选手们的视野，给所有参加信息学竞赛的同学带来或多或少的帮助。

本文第二节对动态传递闭包问题进行了描述，并给出相关的定义。第三至五节，分别对静态传递闭包问题、只有加边操作的部分动态传递闭包问题，以及完全动态传递闭包问题介绍了相应的解决方法，并在每一节都会给出例题。第六节，对本文内容作了总结和归纳。

2 引入问题

2.1 定义

定义**有向图**$G=(V,E)$，其中集合$V=\{1,2,..,n\}$表示点集，集合E表示边集。

称点v是点u的可达点，当且仅当图G中存在一条从点u到点v的路径。此时，称点u是点v的**祖先(ancestor)**，且点v是点u的**后代(descendant)**。特别的，点u既是自己的祖先，又

是自己的后代。同时，称由点u所有祖先点构成的集合为$anc(u)$，由点u所有后代点构成的集合为$desc(u)$。

若有向图$G^* = (V, E^*)$，与图G拥有相同的点集，且将边(u, v)置于集合E^*当且仅当图G中存在从点u到点v的路径，则称图G^*为图G的**传递闭包(transitive closure)**。

2.2 动态传递闭包问题

可以把动态传递闭包问题形式化地描述成以下形式 [7]：

给定有向图G，对其进行一系列的操作，包含以下三种类型：

- $Insert(x, y)$：在图G中加入一条从点x到点y的有向边；
- $Delete(x, y)$：在图G中删除一条从点x到点y的有向边；
- $Query(x, y)$：询问图G中是否存在从点x到点y的路径。

根据支持操作类型的多少，可以做出如下定义：

- 称既支持加边操作又支持删边操作的问题，为**完全动态传递闭包问题**；
- 称只支持加删边操作其中一种的问题，为**部分动态传递闭包问题**；
- 称不存在加删边操作的问题，为**静态传递闭包问题**。

若无特殊说明，本文都将用n和m来表示图G的点数和边数，用q来表示进行的操作次数。

3 静态传递闭包问题

3.1 Floyd-Warshall算法

Floyd-Warshall算法 [4]，通常用于解决所有点对间的最短路问题。同时，它还可以用于解决静态传递闭包问题。大致的思路，就是用逻辑或操作($\vee$)和逻辑与操作($\wedge$)，来代替求最短路时的算数操作min和+。

定义 3.1. 若图G中存在一条从点i到点j的路径，且中间点（除去点i和点j）都取自集合$\{1, 2, .., k\}$，则有$t_{i,j}^{(k)}$为1，否则$t_{i,j}^{(k)}$为0。

$t_{i,j}^{(k)}$可以这样得到：

$$t_{i,j}^{(0)} = \begin{cases} 0, & \text{若} i \neq j \text{且} (i,j) \notin E; \\ 1, & \text{若} i = j \text{或} (i,j) \in E. \end{cases}$$

对于$k \geq 1$，

$$t_{i,j}^{(k)} = t_{i,j}^{(k-1)} \vee (t_{i,k}^{(k-1)} \wedge t_{k,j}^{(k-1)}) \tag{3.1}$$

于是就能在$O(n^3)$的复杂度内求出$t_{i,j}^{(n)}$，从而得到传递闭包。

定义 3.2. 集合$T_i^{(k)} = \{j \mid t_{i,j}^{(k)} = 1\}$.

对于$k \geq 1$，根据式 (3.1)可得：

$$T_i^{(k)} = \begin{cases} T_i^{(k-1)}, & \text{若 } k \notin T_i^{(k-1)}; \\ T_i^{(k-1)} \cup T_k^{(k-1)}, & \text{若 } k \in T_i^{(k-1)}. \end{cases}$$

对于集合并操作(∪)和集合交操作(∩)等集合操作，可以用**压位**来优化。

令w表示机器字长，通常$w = 32$。对于集合S来说，假设S最多有n个元素，则可以用长度为n的01序列来表示集合S中每个元素是否存在。为了加速，可以将01序列中每隔w个位压成一个整数类型，统一进行位运算。这样就能在$O(\frac{n}{w})$的复杂度内对集合S进行操作。

运用这项技巧，就能在$O(\frac{n^3}{w})$的复杂度内求出$T_i^{(n)}$，从而得到传递闭包。

3.2 拓扑序

定义 3.3. 对于**有向无环图**（DAG）$G = (V, E)$来说，**拓扑序**是指图G中关于所有节点的一种线性次序，需要满足：对于所有边$(u, v) \in E$，点u在拓扑序中的位置要在点v的前面。

定理 3.1. 若图G包含环路，则不存在拓扑序。

定理 3.2. 可以在$O(n + m)$的复杂度[4]内求解有向无环图的拓扑序。

从后往前顺次枚举拓扑序中的点。假设当前枚举到点u，设点u的出边为$(u, v_1), (u, v_2), ..., (u, v_k)$，则可得：

$$desc(u) = desc(v_1) \cup desc(v_2) \cup ... \cup desc(v_k) \cup \{u\}$$

运用压位优化，就能在$O(\frac{nm}{w})$的复杂度内，针对**拓扑图**求出传递闭包。

3.3 强连通分量

定义 3.4. 有向图$G = (V, E)$的**强连通分量**(SCC)是指一个极大的点集$C \subseteq V$，对于任意的点$u, v \in C$，点u和点v都互相可达。

定理 3.3. 可以在$O(n + m)$的复杂度[4]内将有向图中所有的SCC都找出来。

若点x和点y属于同一个SCC，则满足$desc(x) = desc(y)$。故不妨将同一个SCC内的点缩成一个点后统一处理。

令有向图G'表示图G缩点后的图，则图G'为一张拓扑图。使用第 3.2节中的算法，就能求出图G'的传递闭包，对其稍加处理就能得到图G的传递闭包了。

该算法时间复杂度为$O(\frac{nm}{w})$。

3.4 例题一

例 1. Explosion.（2014 ACM/ICPC Asia Regional Beijing Online）

有n个房间，每个房间内有打开一些门的钥匙。初始时所有房间门都被锁了，且你手上没有任何钥匙。

当你没法用手上的钥匙打开房间门时，你会在还未被打开的房间中，随机选择一个并轰炸它，从而得到该房间内的所有钥匙。当你能用手上的钥匙打开房间门时，就会立即打开它并获得钥匙，不会选择轰炸。

求期望轰炸多少次，使得所有房间门都被打开。

数据范围：$n \le 2000$。

首先把问题转化为图论模型：

- 建立一张n个点有向图G，第i个房间对应图上的点i；
- 若房间i内有打开房间j的钥匙，则令点i向点j连一条有向边；
- 若轰炸了房间i，则图G中所有点$j \in desc(i)$对应的房间都被打开了。

根据期望的线性性，只需要对于每个房间计算它被轰炸的概率，全部累加就是答案了。

观察发现，房间i能被轰炸，当且仅当图G中所有点$j \in anc(i)$对应的房间都没有被轰炸。换言之，即房间i是这$|anc(i)|$个房间中第一个被选择轰炸的，则它被轰炸的概率为$\frac{1}{|anc(i)|}$。所以最后的答案为$\sum_{i=1}^{n} \frac{1}{|anc(i)|}$。

用第 3.1节或第 3.3节中的算法即可求出$anc(i)$，时间复杂度为$O(\frac{n^3}{w})$或$O(\frac{nm}{w})$。

3.5 例题二

例 2. 最大异或路径.（原创）

给定n个点m条边的有向图G，第i个点的权值为val_i。

定义点对(u, v)是合法的，当且仅当存在一条从点u到点v的路径。

求在所有合法的点对(u, v)中，val_u异或val_v的最大值是多少。

数据范围：$0 \le n, m, val_i \le 100000$.

令集合$S(u) = \{val_v \mid v \in desc(u)\}$，则问题就是要对所有点$u$，求出$val_u$与集合$S(u)$中某个数的异或最大值。

运用压位的技巧，可以把集合$S(u)$压成一共$\frac{|val_i|}{w}$个整数。对于每个整数，可以$O(1)$求出它在二进制位上一共有几个“1”，即包含集合$S(u)$中的几个元素。并对其求出前缀和。

采用贪心的思想，从二进制的最高位枚举至最低位，并枚举这一位是0还是1。对于枚举的答案，只需要判断在集合$S(u)$中权值位于一段区间内是否存在元素即可。通过之前预处理好的前缀和，就可以$O(1)$进行判断。

集合$S(u)$的求法和传递闭包类似，使用第 3.3节中的算法，就能在$O(\frac{(n+m)|val_i|}{w} + n\log|val_i|)$的复杂度内解决该问题。

4 只有加边操作的部分动态传递闭包问题

4.1 最短路算法

建立带权有向图$G' = (V, E')$：

- 对于图G中原有的边$(u, v) \in E$，在图G'中连上从点u到点v权值为0的边；
- 若第i次操作加入边(x_i, y_i)，则在图G'中连上从点x_i到点y_i权值为i的边。

定义 4.1. *若图G'中存在一条从点i到点j的路径，且途径边的权值均小于等于w，则$h_{i,j}^{(w)} = 1$，否则$h_{i,j}^{(w)} = 0$。*

定义 4.2. $g_{i,j} = \min\{w \mid h_{i,j}^{(w)} = 1\}$.

推论 4.1. 第$g_{i,j}$次操作之后，图G中就存在了一条从点i到点j的路径。

若能对所有点对i, j求出$g_{i,j}$，就能得到每次操作后的传递闭包了。

观察发现，这个问题等价于求所有点对间的最短路。使用Floyd-Warshall算法，时间复杂度为$O(n^3)$。使用Dijkstra算法配合斐波那契堆优化 [4] ，时间复杂度为$O(n^2 \log n + n(m + q))$。

值得注意的是，该算法是**离线**的。

4.2 对Floyd-Warshall算法的拓展

Floyd-Warshall算法求解传递闭包的本质，就是动态地加入“中转点”来更新传递闭包。事实上，中转点加入图中的顺序并不一定得是1,2,...,n，它其实可以将所有点以任意的顺序加入图中。这一优美的性质，使得它能在一定程度上支持动态问题。

通过下面这个例子来更好地理解它的应用。

4.2.1 例题三

例 3. 实时路况.（2016计蒜之道复赛）

给定n个点的无向图，定义$d(x,y,z)$表示不经过点y的从点x到点z的最短路。求解下式：

$$\sum_{x=1}^{n}\sum_{y=1}^{n}\sum_{z=1}^{n} d(x,y,z)$$

数据范围：$n \le 200$.

考虑运用分治的思想。

令过程$Work(l,r)$表示处理$y=l,l+1,..,r$时的答案，此时第$1,2,..,l-1,r+1,..,n$号点已经加入图中。同时，维护出$dist_{x,z}$表示所有点对间的最短路。

当$l=r$时，$O(n^2)$遍历$dist_{x,z}$就能得到$y=l$时的答案。

若$l \neq r$，令$mid=\lfloor \frac{l+r}{2} \rfloor$。将第$l,l+1,..,mid$号点加入图中，并递归$Work(mid+1,r)$；再将第$mid+1,mid+2,..,r$号点加入图中，并递归$Work(l,mid)$。每加入一个点，就用Floyd-Warshall算法来更新所有点对间的最短路。

时间复杂度为$O(n^3 \log n)$。

4.2.2 “中转边”

效仿Floyd-Warshall算法动态加入中转点的思想，对于每次操作加入的边，都拿它作为“中转边”来更新传递闭包。

假设当前操作加入边(u,v)。若已有$v \in desc(u)$，则边(u,v)的加入不会更新传递闭包，故忽视这次操作。

否则，令集合$C=(V-anc(v)) \cap anc(u)$，该集合是由所有能到点u但不能到点v的点构成。对于所有点$x \in C$，边(u,v)的加入会使得：

$$desc(x)=desc(x) \cup desc(v)$$

而对于所有点$x \notin C$，边(u,v)的加入不会更新$desc(x)$。

4.2.3 维护$anc(x)$

定义 4.3. *若有向图$\tilde{G}=(V,\tilde{E})$，与图G点集相同，且边集$\tilde{E}=\{(u,v) \mid (v,u) \in E\}$，则称图$\tilde{G}$为图$G$的反图。*

容易发现，图G中点x的祖先集合，等于图$\tilde{G}$中点x的后代集合。故每次加边操作，只需在图G中加入边(u,v)，在图$\tilde{G}$中加入边(v,u)，用维护$desc(x)$的方法来维护$anc(x)$就行了。

4.2.4 复杂度分析

因为每次对$desc(x)$进行集合取并时，新的集合$desc'(x)$都会比$desc(x)$新增元素。而不同的可达关系只有n^2对，意味着该集合取并操作最多只会进行n^2次。而每次加边操作，最坏情况下会进行n次集合取并操作。

运用压位优化，时间复杂度为$O(qn + \min\{n^2, qn\}\frac{n}{w})$。

4.2.5 小优化

可以使用压位优化，来加速遍历集合C中所有元素的效率。

根据之前的处理，集合C已经被$\frac{n}{w}$个整数表示出了。遍历这$\frac{n}{w}$个整数，若某个数二进制数位上有“1”（即该数不为0），则说明它包含了至少一个元素。取出其中所有“1”的位置就能找出集合C的所有元素。

综上，时间复杂度为$O(q\frac{n}{w} + \min\{n^2, qn\}\frac{n}{w})$。

小结 容易发现，该算法在稠密图且操作数较多的情况下最为优秀。

4.3 树形图合并算法

定义 4.4. 对于有向图$G = (V, E)$而言，以点x为根的**树形图**$T_x = (V, E_x)(E_x \subseteq E)$，是指满足以下条件的**拓扑图**：

- 根节点x没有入度边，其余节点**至多**有一条入度边；
- 若图G中点$y \in desc(x)$，则在T_x中同样存在一条从根节点x到点y的路径；
- 若图G中点$y \notin desc(x)$，则在T_x中点y没有入度边。

定义 4.5. 若在树形图$T_x = (V, E_x)$中，有边$(u, v) \in E_x$，则称点v是点u的儿子。

定义 4.6. 在树形图$T_x = (V, E_x)$中，点u的子树集合$subtree_x(u)$为：

$$subtree_x(u) = \{v \mid \text{在} T_x \text{中存在从点} u \text{到点} v \text{的路径}\}$$

推论 4.2. 在树形图$T_x = (V, E_x)$中，对于任意点$u \in V$，有$subtree_x(u) \subseteq desc(u)$。

在这个算法中，将对每个点$x \in V$，建立树形图T_x来维护$desc(x)$。对于每次加边操作，只需将某些树形图进行合并即可。

4.3.1 合并流程

假设当前操作加入边(u,v)。若已有$v \in desc(u)$，则边(u,v)的加入不会更新传递闭包，故忽视这次操作。

否则，令集合$C = (V - anc(v)) \cap anc(u)$。对于所有点$x \in C$，将树形图$T_v$合并于树形图$T_x$中点$u$的儿子上。

令过程$Merge(x,y,a,b)$表示，将树形图T_y中点b的子树，合并于树形图T_x中点a的儿子上。则对于所有点$x \in C$，只需调用$Merge(x,v,u,v)$即可。

具体的$Merge(x,y,a,b)$过程如下：

1. 在树形图T_x中加入边(a,b)，即让点b成为点a的儿子；
2. 枚举树形图T_y中点b的所有儿子w：
   - 若$w \in desc(x)$，根据推论 4.2 可得：
     $$subtree_y(w) \subseteq desc(w) \subseteq desc(x)$$
     故不需要继续递归树形图T_y中点w的子树；
   - 若$w \notin desc(x)$，则调用递归过程$Megre(x,y,b,w)$。

4.3.2 伪代码

通过以下伪代码能更好地理解树形图合并的过程：

算法 1 树形图合并

```
0: procedure MERGE(x, y, a, b)
1: insert b in T_x as a child of a.
2: for each w child of b in T_y do
3:     if w is not in desc(x) then
4:         MERGE(x, y, b, w)
5:     end if
6: end for
```

4.3.3 复杂度分析

每次调用过程$Merge(x,y,a,b)$时，都会把点b加入树形图T_x中，并枚举树形图T_y中点b的所有儿子。而一个点不可能多次加入同一个树形图中。

不妨把树形图T_y中点b的儿子个数放缩成，当前操作后图G中点b的出度边数量。则一个树形图在合并过程中，最多将图G中所有的边都遍历到一次。即每次操作加入的边会有$O(n)$的贡献，则该合并过程每次操作均摊复杂度为$O(n)$。

同时，每次操作找出集合C中所有点只需要$O(n)$遍历即可。则该算法每次操作均摊复杂度为$O(n)$。

小结 容易发现，该算法在稀疏图且操作数较少的情况下最为优秀，这与第 4.2 节中提及的算法互补。同时，该算法还能在$O(n)$的复杂度内求出一条从点u到点v的路径。

5 完全动态传递闭包问题

5.1 离线算法

5.1.1 时间线段树

建立一棵大小为q的时间线段树 [6]，它的一共q个叶子就代表着q次操作。对于时间线段树上每个节点i，它的覆盖区间为$[l_i, r_i]$。

若一条边(u, v)在第l次操作加入到图G中，在第r次操作从图G中被删除，则说明边(u, v)存在于时间线段树上的区间$[l, r-1]$中。可以在时间线段树上找出代表这段区间的$O(\log q)$个节点，在这些节点上都存上边(u, v)。

因原图G有m条边，进行了q次操作，故所有边一共会有$O(m+q)$个时间区间。这些区间会被对应到时间线段树上一共$O((m+q)\log q)$个节点。

遍历时间线段树，到一个节点，就把存在这个节点上的所有边加入图中。每加入一条边，就使用第 4.2节中提及的算法维护传递闭包。

时间复杂度为$O((m+q)\log q\frac{n^2}{w})$。值得注意的是，该算法是**离线**的。

5.1.2 例题四

例 4. 小G的难题.（原创）

给定一张n个点m条边的有向图G，第i个点权值为val_i。

定义$F(u, v)$表示：若图G中存在一条从点u到点v的路径，那么$F(u, v) = 1$，否则$F(u, v) = 0$。

进行q次操作，包含以下两种类型：

- 给出点p，再给出tot个数a_i，在图中加入点a_i到点p的边；
- 给出点p，再给出tot个数a_i，在图中删除点a_i到点p的边。

需要在每次操作后，回答下式的值：

$$\sum_{u=1}^{N}\sum_{v=1}^{N} F(u,v)\times(val_u \ xor \ val_v)$$

数据范围：$tot < n \le 400$，$q \le 800$。

使用第 5.1.1节中的算法，就能在$O((m+q\times tot)\log q\frac{n^2}{w})$的复杂度内解决该问题，然而还可以继续优化。

因为每次操作，只会加删某个点的入度边，所以不妨考虑用“中转点”代替“中转边”来更新传递闭包。

具体来说，令集合$pre(x)=\{y \mid (y,x)\in E\}$，即点$x$的入度点集合。把点$x$加入图中时，枚举所有点$t\in V$，若满足：

$$desc(t)\cap pre(x)\neq\emptyset$$

则令：

$$desc(t)=desc(t)\cup desc(x)$$

这样就能在$O(\frac{n^2}{w})$的复杂度内，维护加点后的传递闭包。

因初始有n个点，进行了q次操作，故所有点一共会有$O(n+q)$个时间区间。这些区间会被对应到时间线段树上一共$O((n+q)\log q)$个节点。

遍历时间线段树，到一个节点，就把存在这个节点上的所有$pre(x)$集合加入图中，而加入一个$pre(x)$的复杂度为$O(\frac{n^2}{w})$。

综上，时间复杂度为$O((n+q)\log q\frac{n^2}{w}+qn^2)$。

5.1.3 例题四的拓展

若每次操作既会加删一个点的入度边，又会加删一个点的出度边，该怎么做呢？

可以在记录$pre(x)$的同时，再记录集合$suf(x)$，即点x的出度点集合。并且：

- 对于所有点$y\in pre(x)$，记录边(y,x)被加入的操作时间；
- 对于所有点$y\notin pre(x)$，记录边(y,x)被删除的操作时间。

对$suf(x)$中的点也做类似的处理。

这样，在加入点x时，根据$pre(x)$和$suf(x)$以及它们对应的操作时间，就可以判断点x的每条出入度边是否存在。使用类似例题 4的算法就能求出传递闭包。

时间复杂度为$O((n+q)\log q\frac{n^2}{w}+qn^2)$。

5.2 一种基于线性代数的算法

5.2.1 问题转化

定义 5.1. 有向图$G=(V,E)$的一个环覆盖是指用若干个环去覆盖图中所有的点，使得每个点都恰好只在一个环上。

假设要询问图G中是否存在从点i到点j的路径，则可对图G做如下处理，就能把问题转换为环覆盖问题：

1. 给每个点连一条自环（假定原图是没有自环的）；
2. 删除点i的所有入边，以及点j的所有出边，包括刚才连的自环；
3. 连上一条从点j到点i的边。

令新图为G'，则图G中存在从点i到点j的路径，当且仅当图G'存在**环覆盖**。

定义 5.2. 对于有向图$G=(V,E)$，定义$n\times n$的矩阵A为：

$$A_{i,j}=\begin{cases} x_{i,j}, & 若i=j或边(i,j)\in E; \\ 0, & 其他情况. \end{cases}$$

其中$x_{i,j}$是一个**变量**，因此矩阵A中一共有$n+m$个变量。

定义 5.3. 定义矩阵$A\natural(j,i)$为对矩阵A进行如下操作后的矩阵：

- 将矩阵A中第j行和第i列的所有元素全部清空为0；
- 令$A_{j,i}=x_{j,i}$。

定理 5.1. 令矩阵$Z=A\natural(j,i)$，则图G中存在从点i到点j的路径当且仅当：

$$\sum_{\pi\in\sum_n}\prod_{k=1}^{n}Z_{k,\pi(k)}\neq 0 \tag{5.1}$$

证明. 枚举排列π的意义为枚举一个可能可行的环覆盖方法，其中$\pi(k)$表示点k的后继点。若对于所有点k都满足$Z_{k,\pi(k)}$不为0，则说明该环覆盖方法合法。 □

定理 5.2. 式 (5.1)成立，当且仅当$\det Z\neq 0$。

证明. 式 (5.1)左边的积和式，实为一个n^2元n阶的多项式。而根据行列式的定义：

$$\det Z=\sum_{\pi\in\sum_n}sign(\pi)\prod_{k=1}^{n}Z_{k,\pi(k)}$$

$\det Z$同样为一个n^2元n阶的多项式，且相对于式 (5.1)的积和式只是在每一项前面加了+1或-1的系数。这系数是不会影响该积和式是否恒为0的，故定理 5.2得证。 □

推论 5.1. 图G中存在从点i到点j的路径当且仅当$\det(A\not{i}(j,i)) \neq 0$。

证明. 可以由定理 5.1和定理 5.2得到。 □

5.2.2 一些线性代数知识

定义 5.4. *$A^{i,j}$为矩阵A去掉第i行和第j列后所剩下的子矩阵。*

定义 5.5. *矩阵A关于第i行和第j列的余子式（记做$M_{i,j}$）为$\det A^{i,j}$。*

定义 5.6. *矩阵A关于第i行和第j列的代数余子式（记做$C_{i,j}$）为$(-1)^{i+j}M_{i,j}$。*

定义 5.7. *矩阵A的余子矩阵为一个$n \times n$的矩阵C，其中$C_{i,j} = (-1)^{i+j}M_{i,j}$。*

定义 5.8. *矩阵A的伴随矩阵为A的余子矩阵的转置矩阵，即：*

$$\operatorname{adj} A = C^T$$

定理 5.3. *A为一个$n \times n$的矩阵，对于任意一行$i \in \{1, 2, .., n\}$有：*

$$\det A = \sum_{j=1}^{n} A_{i,j}(\operatorname{adj} A)_{j,i}$$

定理 5.4. *若矩阵A可逆，则有：*

$$A^{-1} = \frac{\operatorname{adj} A}{\det A}$$

考虑运用以上的线性代数知识来简化原问题。对于图G的矩阵A来说，有：

$$\det(A\not{i}(j,i)) = (\operatorname{adj} A)_{i,j} \tag{5.2}$$

又根据定理 5.4，有：

$$\operatorname{adj} A = \det A \times A^{-1}$$

因此，若能在每次操作后对于图G动态维护出矩阵A的行列式以及逆矩阵，就能得到矩阵A的伴随矩阵。并根据式 (5.2)，就能得到$\det(A\not{i}(j,i))$；再根据推论 5.1，就能够判断图G中是否存在从点i到点j的路径；从而就能得到图G的传递闭包。

直接求解矩阵A的伴随矩阵是很困难的，所幸该问题不需要求解它们的确切值，只要判断它们的多项式是否为0即可。那就可以根据这个引理来简化问题：

引理 5.1. Zippel-Schwartz lemma. [1]

对于域$\mathbb{F}$上的一个不恒为0的n元d阶多项式$P(x_1, x_2, ..., x_n)$，设$r_1, r_2, ..., r_n$为n个$\mathbb{F}$中独立选取的随机数，则：

$$\Pr[P(r_1, r_2, ..., r_n) = 0] \leq \frac{d}{|\mathbb{F}|}$$

因此，只需要将矩阵A中的每个变量，赋值成域$\mathbb{F}$下的一个随机数。根据域$\mathbb{F}$下$\det(A\not{i}(j,i))$是否为0，就能判断图G中是否存在从点i到点j的路径了。

因为此类问题数据规模都较小，且取大质数$P = 10^9 + 7$作为模域，那么出错概率就可以忽略不计了。

5.2.3 动态维护行列式及逆矩阵

对于初始给定的图G，可以使用高斯消元 [2]在$O(n^3)$的复杂度内，求出矩阵A的行列式以及逆矩阵。

对于每次加删边操作，假设当前操作修改了矩阵A中第i列的某个位置。令列向量δ表示矩阵A第i列的变化，令矩阵A'表示进行当前操作后的矩阵，则有：

$$A'_{j,i} = A_{j,i} + \delta_j, j \in \{1, 2, .., n\}$$

令行向量e_i^T表示$(0, 0, ..., 1, ..., 0)$，其中1的所在列为第i列。那么，

$$\delta e_i^T = \begin{pmatrix} 0 & \cdots & 0 & \delta_1 & 0 & \cdots & 0 \\ & & & \vdots & & & \\ 0 & \cdots & 0 & \delta_n & 0 & \cdots & 0 \end{pmatrix}$$

于是有，

$$A' = A + \delta e_i^T$$

若矩阵B满足$A' = A \times B$，则矩阵B为：

$$B = I + A^{-1}\delta e_i^T = I + (A^{-1}\delta)e_i^T = I + be_i^T$$

其中列向量$b = A^{-1}\delta$，这可以在$O(n^2)$的复杂度内计算得到。

接着把矩阵B写下来：

$$B = \begin{pmatrix} 1 & & & b_1 & & & \\ & \ddots & & b_2 & & & \\ & & 1 & \vdots & & & \\ & & & 1 + b_i & & & \\ & & & \vdots & 1 & & \\ & & & b_{n-1} & & \ddots & \\ & & & b_n & & & 1 \end{pmatrix}$$

通过观察，可以发现$\det B = 1 + B_i$。那么根据下式：

$$\det A' = \det A \times \det B$$

就能得到操作后矩阵的行列式了。

因为矩阵B的形式很简单，所以可以通过观察得到逆矩阵B^{-1}：

$$B^{-1}=\begin{pmatrix} 1 & & & -\frac{b_1}{1+b_i} & & \\ & \ddots & & -\frac{b_2}{1+b_i} & & \\ & & 1 & \vdots & & \\ & & & \frac{1}{1+b_i} & & \\ & & & \vdots & 1 & \\ & & & -\frac{b_{n-1}}{1+b_i} & & \ddots \\ & & & -\frac{b_n}{1+b_i} & & 1 \end{pmatrix}$$

逆矩阵B^{-1}为一个稀疏矩阵，其中的非0元素只有$O(n)$个。那么根据下式：

$$A'^{-1}=B^{-1}\times A^{-1}$$

就可以在$O(n^2)$的复杂度内，利用稀疏矩阵的矩阵乘法得到矩阵A'^{-1}。

动态维护出行列式及逆矩阵后，稍加处理就能得到图G的传递闭包。该算法单次操作的最坏时间复杂度为$O(n^2)$。

小结 该算法的主要流程为：

1. 通过对图G的处理，把传递闭包问题转化为环覆盖问题；
2. 对图G定义矩阵A，当$\det(A\not{i}(j,i))\neq 0$时，就说明存在从点$i$到点$j$的路径；
3. 运用线性代数以及引理 5.1，将问题简化为在模域下求矩阵A的伴随矩阵；
4. 进行数学推导，在$O(n^2)$的复杂度内动态维护矩阵$\operatorname{adj} A$，并得到传递闭包。

会发现该算法不只局限于每次操作只加删一条边，它能支持每次操作对一个点的若干出入度边进行加删，且效率不变。

6 总结

本文对动态传递闭包问题提出了以下算法：

- 提出用Floyd-Warshall算法、拓扑排序以及强连通分量算法，来解决静态传递闭包问题；
- 提出用最短路算法、Floyd-Warshall算法的拓展以及树形图合并算法，来解决部分动态传递闭包问题；

- 提出用时间线段树算法以及一种基于线性代数的算法，来解决完全动态传递闭包问题。

因为目前的信息学竞赛中，与动态传递闭包相关的问题还非常少，所以这类问题还会有很大的挖掘空间。希望这篇文章能够起到抛砖引玉的作用，今后在信息学竞赛中能有越来越多对这方面问题的研究和应用。

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和董烨华老师多年来给予的关心和教诲。
感谢国家集训队教练张瑞喆和余林韵的指导。
感谢清华大学的张恒捷和任之洲学长在我写作过程中给予的思路和启发。
感谢绍兴一中的孙奕灿，洪华敦、冯哲、叶珈宁、任轩笛等同学对我的帮助。
感谢所有对我有过帮助的老师和同学。

参考文献

[1] Wikipedia, Schwartz–Zippel lemma.

[2] Wikipedia, Gaussian elimination.

[3] 刘汝佳，黄亮，算法艺术与信息学竞赛。

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

[5] 孙耀峰，小G的难题解题报告， 2017年集训队第二次作业。

[6] 徐寅展，线段树在一类分治问题上的应用， 2014年集训队论文。

[7] Camil Demetrescu, Giuseppe F. Italiano. *Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ Barrier.* FOCS 2000: 381-389.

[8] M. Nivat. *Amortized Efficiency of a Path Retrieval Data Structure.* TCS 1986: 273-281.

[9] Piotr Sankowski. *Dynamic Transitive Closure via Dynamic Matrix Inverse.* FOCS 2004: 509-517.

《A + B Problem》命题报告

安徽师范大学附属中学 汪乐平

摘 要

此题是2016年集训队互测第一场中的一道题，本文主要介绍了此题的解法和命题背景。

1 试题

1.1 问题描述

题目名字是吸引你点进来的~~~

JOHNKRAM最近在研究图论。今天他遇到了这样一道题：给一张有向图，保证没有重边和自环。求把这张图的强连通分量缩成点之后，有多少个点入度为0。

JOHNKRAM使用*tarjan*算法轻而易举地切掉了这道题。但他发现有很多人代码写得比他短多了，于是他要来了程序，结果发现这个程序是打印1的= =。难道随机数据真的这么水吗？于是他随机生成了很多张有向图，结果他发现答案真的全是1= =。于是他更改了随机生成的方式，只生成每个强连通分量大小都属于某个集合S的有向图，结果答案立刻就变大了。

现在JOHNKRAM把那些人都卡掉了，但他想证明一下数据的强度，所以他提出了一个问题：对于所有带标号点数为$i(1 \le i \le n)$的每个强连通分量大小都属于集合S的有向图，之前问题的答案的期望是多少？他发现自己不会证明了，于是他来向你请教。

1.2 输入格式

输入第一行包含1个正整数n，表示生成的有向图的最大点数。

接下来n行，第i行包含一个整数s_i。如果$s_i = 1$，则i在集合S中，否则i不在集合S中。

1.3 输出格式

共输出n行，每行包含一个整数。第i行的整数表示对于所有带标号点数为$i(1 \le i \le n)$的

每个强连通分量大小都属于集合S的有向图，之前问题的答案的期望mod 998244353的结果，如果没有合法的有向图则输出0。设期望值为$\frac{a}{b}$（a和b为互质的正整数），你输出的整数为x，则你需要保证$bx \equiv a(mod\ 998244353)$且$0 \leq x < 998244353$。

1.4 样例输入

```
3
1
0
0
```

1.5 样例输出

```
1
332748119
519087065
```

1.6 样例说明

点数为1的有向图中，答案为1的合法有向图有1张，所以答案为1，mod 998244353后为1。

点数为2的有向图中，答案为1的合法有向图有2张，答案为2的合法有向图有1张，所以答案为$\frac{4}{3}$，mod 998244353后为332748119。

点数为3的有向图中，答案为1的合法有向图有15张，答案为2的合法有向图有9张，答案为3的合法有向图有1张，所以答案为$\frac{36}{25}$，mod 998244353后为519087065。

1.7 数据规模和约定

对于全部测试数据，$1 \leq n \leq 100000$，$0 \leq s_i \leq 1$。

测试点编号	$n \leq$	其他约定
1	4	无
2	1000	$\forall 1 \leq i \leq \lceil \frac{n}{2} \rceil, s_i = 0$
3	1000	$\forall 1 \leq i \leq n, s_i = [i == 1]$
4	1000	$\forall 1 \leq i \leq n, s_i = [i == 1]$
5	1000	$\forall 1 \leq i \leq n, s_i = [i == 1]$
6	1000	$\forall 1 \leq i \leq \lceil \frac{n}{3} \rceil, s_i = 0$
7	1000	$\forall 1 \leq i \leq \lceil \frac{n}{3} \rceil, s_i = 0$
8	1000	$\forall 1 \leq i \leq \lceil \frac{n}{3} \rceil, s_i = 0$
9	1000	$\exists x, \forall 1 \leq i \leq n, s_i = [i == x]$
10	1000	$\exists x, \forall 1 \leq i \leq n, s_i = [i == x]$
11	1000	$\exists x, \forall 1 \leq i \leq n, s_i = [i == x]$
12	1000	无
13	1000	无
14	1000	无
15	100000	无
16	100000	无
17	100000	无
18	100000	无
19	100000	无
20	100000	无

2 算法介绍

后文所有提到图的地方均指带标号图。

2.1 算法一

我们暴力枚举每一个有向图，判定是否符合条件并计算答案。一个有向图的边数是$O(n^2)$级别的，n个点的有向图共有$2^{n(n-1)}$个，所以算法一的时间复杂度是$O(n^2 * 2^{n(n-1)})$。

2.2 算法二

对于测试点2，强连通分量点数大于$\lceil \frac{n}{2} \rceil$，转换一下条件可得合法的有向图一定是强连通的，而强连通图的答案一定是1，所以直接把s数组原样输出即可。算法二的时间复杂度是$O(n)$。

2.3 算法三

对于测试点3 ~ 5，强连通分量点数只能为1，即合法的有向图一定是有向无环图。我们设i个点的有向无环图个数为g_i，i个点且有j个点入度为0的有向无环图个数为$f_{i,j}$。显然，$g_i = \sum_{j=0}^{i} f_{i,j}$。

对于一个点集S，当$\forall i \in S$，点i在有向无环图G中的入度为0时，我们称S是G的一个根。定义一个有根有向无环图指一个有序对(G,S)，满足点集S是有向无环图G的一个根。容易看出，一个有j个点入度为0的有向无环图G_x对应了2^j个有根有向无环图，即有2^j个有根有向无环图满足$G = G_x$。另一方面，我们对于一个有根有向无环图，把所有在S中的点和端点在S中的边删掉，剩下的仍然是一个有向无环图。于是我们可以列出一个等式：

$$\sum_{i=0}^{n} f_{i,n} * 2^i = \sum_{i=0}^{n} g_{n-i} * 2^{i(n-i)} * \binom{n}{i}$$

其中$2^{i(n-i)} * \binom{n}{i}$表示枚举每条可能被删掉的边是否存在及每个点是否在$S$中。我们把左式$2^i$用二项式定理拆开并稍作变换，可得：

$$\sum_{i=0}^{n} \sum_{j=i}^{n} f_{j,n} * \binom{j}{i} = \sum_{i=0}^{n} g_{n-i} * 2^{i(n-i)} * \binom{n}{i}$$

注意到此时两边的i的意义是相同的，都表示S的大小，于是我们把两边同乘上$(-1)^i$，得到：

$$\sum_{i=0}^{n} (-1)^i * \sum_{j=i}^{n} f_{j,n} * \binom{j}{i} = \sum_{i=0}^{n} g_{n-i} * (-1)^i * 2^{i(n-i)} * \binom{n}{i}$$

把左式再次进行变换，得到：

$$\sum_{i=0}^{n} f_{i,n} * \sum_{j=0}^{i} (-1)^j * \binom{i}{j} = \sum_{i=0}^{n} g_{n-i} * (-1)^i * 2^{i(n-i)} * \binom{n}{i}$$

由二项式定理得$\sum_{j=0}^{i} (-1)^j * \binom{i}{j} = [i == 0]$，而$f_{0,n} = [n == 0]$，于是左式变成了$[n == 0]$。我们把$g_n$移到左边，$[n == 0]$移到右边，再两边同乘$-1$，可得：

$$g_n = [n == 0] - \sum_{i=1}^{n} g_{n-i} * (-1)^i * 2^{i(n-i)} * \binom{n}{i}$$

于是我们得到了一个可以在$O(n^2)$时间内计算出前n项g_i的递推式。接下来考虑怎么计算所有有向无环图的答案的总和。

我们考虑一个点x对总答案的贡献。当点x入度为0时对总答案的贡献为1，否则为0。于是我们只需要考虑有多少个i个点的有向无环图满足点x入度为0即可。显然点x没有入边，我们把点x和点x发出去的边全部删掉，剩下的是一个$i-1$个点的有向无环图。点x最多发出$i-1$条边，这$i-1$条边存不存在都是合法的。所以一个点x对答案的贡献是$g_{i-1} * 2^{i-1}$。每个点对总答案的贡献是相同的，所以所有i个点的有向无环图的答案的总和就是$i * g_{i-1} * 2^{i-1}$，直接根据g_{i-1}计算即可。算法三的时间复杂度是$O(n^2)$。

2.4 算法四

对于测试点6 ~ 8，强连通分量点数大于$\lceil \frac{n}{3} \rceil$，也就是说最多有2个强连通分量。当两个强连通分量之间没有边时答案等于2，否则答案等于1，只需要计算合法的有向图个数及两个强连通分量之间没有边的合法有向图个数即可算出总答案。所以现在的关键是计算i个点的强连通图的个数。

把一个有向图的强连通分量缩成点之后剩下的是一个有向无环图，于是我们可以套用之前推出的递推式。但注意我们现在已知的是“有向无环图”的个数，而且每个点对应的实际上是强连通分量，所以我们还得再添加系数。

定义一个集合S的权值为$(-1)^{|S|}$。设d_i表示i个点的强连通图个数，e_i表示所有元素是强连通图的集合中，集合内元素总点数为i的集合的权值和，h_i表示i个点的有向图个数。根据之前的思路进行推导，可以得到一个等式（过程省略）：

$$\sum_{i=0}^{n} h_{n-i} * e_i * 2^{i(n-i)} * \binom{n}{i} = [n == 0]$$

如果没有其他限制的话，$h_i = 2^{i(i-1)}$。于是我们可以在$O(n^2)$的时间复杂度内计算出所有的e_i。接下来考虑如何根据e_i计算d_i。

因为图是带标号的，所以我们枚举点1所在的强连通图的点数即可得到e_i关于d_i的递推式：

$$e_n = \sum_{i=1}^{n} -d_i * e_{n-i} * \binom{n-1}{i-1}$$

移项可以得到d_i关于e_i的递推式：

$$d_n = -e_n + \sum_{i=1}^{n-1} -d_i * e_{n-i} * \binom{n-1}{i-1}$$

计算出所有d_i之后，剩下的部分就很简单了，只需要枚举强连通分量个数及每个强连通分量的点数并乘上对应的系数即可。算法四的时间复杂度是$O(n^2)$。

2.5 算法五

对于测试点9 ~ 11，所有强连通分量大小是同一个数，设这个数是x。我们先使用算法四中的递推式计算出所有d_i，然后我们把所有$i \neq x$的d_i置为0，得到了一个新的d_i，记为d'_i。我们用d'_i计算出新的e_i，记作e'_i。再根据e'_i计算新的h_i，记作h'_i。h'_i就是i个点的合法有向图个数。

至于所有i个点的合法有向图的答案的和，仍然类似于算法三，考虑每个强连通图对答案的贡献，可以算出i个点时的总答案等于$d'_x * h'_{i-x} * 2^{x(i-x)} * \binom{i}{x}$，直接计算即可。算法五的时间复杂度是$O(n^2)$。

2.6 算法六

我们考虑如何计算一般情况下，i个点的合法有向图个数和所有i个点的合法有向图的答案的和。

首先是合法有向图个数。注意到算法五中计算合法有向图个数的方法很容易扩展到限定强连通分量大小属于某一个集合的情况，即我们设$d'_i = d_i * s_i$，套用算法五中的方法计算出h'_i即可。

接着是所有合法有向图的答案和。设ans_i表示所有i个点的合法有向图的答案的和。现在强连通分量点数的取值从一种变成了多种，但不同点数的强连通图对答案的贡献是独立的，我们只需要枚举强连通图的点数，分开计算贡献即可。所以可以得到一个递推式：

$$ans_n = \sum_{i=1}^{n} d'_i * h'_{n-i} * 2^{i(n-i)} * \binom{n}{i}$$

于是我们可以根据d'_i和h'_i在$O(n^2)$的时间内计算出所有的ans_i。所以算法六的时间复杂度是$O(n^2)$。

2.7 算法七

我们考虑使用生成函数来对递推式进行处理。

令$D(x) = \sum_{i\geq 0} \frac{d_i * x^i}{i!}$，$D_1(x) = \sum_{i\geq 0} \frac{d'_i * x^i}{i!}$，$E(x) = \sum_{i\geq 0} \frac{e_i * x^i}{i!}$，$E_1(x) = \sum_{i\geq 0} \frac{e'_i * x^i}{i!}$，$H(x) = \sum_{i\geq 0} \frac{h_i * x^i}{i!}$，$H_1(x) = \sum_{i\geq 0} \frac{h'_i * x^i}{i!}$，$A(x) = \sum_{i\geq 0} \frac{ans_i * x^i}{i!}$。首先考虑$D(x)$和$E(x)$，$D_1(x)$和$E_1(x)$之间的关系：

$$e_n = \sum_{i=1}^{n} -d_i * e_{n-i} * \binom{n-1}{i-1}$$

移项、转换成生成函数形式可得：

$$E'(x) = -D'(x)E(x)$$

移项、两边同时积分可得：

$$\ln E(x) = -D(x)(\int \frac{F'(x)}{F(x)} \mathrm{d}x = \ln F(x))$$

两边同时Exp，得到：

$$E(x) = \mathrm{e}^{-D(x)}$$

$D_1(x)$和$E_1(x)$的关系是类似的。

接下来考虑$E(x)$和$H(x)$，$E_1(x)$和$H_1(x)$，$A(x)$和$D_1(x)$、$H_1(x)$之间的关系。注意到递推式中出现了$2^{i(n-i)}$这个系数，而这个系数并不能方便地拆开。我们需要使用一种新的生成函数。定义一种一元运算符Δ。设$F(x) = \sum_{i\geq 0} \frac{f_i * x^i}{i!}$，定义$\Delta F(x) = \sum_{i\geq 0} \frac{f_i * x^i}{i! * 2^{\binom{i}{2}}}$。我们称$\Delta F(x)$为数列$f_i$的组合生成函数。

利用等式$i(n-i)=\binom{n}{2}-\binom{i}{2}\binom{n-i}{2}$，我们可以把$2^{i(n-i)}$拆成$\frac{2^{\binom{n}{2}}}{2^{\binom{i}{2}}*2^{\binom{n-i}{2}}}$。这样，我们就可以计算出这些组合生成函数之间的关系：

$$\Delta E(x)\Delta H(x)=1$$

$$\Delta E_1(x)\Delta H_1(x)=1$$

$$\Delta A(x)=\Delta H_1(x)\Delta D_1(x)$$

现在，我们可以写出算法的整个流程。

$$\Delta E(x)=(\Delta H(x))^{-1}$$

$$D(x)=-\ln E(x)$$

$$d_i'=d_i*s_i$$

$$E_1(x)=\mathrm{e}^{-D_1(x)}$$

$$\Delta H_1(x)=(\Delta E_1(x))^{-1}$$

$$\Delta A(x)=\Delta H_1(x)\Delta D_1(x)$$

计算期望直接用$A(x)$和$H_1(x)$里的系数相除即可。注意到需要进行的不能在$O(n)$时间内实现的操作包括多项式乘法、多项式求逆、多项式Ln、多项式Exp，时间复杂度就取决于这四个操作的时间复杂度。

多项式乘法可以使用快速数论变换在$O(n\log n)$的时间内计算。

多项式求逆我们使用倍增法。设原多项式为$P(x)$，$P(x)$在$mod\ x^{2^t}$意义下的逆元为$F_t(x)$。考虑如何根据$F_t(x)$计算$F_{t+1}(x)$。

$$P(x)F_t(x)\equiv 1(mod\ x^{2^t})$$

$$P(x)F_t(x)-1\equiv 0(mod\ x^{2^t})$$

$$(P(x)F_t(x)-1)^2\equiv 0(mod\ x^{2^{t+1}})$$

$$1\equiv 2F_t(x)P(x)-P(x)^2F_t(x)^2(mod\ x^{2^{t+1}})$$

$$F_{t+1}(x)\equiv 2F_t(x)-P(x)F_t(x)^2(mod\ x^{2^{t+1}})$$

于是直接多项式乘法计算即可。时间复杂度$T(n)=T(n/2)+O(n\log n)$，解得$T(n)=O(n\log n)$。

多项式Ln直接套用公式$\ln F(x)=\int\frac{F'(x)}{F(x)}\mathrm{d}x$计算。多项式求导和多项式求积分的时间复杂度都是$O(n)$，所以多项式$Ln$的时间复杂度就是$O(n\log n)$。

多项式Exp使用牛顿迭代法解决。设方程为$G(F(x))=0$，$F_t(x)\equiv F(x)(mod\ x^{2^t})$。则牛顿迭代法公式为：

$$F_{t+1}(x)\equiv F_t(x)-\frac{G(F_t(x))}{G'(F_t(x))}(mod\ x^{2^{t+1}})$$

在多项式Exp中，$G(F(x)) = e^P(x) - F(x)$，这个方程不方便计算，改为$G(F(x)) = \ln F(x) - P(x)$。代入牛顿迭代法公式得：

$$F_{t+1}(x) \equiv F_t(x) - (\ln F_t(x) - P(x)) * F_t(x) (mod\ x^{2^{t+1}})$$

使用多项式Ln、多项式乘法计算即可。时间复杂度$T(n) = T(n/2) + O(n \log n)$，解得$T(n) = O(n \log n)$。

现在这四种操作的时间复杂度都是$O(n \log n)$，所以算法七的时间复杂度是$O(n \log n)$。

3 数据生成方式

注意到数据强度和s_i的值并没有直接联系，所以除了有特殊要求的测试点之外，其他的测试点均为随机生成。

4 得分情况

集训队选手中，有1人得到了100分，3人得到了70分，1人得到了35分，1人得到了25分，2人得到了20分，1人得到了15分，1人得到了5分，2人得到了0分。综合来说，本题属于难度较大，区分度较高的题目。

5 命题背景

在最近几年的信息学竞赛当中，计数题的出现频率非常高，而关于图的计数题也是为数不少。但是，绝大多数关于图的计数题都是关于无向图的，而关于有向图的计数题却是极为少见。究其原因，在于有向图比起无向图，多了“边的方向”这个变量，而这个变量关于答案的影响又非常难以计算。

我在学习信息学的过程中，对生成函数进行过一些研究。某天，在看到一道关于有向图的计数题之后，我突然想到：能否使用生成函数解决有向图计数问题呢？随后，我查阅了一些资料，并进行了一定的数学推导之后，成功地推出了有向无环图数量与强连通图数量的递推式，并使用生成函数将时间复杂度优化到了$O(n \log n)$。然后我又对递推式进行了一些转化，并将计数变成期望，最终命了这道题。

6 感谢

感谢中国计算机学会提供交流和学习的平台。

感谢叶国平老师在学习生活等方面的帮助。

感谢彭雨翔、金策等前辈关于生成函数的研究。
感谢罗哲正同学为本文审稿。
感谢毛啸同学的帮助。

参考文献

[1] Ronald L. Graham; Donald E. Knuth; Oren Patashnik (1994). ”Chapter 7: Generating Functions”. Concrete Mathematics. A foundation for computer science (second ed.). Addison-Wesley. pp. 320 – 380.

[2] R. W. Robinson, Counting labeled acyclic digraphs, pp. 239-273 of F. Harary, editor, New Directions in the Theory of Graphs. Academic Press, NY, 1973.

[3] Huantian Cao, AutoGF: An Automated System to Calculate Coefficients of Generating Functions.

[4] 金策，《生成函数的运算与组合计数问题》，2015年信息学奥林匹克中国国家队候选队员论文

[5] 彭雨翔，《Introduction to Polynomials》，2016年全国青少年信息学奥林匹克冬令营讲课课件

非常规大小分块算法初探

北京师范大学附属实验中学 徐明宽

摘 要

分块算法是一种实用性高且简单易实现的算法，它不仅可以解决许多线性数据结构的操作问题，还能优化一些其他算法。本文介绍了确定最优分块大小的方法，通过一些例题介绍了非常规大小分块算法的直接应用，并介绍了几种通过非常规大小分块算法优化其它算法的应用。

1 引言

分块算法是一种重要的算法，常用于线性数据结构的操作，具有实用性高、易于实现等优点。虽然线段树等数据结构可以在 $O(\log n)$ 的时间复杂度内处理一些比较简单的情况，但是遇到比较复杂的情况可能就需要挖掘维护的信息的深层性质，这样可能会导致更高的时间复杂度，甚至可能会无法解决问题。这时分块算法的优势就体现出来了：不需要对问题做复杂的分析，也不需要复杂的实现。所以分块算法的实用性很高。

分块算法不仅可以直接解决许多线性数据结构的操作问题，还能优化一些其它算法。选择适当的非常规分块大小，我们可以将 LCA（最近公共祖先）、RMQ（区间最小值查询）、LA（第 k 祖先）等问题优化到理论最优的时间复杂度。

在 2013 年的集训队论文中，罗剑桥[1]曾研究了分块思想在一类数据处理问题中的应用，王子昱[2]也曾研究了分块的一些应用；在 2015 年的集训队论文中，邹逍遥[3]曾研究了分块在一类在线问题中的应用。

本文基于[1, 2, 3]的研究成果，进一步研究非常规大小的分块算法。

本文的组织结构如下：

在第 2 节中，简单介绍了分块算法。

在第 3 节中，介绍了一个确定最优分块大小的简洁方法，并介绍了非常规大小分块算法在一些例题中的直接应用。

在第 4 节中，介绍了一个分块优化空间复杂度的另类应用，和一种通过分块优化一些算法的时间复杂度的技巧——Method of Four Russians。

本文多次涉及程序的运行时间，运行环境为 64 位 Ubuntu 16.04 LTS 操作系统，Intel Core i3-3240 CPU @ 3.40GHz，16GB 内存，GCC/G++ 版本为 5.4.0，并开启 -O2 优化。所

有运行时间均为运行多次后取中位数，以尽量减小误差。

2 分块算法简介

分块算法是一个在信息学中很常见的算法，常用来处理序列的区间操作等问题。

对一个长度为 n 的序列暴力地进行区间操作是 $O(n)$ 的。我们可以对序列进行分块，这样进行区间操作时，可以对区间中整块的部分一起操作，对不在整块中的部分暴力操作。

例如，我们要进行“区间加一个数”和“单点查值”这两种操作，就可以对序列进行分块，对每个块维护一个加法标记（就是记录这个块中元素一起加了多少），这样每次修改时就能对每个整块直接 $O(1)$ 修改标记，对不在整块中的部分暴力修改元素的值；每次询问时返回元素的值加上其所在块的加法标记即可。

设分块大小为 S，则一共分了 $\left\lceil \frac{n}{S} \right\rceil$ 块。如果对序列中的一个元素操作与对一整块操作的时间复杂度均为 $O(1)$，那么一次区间操作的时间复杂度为 $O\left(\frac{n}{S}+S\right)$，在 $S=\sqrt{n}$ 时取到最小值 $O\left(\sqrt{n}\right)$。

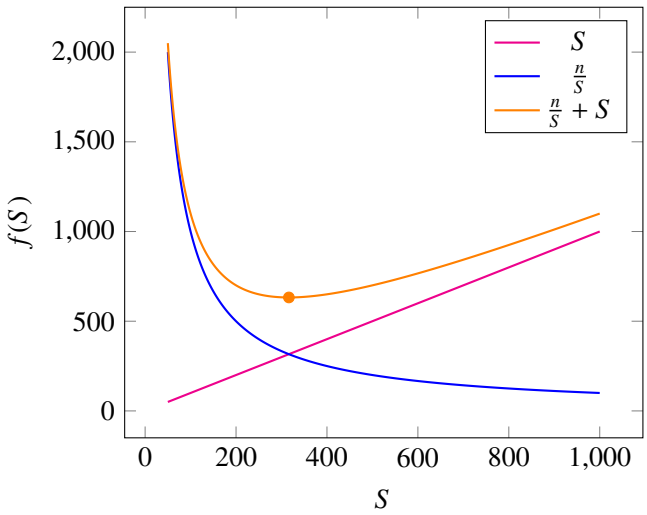


图 4: $\frac{n}{S}+S$ 与 S 取值的关系，$n=10^5$

为什么 $O\left(\frac{n}{S}+S\right)$ 在 $S=\sqrt{n}$ 时取到最小值呢？我们可以用均值不等式证明。

算术—几何均值不等式：对于任意 n（$n\geq 2$）个非负实数 $x_1,x_2,\ldots,x_n$，有：

$$\frac{\sum_{i=1}^{n} x_i}{n} \geq \sqrt[n]{\prod_{i=1}^{n} x_i} \tag{2.1}$$

当且仅当 $x_1=x_2=\cdots=x_n$ 时等号成立。

令 $n=2$，$x_1=\frac{n}{S}$，$x_2=S$，我们可以得出 $\frac{n}{S}+S\geq 2\sqrt{n}$，当且仅当 $\frac{n}{S}=S$ 即 $S=\sqrt{n}$ 时等号成立。

3 分块大小的确定

对于许多分块问题，分块大小为 $\sqrt{n}$ 都是最优的；但也有不少情况下，分块大小为 $\sqrt{n}$ 不是最优的。让我们先来看一道题。

例 1 分块入门 2 by hzwer[44]

给出一个长为 n 的数列，有 n 个操作，操作有两种：区间加一个数，或者询问区间内小于某个值 x 的元素个数。

我们将数列分成 $\sqrt{n}$ 块，每块大小 $\sqrt{n}$。先来思考只有询问操作的情况：不完整的块枚举统计即可；为了在每个整块内寻找小于一个值的元素数，我们可以对每块做一遍排序预处理（总复杂度 $O(n\log n)$），这样就能使用二分法对块内查询。每次查询在不超过 $\sqrt{n}$ 个块内二分，以及暴力不超过 $2\sqrt{n}$ 个元素，总复杂度 $O\left(n\log n+n\sqrt{n}\log n\right)=O\left(n\sqrt{n}\log n\right)$。

那么区间加怎么办呢？我们给每个块设置一个加法标记，每次操作对每个整块直接 $O(1)$ 标记，而不完整的块由于元素比较少，暴力修改元素的值。注意不完整的块修改后可能会使得该块内元素乱序，所以头尾两个不完整块需要重新排序。

在加法标记下的询问操作，块外还是暴力，查询小于（x - 加法标记）的元素个数，块内用（x - 加法标记）作为二分的值即可。

上文设分块大小为 $\sqrt{n}$，这的确易于思考，也会自然地得出 $O\left(n\sqrt{n}\log n\right)$ 的复杂度，但这样思考不易优化。我们不如设分块大小为 S，重新分析一遍时间复杂度。

询问操作，每次在不超过 $\left\lceil\frac{n}{S}\right\rceil$ 个块内二分，以及暴力不超过 $2S$ 个元素，复杂度 $O\left(\frac{n\log S}{S}+S\right)$。

修改操作，每次对不超过 $\left\lceil\frac{n}{S}\right\rceil$ 个块打标记，对两个块排序，复杂度 $O\left(\frac{n}{S}+S\log S\right)$。

预处理只需 $O(n\log n)$ 的时间。

于是总时间复杂度为 $O\left(n\log S\left(\frac{n}{S}+S\right)\right)$，这个式子显然取 $S=\sqrt{n}$ 最优，这样总时间复杂度仍为 $O\left(n\sqrt{n}\log n\right)$，并没有优化。问题出在哪了呢？

实际上，修改操作对不完整块部分每次并不需要 $O(S\log S)$ 的时间排序：本来数组是有序的，现在把其中一部分加上了一个数，那么可以将原数组分为修改后的和没修改的两部分，每部分都是一个有序数组，可以在 $O(S)$ 的时间内归并排序。于是总时间复杂度为 $O\left(n\left(\frac{n\log S}{S}+S\right)\right)$，取 $S=\sqrt{n\log n}$ 即可做到 $O\left(n\sqrt{n\log n}\right)$ 的复杂度。

3.1 分块大小的复杂度

在上题中，最后要让 $O\left(\frac{n\log S}{S}+S\right)=O\left(\frac{n\log n}{S}+S\right)$ 取到最小值。同样使用均值不等式，可以得出 $\frac{n\log n}{S}+S\geq 2\sqrt{n\log n}$，当且仅当 $S=\sqrt{n\log n}$ 时等号成立。

那么，是不是对于所有分块问题，我们都可以设分块大小为 S，最后使用均值不等式推出最优的分块大小与最低的时间复杂度呢？

[44]试题与解法来源：黄哲威[4]

有些分块算法（如带修改的莫队算法）的每次操作的时间复杂度为 $O\left(\frac{n^2}{S^2}+S\right)$。如果直接使用均值不等式，我们只能得出 $\frac{n^2}{S^2}+S\geq 2\frac{n}{\sqrt{S}}$，并没有消掉 S 这个变量。但是，我们可以将 S 拆成 $\frac{S}{2}+\frac{S}{2}$ 再使用均值不等式，可以得出 $\frac{n^2}{S^2}+\frac{S}{2}+\frac{S}{2}\geq 3\sqrt[3]{\frac{n^2}{4}}=O\left(n^{2/3}\right)$，当且仅当 $S=\sqrt[3]{2n^2}$ 时等号成立。图 5 给出了 $\frac{n^2}{S^2}+S$ 与 S 取值的关系。

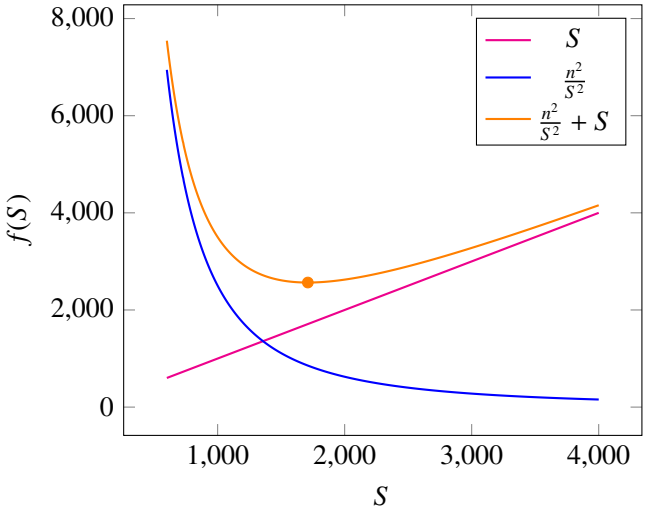


图 5: $\frac{n^2}{S^2}+S$ 与 S 取值的关系，$n=50000$

这种情况下虽然仍然可以使用均值不等式推出最优的分块大小与最低的时间复杂度，但是推导难度大大增加了——我们需要思考如何使用均值不等式才能把 S 消掉。

由于（大部分情况下）分块算法时间复杂度的函数取到最小值时（关于分块大小 S 的）导数为 0，我们也可以通过解导数为 0 的方程来得出最优的分块大小与最低的时间复杂度。例如，对 $\frac{n^2}{S^2}+S$ 求导得到 $-2\frac{n^2}{S^3}+1$，解方程 $-2\frac{n^2}{S^3}+1=0$ 可得 $S=\sqrt[3]{2n^2}$。但是求导也不是一件很简单的事情。

下面给出一种确定最优分块大小的简洁方法：在大部分情况下，我们可以推出时间复杂度为 $O(A+B)$ 的形式（其中 A 随 S 的增大而减小，B 随 S 的增大而增大，例如 $A=\frac{n\log n}{S}$，$B=S$），那么，**解方程** $A=B$ 即可得到最优（复杂度）的分块大小 S（而且方程 $A=B$ 成立时等号任意一侧的值就是最优的时间复杂度）。

看上面的例子：解方程 $\frac{n^2}{S^2}=S$ 得到 $S=n^{2/3}$，最优的时间复杂度就是 $S=n^{2/3}$。这比上面几种方法简洁不少。

上述方法的证明：因为 $A+B\leq 2\max(A,B)$ 且 $\max(A,B)\leq(A+B)$，所以 $O(A+B)=O(\max(A,B))$ 且 $O(\max(A,B))=O(A+B)$，即 $O(A+B)$ 等价于 $O(\max(A,B))$，所以我们只需让 $\max(A,B)$ 取到最小值。又因 A 随 S 的增大而减小，B 随 S 的增大而增大，所以，设 $S=x$ 时 $A=B=y$，则 $S<x$ 时 $A>y$，$S>x$ 时 $B>y$，故 $A=B$ 时 $\max(A,B)$ 取到最小值。

如果问题比较复杂，最后只能推出时间复杂度为 $O(A+B+C)$ 的形式（其中 A 随 S 的增大而减小，B 随 S 的增大而增大，C 随 S 的增大而增大；或：A 随 S 的增大而增大，B 随 S 的增大而减小，C 随 S 的增大而减小），那么同理，可以分别解方程 $A=B$、$A=C$，再带入到原式比较两种分块大小哪个更优即可。

这里再举几个例子：

- $O\left(\frac{nm\log n}{S}+nS\right)$ 应取 $S=\sqrt{m\log n}$，时间复杂度为 $O\left(n\sqrt{m\log n}\right)$
- $O\left(\frac{n^3}{S^2w}+mS\right)$ 应取 $S=\frac{n}{\sqrt[3]{mw}}$，时间复杂度为 $O\left(\frac{nm^{2/3}}{w^{1/3}}\right)$
- $O\left(\frac{n^2}{S^2}+\frac{n}{S}+S\right)$ 应取 $S=n^{2/3}$，时间复杂度为 $O\left(n^{2/3}\right)$

3.2 分块大小的常数

确定了分块大小的复杂度，我们再来确定常数。实际上，对于大多数情况，不考虑常数（即，分块大小的复杂度算出来 $O\left(\sqrt{n}\right)$ 就取 $\sqrt{n}$，算出来 $O\left(\sqrt{n\log n}\right)$ 就取 $\sqrt{n\log n}$）就已经很接近最优分块大小了——这是因为运行时间（运算次数）关于分块大小的函数在最优值附近都比较“平坦”（即导数的绝对值较小），所以分块大小的常数略微差一点并没有什么影响。

但是也有例外情况。例如上述《分块入门 2 by hzwer》一题，最后推出分块大小的复杂度为 $O\left(\sqrt{n\log n}\right)$，但实际最优分块大小的常数很小，以至于只看实际分块大小的话容易让人误以为分块大小的复杂度为 $O\left(\sqrt{n}\right)$。这是因为：

- 这道题每次操作的时间复杂度为 $O\left(\frac{n\log S}{S}+S\right)$，我们在分析时将 $\log S$ 变成了 $\log n$，由于 $S\approx\sqrt{n}$，所以 $\log S\approx\frac{1}{2}\log n$，这带来了 $\sqrt{\frac{1}{2}}$ 的常数。
- 时间复杂度中 $O\left(\frac{n\log S}{S}\right)$ 部分为效率较高的二分，且只有询问操作是这个复杂度，与之相比修改操作的 $\frac{n}{S}$ 可以忽略；而 $O(S)$ 部分为效率较低的归并排序。为了平衡两部分的运行效率，应将分块大小 S 调小，如图 6。

下面再来介绍一个许多分块问题都适用的常数优化：

区间操作时，暴力处理不在一整块中的部分时，如果这部分的元素较多，超过了块长的一半，那么我们可以改为对这一整块进行区间操作，并对这一整块中不在需要操作的区间中的部分进行“逆操作”（即，区间加法改为区间减法，区间求和改为区间求和的相反数等；区间赋值等不具有“可减性”的操作不能采用此优化）。

例如，我们将序列 [1..100] 分成了 10 块 $[1..10],[11..20],\ldots,[91..100]$，现在要对区间 [2..56] 加一个数 x，我们本来需要对块 $[11..20],\ldots,[41..50]$ 打上 $+x$ 的加法标记，并对第 $2,3,4,5,6,7,8,9,10,51,52,53,54,55,56$ 个元素进行 $+x$ 操作；但采取这个优化以后，我们

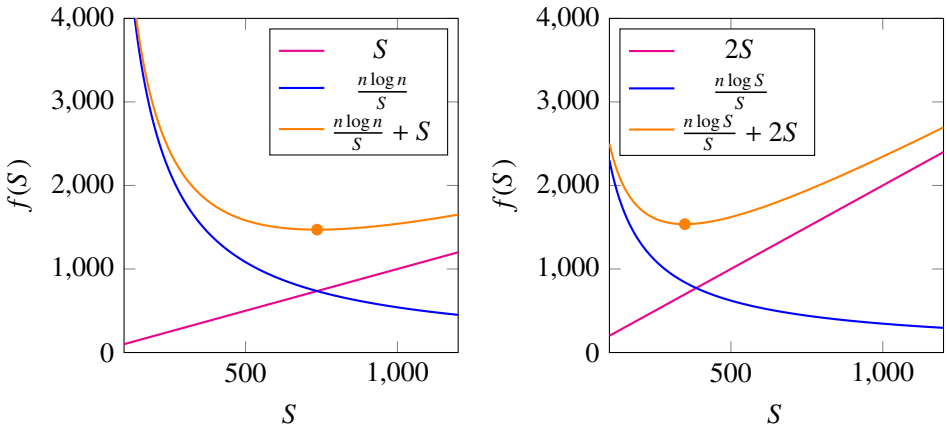


图 6: $\frac{n\log n}{S}+S$ 和 $\frac{n\log S}{S}+2S$ 的最优的 S 取值对比，$n=50000$

可以对块 $[1..10],\ldots,[51..60]$ 打上 $+x$ 的加法标记，并对第 $1,57,58,59,60$ 个元素进行 $-x$ 操作。

加上这个优化以后，暴力部分的常数减半，所以分块大小应该适当调大，以平衡两部分的运行效率。

例 2 挑战（子任务 2）[45]

给定两个长度为 n 的三进制串，现在有 q 个询问，问两个串各取出一段子串后有多少个对应位相同（原题里问的是多少个对应位模 3 意义下相差 1，可以转化为相同）。

数据范围：$n=q=300000$。时间限制：3s。空间限制：2GB。

先来看看标准算法是怎么做的。我们用二进制串来表示输入的三进制串：

- 0 → 100
- 1 → 010
- 2 → 001

于是只要统计两个子串的**按位与**的结果中，1 的个数。

直接使用 `std::bitset` 的话每次运算的时间均为 $\Theta\left(\frac{n}{w}\right)$，要跑几十秒；而手写一个 `bitset` 以后我们可以只计算需要求的部分，用分段查表法统计 1 的个数，虽然时间复杂度仍为 $O\left(\frac{nq}{w}\right)$，但是快了很多，运行时间为 3.898s。加上了读入优化与循环展开等优化以后运行时间降低到了 3.689s。

王逸松讲题时[5]提到了一个 $O\left((n\log n)^{1.5}\right)$ 的分块 FFT 传统做法，但是并不能跑过。现在我们来优化一下这个算法。

[45] 试题来源：WC 2017

回顾一下经转化后的问题：给定两个长度相同且不超过 $9*10^5$ 的二进制串，现在有不超过 $3*10^5$ 次询问，每次询问第一个串的一个子串和第二个串的一个子串的**按位与**的结果中 1 的个数。

注意到两个 01 串的**按位与**的结果中 1 的个数就是它们的点积，将其中一个串反转（`std::reverse`）以后就是它们的卷积中的一项。

我们现在将这两个串各分成 num 块，每块大小为 S （$num = \left\lceil \frac{3n}{S} \right\rceil$），对于第一个串的每一块和第二个串的每一块，将第二个串反转，做一遍卷积（可以使用 FFT 在每次 $O(S \log S)$ 的时间内实现。具体实现时，我们可以把两个串的 $2num$ 块的 FFT 结果存下来，可以省去 $2num^2 - 2num$ 次 FFT （仍需要做 num^2 次 IFFT）。），得到了一个长为 $2S$ 的数组（具体实现时，用 `short` 而不是 `int` 存储可以节约空间，而且能略微优化一点时间常数）。可以发现，这个数组的第 $i(1 \le i \le S)$ 个元素就是第一个串的长为 i 的前缀与第二个串（反转之前）的长为 i 的后缀的点积；这个数组的第 $(2S - i(1 \le i \le S))$ 个元素就是第一个串的长为 i 的后缀与第二个串（反转之前）的长为 i 的前缀的点积。于是我们可以在 $O\left(num^2 S \log S\right) = O\left(\frac{n^2 \log S}{S}\right)$ 的时间复杂度、 $O\left(num^2 S\right) = O\left(\frac{n^2}{S}\right)$ 的空间复杂度内，预处理出第一个串的任意一块的任意一个前缀/后缀与第二个串的任意一块的任意一个后缀/前缀的点积。

每次查询时，考虑查询的第一个串的子串是如何被分块的。对于最前面和最后面的不属于一整块的部分，我们可以直接暴力计算；对于每一整块，如果它对应着第二个串的一整块，那么直接取预处理的结果（可视为第一个串的这一块的长度为 S 的前缀与第二个串的这一块的长度为 S 的后缀）即可；否则，它一定对应着第二个串的某一块的一个后缀和它后面一块的一个前缀，设其中后缀的长度为 x，取第一个串的那一整块的长度为 x 的前缀与第二个串的这一块的这个长度为 x 的后缀的点积，以及第一个串的那一整块的长度为 $S - x$ 的后缀与第二个串的后面一块的长度为 $S - x$ 的前缀的点积，加起来即可。于是我们可以在 $O(num + S) = O\left(\frac{n}{S} + S\right)$ 的时间内回答一次询问。

所以这个做法的总时间复杂度为 $O\left(\frac{n^2 \log S}{S} + q\left(\frac{n}{S} + S\right)\right)$，取 $S = \sqrt{\frac{n}{\log n}}$ 即可得到 $O\left((n \log n)^{1.5} + q\sqrt{n \log n}\right)$ 的时间复杂度。但是，取 $S = \sqrt{n \log n}$ 即可得到 $O\left((n+q)\sqrt{n \log n}\right)$ 的更为优秀的时间复杂度。

我们通过确定分块大小把时间复杂度除掉了一个 log，实际效果如表 1 所示。

虽然这个时间复杂度优化的效果很显著，但是 14.555s / 14.599s（分别表示取 $S = 4096\left(\sqrt{n \log n}\right)$ 和 $S = 8192\left(2\sqrt{n \log n}\right)$ 的运行时间，下同）的运行时间还是远不如标准算法……我们需要一些常数优化。

1. 减少 FFT/IFFT 次数。

   由于输入均为实数（01 串），使用毛啸在 2016 年国家集训队论文中提到的技巧[6]可以将每两次 FFT/IFFT 合并为一次。

   加上这个优化以后运行时间降低到了 11.362s / 12.866s。

[46] 其实 $\le 1024\left(\sqrt{n}\right)$ 的分块大小均会导致超过 2GB 的空间限制

分块大小[46]	运行时间
$256\left(\sqrt{\frac{n}{\log n}}\right)$	130.347s
$512\left(2\sqrt{\frac{n}{\log n}}\right)$	73.173s
$1024\left(\sqrt{n}\right)$	41.272s
$2048\left(2\sqrt{n}\right)$	21.539s
$4096\left(\sqrt{n\log n}\right)$	14.555s
$8192\left(2\sqrt{n\log n}\right)$	14.599s
16384	21.271s

表 1: 运行时间与分块大小的关系

2. 读入优化。

   可以使用 `fread` 来进行读入优化。

   加上这个优化以后运行时间降低到了 11.110s / 12.151s。

3. 减少暴力次数。

   询问时，当最前面和最后面的不属于一整块的部分长度超过一整块的一半时，可以把暴力统计这部分的结果换成减去暴力统计不在这块内的部分的结果、并多统计一整块的结果，这样可以将这部分的常数除以 2。

   加上这个优化以后运行时间降低到了 8.783s / 6.912s。

4. 减小 FFT/IFFT 规模。

   注意到卷积结果的最大值只有 $\frac{S}{3}$，这对于 `double` 类型的精度是一种浪费。我们可以将每相邻两项用一个 `double` 存储，用一个较大的数 x 区分——这里 x 需要超过卷积结果的最大值，不妨取 $x = \frac{S}{2}$ 方便进行位运算（S 为 2 的幂）。

   举个例子，比如说我们要算 $\{a_0, a_1, a_2, a_3\}$ 与 $\{b_0, b_1, b_2, b_3\}$ 的卷积，得到结果是 $\{a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0, a_1b_3 + a_2b_2 + a_3b_1, a_2b_3 + a_3b_2, a_3b_3\}$。现在我们改为算 $\{a_0+a_1x, a_2+a_3x\}$ 与 $\{b_0+b_1x, b_2+b_3x\}$ 的卷积，得到结果将会是 $\{a_0b_0 + (a_0b_1 + a_1b_0)\,x + a_1b_1x^2, (a_0b_2 + a_2b_0) + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)\,x + (a_1b_3 + a_3b_1)\,x^2, a_2b_2 + (a_2b_3 + a_3b_2)\,x + a_3b_3x^2\}$，将所有 x^2 项进位到下一个式子的常数项即可得到原卷积的结果。

   这样做以后卷积结果的最大值将会变成 $\frac{S}{6} \cdot (x+1)^2 = \frac{S^3+4S^2+4S}{24}$，在 $S = 131072$ 时也只有大约 $9.38 * 10^{13}$，在 `double` 类型可承受的范围内。具体实现时需要注意由 `double` 类型转回整数类型时需要使用 `long long` 等类型。

   加上这个优化以后运行时间降低到了 5.854s / 5.260s。

另外也有一种优化方法是放弃这个优化，并把 `double` 换成 `float`，但这样运行时间只会降低到 8.077s / 6.524s，效果远不如上述优化。

5. 回归原题。

   我们在把问题从三进制串转化为二进制串的同时，将数据规模扩大到了原来的 3 倍，这使这个算法的暴力部分跑得很慢。

   我们不如不转化问题，直接对原问题的三进制串做三遍 FFT 预处理——对于 “两个串各取出一段子串后有多少个对应位相同” 中对应位为 0、1、2 分别 FFT 预处理，每次预处理时把目标字符（0 或 1 或 2）视为 1，其它字符视为 0 即可，这样我们还可以把 3 次预处理的结果加起来，节省内存。

   查询时，整块的做法与之前相同，暴力部分可以换成朴素的模拟。

   总时间复杂度不变，暴力部分时间常数除以 3，总空间常数除以 3。

   加上这个优化以后运行时间降低到了 1.646s / 1.302s。

现在我们做到了 1.302s，已经远快于上述标准算法（3.689s）了。结合上述标准算法的优点，我们还能再做一个复杂度优化。再看我们这个算法中的暴力部分，和原问题相比除了询问数据规模较小以外没有区别。那么，为什么不使用 `bitset` 呢？（这个优化和上述常数优化的最后一个不能同时加）

使用 `bitset` 以后，一次询问的时间复杂度降低到了 $O\left(num+\frac{S}{w}\right)=O\left(\frac{n}{S}+\frac{S}{w}\right)$，总时间复杂度为 $O\left(\frac{n^2\log S}{S}+q\left(\frac{n}{S}+\frac{S}{w}\right)\right)$，取 $S=\sqrt{nw\log n}$ 即可得到 $O\left((n+q)\sqrt{\frac{n\log n}{w}}\right)$ 的时间复杂度。加上这个优化以后运行时间降低到了 0.979s / 0.774s（分别表示取 $S=32768\left(\sqrt{nw\log n}\right)$ 和 $S=65536\left(2\sqrt{nw\log n}\right)$ 的运行时间，下同），而且就算去掉前面所有常数优化也能跑到 2.488s / 2.269s。

此外，我们还可以对【使用手写的 `bitset` 对两个子串进行**按位与**、查表统计 1 的个数并求和】这一步进行循环展开，最终将运行时间降低到 0.964s / 0.749s。

4 非常规大小分块算法的应用

4.1 埃氏筛法的空间复杂度优化

埃氏筛法[7]（Sieve of Eratosthenes）是一个简单的求质数方法：将所有不超过 $\sqrt{n}$ 的质数的倍数剔除，即可得到不超过 n 的所有质数。

它的时间复杂度为 $O\left(n\sum\limits_{p\le\sqrt{n},p\text{为质数}}\frac{1}{p}\right)=O\left(n\log\log n\right)$（根据 Mertens 第二定理[8]），空间复杂度为 $O(n)$ **位**。

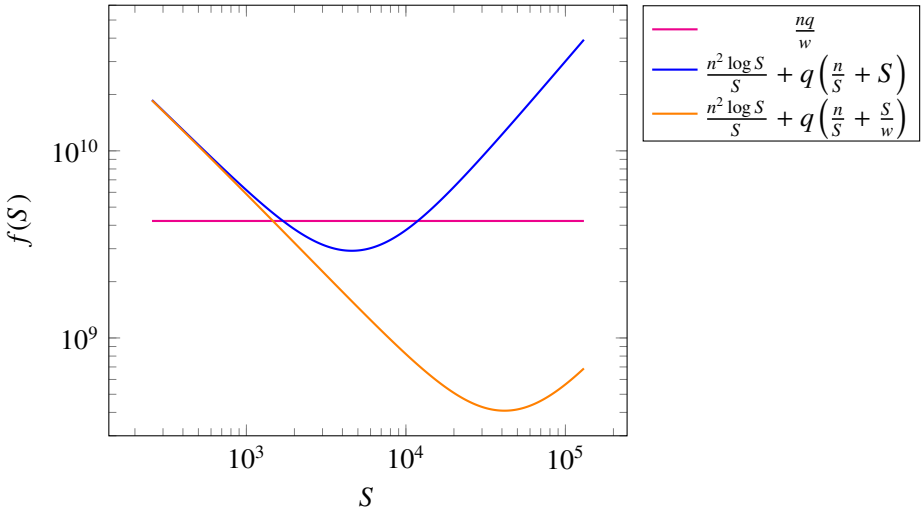


图 7: $\frac{n^2 \log S}{S} + q\left(\frac{n}{S} + \frac{S}{w}\right)$ 与 S 取值的关系，$n = 9 * 10^5$，$q = 3 * 10^5$，$w = 64$

我们也可以在筛质数时对 1 到 n 的每个数分解质因数，时间复杂度仍为 $O\left(n \sum\limits_{p\le\sqrt{n},p\text{为质数}} \sum\limits_{i=1}^{\infty} \frac{1}{p^i}\right) = O\left(n \sum\limits_{p\le n,p\text{为质数}} \frac{1}{p-1}\right) = O\left(n \sum\limits_{p\le n,p\text{为质数}} \frac{2}{p}\right) = O(n \log\log n)$，空间复杂度为 $O(n)$。

这个算法是可以分块的：对于任意区间 $(L,R]$，如果已经预处理了所有不超过 $\sqrt{R}$ 的质数，我们可以在 $\frac{\sqrt{R}}{\log R} + (R-L) \sum\limits_{p\le\sqrt{R},p\text{为质数}} \sum\limits_{i=1}^{\infty} \frac{1}{p^i}$ 的时间内将区间 $(L,R]$ 的每个数分解质因数。于是，把区间 $(0,n]$ 分成每块大小为 S 的 $\left\lceil \frac{n}{S} \right\rceil$ 块以后，时间复杂度为

$$\begin{aligned} &\sum_{i=1}^{\left\lceil \frac{n}{S} \right\rceil} \left(\frac{\sqrt{i \cdot S}}{\log(i \cdot S)} + S \sum_{p\le\sqrt{i\cdot S},p\text{为质数}} \sum_{i=1}^{\infty} \frac{1}{p^i} \right) \\ &= O\left(\sum_{i=1}^{\left\lceil \frac{n}{S} \right\rceil} \left(\sqrt{i} \cdot \frac{\sqrt{S}}{\log(i \cdot S)} + S \log\log(i \cdot S) \right) \right) \\ &= O\left(\frac{n^{1.5}}{S \log n} + n \log\log n \right), \end{aligned} \tag{4.1}$$

空间复杂度为 $O\left(\frac{\sqrt{n}}{\log n} + S\right)$（其中 $\frac{\sqrt{n}}{\log n}$ 是因为要存储不超过 $\sqrt{n}$ 的质数；预处理时可以每个数花 $O\left(n^{1/4}\right)$ 的时间、$O(1)$ 的空间暴力检验是不是质数，故并不需要 $\sqrt{n}$ 的空间）。

所以，取 $S = \frac{\sqrt{n}}{\log n}$ 即可在保持 $O(n \log\log n)$ 的时间复杂度的同时做到 $O\left(\frac{\sqrt{n}}{\log n}\right)$ 的空间复杂度。在空间允许的情况下，可以花 $O\left(\sqrt{n}\right)$ 的空间预处理质数，同时取 $S = \sqrt{n}$，这样时间常数更小。分块埃氏筛法不仅节约了空间，而且避免了开巨大的数组，可以把整个数组放到一级或二级缓存中，所以实际运行效率更高（具体见下例题）。

例 3 小 Z 的公式[47]

记 $\mathrm{S}(n)=\sum_{i=1}^{n} i\cdot\sigma_1\left(i^2\right)$，其中 $\sigma_1(x)$ 表示 x 的约数和。

输入一个正整数 n，输出 $\mathrm{S}(n) \bmod \left(10^9+7\right)$。

数据范围：$n\le 10^9$。空间限制：1GB。

这道题有 $O\left(\frac{n^{3/4}}{\log n}\right)$ 的做法，但推导难度和代码实现难度都很高，且与本文关系不大，故不再赘述。

下面介绍一种分段打表做法：注意到对于每个数 x，只要将 x 分解质因数，就能很方便地求出 $x\cdot\sigma_1\left(x^2\right) \bmod \left(10^9+7\right)$：具体地说，设 $x=\prod_{i=1}^{k} {p_i}^{\alpha_i}$（$p_1,p_2,\ldots,p_k$ 为互不相同的质数，$\alpha_1,\alpha_2,\ldots,\alpha_k\in\mathbb{N}^*$），则 $x\cdot\sigma_1\left(x^2\right)=x\prod_{i=1}^{k}\sum_{j=0}^{2\alpha_i}{p_i}^j=x\prod_{i=1}^{k}\left({p_i}^{2\alpha_i+1}-1\right)\cdot(p_i-1)^{-1}$。所以可以确定一个段长 len（例如 10^7），应用上述分块埃氏筛法，在本地花 $O(n\log\log n)$ 的时间预处理出所有 $\mathrm{S}(i\cdot len) \bmod \left(10^9+7\right)$（$i\in\mathbb{N}$，$i\cdot len\le n$），即可在 $O(len\log\log n)$ 的时间以及 $O\left(\frac{n}{len}\cdot\log answer\right)$ 的代码长度内通过这道题。

分块大小	运行时间
$3052\left(\frac{\sqrt{n}}{\log\sqrt{n}}\right)$	36.882
16384	31.330
$31623\left(\sqrt{n}\right)$	29.840
65536	31.613
10^7	56.506
10^8	57.812
10^9(不分块)	59.674

表 2: 预处理时间与分块大小的关系（预处理规模：$n=10^9$，时间单位：秒）

由表 2 可知，分块对埃氏筛法时间的优化也是很显著的。欧拉筛法（即“线性筛”，时空复杂度均为 $O(n)$）跑 $n=10^9$ 的数据虽然只需要 19.167s，但是空间劣势很大（需要至少 6.5GB 内存），且无法用来分段打表。

在另一台装有 Intel Core i5-6500 CPU @ 3.20 GHz 和 Windows 10 操作系统的电脑上，不分块埃氏筛法跑了 67.35s，欧拉筛法跑了 21.85s，而分块埃氏筛法（分块大小同样为 $31623\left(\sqrt{n}\right)$）由于可以将整个数组放到一级缓存中，只跑了 24.81s，优化效果更加显著。

4.2 Method of Four Russians

Method of Four Russians[9]是一种加速一些算法的技巧。它的主要思想是分块，对块内部暴力预处理，整块之间应用原算法，减小原算法的数据规模，从而优化时间复杂度。与

[47]题目来源：北京师范大学附属实验中学 王靖涵

之前的分块不同，由于 Method of Four Russians 对块内部的预处理很耗时，分块大小一般为 $O(\log n)$ 级别。

毛啸在 2016 年国家集训队论文中介绍了通过 Method of Four Russians 将快速幂中的乘法次数（即加法链的长度）由 $2\log_2 n + C$（C 为常数）降低到 $\log_2 n + O\left(\frac{\log n}{\log\log n}\right)$ 的技巧[6]。如果只考虑非倍增部分的乘法的话，这就是一个将时间复杂度除掉 $\log\log n$ 的例子，但是由于快速幂是 $\Omega(\log n)$ 的，这个技巧就只能算是常数优化了。

下面来介绍 Method of Four Russians 的几个优化时间复杂度的应用。

4.2.1 01 矩阵乘法的优化

本小节中研究的 01 矩阵乘法的定义是：将普通矩阵乘法的所有操作在模 2 意义下进行。

设 A 为 $m \times n$ 的 01 矩阵，B 是 $n \times p$ 的 01 矩阵，那么定义 A 与 B 的乘积 C 为 $m \times p$ 的 01 矩阵，其中（记 c_{ij} 为 C 矩阵第 i 行第 j 列的元素，a_{ij} 和 b_{ij} 类似）

$$c_{ij} = \left(\sum_{k=1}^{n} a_{ik} b_{kj}\right) \bmod 2 \tag{4.2}$$

对于其他定义下的 01 矩阵乘法（例如保证输入是 01 矩阵的普通矩阵乘法等），本小节中也有一部分优化方法能适用。

一维分块

我们枚举每个 c_{ij}，考虑如何快速计算 $\sum_{k=1}^{n} a_{ik} b_{kj}$。

我们可以对这个求和式子进行分块，设分块大小为 S，那么两个长度为 S 的 01 串一共有 4^S 种情况。我们可以对每种情况花 $O(S)$ 的时间求和，并在 $O(mn + np)$ 的时间内预处理出 A 矩阵每一行和 B 矩阵每一列的每个长度为 S 的块对应着哪种情况（长度为 S 的 01 串），这样就能在 $O\left(mn + np + mp + \frac{mnp}{S} + 4^S \cdot S\right)$ 的总时间复杂度计算了，取 $S = \frac{\log(mnp)}{3}$ 即可做到 $O\left(mn + np + mp + \frac{mnp}{\log(mnp)}\right)$，当 $m = n = p$ 时，时间复杂度为 $O\left(\frac{n^3}{\log n}\right)$。

使用 `bitset`

其实上面这个算法在信息学竞赛中并没有多少实际应用价值——因为 01 矩阵乘法可以使用 `bitset`，$\sum_{k=1}^{n} a_{ik} b_{kj}$ 可以直接由两个 `bitset` 进行**按位与**操作再统计 1 的个数得到。

这样时间复杂度为 $O\left(mn + np + mp + \frac{mnp}{w}\right)$，当 $m = n = p$ 时，时间复杂度为 $O\left(\frac{n^3}{w}\right)$。而在矩阵乘法可以接受的数据范围下，$w$ 远大于 $\log n$。

二维分块

我们可以将 A、B、C 矩阵分块成 $S\times S$ 的子矩阵。方便起见，不妨设 n,m,p 都是 S 的倍数（A 为 $m\times n$ 的矩阵，B 是 $n\times p$ 的矩阵，C 为 $m\times p$ 的矩阵），如果不是的话可以在 A、B、C 矩阵的右边或下面补 0 来将 n,m,p 补成 S 的倍数。

$$A=\begin{bmatrix} A_{11} & A_{12} & \dots & A_{1\frac{n}{S}} \\ A_{21} & A_{22} & \dots & A_{2\frac{n}{S}} \\ \vdots & \vdots & \ddots & \vdots \\ A_{\frac{m}{S}1} & A_{\frac{m}{S}2} & \dots & A_{\frac{m}{S}\frac{n}{S}} \end{bmatrix}, B=\begin{bmatrix} B_{11} & B_{12} & \dots & B_{1\frac{p}{S}} \\ B_{21} & B_{22} & \dots & B_{2\frac{p}{S}} \\ \vdots & \vdots & \ddots & \vdots \\ B_{\frac{n}{S}1} & B_{\frac{n}{S}2} & \dots & B_{\frac{n}{S}\frac{p}{S}} \end{bmatrix}, C=\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1\frac{p}{S}} \\ C_{21} & C_{22} & \dots & C_{2\frac{p}{S}} \\ \vdots & \vdots & \ddots & \vdots \\ C_{\frac{m}{S}1} & C_{\frac{m}{S}2} & \dots & C_{\frac{m}{S}\frac{p}{S}} \end{bmatrix}$$

根据分块矩阵的乘法，有

$$C_{ij}=\left(\sum_{k=1}^{\frac{n}{S}} A_{ik}B_{kj}\right) \bmod 2 \tag{4.3}$$

这里将一个矩阵 mod 2 的意思是将这个矩阵中的每一项 mod 2。

两个大小为 $S\times S$ 的矩阵一共有 4^{S^2} 种情况，我们可以对每种情况花 $O\left(S^3\right)$ 的时间预处理乘积，并在 $O(mn+np)$ 的时间内预处理出 A、B 矩阵的每一块对应着哪种情况（大小为 $S\times S$ 的 01 矩阵），这样就能在 $O(nS)$ 的时间复杂度内计算出 C 矩阵中一块的答案了。总时间复杂度为 $O\left(mn+np+mp+\frac{mnp}{S}+4^{S^2}\cdot S^3\right)$，取 $S=\sqrt{\frac{\log(mnp)}{3}}$ 即可做到 $O\left(mn+np+mp+\frac{mnp}{\sqrt{\log(mnp)}}\right)$，当 $m=n=p$ 时，时间复杂度为 $O\left(\frac{n^3}{\sqrt{\log n}}\right)$。

虽然这个时间复杂度还不如上文的一维分块，但是它可以结合 `bitset`：预处理两个大小为 $S\times S$ 的矩阵的乘积时可以使用 `bitset` 做到 $O\left(\frac{S^3}{w}\right)$，每次计算 C 矩阵中一块的答案也可以使用 `bitset` 做到 $O\left(\frac{nS}{w}\right)$。这样总时间复杂度为 $O\left(mn+np+mp+\frac{mnp}{Sw}+4^{S^2}\cdot\frac{S^3}{w}\right)$，取 $S=\sqrt{\frac{\log(mnp)}{3}}$ 即可做到 $O\left(mn+np+mp+\frac{mnp}{w\sqrt{\log(mnp)}}\right)$，当 $m=n=p$ 时，时间复杂度为 $O\left(\frac{n^3}{w\sqrt{\log n}}\right)$。

4.2.2 LCA 与 RMQ 的优化

LCA[11]（Lowest Common Ancestor），即最近公共祖先，是指这样一个问题：在一棵有根树中，找出某两个节点的最近的公共祖先。RMQ[12]（Range Minimum Query），即区间最小值查询，是指这样一个问题：在一个数列中，查询一个区间中所有数的最小值（也可以查询最小值的位置）。

这两个问题都很经典，有很多种算法。下面用 $O(f(n))-O(g(n))$ 描述一个算法的时间复杂度：当数据规模（LCA 中树的大小，或 RMQ 中数列的长度）为 n 时，这个算法要花

$O(f(n))$ 的时间预处理，然后可以做到每次查询 $O(g(n))$ 的时间复杂度。如果有 m 次查询，那么总时间复杂度为 $O(f(n)+m\cdot g(n))$。

LCA 有 $O(n)$–$O(n)$ 或 $O\left(n^2\right)$–$O(1)$ 的暴力，$O(n\log n)$–$O(\log n)$ 的倍增、$O(n)$–$O(\log n)$ 的轻重链剖分、$O(n)-O\left(\sqrt{n}\right)$ 的长链剖分、$O((n+m)\alpha(n))$ [48] 的 Tarjan （离线）等算法。

RMQ 有 $O(n)-O(n)$ 或 $O\left(n^3\right)-O(1)$ 的暴力、$O\left(n^2\right)-O(1)$ 的简单 DP、$O(n)-O\left(\sqrt{n}\right)$ 的分块、$O(n\log n)-O(1)$ 的 ST 表、$O(n)-O(\log n)$ 的线段树等算法，王子昱也在 2013 年的国家集训队论文中介绍了 $O(n\log\log n)-O(1)$ 和 $O(n\log^* n)-O(1)$ 的算法[2]。

本小节接下来会介绍这两个问题 $O(n)-O(1)$（这是最优的时间复杂度，因为输入就已经要 $\Omega(n)$ 的时间了）的 Method of Four Russians 算法。

将 LCA 转化为 ± 1 RMQ

首先，我们对整棵树进行欧拉遍历（即从根节点开始 DFS，每走过一条边（无论向上还是向下）就记录下当前节点，最后得到一条欧拉回路），得到一个节点序列与对应的深度序列，如图 8。

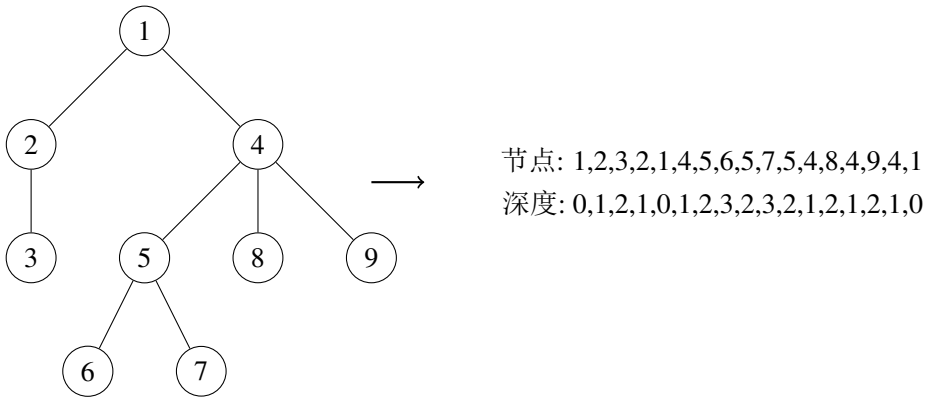


图 8: 对整棵树进行欧拉遍历，得到一个节点序列与对应的深度序列

可以发现这两个序列有一些性质：

- 因为一共有 $n-1$ 条边，所以序列的长度为 $2n-1$。
- 每个节点在节点序列中都出现了（它的儿子个数 + 1）次。
- 深度序列中相邻两个数的差一定为 ± 1。
- 任意两个节点的 LCA 就是欧拉遍历中这两个节点之间的深度最小的节点：记 $\text{place}(u)$ 表示节点 u 在节点序列中**任意**一次出现的位置，则对于任意两个节点 u 和 v（可以

[48] m 为查询次数，α 为反阿克曼函数

相同），设深度序列中区间 $[\text{place}(u), \text{place}(v)]$ （不妨设$\text{place}(u) \le \text{place}(v)$）的最小值位置为 w，则节点 u 和节点 v 的 LCA 就是节点序列中位置为 w 的节点。

所谓 ± 1 RMQ，就是数列中相邻两个数的差都为 ± 1 的 RMQ。于是，我们在 $O(n)$ 的时间内将 LCA 转化成了 ± 1 RMQ。

RMQ $O(n \log n) - O(1)$ 的 ST 算法

ST（Sparse Table）算法通过预处理所有长度为 2 的幂的区间的最小值，来做到每次 $O(1)$ 查询任意区间的最小值。

可使用以下 DP 在 $O(n \log n)$ 的时间内预处理。

定义：$f[i][j]$ 表示区间 $[j, j + 2^i)$ 的最小值。

初值：$f[0][j] = a[j]$（a 为初始数列）；$j > n$时 $f[i][j] = \infty$

转移：$f[i][j] = \min\left(f[i-1][j], f[i-1][j + 2^{i-1}]\right)$

查询：区间 $[l, r)$ 的最小值等于 $\min\left(f[\lfloor \log_2(r-l) \rfloor][l], f[\lfloor \log_2(r-l) \rfloor][r - 2^{\lfloor \log_2(r-l) \rfloor}]\right)$。（其中 $\lfloor \log_2(x) \rfloor$ 可以在 $O(n)$ 的时间内预处理）

如果要查询最小值的位置，将预处理 DP 与查询均稍作改动（预处理数组存储最小值的位置，取 min 时用对应位置的值来比较）即可。

Method of Four Russians

我们可以对 ± 1 RMQ 的数列进行分块，设分块大小为 S。

我们可以在 $O(n)$ 的时间内计算出每块的最小值（及其位置），对这 $\left\lceil \frac{n}{S} \right\rceil$ 个最小值（及其位置）组成的数列应用 ST 算法，预处理时间复杂度为 $O\left(\frac{n}{S} \log \frac{n}{S}\right)$，每次查询时对于整块的部分即可 $O(1)$ 查询。

那么对于查询区间中不属于一整块的部分怎么办呢？这时终于要用上“相邻两个数的差都为 ± 1”的性质了：如果将每个块的 S 个数中每相邻两个数做差得到 $S - 1$ 个数，那么一共只有不超过 2^{S-1} 种。我们可以对这所有 2^{S-1} 种 ± 1 序列取前缀和还原成 S 个数的数列（注意我们不知道第一数是几，可以先任取一个数，如 0），然后预处理出所有 $\frac{S(S+1)}{2}$ 个区间的最小值的位置（可以发现无论第一个数是几，这些区间最小值的位置都相同）。最后，我们预处理出所有 $\left\lceil \frac{n}{S} \right\rceil$ 块每块属于这 2^{S-1} 种的哪一种，即可在 $O(1)$ 的时间内查询任意一块内的任意一个区间的最小值的位置。

综上，我们做到了每次 $O(1)$ 查询，同时预处理的时间复杂度为 $O\left(\frac{n}{S} \log \frac{n}{S} + 2^S \cdot S^2\right)$，取 $S = \frac{\log n}{2}$ 即可做到 $O(n)$！

LCA 运行时间对比

表 3 和表 4 中，“$a + b = c$” 的形式表示预处理需要花 a 秒的时间，n 次询问一共需要花 b 秒的时间，输入 n 个点的树并回答 n 次询问一共需要花 c 秒的时间。这样把预处理和询问分开计时，即可得出：树的大小为 n、询问个数为 m 时，总时间应为 $a + \frac{m}{n} \cdot b$。

树以父亲数组的形式表示，记 $father[i]$ 为点 i 的父亲（以 0 号点为根，$i \in [1, n), father[i] \in [0, i)$）。对于这 4 种树的形态，“随机” 表示 $\forall i \in [1, n)$，$father[i]$ 在 $[0, i)$ 内均匀随机生成；“链” 表示整棵树为一条链（$father[i] = i - 1$）；“ 完全二叉树” 表示整棵树为一棵完全二叉树（$father[i] = \left\lfloor \frac{i-1}{2} \right\rfloor$）；“混合” 表示有 $\frac{n}{3}$ 的点形成一条链，$\frac{n}{3}$ 的点形成一棵完全二叉树，剩下约 $\frac{n}{3}$ 的点的父亲随机生成。另外，所有询问均随机生成。

Tarjan 离线算法中，可以写常数较小但平均每次操作复杂度为 $O(\log n)$ 的只路径压缩不按秩合并的并查集，也可以写平均每次操作复杂度为 $O(\alpha(n))$ 的路径压缩 + 按秩合并的并查集。表 3 和表 4 中 Tarjan 算法的第一行表示只路径压缩不按秩合并的并查集的运行时间，第二行表示路径压缩 + 按秩合并的并查集的运行时间。

表 3 和表 4 中 “ST 算法” 指的是将 LCA 转化为 RMQ 后直接使用 $O(n \log n) - O(1)$ 的 ST 算法。

树的形态	随机	链	完全二叉树	混合
Four Russians	0.221 + 0.354 = 0.575	0.105 + 0.331 = 0.436	0.056 + 0.339 = 0.395	0.172 + 0.372 = 0.544
ST 算法	0.444 + 0.127 = 0.571	0.190 + 0.161 = 0.351	0.177 + 0.130 = 0.307	0.290 + 0.140 = 0.430
轻重链剖分	0.091 + 0.632 = 0.723	0.014 + 0.062 = 0.076	0.008 + 0.945 = 0.953	0.045 + 0.606 = 0.651
倍增	0.043 + 0.388 = 0.431	0.042 + 0.865 = 0.907	0.009 + 0.390 = 0.399	0.081 + 0.947 = 1.028
Tarjan	0.595 0.754	0.707 0.450	0.424 0.450	0.617 0.556
暴力	0.010 + 0.546 = 0.556	—	0.002 + 0.477 = 0.479	—

表 3: LCA 运行时间对比，$n = 2 * 10^6$

由于近年来 OI 界捆绑测试（即若干个测试点构成一个子任务，只有这些测试点全部通过才能得到这个子任务的分数）逐渐盛行，我们比较关心算法在最坏情况下的运行效率。在这几个算法中，Method of Four Russians 算法在 $n = 10^7$ 的最坏情况下（上述 4 种情况中的最坏情况，下同）的确是最快的，但与 ST 算法相比优势很小，因为虽然每次查询都是 $O(1)$，但是 Method of Four Russians 算法是至多 4 个数取 min，而 ST 算法是至多 2 个数取

树的形态	随机	链	完全二叉树	混合
Four Russians	1.189 + 1.956 = 3.145	0.534 + 1.912 = 2.446	0.291 + 1.825 = 2.116	0.860 + 1.935 = 2.795
ST 算法	2.611 + 0.685 = 3.296	1.087 + 0.889 = 1.976	1.012 + 0.688 = 1.700	1.767 + 0.779 = 2.546
轻重链剖分	0.522 + 4.415 = 4.937	0.145 + 0.333 = 0.478	0.071 + 6.918 = 6.989	0.336 + 4.391 = 4.727
倍增	0.308 + 2.089 = 2.397	0.244 + 5.154 = 5.398	0.056 + 2.090 = 2.146	0.579 + 6.180 = 6.759
Tarjan	4.287 5.380	4.961 2.898	2.997 3.071	3.890 3.759
暴力	0.081 + 4.406 = 4.487	—	0.010 + 3.756 = 3.766	—

表 4: LCA 运行时间对比，$n = 10^7$

min。当询问次数相对于树的大小较小时，Method of Four Russians 算法便能体现出优势了：例如当树的大小为 10^7 而询问次数为 $5 * 10^6$ 时，Method of Four Russians 算法在最坏情况下（树的形态为“随机”）运行时间为 2.167s，而 ST 算法在最坏情况下（树的形态为“随机”）要跑 2.953s，轻重链剖分算法在最坏情况下（树的形态为“ 完全二叉树”）要跑 3.530s，倍增算法在最坏情况下（树的形态为“混合”）要跑 3.669s，Tarjan 离线算法在最坏情况下（树的形态为“随机”）也要跑 3.034s （只路径压缩不按秩合并的并查集）/ 4.101s（路径压缩 + 按秩合并的并查集）。

将 RMQ 转化为 LCA

RMQ 可以通过构造笛卡尔树转化为 LCA。笛卡尔树（Cartesian tree），是一种二叉树，树的每个节点有两个值，一个为 *key*，一个为 *value*，*key* 满足二叉查找树的性质，而 *value* 满足堆性质（本文中为小根堆）。本文中笛卡尔树每个节点的 *key* 值为这个节点在原数列中对应的位置，将不显式表示；*value* 值为这个节点在原数列中对应的数。

首先，我们可以找出数列中的最小值（我们可以假设数列中没有重复元素，如果有的话我们可以以下标为第二关键字比较，从而去重），把它设为根节点，然后这个数将原数列切成了两半，我们分别构造左右两个数列的笛卡尔树，把它们的根节点分别接到原数列最小值对应的根节点的左右儿子上，这样就构造出了整个数列的笛卡尔树。

如图 9，数列 8,7,2,8,6,9,4,5 对应的笛卡尔树的根是点 2，左子树是数列 8,7 对应的笛卡尔树，右子树是数列 8,6,9,4,5 对应的笛卡尔树。

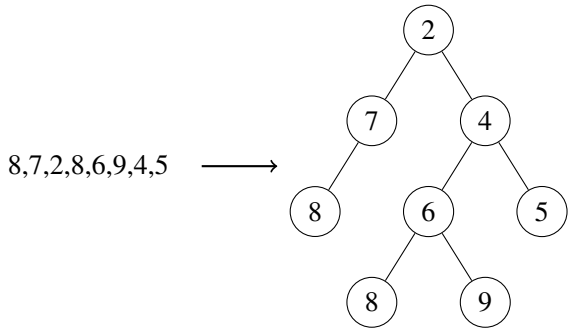


图 9: 例子：一个数列对应的笛卡尔树

易证对于任意区间 $[l,r]$，这段区间的最小值就是笛卡尔树上 l 和 r 对应节点的 LCA。

但是这样递归构造太慢了，能不能在线性时间内构造呢？答案是肯定的。

我们可以使用**单调栈**维护笛卡尔树的**右链**，从左向右依次加入节点，将栈顶比新加入的节点大的节点弹出，将新加入的节点接在当前栈顶节点（若不存在则将当前节点设为根）的右子树，将最后一个弹出的节点（若不存在则不操作）接在新加入的节点的左子树，再将新加入的节点推入栈即可。

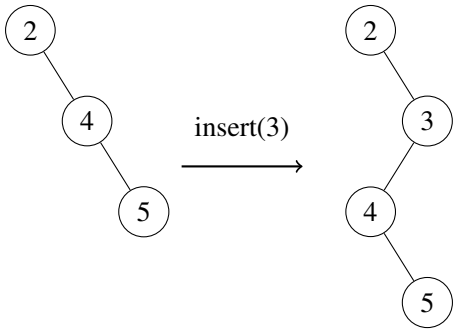


图 10: 例子：线性构造笛卡尔树算法中的一步

图 10 中，**单调栈**中本来存储的是 2,4,5，现在要加入节点 3，则将 4、5 弹出，将 3 接在 2 的右子树，将 4 接在 3 的左子树，再将 3 推入栈即可。

于是，我们在 $O(n)$ 的时间内将 RMQ 转化成了 LCA，再应用上述 Method of Four Russians 算法即可做到 $O(n) - O(1)$ 的 RMQ。

4.2.3 LA（第 k 祖先）的优化

LA[13]（Level Ancestor），即第 k 祖先，是指这样一个问题：在一棵有根树中，查询某个节点 v 的第 k 个祖先；或查询某个节点 v 的祖先中**深度**为 d 的节点。（记根节点的深度为

0，其它节点的深度是父亲节点的深度 +1；由于可以预处理每个节点的深度，所以这两个问题等价）

这个问题有 $O(n)-O(n)$ 或 $O\left(n^2\right)-O(1)$ 的暴力、$O(n\log n)-O(\log n)$ 的倍增、$O(n)-O(\log n)$ 的轻重链剖分、$O(n)-O\left(\sqrt{n}\right)$ 的长链剖分等算法。本小节接下来会介绍这个问题最优时间复杂度（$O(n)-O(1)$）的 Method of Four Russians 算法。

$O(n\log n)-O(\log n)$ 的倍增算法

花 $O(n\log n)$ 的时间预处理出每个节点 v 的第 1、2、4、8、……、2^i、……、2^l（其中 $l=\lfloor\log_2(\text{depth}(v))\rfloor$（depth 为深度））层祖先，即可在每次查询时（如果查询第 k 个祖先）将 k 二进制分解，依次向上跳即可。

例如要查询 v 第 11 个祖先，则从节点 v 开始依次向上跳 8、2、1 层即可。显然每次查询的时间复杂度为 $O(\log n)$。

$O(n)-O\left(\sqrt{n}\right)$ 的长链剖分算法

长链剖分，即每次选择最长的一条从根节点向下的链（如果有多条最长的可以任选一条），并把它删除，再对生成的每棵树进行长链剖分。

我们可以记录下剖分出的每条链的节点序列（按深度从小到大的顺序），并记录下每个节点在哪条链的节点序列中的第几个，记节点 v 在它所在链的节点序列的第 $\text{place}(v)$ 个。

在每次查询节点 v 的第 k 个祖先时，如果 $\text{place}(v)>k$，那么取节点 v 所在链的节点序列的第 $(\text{place}(v)-k)$ 个节点就是答案；否则，我们可以查询节点 v 所在链顶部节点（即节点序列的第 1 个节点）的父亲节点的第 $(k-\text{place}(v))$ 个祖先。

在上述递归查询的过程中，每递归一层，查询节点所在链的长度至少 +1，所以至多会递归 $O\left(\sqrt{n}\right)$ 层（因为 $\sum_{i=1}^{2\lceil\sqrt{n}\rceil}=\lceil\sqrt{n}\rceil\cdot\left(2\lceil\sqrt{n}\rceil+1\right)>n$），于是每次查询的时间复杂度为 $O\left(\sqrt{n}\right)$。

例如在图 11 中查询节点 17 的第 5 个祖先：$\text{LA}(17,5)=\text{LA}(15,4)=\text{LA}(12,3)=\text{LA}(1,0)=1$。

$O(n)-O(\log n)$ 的梯子剖分（ladder decomposition）算法

在长链剖分的基础上，将每条链（的节点序列）反向（即向根节点的方向）延长一倍的长度（如果延长时碰到根节点就停止），称延长后的链为“梯子”，同时保持每个节点所属的链（梯子）不变（虽然这时每个节点可能会被若干个梯子覆盖）。记节点 v 是它所在梯子的节点序列的第 $\text{place}(v)$ 个节点。

在每次查询节点 v 的第 k 个祖先时，如果 $\text{place}(v)>k$，那么同样取节点 v 所在梯子的节点序列的第 $(\text{place}(v)-k)$ 个节点就是答案；否则，我们可以查询节点 v 所在梯子顶部节

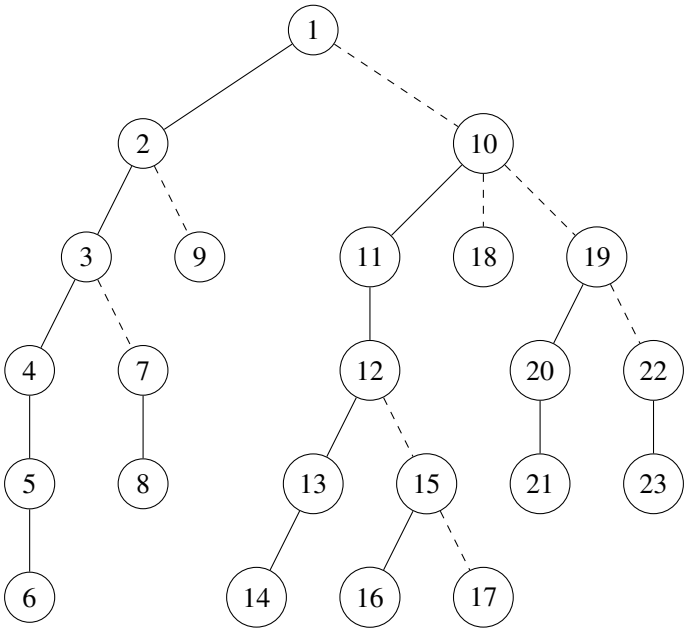


图 11: 例子：一棵树的长链剖分

点（即节点序列的第 1 个节点）的第 $(k - \text{place}(v) + 1)$ 个祖先。

在上述递归查询的过程中，每递归一层，查询节点的**高度**（记每个叶子节点的高度为 1，其它节点的高度是所有儿子高度的最大值 +1）至少 ×2，所以至多会递归 $O(\log n)$ 层，于是每次查询的时间复杂度为 $O(\log n)$。

例如在图 11 中查询节点 17 的第 5 个祖先：$\text{LA}(17, 5) = \text{LA}(15, 4) = \text{LA}(11, 2) = 1$。（长链 (17) 对应着梯子 (15, 17)，长链 (15, 16) 对应着梯子 (11, 12, 15, 16)，长链 (10, 11, 12, 13, 14) 对应着梯子 (1, 10, 11, 12, 13, 14)）

$O(n \log n) - O(1)$ 的算法

结合上述倍增算法与梯子剖分算法，我们可以得到一个 $O(n \log n) - O(1)$ 的算法。

首先分别花 $O(n \log n)$ 和 $O(n)$ 的时间做这两个算法的预处理部分。

每次查询节点 v 的第 k 个祖先时，记节点 v 的**高度**为 $\text{height}(v)$，我们先用倍增算法跳第一步，设跳的距离为 len（$len \geq \left\lceil \frac{k}{2} \right\rceil$），跳到的点为 u，则点 u 的高度至少为 $\text{height}(v)+len$。接下来使用梯子剖分算法查询点 u 的第 $(k - len)$ 个祖先，这时我们发现因为 $\text{height}(u) \geq \text{height}(v) + len \geq \frac{k}{2} \geq k - len$，所以点 u 的第 $(k - len)$ 个祖先一定在点 u 所在的梯子上，可以在 $O(1)$ 的时间内计算出答案。

$O(n + L\log n) - O(1)$ 的算法

这里 L 表示树中叶子的个数。

我们可以在 $O(n)$ 的时间内预处理出每个节点子树中的任意一个叶子，以及这个节点到这个叶子的距离。这样我们就能在 $O(1)$ 的时间内将问题从查询任意一个节点的第 k 个祖先转化成查询任意一个**叶节点**的第 k' 个祖先。

由于上述算法中我们只需要利用倍增算法跳的第一步，所以我们只需要预处理每个**叶节点** v 的第 1、2、4、8、……、2^i、……、2^l（其中 $l = \lfloor \log_2(\text{depth}(v)) \rfloor$（depth 为深度））层祖先。这可以在 $O(n + L\log n)$ 的时间内完成——我们可以对整棵树进行一遍 DFS，同时使用 `std::vector` 或数组等数据结构（而不是 `std::stack`）维护深搜栈，这样每当访问到一个叶节点时，深搜栈里依次存储着这个叶节点到根的路径上的所有点，于是我们就可以在 $O(\log n)$ 的时间内求出这个叶节点的第 1、2、4、8、……、2^i、……、2^l 层祖先。

查询时跳完第一步以后同样使用梯子剖分算法即可，每次查询的时间复杂度仍为 $O(1)$。

不幸的是，由于最坏情况下 $L = O(n)$，所以这个算法最坏情况下仍需要 $O(n\log n)$ 的时间预处理。

Method of Four Russians

为了减少叶子的个数，我们可以对树中靠近叶子的部分进行分块。设分块大小为 S，将所有（在原树中的）子树大小不超过 S 的节点删掉，那么新树只有不超过 $\frac{n}{S}$ 个叶节点（因为新树中每个叶节点在原树中对应的子树大小都大于 S），可以应用上述算法，在 $O\left(n + \frac{n\log n}{S}\right)$ 的时间内预处理新树中的每个叶节点的所有 2 的幂层祖先以及新树的梯子剖分，这样就能在 $O(1)$ 的时间内回答在新树中的点的询问。

对于被删掉的点的询问，我们可以在 $O(n)$ 的时间内预处理每个被删掉的点与最近的没被删掉的点（即新树中的叶节点）的距离，这样如果询问第 k 个祖先时 k 大于等于这个距离，我们就能在 $O(1)$ 的时间内将原询问转化为对新树中的对应叶节点的询问，也就能在 $O(1)$ 的时间内回答。

现在只剩下一种情况还没被解决：询问的点和答案的点都被删掉了。观察所有被删掉的点，它们构成了每棵树大小不超过 S 的一个森林。

考虑一棵大小不超过 S 的有根树，我们可以对它进行欧拉遍历，即从根节点开始 DFS，每走过一条向下的边就记一个左括号，每走过一条向上的边就记一个右括号，以得到一个长度不超过 $2S$ 的括号序列。长度不超过 $2S$ 的括号序列的数量为 Catalan 数的前 S 项的和，显然是 $O\left(4^S\right)$ 的（具体地说，大约为 $\frac{4^S}{S^{3/2}}$ 级别，但我们并不需要这么精确的复杂度）。我们可以对每个长度不超过 $2S$ 的括号序列按 DFS 序还原出一棵大小不超过 S 的有根树，对每棵有根树使用暴力算法在 $O\left(S^2\right)$ 的时间内计算出每个点的每个祖先，总时间复杂度为 $O\left(4^S \cdot S^2\right)$。

我们可以在 $O(n)$ 的时间内预处理出这个森林中每棵树对应的括号序列，以及每个节点的标号与它在这棵树中 DFS 序列中的位置的一一对应关系。这样对于每次询问的点和答案的点都被删掉了的查询，我们可以对询问点所在的大小不超过 S 的树对应的括号序列查表得出它的第 k 个祖先在 DFS 序中的位置，再对应到这棵树中的对应节点标号即可。

综上，我们做到了每次 $O(1)$ 查询，同时预处理的时间复杂度为 $O\left(n+\frac{n\log n}{S}+4^S\cdot S^2\right)$，取 $S=\frac{\log n}{4}$ 即可做到 $O(n)$！

4.2.4 小结

应用 Method of Four Russians 的条件是：可以选择一个分块大小 S，使得将问题分块后，可能出现的不同的块的数量远小于原问题的规模。像一般的矩阵乘法（例如矩阵大小为 1000×1000，矩阵中每个元素的绝对值不超过 10^6），就算分块成 1×1 的矩阵，可能出现的不同的块的数量也很多，为矩阵中每个元素的取值个数的平方，这样就不能应用 Method of Four Russians 了。

如果可能出现的不同的块的数量为 $O\left({c_1}^{S^c}\right)$（$c$、$c_1$ 为常数，$c_1>1$），那么一般可以取分块大小为 $S=c_3\log^{\frac{1}{c}}n$（c_3 为较小的常数，一般不超过 1），从而在预处理的时间复杂度不超过原算法的时间复杂度的前提下，将原算法的数据规模复杂度除以 $\log^{\frac{1}{c}}n$。

5 总结

分块算法将原问题拆分为整块和不完整的块两个子问题，分别解决，通过选择 $S=\sqrt{n}$ 等分块大小平衡两部分的运行效率，从而达到一个较为优秀的时间复杂度。当这两个子问题的时间复杂度不同时，非常规大小的分块算法便应运而生了——最优的分块大小不仅可能是 $O\left(\sqrt{n}\right)$，也可能是 $O\left(\sqrt{nw\log n}\right)$ 或 $O\left(n^{2/3}\right)$，甚至可能是 $O(\log n)$ 或 $O\left(\sqrt{\log n}\right)$……

本文初探了非常规大小的分块算法，介绍了分块算法的一些优化方法，也介绍了非常规大小的分块算法优化一些其它算法的应用——其中 LCA、RMQ、LA 这三个问题可以优化到理论最优的时间复杂度。使用非常规大小分块算法，不仅可以对许多区间操作问题做到一个比常规大小（$\sqrt{n}$）分块算法优秀的时间复杂度，还能优化一些其它算法。希望这些例子能对读者有所启发。

致谢

感谢中国计算机学会提供学习与交流的平台。

感谢胡伟栋老师多年以来给予的关心和帮助。

感谢其他对作者有过帮助和启发的老师和同学们。

感谢我的父母对我无微不至的关心和照顾。

参考文献

[1] 罗剑桥，《浅谈分块思想在一类数据处理问题中的应用》，2013 年国家集训队论文

[2] 王子昱，《分块方法的应用》，2013 年国家集训队论文

[3] 邹逍遥，《浅谈分块在一类在线问题中的应用》，2015 年国家集训队论文

[4] 黄哲威，“【分块】数列分块入门1-9 by hzwer”， http://hzwer.com/8053.html

[5] 王逸松，《挑战》讲评课件，2017 年全国冬令营

[6] 毛啸，《再探快速傅里叶变换》，2016 年国家集训队论文

[7] 埃氏筛法的英文维基百科， https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

[8] Mertens 定理的英文维基百科， https://en.wikipedia.org/wiki/Mertens%27_theorems

[9] Method of Four Russians 的英文维基百科， https://en.wikipedia.org/wiki/Method_of_Four_Russians

[10] Prof. Erik Demaine, “Lecture 15 — April 12, 2012”, 6.851: Advanced Data Structures (Spring 2012) (https://courses.csail.mit.edu/6.851/spring12/scribe/lec15.pdf)

[11] LCA 的英文维基百科， https://en.wikipedia.org/wiki/Lowest_common_ancestor

[12] RMQ 的英文维基百科， https://en.wikipedia.org/wiki/Range_minimum_query

[13] LA 的英文维基百科， https://en.wikipedia.org/wiki/Level_ancestor_problem

回文树及其应用

中山市中山纪念中学 翁文涛

摘 要

本文介绍了一种较为新颖的处理回文子串的数据结构，回文树。本文首先阐述了回文树的结构及基础构造，接下来进行了针对回文树的一些扩展，最后用若干习题展示了回文树解决问题的能力。

1 引言

回文串是一种十分特殊的字符串，拥有很多优美的性质。近年来，算法竞赛中有关回文串的题目比较热门，但由于与回文串相关的算法比较贫乏，导致题目的解法比较单一。回文树是一种新兴的数据结构，由Mikhail Rubinchik在2015年发表，国内暂时没有对其比较深入的研究。因此本文将着重介绍回文树，并展示回文树强大的可塑性与解决问题的能力，希望能给读者带来一些思路上的启发。

本文的结构如下：

第二节给出一些文中的记号与定义。

第三节介绍了回文树的基本结构与构造。

第四节列举了回文树的若干扩展，具体介绍了如何在回文树前后端插入删除，如何对TRIE建回文树以及如何将回文树可持久化。并配备了一些例子与题目便于读者理解与扩展。

第五节给出了若干算法竞赛中典型的回文串问题，并展示了如何用回文树这一强力工具解决这些问题。

2 一些记号与定义

记号 1. Σ表示字符集大小。

记号 2. $|s|$表示字符串s的长度。s_i表示s的第i个字符。$s[l..r], 1 \le l \le r \le |s|$表示由串$s$中的第$l$个字符到第$r$个字符所组成的子串。

记号 3. 记$pre[s][i]$表示$s[1..i]$，称为s的前缀i。$suf[s][i]$表示$s[i..|s|]$，称为s的后缀i。

记号 4. 设有串t与字符c，记tc表示在t后接上字符c对应的字符串。ct表示在t前接上字符c对应的字符串。

定义 2.1. 对于两个串s,t，定义s与t不同，当且仅当满足以下条件中至少一条：

- $|s| \neq |t|$
- $|s| = |t|$，且存在一个位置$j, 1 \leq j \leq |s|$，满足$s_j \neq t_j$。

定义 2.2. 对于一个串$s = s_1 s_2 \cdots s_n$，定义其翻转为$s_n s_{n-1} \cdots s_1$

定义 2.3. 一个串t为s的子串，当且仅当存在$1 \leq l \leq r \leq |s|$，使得$t = s[l..r]$。注意空串不是s的子串。对应s的两个子串$s[a..b], s[l..r]$，他们是不同的，当且仅当其代表的字符串不同。

定义 2.4. 一个串s是回文的，当且仅当s的翻转与s相同。

定义 2.5. 一个串t为s的回文子串，当且仅当t为s的子串且t是回文的。

定义 2.6. 一个串t为s的回文后缀，当且仅当$t \neq s$且t为s的一个后缀且t是回文的。s的最长回文后缀定义为长度最长的，且不为s的回文后缀的串。注意一个串可能不存在回文后缀，定义他的回文后缀只包含一个空串。同样的也可以定义一个串的回文前缀与最长回文前缀。

3 回文树的结构与构造

3.1 结构

一个串s的回文树为一个森林，由两棵树组成。设两棵树的根分别为$even, odd$。树上每个节点对应着一个字符串，除根外与s的回文子串一一对应。树上每条边都有对应的一个字符，且满足一个点的所有出边对应的字符各不相同。对于树上一个点i，令fa_i 为其父亲，则i 对应的字符串s_i为fa_i 对应的字符串s_{fa_i}在两端加上连接i与fa_i的边对应的字符c，即$s_i = cs_{fa_i}c$。特别的，对于根$even$，令其对应的字符串为空串，根odd 对应的字符串为一长度为-1 的实际并不存在的字符串，odd的儿子对应的字符串为连出边对应的字符。在本文中，会直接用树上一个点指代其对应的字符串。

此外，对于回文树上一点i，定义其失配指针$fail_i$，指向i的最长回文后缀在回文树上对应的节点。特别的，定义$fail_{even}$与$fail_{odd}$均为odd。

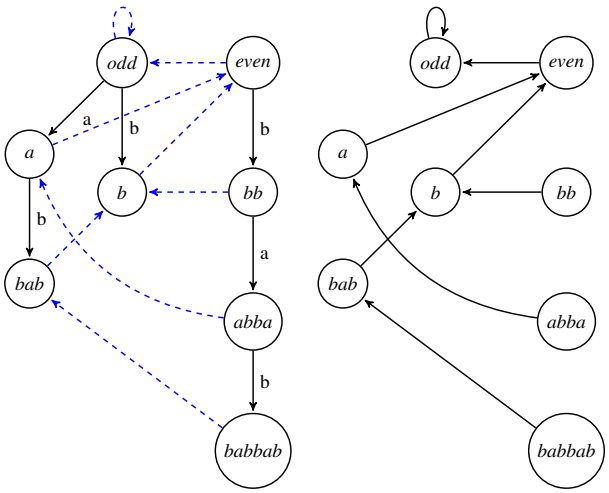
具体而言，对于回文树上每个节点，记录以下三种值：

- len，表示该节点对应的字符串的长度，特别的$even$与odd的len分别为0与-1。
- $go[c]$，该节点的对应字符为c的出边连接的节点，假如不存在则为空。
- $fail$，该节点的失配指针指向的节点。

由$fail$指针可以定义一个字符串s的$fail$树为以$fail$指针为连边所生成的树。对于$fail$树上一点i，定义其$fail$链为一个集合，等于$fail_i$的$fail$链并上i。特别的，定义odd 的$fail$ 链为其本身。

例 1. 串babbab对应的回文树与$fail$树

在图中用蓝色虚线代表一个节点的$fail$转移，实线代表go转移，实线上的字母代表转移的字符，一个点上的字符串代表其对应的s中的回文子串。



(a) babbab的回文树 (b) babbab的fail树

3.2 节点数以及转移数

可以发现的是，假如一个串s有n个不同的回文子串，那么s对应的回文树的节点数就是$n+2$，问题变为分析s的不同回文子串的个数。

定理 3.1. 对于一个字符串s，不同的回文子串个数最多只有$|s|$个。

证明. 使用数学归纳法。

- 当$|s|=1$时，只有$s[1..1]$一个子串，并且他是回文的，所以结论成立。
- 当$|s|>1$时，设$s=s'c$，其中c为s的最后一个字符，并且结论对s'成立。考虑以末尾字符c为结尾的回文子串，假设他们的左端点从左到右依次为$l_1, l_2, \cdots, l_k$，那么由于$s[l_1..|s|]$ 为回文串，那么对于所有的位置$l_1 \le p \le |s|$，都会有$s[p..|s|] = s[l_1..l_1+|s|-p]$，所以对于回文子串$s[l_i..|s|]$，都会有$s[l_i..|s|] = s[l_1..l_1+|s|-l_i]$，当$i \neq 1$ 时，总会有$l_1+|s|-l_i < |s|$，从而$s[l_i..|s|]$已经在$s[1..|s|-1]$中出现，因此每次不同的回文串最多新增一个，即$s[l_1..|s|]$。因此结论对于s依然成立。

由数学归纳法可知定理3.1成立。 □

因此回文树的状态是$O(|s|)$级别的。考虑转移，一个点的转移数是$O(\Sigma)$的，但对于一个状态，能转移到他的节点其实是唯一的，所以总转移数是$O(|s|)$的。一个节点只有一个失配指针，因此回文树的总节点数与总转移数都是$O(|s|)$的。

3.3 构造方法

我们采用增量法来构造出字符串s的回文树。假设已经构造出s的回文树，接下来需要在s的末尾加入一个新的字符c，并维护出sc的回文树。

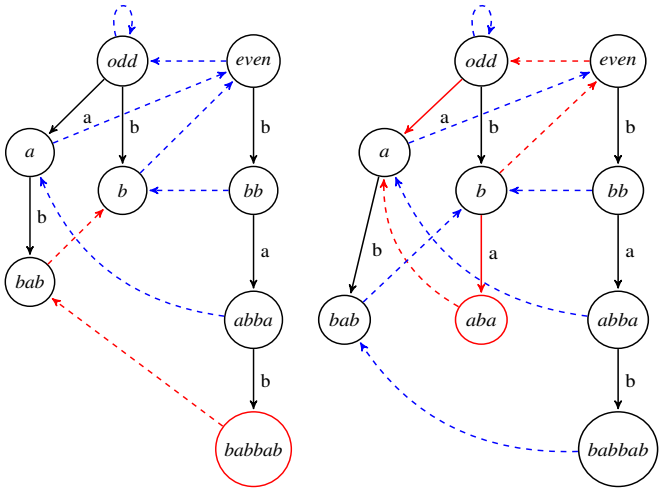
定理 3.2. *以新加入的字符c为结尾的，未在s中出现过的回文子串最多只有1个，且必为sc的最长回文后缀。*

证明. 可由3.1的证明过程直接推得。 □

每次只需求出sc的最长回文后缀即可。不妨设sc的最长回文后缀为$s[i..|s|]c$，那么由回文串的定义，要么$i > |s|$，即最长回文后缀只有c这个字符，要么$s[i+1..|s|]$也是一个回文串。也就是说，$s[i+1..|s|]$必须是s的回文后缀，或是长度为0或-1的串（方便来讲就是回文树上的*even*和*odd*）。现在要在s的最长回文后缀对应的$fail$链中找到长度最大的一个节点t，满足$s[|s|-len_t]=c$，则sc的最长回文后缀就是ctc。

插入时，假如回文树上不存在一个节点代表ctc，则需要新建一个新的节点来代表ctc，否则直接使用这个节点。假如新建了一个节点，还需要求出这个新节点的$fail$ 指针对应的节点，相当于在$fail[t]$ 对应的$fail$ 链中再找一个长度最长的节点v满足$s[|s|-len_v]=c$。这里需要注意假如$len_t=-1$，那么ctc的$fail$指针应当指向长度为0 的节点，否则ctc最长回文后缀必然在回文树上，直接将指针指向对应节点即可。

例 2. 比如$s = babbab$，要加入的新字符为a。s的最长回文后缀为$babbab$，其$fail$链对应为$\{babbab, bab, b, even, odd\}$，最长的合法的回文子串为$b$，因此$sa$的最长回文后缀就是$aba$，再继续寻找$aba$的最长回文后缀，合法的“回文子串”为$odd$，因此$aba$的最长回文后缀就是$odd$的转移$a$对应的节点，就是$a$。



(c) babbab的回文树　　(d) babbaba的回文树

在具体实现中，只需要在回文树中从最长回文后缀开始沿着$fail$指针往根走，直到合法为止。**称这种插入算法为基础插入算法**。

定理 3.3. *这种实现的总时间复杂度为$O(|s|\log\Sigma)$，空间复杂度为$O(|s|)$。*

证明. 考虑总共会沿$fail$链往上跳多少次。将贡献分为两部分统计，一种是寻找新串的最长回文后缀的贡献，一种是寻找新节点的最长回文后缀的贡献，显然这两部分的贡献是同阶的，因此只考虑前一部分的贡献。不妨设势能函数$\Phi(s)$等于s的最长回文后缀的长度。考虑在s后面添加新的字符c，首先会有$\Phi(sc) \le \Phi(s)+1$，每沿着$fail$链往上走1步，$\Phi(sc)$就会减少1，并且始终会有$\Phi(sc) \ge 0$，所以势能总共只会增加$O(|s|)$，也总共只会减少$O(|s|)$，因此在$fail$链上跳的复杂度是$O(|s|)$的。

每次加入新字符时，需要检查节点t是否已经存在转移边c，假如存在则无需添加新节点，否则添加。需要对每个节点建一棵平衡树（或直接使用c++中的map）来记录他的转移，每次查询的复杂度为$O(\log\Sigma)$的，总的时间复杂度为$O(|s|\log\Sigma)$，空间复杂度为$O(|s|)$。在实际情况中Σ可能较小，可以直接对每个节点开一个数组来记录转移，假如不计算开辟数组的开销，时间复杂度可以降为$O(|s|)$，但空间复杂度为$O(|s|\Sigma)$。 □

3.4 基础应用

例 3. 经典问题

给定一个只包含小写字母的字符串s，对于s的每个前缀$s[1..i]$，求出其不同的回文子串个数以及位置不同的回文子串个数。

$|s| \leq 10^5$

由于回文树是增量构造，因此可以在加入每个字符后得到对应的回文树结构。不同的回文子串个数就是当前回文树上的节点数量减2（有两个根）。位置不同的回文子串数量，相当于之前的数量加上以当前位置为结尾的回文串数量，这就等于这个串的回文后缀的数量，就等于这个点的$fail$链的长度（注意odd与$even$不算入长度）。在回文树上另外维护一个节点$fail$链上的节点数量。这种做法的时间复杂度为$O(|s|\log\Sigma)$，空间复杂度为$O(|s|)$。

例 4. Palindrome [49]

考虑一个只包含小写字符的字符串s，定义s的一个子串t的出现值为t在s中的出现次数乘上t的长度。求出s的所有回文子串中的最大出现值。

$|s| \leq 300000$

首先建出s的回文树。对于回文树上的一个节点t，定义$right_t$为一个端点集合，表示t在s中的出现位置的右端点集合，那么t的出现值就是$len_t \times |right_t|$，只需要对每个节点求出$|right_t|$即可。对于s的每个前缀$pre[s][i]$，可以求出其最长回文后缀，设为u，将i加入$right_u$当中。做完每个前缀后，每个节点的$right$就是$fail$树上他的儿子的$right$的并集，可以证明的是，$fail$树上一个点的儿子之间的$right$两两无交，所以其$right$的大小就是所有儿子的$right$的大小的和。这种做法的时间复杂度也是$O(|s|\log\Sigma)$，空间复杂度为$O(|s|)$。

4 回文树的扩展

4.1 前端插入

在之前回文树的构造中，实现了在一个串末端加入一个字符并维护出新的回文树的操作。现在可以尝试对字符串的开头加入一个字符并维护新的回文树。

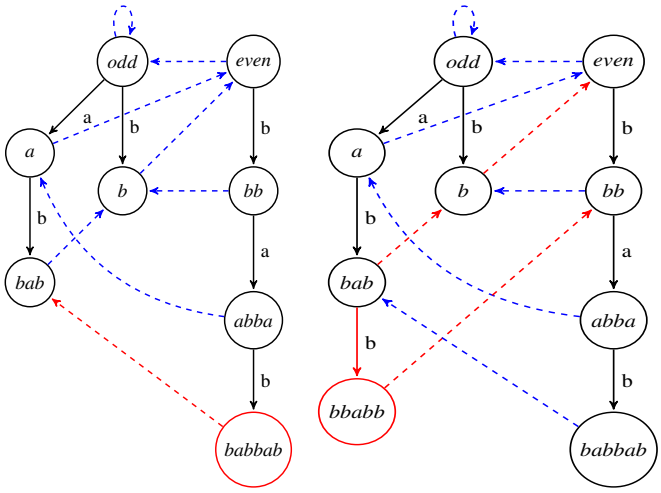
[49] APIO2014, "Palindromes"

与末端插入类似的，可以发现在前端插入时，比如当前字符串为s，要求出cs的回文树，那么以c为开头的产生的新的回文串最多只有1个，并且是cs的最长回文前缀。并且由之前的过程可知，只需要找到s的一个最长的回文前缀t，满足$s[|t|+1]=c$，那么新的最长回文前缀就是ctc。要找到t，不妨在回文树上对每个节点再维护一个$fail'$指针，表示这个节点的最长回文前缀指向的节点，那么每次加入新字符的时候，只需要沿着s的最长回文前缀的$fail'$链找到一个最长的合法的t即可。

注意到回文树上的每个节点都是一个回文串（除了根节点），对于一个回文串t，会有t的翻转与t相同，也就是说，对于t的每个回文后缀$t[i..|t|]$，他的翻转与$t[1..|t|-i+1]$的是相同的，而他本身就是一个回文串，因此$t[i..|t|]$的翻转就等于$t[i..|t|]$，也就是说$t[i..|t|]=t[1..|t|-i+1]$，所以，t的回文前缀与回文后缀的字符串集合其实是相同的。也就是说，回文树上一个节点的$fail'$指针，其实就是它的$fail$指针。

因此只需要维护出字符串s的最长回文前缀与最长回文后缀，就能支持字符串的前端和末端插入了。需要注意在前（后）端插入时可能会对最长回文后（前）缀造成影响，另外维护一下就好了。时间复杂度的分析与之前类似，可以得到也是总时间复杂度$O(|s|\log\Sigma)$，空间复杂度$O(|s|)$。

例 5. 比如$s=babbab$，要在开头加入的新字符为b。s的最长回文前缀为$babbab$，其$fail$链对应为$\{babbab, bab, b, even, odd\}$，最长的合法的回文子串为$bab$，因此$bs$的最长回文前缀就是$bbabb$，再继续寻找$bbabb$的最长回文后缀，合法的“回文子串”为$even$，因此$bbabb$的最长回文后缀就是$even$的转移$b$对应的节点，就是$bb$。



(e) babbab的回文树　　(f) bbabbab的回文树

例 6. Victor and String [50]

一开始有一个空串s。接下来进行n次操作，操作有以下4种：

- 在s的前端加入一个给定的字符。
- 在s的末端加入一个给定的字符。
- 询问当前s中有多少不一样的回文串。
- 询问当前s中有多少位置不同的回文串。

$n \leq 10^5$

假如没有前端插入，那么本题就与3完全一样了，只要在回文树上另外维护每个节点在$fail$树上的深度，在末端加入字符时维护好位置不同的回文串的数目即可。现在需要支持前端插入。问题依然是类似的，s中不同的回文串数目依然是回文树上的节点数目，但每个节点的$right$集合似乎就不太好维护了。注意到每次前端插入时，只需要对$right$集合的大小做出更新，因此只需要对加入新字符后的最长回文前缀的$right$集合大小加1。位置不同的回文串数目就相当于加了最长回文前缀对应节点的$fail$链长度。

4.2 删除后端字符

考虑一个更加有趣的问题，给定一个串s以及对应的回文树，是否能求出删除s的最后一个字符后新的字符串对应的回文树呢？

一种很简单的思路就是，对于s串的每个前缀都记录下它的最长回文后缀，回文树上每个节点记录下这个节点作为一个前缀的最长回文后缀的次数。当删除s 的最后一个字符的时候，相当于删掉其最长回文后缀（因为其他回文后缀依然在s中存在），直接在回文树上对应的节点将其次数减1，假如一个节点的次数变为0了就将这个节点从回文树上删去。但很可惜的是，这种做法在带插入时的复杂度是不对的，考虑下面的一个例子：

例 7.

设$push(c)$表示在末尾加入一个字符c，$pop()$表示删除末尾的字符。考虑以下操作序列：

$$\underbrace{push(a), \cdots, push(a)}_{n}\ \underbrace{push(b), pop(), \cdots, push(b), pop()}_{n}$$

可以发现对于每个$push(b)$，都会在$fail$链上往上跳n次，每次把b删掉后这些被跳过的势能就会恢复回来，所以总的复杂度就是$O(n^2)$。

[50] Bestcoder #52 1004, "Victor and String"

这个算法的漏洞在于，回文树的插入部分的复杂度是通过势能分析得到的，假如可以删除一个字符，我们就能不断的重复进行势能消耗大的操作，从而使得整体的复杂度变为$O(n^2)$。因此，要想实现删除操作，首先应该找到一个复杂度不依赖于势能分析的、稳定的插入算法。

4.3 一种不基于势能分析的插入算法

之前的插入算法的瓶颈在于，每次插入一个字符c，都要沿着当前最长回文后缀t的$fail$链往上找到第一个v使得v在s中的前驱（即v的前一个字符）为c。注意到除了$v = t$的情况，v的前驱总是在t内的，也就是说对于每个t，是要找到一个最长的t的回文后缀满足其前驱为c，这个回文后缀只与t相关而不与s相关。因此，对于每个回文树中的节点t，另外维护一个失配转移数组$quick[c]$，存储t的最长的满足前驱为c的回文后缀。那么在插入时，我们只需要首先检查当前最长回文后缀t的前缀是否为c，假如不是，那么合法的v直接就是t的$quick[c]$。

接下来考虑怎么维护每个节点的$quick$。对于一个节点t，t的$quick$与$fail_t$的$quick$几乎没有什么差别，因为t的回文后缀只是在$fail_t$的回文后缀中再加入了$fail_t$而已。首先将把 $fail_t$ 的 $quick$ 复制一遍作为t的$quick$，令c为$fail_t$在t中的前驱，用$fail_t$更新t的$quick[c]$即可。直接暴力操作每次插入的时空复杂度都是$O(\Sigma)$。

可以将每个t的$quick$可持久化，一种做法是用可持久化线段树来可持久化数组。每次复制时只需要将版本复制，复杂度降为$O(1)$。插入的时空复杂度为$O(\log \Sigma)$。最后可以得到一个不基于势能分析的单次插入时空复杂度为$O(\log \Sigma)$ 的算法。

4.4 前后插入、删除

接下来尝试做到同时支持对字符串前后插入、删除并维护出新的回文树。

首先引入一个概念，一个s中的子串$s[l..r]$被称为**重要的**，当且仅当$s[l..r]$是回文的，且不存在一个$r' > r$，使得$s[l..r']$是回文的，不存在一个$l' < l$，使得$s[l'..r]$是回文的。对于每个位置r，显然最多只有一个$l \le r$，满足$s[l..r]$是重要的，称$s[l..r]$为r的**重要后缀**，r可以没有重要后缀；同样的，可以对l定义**重要前缀**。接下来考虑4种操作的影响：

- 在末端插入新的字符

  不妨设r表示原来末端的位置，那么r必然会有一个重要后缀，而这个后缀就是当前s的最长回文后缀，直接套用末端插入的算法求出新的回文树并得到新的最长回文后缀，令l为新最长回文后缀的左端点，则$s[l..r+1]$是重要的。注意可能会让$s[l..r+1]$的最长回文前缀变得**不重要**。另外维护一个桶$cnt[l][r]$表示$s[l..r]$被标记了多少次“不重要”，$cnt[l][r]$可以用哈希来实现。

- 在末端删除字符

  令l为r的重要后缀的左端点，将$s[l..r]$的最长回文前缀对应的cnt减1，假如被减为0，意味着他再次变成**重要的**。现在还有一个问题是，我们需要知道$s[l..r]$对应的回文树的节点是否需要被删掉。不妨对回文树上每个节点t定义两个权值$imp_t, child_t$，imp_t表示有多少$1 \le l' \le r' \le |s|, s[l'..r'] = t$且$s[l'..r']$是重要的，$child_t$表示在$fail$树上$t$的儿子数。当改变一个串的**重要情况**时，需要同时改变其回文树上对应点的imp值。假如一个点的imp与$child$都是0了，这个点就不可能在s中出现，直接把他删掉。注意每次最多只可能删掉一个点，同时在删掉一个点时需要注意对其$fail$的$child$也减1。

- 在前端插入新的字符

  与在末端插入类似的，只需要找到新的最长回文前缀，并对其最长回文后缀的重要情况与cnt做出修改即可。

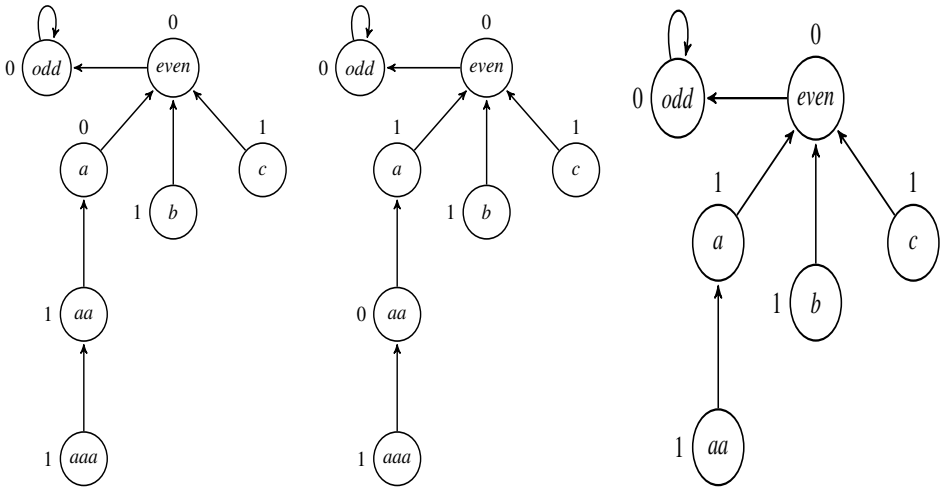
- 在前端删除字符

  与在末端删除类似，只需要处理好重要情况的变化与回文树的变化即可。

例 8. 考虑当前有一个字符串$aacbaaa$，它的重要的子串为$s[1..2], s[3..3], s[4..4], s[5..7]$，将回文树中一个节点的$imp$值标注在节点一侧，则$aacbaaa$对应的$fail$树为12(g)。

将首字母删去，字符串变为$acbaaa$，重要子串变为了$s[1..1], s[2..2], s[3..3], s[4..6]$，对应的$fail$树为12(h)。

再将尾字符删去，字符串变为$acbaa$，重要子串变为$s[1..1], s[2..2], s[3..3], s[4..5]$，对应的$fail$树为12(i)。



(g) aacbaaa的fail树 (h) acbaaa的fail树 (i) acbaa的fail树

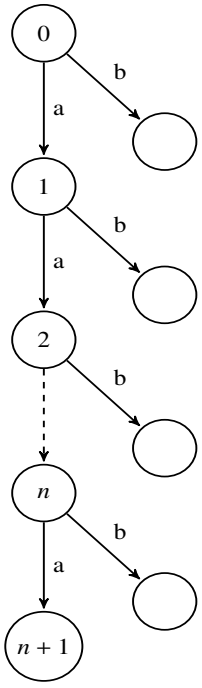
4.5 TRIE上建回文树

给定一棵TRIE，TRIE上的节点数为n，TRIE上的有效字符串定义为从根出发到任意节点所经过的边的字符所组成的字符串，现在要求出这棵TRIE所有有效字符串的回文树的并集。

类似于一个串s只有$O(|s|)$种不同的回文子串，可以证明，对于一棵节点数为n的TRIE，其有效字符串的回文子串所组成的集合中元素个数依然是$O(n)$，考虑一个节点对应的字符串，相当于在他父亲的字符串后接入一个字符，那么最多就只会新增一个回文子串，并且必然为其对应字符串的最长回文后缀，因此不同的回文子串个数是$O(n)$的，所以回文树的大小也是$O(n)$ 的。对于TRIE上每个节点，记录其对应的字符串的最长回文后缀在回文树上对应的节点，那么每次扩展到其儿子时，直接套用不基于势能分析的插入算法，求出儿子对应的最长回文后缀即可。总的时空复杂度就是$O(n \log \Sigma)$。

但这里有一个有趣的问题，假如套用最基础的插入算法，复杂度会是多少呢？

定理 4.1. *对于下面的这棵TRIE，最基础的插入算法的复杂度为O(所有叶子深度总和)。*



证明. 定义$\Phi(i)$表示树上的节点i的最长回文后缀在fail树上的深度，那么显然有$\Phi(i) = i$。对于每个节点i延伸出来的无标号节点，因为转移边上的字符与之前的字符均不相同，所以必定会把$\Phi(i)$完全消耗，所以总势能的消耗就是$\sum_{i=1}^{n} \Phi(i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O$(所有叶子深度总和)。

□

所以对于TRIE建回文树必须要用不基于势能分析的插入算法，否则时间复杂度会退化为$O(n^2)$。

例 9. Tree Palindromes [51]

给定一棵以1为根的N个点的树，树上每条边都有一个小写字母。对于树上每个点，求出从根到这个点形成的字符串的最长回文后缀，输出所有点的答案的和。

$N \leq 100000$

题中给出的树就是一棵TRIE。直接对这个TRIE建回文树，在建回文树时可以求出以每个点结尾的最长回文后缀。本题字符集较小，在插入的时候，没有必要将失配转移可持久化，直接开一个数组记录即可。时间复杂度$O(N\Sigma)$。

4.6 可持久化回文树

最基础的插入算法是基于势能分析的，假如将其可持久化那么复杂度是不对的。所以考虑将不基于势能分析的插入算法进行可持久化。回文树上一个节点的转移是用一个平衡树或是长度为Σ 的数组存储的，假如用平衡树，可以用可持久化TREAP，那么就只需要记录其对应版本即可，但这种做法的代码量是巨大的，另外一种做法是，与用可持久化线段树存储失配转移一样，将长度为Σ的数组也用可持久化线段树来存储，这样也只需要记录版本信息了。

对于当前版本的回文树上的一个节点版本，只需要记录其长度，$fail$指针对应的节点版本，以及转移边和失配转移对应的线段树版本即可。这种做法中，单个节点的空间复杂度为$O(1)$。对于每个版本的回文树，用一个可持久线段树来存储，每个叶子节点存储对应的回文树上节点版本的信息，每次插入可能会需要新建一个回文树上的节点，还要存储这个版本对应的最长回文后缀。这个算法单次插入的时空复杂度都是$O(\log(|s| + \Sigma))$。

很显然地，我们可以将可持久化回文树与支持前后插入删除的回文树结合在一起，但由于笔者也未曾实现这个版本的回文树，所以无法估计最终的代码实现复杂度，有兴趣的读者可以尝试实现。

[51] ICPC Preparatory Series by Team Akatsuki, "Three Palindromes"

5 应用

5.1 Country [52]

给定一个长度为n的字符串s，现在要从s中选出尽量多的子串，满足选出的子串都是回文的，并且不存在一个串是另外一个串的子串的情况。

$n \leq 100000$

首先建出s的回文树，那么对于回文树上的每个节点都是可以选择的串。现在要解决一个串不能是另外一个串子串的问题。

定理 5.1. *若对于两个回文串a和b，有a是b的子串。那么，要么$a = b$，要么a是b的最长回文后缀的子串，要么a是b缩去两侧字符后形成的回文串的子串。*

证明. 只需要考虑a是b的后（前）缀的情况。因为a是一个回文串，那么假如a是b的后（前）缀，那么a就必然是b的一个回文后（前）缀，那就必然是b的最长回文后（前）缀的后（前）缀。 □

根据这个定理，可以构建一个有向图G，G上的节点对应着回文树上的节点，对于一个点i，连一条从i出发到回文树上i的父亲的单向边，以及一条从i出发到$fail_i$的单向边，那么假如点j是i的子串，必然存在一条从i出发到j的路径。现在的问题转化为：给定一个有向图G，要从G中选出尽量多的节点使得两两不存在路径。这就是一个最长反链问题。根据Dilworth定理，一个有向无环图的最长反链等于这个图的最小链覆盖，直接对图G求最小链覆盖即可。时间复杂度为$O(Maxflow(n,n))$

5.2 Palindromeness [53]

定义一个字符串的回文指数如下：

- 若一个字符串不是回文串，则回文指数为0。

[52] GDKOI2013 大山王国的城市规划

[53] Codechef April Lunchtime 2015, "Palindromeness"

- 仅有一个字符的字符串回文指数为1。
- 递归定义其他回文字符串s，等于1+前$\lfloor\frac{|s|}{2}\rfloor$个字符所组成的子串的回文指数。

给定一个字符串s，求出s的所有子串的回文指数的和。
$|s| \leq 10^5$

建出s的回文树，并维护出每个节点的*right*集合大小，那么剩下的问题就是求出每个回文串的回文指数。考虑对于回文树上每个节点i再维护一个指针$half_i$，指向最长的长度不超过i的一半的i的回文后缀。假如$half_i$的长度恰好为$\lfloor\frac{|s|}{2}\rfloor$，$i$的回文指数就等于$half_i$的回文指数加1，否则为1。对于$half_i$有两种维护的方式：

- 在fail树上倍增。i在fail树上的祖先对应的字符串长度必然是递减的，可以倍增到长度最长的一个祖先使得其长度不超过i的一半。时间复杂度为$O(|s|\log|s|)$。
- 令j为i在回文树上的父亲。$half_i$必然是$half_j$的某个回文后缀的儿子。直接沿着 $half_j$ 的 $fail$链找到第一个合法的节点。可以用与证明基础插入算法复杂度相同的方法证明这种算法的总复杂度是$O(|s|)$。

5.3 Virus synthesis [54]

给定一个由大写字符A,G,T,C组成的长度为n的字符串s，一开始有一个空串t，每一步你可以对t进行以下操作中的一种：

- 在t的开头或结尾加入一个字符
- 将t的翻转接在t的开头或末尾

问最少需要多少步才能使得t变成s。
$n \leq 10^5$

[54]Central Europe Regional Contest 2014, "Virus synthesis"

注意到第二种操作后的字符串必然是一个回文串，考虑枚举s的一个回文子串$s[i..j]$，求出构造出$s[i..j]$的最小代价，用其加上$i-1+n-j$来更新答案即可。建出s的回文树，现在要求出构造出每个节点的最小代价。从短到长地计算每个回文串的代价。对于一个回文串t，首先考虑第一种操作，可以从t的最长回文前（后）缀扩展出t，或从t的父亲扩展出t，两种方案的代价都可直接计算。考虑通过第二种操作得到t，首先t的长度必须为偶数，令$t=bb^r$，求出b的最长回文前（后）缀c，那么t的代价就是c的代价加上c扩展成b的代价再加1。直接套用5.2中的算法，求出$half_t$后即可得到c。

总时间复杂度为$O(n)$。

5.4 基因[55]

给定一个长度为n的只包含小写字母的字符串s，有q组询问l,r，询问$s[l..r]$中本质不同的回文串数量。强制在线。

$n,q\le 10^5$

令$L=\sqrt{n}$，每L个位置设置一个关键点，对于每次询问，假如$r-l\le L$，可以直接暴力维护回文树，否则必然会经过一个关键点，维护每个关键点到每个右端点的回文树，每次相当于在回文树前插入字符，通过一些全局记忆化的思想可以将$O((n+q)\sqrt{n})$次插入中的$O(\Sigma)$的复杂度去掉，总的复杂度为$O((n+q)\sqrt{n}+n\Sigma)$。更具体的实现细节可以参考5

5.5 AlphaDog（原创）

对于一个串s的特殊值$F(s)$ 定义为

$$F_s=\sum_{1\le x\le y\le |s|}\mathrm{LCP}(x,y)$$

LCP即Longest Common Palindrome的缩写，也就是说$\mathrm{LCP}(x,y)$表示最长的字符串t 的长度，其中t要满足三个条件：

- t是回文的

[55]2017中国国家队候选队员互测四，《基因》

- 存在一个$i \le x$，满足$s[i..x] = t$。
- 存在一个$j \le y$，满足$s[j..y] = t$

一开始有一个空串s。接下来进行q次插入，每次在串的末尾加入一个新的字符。每次加入后，询问当前串的F值。强制在线。

$q \le 10^5$

首先考虑如何计算LCP(x,y)，令a为$s[1..x]$的最长回文后缀，b为$s[1..y]$的最长回文后缀，那么LCP(x,y)就等于a,b在$fail$树上的LCA的长度。每次插入一个新的字符，需要求出这个字符带来的贡献，相当于每次可能会在$fail$树上接入一个新的节点，并查询某点与其他所有点的LCA的长度总和。这个问题也有两种做法。

- 在$fail$树中令点i到$fail_i$的距离为$len_i - len_{fail_i}$。考虑怎么计算两个点的距离，$dis(x,y) = deep(x) + deep(y) - 2 * deep(lca(x,y))$，有$deep(lca(x,y)) = \frac{deep(x)+deep(y)-dis(x,y)}{2}$，现在给定$x$，要求$\sum_y deep(lca(x,y))$，只需要求出$\sum_y dis(x,y)$。动态在树上加点并求出某点到其他所有点距离是有经典的动态点分治做法的。大体思路就是维护一棵树的点分治结构，在插入时利用替罪羊树的思想重构这棵树的点分治结构，具体实现细节可以参考[6]，总时间复杂度为$O(q \log^2 q)$。
- 对于一个$lca(x,y)$，其做出的贡献为其长度，相当于它到根上每条边的距离的和，可以将贡献转化为每条边的贡献之和。对于一条边，假设他是i到$fail_i$的边，它能做出贡献当且仅当x与y 都在i的子树中。对于每条边令$size$表示其子树内的点数，维护每条边的$size \times len$，其中len为它的长度。那么加入一个点x，相当于查询x 到根所有边的$size \times len$的和，并且对所有边的$size \times len$都加上len，这是可以用一个Link-Cut Tree维护的。所以总的时间复杂度就是$O(q \log q)$。

6 总结

回文树是专门用于处理一类回文串问题的数据结构。它利用了回文串本身具有的优美性质，与树形结构相结合，从而可以轻松地解决许多普通算法所不能及的问题。但其本身需要较大的空间开销，并且代码实现复杂度也较高；同时，算法本身侧重于处理回文后缀，所以也不适合处理所有的回文串问题（Manacher算法对于部分问题就足够适用了）。这就好比后缀数组与后缀自动机各有自己的优势，回文树与Manacher算法在处理回文问题时也各有侧重。

回文树算法仍有很大的挖掘空间：在字符串中插入或删除字符时，是否能维护对应的回文树？对于一棵字符串树、一个DAG，是否能构建其对应的回文树？这几个方面仍有很大的研究价值。希望这篇论文能抛砖引玉，给予大家在回文串问题上更大的启迪。

7 鸣谢

感谢中国计算机学会提供学习和交流的平台。
感谢国家集训队教练张瑞喆、余林韵的指导与帮助。
感谢中山纪念中学的宋新波老师，熊超老师多年来给予的关心和指导。
感谢清华大学杨乐学长验稿。
感谢常州高级中学沈睿，雅礼中学毛啸、欧阳思琦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。
感谢我的父母一直以来无微不至的关心与照顾。

参考文献

[1] Aditya Shah, ”TREEPAL’s Editorial”, https://discuss.codechef.com/questions/85636/treepal-editorial

[2] Mikhail Rubinchik, Arseny M. Shur, ”EERTREE: An Efficient Data Structure for Processing Palindromes in Strings”, arXiv:1506.04862v2 [cs.DS] 17 Aug 2015

[3] Victor Wonder, 《回文树的构建及其应用》

[4] 徐毅, 《浅谈回文子串问题》, 2014年国家集训队论文

[5] 翁文涛, 《基因》解题报告

[6] WC2014, 《紫荆花之恋》, http://uoj.ac/problem/55

《黑白树》命题报告

福州三中 闫书弈

1 试题

1.1 题目描述

给定一棵n个点的树，每个点有50%的概率为黑点，50%的概率为白点。

给定正整数m，等概率随机选出一个边集(即每条边有50%的概率在边集中)，记边集大小为x。若对于边集中的每条边，都满足这条边两侧的黑点个数不同(边集为空集时也视为都满足)，则得分为m^x，否则得分为0。

你需要求出期望得分。

有多组数据。

1.2 输入格式

第一行一个正整数t表示数据组数。

每组数据第一行两个正整数n，m。接下来$n-1$行每行两个正整数u，v表示一条边。

1.3 输出格式

对于每组数据，输出一行一个整数表示期望得分$\times 2^{2n-1}$对998244353取模的结果。

1.4 样例输入

```
2
3 5
1 2
2 3
10 23333333
3 1
4 2
6 7
```

8 6
2 5
3 6
10 1
9 7
1 2

1.5 样例输出

158
167850015

1.6 数据规模

对于10%的数据，$n \leq 10$。
对于20%的数据，$n \leq 20$。
对于30%的数据，$n \leq 50$。
对于40%的数据，$n \leq 200$。
对于50%的数据，$n \leq 3000$。
对于60%的数据，$n \leq 10000$。
对于70%的数据，$n \leq 15000$。
对于100%的数据，$t \leq 5$，$2 \leq n \leq 20000$，$\sum n \leq 70000$，$1 \leq m < 998244353$，$1 \leq u, v \leq n$。

1.7 时空限制

时间限制：3s
空间限制：512M

2 算法

2.1 算法一

暴力枚举边集和点集，判断是否合法并统计贡献。

时间复杂度$O(2^{2n} * n)$，预计得分10分。

2.2 算法二

枚举点集，判断有哪些边是满足条件的。

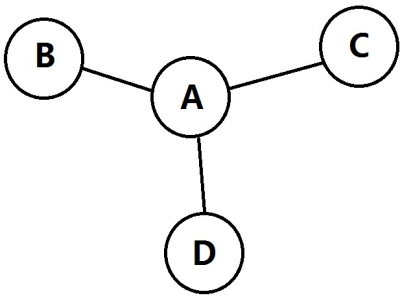
设有x条边满足条件，由于每条边可以选也可以不选，因此贡献为$(m+1)^x$。

时间复杂度$O(2^n * n)$，预计得分20分。

2.3 算法三

确定点集后，我们发现，在一般情况下(即所有点不全为白点)，不满足条件(即两侧黑点个数相同)的边集要么是空集，要么是一条链。

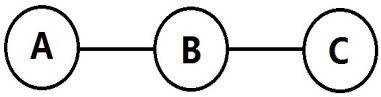
这是因为如果不满足条件的边集不是一条链的子集，那么边集中一定存在形如以下样子的三条边：



我们设f(X)表示X中黑点的个数，那么我们有f(B)=f(A)+f(C)+f(D)，f(C)=f(A)+f(B)+f(D)，f(D)=f(A)+f(B)+f(C)。相加得到3f(A)+f(B)+f(C)+f(D)=0。

由于$f(X) \geq 0$，不难得出f(A)=f(B)=f(C)=f(D)=0，即所有点均为白点。

而如果有两条边都在边集中，形如这样：



那么我们有f(A)=f(B)+f(C)，f(C)=f(A)+f(B)，相加得到2f(B)=0，即B中的点均为白点，因此以这两条边为两端的链上的所有边均在边集中。

所有点均为白点的情况在最后将贡献加到答案上即可，下面我们考虑一般情况。

设一条链的两端分别为a和b，a不包含链上的边的一侧子树的点数为sa，b不包含链上的边的一侧子树的点数为sb，那么这条链在不满足条件的边集中当且仅当这两个子树中的黑点个数相同。

考虑从这sa+sb中选出sa个点，在a的子树中没有选出的点以及在b的子树中选出的点为黑点，其余为白点，这恰好与两个子树中黑点个数相同的方案一一对应。因此这条链在不满足条件的边集中的点权方案数为C_{sa+sb}^{sa}。

再扣除掉包含这条链的链的方案即可求出这条链恰好为不满足条件的边集的方案。

时间复杂度$O(n^4)$，预计得分30分。

2.4 算法四

记t为m+1的逆元，那么算法二中的贡献为$(m+1)^{n-1} * t^{n-1-x}$。将$(m+1)^{n-1}$提出来最后乘上，那么贡献为t^{n-1-x}。

n-1-x是不满足条件的边数，类比算法二中的推理过程，这等价于随机一个边集(设大小为y)，若每条边均不满足条件则贡献为$(t-1)^y$

前面已经证明过这样的边集是链，枚举链的两端，同样设子树的点数为sa和sb，那么链上其它边可以选也可以不选，贡献为$(t-1)^2 t^{z-2} C_{sa+sb}^{sa}$，其中z为链上的边数，预处理lca即可$O(1)$求出。

当m=998244352时m+1不存在逆元，但是由于贡献为$(m+1)^x$，不难发现只有x=0时有贡献，此时要么是全为白点，要么树为一条链且两端为黑点其余点为白点，特判即可。

时间复杂度$O(n^2)$，预计得分50分。

2.5 算法五

我们发现算法四中的贡献式子$(t-1)^2 t^{z-2} C_{sa+sb}^{sa}$可以用树分治+FFT进行优化。

我们每次考虑过重心的链。对于每个点，将$\frac{(t-1)t^{k-1}}{sa!}$加到$f[sa]$中(其中k表示这个点到重心的路径上的边数)，然后将f与自己卷积一次，最后统计$\sum f[i] * i!$。再用同样的方法扣去链的两端在重心的同一个子树里的情况，就可以统计出过重心的链的贡献。注意，当边集是空集或一条边时需要特判掉单独计算。

接着我们发现这里存在一个问题：无论树分治到多小，由于sa和sb大小是$O(n)$的，因此f的大小也是$O(n)$的。那么卷积的复杂度始终是$O(nlogn)$的。

为了解决这个问题，我们设定一个阈值K，当需要计算的点数不超过K时，直接平方计算所有情况，然后不用继续分治下去，否则使用FFT优化卷积。

不难发现，树分治到的每个大小不小于K的连通块只会使用不超过2次FFT(1次是作为重心的整棵树累加，1次是作为重心的子树扣掉)。下面我们用归纳法证明，当$n \geq K$时，树分治到的大小不小于K的连通块的数量(记为$f(n)$)不超过$2\lfloor \frac{n}{K} \rfloor - 1$：

当$0 < n < K$时，显然有$f(n) = 0 = 2\lfloor \frac{n}{K} \rfloor$

当$K \leq n < 2K$时，显然有$f(n) = 1 = 2\lfloor \frac{n}{K} \rfloor - 1$

假设$K \leq n < iK$时都有$f(n) \leq 2\lfloor \frac{n}{K} \rfloor - 1$。下面我们考虑$iK \leq n < (i+1)K$的n。由于树分治选取的是重心，那么$f(n) = 1 + \sum f(x_i)$，其中$\sum x_i = n - 1$，$x_i \leq \frac{n}{2} < iK$。如果

满足$x_i \geq K$的x_i数量不小于1，那么$f(n) = 1 + \sum f(x_i) \leq 1 + (\sum 2\lfloor \frac{x_i}{K} \rfloor) - 2 \leq 2\lfloor \frac{n}{K} \rfloor - 1$，否则$f(n) = 1 + \sum f(x_i) \leq 1 + 2\lfloor \frac{iK-1}{K} \rfloor - 1 \leq 2i - 2 \leq 2\lfloor \frac{n}{K} \rfloor - 1$。

因此对于$n \geq K$，有$f(n) \leq 2\lfloor \frac{n}{K} \rfloor - 1$，那么我们使用FFT的次数是$O(\frac{n}{K})$的。

而平方计算的总复杂度显然是O(nK)的。

时间复杂度为$O(\frac{n^2 logn}{K} + nK)$，当K取$O(\sqrt{nlogn})$时为$O(n\sqrt{nlogn})$，预计得分100分。

3 总结

本题题意简洁但解法巧妙，既考察了选手对题目性质的挖掘与推导，即发现不满足条件的边集通常情况下是一条链，并将“满足条件”一步步推至“不满足条件”，又考察了选手使用算法优化公式计算的能力，是一道思维含量丰富的题目。

4 感谢

感谢父母对我的关心和照顾。

感谢黄志刚老师多年来给予的关怀和指导。

感谢张瑞喆教练、余林韵教练的指导。

感谢钟子谦同学提供命题灵感。

感谢中国计算机学会提供学习和交流的机会。

参考文献

[1] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[2] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

《正多边形》命题报告

成都市第七中学 杨景钦

摘 要

本文将介绍作者在校内训练时命制的一道以传统计算几何为背景的题目《正多边形》。题目深度挖掘了常见的“整点”（即格点）的特殊性质，将题目的特殊限制隐藏在这个大家习以为常的限制中。本题同时综合了动态规划，Hash算法以及数学上的归纳和证明，是一道涉及算法广，思维难度较高的好题。

除此之外，题目的部分分设置较多且合理，数据强度高，区分度高，无论是只写了暴力的选手，还是稍微进行过思考的选手，以及深度研究找到了所有性质的选手，都能获得适度的部分分。

本题同时提供了一种命题思路：深度挖掘常见的条件的性质，并将重要条件隐藏在这些性质中，使之具有更高的思维难度，需要选手有着更加敏锐的洞察力来解题。希望能对大家有所帮助，引发大家对于常见限制的一些思考。

1 试题

1.1 试题大意

二维平面上有N个整点，第i个点的坐标记为(X_i, Y_i)，现在需要你在N个整点中选出一些点，假设选出的点分别是$V_1, V_2, V_3, ..., V_k$，我们需要这些点满足以下条件：

- $3 \le k \le N$。
- 对于任意$1 \le i \le k$，不存在一个顶点均为N个给出的整点之一的三角形，使得V_i严格在三角形内（在顶点上以及边上不算严格在三角形内）。
- 存在一个正k边形，使得它的顶点均为选出的点$(V_1, V_2, V_3....V_k)$。

  求选点的不同的合法方案数，我们认为两个方案不同，当且仅当存在至少一个点在一个方案中被选中，在另一个方案中未被选中。

1.2 数据范围

对于5%的数据，$N \leq 7$。

对于25%的数据，$N \leq 20$。

对于45%的数据，$N \leq 80$。

对于70%的数据，$N \leq 200$。

对于100%的数据，$3 \leq N \leq 50000, 1 \leq X_i, Y_i \leq 50000$，不存在三个点在一条直线上。

2 算法

2.1 算法一

对于$N \leq 7$的数据，我们可以枚举每一个点是否被选入，再枚举一个点V_i，以及一个三角形对第二条限制进行检验，再枚举这些点与正多边形顶点的对应关系，对第三条性质进行检验，复杂度$O(2^N \times (N^4 + N!))$。

期望得分5分。

2.2 算法二

观察发现，第三条限制并不需要枚举选出的点，与正多边形顶点的对应关系去检验，因为一个正多边形一定是一个凸多边形，在枚举了选出的点之后，我们可以使用简单的求凸包算法，便可求出这个以这些点为顶点的唯一凸多边形（可能没有），之后检验这个多边形是否是正多边形即可。

而对于第二条限制，我们可以发现，一个点能否被选中，与其他所选的点没有关系，只与这个点本身有关。

我们可以在枚举选出的点之前，预处理每一个点能否被选中，复杂度$O(N^4)$。

故总复杂度为$O(N^4 + 2^N \times NlogN)$，其中$NlogN$是求经典算法求凸包的复杂度。

我们发现这个求凸包的算法，复杂度瓶颈在于排序，我们可以预先对点排序，这样在枚举完之后就不用排序，直接调用预处理的顺序即可，复杂度可以优化到$O(N^4 + 2^N \times N)$

期望得分25分

2.3 算法三

2.3.1 对于第二条限制的解决办法

根据算法二中的观察，我们需要预处理出每一个点是否能作为被选的点。通过进一步的观察，我们发现，将N个点的凸包求出，只有在凸包上的点是可选点，凸包内的点均为

不可选点，下面我们来证明这个结论：

- 所选点在凸包内时，可以把凸包上的点所构成的凸多边形三角剖分，因为该点一定在凸包所构成的凸多边形内部（否则与凸包定义矛盾），我们可以找到三角剖分中覆盖这个点的三角形，由于不存在三点共线的情况，因此点不会落在三角形的边上，而这个三角形的顶点均为凸包上的点，也一定是给出的N个点之一，所以凸包内的点均为不可选点。

- 所选点在凸包上时，由于没有三点共线的情况，一定不存在包含该点的顶点均为给出点的三角形，否则与凸包的定义矛盾，故所有在凸包上的点都是可选点。

2.3.2 对于限制三的处理办法

我们之前的思路一直是按照题意所说的，枚举一个点集，判断其是否能构成正多边形。事实上，我们要求的就是有多少个顶点均为给出的点凸包上的点的正多边形。

对于一个正多边形，假设知道了相邻两个点，以及该多边形的点数，那么我们就可以知道多边形每个点的坐标（有两种可能）。

我们可以枚举凸包上两个点作为答案多边形的相邻顶点，再枚举多边形的边数，然后暴力检查每个点是否存在。

对于一个i个顶点的正多边形，他会被统计进答案i次，于是我们对于边数不同的多边形分别统计即可，复杂度$O(N^4 \times T_{check})$。其中T_{check}是检验一个点是否在给出的点内的时间复杂度。

使用$Hash$表来检验一个点是否给出，可以做到预处理$O(N)$，单次询问期望下$O(1)$的复杂度，于是总复杂度期望下$O(N^4)$，期望得分45分

2.4 算法四

在算法三的基础上，我们还能进行优化。因为算法四进行了很多冗余的检验。

首先我们可以把每个正多边形的每条边，按照逆时针顺序变成有向边。

我们可以令$Vis_{i,j,k}$表示一条从凸包上的第i个点到凸包上第j个点的有向边，作为正k边形的一条边的情况，是否已经被检验过。

在枚举的时候，假如枚举到凸包上第i个点和第j个点，是否能作为正k边形的相邻顶点，如果$Vis_{i,j,k} = true$，即已被计算过，则跳过此次检验，否则就暴力检验，并把访问过的边的Vis数组置为true。

显然$Vis_{i,j,k}$数组的每一个状态用于计算一次，每次计算的复杂度同样是检验一个点是否是给出的点的复杂度，而总状态数为$O(N^3)$，因此该算法复杂度为期望下$O(N^3)$，期望得分70分。

2.5 算法五

2.5.1 对于正多边形边数k的观察

通过显然的观察我们可以发现，对于某些k，一定不存在顶点均为整点的正k边形，接下来将给出，在顶点均为整点的前提下，只存在正四边形的证明。

首先我们需要知道著名的皮克定理[56]

定理 2.1. *对于一个顶点为整点的简单多边形，其面积*

$$S = a + \frac{b}{2} - 1$$

其中a表示多边形内部的整点数，b表示多边形边界上的整点数。

通过皮克定理我们可以知道：

定理 2.2. *一个顶点均为整点的正多边形，面积一定是$\frac{1}{2}$的正整数倍。*

2.5.2 对于$k = 3$的证明

当$k = 3$时，正三角形的面积S 有

$$S = \frac{\sqrt{3}a^2}{4}$$

。

其中a是正三角形边长，因为顶点均为整点，所以a^2一定可以写成$x^2 + y^2$的形式，其中$x, y \in \mathbb{Z}$。

此时S一定不是有理数，故肯定不符合是$\frac{1}{2}$的整数倍，由此可得不存在一个顶点均为整点的正三边形。

2.5.3 对于$k = 5$的证明

假设存在一个顶点均为整点的正五边形，由定理2.2，一定存在一个面积最小的正五边形。

[56]皮克定理: http://baike.baidu.com/link?url=YPZMMmQnYxImXzLGXu2CGC3QJoAFrs-Qvy7KyfwPzco_QsiQ88zfXk30nT3rgCYmg5TPu25m5JiDXKQIodxUYsh_IoByRwxJEFuFkn9q

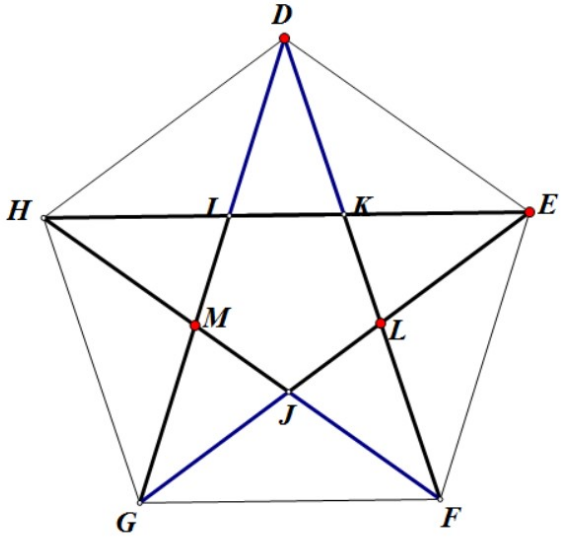


图 12: 图1

如上图所示，假设面积最小的正五边形为$DEFGH$，作平行四边形$DEJH$，$DHGL$,$HGFK$，$EFGI$,$DEFM$。

由平行四边形可得，I,K,L,J,M均为整点，且五边形$IKLJM$各内角度数全部相等，故五边形$IKLJM$也是一个顶点均为整点的正五边形，显然正五边形$IKLJM$面积比正五边形$DEFGH$小，与正五边形$DEFGH$是面积最小的定义矛盾，故假设不成立。

所以不存在顶点均为整点的正五边形。

2.5.4 对于$k = 6$的证明

当$k = 6$时，我们可以发现，如果存在一个顶点均为整点的正六边形，我们可以将该正六边形的顶点从任意点顺序开始按照逆时针标号，则标号为奇数的三个点构成了一个正三角形，且顶点均为整点，与之前证明的不存在正三角形矛盾，故不存在顶点均为整点的正六边形。

2.5.5 对于$k \geq 7$的证明

类似$k = 5$的情况，我们使用无穷递降法。由定理2，假设存在一个顶点均为整点的正k边形，则一定存在一个面积最小的正k边形。为了方便作图，在此以正八边形为例。

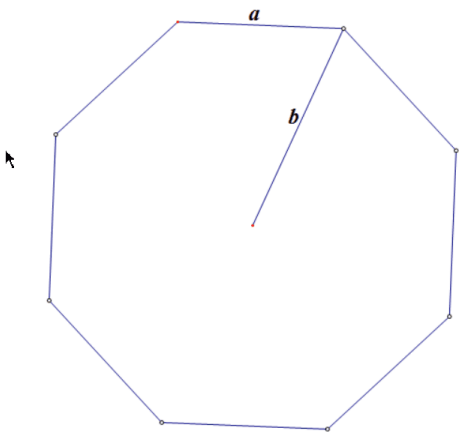


图 13: 图2

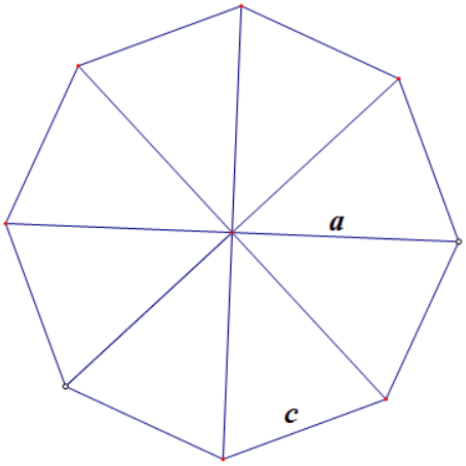


图 14: 图3

如图所示，假设图2中是一个面积最小的顶点均为整点的正八边形。记边长为a，由于正八边形的内角和大于120度，所以假如我们令正八边形的中心到顶点的距离为b，有$b > a$。

考虑将图2中的正八边形的各条边平移，使所有边恰有一个公共端点，显然此时我们得到了一个边长为c的顶点均为整点的正八边形（如图3），而a是新的正八边形中心到顶点的距离，又有$c < a$，所以新的正八边形比原来的正八边形面积更小，因此矛盾。故不存在顶点均为整点的正八边形。其余$k \geq 7$的情况同理。

故当$k \geq 7$时，不存在顶点均为整点的正k边形。

2.5.6 尝试改进算法

在算法四的基础上，我们在枚举的时候，不需要枚举k，因为k只能等于4，复杂度可以降为期望下$O(N^2)$。

看起来已经是一个无法优化的算法，事实上在目前看来确实是的，但是注意到本题的实际复杂度并不是$O(N^2)$，而是$O(N_B^2)$，其中N_B 指给出N个点的凸包上的点数。

接下来，我们将证明N_B 是$R^{\frac{2}{3}}$级别，其中R是点的坐标范围，表示每个点的坐标都在$[0, R]$内。

首先我们只考虑右下凸包，即凸包上斜率大于0且严格递增的部分（因为不存在三点共线，所以不会出现斜率相同的情况），设右下凸包上的点数上界为T，那么整个凸包上的点易得上界为$4 \times T$。

将右下凸包上的点按照横坐标排序后，设第i个点坐标为(x_i, y_i)。

考虑右下凸包上相邻的两个点$V_i(x_i, y_i), V_{i+1}(x_{i+1}, y_{i+1})$，由于$V_i, V_{i+1}$均是整点，故可以设过$V_i, V_{i+1}$的直线斜率为$\frac{p}{q}$，其中$p$与$q$互质，由于是右下凸包，可以得到 $x_{i+1} + y_{i+1} - (x_i + y_i) \geq p + q$

考虑将下凸包上的所有斜率求出，设对于$1 \leq i < T$ 第i段直线的斜率为$\frac{p_i}{q_i}$，则有

$$x_T + y_T - (x_1 + y_1) \geq \sum_{i=1}^{T-1} p_i + q_i$$

又有

$$x_T + y_T - (x_1 + y_1) \leq 2 \times R$$

于是我们只需证明，对于分子分母和前T小的分子分母均为正整数的既约分数，其分子分母和超过$2 \times R$即可。

构造一个下标从1开始的新的不降正整数数列P，使得每个正整数i都出现了恰好i次，更详细的，对于$i \geq 1, \frac{i \times (i-1)}{2} < j \leq \frac{i \times (i+1)}{2}$，有$P_j = i$。[57]

记$S(i)$为分子分母和前i小的既约分数的分子分母和，SP_i为P序列的前i项的和。由于分子分母和为i且分子分母均为正整数的既约分数显然不超过i个，故我们有$S_i \geq SP_i$，又因为我们要证明的是$S_T \geq 2 \times R$，于是我们只需证明 $SP_T \geq 2 \times R$即可。

设$t = \lfloor \sqrt{T} \rfloor$，有

$$SP_T \geq SP_{\frac{t \times (t+1)}{2}}$$

又有

$$SP_{\frac{t \times (t+1)}{2}} = \sum_{i=1}^{t} i^2$$

[57] P序列即为$1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, ...$

由平方和公式，

$$\sum_{i=1}^{t} i^2 = \frac{t \times (t+1) \times (2t+1)}{6}$$

将$T = 4 \times R^{\frac{2}{3}}$带入得到，

$$SP_T \geq \frac{16 \times R}{6} \geq 2 \times R$$

故命题得证，即若干坐标范围在$[0, R]$的整点，不存在三点共线情况时，其凸包上的点数为$R^{\frac{2}{3}}$级别。

3 数据构造

3.1 随机数据

3.1.1 随机方法1

直接在坐标范围内随机若干个整点，去重后输出，答案在大多数情况下为0。

由于有可能存在三点共线，因此输出的时候枚举每一条直线，把中间的点全部去掉。

3.1.2 随机方法2

随机一个坐标范围内的正方形，在正方形内随机若干整点，然后用随机方法1的方法去掉三点共线情况。此时答案为1，即最外层的正方形。

3.2 构造数据

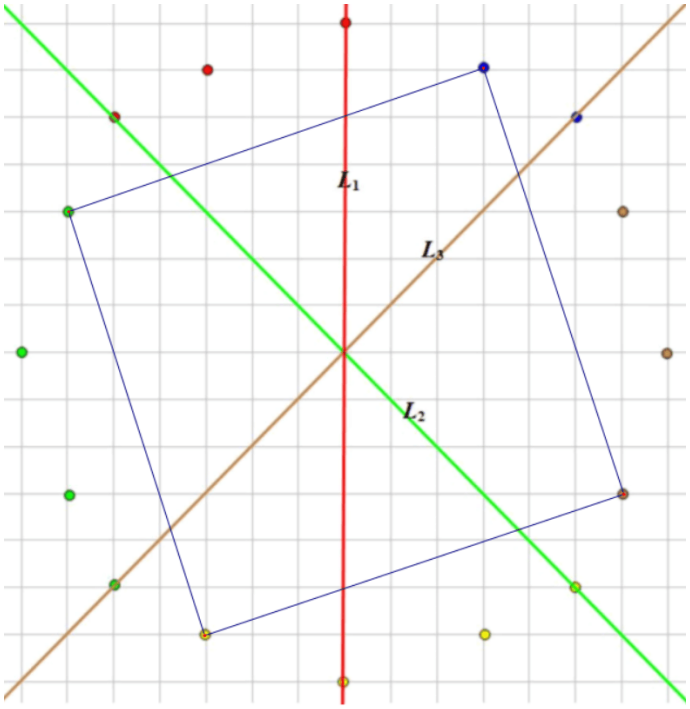


图 15: 图4

如图4所示

最左红点为任意一个整点(x_0, y_0)开始，随机若干个小于$\frac{1}{2}$的既约分数，并保证每个既约分数的分子分母均不超过$R^{\frac{1}{3}}$，将这些既约分数按照斜率从大到小排序，设第i个既约分数为$\frac{p_i}{q_i}$，那么对于第i个点的横纵坐标，有

$$x_i = x_{i-1} + q_i, y_i = y_{i-1} + p_i$$

我们可以依次得到所有红点，设总共有n个既约分数，则总共有$n+1$个红点，此时得到一个八分之一凸包。

将所有红点以$x = y_n$直线(图中L_1)为对称轴做轴对称，得到所有蓝点，此时红点与蓝点形成的凸包为整个凸包的四分之一凸包。

将红点与蓝点看做一个整体，以过(x_0, y_0)的斜率为-1的直线(图中L_2)为对称轴，做轴对称，得到所有绿点，此时所有点构成了二分之一凸包。

类似的，将所有点以直线L_3为对称轴，做轴对称，得到的所有点构成了一个完整的凸包。

根据轴对称的特性，我们任取一个红点或蓝点，都能找到一个以该点为顶点的正方形（图中给出了一个例子）。

在凸包内随机加入扰动点，并去掉三点共线的情况，再在凸包上随机去掉一些点即可。

3.3 测试数据的构造

对于每个部分分，有且仅有一个点为随机数据，随机方式为两种随机方法中随机一个，其他所有数据均为构造数据。

4 命题契机及总结

NOI2016四川省队选拔赛上，作者做到了一道题目，问题是在给出的10^6个点的凸包上，对于每个点进行$O(1)$的解方程，而所有给出的点均为坐标在10^8以内的整点。赛后不少同学表示，在凸包上使用每个点$O(logN)$的三分算法，也能通过此题，得知消息之后我试图构造一个所有点均在凸包上的数据来卡掉复杂度错误的三分算法，然而却怎么都构造不出来。在查阅相关文献之后，我发现整点意义下，凸包上的点数是有一个十分优美的上界的，这启示我去关注整点的其他性质，最终命制了这样一道题目。

出题人在出平面坐标系上的题目时，往往会让所有点都拥有“整点”这一约束，其中大多只是为了方便造数据，很少有出题人利用“整点”的特殊性去出题。本题从“整点”入手，展现了凸包和正多边形在“整点”下的特殊性质，同时将平面上的几何问题，与动态规划，数学上的证明以及放缩，*hash*方法，有机结合在一起，是一道涉及算法多，思维难度中等的题目。

本题提供了一种命题思路：将一些经常出现的习以为常的限制，进行深度挖掘之后，可以把性质隐藏在大家一般会忽略的常规限制，从而提高思维难度。希望能引起大家的思考，拓展出更多的同类题目。

5 感谢

感谢中国计算机学会提供学习和交流的平台。

感谢张君亮老师，林鸿老师以及父母多年来的关心和指导。

感谢骆可强学长，王迪学长，张瑞喆学长，钟皓曦学长的指导和帮助。

感谢杨家齐同学，洪华敦同学与我交流题目并为本文审稿。

感谢成都七中的同学们为本文提出修改意见。

感谢OI路上支持和帮助我的所有人。

参考文献

[1] 杨弋，《Hash在信息学竞赛中的一类应用》

[2] 田廷彦，《数学奥林匹克命题人讲座——组合几何》，上海科技教育出版社

[3] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[4] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

浅谈决策单调性动态规划的线性解法

绍兴市第一中学 冯哲

摘 要

决策单调性是动态规划中比较经典的题型之一。作者在对常见解法进行归纳的基础上，介绍了一种能够在线性时间内解决该类问题的一般算法，并将其与非线性算法比较，取得了不错的效果。

1 引言

决策单调性是动态规划中一类比较经典，有趣的问题。随着人类思维的不断进步与发展，越来越多的问题，如单源最短路，最小生成树等模型都有了各自在渐进意义下的线性解法。在思考《数据分块鸡》这一问题的过程中，我对这类模型的解法产生了极大的兴趣，并开始寻找是否存在优秀的线性解法。

本文在第二节中对决策单调性所满足的性质以及需要的条件作了简单的介绍。在第三、四节中，本文介绍了两种满足特殊性质的决策单调性模型的解法。其中第四节中的线性解法为之后更普遍的算法做了铺垫。第五节中，本文介绍了一种不基于高级数据结构且实现较为简单的线性做法，希望能够对广大读者的解题和出题带来些许的帮助。

2 决策单调性的定义

对于标准的$1D/1D$方程

$$F_j = \min_{0 \le i < j} D_i + w_{i,j} \quad j \in [1, N]$$

其中D_i是关于F_i的任意函数。

若权函数w满足$\forall i, j, w_{i,j} + w_{i+1,j+1} \le w_{i,j+1} + w_{i+1,j}$，则称其满足凸完全单调性。

易证明$k > 0$时，有$w_{i,j+k} - w_{i,j}$随i单调不降，$w_{i+k,j} - w_{i,j}$随j单调不降。

那么对于$\forall 1 \le a \le b \le c \le d \le N, w_{a,c} + w_{b,d} \le w_{a,d} + w_{b,c}$，称该方程具有决策单调性。

当$w_{a,c} \ge w_{b,c}$时，由上式可以得到$w_{a,d} \ge w_{b,d}$，即对于$\forall i < j$的最优决策点k_i, k_j,恒有$k_i \le k_j$。

发现决策单调性的方法是证明四边形不等式，而在实际应用中，直接观察或者打表也不失为好的选择。

3 利用斜率优化

斜率优化是决策单调性问题一种衍生算法，一类特殊的决策单调性问题可以利用斜率优化在线性时间内得到解决。

将一个决策j看作平面上的一个点(A_j, B_j)。对于i来说，$\forall k < j\ f_j + w_{j,i} \le f_k + w_{k,i}$，当且仅当$\frac{A_j - A_k}{B_j - B_k} \le C_i$，即两点之间斜率与$C_i$的比较，其中$A_j, B_j$均在计算$f_j$后已知且$B_i$一般递增，$C_i$已知。

实现这一算法的一般方法为维护所有决策点构成的凸包，询问时在凸包上二分寻找第一个斜率不超过C_i的位置，时间复杂度为$O(N \log N)$。而当C_i满足数值递增时，可以利用单调队列直接维护凸包，询问时只需在线性时间内弹出队首不满足条件的元素即可。

例 1. (*Atcoder Regular Contest* 66 *D*)*Contest with Drinks Hard*

在一场比赛中一共有N道题，其中解决第i道题需要花费T_i秒。你可以在所有题中挑选任意一些题并解决它们。定义一种解决方案所得的分数为满足$\forall L \le i \le R$ 第i道题被解决且$1 \le L \le R \le N$的L, R对数减去解决问题所需要的时间和。

现给出Q组询问，第i组询问需要你求出将第X_i道题的解决时间变为Y_i秒后，分数最高方案的得分(询问后序列还原)。

$1 \le N, Q \le 300000, 1 \le T_i \le 10^9$

首先考虑无修改时的求解方法。设$s_i = \sum_{j=1}^{i} T_j, w_{i,j} = \frac{(j-i)*(j-i-1)}{2} - s_j + s_i$，可以列出方程

$$f_j = \max(f_{j-1}, \max_{i=0}^{j-1} f_i + w_{i,j})$$

f_{j-1}项可以直接计算，考虑四边形不等式，有$w_{a,c} + w_{b,d} - w_{a,d} - w_{b,c} = ac * bd - ad * bc \ge 0$，在该式中，若某个时刻 $f_k + w_{k,j} > f_i + w_{i,j}, k < i$,那么$k$将永远比$i$优。这并非常见的问题形式，但利用该性质我们仍然可以解决本题。

3.1 通用解法

根据决策单调性，可以对所有决策维护一个双端队列。加入一个新的决策时，求出该决策比队尾决策第一次优的时间，不停退队直到队尾决策有可能成为最优决策,再将新的决策加入队列。询问时弹出队首已经不优的元素。

在这一个过程中，我们总共需要比较$O(N)$对决策，比较决策时无法像斜率优化一样快速得出，而是需要二分时间来确定优劣，总共需要询问$O(N \log N)$次$w_{i,j}$的值。

在解决本问题时，我们只需要改为在询问时弹出队尾不优的元素即可。

3.2 斜率比较

若设$A_i = f_i + \frac{i^2}{2} + s_i$，对某一待决策点$j, \forall 0 \le k < i < j, f_i + w_{i,j} > f_k + w_{k,j}$当且仅当$\frac{s_i - s_k}{i-k} > j$。由于题目要求斜率大于某个值，可以用栈维护一个斜率递减的上凸壳，并在询问j的决策时二分查询第一个斜率不超过j的位置，时间复杂度$O(N \log N)$。

更进一步，维护一个指针，指向当前最优决策点的位置，当该决策被退栈时，将指针移至新的栈顶处；询问时将指针向前移动直到斜率不超过j。该指针的移动次数不会超过总元素的数量，时间复杂度$O(N)$。

接下来考虑修改的情况，修改一个点后，将方案按是否覆盖这个点分类。

对于所有不覆盖修改点的情况，可以前后各计算一次答案，通过前缀和后缀的答案得到。

对于覆盖修改点的方案，我们将所有的询问离线，并对序列进行分治。

当分治至$[L, R]$时，对每个超过mid的右端点利用斜率优化预处理存在一个以这个点为右端点且跨越mid的区间时的最大分数。做后缀和，对于一个处于$[mid + 1, R]$的修改点，可以在$O(1)$的时间内更新它的答案，将序列反转即可得到$[L, mid]$中的答案，时间复杂度为$O(N \log N)$。

通过斜率优化，我们可以将一些决策单调性问题在线性的时间内解决。然而，大多数决策单调性问题的决策并不能转化为平面上的斜率比较。

4 一个简单的例子

例 2. $(NAIPC2016) Jewel\ Thief$

有N个物品，每个物品有一个体积w_i和价值v_i，现在要求对$V \in [1, K]$，求出体积为V的背包能够装下的最大价值。

$1 \le N \le 1000000, 1 \le K \le 100000, 1 \le w_i \le 300, 1 \le v_i \le 10^9$

首先将所有物品按体积分类，选取同种体积的物品时，一定优先选择价值大的物品。

设$f_{i,j}$为使用前i种体积的物品，体积为j的最大价值。将模i同余的所有位置拿出来重新标号，则有

$$f_{i,j} = \max_{k=0}^{j} f_{i-1,k} + w_{k,j}$$

其中$w_{k,j}$为体积为i的物品中前$j-k$大的价值和。注意到$w_{k,j}$函数的值仅与其长度有关，且增量随长度增加递减。在四边形不等式中，两边的长度之和相同，那么较为平均的一边和较大，容易证明该函数也满足四边形不等式。

同时，该转移还满足一个特殊的性质：每个点在决策时可以不依赖之前的决策，并不需要按照顺序依次决策。

4.1 问题转化

定义 4.1. 令$A_{i,j}$描述矩阵的一个元素，A_i描述矩阵A的第i行，A^j描述矩阵A的第j列，$A[i_1,...,i_k:j_1,...,j_m](i_1<...<i_k,j_1<...<j_m)$描述矩阵$A$以$i_1,...,i_k$行以及$j_1,...,j_m$列构成的子矩阵。

令$pos(j)=\max\{i|\forall 0\le p\le K,A_{i,j}\ge A_{p,j}\}$。

在原问题中，仅仅考虑其中一层的转移，令$A_{i,j}=f_i+w_{i,j}$，当$i>j$时，$A_{i,j}=-\infty$。

则新的问题为：

已知一个$(K+1)*(K+1)$大小的矩阵，其中每个元素的值均可以在$O(1)$时间内查询。现要求对于$j\in[0,K]$，求出$A_{pos(j),j}$。

4.2 一种简单的分治法

在原问题中，有$\forall 0\le i<j,\le K\ pos(i)\le pos(j)$。

一种比较简单的思路是对列$[l,r]$进行分治，维护当前可能成为$pos()$的行$[x,y]$，令$mid=\frac{l+r}{2}$，暴力枚举所有可能的行求出$pos(mid)$，分治递归操作$[l,mid-1],[x,pos(mid)]$以及$[mid+1,r],[pos(mid),y]$直至$l=r$或$x=y$。

设$S(n,m)$为列数为n，行数为m时的时间复杂度，那么有

$$S(n,m)\le m+\max_{1\le j\le m}S(\lceil n/2\rceil-1,j)+S(\lfloor n/2\rfloor,j)$$

显然该分治的时间复杂度是$O(K\log K)$的，并且询问了$O(K\log K)$次矩阵中某个位置的值。

4.3 SMAWK算法[1]

在上述做法中，我们仅仅依赖了决策单调不降的性质，而在定义中，还有着一条重要的性质————四边形不等式。

定义 4.2. 若矩阵A满足$\forall 0\le i<j\le K\ pos(i)\le pos(j)$，则称$A$为单调矩阵。更进一步，若$A$的任意子矩阵$A[i_1,...,i_k:j_1,...,j_m]$均为单调矩阵，则称$A$为完全单调矩阵。

若$pos(j)\ne i$，则称元素$A_{i,j}$为无用元素。

引理 4.1. *A为完全单调矩阵，当且仅当$\forall A[i_1,i_2:j_1,j_2]$均为单调矩阵。*

证明：由定义2可知必要性显然。

使用归纳法证明充分性。对于$A[i_1,...,i_k:j_1,...,j_m]$，且$m\ge 3$,假设其所有子矩阵均为单调矩阵，那么由子矩阵$A[i_1,...,i_k:j_1,...j_m-1]$可知$pos(j_1)\le...\le pos(j_{m-1})$。同理由$A[i_1,...,i_k:j_2,...,j_m]$得$pos(j_2)\le...\le pos(j_m)$，得原矩阵也是单调矩阵，且结合假设得其为完全单调矩阵。而该假设在矩阵的行数或列数为1时成立，当行数≤ 3，列数为2时也成

立。当行数> 3，列数为2时，提出$k(j_1), k(j_2)$行及j_1, j_2列组成$2*2$的子矩阵，而所有$2*2$矩阵都是单调矩阵，证毕。

在之前的算法中，我们维护了可能成为当前所有列的最优答案的行区间$[x, y]$。有时，行区间的决策数量将比列的数量多很多，但大多数都是无用决策，那么我们是否能将行区间里的一些无用决策去掉呢?

4.3.1 一个新的问题

例 3. 给出一个$M*N(M>N)$的完全单调矩阵A，每个位置都可以快速询问。要求求出任意一个大小为N的行集合S，使得$\forall j \in [1,N], pos(j) \in S$。

在$SMAWK$算法中，我们使用$Reduce()$函数来解决这个问题。

算法 1 REDUCE

```
0:  procedure REDUCE(A)
1:  S ← 1, ..., M; p ← 1
2:  while |S| > N do
3:    if A_{S_p,p} > A_{S_{p+1},p} and p < N then
4:      p ← p + 1
5:    else
6:      if A_{S_p,p} > A_{S_{p+1},p} and p = N then
7:        Delete S_{p+1}.
8:      else
9:        Delete S_p; p ← p − 1
10:     end if
11:   end if
12: end while
13: return A[S : 1, ..., N]
```

那么为什么只要这样做就是对的呢?

在$Reduce$函数中，我们使用一个数组S来储存每列可能成为答案的行。初始时，我们假定S_p就是第p列对应取到极值的行，而$> N$的所有位置存储的都是备选决策。

接下来，我们需要顺次比较S_p与S_{p+1}，以确定S_p到底是否可能成为p在决策点$\geq S_p$时的最优决策。

引理 4.2. *设A为一个完全单调矩阵，若对$i_1 < i_2$，有$A_{i_1,j} > A_{i_2,j}$，则$\{A_{i_2,c} | 1 \leq c \leq j\}$都是无用元素；反之，若$A_{i_1,j} \leq A_{i_2,j}$, 则$\{A_{i_1,c} | j \leq c \leq N\}$都是无用元素。*

证明：当$A_{i_1,j} > A_{i_2,j}$时，有$\forall 1 \le k < j, pos(k) = pos(j) = i_1 \in A[i_1, i_2 : k, j]$；类似地，若$A_{i_1,j} \le A_{i_2,j}$，$\forall j < k \le N, pos(k) = pos(j) = i_2 \in A[i_1, i_2 : j, k]$，证毕。

若$A_{S_p,p} \le A_{S_{p+1},p}$，那么对于$S_p$来说，它在$p-1$处不是最优决策，在$p$处也不如$S_{p+1}$优，显然这个决策没有任何用。我们直接把这个决策删去，然后将后面的决策向左平移一位。由于p的决策更换了，我们无法确定S_{p-1}和当前的S_p的优劣性，所以还需要退回$p-1$重新比较。

否则，我们暂时认为S_p是p的决策，然后继续去判定下一列。若$p = N$，也就是说S_{N+1}在N处都不能变优，那么直接删掉这个决策就可以了。

注意我们求得的N个决策，并不是其对应列的真正决策行。我们所能够知道的，仅仅是列$pos(p) \le S_p$，也就是$A_{S_p,p} > \max\{A_{S_{p+1},p}, ..., A_{S_N,p}\}$。

复杂度证明

在分析时间复杂度时，我们考虑集合中元素的个数$|S|$以及指针p的位置。在一次比较中，可能有如下三种情况：

1.$|S|, p+1$,p的上限为N。2.$|S|-1, p$。3.$|S|-1, p-1$。

注意到p每减少1，必然伴随着$|S|$的减少。也就是说p的每一次抵消都可以视为花费2的代价使得$|S|-1$。

也就是$|S|$的减少至多需要花费$2*(M-N) \approx O(M)$,再加上p指针额外的$O(N)$的代价，总的复杂度就是$O(M)$的。

解决问题

将这个算法再放回之前的分治法，当行与列在同等程度上均分时，这个算法的复杂度仍为$O(N \log N)$，这启示我们要使用更高效的方法。

一种更好的解法是每次将所有奇数位的行取出来进行递归，计算出所有奇数位的决策后，再利用决策单调性的性质求出偶数位的决策。

算法 2 SMAWK

```
0: procedure SMAWK(A)
1: Reduce(A)
2: B = A[1, ..., N : 1, 3, ..., 2⌊n/2⌋ + 1]
3: SMAWK(B)
4: Using pos(2*i-1) and pos(2*i+1) find pos(2*i)
```

设$T(a,b)$为计算一个$a*b(a<b)$的矩阵的所有pos的时间复杂度，有

$$T(a,b) \le T(a/2,a) + O(a+b)$$

在每一层递归时，经过*Reduce*后行数与列数都是相同的。每层中花费的代价都是O(a+b),总的复杂度为$O(K) + O(\frac{K}{2}) + ... + O(1) = O(K)$.

5 更常见的问题

在之前的解法中，我们利用了问题中列决策互不影响的性质，而在大多数问题中往往需要对每一列进行依次决策。但之前的解法为我们提供了一个新的思路。

例 4. *UOJ* 285 数据分块鸡

你有一个下标为$[1,N)$的数组。要求将数组分块，设分解点为$a_0 = 1, a_1, ..., a_k = N$，其中第$i$块为$[a_i, a_{i+1})$。

定义一次询问j为询问$[l_j, r_j)$区间，而这次询问的代价如下：

若存在i使得$[l_j, r_j) \subseteq [a_i, a_{i+1})$，那么这次询问的代价是$(l_j - a_i) + (a_{i+1} - r_j)$。

否则，对于该区间覆盖的每一块，若$[a_i, a_{i+1}) \subseteq [l_j, r_j)$，则代价为1。除此之外的块的代价为两个区间的交集大小与该块大小减去交集大小中的较小值。

现给出Q组询问，要求给出一组分块方案，使得这Q组询问的代价和最小。

$N, Q \le 50000$

设f_i为在$[1,i)$中i为最后一个分界点的最小值，$w_{i,j}$为当存在块$[i,j)$，所有询问在本块中产生的贡献，即可得到与第二节中完全相同的式子。

在本题中，我们需要证明$\forall i < j, w_{i,j} + w_{i+1,j+1} \le w_{i+1,j} + w_{i,j+1}$.

证明：若$l_k \ge i+1, r_k \le j$或$l_k \le i, r_k \ge j+1$，这些询问的贡献是相同的。

若$l_k \le i$，考虑每个函数的值考虑的是哪一种代价。设$ch_{i,j}$为询问右端点在块$[i,j)$内移动时，改变最优选取方式的位置。

每个函数变换代价的位置有$ch_{i,j} \le ch_{i+1,j} = ch_{i,j+1} \le ch_{i+1,j+1}$，即分成四段。

以$ch_{i+1,j} \le r_k \le ch_{i+1,j+1}$为例，此时$w_{i,j} = w_{i+1,j}$,$w_{i+1,j+1} = (r_k - i - 1) \le (j + 1 - r_k) = w_{i,j+1}$，则$w_{i,j} + w_{i+1,j+1} \le w_{i+1,j} + w_{i,j+1}$。

$r_k \ge j+1$时同理。证毕。

利用分类讨论l_k, r_k的情况可以在$O(\log N)$的时间内求出某一$w_{i,j}$的值。

5.1 单调栈

在本题中，我们显然可以直接套用通用的单调栈解法。这样做一共会询问$O(N \log N)$次权值函数，总的复杂度就是$O(N \log^2 N)$。

5.2 线性做法[2]

我们利用第四节中的转化来考虑这个问题。

定义 5.1. 设r, c为两个指针，且初始时$r = 0, c = 1$。决策过程中始终有$\forall 0 \le i < c, i$的决策已知。

定义$E_j = A[x, j], x < r$，且时刻满足$A_{pos(j),j} = \min\{E_j, \min_{i \ge r} A_{i,j}\}$。在之后的解法中，我们往往记录$E$中存储值的下标。

定义$A[l \sim r]$为数组A的子区间$[l, r]$，$A[i_1 \sim i_2 : j_1 \sim j_2]$为矩阵$A$的子矩阵$A[i_1, i_1 + 1..., i_2 : j_1, j_1 + 1, ..., j_2]$。

在每一次操作中，我们需要扩展r, c的值，并动态维护E的取值。

设$p = min(2 * c - r, N)$,将$E[c \sim p]$作为一行置于$A[r \sim c - 1 : c \sim p]$之前作为第一行,构成一个$(c - r + 1) * (c - r + 1)$的矩阵。

引理 5.1. *若A是完全单调矩阵，上述方法构成的新矩阵是一个完全单调矩阵。*

证明：在新矩阵中，我们只需将有关$E[c \sim p]$证明时的行换成最小值所在行，列数不变，利用之前的证明方法。证毕。

注意到上述矩阵中的所有值都是已知的，我们可以利用之前的$SMAWK$算法在$O(c - r + 1)$的时间内求出这个矩阵每列的最小值，记为G。

此时，在G中存储的是$\forall c \le j \le p, \min_{i=0}^{c-1}\{A_{i,j}\}$。接下来，我们利用求出的值对$j \in [c, p]$进行依次求解。

注意到$j = c$时，已经有$pos(j) = G_j$。

引理 5.2. *当求解$pos(j)$时，$A[c \sim j - 2 : j \sim p]$必定都为无用元素。*

这并不是很显然，仅当$j = c + 1$时是比较容易证明的。但这条定理的正确性将在之后的决策中得到体现。

根据定理，我们只需要从$j - 1$和G_j中挑出一个作为决策。

若$A_{j-1,j} < G_j$，那么$F_j = A_{j-1,j}$，此时所有$< j - 1$的决策在j之后都不如$j - 1$优秀。取$c = j + 1, r = j - 1$，就可以结束这一次操作。由于已经知道之后的决策都不会小于$j - 1$，我们也不需要更新E。

否则$F_j = G_j$。但是仅仅如此我们无法维护E，还需要继续比较$A_{j-1,p}$与G_p的值。

若$A_{j-1,p} < G_p$,由于$A[c \sim j - 2 : j \sim p]$已经假定为无用元素，对于$A^{j+1}, ..., A^p$,在所有小于$< j - 1$的决策中仅有之前$G$中求出的行有可能成为$pos$。令$c = j + 1, r = j - 1, E[j + 1 \sim p] = G[j + 1 \sim p]$，因为有$pos(p) \ge j - 1$，所以$> p$的所有列的$E$值并不用更新，仍然满足所需的条件。

当$A_{j-1,p} \ge G_p$时，有$pos(p) \ne j - 1, pos(j) \ne j - 1$，即$A[j - 1 \sim j - 1 : j \sim p]$都是无用元素。注意到此时$E$不会改变，而对于一个$j < k \le p$的$k$来说，$A[c \sim k - 2 : k \sim k]$将在$j$不断递增时被淘汰，之前的定理也得到证明。

若$j > p$，我们得到了$F[c \sim p]$的值，此时$c = p + 1$，而r的最优取值显然为$pos(p)$。

5.2.1 复杂度分析

在一次决策中，我们所花费的时间为$O(c - r)$。若在前两种情况终止，那么r至少增加了$(j-1) - r \geq c - r + 1$。若在$j > p$时终止，则c至少增加了$(p+1) - c \geq c - r + 1$，或是已经结束了整个算法。而c, r的上限均为N，由此得出该算法的总时间复杂度为$O(N * val())$，在原问题中，即为$O(N \log N)$。

5.2.2 具体实现

这个算法在使用时需要用到许多的子矩阵信息，作者在这里介绍一下自己所用的方法，希望起到抛砖引玉的效果，能够有常数更好的实现方法出现。

在利用$SMAWK$算法时，作者将行列的信息分别存放在一段连续的数组内，当进行 *Reduce* 操作时再将行的部分单独拿出来用双向链表连接。对于$E[j : p]$,作者用一个特殊的数字来代替它，计算具体权值是再将其还原为正确的行数，这样做也可以方便的求出每一列所对应的决策究竟是哪一行。

6 总结

我们从一个特殊的例子和一个简单的分治法出发，仅仅通过四边形不等式本身所拥有的一些优秀的性质，不借助任何高级的数据结构，就在线性时间内解决了这一问题。这也是动态规划这一类问题所深深吸引我的地方。

文中提到的算法各有优劣，作者也以询问权值的次数做了一个小小的实验。线性的斜率优化算法在所有算法中是时间复杂度最优秀的算法，但在使用时的局限性也最大。当函数的变化比较平缓时，决策单调性中不同的决策点比较少，此时分治法与二分法的询问次数较少，有时在随机数据下能够做到期望线性。而当函数比较陡峭(如$NOI2009$《诗人小G》)时，队列中元素较多，二分算法的询问次数就是接近$O(N \log N)$的。而线性算法的询问次数则比较稳定，在$N = 1000000$的数据下能够做到前一种算法的$\frac{1}{2}$。这也启示我们应该根据题目选择更为合适的算法来解题。

7 致谢

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和董烨华老师的关心和指导。

感谢国家集训队教练余林韵和张瑞喆的指导。

感谢清华大学的任之洲学长和绍兴一中的洪华敦,孙耀峰等同学对我的帮助。
感谢绍兴一中的孙奕灿，任轩笛同学为本文验稿。
感谢其他对我有过帮助和启发的同学和老师。
感觉父母对我的关心和支持。

参考文献

[1] Zvi Galil,Kunsoo Park A Linear-Time Algorithm for Concave One-Dimensional Dynamic Programming

[2] Aggarwal, A., Klawe, M. M., Moran, S., ShOT, P., and Wilber, R. Geometric applications of a matrix-searching algorithm. Algorithmica 2 (1987), 195-208

[3] *UOJ Goodbye Bingshen* 题解

《被操纵的线段树》命题报告

江苏省常州高级中学 沈睿

摘 要

来自于笔者2017年集训队互测第四轮中命制的一道以线段树为背景的多解法数据结构题。本文记述了三种不同的解决该问题的思路框架，每条思路都需要选手以一定的抽象能力来架构模型，同时还需要灵活运用贪心、倍增、分治等算法及线段树、树套树、动态树等数据结构，才能方便地维护模型中的信息并解决问题。

1 试题大意及数据范围

给一棵长度为N线段树，但该线段树不同于普通线段树，在普通的线段树当中，对于一个区间$[l,r]$，我们会选择$mid=\left\lfloor\frac{l+r}{2}\right\rfloor$，将该区间分成$[l,mid],[mid+1,r]$。在本题的线段树里，$mid$的值是给定的（但$mid$依然满足$l\le mid<r$），其余条件基本同原OI中的线段树。如下图，是一棵mid值给定的线段树：

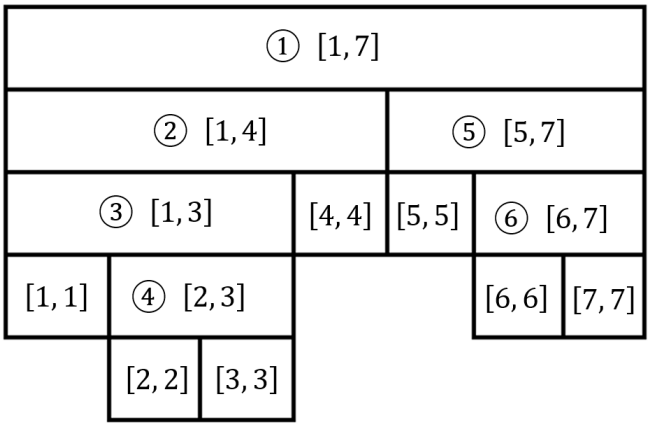


图 16: 图例1

题目给定Q个修改和询问，每次修改会对一个树中的一个子结构进行左旋和右旋，具体旋转模式如**图例2**：

不难发现该旋转规则同$splay$的旋转模式类似。

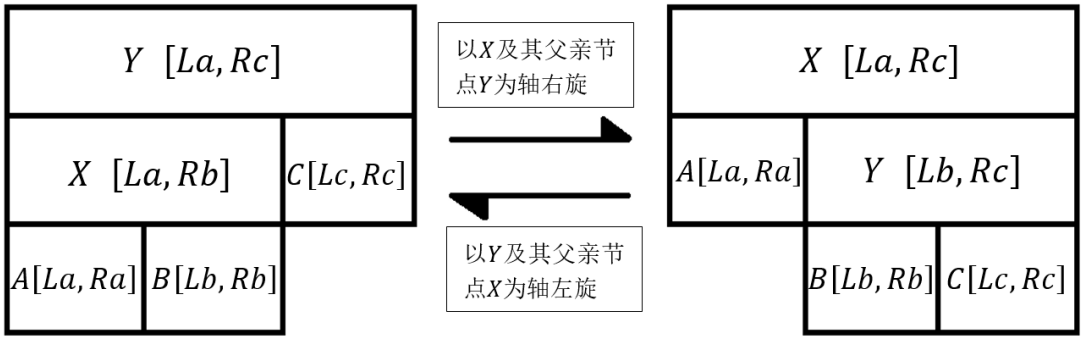


图 17: 图例2

- 子树A，B，C不会产生任何变化。
- 子树A，B，C及节点X，Y之间的相对位置会发生改变。
- 节点X，Y的区间刻度会发生改变。

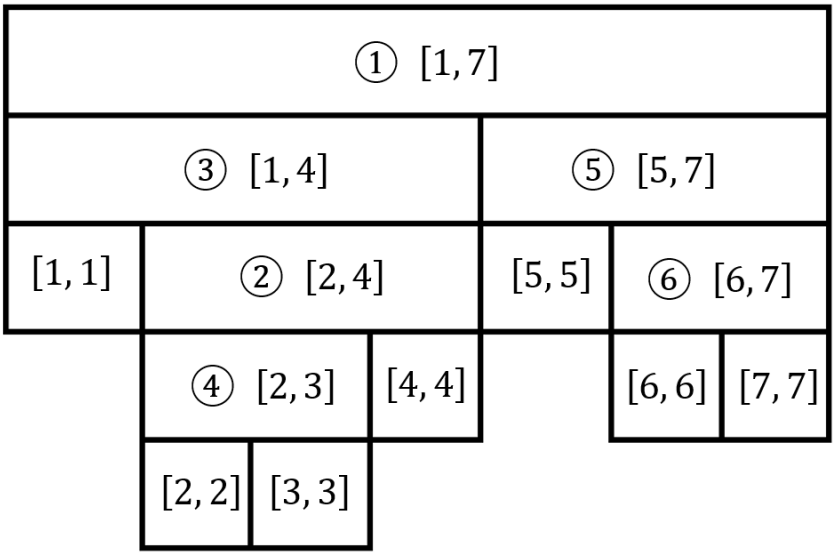


图 18: 图例3

如**图例3**，是将**图例1**中的线段树，以3号结点及其父亲2号节点为轴进行右旋操作后的结果。

题目中每次询问会给定一个区间，问给定区间在当前线段树上的区间定位个数。

*什么是区间定位：用最少的线段树上节点表示的线段去拟合这个区间，使得每个单位长度有且仅在这些线段中出现一次。如**图例3**中的线段树，区间[2, 6]在该线段树上的区间

定位为$[2,4],[5,5],[6,6]$，个数为3。

其中部分数据要求在线。时间限制1-2s（根据评测机不同选择时限）。

表 5: 数据范围

测试点	N的大小不超过	Q的大小不超过	是否在线	是否有修改	空间限制
1	1000	1000	否	无	512MB
2			是		
3			否	有	
4			是		
5	200000	200000	否	无	
6					
7					
8					
9			是		
10					64MB
11					
12			否	有	512MB
13					64MB
14					32MB
15			是		512MB
16					
17					64MB
18					32MB
19					
20					

2 一个显然的暴力

暴力模拟线段树旋转的过程，对于每个询问，我们采用平时线段树区间递归访问的做法。从根出发，一旦当前节点表示的区间被询问区间覆盖，则计入。若无交集则回溯，有交集则递归向左右子树求解。

不妨来分析一下该暴力做法的复杂度。

定理 2.1. *对于整个递归访问过程来说，线段树每层至多被访问4个区间。*

证明 1. 采取反证法：

我们不妨假设在一次访问操作中，存在某层被访问了$k(k > 4)$个区间，容易知道的是这k个区间在刻度上必然是连续的，不妨假设这k个区间在按刻度顺序标号为$s_1, s_2, s_3....s_k$。又因为只有与询问区间有交集的区间才会被访问，那么我们不难发现s_1, s_k两个区间要么被询问区间$[l, r]$完全覆盖，要么与询问区间$[l, r]$存在一部分的交集；而$s_{2..k-1}$是必然被询问区间$[l, r]$完全覆盖。由于被覆盖的这些区间个数至少有$k - 2(\geq 3)$个，必然存在相邻两个区间可以作为某个大区间的左右儿子。由于该大区间也被询问区间$[l, r]$完全覆盖，所以访问到该大区间时便会停止递归，与访问到该层区间的左右儿子事实不符。故假设不成立。证毕。

因为每层至多访问4个区间，所以一次访问区间的复杂度为O(线段树的深度)。在普通的线段树中，由于深度为$O(\log N)$数量级，所以一次区间操作的复杂度也为$O(\log N)$。但由于该线段树mid值是任意取值的，于是该线段树的深度可以达到$O(N)$级别，所以一次区间访问的复杂度最坏是$O(N)$的，总体时间复杂度$O(NQ)$。

对于第5个测试点和第9个测试点，初始线段树的mid是随机生成的，故期望深度$O(\log N)$，每次询问的时间复杂度$O(\log N)$，总复杂度$O(Q \log N)$，上述方法可以通过。

期望得分20-30分。

3 算法介绍

3.1 idea1：从区间入手的算法一

记号 1. 将所有节点所代表的区间从线段树的结构中离散出来，定义这个区间集合为S。

记号 2. 定义当前询问操作的询问区间为$[l, r]$。

根据对线段树的观察，我们得到如下定理：

定理 3.1. $[l, r]$的区间定位个数（答案）$= \{ 2 * (r - l + 1) \} - \{S$中完全被$[l, r]$包含的区间个数$\}$。

证明 2. 我们将完全被$[l, r]$包含的区间的集合返还至原来线段树的结构上，发现所有区间集合形成了一个森林。而对答案产生贡献那些区间节点在这些森林当中均为根。

由于该森林中的树均满足“每个非叶子结点存在两个儿子”这个性质，所以对于一棵树来说我们又有：叶子结点个数 = 非叶子结点个数 +1。

对于整片森林来说又有等式：叶子结点个数 = 非叶子结点个数 + 根的个数。叶子结点个数为$r - l + 1$，非叶子结点个数为S中完全被$[l, r]$包含的区间个数$-(r - l + 1)$。

移项得定理3.1。

于是我们将问题转化为维护所有线段树上的区间的集合，每次询问被一个区间所完全包含的区间个数。

3.1.1 经典的二维数点问题

设查询区间为$[l,r]$，那我们其实是查询，在集合S中的区间$[l',r']$，满足$l \le l', r \ge r'$的所有区间个数。

我们不妨将一个区间$[l',r']$视作平面上的一个点(l',r')，然后问题转化为了平面上有N个点，有Q个询问。每次询问位于给定点的左上角的点的个数。显然，在OI中，这是一个经典的二维数点问题。

3.1.2 part1

考虑对于离线无修改的部分，不妨可以将所有询问离线出来，将所有询问点和平面上的点放在一起进行排序，然后按顺序使用树状数组统计即可。即一维排序+二维树状数组的做法。

时间复杂度$O(N \log N)$，空间复杂度$O(N)$，结合算法一期望得分40-45分。

3.1.3 part2

考虑一下在线无修改的部分如何解决。既然强制在线，那我们干脆使用更暴力的数据结构树套树进行维护，只是每个区间插入需要$O(\log^2 N)$的复杂度，而且每次询问也多了一个log的复杂度。

时间复杂度$O((Q+N)\log^2 N)$，空间复杂度$O(N \log N)$（线段树/树状数组套平衡树）/$O(N \log^2 N)$（线段树/树状数组套线段树），期望得分45-55分。

3.1.4 part3

考虑带修改的部分。不难发现一个旋转操作在该算法中，本质上只是在原有的区间集合中删除一个区间再添加一个区间。如在**图例2**中的右旋操作中，本质上是在集合S中删除了$[L_a,R_b]$区间，添加了$[L_b,R_c]$ 这个区间。

那带修改的问题便转换成了，支持动态添加删除的二维数点问题。

那么对于离线的部分，由于每个修改都是**独立**的，而且答案是可以**合并**的。那我们不妨可以采用分治。对于添加一个区间的修改，我们视为在二维平面中添加一个权值为1的点，对于删除一个区间的修改，视为在二维平面上添加一个权值为−1的点。那么问题转换成，询问一个矩形区域的总权值。

那我们直接对时间分治，在对于时间区间$[L,R]$进行分治的时候，我们计算$\left[L,\left\lfloor\frac{L+R}{2}\right\rfloor\right]$中的修改对于$\left[\left\lfloor\frac{L+R}{2}\right\rfloor,R\right]$中询问的贡献。之后递归处理$\left[L,\left\lfloor\frac{L+R}{2}\right\rfloor\right],\left[\left\lfloor\frac{L+R}{2}\right\rfloor,R\right]$两个子区间即可。不难发现，这样每个修改对于之后的询问的贡献都有被计算到。

在一个时间区间内处理修改对询问的贡献的问题等同于一个离线的二维偏序问题，直接使用一维排序+二维树状数组即可。由于处理k个询问和修改的时间复杂度为$O(k\log k)$，根据主定理可得整个分治的时间复杂度为$O(Q\log^2 Q)$，空间复杂度为$O(N+Q)$，可以通过12-14等测试点，结合算法一可以获得60分。

3.1.5 part4

既然在无修改的时候我们直接使用了树套树去维护区间，在有修改且在线的时候我们不妨也可以使用该数据结构，对于每次修改只是在该数据结构中删除一个点，加入一个点罢了。修改的时间复杂度同询问一样均为$O(\log^2 N)$。时间复杂度时间复杂度$O((Q+N)\log^2 N)$，空间复杂度$O((Q+N)\log N)$（线段树/树状数组套平衡树）/$O((Q+N)\log^2 N)$（线段树/树状数组套线段树），由于对空间限制有所要求，所以该算法并不能获得所有分数，期望得分60-80分。

3.1.6 conclusion

综合算法二的所有部分，总得分最高可达85分。

3.1.7 一些基于idea1但令人意外的满分做法

虽然idea1的一系列算法看似最多只能获得85分。但在集训队互测的现场，有选手使用idea1的做法获得了满分。

来自北京师范大学附属实验中学的徐明宽同学利用树状数组套平衡树的做法，加上巧妙地将3个int压成一个long long突破了空间限制，获得了满分。但是代码的长度以及在该题花费的时间过长，不宜提倡。

来自绍兴市第一中学的冯哲同学用了kd-tree维护二维平面上的点，每次插入点的时候暴力在kd-tree中加入新的点，由于需要维护kd-tree的结构，他选择每隔添加13000次点后暴力重构kd-tree。对于一个询问操作，在kd-tree中暴力递归询问，即当前子树的所有点均在询问区域内则计入答案，若在区域外则回溯，有交集则递归询问。本做法并不能给出较为直观而确定的时间复杂度，但实际上该程序在TUOJ上运行的时间只有std(0.4s) 的1.6倍，相较于现场来自陈俊锟的最快代码(0.2s)也只有约为3倍时间，再加上只需$O(Q+N)$的空间，可以获得满分。

3.2 idea2：基于贪心的算法二

对于一种特殊的情况：

定理 3.2. 当询问区间$[l,r]$的l位于某个线段树子树的最左端时，且r在该线段树子树的刻度内。那么对于r，我们贪心的选取右端点为r的最大区间$[l',r]$，然后让$r=l'-1$重复本操作直到$l=r+1$。选取的区间个数就是$[l,r]$定位的个数。

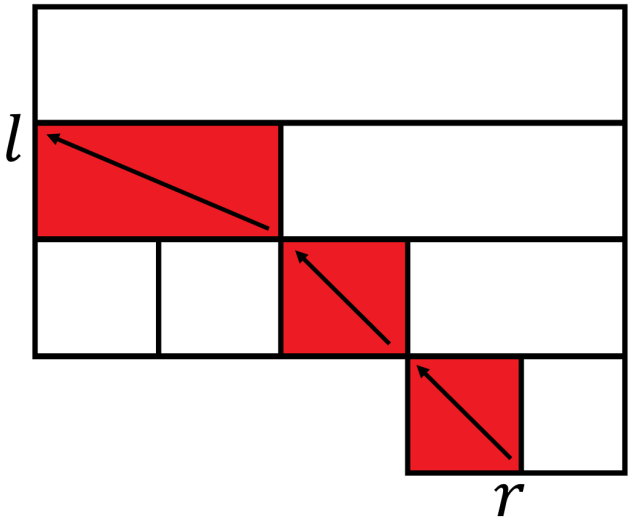


图 19: 图例4

证明 3. 根据**图例4**，由于r在该线段树子树刻度内，所以我们不断贪心地选取右端点为r的最大区间$[l',r]$中的l'必然大于等于整个子树的左端点l。又由于我们每次是贪心地取最大的区间，且所有区间之间只存在相互包含或者不相交两种关系。故我们每次贪心取最大的区间可以保证取得的区间个数最少。

于是对于一般的情况，我们又有如下结论：

定理 3.3. l贪心地选取以l为左端点的最大的区间$[l,r']$，让$l=r'+1$并重复本操作，直到$r'>=r$为止。记录此时的l。

r贪心地选取以r为右端点的最大的区间$[l',r]$，让$r=l'-1$并重复本操作，直到$l=r+1$为止。

答案等于所有贪心选取的区间个数总和。

证明 4. 对于询问的区间的l我们贪心的选取向右跳的最大的区间$[l,r']$，直到r'大于等于询问区间的r时停止，若$r=r'$则转换成了定理5.1的情况，那我们讨论$r'>r$的类型。

如**图例5**，我们把当前选取到的这个右端点大于等于r的大区间定义为$[l',r']$。

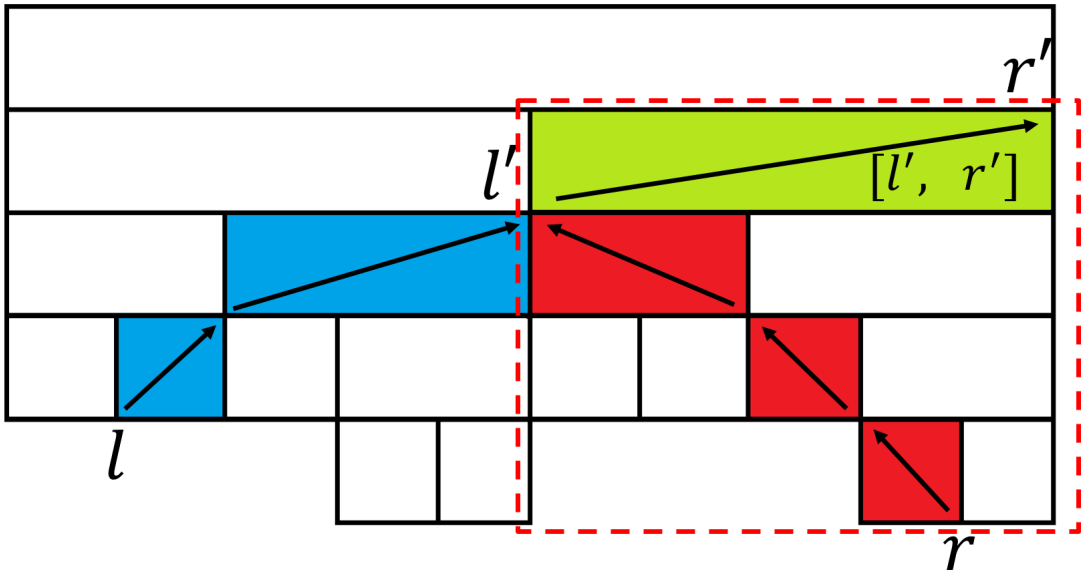


图 20: 图例5

把以区间$[l',r']$为根的子树提出，在这个线段树子树中，当前的询问区间的左端点l'已经在该线段树最左边，且询问区间右端点r在该线段树子树的区间刻度内。那么当前情况和定理5.1讨论出来的情况相同。根据定理5.1此时我们只需要对于r不断求出可以向左跳的最大区间即可。

由于我们每次贪心选取的都是当前合法的最大区间，所以不难证明这样的做法一定可以选取到最少个数的区间。

总结这整个做法：

就是，l贪心地选取向右跳的最大的区间，超过范围时停止，r贪心地选取向左跳的最大的区间，超过范围时停止。答案为所有选取的区间个数总和。

当然我们暴力地选取可以跳跃的区间的复杂度是$O(NQ)$的，与暴力无异。所以我们需要一些算法或者数据结构来维护跳跃的过程。

3.2.1 part1

对于无修改的部分，线段树的结构必然是固定的。那我们不妨先预处理出每个位置向左、向右可以选取的最大区间，等同于维护每个位置向左或向右选取最大区间后跳跃到的位置。这些跳跃的关系是一种多对一的映射。根据观察，我们发现对于同一个方向的跳跃关系来说，形成了一棵树的关系。于是使用倍增+二分的算法加速跳跃的过程即可。

时间复杂度$O((Q+N)\log N)$，空间复杂度$O(N\log N)$，期望得分55分。

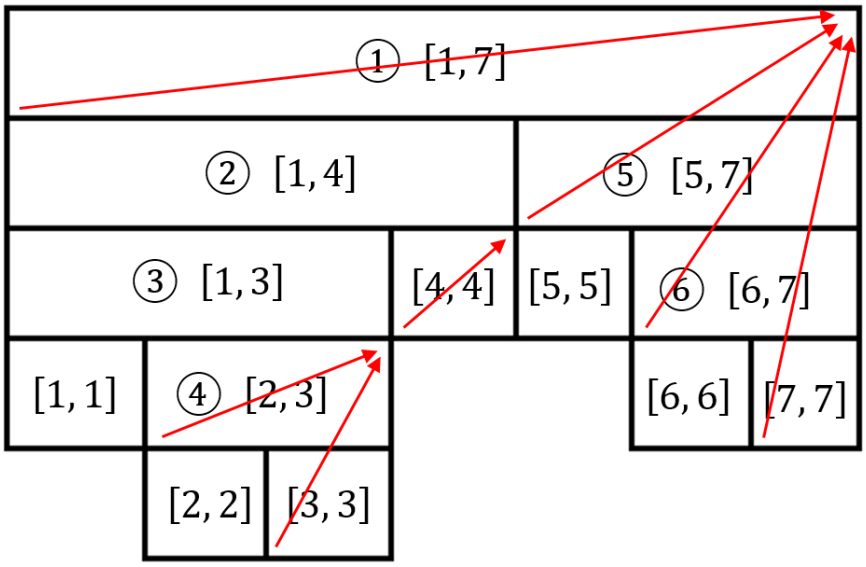


图 21: 跳跃的关系形成了树的结构

3.2.2 part2

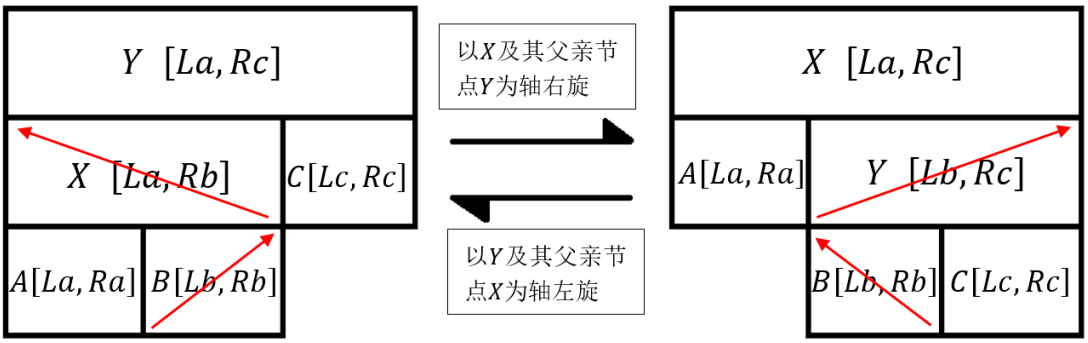


图 22: 旋转操作中改变的跳跃

再考虑修改操作对于跳跃的影响。观察旋转模式图，不难发现本质上只是动态地修改了L_b向左跳跃的位置和R_b向右跳跃的位置。

根据上面的分析我们得出这种跳跃关系形成了一种树的关系，现在我们需要动态地维护这两棵跳跃树。

那么我们不妨使用*LCT*（动态树）来维护这两棵跳跃树：

- 对于一次修改操作来说，本质上是树上一些边的断开和一些点的链接，对应着*LCT*的*link*和 *cut*两个操作，时间复杂度$O(\log N)$。

- 对于一次询问操作，本质上是在跳跃树上询问某个节点有多少个不超过某个范围内的祖先，那对应在*LCT*上的操作是先*access*当前节点，并在当前节点所在的*splay*上二分，便可得到答案。时间复杂度$O(\log N)$。

时间复杂度$O((Q+N)\log N)$，空间复杂度$O(N)$，期望得分100分。

这也是作者给出的标准做法。在互测的现场，来自福建省福州市第一中学的陈俊锟运用该做法获得了满分。

3.3 idea3：抽象成树的算法三

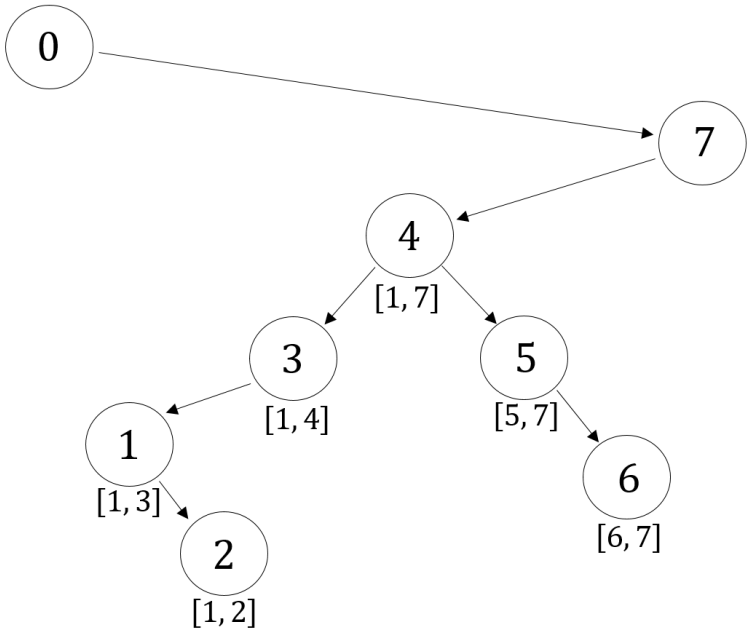


图 23: 图例6

我们不妨尝试将这棵线段树以树的结构画出，同时进行重标号，对于每个非叶子节点我们将它的*mid*作为其标号构建一棵树。如**图例6**，是**图例1**抽象成树的样子，其中只保留了非叶子结点，该树中空的左右儿子部分均为叶子结点但没有编号也没有画出，因为在做法中不需要用到但在证明环节中可能会需要使用。而且该树在原来的基础上在根之上添加了两个新的节点0和N。本质上是在线段树的外层添加了两个结点，然后使得求解问题的线段树处于一棵子树内。目的是为了便于使该算法便于描述，不需要加入一些特判环节。

我们发现这棵树有如下优美的性质：

- 该树中序遍历依次递增为0..N。即使进行旋转操作后，依然不变。
- 一棵子树的编号是连续一段，且中序遍历递增，以$[l,r]$区间为根的子树的中所有节点的编号集合为$l..r-1$。

通过一定的观察，我们得到以下定理：

定理 3.4. *设询问区间为$[l,r]$，设$t = lca(l-1,r)$。那么区间定位的个数对应到这棵抽象树上等于$l-1$到t的路径上（不包括t）的左孩子个数与r到t路径上（不包括t）的右孩子个数之和。*

证明 5. $l-1$到t的路径上（不包括t）的除t的左孩子以外的所有左孩子的父亲的右孩子，再加上$l-1$的右孩子对应着区间定位的左半部分（这些定位的区间全是右孩子）。

r到t路径上（不包括t）的除t的右孩子以外的所有右孩子的父亲的左孩子，加上r的左孩子对应着区间定位的右半部分（这些定位的区间全是左孩子）。

其中t的左孩子被计入和不被计入的$l-1$的右孩子相抵消。同时t的右孩子被计入和不被计入的r的左孩子相抵消。

为了便于理解，不妨以**图例1**的线段树（对应着**图例6**的抽象树）为例。当前询问区间$[3,6]$，在树上找到2和6的最近公共祖先4，2到4的路径上的左孩子为1,3，6到4中路径上的右孩子为5,6。共4个，对应的区间定位个数也是4个。其中左孩子1对应着其父亲3的右孩子$[4,4]$，左孩子3是最近公共祖先4的左孩子和2的右孩子$[3,3]$相抵消，右孩子6对应着其父亲5的左孩子$[5,5]$，右孩子5是最近公共祖先4的右孩子和6的左孩子$[6,6]$相抵消。左右孩子和定位区间一一对应。故定理成立。

3.3.1 一个直观的做法–LCT

由于本题要求在线，所以一个直观的做法就是动态地维护这棵树。那么可以胜任的数据结构非LCT莫选。

- 对于一个修改操作，本质上是一些边的断开和一些点的链接，对应到LCT上的操作 *link* 和 *cut* 。其中还会修改一些点为左孩子还是右孩子的权值，在*splay*上修改并维护即可。时间复杂度$O(\log N)$。
- 对于一个询问操作，需要支持查询*lca*。设要查询x和y两个点的*lca*，那么对应LCT上的操作是先*access*(x)，后*access*(y)，于是x所在的*splay*的虚边连接的父亲就是*lca*。对于接下来的链操作来说，通过LCT的换根操作和*access*操作，并在*splay*上维护左儿子个数和右儿子个数即可完成询问。时间复杂度$O(\log N)$。

时间复杂度$O((Q+N)\log N)$，空间复杂度$O(N)$，期望得分100分。

3.3.2 ☆以中序遍历为坐标轴维护线段树

经过观察，我们只需要维护出每个点到根的左儿子数目和右儿子数目，并支持动态的求lca的操作便可获得答案。因为$l-1$到lca的左儿子数目可以由$l-1$到根的数目减去lca到根的数目得到，r到lca的右儿子数目也同理。

那我们不妨可以维护以中序遍历为坐标的线段树。维护这样的三棵线段树：每个点到根的左儿子数目L，每个点到根的右儿子数目R，以及每个点的深度Dep（即每个点到根的点的数目）。

之前我们描述到这棵树的性质有：该树的中序遍历就是编号顺序，且一棵子树中的所有点的编号处在中序遍历中连续的一段。这棵树的中序遍历和树的dfs序相类似，即有共同性质：一棵子树位于这个序列中相连续的部分。便可以如此维护：

- 对于一个左儿子节点，我们可以通过线段树的区间修改将其子树所在区间的$L+1$；
- 对于一个右儿子节点，我们也可以通过线段树的区间修改将其子树所在区间的$R+1$。
- 对于每个树上的节点，用区间修改将子树区间的$Dep+1$。

按此方法维护，线段树L,R,Dep中单点的值就代表每个点到根的左儿子数目、右儿子数目和深度。

考虑每个操作需要如何维护：

- 对于一个修改操作，本质上是修改了一个点的子树区间，以及将一个左孩子变换成右孩子，将一个右孩子变成左孩子。在修改子树区间时，对应到线段树上将原来所在子树区间的$Dep-1$，将修改后子树区间的值$Dep+1$。修改结点的左右孩子性质时也是同理的操作。时间复杂度$O(\log N)$。
- 对于一个询问操作，我们仍然需要支持查询lca。设询问区间为$[l,r]$，那$lca(l-1,r)$是$[l-1,r]$这个区间中深度最小的那个点，由于我们在线段树上维护了深度，所以运用线段树的区间询问求区间最小值即可。对于剩下求$l-1,lca$到根的左孩子数目和r,lca到根的右孩子数目，使用单点查询即可。时间复杂度$O(\log N)$。

至此，这个做法只运用了线段树这一个简单的数据结构便成功解决问题，相较于前面的LCT或者树套树等较为繁琐的数据结构来说显得十分清晰和简洁。总体的时间复杂度为$O((Q+N)\log N)$，空间复杂度$O(N)$，期望得分100分。

在互测的现场，来自福建省福州第三中学的闫书弈运用了该做法（*idea*3+以中序遍历为坐标轴维护线段树）获得了满分。

4 数据生成方法

对于*mid*值如何取值是本题的一个关键所在，如果随机取值*mid*，那线段树的期望深度依然是$O(\log N)$，同原有的线段树基本上没有变化，直接暴力即可AC。于是我采取了大框架，小随机的生成方法。先构造一个大约长度为L链式的大框架，在链下随机构造一个小的线段树。这样线段树的深度至少为$O(L)$，每次暴力询问复杂度约为$O(L)$。对于这个L我取值10000-100000，随着数据编号的增加而增加。达到了加强数据，卡掉一些暴力做法的目的。

5 命题契机

本题原型是在2016年我校暑期夏令营中笔者出的一道题，当时是没有修改且不强制在线的版本。当时考场中河北省石家庄二中的郭资政同学用了我当时没有想到的*idea*1获得了满分。然后在互测期间，我添加了旋转操作使得题目模型看起来更为复杂一些，而且相信放在互测中一定可以集思广益创造出更多有趣的算法。果然，参与互测的同学们非常给力，利用各种不同的做法获得了高分。关于题目背景，灵感取自于UNR中吉如一学长出的《奇怪的线段树》。是一道同样以*mid*值任意的线段树为蓝本，但解题思路和算法却是大相径庭的题目。

6 总结

本题的知识点非常全面，涵盖了OI中的模拟、贪心、倍增、分治等算法，充分考验了选手对于线段树、平衡树、树套树、动态树等基础数据结构的掌握和运用。考查基本功的同时，更多的考验选手观察题目、总结方法的一个能力。此题根据观察、归纳的方法不同，可以衍生出三条以此方法为出发点的线路，充分增加了解题方法的多样化，提高了题目的可做性，带给选手们无与伦比的做题体验。

在互测现场一共有4位同学获得了满分，他们使用的方法各不相同，在前面也有所提及。值得一提的是，笔者在出题时只想到了*idea*1和*idea*2这两个大算法框架。而闫书弈选手巧妙的*idea*3和利用线段树去维护的做法，在开场1个半小时就获得满分，也才有笔者将其记录于解题报告这么一回事。总得来说，本题是一道看似单一，实则多变而有趣的数据结构题。期待能有更多的算法来解决本问题，欢迎有想法的读者联系笔者进行学术交流。

7 鸣谢

感谢中国计算机学会提供学习和交流的平台。

感谢江苏省常州高级中学的曹文老师，吴涛老师多年来给予我的关心和指导。

感谢中山纪念中学的翁文涛，长沙市雅礼中学的欧阳思琦同学一路上对我的支持和帮助。

感谢北师大附属实验中学的徐明宽，绍兴市第一中学的冯哲，福州第一中学的陈俊锟，福州第三中学的闫书弈同学在互测时给我的良好反馈，以及互测后对我的启发和帮助。

感谢长沙市雅礼中学的毛啸同学与我讨论关于将问题转化为维护权值上升序列来解决*splay*到根的一类问题。

感谢江苏省常州高级中学的姚嘉和为本题验证数据。

感谢中山纪念中学的翁文涛，成都七中的杨景钦对本文提出修改意见。

参考文献

[1] 徐寅展，《线段树在一类分治问题上的应用》，2014年国家集训队论文。

[2] 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。

[3] 吉如一，UNR《奇怪的线段树》解题报告， `http://c-sunshine.blog.uoj.ac/blog/1860`

计算机逻辑与艺术初探——基于逻辑的钢琴演奏音符力度模型

中国人民大学附属中学 赵晟宇

摘 要

本文涉足于人工智能与艺术领域，给出并实现了一种为音符确定力度的算法，以得到一种可能的有表现力的钢琴演奏方案。本文基于作者的钢琴演奏及作曲经验，尝试为音乐与感情(表现力)建立逻辑上的联系并加以分析，给出了一系列创造性的性质及定义。并以此为模型，通过假定的复合乐段结构以及复合旋律结构，实现了一种最优化构造算法。相较于一贯的统计或学习类方法，本算法已经能表现出相当好的不受限性。本文期望可以提供人工智能在艺术方向上基于认知的发展前景。

1 引言

到目前为止，世界范围内人工智能在音乐上的研究似乎仍处于起步阶段。由于艺术创作具有高度主观性，很难被量化定义，无法直接在数据层面上进行分析。多数研究者从机器学习或数据归纳入手，试图挖掘其中潜在的性质与规律，但效果仍然不甚理想。

人工智能在音乐上可以处理的任务，如人工智能作曲、人工智能伴奏、人工智能演奏等，在大体上是相通的。本文单从钢琴演奏中的音符力度变化开始进行研究。

有学者发表了基于概率论模型在钢琴演奏上的研究[58]，并将其结果与古典名曲进行了比较，结果尚可。全国范围内尚未检索到相关文献。现有的研究大多是基于统计规律的，而本文希望尝试充分利用人类的艺术认知，建立音乐与感情(表现力)在逻辑上的联系。

人工智能在艺术上的应用还有很大的探索空间。这项技术可以方便作曲人或编曲人得到理想的强弱处理结果，也能帮助广大音乐爱好者试听自己的创作经过艺术化处理后的效果。配合自动识谱模块，便可以应用于钢琴自动演奏系统上的读谱弹奏功能。

[58]见参考文献[2]

2 采集与输出

我们使用MIDI文件[59]进行输入输出。输入包含每个音符的音高、开始时刻与结束时刻，以及延音踏板信息。输出则应在输入文件的基础上，附加每个音符的力度信息。在MIDI文件中，力度是 $[1, 127]$ 中的一个整数。

测试数据是我在MIDI电子键盘上弹奏采集的。采集数据可作为正对照。将采集数据的力度信息清除，便可产生输入数据，同时也是空白对照。输出的力度信息应提供一种可能的演奏方案。由于音乐性很难被衡量，我们只能将结果与空白对照和正对照进行主观比较。

在全文中， $pitch_u$ 表示音符 u 的音高(半音阶数)， $start_u$ 与 end_u 分别表示音符 u 的开始时刻与结束时刻， $duration_u = end_u - start_u$ 表示持续时间， vel_u 表示音符 u 的力度。

3 单旋律算法

不妨先从单旋律入手。

这里的单旋律，指乐谱中所有音符均作为一个整体的旋律体现，同一时刻上只存在一个正在发展的旋律。每个音符在整体旋律中应具有相似的权重，即不会特殊强调或减弱某些特定音符。音符随时间的关联应是连续的，不会产生力度突变。

3.1 乐段结构

在钢琴演奏中，应该经常会接触乐段(或乐句)这个概念。乐谱中这些潜在的结构，影响了我们的演奏方式。我们可以先从分析乐段开始。

一个乐段是在时间线上连续的一段音符。可以用开始时刻与结束时刻表示一个乐段。

3.1.1 乐段结合度

一个乐段内部应是具有和声关联的。此外，如果存在较大的音符空隙，我们更倾向于将乐段从此处断开。

我们希望量化定义一个乐段 P 的结合度 $BindingDeg(P)$ ，以便于接下来的乐段分解工作：

$$BindingDeg(P) = (1 - \frac{emptylength_P}{totallength_P})HarmonyDeg(P)$$

其中 $totallength_P$ 表示乐段 P 的总时长， $emptylength_P$ 表示乐段 P 中完全没有音符覆盖的总时长。我们需要继续定义乐段 P 的和谐度，即 $HarmonyDeg(P)$ 。

[59]标准MIDI文件格式，扩展名为.mid

3.1.2 音符响度

首先，持续时间较短的音符对整体和声的影响较小。假定每个音符的音量都是指数衰减的，那么我们可以对一个音符 u 的总响度 $V(u)$ 作如下定义：

$$\begin{aligned} V(u) &= \int_0^{duration_u} \mathrm{e}^{-\alpha_s x}\mathrm{d}x \\ &= \frac{1-\mathrm{e}^{-\alpha_s * duration_u}}{\alpha_s} \\ V(P) &= \sum_{u\in P} V(u) \end{aligned}$$

一个乐段 P 的总响度即为其中所有音符的响度之和。

其中 α_s 为衰减常数，本实例中取 $\alpha_s = 1.0$ 。

3.1.3 乐段和谐度

两个音符之间的和谐程度是比较容易判断的。将整个乐段看作两两音符的结合，我们继续作出如下定义：

$$HarmonyDeg(P) = \sum_{u\in P} V(u)(\frac{\sum_{v\in P} V(v)H(u,v)}{V(P)})^{\beta_h}$$

其中 β_h 为一个指数常数，本实例中取 $\beta_h = 3.0$ 。

$H(u,v)$ 衡量了一对音符 u,v 的和谐程度，由它们之间的音程(半音阶间隔)决定。忽略整八度(12 个半音阶间隔)的音程差，我们这里定义和声音程：

$$PitchInvl(u,v) = (pitch_u - pitch_v) \mod 12$$

同音高与纯五度音程是最和谐的，而小二度(一个半音阶间隔)音程是最不和谐的。本实例中 $H(u,v)$ 的值由下表给出：

$PitchInvl(u,v)$	0	1	2	3	4	5	6	7	8	9	10	11
$H(u,v)$	1.0	0.0	0.25	0.5	0.5	1.0	0.0	1.0	0.5	0.5	0.25	0.0

3.1.4 均摊结合度

乐段 P 的结合度是与乐段长度和音符密度正相关的，因此仍需要定义均摊结合度 $AvgBindingDeg(P)$ ：

$$AvgBindingDeg(P) = \frac{BindingDeg(P)}{V(P)}$$

均摊结合度是一个 $[0,1]$ 的实数，可以直观地体现一个乐段的结合程度。

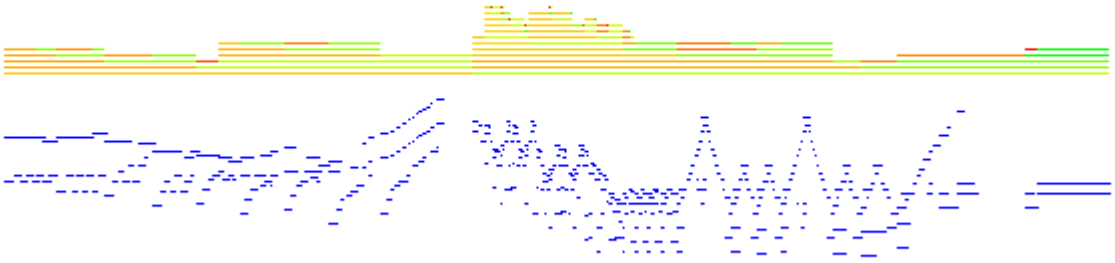
3.2 复合乐段结构

我们应当充分挖掘乐谱中的乐段结构。整首曲子可能分为几个乐章，每个乐章又可以继续分为乐段、乐句，而从逻辑上可以将它们看作递归的复合乐段结构。

3.2.1 复合乐段的树状结构

首先将乐谱整体看成是一个整体乐段，整体乐段应具有比较低的均摊结合度。规模越大的乐段就越为混乱，我们需要递归地将其分解为结合度尽量大的子乐段，这样便形成了复合乐段的树状结构。为方便处理，始终用二叉结构进行乐段分解，即将一个乐段 P 分解为左右的两个子乐段 l_P 及 r_P 。

假定每个乐段的开始时刻都必须是乐谱中某个音符的开始时刻。只能以某个音符的开始时刻为界进行乐段分解，越界的音符将被截断。在我们的定义中，相同层级的乐段间一定是没有交集的，而每一层级所有乐段的并集则必须包含了乐谱中所有的音符。



如上图所示。下半部分描绘了整个乐谱的所有音符，纵轴对应着音高，横轴对应着开始时刻与结束时刻。上半部分描绘了一种可能的复合乐段结构，每条线段对应着一个子乐段，由下至上逐级递归分解，红色段表示左边的子乐段，绿色段表示右边的子乐段，其中色彩的纯度表示着它们的均摊结合度。

3.2.2 复合结合度

为方便最优化整体结构，需要综合乐段 P 及其所有子乐段的结合度对整个子结构的复合结合度进行评估。定义乐段子结构 P 的复合结合度 $ComBindingDeg(P)$：

$$ComBindingDeg(P)=\begin{cases} BindingDeg(P)+\alpha_c(BindingDeg(l_P)+BindingDeg(r_P)) & \text{若 } P \text{ 存在子乐段} \\ \frac{1}{1-\alpha_c}BindingDeg(P) & \text{若 } P \text{ 不存在子乐段} \end{cases}$$

其中 α_c 为复合乐段结合度评估中的逐层衰减常数，本实例中取 $\alpha_c=0.5$ 。

当 P 不存在子乐段时，仍需要将其视为继续无限下分但结合度不变的结构。在 $0 < \alpha_c < 1$ 时，有 $\sum_{i\geq 1} \alpha_c^i = \frac{1}{1-\alpha_c}$ 。

3.2.3 复合乐段分解

现在我们只需要最大化整体乐段的复合结合度即可。

若乐谱中共有 N 个音符，则可能的乐段数量是 $O(N^2)$ 级别的，我们同样可以在 $O(N^2)$ 的时间复杂度内先简单计算出每个可能的乐段的结合度。

考虑复合结合度，其值是与整个子结构的划分方式相关的。可以用区间动态规划的方法计算。自底向上依次考虑每个可能的乐段，每个乐段都最多可能有 N 种划分，只要枚举当前乐段 P 的所有可能划分便可以求得当前最优的复合结合度。

整个过程可以在 $O(N^3)$ 的时间复杂度内完成。

3.3 复合乐段增益

若是仅以完全相同的力度弹奏所有音符，再完美的演奏也是没有表现力的。完成有音乐表现力而又不混乱的演奏，往往在于局部能表现出微妙的强弱变化，而在于乐句或乐段则有一个整体的强弱变化。我们期望可以在复合乐段结构的基础上逐层拟合出尽量真实的情感变化。

由于情感表达的多样性，音乐在力度变化上的限定性是比较弱的。如果没有演奏记号，大部分音乐段落在强弱变化，即强、弱、渐强或减弱的选择上，都是非常自由的。

即便如此，我们发现它与乐段的音高偏离度，即音高的加权标准差，在一定程度上是相关的。即，音高偏离度越大的乐段，在力度上可能会倾向于表现得更强一些。

3.3.1 音高偏离度

定义乐段 P 的音高偏离度 σ_P：

$$\mu_P = \frac{\sum_{u\in P} V(u) pitch_u}{V(P)}$$

$$\sigma_P = \sqrt{\frac{\sum_{u\in P} V(u)(pitch_u - \mu_P)^2}{V(P)}}$$

这对于力度的变化趋势可以起到一定的导向作用。

3.3.2 局部增益

假定一个子乐段在力度增益上的贡献是随时间线性变化的。为保证整体增益变化的连续性，每个乐段从其划分时刻开始，造成一定量的增益，然后向两边线性过渡到 0 。可以

计算出其对于每个音符增益的贡献：

$$gain_{u|P} = \min(\frac{start_u - start_P}{duration_{l_P}}, \frac{end_P - end_u}{duration_{r_P}})gain_P$$

我们仍需要决定每个乐段 P 的增益数量 $gain_P$ 。注意这可以是一个负值。

一个乐段造成的增益幅度，应与其均摊结合度正相关。根据上述原则，半随机化地进行计算：

$$gain_P = \frac{\min(start_u - start_P, end_P - end_u)}{duration_P}\gamma_p AvgBindingDeg(P)Rand(P)$$

其中 $Rand(P)$ 需要按照一定规则生成随机数，例如可以根据 σ_P 的值适量偏移。为避免划分时刻远离中间时刻的乐段可能造成其子乐段的增益变化速率过快，我们还根据划分时刻与边界的距离进行了线性加权。

3.3.3 整体效益

完成复合乐段分解与所有乐段的局部增益后，最终每个音符的增益即为所有乐段对其造成的增益之和。

至此，我们便完成了乐段结构分析以及针对单旋律的音符力度增益。

4 复合旋律算法

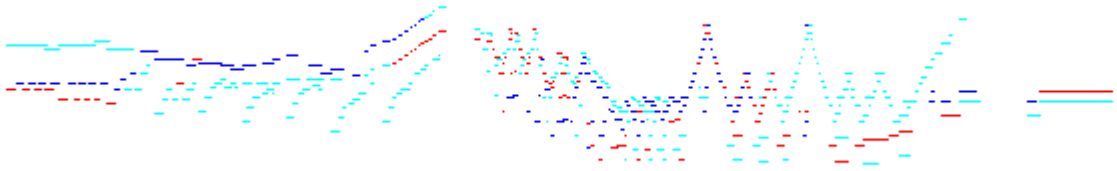
一般的钢琴演奏都是具有复合旋律结构的。如钢琴的左右手在大多数情况下都演绎着相互结合而又相对独立的两个部分，将这两个部分混为单旋律考虑显然是不科学的。先不论标明双声部或多声部的乐曲，即便是有明确主旋律的乐曲，我们也应将其“伴奏”部分看作一些旋律性较低的副旋律的组合。

4.1 复合旋律结构

类似于复合乐段的树状结构，我们也希望将整首乐曲分解为复合旋律的树状结构。复合旋律结构，应整体存在于复合乐段结构之上，所有子旋律围绕这个共同的乐段结构发展。

4.1.1 复合旋律的树状结构

相似地，将乐谱整体看成是一个整体旋律，我们需要递归地将其分解为旋律性尽量大的子旋律，这样便形成了复合旋律的树状结构。为方便处理，始终用二叉结构进行旋律分解，即将一个旋律 Q 分解两个子旋律 l_Q 及 r_Q ， $l_Q \cup r_Q = Q$ 且 $l_Q \cap r_Q = \varnothing$ 。



如上图所示，深色部分共同描绘了某一条子旋律，而红色和蓝色分别为其进一步旋律分解的结果。

4.1.2 旋律性

将一条旋律上每个音符，以开始时刻为横坐标，音高为纵坐标，按时间顺序依次连结成一条折线，理想的旋律应该是比较平滑的。考虑每个音符折点处两边的斜率差。此外，如果在这一条旋律上的某个音符与前后音符之间有较大的空隙，旋律性也将受到影响。

基于这些性质，按开始时刻次序将旋律 Q 的所有音符表示为 $Q_1, Q_2, ..., Q_n$ ，对旋律 Q 的旋律性 $TotalMelodyDeg(Q)$ 作如下定义：

$$TotalMelodyDeg(Q) = \sum_{i=2}^{n-1} V(Q_i|Q_{i+1}) MelodyDeg(Q_i|Q_{i-1}, Q_{i+1})$$

$$\mathrm{MelodyDeg}(Q_i|Q_{prev}, Q_{next}) = \frac{duration_{Q_{prev}|Q_i}}{start_{Q_i} - start_{Q_{prev}}} \frac{duration_{Q_i|Q_{next}}}{start_{Q_{next}} - start_{Q_i}} \mathrm{e}^{-\gamma_r^2 (\frac{pitch_{Q_i} - pitch_{Q_{prev}}}{start_{Q_i} - start_{Q_{prev}}} - \frac{pitch_{Q_{next}} - pitch_{Q_i}}{start_{Q_{next}} - start_{Q_i}})^2}$$

其中 γ_r 为一个压缩斜率纵轴的常数，本实例中取 $\gamma_r = \frac{1}{32}$ 。

$duration_{u|v}$ 与 $V(u|v)$ 仍表示音符 u 的持续时间与总响度，但从下一个音符 v 的开始时刻截断：

$$duration_{u|v} = \min(end_u, start_v) - start_u$$

$$V(u|v) = \frac{1 - \mathrm{e}^{-\alpha_s * duration_{u|v}}}{\alpha_s}$$

4.1.3 均摊旋律性

由于一条旋律在时间上可能是分散的，我们定义旋律 Q 在乐段 P 中的均摊旋律性 $AvgMelodyDeg(Q|P)$ ：

$$AvgMelodyDeg(Q|P) = \frac{\sum_{i \in [2, n-1], Q_i \in P} V(Q_i|Q_{i+1}) MelodyDeg(Q_i|Q_{i-1}, Q_{i+1})}{duration_P}$$

均摊旋律性是一个 $[0, 1)$ 的实数。注意上式中分母为乐段总长度。由于音符音量是衰减的，均摊旋律性接近 1 的程度也一定程度上取决于音符密度。

4.2 复合旋律分解

先只考虑一次分解，即分解旋律 Q ，使得 $TotalMelodyDeg(l_Q) + TotalMelodyDeg(r_Q)$ 最大。

4.2.1 动态规划

若我们按时间顺序依次考虑每个音符，则它对旋律性的贡献只与所在旋律上相邻的两个音符有关。

用 f_{Q_i,u_l,v_l,u_r,v_r} 表示按开始时刻顺序考虑到了音符 Q_i ，子旋律 l_Q 中的最后一个音符与倒数第二个音符分别为 u_l, v_l ，子旋律 r_Q 中的最后一个音符与倒数第二个音符分别 u_r, v_r ，在当前状态下的最优旋律性之和。只需要枚举将当前音符 Q_i 加入 l_Q 或是 r_Q 转移即可：

$$f_{Q_i,u_l,v_l,u_r,v_r} = \max \begin{cases} f_{Q_{i+1},Q_i,u_l,u_r,v_r} + V(u_l|Q_i)MelodyDeg(u_l|v_l,Q_i) \\ f_{Q_{i+1},u_l,v_l,Q_i,u_r} + V(u_r|Q_i)MelodyDeg(u_r|v_r,Q_i) \end{cases}$$

按照动态规划方法，可以求出所有状态的最优解。状态数似乎是比较多的。而在 f_{Q_i,u_l,v_l,u_r,v_r} 的 u_l, v_l, u_r, v_r 四个变量中，一定有两个是与 Q_{i-1}, Q_{i-2} 相同的。实际只有 $O(n^3)$ 级别的状态数。

4.2.2 逐层分解

由于上层的分解结果对下层的影响较小，我们直接采用逐层最优化的方式分解。

递归地将一个旋律 Q 按上述的动态规划方法分解为最优的子旋律 l_Q, r_Q 。若 $TotalMelodyDeg(l_Q)+TotalMelodyDeg(r_Q) \le TotalMelodyDeg(Q)$ ，则停止分解。

单次分解的时间复杂度为 $O(n^3)$ 。若乐谱中共有 N 个音符，最终可分为 M 个子旋律，则最坏的总时间复杂度为 $O(MN^3)$ 。

4.3 复合旋律增益

复合旋律结构中的每个子旋律可以按照单旋律算法单独完成力度增益，然后再进行合成。

设 $gain_{u|Q,P}$ 为单独考虑旋律 Q 时，在乐段 P 上局部对音符 u 的增益。对于每个乐段，最终一个音符的总增益为所有旋律按在该乐段上的均摊旋律性加权的平均：

$$gain_u = \sum_P \frac{\sum_Q AvgMelodyDeg(Q|P)gain_{u|Q,P}}{\sum_Q AvgMelodyDeg(Q|P)}$$

4.4 复合旋律合成

我们仍需要以某种方式对我们希望强调或减弱的子旋律作出适当的处理，使得子旋律间的层次是分明的。

4.4.1 旋律音高偏离度

在3.3.1节中我们提到，强弱走势与音高的标准差有一定联系。若我们单独考虑子旋律，也应该同样如此。同时，子旋律的音高也是有意义的，因为高音区的旋律是更倾向于被强调的。

先给出旋律 Q 在乐段 P 中的旋律音高平均值与旋律音高标准差的定义：

$$\mu_{Q|P} = \frac{\sum_{i,Q_i \in P} V(Q_i|Q_{i+1}) pitch_{Q_i}}{\sum_{i,Q_i \in P} V(Q_i|Q_{i+1})}$$

$$\sigma_{Q|P} = \sqrt{\frac{\sum_{i,Q_i \in P} V(Q_i|Q_{i+1})(pitch_{Q_i} - \mu_{Q|P})^2}{duration_P}}$$

综合旋律音高平均值与旋律音高标准差，考虑旋律 Q 在乐段 P 中的旋律音高偏离度 $PitchDeviation(Q|P)$：

$$PitchDeviation(Q|P) = \gamma_a \mu_{Q|P} + \gamma_d \sigma_{Q|P}$$

其中 γ_a, γ_d 为两个均衡常数，本实例中取 $\gamma_a = \gamma_d$。

当乐段 P 的长度小到一定程度时，旋律 Q 在 P 中的旋律音高偏离度将不再具有参考价值。因此，我们用上层乐段中历史旋律音高偏离度的最大值来定义旋律 Q 在 P 中的全局旋律音高偏离度 $GlobalDeviation(Q|P)$：

$$GlobalDeviation(Q|P) = \max_{P' \supseteq P} PitchDeviation(Q|P')$$

这一项标准可用于比较子旋律间的层次关系。

4.4.2 旋律合成补偿

自底向上考虑将 l_Q, r_Q 与旋律 Q 合成时，在每一个叶子乐段上的增益补偿。叶子乐段，即不存在子乐段的末层乐段。

考虑每一个叶子乐段 P_{leaf}，我们将对 $GlobalDeviation(l_Q|P_{leaf})$ 与 $GlobalDeviation(r_Q|P_{leaf})$ 中值较小的一个子旋律在 P_{leaf} 中的增益进行衰减，衰减量为两者之差。

如此，在增益补偿之后再计算出每个音符的总增益即可。

5 输出

完成力度增益的计算之后，还可以对它们整体进行一些处理，再对应到输出力度上。

5.1 增益均衡

我们一般不希望音符力度整体偏强或偏弱，而总体上的强弱差异适中即可。于是可以先进行均衡处理：

$$\mu = \frac{\sum_u V(u) gain_u}{\sum_u V(u)}$$

$$\sigma = \sqrt{\frac{\sum_u V(u)(gain_u - \mu)^2}{\sum_u V(u)}}$$

$$gain'_u = \frac{\sigma_{expected}}{\sigma}(gain_u - \mu)$$

其中 $\sigma_{expected}$ 可以设定为我们期望的最终增益的标准差。如果信任已经设定好的参数，这里也可以不作处理。

5.2 力度映射

最后，可用中心对称的标准S型曲线函数，将实数范围内的增益映射到输出力度上：

$$vel_u = \frac{127}{1 + \mathrm{e}^{-\gamma_o gain'_u}}$$

如果输出力度必须是一个整数，再将最终的 vel_u 作上取整处理即可。我们也完全可以对最终的力度映射函数做一些其它的修改。

对于最终输出的MIDI文件，我们可以直接播放试听，也可以输出到其它MIDI音源[60]中播放。为更好地适应不同音源，可能还需要继续调整音源中的力度映射的灵敏度曲线[61]。

6 总结

为钢琴演奏确定音符力度是一项艰难的任务。我们无法对其进行客观评价。本文中实现的算法，从一个全新的基于逻辑的角度，对该问题进行了初步的探究，并已经得到了可观的成果。

但是，从我们的结果中仍能看出很多问题。个别音符有不正常的强调或减弱，这可能是因为我们只根据音高趋势进行了旋律分解，而没有在音乐性上考虑，也可能需要额外输入乐谱的节拍信息。有时强弱变化不符合聆听者的预期，虽然可能确实有多种演奏方式，但仍需要作进一步分析。我将继续修正这些问题。

当音符数量较多时，由于本模型对算法精确性的要求不高，可改用近似算法分析乐段及旋律结构。可以确定的是时间复杂度不会约束本模型的应用。

[60]音源内部包含对声音的真实采样，用于将输入的音符信息转化为波形音频输出

[61]钢琴音源中大多可以调整dynamics参数，主要用于适应力度的敏感程度

我们还可以在本文的基础上做很多延伸的后续工作。如钢琴演奏中的速度变化模型，已经可以猜想，钢琴演奏中的速度变化与复合乐段结构有密切联系。对复合乐段结构进行和声分析，并加入创造性元素，有望实现自动钢琴伴奏。此外，为取得更好的应用环境，仍待实现一套五线谱读谱系统，实现自动读谱演奏的功能，同时可以结合五线谱上的演奏记号，优化自动力度系统中的强弱变化以及乐段与旋律的结构分析。

可能会继续发表后续研究。

参考文献

[1] Ramon Lopez de Mantaras and Josep Lluis Arcos, “AI and Music From Composition to Expressive Performance”

[2] Sebastian Flossmann, Maarten Grachten and Gerhard Widmer, “Expressive Performance Rendering with Probabilistic Model”

《基因组重构》命题报告

浙江省绍兴市第一中学 洪华敦

摘 要

字符串处理是信息学竞赛中的经典问题，对此也有非常多的经典算法。然而在以往的比赛中，大多数竞赛题目的对象都是单个串，或者多个串组成的字母树，问题也大多跟子串有关。于是笔者尝试往多串的子串方向思考，并在集训队互测中命了一道相关的题，希望能起到抛砖引玉的作用，吸引大家去挖掘跟多个串有关的字符串算法。

1 概述

字符串处理是传统信息学竞赛中比较重要的一个分支，然而在以往的题目中，对象都以单串和字母树为主。于是作者开始思考，能否能定义多个串的子串，并将一些对单串的子串的传统问题移植到这上面来。最终作者在集训队互测中命了一道统计多个串的本质不同子串个数的题，希望本题能起到抛砖引玉的作用，吸引大家去挖掘跟多个串有关的字符串算法。

作者围绕着命题，给出了三种解题思路，其中在算法一中，作者讲述了如何用字母树去优化复杂度较差的暴力，并且发现了暴力的瓶颈在于三个串的子串要一起枚举，于是作者挖掘了一些题目的性质，将三个串的子串一起枚举变成了枚举单个串的子串，大大降低了复杂度。并在算法二中给出了用可持久化字母树预处理来优化暴力的做法。在算法三中，作者放弃了结构简单的字母树，使用了后缀自动机，并最后给出了如何用动态树来对暴力枚举进行优化。

2 题目描述

生物学家小T最近在研究合成基因串的课题，所谓基因串就是字符集为$\{A, C, G, T\}$的字符串。

对于一次研究，对象是3个基因串X，Y，Z。小T会从X中选出一个任意连续子串$X1$，从Y中选出一个任意连续子串$Y1$，从Z中选出一个任意连续子串$Z1$，然后按顺序拼接它们得到$X1 + Y1 + Z1$，设$f(X, Y, Z)$为这样能组成多少不同的基因串。（注意以上的任意连续子串都可以是空串）

现在小T有3个长度为n的基因串A, B, C和Q次研究，每次研究由6个正整数Al, Ar, Bl, Br, Cl, Cr描述，表示她需要求$f(A[Al : Ar], B[Bl : Br], C[Cl : Cr])$[62]对一个大质数998244353取模的值。

2.1 输入格式

第一行两个整数n, Q。

接下来三行，每行一个长度为n的字符集为$\{A, C, G, T\}$的字符串，分别为题目描述中的基因组X, Y, Z。

接下来Q行，每行6个正整数$L_X, R_X, L_Y, R_Y, L_Z, R_Z$，描述一组实验。

2.2 输出格式

输出Q行，每行一个非负整数，表示该组询问的答案对998244353取模后的值。

2.3 样例输入1

```
2 1
AC
CC
AA
1 2 1 2 1 1
```

2.4 样例输出1

```
15
```

2.5 样例解释1

能组成的基因组有：

空串,A,AC,AA,ACA,ACCA,ACCCA,C,CC,

CCC,CA,CCA,CCCA,ACC,ACCC

2.6 样例输入2

```
3 1
```

[62] $S[l : r]$表示串S从下标l到下标r的子串，下标从1开始

ACG
GCA
TTT
1 3 1 3 2 1

2.7 样例输出2

35

2.8 样例解释2

如果$l > r$，则$S[l : r]$是空串，空串是空串的连续子串

2.9 数据范围

对于100%的数据，有$n, Q \leq 10^5$，$1 \leq Al, Ar, Bl, Br, Cl, Cr \leq n$。

定义*type*1为：对于所有询问有$Bl > Br$且$Cl > Cr$。

定义*type*2为：对于所有询问有$Cl > Cr$。

测试数据点编号	n的上限	Q的上限	上文的type限制类型
1	10	10	
2	20	20	
3	50	50	
4	100	100	
5	100	100	
6	500	500	
7	500	500	
8	100	100000	*type*1
9	250	100000	
10	3000	100000	
11	3000	100000	*type*1
12	3000	100000	*type*2
13	5000	5000	*type*1
14	5000	5000	*type*2
15	5000	5000	
16	50000	50000	*type*2
17	100000	100000	*type*1
18	50000	50000	
19	70000	70000	
20	100000	100000	

3 得分分布

在集训队互测中，有3人获得60分，1人获得35分，3人获得10分，5人获得0分。集训队中无人获得满分，说明国内选手还不是特别熟悉这一方面的知识，所以这方面的算法有很大的推广价值。

4 算法一

4.1 最简单的暴力

对于询问的三个串X，Y，Z，考虑采取朴素枚举的方法。

我们枚举X的任意连续子串$X1$，Y的任意连续子串$Y1$，Z的任意连续子串$Z1$，然后将$X1+Y1+Z1$插入*trie*中，最后统计下有几个串即可。

时间复杂度：$O(Qn^7)$

空间复杂度：$O(n^7)$

期望得分：5分

4.2 利用*hash*优化

算法一有一个缺陷，所有串都要插入到*trie*中，导致复杂度要多乘一个字符串长度。

我们可以换一种方法去重，考虑*hash*函数f，如果我们已知$f(X1)$，$f(Y1)$，$f(Z1)$，那我们就可以在$O(1)$时间内求出$f(X1+Y1+Z1)$。

于是我们可以用$O(n^2)$的时间复杂度预处理所有连续子串的*hash*值，然后枚举子串并用一个*map*对*hash*值进行去重。

时间复杂度：$O(Qn^6 \log n)$

空间复杂度：$O(n^6)$

期望得分：10分

5 算法二

5.1 最大划分

为了方便优化复杂度，我们定义一个串的划分：

定义 5.1. *在计算$f(X,Y,Z)$时，$(X1,Y1,Z1)$是串S的一个划分，当且仅当以下两个条件都满足：*

(1).：$S = X1+Y1+Z1$。

(2).：*$X1$是X的任意连续子串，$Y1$是Y的任意连续子串，$Z1$是Z的任意连续子串。*

下面我们举几个例子来更好地理解划分的定义：

例 1. 当询问$f(ACGC,TCT,CGTT)$时：

(AC,T,T)和(A,C,TT)都是$ACTT$的一个划分。

(ϵ, C, GT)[63]和$(\epsilon, \epsilon, CGT)$都是$CGT$的一个划分。

从直观上理解：一个字符串的划分实际上就是它被统计到时的构成方法，一个比较简单的想法是直接计算划分$(X1, Y1, Z1)$的个数作为答案。

但这个算法的错误性是显然的，因为每个串都有多种不同的划分，所以直接计算划分个数会重复计算答案。

于是我们需要引入一种每个串都有且仅有一个的划分方法。

定义 5.2. *在计算$f(X,Y,Z)$时，对于S的所有划分$(X1, Y1, Z1)$，定义$|X1|$最大的中$|Y1|$最大的划分为字符串S的最大划分。*

我们再举几个例子：

例 2. 当询问$f(ACGC, TCT, CGTT)$时：

(AC, T, T)是$ACTT$的最大划分。

(CG, T, ϵ)是CGT的最大划分。

显然对于一个字符串S，他的最大划分是唯一的，所以我们只要统计最大划分的个数就行了，这样每个串刚好被计算一次。

于是我们将问题转化成了，求满足以下两个条件的三元组$(X1, Y1, Z1)$的个数：

(1).$X1$是X的任意连续子串，$Y1$是Y的任意连续子串，$Z1$是Z的任意连续子串。

(2).$(X1, Y1, Z1)$是字符串$X1 + Y1 + Z1$的最大划分。

这个并不能直接计算，我们需要再挖掘一些最大划分的性质。

5.2 最大划分的性质

算法一和算法二的主要瓶颈在于要同时枚举三个串的子串，导致复杂度至少是$O(n^6)$起步的，我们可以通过挖掘最大划分的一些性质来将三个枚举分开来。

结论 1. 在询问$f(X,Y,Z)$时，对于字符串S的一个划分$(X1, Y1, Z1)$，它是串S的最大划分当且仅当以下两个条件都满足：

1. $Y1 + Z1 = \epsilon$或者$X1 + First(Y1 + Z1)$[64]不是X的任意连续子串。

2. $Z1 = \epsilon$或者$Y1 + First(Z1)$不是Y的任意连续子串。

证明. 考虑反证法：

假设$(X1, Y1, Z1)$是S的最大划分。

若它不满足条件1，则：

若$Y1 = \epsilon$，则$(X1 + First(Z1), Y1, Z1[2 : |Z1|])$[65]是$S$的一个更大的划分。

[63] ϵ表示空串，ϵ是任何字符串的连续子串

[64] $First(S)$表示S的第一个字符。

[65] $S[l : r]$表示S的下标从l至r的子串。

若$Y1 \neq \epsilon$，则$(X1 + First(Y1), Y1[2 : |Y1|], Z1)$是$S$的一个更大的划分。

若它不满足条件2，则$(X1, Y1 + First(Z1), Z1[2 : |Z1|])$是$S$的一个更大的划分。 □

5.3 最大划分的计数

由于空串比较繁琐，我们先假设我们统计的划分$(X1, Y1, Z1)$都满足$min(|X1|, |Y1|, |Z1|) > 0$，有空串的情况也类似，讨论一下即可。

由于字符集较小，考虑枚举$First(Y1)$和$First(Z1)$，由于没有了空串，现在我们只要满足以下条件即可。

(1). $X1 + First(Y1)$不是X的任意连续子串。

(2). $Y1 + First(Z1)$不是Y的任意连续子串。

于是我们将问题转化成了对于每个串的子问题：

定义 5.3. *$Str(S, u, v)$表示S有多少本质不同的子串T满足以下两个条件：*

(1). $T \neq \epsilon$且$First(T) = u$。

(2). $T + y$不是S的任意连续子串。

定义 5.4. *$Head(S, u)$表示S有多少本质不同的非空子串T满足$First(T) = u$。*

定义 5.5. *$Tail(S, v) = \sum_{u \in \{A,C,G,T\}} Str(S, u, v)$，即$S$有多少本质不同的非空子串$T$满足$T + v$不是$S$的任意连续子串。*

于是我们可以通过枚举$First(Y1)$和$First(Z1)$计算答案，答案如下：

$$\sum_{u \in \{A,C,G,T\}} \sum_{v \in \{A,C,G,T\}} Tail(X, u) * Str(Y, u, v) * Head(Z, v)$$

5.4 利用字母树暴力

通过挖掘最大划分的性质，我们将问题转化成了计算$Str(S, u, v)$和$Head(S, u)$，$Tail(S, u)$可以通过累加$Str(S, u, v)$算出来。

$Head(S, u)$可以简单利用*trie*树$O(n^2)$计算。

至于$Str(S, u, v)$，我们可以枚举S每个子串T，然后去重，顺便利用*trie*来判断$T + u$是不是S的一个子串，不是的话可以累加到$Str(S, First(T), u)$里。

时间复杂度：$O(Qn^3)$

空间复杂度：$O(n^2)$

期望得分：25分。

5.5 优化检查

观察一下算法三，可以发现瓶颈主要在于用$O(n^2)$时间枚举了子串T后还要花$O(n)$的时间检查$T+u$是否在S内。

我们可以先将T都插到字母树里，然后检查$T+u$是否在S内只要检查字母树上的结点里是否有u这条出边即可。

这样就可以$O(1)$检查。

时间复杂度：$O(Qn^2)$。

空间复杂度：$O(n^2)$。

期望得分：35分。

5.6 利用预处理进行优化

对于算法三和算法四，它们都要在每次询问时进行枚举，当Q特别大时复杂度很不理想（例如测试点8和测试点9）。

可以发现我们一次枚举计算所需要的时间是$O(n^2)$的。由于询问都是一开始给定的三个串A，B，C的子串，我么可以预先对每个子串$A[l:r]$计算。好$Str(A[l:r],u,v)$和$Head(A[l:r],u)$，然后每次询问就可以$O(1)$计算答案了。

时间复杂度：$O(n^4+Q)$。

空间复杂度：$O(n^2)$。

期望得分：40分。

5.7 可持久化字母树

对于算法五，它的瓶颈在于对每个子串都计算了一次，考虑利用连续子串的性质进行优化。

对于子串$A[l:r]$，它的子串集合相对于子串$A[l+1:r]$的子串集合只多了串$A[l:r]$的所有前缀。

令$Trie(S)$表示串S的字母树，相当于$Trie(A[l:r])$就等于$Trie(A[l+1:r])$加上串$A[l:r]$的所有前缀。

我们都知道字母树添加一个串S的所有前缀是$O(|S|)$的。

我们对字母树进行可持久化，这样复制一颗字母树是$O(1)$的，然后添加即可，在添加时统计答案的增量。

因为每个子串添加一次串，所以总的添加次数是$O(n^3)$的。

时间复杂度：$O(n^3+Q)$。

空间复杂度：$O(n^2)$。

期望得分：45分。

6 算法三

6.1 解决子问题限制1

对于部分测试点，我们发现它们满足限制1，也就是说每次询问的都是$f(X,\epsilon,\epsilon)$。

问题变成了：求X的本质不同的子串个数。

我们对X建立后缀自动机，为了下文方便讲述，这里定义一些符号：

定义 6.1. $right(p)$：后缀自动机结点p所表示的子串的右端点的集合。

$min(p)$：后缀自动机结点p所表示的长度最短的子串的长度。

$max(p)$：后缀自动机结点p所表示的长度最长的子串的长度。

$fail(p)$：后缀自动机结点p的$fail$结点。

$last(p)$：后缀自动机结点p的$right$集合里的最大值。

我们枚举X的后缀自动机中的所有结点p，它代表的串有$max(p)-min(p)+1$个，直接累加即可。

时间复杂度：$O(Qn)$。

空间复杂度：$O(n)$。

期望得分：50分。

6.2 利用预处理进行优化

考虑测试点11，Q比较大，算法七无法通过，我们考虑通过预处理来进行优化。

对于串$A[l:r]$，它的后缀自动机可以通过在$A[l:r-1]$后面添加一个字符得到，至于对答案的统计，在每个结点的max和min改变时重新局部计算即可。

全部预处理出来后，每次询问的复杂度就是$O(1)$的了。

由于利用增量法构建一个后缀的后缀自动机是$O(n)$的，所以总的时间复杂度是$O(n^2)$的。

时间复杂度：$O(n^2+Q)$。

空间复杂度：$O(n^2)$。

期望得分：55分。

6.3 解决子问题限制2

可以发现一些测试点有限制2，相当于每次询问的都是$f(X,Y,\epsilon)$。

于是我们只需要求$Head(Y,u)$和$Tail(X,u)$即可，不需要求$Str(S,u,v)$。

对于$Head(Y,u)$，我们可以把原串反过来，就变成了求最后一个字符是u的本质不同的子串个数，我们在枚举结点p时，可以记下$right(p)$的任意一个值，就可以知道这$max(p)-min(p)+1$个串的最后一个字符是什么了。

对于$Tail(X,u)$，我们可以通过后缀自动机的出边$O(1)$知道结点p所代表的字符串集合后面加上u后还是不是X的子串。

对X和Y构建后缀自动机后$O(n)$统计即可。

时间复杂度：$O(Qn)$。

空间复杂度：$O(n)$。

期望得分：60分。

6.4 利用预处理进行优化

我们可以利用类似算法八的预处理，预处理好所有区间的答案，询问时就可以$O(1)$询问了。

时间复杂度：$O(n^2+Q)$

空间复杂度：$O(n^2)$

期望得分：65分

6.5 $Str(S,u,v)$的一些性质

定义 6.2. 令$Cnt(S,u,v)$表示S有多少本质不同的子串T满足开头是u，结尾是v。

对于$Str(S,u,v)$，我们考虑计算$Head(S,u)-Str(S,u,v)$，这个值相当于开头为u的子串T中，满足$T+v$是S的子串的个数。

对于所有这样的子串T，对它们和$T+v$建立一个一一对应的映射，而$T+v$就是S的所有开头为u结尾为v的本质不同的子串。

于是我们可以得到：

$Str(S,u,v)=Head(S,u)-Cnt(S,u,v)$。

问题从计算$Str(S,u,v)$变成了计算$Cnt(S,u,v)$。

6.6 利用后缀自动机统计子串

对于询问$f(X,Y,Z)$，通过后缀自动机我们已经可以在$O(n)$的时间内求出$Head(S,u)$了，现在的问题只剩下求出$Cnt(S,u,v)$。

我们用一个数组$sum[x]$表示左端点在x的本质不同的字符串的个数，特别地，对于每个串，我们只在它出现的最后一个位置统计它。

对于后缀自动机的每个结点p，它相当于给$sum[last(p)-max(p)+1..last(p)-min(p)+1]$都贡献了1，我们直接区间加即可。

用线段树进行区间加的话复杂度还要多一个$\log n$，由于这是静态的，所以我们可以将序列差分后转为单点加。

算出$sum[x]$后，位置x对开头为$S[x]$的本质不同的字符串个数有$sum[x]$的贡献。

我们将$sum[x]$从记个数变成记一个数组，令$sum[x][y]$表示左端点在x的末尾为y的本质不同的字符串的个数，这样我们就可以计算出Cnt数组了。

时间复杂度：$O(Qn)$。

空间复杂度：$O(n)$。

期望得分：70分。

6.7 离线预处理

可以发现算法十一无法通过一些Q比较大的测试点，我们考虑用类似预处理的方法进行计算。

我们考虑枚举r，维护$sum[l][x][y]$表示$Cnt(A[l:r],x,y)$。

可以发现后缀自动机添加一个字符后对应的后缀树的变化，只有两种：

(1). 新增一个叶子结点。

(2). 在一条边中间插入一个新的结点。

对于操作2，显然对答案的贡献不变，这里不作考虑。

考虑操作1，新增了一个结点k后，它会改变它的所有祖先p的$last(p)$。

我们枚举k的所有祖先p，对于每个i满足$i \in [min(p),max(p)]$，原本l要满足$l \le last(p) - i + 1$才能包含这个串，现在只要满足$l \le last(k) - i + 1$就行了。

于是这相当于是对区间$[last(p) - i + 2, last(k) - i + 1]$区间加1，将序列差分后，相当于对$last(p) - i + 2$加1，对$last(k) - i + 2$减1。

然后就变成了一个区间加问题，统计l的答案时只要询问区间$[l,n]$的和即可，我们可以维护一个数组，对于每个r做完后搞一遍区间和。

那么我们怎么维护子串的开头字符是哪些呢，可以发现对$sum[x]$有贡献的所有子串的首字符都是$S[x-1]$，于是我们给每个$sum[x][S[x-1]]$带个权即可。

时间复杂度：$O(n^2 + Q)$。

空间复杂度：$O(n^2)$。

期望得分：75分。

6.8 利用动态树优化枚举

算法十二中主要有两个瓶颈，一个是每次加完后对静态数组求和，一个是每次加的时候枚举的祖先个数是$O(n)$的。对于前者，我们可以用可持久化线段树代替，现在考虑优化后者。

可以发现，每次添加一个叶子k，相当于枚举叶子k的每个祖先p，然后进行一次可持久化线段树区间加。之后将所有$last(p)$改为$last(k)$。

我们可以考虑对暴力加一点有理有据的常数优化，令$anc(x)$表示x最近的$last$与$last(x)$不同的祖先。

对于后缀树的一段链，他们的长度区间并起来后还是一个区间，我们可以把它们的贡献一起计算。

引理 6.1. *该算法一共会对线段树进行$O(n\log n)$次区间加。*

证明. 对于后缀树上的一条边，如果边两侧的点$last$相等，则将这条边设为实边，否则设为虚边。

对于后缀树上的一个结点p，显然$last$等于$last(p)$的儿子结点只有一个。

所以每个点最多只有两条出边是实边，进一步地可以发现每个实边的联通块都是一条直链。

考虑我们每次添加叶子k后进行的工作：枚举祖先里所有直链，对它们计算贡献，然后将叶子k和它的所有祖先通过实边连在一起。

观察后可以发现这就是动态树的$Access$操作，可以知道$Access$的均摊复杂度是$O(n\log n)$的，于是线段树的区间加次数也就只有$O(n\log n)$了。 □

具体实现时可以用经典的动态树算法进行实现。

时间复杂度：$O(n\log^2 n)$

空间复杂度：$O(n\log^2 n)$

期望得分：75 ~ 100分

6.9 离线优化空间

上述算法的空间十分差，可以发现我们的问题是离线的，所以可以采取离线算法，将线段树改为静态，空间复杂度降为$O(n)$，就可以通过此题。

时间复杂度：$O(n\log^2 n)$。

空间复杂度：$O(n)$。

期望得分：100分。

7 总结

本题是一道关于多个字符串的子串计数的字符串题，考察了选手的计数能力和数据结构能力。涉及的后缀自动机，后缀树，线段树，动态树都属于NOI选手的知识范围。然而得出有效的算法对选手的分析能力和复杂度计算能力有一定的要求。

作者从2016年一场$ACM-ICPC$区域赛的一道题目 [4]上学习到了这一类动态树维护后缀树的方法，然后结合了一些自己的想法命了本题。

对于测试数据点的前半部分，只要选手往最大划分这一方向思考，经过一定的分析都能拿到50分以上的分数；对于测试数据点的后半部分，选手需要对后缀数据结构有一定的了解，以及能巧妙地运用动态树进行复杂度优化。

最后的动态树算法是一种通用算法，对于一些字符串区间询问问题很有作用；前面的最大划分也是一种计数思想，对于多个串的拼接的问题很有作用。当问题涉及多串询问或者区间询问时，以上两种方法值得考虑。

希望本题对大家有所启发，进一步地认识后缀数据结构如何利用动态树缩减复杂度。

8 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，董烨华老师多年来给予的关心和指导。

感谢国家集训队教练张瑞喆和余林韵的指导。

感谢成都七中的杨景钦同学为本文审稿。

感谢对我有过帮助和启发的老师和同学。

感谢父母家人的支持和无微不至的关心，照顾。

参考文献

[1] 黄志翱，浅谈动态树的相关问题及简单拓展，2014 国家集训队论文。

[2] 王鉴浩，浅谈字符串匹配的几种方法，2015 国家集训队论文。

[3] 张天扬，后缀自动机及其应用，2015 国家集训队论文。

[4] String，2016 ACM/ICPC Asia Regional Qingdao Online