

All-in at the River

Standard Code Library

Shanghai Jiao Tong University

Boren Tan

Zonghan Yang

Shangfei Yang



Regentropfen sind meine Tränen
Wind ist mein Atem und mein Erzählung
Zweige und Blätter sind meine Hände
denn mein Körper ist in Wurzeln gehüllt

wenn die Jahreszeit des Tauens kommt
werde ich wach und singe ein Lied
das Vergissmeinnicht, das du mir gegeben hast
ist hier

Contents

1 数学

| | |
|--------------------------------------|----|
| 1.1 插值 | 2 |
| 1.1.1 牛顿插值 | 2 |
| 1.1.2 拉格朗日插值 | 2 |
| 1.2 多项式 | 2 |
| 1.2.1 FFT | 2 |
| 1.2.2 NTT | 2 |
| 1.2.3 任意模数卷积 | 3 |
| 1.2.4 多项式操作 | 4 |
| 1.2.5 更优秀的多项式多点求值 | 6 |
| 1.2.6 拉格朗日反演 | 7 |
| 1.2.7 半在线卷积 | 7 |
| 1.2.8 常系数齐次线性递推 $O(k \log k \log n)$ | 8 |
| 1.3 FWT快速沃尔什变换 | 9 |
| 1.4 单纯形 | 9 |
| 1.5 线性代数 | 10 |
| 1.5.1 行列式取模 | 10 |
| 1.5.2 线性基 | 10 |
| 1.5.3 线性代数知识 | 10 |
| 1.6 常见数列 | 10 |
| 1.6.1 伯努利数 | 10 |
| 1.6.2 分拆数 | 11 |
| 1.6.3 斯特林数 | 11 |

2 数论

| | |
|-------------------|----|
| 2.1 $O(n)$ 预处理逆元 | 11 |
| 2.2 杜教筛 | 11 |
| 2.3 线性筛 | 11 |
| 2.4 Miller-Rabin | 12 |
| 2.5 Pollard's Rho | 12 |
| 2.6 常用公式 | 13 |
| 2.6.1 莫比乌斯反演 | 13 |
| 2.6.2 其他常用公式 | 13 |

3 图论

| | |
|-------------------------|----|
| 3.1 最小生成树 | 13 |
| 3.1.1 Boruvka算法 | 13 |
| 3.1.2 动态最小生成树 | 13 |
| 3.1.3 Steiner Tree 斯坦纳树 | 15 |
| 3.2 最短路 | 15 |
| 3.2.1 k短路 | 15 |
| 3.3 仙人掌 | 16 |
| 3.3.1 仙人掌DP | 16 |
| 3.4 二分图 | 17 |
| 3.4.1 KM二分图最大权匹配 | 17 |
| 3.5 一般图匹配 | 18 |
| 3.5.1 高斯消元 | 18 |
| 3.5.2 带花树 | 19 |
| 3.5.3 带权带花树 | 20 |
| 3.6 最大流 | 21 |
| 3.6.1 Dinic | 21 |
| 3.6.2 ISAP | 22 |
| 3.6.3 HLPP最高标号预流推进 | 23 |
| 3.7 费用流 | 24 |
| 3.7.1 SPFA费用流 | 24 |

4 数据结构

| | |
|--------------|----|
| 4.1 线段树 | 24 |
| 4.1.1 非递归线段树 | 24 |
| 4.1.2 主席树 | 25 |
| 4.2 陈丹琦分治 | 25 |
| 4.3 Treap | 26 |
| 4.4 Splay | 27 |
| 4.5 树分治 | 27 |

| | |
|-------------------------------|----|
| 4.5.1 动态树分治 | 27 |
| 4.5.2 紫荆花之恋 | 28 |
| 4.6 LCT | 29 |
| 4.6.1 不换根(弹飞绵羊) | 29 |
| 4.6.2 换根/维护生成树 | 30 |
| 4.6.3 维护子树信息 | 31 |
| 4.6.4 模板题:动态QTREE4(询问树上相距最远点) | 33 |
| 4.7 虚树 | 35 |
| 4.8 长链剖分 | 36 |
| 4.9 梯子剖分 | 36 |
| 4.10 左偏树 | 36 |
| 4.11 常见根号思路 | 36 |

5 字符串

| | |
|-----------------|----|
| 5.1 KMP | 37 |
| 5.1.1 ex-KMP | 37 |
| 5.2 AC自动机 | 38 |
| 5.3 后缀数组 | 38 |
| 5.3.1 SA-IS | 38 |
| 5.3.2 SAMSA | 39 |
| 5.4 后缀自动机 | 40 |
| 5.5 回文树 | 40 |
| 5.5.1 广义回文树 | 40 |
| 5.6 Manacher马拉车 | 42 |
| 5.7 字符串原理 | 42 |

6 动态规划

| | |
|-------------------------|----|
| 6.1 决策单调性 $O(n \log n)$ | 43 |
|-------------------------|----|

7 Miscellaneous

| | |
|------------------|----|
| 7.1 $O(1)$ 快速乘 | 43 |
| 7.2 $O(n^2)$ 高精度 | 43 |
| 7.3 笛卡尔树 | 46 |
| 7.4 常用NTT素数及原根 | 47 |
| 7.5 xorshift | 47 |
| 7.6 枚举子集 | 47 |
| 7.7 STL | 47 |
| 7.7.1 vector | 47 |
| 7.7.2 list | 47 |
| 7.8 pb_ds | 47 |
| 7.8.1 哈希表 | 47 |
| 7.8.2 堆 | 48 |
| 7.8.3 平衡树 | 48 |
| 7.9 rope | 48 |
| 7.10 编译选项 | 48 |

8 注意事项

| | |
|---------------|----|
| 8.1 常见下毒手法 | 48 |
| 8.2 场外相关 | 48 |
| 8.3 做题策略与心态调节 | 49 |

1. 数学

1.1 插值

1.1.1 牛顿插值

牛顿插值的原理是二项式反演。

二项式反演：

$$f(n) = \sum_{k=0}^n \binom{n}{k} g(k) \Leftrightarrow g(n) = \sum_{k=0}^n (-1)^{n-k} \binom{n}{k} f(k)$$

可以用 e^x 和 e^{-x} 的麦克劳林展开式证明。

套用二项式反演的结论即可得到牛顿插值：

$$f(n) = \sum_{i=0}^k \binom{n}{i} r_i$$

$$r_i = \sum_{j=0}^i (-1)^{i-j} \binom{i}{j} f(j)$$

其中 k 表示 $f(n)$ 的最高次项系数。

实现时可以用 k 次差分替代右边的式子：

```
1 for (int i = 0; i <= k; i++)
2   r[i] = f(i);
3 for (int j = 0; j < k; j++)
4   for (int i = k; i > j; i--)
5     r[i] -= r[i - 1];
```

注意到预处理 r_i 的式子满足卷积形式,必要时可以用FFT优化至 $O(k \log k)$ 预处理。

1.1.2 拉格朗日插值

$$f(x) = \sum_i f(x_i) \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

1.2 多项式

1.2.1 FFT

```
1 // 使用时一定要注意double的精度是否足够(极限大概是10 ^
   // 14)
2
3 const double pi = acos((double)-1.0);
4
5 // 手写复数类
6 // 支持加减乘三种运算
7 // += 运算符如果用的不多可以不重载
8 struct Complex {
9     double a, b; // 由于long double精度和double几乎相同,
   // 通常没有必用long double
10
11     Complex(double a = 0.0, double b = 0.0) : a(a), b(b)
   // {}
12
13     Complex operator + (const Complex &x) const {
14         return Complex(a + x.a, b + x.b);
15     }
16
17     Complex operator - (const Complex &x) const {
18         return Complex(a - x.a, b - x.b);
19     }
20
21     Complex operator * (const Complex &x) const {
22         return Complex(a * x.a - b * x.b, a * x.b + b *
   // x.a);
23     }
24 }
```

```
25 Complex &operator += (const Complex &x) {
26     return *this = *this + x;
27 }
28 w[maxn], w_inv[maxn];
29
30 // FFT初始化 O(n)
31 // 需要调用sin, cos函数
32 void FFT_init(int n) {
33     for (int i = 0; i < n; i++) // 根据单位根的旋转性质可
   // 以节省计算单位根逆元的时间
34         w[i] = w_inv[n - i - 1] = Complex(cos(2 * pi / n
   // * i), sin(2 * pi / n * i));
35     // 当然不存单位根也可以, 只不过在FFT次数较多时很可能
   // 会增大常数
36 }
37
38 // FFT主过程 O(n \log n)
39 void FFT(Complex *A, int n, int tp) {
40     for (int i = 1, j = 0, k; i < n - 1; i++) {
41         k = n;
42         do
43             j ^= (k >>= 1);
44         while (j < k);
45
46         if (i < j)
47             swap(A[i], A[j]);
48     }
49
50     for (int k = 2; k <= n; k *= 2)
51         for (int i = 0; i < n; i += k)
52             for (int j = 0; j < k * 2; j++) {
53                 Complex a = A[i + j], b = (tp > 0 ? w :
   // w_inv)[n / k * j] * A[i + j + (k /
   // 2)];
54
55                 A[i + j] = a + b;
56                 A[i + j + k / 2] = a - b;
57             }
58
59     if (tp < 0)
60         for (int i = 0; i < n; i++)
61             A[i].a /= n;
62 }
```

1.2.2 NTT

```
1 constexpr int p = 998244353, g = 3; // p为模数, g为p的任
   // 意一个原根
2
3 void NTT(int *A, int n, int tp) { // n为变换长度,
   // tp为1或-1, 表示正/逆变换
4     for (int i = 1, j = 0, k; i < n - 1; i++) { // O(n)旋
   // 转算法, 原理是模拟加1
5         k = n;
6         do
7             j ^= (k >>= 1);
8         while (j < k);
9
10        if (i < j)
11            swap(A[i], A[j]);
12    }
13
14    for (int k = 2; k <= n; k <= 1) {
15        int wn = qpow(g, (tp > 0 ? (p - 1) / k : (p - 1)
   // * (long long)(p - 2) % (p - 1)));
16        for (int i = 0; i < n; i += k) {
17            int w = 1;
18            for (int j = 0; j < (k >> 1); j++, w = (long
   // * w % p){
```

```

19     int a = A[i + j], b = (long long)w * A[i
    ↪ + j + (k >> 1)] % p;
20     A[i + j] = (a + b) % p;
21     A[i + j + (k >> 1)] = (a - b + p) % p;
22 } // 更好的写法是预处理单位根的次幂
23 }
24 }
25
26 if (tp < 0) {
27     int inv = qpow(n, p - 2); // 如果能预处理逆元更好
28     for (int i = 0; i < n; i++)
29         A[i] = (long long)A[i] * inv % p;
30 }
31 }

```

```

40     for (int i = 0; i < N; i++)
41         ans[i] = (long long)C[i] * D[i] % p;
42
43     NTT(ans, N, -1, p);
44 }
45
46 // 中国剩余定理  $O(1)$ 
47 // 由于直接合并会爆Long Long, 采用神奇的方法合并
48 // 需要调用 $O(1)$ 快速乘
49 inline int China(int a0, int a1, int a2) {
50     long long A = (mul((long long)a0 * m1_inv, m[1], M) +
    ↪ mul((long long)a1 * m0_inv, m[0], M)) % M;
51     int k = ((a2 - A) % m[2] + m[2]) % m[2] * M_inv %
    ↪ m[2];
52     return (k % Mod * (M % Mod) % Mod + A % Mod) % Mod;
53 }

```

1.2.3 任意模数卷积

任意模数卷积有两种比较naive的做法, 三模数NTT和拆系数FFT. 一般来说后者常数比前者小一些.

但卷积答案不超过 10^{18} 的时候可以改用双模数NTT, 比FFT是要快的.

三模数NTT

原理是选取三个乘积大于结果的NTT模数, 最后中国剩余定理合并.

```

1 //以下三模数NTT, 原理是选取三个乘积大于结果的NTT模数,
  ↪ 最后中国剩余定理合并
2 //以对23333333(不是质数)取模为例
3 constexpr int maxn = 262200, Mod = 23333333, g = 3, m[] =
  ↪ {998244353, 1004535809, 1045430273}, m0_inv =
  ↪ 669690699, m1_inv = 332747959, M_inv = 942377029; //
  ↪ 这三个模数最小原根都是3
4 constexpr long long M = (long long)m[0] * m[1];
5
6 // 主函数(当然更多时候包装一下比较好)
7 // 用来卷积的是A和B
8 // 需要调用mul
9 int n, N = 1, A[maxn], B[maxn], C[maxn], D[maxn], ans[3]
  ↪ [maxn];
10 int main() {
11     scanf("%d", &n);
12
13     while (N < n * 2)
14         N *= 2;
15
16     for (int i = 0; i < n; i++)
17         scanf("%d", &A[i]);
18     for (int i = 0; i < n; i++)
19         scanf("%d", &B[i]);
20
21     for (int i = 0; i < 3; i++)
22         mul(m[i], ans[i]);
23
24     for (int i = 0; i < n; i++)
25         printf("%d ", China(ans[0][i], ans[1][i], ans[2]
    ↪ [i]));
26
27     return 0;
28 }
29
30 // mul  $O(n \log n)$ 
31 // 包装了模NTT模数的卷积
32 // 需要调用NTT
33 void mul(int p, int *ans) {
34     copy(A, A + N, C);
35     copy(B, B + N, D);
36
37     NTT(C, N, 1, p);
38     NTT(D, N, 1, p);
39

```

拆系数FFT

原理是选一个数 M , 把每一项改写成 $aM + b$ 的形式再分别相乘.

```

1 constexpr int maxn = 262200, p = 23333333, M = 4830; //
  ↪ M取值要使得结果不超过 $10^{14}$ 
2
3 // 需要开的数组
4 struct Complex {
5     // 内容略
6 } w[maxn], w_inv[maxn], A[maxn], B[maxn], C[maxn],
  ↪ D[maxn], F[maxn], G[maxn], H[maxn];
7
8 // 主函数(当然更多时候包装一下比较好)
9 // 需要调用FFT初始化, FFT
10 int main() {
11     scanf("%d", &n);
12
13     int N = 1;
14     while (N < n * 2)
15         N *= 2;
16
17     for (int i = 0, x; i < n; i++) {
18         scanf("%d", &x);
19         A[i] = x / M;
20         B[i] = x % M;
21     }
22
23     for (int i = 0, x; i < n; i++) {
24         scanf("%d", &x);
25         C[i] = x / M;
26         D[i] = x % M;
27     }
28
29     FFT_init(N);
30
31     FFT(A, N, 1);
32     FFT(B, N, 1);
33     FFT(C, N, 1);
34     FFT(D, N, 1);
35
36     for (int i = 0; i < N; i++) {
37         F[i] = A[i] * C[i];
38         G[i] = A[i] * D[i] + B[i] * C[i];
39         H[i] = B[i] * D[i];
40     }
41
42     FFT(F, N, -1);
43     FFT(G, N, -1);
44     FFT(H, N, -1);
45
46     for (int i = 0; i < n; i++)

```

```

47     printf("%d ", (int)((M * M * ((long long)(F[i].a
    ↪ + 0.5) % p) % p + M * ((long long)(G[i].a +
    ↪ 0.5) % p) % p + (long long)(H[i].a + 0.5) %
    ↪ p) % p));
48
49     return 0;
50 }

```

1.2.4 多项式操作

```

1 // A为输入, C为输出, n为所需长度且必须是2^k
2 // 多项式求逆, 要求A常数项不为0
3 void get_inv(int *A, int *C, int n) {
4     static int B[maxn];
5
6     memset(C, 0, sizeof(int) * (n * 2));
7     C[0] = qpow(A[0], p - 2); // 一般常数项都是1, 直接赋值
    ↪ 为1就可以
8
9     for (int k = 2; k <= n; k <= 1) {
10         memcpy(B, A, sizeof(int) * k);
11         memset(B + k, 0, sizeof(int) * k);
12
13         NTT(B, k * 2, 1);
14         NTT(C, k * 2, 1);
15
16         for (int i = 0; i < k * 2; i++) {
17             C[i] = (2 - (long long)B[i] * C[i]) % p *
    ↪ C[i] % p;
18             if (C[i] < 0)
19                 C[i] += p;
20         }
21
22         NTT(C, k * 2, -1);
23
24         memset(C + k, 0, sizeof(int) * k);
25     }
26 }
27
28 // 开根
29 void get_sqrt(int *A, int *C, int n) {
30     static int B[maxn], D[maxn];
31
32     memset(C, 0, sizeof(int) * (n * 2));
33     C[0] = 1; // 如果不是1就要考虑二次剩余
34
35     for (int k = 2; k <= n; k *= 2) {
36         memcpy(B, A, sizeof(int) * k);
37         memset(B + k, 0, sizeof(int) * k);
38
39         get_inv(C, D, k);
40
41         NTT(B, k * 2, 1);
42         NTT(D, k * 2, 1);
43
44         for (int i = 0; i < k * 2; i++)
45             B[i] = (long long)B[i] * D[i] % p;
46
47         NTT(B, k * 2, -1);
48
49         for (int i = 0; i < k; i++)
50             C[i] = (long long)(C[i] + B[i]) * inv_2 %
    ↪ p; // inv_2是2的逆元
51     }
52 }
53
54 // 求导
55 void get_derivative(int *A, int *C, int n) {
56     for (int i = 1; i < n; i++)

```

```

57     C[i - 1] = (long long)A[i] * i % p;
58
59     C[n - 1] = 0;
60 }
61
62 // 不定积分, 最好预处理逆元
63 void get_integrate(int *A, int *C, int n) {
64     for (int i = 1; i < n; i++)
65         C[i] = (long long)A[i - 1] * qpow(i, p - 2) % p;
66
67     C[0] = 0; // 不定积分没有常数项
68 }
69
70 // 多项式ln, 要求A常数项不为0
71 void get_ln(int *A, int *C, int n) { // 通常情况下A常数项
    ↪ 都是1
72     static int B[maxn];
73
74     get_derivative(A, B, n);
75     memset(B + n, 0, sizeof(int) * n);
76
77     get_inv(A, C, n);
78
79     NTT(B, n * 2, 1);
80     NTT(C, n * 2, 1);
81
82     for (int i = 0; i < n * 2; i++)
83         B[i] = (long long)B[i] * C[i] % p;
84
85     NTT(B, n * 2, -1);
86
87     get_integrate(B, C, n);
88
89     memset(C + n, 0, sizeof(int) * n);
90 }
91
92 // 多项式exp, 要求A没有常数项
93 // 常数很大且总代码较长, 一般来说最好替换为分治FFT
94 // 分治FFT依据: 设 $G(x) = \exp F(x)$ , 则有  $g_i = \sum_{k=1}^i f_{i-k} * k * g_k$ 
    ↪  $\wedge \{i-1\} f_{i-k} * k * g_k$ 
95 void get_exp(int *A, int *C, int n) {
96     static int B[maxn];
97
98     memset(C, 0, sizeof(int) * (n * 2));
99     C[0] = 1;
100
101     for (int k = 2; k <= n; k <= 1) {
102         get_ln(C, B, k);
103
104         for (int i = 0; i < k; i++) {
105             B[i] = A[i] - B[i];
106             if (B[i] < 0)
107                 B[i] += p;
108         }
109         (B[0] += B[0]) %= p;
110
111         NTT(B, k * 2, 1);
112         NTT(C, k * 2, 1);
113
114         for (int i = 0; i < k * 2; i++)
115             C[i] = (long long)C[i] * B[i] % p;
116
117         NTT(C, k * 2, -1);
118
119         memset(C + k, 0, sizeof(int) * k);
120     }
121 }
122
123 // 多项式k次幂, 在A常数项不为1时需要转化

```

```

124 // 常数较大且总代码较长, 在时间要求不高时最好替换为暴力
    ↪ 快速幂
125 void get_pow(int *A, int *C, int n, int k) {
126     static int B[maxn];
127
128     get_ln(A, B, n);
129
130     for (int i = 0; i < n; i++)
131         B[i] = (long long)B[i] * k % p;
132
133     get_exp(B, C, n);
134 }
135
136 // 多项式除法,  $A / B$ , 结果输出在C
137 // A的次数为n, B的次数为m
138 void get_div(int *A, int *B, int *C, int n, int m) {
139     static int f[maxn], g[maxn], gi[maxn];
140
141     if (n < m) {
142         memset(C, 0, sizeof(int) * m);
143         return;
144     }
145
146     int N = 1;
147     while (N < (n - m + 1))
148         N <= 1;
149
150     memset(f, 0, sizeof(int) * N * 2);
151     memset(g, 0, sizeof(int) * N * 2);
152     // memset(gi, 0, sizeof(int) * N);
153
154     for (int i = 0; i < n - m + 1; i++)
155         f[i] = A[n - i - 1];
156     for (int i = 0; i < m && i < n - m + 1; i++)
157         g[i] = B[m - i - 1];
158
159     get_inv(g, gi, N);
160
161     for (int i = n - m + 1; i < N; i++)
162         gi[i] = 0;
163
164     NTT(f, N * 2, 1);
165     NTT(gi, N * 2, 1);
166
167     for (int i = 0; i < N * 2; i++)
168         f[i] = (long long)f[i] * gi[i] % p;
169
170     NTT(f, N * 2, -1);
171
172     for (int i = 0; i < n - m + 1; i++)
173         C[i] = f[n - m - i];
174 }
175
176 // 多项式取模, 余数输出到C, 商输出到D
177 void get_mod(int *A, int *B, int *C, int *D, int n, int
    ↪ m) {
178     static int b[maxn], d[maxn];
179
180     if (n < m) {
181         memcpy(C, A, sizeof(int) * n);
182
183         if (D)
184             memset(D, 0, sizeof(int) * m);
185
186         return;
187     }
188
189     get_div(A, B, d, n, m);
190
191     if (D) { // D是商, 可以选择不要
192         for (int i = 0; i < n - m + 1; i++)
193             D[i] = d[i];
194     }
195
196     int N = 1;
197     while (N < n)
198         N *= 2;
199
200     memcpy(b, B, sizeof(int) * m);
201
202     NTT(b, N, 1);
203     NTT(d, N, 1);
204
205     for (int i = 0; i < N; i++)
206         b[i] = (long long)d[i] * b[i] % p;
207
208     NTT(b, N, -1);
209
210     for (int i = 0; i < m - 1; i++)
211         C[i] = (A[i] - b[i] + p) % p;
212
213     memset(b, 0, sizeof(int) * N);
214     memset(d, 0, sizeof(int) * N);
215 }
216
217 // 多点求值要用的数组
218 int q[maxn], ans[maxn]; // q是要代入的各个系数, ans是求出
    ↪ 的值
219 int tg[25][maxn * 2], tf[25][maxn]; // 辅助数组, tg是预处
    ↪ 理乘积,
220 // tf是项数越来越少的f, tf[0]就是原来的函数
221
222 void pretreat(int l, int r, int k) { // 多点求值预处理
223     static int A[maxn], B[maxn];
224
225     int *g = tg[k] + l * 2;
226
227     if (r - l + 1 <= 200) {
228         g[0] = 1;
229
230         for (int i = l; i <= r; i++) {
231             for (int j = i - l + 1; j; j--) {
232                 g[j] = (g[j - 1] - (long long)g[j] *
                    ↪ q[i]) % p;
233                 if (g[j] < 0)
234                     g[j] += p;
235             }
236             g[0] = (long long)g[0] * (p - q[i]) % p;
237         }
238
239         return;
240     }
241
242     int mid = (l + r) / 2;
243
244     pretreat(l, mid, k + 1);
245     pretreat(mid + 1, r, k + 1);
246
247     if (!k)
248         return;
249
250     int N = 1;
251     while (N <= r - l + 1)
252         N *= 2;
253
254     int *gl = tg[k + 1] + l * 2, *gr = tg[k + 1] + (mid +
        ↪ 1) * 2;
255
256     memset(A, 0, sizeof(int) * N);

```

```

257 memset(B, 0, sizeof(int) * N);
258
259 memcpy(A, gl, sizeof(int) * (mid - 1 + 2));
260 memcpy(B, gr, sizeof(int) * (r - mid + 1));
261
262 NTT(A, N, 1);
263 NTT(B, N, 1);
264
265 for (int i = 0; i < N; i++)
266     A[i] = (long long)A[i] * B[i] % p;
267
268 NTT(A, N, -1);
269
270 for (int i = 0; i <= r - 1 + 1; i++)
271     g[i] = A[i];
272 }
273
274 void solve(int l, int r, int k) { // 多项式多点求值主过程
275     int *f = tf[k];
276
277     if (r - l + 1 <= 200) {
278         for (int i = 1; i <= r; i++) {
279             int x = q[i];
280
281             for (int j = r - 1; ~j; j--)
282                 ans[i] = ((long long)ans[i] * x + f[j]) %
283                     ↪ p;
284
285             return;
286         }
287
288         int mid = (l + r) / 2;
289         int *ff = tf[k + 1], *gl = tg[k + 1] + 1 * 2, *gr =
290             ↪ tg[k + 1] + (mid + 1) * 2;
291
292         get_mod(f, gl, ff, NULL, r - l + 1, mid - l + 2);
293         solve(l, mid, k + 1);
294
295         memset(gl, 0, sizeof(int) * (mid - l + 2));
296         memset(ff, 0, sizeof(int) * (mid - l + 1));
297
298         get_mod(f, gr, ff, NULL, r - l + 1, r - mid + 1);
299         solve(mid + 1, r, k + 1);
300
301         memset(gr, 0, sizeof(int) * (r - mid + 1));
302         memset(ff, 0, sizeof(int) * (r - mid));
303     }
304
305     //  $f < x^n$ ,  $m$  个询问, 询问是  $\theta$ -based, 当然改成  $1$ -based 也很简
306     ↪ 单
307     void get_value(int *f, int *x, int *a, int n, int m) {
308         if (m <= n)
309             m = n + 1;
310         if (n < m - 1)
311             n = m - 1; // 补零方便处理
312
313         memcpy(tf[0], f, sizeof(int) * n);
314         memcpy(q, x, sizeof(int) * m);
315
316         pretreat(0, m - 1, 0);
317         solve(0, m - 1, 0);
318
319         if (a) // 如果  $a$  是  $NULL$ , 代表不复制答案, 直接用  $ans$  数组
320             memcpy(a, ans, sizeof(int) * m);
321     }

```

1.2.5 更优秀的多项式多点求值

这个做法不需要写求逆和取模, 但是神乎其技, 完全搞不懂原理
清空和复制之类的地方容易抄错, 抄的时候要注意

```

1 int q[maxn], ans[maxn]; //  $q$  是要代入的各个系数,  $ans$  是求出
   ↪ 的值
2 int tg[25][maxn * 2], tf[25][maxn]; // 辅助数组,  $tg$  是预处理乘积,
   ↪ 理乘积,
3 //  $tf$  是项数越来越少的  $f$ ,  $tf[0]$  就是原来的函数
4
5 void pretreat(int l, int r, int k) { // 预处理
6     static int A[maxn], B[maxn];
7
8     int *g = tg[k] + 1 * 2;
9
10    if (r - l + 1 <= 1) {
11        g[0] = 1;
12
13        for (int i = 1; i <= r; i++) {
14            for (int j = i - 1 + 1; j; j--) {
15                g[j] = (g[j - 1] - (long long)g[j] *
16                    ↪ q[i]) % p;
17                if (g[j] < 0)
18                    g[j] += p;
19            }
20            g[0] = (long long)g[0] * (p - q[i]) % p;
21        }
22
23        reverse(g, g + r - l + 2);
24
25        return;
26    }
27
28    int mid = (l + r) / 2;
29
30    pretreat(l, mid, k + 1);
31    pretreat(mid + 1, r, k + 1);
32
33    int N = 1;
34    while (N <= r - l + 1)
35        N *= 2;
36
37    int *gl = tg[k + 1] + 1 * 2, *gr = tg[k + 1] + (mid +
38        ↪ 1) * 2;
39
40    memset(A, 0, sizeof(int) * N);
41    memset(B, 0, sizeof(int) * N);
42
43    memcpy(A, gl, sizeof(int) * (mid - l + 2));
44    memcpy(B, gr, sizeof(int) * (r - mid + 1));
45
46    NTT(A, N, 1);
47    NTT(B, N, 1);
48
49    for (int i = 0; i < N; i++)
50        A[i] = (long long)A[i] * B[i] % p;
51
52    NTT(A, N, -1);
53
54    for (int i = 0; i <= r - l + 1; i++)
55        g[i] = A[i];
56
57    void solve(int l, int r, int k) { // 主过程
58        static int a[maxn], b[maxn];
59
60        int *f = tf[k];
61
62        if (l == r) {
63            ans[l] = f[0];

```



```

63     return;
64 }
65
66 int mid = (l + r) / 2;
67 int *ff = tf[k + 1], *gl = tg[k + 1] + 1 * 2, *gr =
    ↪ tg[k + 1] + (mid + 1) * 2;
68
69 int N = 1;
70 while (N < r - l + 2)
71     N *= 2;
72
73 memcpy(a, f, sizeof(int) * (r - l + 2));
74 memcpy(b, gr, sizeof(int) * (r - mid + 1));
75 reverse(b, b + r - mid + 1);
76
77 NTT(a, N, 1);
78 NTT(b, N, 1);
79 for (int i = 0; i < N; i++)
80     b[i] = (long long)a[i] * b[i] % p;
81
82 reverse(b + 1, b + N);
83 NTT(b, N, 1);
84 int n_inv = qpow(N, p - 2);
85 for (int i = 0; i < N; i++)
86     b[i] = (long long)b[i] * n_inv % p;
87
88 for (int i = 0; i < mid - l + 2; i++)
89     ff[i] = b[i + r - mid];
90
91 memset(a, 0, sizeof(int) * N);
92 memset(b, 0, sizeof(int) * N);
93
94 solve(l, mid, k + 1);
95
96 memset(ff, 0, sizeof(int) * (mid - l + 2));
97
98 memcpy(a, f, sizeof(int) * (r - l + 2));
99 memcpy(b, gl, sizeof(int) * (mid - l + 2));
100 reverse(b, b + mid - l + 2);
101
102 NTT(a, N, 1);
103 NTT(b, N, 1);
104 for (int i = 0; i < N; i++)
105     b[i] = (long long)a[i] * b[i] % p;
106
107 reverse(b + 1, b + N);
108 NTT(b, N, 1);
109 for (int i = 0; i < N; i++)
110     b[i] = (long long)b[i] * n_inv % p;
111
112 for (int i = 0; i < r - mid + 1; i++)
113     ff[i] = b[i + mid - l + 1];
114
115 memset(a, 0, sizeof(int) * N);
116 memset(b, 0, sizeof(int) * N);
117
118 solve(mid + 1, r, k + 1);
119
120 memset(gl, 0, sizeof(int) * (mid - l + 2));
121 memset(gr, 0, sizeof(int) * (r - mid + 1));
122 memset(ff, 0, sizeof(int) * (r - mid + 1));
123 }
124
125 // f < x^n, m个询问, 0-based
126 void get_value(int *f, int *x, int *a, int n, int m) {
127     static int c[maxn], d[maxn];
128
129     if (m <= n)
130         m = n + 1;

```

```

131     if (n < m - 1)
132         n = m - 1; // 补零
133
134     memcpy(q, x, sizeof(int) * m);
135
136     pretreat(0, m - 1, 0);
137
138     int N = 1;
139     while (N < m)
140         N *= 2;
141
142     get_inv(tg[0], c, N);
143
144     fill(c + m, c + N, 0);
145     reverse(c, c + m);
146
147     memcpy(d, f, sizeof(int) * m);
148
149     NTT(c, N * 2, 1);
150     NTT(d, N * 2, 1);
151     for (int i = 0; i < N * 2; i++)
152         c[i] = (long long)c[i] * d[i] % p;
153     NTT(c, N * 2, -1);
154
155     for (int i = 0; i < m; i++)
156         tf[0][i] = c[i + n];
157
158     solve(0, m - 1, 0);
159
160     if (a) // 如果a是NULL, 代表不复制答案, 直接用ans数组
161         memcpy(a, ans, sizeof(int) * m);
162 }

```

1.2.6 拉格朗日反演

如果 $f(x)$ 与 $g(x)$ 互为复合逆 则有

$$[x^n]g(x) = \frac{1}{n}[x^{n-1}] \left(\frac{x}{f(x)} \right)^n$$

$$[x^n]h(g(x)) = \frac{1}{n}[x^{n-1}]h'(x) \left(\frac{x}{f(x)} \right)^n$$

1.2.7 半在线卷积

```

1 void solve(int l, int r) {
2     if (r <= m)
3         return;
4
5     if (r - l == 1) {
6         if (l == m)
7             f[l] = a[m];
8         else
9             f[l] = (long long)f[l] * inv[l - m] % p;
10
11     for (int i = l, t = (long long)l * f[l] % p; i <=
        ↪ n; i += 1)
12         g[i] = (g[i] + t) % p;
13
14     return;
15 }
16
17 int mid = (l + r) / 2;
18
19 solve(l, mid);
20
21 if (l == 0) {
22     for (int i = 1; i < mid; i++) {
23         A[i] = f[i];
24         B[i] = (c[i] + g[i]) % p;

```



```

25     }
26     NTT(A, r, 1);
27     NTT(B, r, 1);
28     for (int i = 0; i < r; i++)
29         A[i] = (long long)A[i] * B[i] % p;
30     NTT(A, r, -1);
31
32     for (int i = mid; i < r; i++)
33         f[i] = (f[i] + A[i]) % p;
34 }
35 else {
36     for (int i = 0; i < r - 1; i++)
37         A[i] = f[i];
38     for (int i = 1; i < mid; i++)
39         B[i - 1] = (c[i] + g[i]) % p;
40     NTT(A, r - 1, 1);
41     NTT(B, r - 1, 1);
42     for (int i = 0; i < r - 1; i++)
43         A[i] = (long long)A[i] * B[i] % p;
44     NTT(A, r - 1, -1);
45
46     for (int i = mid; i < r; i++)
47         f[i] = (f[i] + A[i - 1]) % p;
48
49     memset(A, 0, sizeof(int) * (r - 1));
50     memset(B, 0, sizeof(int) * (r - 1));
51
52     for (int i = 1; i < mid; i++)
53         A[i - 1] = f[i];
54     for (int i = 0; i < r - 1; i++)
55         B[i] = (c[i] + g[i]) % p;
56     NTT(A, r - 1, 1);
57     NTT(B, r - 1, 1);
58     for (int i = 0; i < r - 1; i++)
59         A[i] = (long long)A[i] * B[i] % p;
60     NTT(A, r - 1, -1);
61
62     for (int i = mid; i < r; i++)
63         f[i] = (f[i] + A[i - 1]) % p;
64 }
65
66     memset(A, 0, sizeof(int) * (r - 1));
67     memset(B, 0, sizeof(int) * (r - 1));
68
69     solve(mid, r);
70 }

```

1.2.8 常系数齐次线性递推 $O(k \log k \log n)$

如果只有一次这个操作可以像代码里一样加上一个只求一次逆的优化, 否则就乖乖每次做完整的除法和取模

```

1 // 多项式取模, 余数输出到C, 商输出到D
2 void get_mod(int *A, int *B, int *C, int *D, int n, int
   ↪ m) {
3     static int b[maxn], d[maxn];
4     static bool flag = false;
5
6     if (n < m) {
7         memcpy(C, A, sizeof(int) * n);
8
9         if (D)
10             memset(D, 0, sizeof(int) * m);
11
12         return;
13     }
14
15     get_div(A, B, d, n, m);
16

```

```

17     if (D) { // D是商, 可以选择不要
18         for (int i = 0; i < n - m + 1; i++)
19             D[i] = d[i];
20     }
21
22     int N = 1;
23     while (N < n)
24         N *= 2;
25
26     if (!flag) {
27         memcpy(b, B, sizeof(int) * m);
28         NTT(b, N, 1);
29
30         flag = true;
31     }
32
33     NTT(d, N, 1);
34
35     for (int i = 0; i < N; i++)
36         d[i] = (long long)d[i] * b[i] % p;
37
38     NTT(d, N, -1);
39
40     for (int i = 0; i < m - 1; i++)
41         C[i] = (A[i] - d[i] + p) % p;
42
43     // memset(b, 0, sizeof(int) * N);
44     memset(d, 0, sizeof(int) * N);
45 }
46
47 // g < x^n, f是输出答案的数组
48 void pow_mod(long long k, int *g, int n, int *f) {
49     static int a[maxn], t[maxn];
50
51     memset(f, 0, sizeof(int) * (n * 2));
52
53     f[0] = a[1] = 1;
54
55     int N = 1;
56     while (N < n * 2 - 1)
57         N *= 2;
58
59     while (k) {
60         NTT(a, N, 1);
61
62         if (k & 1) {
63             memcpy(t, f, sizeof(int) * N);
64
65             NTT(t, N, 1);
66             for (int i = 0; i < N; i++)
67                 t[i] = (long long)t[i] * a[i] % p;
68             NTT(t, N, -1);
69
70             get_mod(t, g, f, NULL, n * 2 - 1, n);
71         }
72
73         for (int i = 0; i < N; i++)
74             a[i] = (long long)a[i] * a[i] % p;
75         NTT(a, N, -1);
76
77         memcpy(t, a, sizeof(int) * (n * 2 - 1));
78         get_mod(t, g, a, NULL, n * 2 - 1, n);
79         fill(a + n - 1, a + N, 0);
80
81         k >>= 1;
82     }
83
84     memset(a, 0, sizeof(int) * (n * 2));
85 }

```

```

86 //  $f_n = \sum_{i=1}^m f_{n-i} a_i$ 
87 //  $f$  是  $0 \sim m-1$  项的初值
88 int linear_recurrence(long long n, int m, int *f, int *a)
89 {
90     static int g[maxn], c[maxn];
91
92     memset(g, 0, sizeof(int) * (m * 2 + 1));
93
94     for (int i = 0; i < m; i++)
95         g[i] = (p - a[m - i]) % p;
96     g[m] = 1;
97
98     pow_mod(n, g, m + 1, c);
99
100     int ans = 0;
101     for (int i = 0; i < m; i++)
102         ans = (ans + (long long)c[i] * f[i]) % p;
103
104     return ans;
105 }

```

1.3 FWT快速沃尔什变换

```

1 // 注意FWT常数比较小, 这点与FFT/NTT不同
2 // 以下代码均以模质数情况为例, 其中 $n$ 为变换长度,  $tp$ 表示
   ↪ 正/逆变换
3
4 // 按位或版本
5 void FWT_or(int *A, int n, int tp) {
6     for (int k = 2; k <= n; k *= 2)
7         for (int i = 0; i < n; i += k)
8             for (int j = 0; j < k / 2; j++) {
9                 if (tp > 0)
10                     A[i + j + k / 2] = (A[i + j + k / 2]
11                                     ↪ + A[i + j]) % p;
12                 else
13                     A[i + j + k / 2] = (A[i + j + k / 2]
14                                     ↪ - A[i + j] + p) % p;
15             }
16 }
17
18 // 按位与版本
19 void FWT_and(int *A, int n, int tp) {
20     for (int k = 2; k <= n; k *= 2)
21         for (int i = 0; i < n; i += k)
22             for (int j = 0; j < k / 2; j++) {
23                 if (tp > 0)
24                     A[i + j] = (A[i + j] + A[i + j + k / 2]) % p;
25                 else
26                     A[i + j] = (A[i + j] - A[i + j + k / 2]
27                             ↪ + p) % p;
28             }
29 }
30
31 // 按位异或版本
32 void FWT_xor(int *A, int n, int tp) {
33     for (int k = 2; k <= n; k *= 2)
34         for (int i = 0; i < n; i += k)
35             for (int j = 0; j < k / 2; j++) {
36                 int a = A[i + j], b = A[i + j + k / 2];
37                 A[i + j] = (a + b) % p;
38                 A[i + j + k / 2] = (a - b + p) % p;
39             }
40
41     if (tp < 0) {
42         int inv = qpow(n % p, p - 2); //  $n$  的逆元, 在不取
43         ↪ 模时需要用每层除以2代替

```

```

40         for (int i = 0; i < n; i++)
41             A[i] = A[i] * inv % p;
42     }
43 }

```

1.4 单纯形

```

1 const double eps = 1e-10;
2
3 double A[maxn][maxn], x[maxn];
4 int n, m, t, id[maxn * 2];
5
6 // 方便起见, 这里附上主函数
7 int main() {
8     scanf("%d%d%d", &n, &m, &t);
9
10     for (int i = 1; i <= n; i++) {
11         scanf("%lf", &A[0][i]);
12         id[i] = i;
13     }
14
15     for (int i = 1; i <= m; i++) {
16         for (int j = 1; j <= n; j++)
17             scanf("%lf", &A[i][j]);
18
19         scanf("%lf", &A[i][0]);
20     }
21
22     if (!inititalize())
23         printf("Infeasible"); // 无解
24     else if (!simplex())
25         printf("Unbounded"); // 最优解无限大
26
27     else {
28         printf("%.15lf\n", -A[0][0]);
29         if (t) {
30             for (int i = 1; i <= m; i++)
31                 x[id[i + n]] = A[i][0];
32             for (int i = 1; i <= n; i++)
33                 printf("%.15lf ", x[i]);
34         }
35     }
36     return 0;
37 }
38
39 // 初始化
40 // 对于初始解可行的问题, 可以把初始化省略掉
41 bool inititalize() {
42     while (true) {
43         double t = 0.0;
44         int l = 0, e = 0;
45
46         for (int i = 1; i <= m; i++)
47             if (A[i][0] + eps < t) {
48                 t = A[i][0];
49                 l = i;
50             }
51
52         if (!l)
53             return true;
54
55         for (int i = 1; i <= n; i++)
56             if (A[l][i] < -eps && (!e || id[i] < id[e]))
57                 e = i;
58
59         if (!e)
60             return false;
61     }

```

```

62     pivot(l, e);
63 }
64 }
65 //求解
66 bool simplex() {
67     while (true) {
68         int l = 0, e = 0;
69         for (int i = 1; i <= n; i++)
70             if (A[0][i] > eps && (!e || id[i] < id[e]))
71                 e = i;
72
73         if (!e)
74             return true;
75
76         double t = 1e50;
77         for (int i = 1; i <= m; i++)
78             if (A[i][e] > eps && A[i][0] / A[i][e] < t) {
79                 l = i;
80                 t = A[i][0] / A[i][e];
81             }
82
83         if (!l)
84             return false;
85
86         pivot(l, e);
87     }
88 }
89
90 //转轴操作, 本质是在凸包上沿着一条棱移动
91 void pivot(int l, int e) {
92     swap(id[e], id[n + 1]);
93     double t = A[l][e];
94     A[l][e] = 1.0;
95
96     for (int i = 0; i <= n; i++)
97         A[l][i] /= t;
98
99     for (int i = 0; i <= m; i++)
100         if (i != l) {
101             t = A[i][e];
102             A[i][e] = 0.0;
103             for (int j = 0; j <= n; j++)
104                 A[i][j] -= t * A[l][j];
105         }
106 }
107

```

1.5 线性代数

1.5.1 行列式取模

```

1  int p;
2
3  int Gauss(int A[maxn][maxn], int n) {
4      int det = 1;
5
6      for (int i = 1; i <= n; i++) {
7          for (int j = i + 1; j <= n; j++)
8              while (A[j][i]) {
9                  int t = (p - A[i][i] / A[j][i]) % p;
10                 for (int k = i; k <= n; k++)
11                     A[i][k] = (A[i][k] + (long long)A[j][k]
12                        * t) % p;
13
14                 swap(A[i], A[j]);
15                 det = (p - det) % p; // 交换一次之后行列
16                 式取负
17             }
18         }
19     }
20 }

```

```

17         if (!A[i][i])
18             return 0;
19
20         det = (long long)det * A[i][i] % p;
21     }
22
23     return det;
24 }

```

1.5.2 线性基

```

1  void add(unsigned long long x) {
2      for (int i = 63; i >= 0; i--)
3          if (x >> i & 1) {
4              if (b[i])
5                  x ^= b[i];
6              else {
7                  b[i] = x;
8
9                  for (int j = i - 1; j >= 0; j--)
10                     if (b[j] && (b[i] >> j & 1))
11                         b[i] ^= b[j];
12
13                  for (int j = i + 1; j < 64; j++)
14                     if (b[j] >> i & 1)
15                         b[j] ^= b[i];
16
17                  break;
18              }
19          }
20 }

```

1.5.3 线性代数知识

行列式:

$$\det A = \sum_{\sigma} \operatorname{sgn}(\sigma) \prod_i a_{i, \sigma_i}$$

逆矩阵:

$$B = A^{-1} \iff AB = 1$$

代数余子式:

$$M_{i,j} = (-1)^{(i+j)} \det A - \{i,j\}$$

也就是A去掉一行一列之后的行列式

同时我们有

$$M = \frac{A^{-1}}{\det A}$$

1.6 常见数列

1.6.1 伯努利数

$$B(x) = \sum_{i \geq 0} \frac{B_i x^i}{i!} = \frac{x}{e^x - 1}$$

$$B_n = [n = 0] - \sum_{i=0}^{n-1} \binom{n}{i} \frac{B_i}{n-k+1}$$

$$\sum_{i=0}^n \binom{n+1}{i} B_i = 0$$

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

1.6.2 分拆数

```

1 int b = sqrt(n);
2 ans[0] = tmp[0] = 1;
3
4 for (int i = 1; i <= b; ++i) {
5     for (int rep = 0; rep < 2; ++rep)
6         for (int j = i; j <= n - i * i; ++j)
7             add(tmp[j], tmp[j - i]);
8
9     for (int j = i * i; j <= n; ++j)
10        add(ans[j], tmp[j - i * i]);
11 }

```

1.6.3 斯特林数

第一类斯特林数

$[n_k]$ 表示 n 个元素划分成 k 个轮换的方案数。

求同一行: 分治FFT $O(n \log^2 n)$

求同一列: 用一个轮换的指数生成函数做 k 次幂

$$\sum_{n=0}^{\infty} \begin{bmatrix} n \\ k \end{bmatrix} \frac{x^n}{n!} = \frac{(\ln(1-x))^k}{k!}$$

第二类斯特林数

$\{n_k\}$ 表示 n 个元素划分成 k 个子集的方案数。

求一个: 容斥, 狗都会做

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \frac{1}{k!} \sum_{i=0}^k (-1)^i \binom{k}{i} (k-i)^n$$

求同一行: FFT, 狗都会做

求同一列: 指数生成函数

$$\sum_{n=0}^{\infty} \left\{ \begin{matrix} n \\ k \end{matrix} \right\} \frac{x^n}{n!} = \frac{(e^x - 1)^k}{k!}$$

普通生成函数

$$\sum_{n=0}^{\infty} \left\{ \begin{matrix} n \\ k \end{matrix} \right\} x^n = x^k \left(\prod_{i=1}^k (1 - ix) \right)^{-1}$$

2. 数论

2.1 $O(n)$ 预处理逆元

```

1 // 要求p为质数
2
3 inv[0] = inv[1] = 1;
4 for (int i = 2; i <= n; i++)
5     inv[i] = (long long)(p - (p / i)) * inv[p % i] % p;
6     // i ^ -1 = -(p / i) * (p % i) ^ -1

```

2.2 杜教筛

```

1 // 用于求可以用狄利克雷卷积构造出好求和的东西的函数的前
2 // 缀和(有点绕)
3 // 有些题只要求n <= 10 ^ 9, 这时就没必要开long long了, 但
4 // 记得乘法时强转
5 // 常量/全局变量/数组定义
6 const int maxn = 5000005, table_size = 5000000, p =
7     1000000007, inv_2 = (p + 1) / 2;

```

```

6 bool notp[maxn];
7 int prime[maxn / 20], phi[maxn], tbl[100005];
8 // tbl用来顶替哈希表, 其实开到n ^ {1 / 3}就够了, 不过保
9 // 险起见开成sqrt n比较好
10 long long N;
11 // 主函数前面加上这么一句
12 memset(tbl, -1, sizeof(tbl));
13 // 线性筛预处理部分略去
14 // 杜教筛主过程 总计O(n ^ {2 / 3})
15 // 递归调用自身
16 // 递推式还需具体情况具体分析, 这里以求欧拉函数前缀和(mod
17 // 10 ^ 9 + 7)为例
18 int S(long long n) {
19     if (n <= table_size)
20         return phi[n];
21     else if (~tbl[N / n])
22         return tbl[N / n];
23     // 原理: n除以所有可能的数的结果一定互不相同
24
25     int ans = 0;
26     for (long long i = 2, last; i <= n; i = last + 1) {
27         last = n / (n / i);
28         ans = (ans + (last - i + 1) % p * S(n / i)) % p;
29         // 如果n是int范围的话记得强转
30     }
31
32     ans = (n % p * ((n + 1) % p) % p * inv_2 - ans + p) %
33         p; // 同上
34     return tbl[N / n] = ans;
35 }

```

2.3 线性筛

```

1 // 此代码以计算约数之和函数\sigma_1(对10^9+7取模)为例
2 // 适用于任何f(p^k)便于计算的积性函数
3 constexpr int p = 1000000007;
4
5 int prime[maxn / 10], sigma_one[maxn], f[maxn], g[maxn];
6 // f: 除掉最小质因子后剩下的部分
7 // g: 最小质因子的幂次, 在f(p^k)比较复杂时很有用,
8 // 但f(p^k)可以递推时就可以省略了
9 // 这里没有记录最小质因子, 但根据线性筛的性质, 每个合数
10 // 只会被它最小的质因子筛掉
11 bool notp[maxn]; // 顾名思义
12
13 void get_table(int n) {
14     sigma_one[1] = 1; // 积性函数必有f(1) = 1
15     for (int i = 2; i <= n; i++) {
16         if (!notp[i]) { // 质数情况
17             prime[++prime[0]] = i;
18             sigma_one[i] = i + 1;
19             f[i] = g[i] = 1;
20         }
21
22         for (int j = 1; j <= prime[0] && i * prime[j] <=
23             n; j++) {
24             notp[i * prime[j]] = true;
25
26             if (i % prime[j]) { // 加入一个新的质因子, 这
27                 // 种情况很简单
28                 sigma_one[i * prime[j]] = (long
29                     long)sigma_one[i] * (prime[j] + 1) %
30                     p;
31                 f[i * prime[j]] = i;
32                 g[i * prime[j]] = 1;
33             }
34         }
35     }
36 }

```

```

28     else { // 再加入一次最小质因子, 需要再进行分
        ↳ 类讨论
29         f[i * prime[j]] = f[i];
30         g[i * prime[j]] = g[i] + 1;
31         // 对于 $f(p^k)$ 可以直接递推的函数, 这里的判
        ↳ 断可以改成
32         //  $i / \text{prime}[j] \% \text{prime}[j] \neq 0$ , 这样可以
        ↳ 省下 $f[]$ 的空间,
33         // 但常数很可能会稍大一些

34
35         if (f[i] == 1) // 质数的幂次, 这
        ↳ 里 $\text{sigma}_1$ 可以递推
36         sigma_one[i * prime[j]] =
        ↳ (sigma_one[i] + i * prime[j]) %
        ↳ p;
37         // 对于更一般的情况, 可以借助 $g[]$ 计
        ↳ 算 $f(p^k)$ 
38         else sigma_one[i * prime[j]] = // 否则直
        ↳ 接利用积性, 两半乘起来
39         (long long)sigma_one[i * prime[j] /
        ↳ f[i]] * sigma_one[f[i]] % p;
40         break;
41     }
42 }
43 }
44 }

```

2.4 Miller-Rabin

```

1 // 复杂度可以认为是常数
2
3 // 封装好的函数体
4 // 需要调用check
5 bool Miller_Rabin(long long n) {
6     if (n == 1)
7         return false;
8     if (n == 2)
9         return true;
10    if (n % 2 == 0)
11        return false;
12
13    for (int i : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
        ↳ 37}) {
14        if (i > n)
15            break;
16        if (!check(n, i))
17            return false;
18    }
19
20    return true;
21 }
22
23 // 用一个数检测
24 // 需要调用Long Long快速幂和 $O(1)$ 快速乘
25 bool check(long long n, long long b) { // b: base
26     long long a = n - 1;
27     int k = 0;
28
29     while (a % 2 == 0) {
30         a /= 2;
31         k++;
32     }
33
34     long long t = qpow(b, a, n); // 这里的快速幂函数需要
        ↳ 写 $O(1)$ 快速乘
35     if (t == 1 || t == n - 1)
36         return true;
37
38     while (k--) {

```

```

39         t = mul(t, t, n); // mul是 $O(1)$ 快速乘函数
40         if (t == n - 1)
41             return true;
42     }
43
44     return false;
45 }

```

2.5 Pollard's Rho

```

1 // 注意, 虽然Pollard's Rho的理论复杂度是 $O(n^{1/4})$ 的,
2 // 但实际跑起来比较慢, 一般用于做Long Long范围内的质因数
        ↳ 分解
3
4
5 // 封装好的函数体
6 // 需要调用solve
7 void factorize(long long n, vector<long long> &v) { //
        ↳ v用于存分解出来的质因子, 重复的会放多个
8     for (int i : {2, 3, 5, 7, 11, 13, 17, 19})
9         while (n % i == 0) {
10             v.push_back(i);
11             n /= i;
12         }
13
14     solve(n, v);
15     sort(v.begin(), v.end()); // 从小到大排序后返回
16 }
17
18 // 递归过程
19 // 需要调用Pollard's Rho主过程, 同时递归调用自身
20 void solve(long long n, vector<long long> &v) {
21     if (n == 1)
22         return;
23
24     long long p;
25     do
26         p = Pollards_Rho(n);
27     while (!p); // p是任意一个非平凡因子
28
29     if (p == n) {
30         v.push_back(p); // 说明n本身就是质数
31         return;
32     }
33
34     solve(p, v); // 递归分解两半
35     solve(n / p, v);
36 }
37
38 // Pollard's Rho主过程
39 // 需要使用Miller-Rabin作为子算法
40 // 同时需要调用 $O(1)$ 快速乘和gcd函数
41 long long Pollards_Rho(long long n) {
42     // assert(n > 1);
43
44     if (Miller_Rabin(n))
45         return n;
46
47     long long c = rand() % (n - 2) + 1, i = 1, k = 2, x =
        ↳ rand() % (n - 3) + 2, u = 2; // 注意这里rand函数
        ↳ 需要重定义一下
48     while (true) {
49         i++;
50         x = (mul(x, x, n) + c) % n; // mul是 $O(1)$ 快速乘函
        ↳ 数
51
52         long long g = gcd((u - x + n) % n, n);
53         if (g > 1 && g < n)
54             return g;

```

```

55 |         if (u == x)
56 |             return 0; // 失败, 需要重新调用
57 |
58 |         if (i == k) {
59 |             u = x;
60 |             k *= 2;
61 |         }
62 |     }
63 | }
64 |

```

2.6 常用公式

2.6.1 莫比乌斯反演

$$f(n) = \sum_{d|n} g(d) \Leftrightarrow g(n) = \sum_{d|n} \mu\left(\frac{n}{d}\right) f(d)$$

$$f(d) = \sum_{d|k} g(k) \Leftrightarrow g(d) = \sum_{d|k} \mu\left(\frac{k}{d}\right) f(k)$$

2.6.2 其他常用公式

$$\mu * I = e \quad (e(n) = [n = 1])$$

$$\varphi * I = id$$

$$\mu * id = \varphi$$

$$\sigma_0 = I * I, \sigma_1 = id * I, \sigma_k = id^{k-1} * I$$

$$\sum_{i=1}^n [(i, n) = 1] i = n \frac{\varphi(n) + e(n)}{2}$$

$$\sum_{i=1}^n \sum_{j=1}^i [(i, j) = d] = S_{\varphi}\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$$

$$\sum_{i=1}^n \sum_{j=1}^m [(i, j) = d] = \sum_{d|k} \mu\left(\frac{k}{d}\right) \left\lfloor \frac{n}{k} \right\rfloor \left\lfloor \frac{m}{k} \right\rfloor$$

3. 图论

3.1 最小生成树

3.1.1 Boruvka算法

思想: 每次选择连接每个连通块的最小边, 把连通块缩起来.

每次连通块个数至少减半, 所以迭代 $O(\log n)$ 次即可得到最小生成树.

一种比较简单的实现方法: 每次迭代遍历所有边, 用并查集维护连通性和每个连通块的最小边权.

应用: 最小异或生成树

3.1.2 动态最小生成树

```

1 // 动态最小生成树的离线算法比较容易, 而在线算法通常极为复
  // 杂
2 // 一个跑得比较快的离线做法是对时间分治, 在每层分治时找出
  // 一定在/不在MST上的边, 只带着不确定边继续递归
3 // 简单起见, 找确定边的过程用Kruskal算法实现, 过程中的两种
  // 重要操作如下:
4 // - Reduction: 待修改边标为+INF, 跑MST后把非树边删掉, 减少
  //   无用边
5 // - Contraction: 待修改边标为-INF, 跑MST后缩除待修改边之
  //   外的所有MST边, 计算必须边
6 // 每轮分治需要Reduction-Contraction, 借此减少不确定边, 从
  // 而保证复杂度

```

```

7 // 复杂度证明: 假设当前区间有k条待修改边, n和m表示点数和边
  // 数, 那么最坏情况下R-C的效果为(n, m) -> (n, n + k - 1)
  // -> (k + 1, 2k)
8
9
10 // 全局结构体与数组定义
11 struct edge { // 边的定义
12     int u, v, w, id; // id表示边在原图中的编号
13     bool vis; // 在Kruskal时用, 记录这条边是否是树边
14     bool operator < (const edge &e) const { return w <
        e.w; }
15 } e[20][maxn], t[maxn]; // 为了便于回滚, 在每层分治存一个
  // 副本
16
17
18 // 用于存储修改的结构体, 表示第id条边的权值从u修改为v
19 struct A {
20     int id, u, v;
21 } a[maxn];
22
23
24 int id[20][maxn]; // 每条边在当前图中的编号
25 int p[maxn], size[maxn], stk[maxn], top; // p和size是并查
  // 集数组, stk是用来撤销的栈
26 int n, m, q; // 点数, 边数, 修改数
27
28
29 // 方便起见, 附上可能需要用到的预处理代码
30 for (int i = 1; i <= n; i++) { // 并查集初始化
31     p[i] = i;
32     size[i] = 1;
33 }
34
35 for (int i = 1; i <= m; i++) { // 读入与预标号
36     scanf("%d%d%d", &e[0][i].u, &e[0][i].v, &e[0][i].w);
37     e[0][i].id = i;
38     id[0][i] = i;
39 }
40
41 for (int i = 1; i <= q; i++) { // 预处理出调用数组
42     scanf("%d%d", &a[i].id, &a[i].v);
43     a[i].u = e[0][a[i].id].w;
44     e[0][a[i].id].w = a[i].v;
45 }
46
47 for(int i = q; i; i--)
48     e[0][a[i].id].w = a[i].u;
49
50 CDQ(1, q, 0, m, 0); // 这是调用方法
51
52
53 // 分治主过程 O(nLog^2n)
54 // 需要调用Reduction和Contraction
55 void CDQ(int l, int r, int d, int m, long long ans) { //
  // CDQ分治
56     if (l == r) { // 区间长度已减小到1, 输出答案, 退出
57         e[d][id[d][a[l].id]].w = a[l].v;
58         printf("%lld\n", ans + Kruskal(m, e[d]));
59         e[d][id[d][a[l].id]].w = a[l].u;
60         return;
61     }
62
63     int tmp = top;
64
65     Reduction(l, r, d, m);
66     ans += Contraction(l, r, d, m); // R-C
67
68     int mid = (l + r) / 2;
69
70     copy(e[d] + 1, e[d] + m + 1, e[d + 1] + 1);
71     for (int i = 1; i <= m; i++)

```



```

72     id[d + 1][e[d][i].id] = i; // 准备好下一层要用的
    ↪ 数组
73
74     CDQ(l, mid, d + 1, m, ans);
75
76     for (int i = l; i <= mid; i++)
77     |   e[d][id[d][a[i].id]].w = a[i].v; // 进行左边的修
    ↪ 改
78
79     copy(e[d] + 1, e[d] + m + 1, e[d + 1] + 1);
80     for (int i = 1; i <= m; i++)
81     |   id[d + 1][e[d][i].id] = i; // 重新准备下一层要用
    ↪ 的数组
82
83     CDQ(mid + 1, r, d + 1, m, ans);
84
85     for (int i = top; i > tmp; i--)
86     |   cut(stk[i]); // 撤销所有操作
87     top = tmp;
88 }
89
90 // Reduction(减少无用边):待修改边标为+INF,跑MST后把非树
    ↪ 边删掉,减少无用边
91 // 需要调用Kruskal
92 void Reduction(int l, int r, int d, int &m) {
93     for (int i = l; i <= r; i++)
94     |   e[d][id[d][a[i].id]].w = INF; // 待修改的边标为INF
95
96     Kruskal(m, e[d]);
97
98     copy(e[d] + 1, e[d] + m + 1, t + 1);
99
100
101     int cnt = 0;
102     for (int i = 1; i <= m; i++)
103     |   if (t[i].w == INF || t[i].vis) { // 非树边扔掉
104     |   |   id[d][t[i].id] = ++cnt; // 给边重新编号
105     |   |   e[d][cnt] = t[i];
106     |   }
107
108     for (int i = r; i >= l; i--)
109     |   e[d][id[d][a[i].id]].w = a[i].u; // 把待修改的边
    ↪ 改回去
110
111     m = cnt;
112 }
113
114 // Contraction(缩必须边):待修改边标为-INF,跑MST后缩除待
    ↪ 修改边之外的所有树边
115 // 返回缩掉的边的总权值
116 // 需要调用Kruskal
117 long long Contraction(int l, int r, int d, int &m) {
118     long long ans = 0;
119
120     for (int i = l; i <= r; i++)
121     |   e[d][id[d][a[i].id]].w = -INF; // 待修改边标
    ↪ 为-INF
122
123
124     Kruskal(m, e[d]);
125     copy(e[d] + 1, e[d] + m + 1, t + 1);
126
127     int cnt = 0;
128     for (int i = 1; i <= m; i++) {
129
130     |   if (t[i].w != -INF && t[i].vis) { // 必须边
131     |   |   ans += t[i].w;
132     |   |   mergeset(t[i].u, t[i].v);
133     |   }
134     |   else { // 不确定边
135
136     |   |   id[d][t[i].id] = ++cnt;
137     |   |   e[d][cnt] = t[i];
138     |   }
139
140     for (int i = r; i >= l; i--) {
141     |   e[d][id[d][a[i].id]].w = a[i].u; // 把待修改的边
    ↪ 改回去
142     |   e[d][id[d][a[i].id]].vis = false;
143     }
144
145     m = cnt;
146
147     return ans;
148 }
149
150 // Kruskal算法 O(mlogn)
151 // 方便起见,这里直接沿用进行过缩点的并查集,在过程结束后
    ↪ 撤销即可
152 long long Kruskal(int m, edge *e) {
153     int tmp = top;
154     long long ans = 0;
155
156     sort(e + 1, e + m + 1); // 比较函数在结构体中定义过了
157
158     for (int i = 1; i <= m; i++) {
159     |   if (findroot(e[i].u) != findroot(e[i].v)) {
160     |   |   e[i].vis = true;
161     |   |   ans += e[i].w;
162     |   |   mergeset(e[i].u, e[i].v);
163     |   }
164     |   else
165     |   |   e[i].vis = false;
166     }
167
168     for (int i = top; i > tmp; i--)
169     |   cut(stk[i]); // 撤销所有操作
170     top = tmp;
171
172     return ans;
173 }
174
175 // 以下是并查集相关函数
176 int findroot(int x) { // 因为需要撤销,不写路径压缩
177     while (p[x] != x)
178     |   x = p[x];
179
180     return x;
181 }
182
183 void mergeset(int x, int y) { // 按size合并,如果想跑得更快
    ↪ 就写一个按秩合并
184     x = findroot(x); // 但是按秩合并要再开一个栈记录合并
    ↪ 之前的秩
185     y = findroot(y);
186
187     if (x == y)
188     |   return;
189
190     if (size[x] > size[y])
191     |   swap(x, y);
192
193     p[x] = y;
194     size[y] += size[x];
195     stk[++top] = x;
196 }
197
198 void cut(int x) { // 并查集撤销

```



```

201     int y = x;
202
203     do
204         size[y = p[y]] -= size[x];
205     while (p[y] != y);
206
207     p[x] = x;
208 }

```

3.1.3 Steiner Tree 斯坦纳树

问题: 一张图上有 k 个关键点, 求让关键点两两连通的最小生成树

做法: 状压DP, $f_{i,S}$ 表示以 i 号点为树根, i 与 S 中的点连通的最小边权和

转移有两种:

1. 枚举子集:

$$f_{i,S} = \min_{T \subset S} \{f_{i,T} + f_{i,S \setminus T}\}$$

2. 新加一条边:

$$f_{i,S} = \min_{(i,j) \in E} \{f_{j,S} + w_{i,j}\}$$

第一种直接枚举子集DP就行了, 第二种可以用SPFA或者Dijkstra松弛(显然负边一开始全选就行了, 所以只需要处理非负边).

复杂度 $O(n3^k + 2^k m \log n)$.

3.2 最短路

3.2.1 k短路

```

1 // 注意这是个多项式算法, 在k比较大时很有优势, 但k比较小
  // 时最好还是用A*
2 // DAG和有环的情况都可以, 有重边或自环也无所谓, 但不能有
  // 零环
3 // 以下代码以Dijkstra + 可持久化左偏树为例
4
5 constexpr int maxn = 1005, maxe = 10005, maxm = maxe *
  // 30; // 点数, 边数, 左偏树结点数
6
7 // 结构体定义
8 struct A { // 用来求最短路
9     int x, d;
10
11     A(int x, int d) : x(x), d(d) {}
12
13     bool operator < (const A &a) const {
14         return d > a.d;
15     }
16 };
17
18 struct node { // 左偏树结点
19     int w, i, d; // i: 最后一条边的编号 d: 左偏树附加信息
20     node *lc, *rc;
21
22     node() {}
23
24     node(int w, int i) : w(w), i(i), d(0) {}
25
26     void refresh(){
27         d = rc -> d + 1;
28     }
29 } null[maxm], *ptr = null, *root[maxn];
30
31 struct B { // 维护答案用
32     int x, w; // x是结点编号, w表示之前已经产生的权值
33     node *rt; // 这个答案对应的堆顶, 注意可能不等于任何一
  // 个结点的堆

```

```

34     B(int x, node *rt, int w) : x(x), w(w), rt(rt) {}
35
36     bool operator < (const B &a) const {
37         return w + rt -> w > a.w + a.rt -> w;
38     }
39 };
40
41 // 全局变量和数组定义
42 vector<int> G[maxn], W[maxn], id[maxn]; // 最开始要存反向
  // 图, 然后把G清空作为儿子列表
43 bool vis[maxn], used[maxe]; // used表示边是否在最短路树上
44 int u[maxe], v[maxe], w[maxe]; // 存下每条边, 注意是有向边
45 int d[maxn], p[maxn]; // p表示最短路树上每个点的父边
46 int n, m, k, s, t; // s, t分别表示起点和终点
47
48 // 以下是主函数中较关键的部分
49 for (int i = 0; i <= n; i++)
50     root[i] = null; // 一定要加上!!!
51
52 // (读入&建反向图)
53
54 Dijkstra();
55
56 // (清空G, W, id)
57
58 for (int i = 1; i <= n; i++)
59     if (p[i]) {
60         used[p[i]] = true; // 在最短路树上
61         G[v[p[i]]].push_back(i);
62     }
63
64 for (int i = 1; i <= m; i++) {
65     w[i] = d[u[i]] - d[v[i]]; // 现在的w[i]表示这条边能
  // 使路径长度增加多少
66     if (!used[i])
67         root[u[i]] = merge(root[u[i]], newnode(w[i], i));
68 }
69
70 dfs(t);
71
72 priority_queue<B> heap;
73 heap.push(B(s, root[s], 0)); // 初始状态是找贡献最小的边
  // 加进去
74
75 printf("%d\n", d[s]); // 第1短路需要特判
76 while (--k) { // 其余k - 1短路路径用二叉堆维护
77     if (heap.empty())
78         printf("-1\n");
79     else {
80         int x = heap.top().x, w = heap.top().w;
81         node *rt = heap.top().rt;
82         heap.pop();
83
84         printf("%d\n", d[s] + w + rt -> w);
85
86         if (rt -> lc != null || rt -> rc != null)
87             heap.push(B(x, merge(rt -> lc, rt -> rc),
  // w)); // pop掉当前边, 换成另一条贡献大一点
  // 的边
88         if (root[v[rt -> i]] != null)
89             heap.push(B(v[rt -> i], root[v[rt -> i]], w +
  // rt -> w)); // 保留当前边, 往后面再接上另
  // 一条边
90     }
91 }
92 // 主函数到此结束
93
94
95
96

```

```

97 // Dijkstra预处理最短路  $O(m \log n)$ 
98 void Dijkstra() {
99     memset(d, 63, sizeof(d));
100     d[t] = 0;
101     priority_queue<A> heap;
102     heap.push(A(t, 0));
103
104     while (!heap.empty()) {
105         int x = heap.top().x;
106         heap.pop();
107
108         if (vis[x])
109             continue;
110
111         vis[x] = true;
112         for (int i = 0; i < (int)G[x].size(); i++)
113             if (!vis[G[x][i]] && d[G[x][i]] > d[x] + W[x][i])
114                 d[G[x][i]] = d[x] + W[x][i];
115                 p[G[x][i]] = id[x][i];
116
117         heap.push(A(G[x][i], d[G[x][i]]));
118     }
119 }
120
121 // dfs求出每个点的堆 总计 $O(m \log n)$ 
122 // 需要调用merge, 同时递归调用自身
123 void dfs(int x) {
124     root[x] = merge(root[x], root[v[p[x]]]);
125
126     for (int i = 0; i < (int)G[x].size(); i++)
127         dfs(G[x][i]);
128 }
129
130 // 包装过的new node()  $O(1)$ 
131 node *newnode(int w, int i) {
132     *++ptr = node(w, i);
133     ptr -> lc = ptr -> rc = null;
134     return ptr;
135 }
136
137 // 带可持久化的左偏树合并 总计 $O(n \log n)$ 
138 // 递归调用自身
139 node *merge(node *x, node *y) {
140     if (x == null)
141         return y;
142     if (y == null)
143         return x;
144
145     if (x -> w > y -> w)
146         swap(x, y);
147
148     node *z = newnode(x -> w, x -> i);
149     z -> lc = x -> lc;
150     z -> rc = merge(x -> rc, y);
151
152     if (z -> lc -> d > z -> rc -> d)
153         swap(z -> lc, z -> rc);
154     z -> refresh();
155
156     return z;
157 }
158

```

3.3 仙人掌

3.3.1 仙人掌DP

```

1 struct edge{
2     int to, w, prev;
3 }e[maxn * 2];
4
5 vector<pair<int, int>> v[maxn];
6
7 vector<long long> d[maxn];
8
9 stack<int> stk;
10
11 int p[maxn];
12
13 bool vis[maxn], vise[maxn * 2];
14
15 int last[maxn], cnte;
16
17 long long f[maxn], g[maxn], sum[maxn];
18
19 int n, m, cnt;
20
21 void addedge(int x, int y, int w) {
22     v[x].push_back(make_pair(y, w));
23 }
24
25 void dfs(int x) {
26
27     vis[x] = true;
28
29     for (int i = last[x]; ~i; i = e[i].prev) {
30         if (vise[i ^ 1])
31             continue;
32
33         int y = e[i].to, w = e[i].w;
34
35         vise[i] = true;
36
37         if (!vis[y]) {
38             stk.push(i);
39             p[y] = x;
40             dfs(y);
41
42             if (!stk.empty() && stk.top() == i) {
43                 stk.pop();
44                 addedge(x, y, w);
45             }
46         }
47
48         else {
49             cnt++;
50
51             long long tmp = w;
52             while (!stk.empty()) {
53                 int i = stk.top();
54                 stk.pop();
55
56                 int yy = e[i].to, ww = e[i].w;
57
58                 addedge(cnt, yy, 0);
59
60                 d[cnt].push_back(tmp);
61
62                 tmp += ww;
63
64                 if (e[i ^ 1].to == y)
65                     break;
66             }
67

```

```

68         addedge(y, cnt, 0);
69     }
70     sum[cnt] = tmp;
71 }
72 }
73 }
74
75 void dp(int x) {
76     for (auto o : v[x]) {
77         int y = o.first, w = o.second;
78         dp(y);
79     }
80
81     if (x <= n) {
82         for (auto o : v[x]) {
83             int y = o.first, w = o.second;
84
85             f[x] += 2 * w + f[y];
86         }
87
88         g[x] = f[x];
89
90         for (auto o : v[x]) {
91             int y = o.first, w = o.second;
92
93             g[x] = min(g[x], f[x] - f[y] - 2 * w + g[y] +
94                 ↪ w);
95         }
96     }
97     else {
98         f[x] = sum[x];
99         for (auto o : v[x]) {
100             int y = o.first;
101
102             f[x] += f[y];
103         }
104
105         g[x] = f[x];
106
107         for (int i = 0; i < (int)v[x].size(); i++) {
108             int y = v[x][i].first;
109
110             g[x] = min(g[x], f[x] - f[y] + g[y] +
111                 ↪ min(d[x][i], sum[x] - d[x][i]));
112         }
113     }

```

3.4 二分图

3.4.1 KM二分图最大权匹配

```

1  const long long INF = 0x3f3f3f3f3f3f3f3f;
2
3  long long w[maxn][maxn], lx[maxn], ly[maxn], slack[maxn];
4  // 边权 顶标 slack
5  // 如果要求最大权完美匹配就把不存在的边设为-INF, 否则所有
6  ↪ 边对0取max
7  bool visx[maxn], visy[maxn];
8
9  int boy[maxn], girl[maxn], p[maxn], q[maxn], head, tail;
10 ↪ // p : pre
11
12 int n, m, N, e;
13
14 // 增广
15 bool check(int y) {
16     visy[y] = true;

```

```

16     if (boy[y]) {
17         visx[boy[y]] = true;
18         q[tail++] = boy[y];
19         return false;
20     }
21
22     while (y) {
23         boy[y] = p[y];
24         swap(y, girl[p[y]]);
25     }
26
27     return true;
28 }
29
30 // bfs每个点
31 void bfs(int x) {
32     memset(q, 0, sizeof(q));
33     head = tail = 0;
34
35     q[tail++] = x;
36     visx[x] = true;
37
38     while (true) {
39         while (head != tail) {
40             int x = q[head++];
41
42             for (int y = 1; y <= N; y++)
43                 if (!visy[y]) {
44                     long long d = lx[x] + ly[y] - w[x]
45                         ↪ [y];
46
47                     if (d < slack[y]) {
48                         p[y] = x;
49                         slack[y] = d;
50
51                         if (!slack[y] && check(y))
52                             return;
53                     }
54                 }
55         }
56
57         long long d = INF;
58         for (int i = 1; i <= N; i++)
59             if (!visy[i])
60                 d = min(d, slack[i]);
61
62         for (int i = 1; i <= N; i++) {
63             if (visx[i])
64                 lx[i] -= d;
65
66             if (visy[i])
67                 ly[i] += d;
68             else
69                 slack[i] -= d;
70         }
71
72         for (int i = 1; i <= N; i++)
73             if (!visy[i] && !slack[i] && check(i))
74                 return;
75     }
76 }
77
78 // 主过程
79 long long KM() {
80     for (int i = 1; i <= N; i++) {
81         // lx[i] = 0;
82         ly[i] = -INF;

```

```

83     // boy[i] = girl[i] = -1;
84
85     for (int j = 1; j <= N; j++)
86         ly[i] = max(ly[i], w[j][i]);
87 }
88
89 for (int i = 1; i <= N; i++) {
90     memset(slack, 0x3f, sizeof(slack));
91     memset(visx, 0, sizeof(visx));
92     memset(visy, 0, sizeof(visy));
93     bfs(i);
94 }
95
96 long long ans = 0;
97 for (int i = 1; i <= N; i++)
98     ans += w[i][girl[i]];
99 return ans;
100 }
101
102 // 为了方便贴上主函数
103 int main() {
104
105     scanf("%d%d", &n, &m, &e);
106     N = max(n, m);
107
108     while (e--) {
109         int x, y, c;
110         scanf("%d%d", &x, &y, &c);
111         w[x][y] = max(c, 0);
112     }
113
114     printf("%lld\n", KM());
115
116     for (int i = 1; i <= n; i++) {
117         if (i > 1)
118             printf(" ");
119         printf("%d", w[i][girl[i]] > 0 ? girl[i] : 0);
120     }
121     printf("\n");
122
123     return 0;
124 }

```

```

19
20 scanf("%d", &n, &m); // 点数和边数
21 while (m--) {
22     int x, y;
23     scanf("%d", &x, &y);
24     A[x][y] = rand() % p;
25     A[y][x] = -A[x][y]; // Tutte矩阵是反对称矩阵
26 }
27
28 for (int i = 1; i <= n; i++)
29     id[i] = i; // 输出方案用的, 因为高斯消元的时候会
                // → 交换列
30 memcpy(t, A, sizeof(t));
31 Gauss(A, NULL, n);
32
33 m = n;
34 n = 0; // 这里变量复用纯属个人习惯.....
35
36 for (int i = 1; i <= m; i++)
37     if (A[id[i]][id[i]])
38         a[++n] = i; // 找出一个极大满秩子矩阵
39
40 for (int i = 1; i <= n; i++)
41     for (int j = 1; j <= n; j++)
42         A[i][j] = t[a[i]][a[j]];
43
44 Gauss(A, B, n);
45
46 for (int i = 1; i <= n; i++)
47     if (!girl[a[i]])
48         for (int j = i + 1; j <= n; j++)
49             if (!girl[a[j]] && t[a[i]][a[j]] && B[j]
                    // → [i]) {
                    // 注意上面那句if的写法, 现在t是邻接矩
                    // → 阵的备份,
                    // 逆矩阵j行i列不为0当且仅当这条边可
                    // → 行
                    girl[a[i]] = a[j];
                    girl[a[j]] = a[i];
                    eliminate(i, j);
                    eliminate(j, i);
                    break;
50 }
51
52
53
54
55
56
57
58
59 printf("%d\n", n >> 1);
60 for (int i = 1; i <= m; i++)
61     printf("%d ", girl[i]);
62
63
64 }
65
66 // 高斯消元  $O(n^3)$ 
67 // 在传入B时表示计算逆矩阵, 传入NULL则只需计算矩阵的秩
68 void Gauss(int A[][maxn], int B[][maxn], int n){
69     if(B) {
70         memset(B, 0, sizeof(t));
71         for (int i = 1; i <= n; i++)
72             B[i][i] = 1;
73     }
74
75     for (int i = 1; i <= n; i++) {
76         if (!A[i][i]) {
77             for (int j = i + 1; j <= n; j++)
78                 if (A[j][i]) {
79                     swap(id[i], id[j]);
80                     for (int k = i; k <= n; k++)
81                         swap(A[i][k], A[j][k]);
82

```

3.5 一般图匹配

3.5.1 高斯消元

```

1 // 这个算法基于Tutte定理和高斯消元, 思维难度相对小一些, 也
  // → 更方便进行可行边的判定
2 // 注意这个算法复杂度是满的, 并且常数有点大, 而带花树通常
  // → 是跑不满的
3 // 以及, 根据Tutte定理, 如果求最大匹配的大小的话直接输
  // → 出Tutte矩阵的秩/2即可
4 // 需要输出方案时才需要再写后面那些乱七八糟的东西
5
6
7 // 复杂度和常数所限, 1s之内500已经是这个算法的极限了
8 const int maxn = 505, p = 1000000007; // p可以是任
  // → 意 $10^9$ 以内的质数
9
10 // 全局数组和变量定义
11 int A[maxn][maxn], B[maxn][maxn], t[maxn][maxn],
    // → id[maxn], a[maxn];
12 bool row[maxn] = {false}, col[maxn] = {false};
13 int n, m, girl[maxn]; // girl是匹配点, 用来输出方案
14
15 // 为了方便使用, 贴上主函数
16 // 需要调用高斯消元和eliminate
17 int main() {
18     srand(19260817); // 膜蛤专用随机种子, 换一个也无所谓

```

```

83         if (B)
84             for (int k = 1; k <= n; k++)
85                 swap(B[i][k], B[j][k]);
86         break;
87     }
88
89     if (!A[i][i])
90         continue;
91 }
92
93 int inv = qpow(A[i][i], p - 2);
94
95 for (int j = 1; j <= n; j++)
96     if (i != j && A[j][i]){
97         int t = (long long)A[j][i] * inv % p;
98
99         for (int k = i; k <= n; k++)
100             if (A[j][k])
101                 A[j][k] = (A[j][k] - (long long)t
102                     ↪ * A[i][k]) % p;
103
104         if (B)
105             for (int k = 1; k <= n; k++)
106                 if (B[i][k])
107                     B[j][k] = (B[j][k] - (long
108                         ↪ long)t * B[i][k])%p;
109     }
110
111 if (B)
112     for (int i = 1; i <= n; i++) {
113         int inv = qpow(A[i][i], p - 2);
114
115         for (int j = 1; j <= n; j++)
116             if (B[i][j])
117                 B[i][j] = (long long)B[i][j] * inv %
118                 ↪ p;
119     }
120 // 消去一行一列  $O(n^2)$ 
121 void eliminate(int r, int c) {
122     row[r] = col[c] = true; // 已经被消掉
123
124     int inv = qpow(B[r][c], p - 2);
125
126     for (int i = 1; i <= n; i++)
127         if (!row[i] && B[i][c]) {
128             int t = (long long)B[i][c] * inv % p;
129
130             for (int j = 1; j <= n; j++)
131                 if (!col[j] && B[r][j])
132                     B[i][j] = (B[i][j] - (long long)t *
133                         ↪ B[r][j]) % p;
134         }
135 }

```

3.5.2 带花树

```

1 // 带花树通常比高斯消元快很多,但在只要求最大匹配大小的
2   ↪ 时候并没有高斯消元好写
3 // 当然输出方案要方便很多
4 // 全局数组与变量定义
5 vector<int> G[maxn];
6 int girl[maxn], f[maxn], t[maxn], p[maxn], vis[maxn],
7   ↪ tim, q[maxn], head, tail;
8 int n, m;

```

```

9 // 封装好的主过程  $O(nm)$ 
10 int blossom() {
11     int ans = 0;
12
13     for (int i = 1; i <= n; i++)
14         if (!girl[i])
15             ans += bfs(i);
16
17     return ans;
18 }
19
20 // bfs找增广路  $O(m)$ 
21 bool bfs(int s) {
22     memset(t, 0, sizeof(t));
23     memset(p, 0, sizeof(p));
24
25     for (int i = 1; i <= n; i++)
26         f[i] = i; // 并查集
27
28     head = tail = 0;
29     q[tail++] = s;
30     t[s] = 1;
31
32     while (head != tail){
33         int x = q[head++];
34         for (int y : G[x]){
35             if (findroot(y) == findroot(x) || t[y] == 2)
36                 continue;
37
38             if (!t[y]){
39                 t[y] = 2;
40                 p[y] = x;
41
42                 if (!girl[y]){
43                     for (int u = y, t; u; u = t) {
44                         t = girl[p[u]];
45                         girl[p[u]] = u;
46                         girl[u] = p[u];
47                     }
48                     return true;
49                 }
50                 t[girl[y]] = 1;
51                 q[tail++] = girl[y];
52             }
53             else if (t[y] == 1) {
54                 int z = LCA(x, y);
55                 shrink(x, y, z);
56                 shrink(y, x, z);
57             }
58         }
59     }
60
61     return false;
62 }
63
64 //缩奇环  $O(n)$ 
65 void shrink(int x, int y, int z) {
66     while (findroot(x) != z){
67         p[x] = y;
68         y = girl[x];
69
70         if (t[y] == 2) {
71             t[y] = 1;
72             q[tail++] = y;
73         }
74
75         if (findroot(x) == x)

```

```

78     f[x] = z;
79     if(findroot(y) == y)
80         f[y] = z;
81
82     x = p[y];
83 }
84 }
85
86 //暴力找LCA O(n)
87 int LCA(int x, int y) {
88     tim++;
89     while (true) {
90         if (x) {
91             x = findroot(x);
92
93             if (vis[x] == tim)
94                 return x;
95             else {
96                 vis[x] = tim;
97                 x = p[girl[x]];
98             }
99         }
100         swap(x, y);
101     }
102 }
103
104 //并查集的查找 O(1)
105 int findroot(int x) {
106     return x == f[x] ? x : (f[x] = findroot(f[x]));
107 }

```

3.5.3 带权带花树

(有一说一这玩意实在太难写了，抄之前建议先想想算法是不是假的或者有SB做法)

```

1  #define DIST(e) (lab[e.u]+lab[e.v]-g[e.u][e.v].w*2)
2  struct Edge{ int u,v,w; } g[N][N];
3  int n,m,n_x,lab[N],match[N],slack[N],
4  st[N],pa[N],fl_from[N][N],S[N],vis[N];
5  vector<int> fl[N];
6  deque<int> q;
7  void update_slack(int u,int x){
8      if(!slack[x]||DIST(g[u][x])<DIST(g[slack[x]][x]))
9          ↪ slack[x]=u; }
10 void set_slack(int x){
11     slack[x]=0;
12     for(int u=1; u<=n; ++u)
13         if(g[u][x].w>0&&st[u]!
14             ↪ =x&&S[st[u]]==0)update_slack(u,x);
15 }
16 void q_push(int x){
17     if(x<=n)return q.push_back(x);
18     for(int i=0; i<fl[x].size(); i++)q_push(fl[x][i]);
19 }
20 void set_st(int x,int b){
21     st[x]=b;
22     if(x<=n)return;
23     for(int i=0; i<fl[x].size(); ++i)set_st(fl[x][i],b);
24 }
25 int get_pr(int b,int xr){
26     int
27     ↪ pr=find(fl[b].begin(),fl[b].end(),xr)-fl[b].begin();
28     if(pr%2==1){
29         reverse(fl[b].begin()+1,fl[b].end());
30         return (int)fl[b].size()-pr;
31     }
32     else return pr;
33 }
34 void set_match(int u,int v){

```

```

32     match[u]=g[u][v].v;
33     if(u<=n)return;
34     Edge e=g[u][v];
35     int xr=fl_from[u][e.u],pr=get_pr(u,xr);
36     for(int i=0; i<pr; ++i)set_match(fl[u][i],fl[u]
37         ↪ [i^1]);
38     set_match(xr,v);
39     rotate(fl[u].begin(),fl[u].begin()+pr,fl[u].end());
40 }
41 void augment(int u,int v){
42     int xnv=st[match[u]];
43     set_match(u,v);
44     if(!xnv)return;
45     set_match(xnv,st[pa[xnv]]);
46     augment(st[pa[xnv]],xnv);
47 }
48 int get_lca(int u,int v){
49     static int t=0;
50     for(++t; u||v; swap(u,v)){
51         if(u==0)continue;
52         if(vis[u]==t)return u;
53         vis[u]=t;
54         u=st[match[u]];
55         if(u)u=st[pa[u]];
56     }
57     return 0;
58 }
59 void add_blossom(int u,int lca,int v){
60     int b=n+1;
61     while(b<=n_x&&st[b])++b;
62     if(b>n_x)++n_x;
63     lab[b]=0,S[b]=0;
64     match[b]=match[lca];
65     fl[b].clear();
66     fl[b].push_back(lca);
67     for(int x=u,y; x!=lca; x=st[pa[y]])
68         fl[b].push_back(x),
69         fl[b].push_back(y=st[match[x]]),q_push(y);
70     reverse(fl[b].begin()+1,fl[b].end());
71     for(int x=v,y; x!=lca; x=st[pa[y]])
72         fl[b].push_back(x),
73         fl[b].push_back(y=st[match[x]]),q_push(y);
74     set_st(b,b);
75     for(int x=1; x<=n_x; ++x)g[b][x].w=g[x][b].w=0;
76     for(int x=1; x<=n; ++x)fl_from[b][x]=0;
77     for(int i=0; i<fl[b].size(); ++i){
78         int xs=fl[b][i];
79         for(int x=1; x<=n_x; ++x)
80             if(g[b][x].w==0||DIST(g[xs][x])<DIST(g[b]
81                 ↪ [x]))
82                 g[b][x]=g[xs][x],g[x][b]=g[x][xs];
83         for(int x=1; x<=n; ++x)
84             if(fl_from[xs][x])fl_from[b][x]=xs;
85     }
86     set_slack(b);
87 }
88 void expand_blossom(int b){
89     for(int i=0; i<fl[b].size(); ++i)
90         set_st(fl[b][i],fl[b][i]);
91     int xr=fl_from[b][g[b][pa[b]].u],pr=get_pr(b,xr);
92     for(int i=0; i<pr; i+=2){
93         int xs=fl[b][i],xns=fl[b][i+1];
94         pa[xs]=g[xns][xs].u;
95         S[xs]=1,S[xns]=0;
96         slack[xs]=0,set_slack(xns);
97         q_push(xns);
98     }
99     S[xr]=1,pa[xr]=pa[b];

```

```

98     for(int i=pr+1; i<fl[b].size(); ++i){
99         int xs=fl[b][i];
100         S[xs]=-1,set_slack(xs);
101     }
102     st[b]=0;
103 }
104 bool on_found_Edge(const Edge &e){
105     int u=st[e.u],v=st[e.v];
106     if(S[v]==-1){
107         pa[v]=e.u,S[v]=1;
108         int nu=st[match[v]];
109         slack[v]=slack[nu]=0;
110         S[nu]=0,q_push(nu);
111     }
112     else if(S[v]==0){
113         int lca=get_lca(u,v);
114         if(!lca)return augment(u,v),augment(v,u),1;
115         else add_blossom(u,lca,v);
116     }
117     return 0;
118 }
119 bool matching(){
120     fill(S,S+n_x+1,-1),fill(slack,slack+n_x+1,0);
121     q.clear();
122     for(int x=1; x<=n_x; ++x)
123         if(st[x]==x&&!match[x])pa[x]=0,S[x]=0,q_push(x);
124     if(q.empty())return 0;
125     for(;;){
126         while(q.size()){
127             int u=q.front();
128             q.pop_front();
129             if(S[st[u]]==1)continue;
130             for(int v=1; v<=n; ++v)
131                 if(g[u][v].w>0&&st[u]!=st[v]){
132                     if(DIST(g[u][v])==0){
133                         if(on_found_Edge(g[u][v]))return 1;
134                     }
135                     else update_slack(u,st[v]);
136                 }
137         }
138         int d=INF;
139         for(int b=n+1; b<=n_x; ++b)
140             if(st[b]==b&&S[b]==1)d=min(d,lab[b]/2);
141         for(int x=1; x<=n_x; ++x)
142             if(st[x]==x&&slack[x]){
143                 if(S[x]==-1)d=min(d,DIST(g[slack[x]]
144                     ↪ [x]));
145                 else if(S[x]==0)d=min(d,DIST(g[slack[x]]
146                     ↪ [x])/2);
147             }
148         for(int u=1; u<=n; ++u){
149             if(S[st[u]]==0){
150                 if(lab[u]<=d)return 0;
151                 lab[u]-=d;
152             }
153             else if(S[st[u]]==1)lab[u]+=d;
154         }
155         for(int b=n+1; b<=n_x; ++b)
156             if(st[b]==b){
157                 if(S[st[b]]==0)lab[b]+=d*2;
158                 else if(S[st[b]]==1)lab[b]-=d*2;
159             }
160         q.clear();
161         for(int x=1; x<=n_x; ++x)
162             if(st[x]==x&&slack[x]&&st[slack[x]]!
163                 ↪ =x&&DIST(g[slack[x]][x])==0)
164                 if(on_found_Edge(g[slack[x]][x]))return 1;

```

```

162         for(int b=n+1; b<=n_x; ++b)
163             if(st[b]==b&&S[b]==1&&lab[b]==0)expand_blossom(b);
164         ↪ }
165     return 0; }
166 pair<ll,int> weight_blossom(){
167     fill(match,match+n+1,0);
168     n_x=n;
169     int n_matches=0;
170     ll tot_weight=0;
171     for(int u=0; u<=n; ++u)st[u]=u,fl[u].clear();
172     int w_max=0;
173     for(int u=1; u<=n; ++u)
174         for(int v=1; v<=n; ++v){
175             fl_from[u][v]=(u==v?u:0);
176             w_max=max(w_max,g[u][v].w); }
177     for(int u=1; u<=n; ++u)lab[u]=w_max;
178     while(matching())n_matches++;
179     for(int u=1; u<=n; ++u)
180         if(match[u]&&match[u]<u)
181             tot_weight+=g[u][match[u]].w;
182     return make_pair(tot_weight,n_matches); }
183 int main(){
184     cin>>n>>m;
185     for(int u=1; u<=n; ++u)
186         for(int v=1; v<=n; ++v)
187             g[u][v]=Edge {u,v,0};
188     for(int i=0,u,v,w; i<m; ++i){
189         cin>>u>>v>>w;
190         g[u][v].w=g[v][u].w=w; }
191     cout<<weight_blossom().first<<'\n';
192     for(int u=1; u<=n; ++u)cout<<match[u]<<' '; }

```

3.6 最大流

3.6.1 Dinic

```

1 // 注意Dinic适用于二分图或分层图,对于一般稀疏图ISAP更
2   ↪ 优,稠密图则HLPP更优
3
4 struct edge{
5     int to, cap, prev;
6 } e[maxe * 2];
7
8 int last[maxn], len, d[maxn], cur[maxn], q[maxn];
9 memset(last, -1, sizeof(last));
10
11 void AddEdge(int x, int y, int z) {
12     e[len].to = y;
13     e[len].cap = z;
14     e[len].prev = last[x];
15     last[x] = len++;
16 }
17
18 int Dinic() {
19     int flow = 0;
20     while (bfs(), ~d[t]) {
21         memcpy(cur, last, sizeof(int) * (t + 5));
22         flow += dfs(s, inf);
23     }
24     return flow;
25 }
26
27 void bfs() {
28     int head = 0, tail = 0;
29     memset(d, -1, sizeof(int) * (t + 5));
30     q[tail++] = s;
31     d[s] = 0;

```



```

32
33     while (head != tail){
34         int x = q[head++];
35         for (int i = last[x]; ~i; i = e[i].prev)
36             if (e[i].cap > 0 && d[e[i].to] == -1) {
37                 d[e[i].to] = d[x] + 1;
38                 q[tail++] = e[i].to;
39             }
40     }
41 }
42
43 int dfs(int x, int a) {
44     if (x == t || !a)
45         return a;
46
47     int flow = 0, f;
48     for (int &i = cur[x]; ~i; i = e[i].prev)
49         if (e[i].cap > 0 && d[e[i].to] == d[x] + 1 && (f
50             ↪ = dfs(e[i].to, min(e[i].cap, a)))) {
51
52             e[i].cap -= f;
53             e[i^1].cap += f;
54             flow += f;
55             a -= f;
56
57             if (!a)
58                 break;
59
60     return flow;
61 }

```

3.6.2 ISAP

```

1 // 注意ISAP适用于一般稀疏图,对于二分图或分层图情
  ↪ 况Dinic比较优,稠密图则HLPP更优
2
3 // 边的定义
4 // 这里没有记录起点和反向边,因为反向边即为正向边xor 1,起
  ↪ 点即为反向边的终点
5
6 struct edge{
7     int to, cap, prev;
8 } e[maxe * 2];
9
10 // 全局变量和数组定义
11 int last[maxn], cnte = 0, d[maxn], p[maxn], c[maxn],
  ↪ cur[maxn], q[maxn];
12 int n, m, s, t; // s, t一定要开成全局变量
13
14 // 重要!!!
15 // main函数最前面一定要加上如下初始化
16 memset(last, -1, sizeof(last));
17
18 // 加边函数 O(1)
19 // 包装了加反向边的过程,方便调用
20 // 需要调用AddEdge
21 void addedge(int x, int y, int z) {
22     AddEdge(x, y, z);
23     AddEdge(y, x, 0);
24 }
25
26 // 真·加边函数 O(1)
27 void AddEdge(int x, int y, int z) {
28     e[cnte].to = y;
29     e[cnte].cap = z;

```

```

34     e[cnte].prev = last[x];
35     last[x] = cnte++;
36 }
37
38 // 主过程 O(n^2 m)
39 // 返回最大流的流量
40 // 需要调用bfs, augment
41 // 注意这里的n是编号最大值,在这个值不为n的时候一定要开个
  ↪ 变量记录下来并修改代码
42 // 非递归
43 int ISAP() {
44     bfs();
45
46     memcpy(cur, last, sizeof(cur));
47
48     int x = s, flow = 0;
49
50     while (d[s] < n) {
51         if (x == t) { // 如果走到了t就增广一次,并返回s重新
  ↪ 找增广路
52             flow += augment();
53             x = s;
54         }
55
56         bool ok = false;
57         for (int &i = cur[x]; ~i; i = e[i].prev)
58             if (e[i].cap && d[x] == d[e[i].to] + 1) {
59                 p[e[i].to] = i;
60                 x = e[i].to;
61
62                 ok = true;
63                 break;
64             }
65
66         if (!ok) { // 修改距离标号
67             int tmp = n - 1;
68             for (int i = last[x]; ~i; i = e[i].prev)
69                 if (e[i].cap)
70                     tmp = min(tmp, d[e[i].to] + 1);
71
72             if (!--c[d[x]])
73                 break; // gap优化,一定要加上
74
75             c[d[x] = tmp]++;
76             cur[x] = last[x];
77
78             if (x != s)
79                 x = e[p[x] ^ 1].to;
80         }
81     }
82     return flow;
83 }
84
85 // bfs函数 O(n+m)
86 // 预处理到t的距离标号
87 // 在测试数据组数较少时可以省略,把所有距离标号初始化为0
88 void bfs() {
89     memset(d, -1, sizeof(d));
90
91     int head = 0, tail = 0;
92     d[t] = 0;
93     q[tail++] = t;
94
95     while (head != tail) {
96         int x = q[head++];
97         c[d[x]]++;
98
99         for (int i = last[x]; ~i; i = e[i].prev)

```

```

101     if (e[i ^ 1].cap && d[e[i].to] == -1) {
102         d[e[i].to] = d[x] + 1;
103         q[tail++] = e[i].to;
104     }
105 }
106
107 // augment函数 O(n)
108 // 沿增广路增广一次, 返回增广的流量
109 int augment() {
110     int a = (~0u) >> 1; // INT_MAX
111
112     for (int x = t; x != s; x = e[p[x] ^ 1].to)
113         a = min(a, e[p[x]].cap);
114
115     for (int x = t; x != s; x = e[p[x] ^ 1].to) {
116         e[p[x]].cap -= a;
117         e[p[x] ^ 1].cap += a;
118     }
119
120     return a;
121 }
122

```

3.6.3 HLPP最高标号预流推进

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  constexpr int maxn = 1205, maxe = 120005, inf =
6      ↪ 2147483647;
7
8  struct edge {
9      int to, cap, prev;
10 } e[maxe * 2];
11
12 int n, m, s, t;
13 int last[maxn], cnte;
14 int h[maxn], ex[maxn], gap[maxn * 2];
15 bool inq[maxn];
16
17 struct cmp {
18     bool operator() (int x, int y) const {
19         return h[x] < h[y];
20     }
21 };
22
23 priority_queue<int, vector<int>, cmp> heap;
24
25 void AddEdge(int x, int y, int z) {
26     e[cnte].to = y;
27     e[cnte].cap = z;
28     e[cnte].prev = last[x];
29     last[x] = cnte++;
30 }
31
32 void addedge(int x, int y, int z) {
33     AddEdge(x, y, z);
34     AddEdge(y, x, 0);
35 }
36
37 bool bfs() {
38     static int q[maxn];
39
40     fill(h, h + n + 1, 2 * n);
41     int head = 0, tail = 0;
42     q[tail++] = t;
43     h[t] = 0;
44
45     while (head < tail) {

```

```

45         int x = q[head++];
46         for (int i = last[x]; ~i; i = e[i].prev)
47             if (e[i ^ 1].cap && h[e[i].to] > h[x] + 1) {
48                 h[e[i].to] = h[x] + 1;
49                 q[tail++] = e[i].to;
50             }
51     }
52
53     return h[s] < 2 * n;
54 }
55
56 void push(int x) {
57     for (int i = last[x]; ~i; i = e[i].prev)
58         if (e[i].cap && h[x] == h[e[i].to] + 1) {
59             int d = min(ex[x], e[i].cap);
60
61             e[i].cap -= d;
62             e[i ^ 1].cap += d;
63             ex[x] -= d;
64             ex[e[i].to] += d;
65
66             if (e[i].to != s && e[i].to != t &&
67                 ↪ !inq[e[i].to]) {
68                 heap.push(e[i].to);
69                 inq[e[i].to] = true;
70             }
71
72             if (!ex[x])
73                 break;
74         }
75     }
76
77 void relabel(int x) {
78     h[x] = 2 * n;
79
80     for (int i = last[x]; ~i; i = e[i].prev)
81         if (e[i].cap)
82             h[x] = min(h[x], h[e[i].to] + 1);
83     }
84
85 int hlpp() {
86     if (!bfs())
87         return 0;
88
89     // memset(gap, 0, sizeof(int) * 2 * n);
90     h[s] = n;
91
92     for (int i = 1; i <= n; i++)
93         gap[h[i]]++;
94
95     for (int i = last[s]; ~i; i = e[i].prev)
96         if (e[i].cap) {
97             int d = e[i].cap;
98
99             e[i].cap -= d;
100             e[i ^ 1].cap += d;
101             ex[s] -= d;
102             ex[e[i].to] += d;
103
104             if (e[i].to != s && e[i].to != t &&
105                 ↪ !inq[e[i].to]) {
106                 heap.push(e[i].to);
107                 inq[e[i].to] = true;
108             }
109         }
110
111     while (!heap.empty()) {
112         int x = heap.top();
113         heap.pop();

```

```

112     inq[x] = false;
113
114     push(x);
115     if (ex[x]) {
116         if (!--gap[h[x]]) { // gap
117             for (int i = 1; i <= n; i++)
118                 if (i != s && i != t && h[i] > h[x])
119                     h[i] = n + 1;
120         }
121
122         relabel(x);
123         ++gap[h[x]];
124         heap.push(x);
125         inq[x] = true;
126     }
127 }
128
129 return ex[t];
130 }
131
132 int main() {
133     memset(last, -1, sizeof(last));
134
135     scanf("%d%d%d", &n, &m, &s, &t);
136
137     while (m--) {
138         int x, y, z;
139         scanf("%d%d%d", &x, &y, &z);
140         addedge(x, y, z);
141     }
142
143     printf("%d\n", hlpp());
144
145     return 0;
146 }

```

```

29         if (d[x] + e[i].w > d[y]) {
30             p[y] = i;
31             d[y] = d[x] + e[i].w;
32             if (!inq[y]) {
33                 q.push(y);
34                 inq[y] = true;
35             }
36         }
37     }
38 }
39
40 int mcmf(int s, int t) {
41     int ans = 0;
42
43     while (spfa(s), d[t] > 0) {
44         int flow = 0x3f3f3f3f;
45         for (int x = t; x != s; x = e[p[x] ^ 1].to)
46             flow = min(flow, e[p[x]].cap);
47
48         ans += flow * d[t];
49
50         for (int x = t; x != s; x = e[p[x] ^ 1].to) {
51             e[p[x]].cap -= flow;
52             e[p[x] ^ 1].cap += flow;
53         }
54     }
55
56     return ans;
57 }
58
59 void add(int x, int y, int c, int w) {
60     e[cnte].to = y;
61     e[cnte].cap = c;
62     e[cnte].w = w;
63
64     e[cnte].prev = last[x];
65     last[x] = cnte++;
66 }
67
68 void addedge(int x, int y, int c, int w) {
69     add(x, y, c, w);
70     add(y, x, 0, -w);
71 }
72

```

3.7 费用流

3.7.1 SPFA费用流

```

1 constexpr int maxn = 20005, maxm = 200005;
2
3 struct edge {
4     int to, prev, cap, w;
5 } e[maxm * 2];
6
7 int last[maxn], cnte, d[maxn], p[maxn]; // 记得把last初始
    ↪ 化成-1, 不然会死循环
8 bool inq[maxn];
9
10 void spfa(int s) {
11
12     memset(d, -63, sizeof(d));
13     memset(p, -1, sizeof(p));
14
15     queue<int> q;
16
17     q.push(s);
18     d[s] = 0;
19
20     while (!q.empty()) {
21         int x = q.front();
22         q.pop();
23         inq[x] = false;
24
25         for (int i = last[x]; ~i; i = e[i].prev)
26             if (e[i].cap) {
27                 int y = e[i].to;
28

```

4. 数据结构

4.1 线段树

4.1.1 非递归线段树

让fstqwq手撕

- 如果 $M = 2^k$, 则只能维护 $[1, M - 2]$ 范围
- 找叶子: i 对应的叶子就是 $i + M$
- 单点修改: 找到叶子然后向上跳
- 区间查询: 左右区间各扩展一位, 转换成开区间查询

```

1 int query(int l, int r) {
2     l += M - 1;
3     r += M + 1;
4
5     int ans = 0;
6     while (l ^ r != 1) {
7         ans += sum[l ^ 1] + sum[r ^ 1];
8
9         l >>= 1;

```

```

10     r >>= 1;
11 }
12
13 return ans;
14 }

```

区间修改要标记永久化,并且求区间和和求最值的代码不太一样

区间加, 区间求和

```

1 void update(int l, int r, int d) {
2     int len = 1, cntl = 0, cntr = 0; // cntl, cntr是左右
3     // 两边分别实际修改的区间长度
4     for (l += n - 1, r += n + 1; l ^ r ^ 1; l >>= 1, r
5         >>= 1, len <= 1) {
6         tree[l] += cntl * d, tree[r] += cntr * d;
7         if (~l & 1) tree[l ^ 1] += d * len, mark[l ^ 1]
8             += d, cntl += len;
9         if (r & 1) tree[r ^ 1] += d * len, mark[r ^ 1] +=
10            d, cntr += len;
11     }
12
13     for (; l; l >>= 1, r >>= 1)
14         tree[l] += cntl * d, tree[r] += cntr * d;
15 }
16
17 int query(int l, int r) {
18     int ans = 0, len = 1, cntl = 0, cntr = 0;
19     for (l += n - 1, r += n + 1; l ^ r ^ 1; l >>= 1, r
20         >>= 1, len <= 1) {
21         ans += cntl * mark[l] + cntr * mark[r];
22         if (~l & 1) ans += tree[l ^ 1], cntl += len;
23         if (r & 1) ans += tree[r ^ 1], cntr += len;
24     }
25
26     for (; l; l >>= 1, r >>= 1)
27         ans += cntl * mark[l] + cntr * mark[r];
28
29     return ans;
30 }

```

区间加, 区间求最大值

```

1 void update(int l, int r, int d) {
2     for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r
3         >>= 1) {
4         if (l < N) {
5             tree[l] = max(tree[l << 1], tree[l << 1 | 1])
6                 + mark[l];
7             tree[r] = max(tree[r << 1], tree[r << 1 | 1])
8                 + mark[r];
9         }
10
11         if (~l & 1) {
12             tree[l ^ 1] += d;
13             mark[l ^ 1] += d;
14         }
15         if (r & 1) {
16             tree[r ^ 1] += d;
17             mark[r ^ 1] += d;
18         }
19     }
20
21     for (; l; l >>= 1, r >>= 1)
22         if (l < N) tree[l] = max(tree[l << 1], tree[l <<
23             1 | 1]) + mark[l],
24         tree[r] = max(tree[r << 1], tree[r <<
25             1 | 1]) + mark[r];
26 }

```

```

23 void query(int l, int r) {
24     int maxl = -INF, maxr = -INF;
25
26     for (l += N - 1, r += N + 1; l ^ r ^ 1; l >>= 1, r
27         >>= 1) {
28         maxl += mark[l];
29         maxr += mark[r];
30
31         if (~l & 1)
32             maxl = max(maxl, tree[l ^ 1]);
33         if (r & 1)
34             maxr = max(maxr, tree[r ^ 1]);
35     }
36
37     while (l) {
38         maxl += mark[l];
39         maxr += mark[r];
40
41         l >>= 1;
42         r >>= 1;
43     }
44
45     return max(maxl, maxr);
46 }

```

4.1.2 主席树

这种东西能不能手撕啊

4.2 陈丹琦分治

```

1 // 四维偏序
2
3 void CDQ1(int l, int r) {
4     if (l >= r)
5         return;
6
7     int mid = (l + r) / 2;
8
9     CDQ1(l, mid);
10    CDQ1(mid + 1, r);
11
12    int i = l, j = mid + 1, k = l;
13
14    while (i <= mid && j <= r) {
15        if (a[i].x < a[j].x) {
16            a[i].ins = true;
17            b[k++] = a[i++];
18        }
19        else {
20            a[j].ins = false;
21            b[k++] = a[j++];
22        }
23    }
24
25    while (i <= mid) {
26        a[i].ins = true;
27        b[k++] = a[i++];
28    }
29
30    while (j <= r) {
31        a[j].ins = false;
32        b[k++] = a[j++];
33    }
34
35    copy(b + l, b + r + 1, a + l); // 后面的分治会破坏排
36    // 序, 所以要复制一份

```

```

37 CDQ2(l, r);
38 }
39
40 void CDQ2(int l, int r) {
41     if (l >= r)
42         return;
43
44     int mid = (l + r) / 2;
45
46     CDQ2(l, mid);
47     CDQ2(mid + 1, r);
48
49     int i = l, j = mid + 1, k = 1;
50
51     while (i <= mid && j <= r) {
52         if (b[i].y < b[j].y) {
53             if (b[i].ins)
54                 add(b[i].z, 1); // 树状数组
55
56             t[k++] = b[i++];
57         }
58         else {
59             if (!b[j].ins)
60                 ans += query(b[j].z - 1);
61
62             t[k++] = b[j++];
63         }
64     }
65
66     while (i <= mid) {
67         if (b[i].ins)
68             add(b[i].z, 1);
69
70         t[k++] = b[i++];
71     }
72
73     while (j <= r) {
74         if (!b[j].ins)
75             ans += query(b[j].z - 1);
76
77         t[k++] = b[j++];
78     }
79
80     for (i = l; i <= mid; i++)
81         if (b[i].ins)
82             add(b[i].z, -1);
83
84     copy(t + 1, t + r + 1, b + 1);
85 }

```

4.3 Treap

```

1 // 注意：相同键值可以共存
2
3 struct node { // 结点类定义
4     int key, size, p; // 分别为键值，子树大小，优先度
5     node *ch[2]; // 0表示左儿子，1表示右儿子
6
7     node(int key = 0) : key(key), size(1), p(rand()) {}
8
9     void refresh() {
10         size = ch[0] -> size + ch[1] -> size + 1;
11     } // 更新子树大小(和附加信息，如果有的话)
12 } null[maxn], *root = null, *ptr = null; // 数组名叫
13 // 做null是为了方便开哨兵节点
14 // 如果需要删除而空间不能直接开下所有结点，则需要再写一
15 // 个垃圾回收
16 // 注意：数组里的元素一定不能delete，否则会导致RE

```

```

16 // 重要!在主函数最开始一定要加上以下预处理:
17 null -> ch[0] = null -> ch[1] = null;
18 null -> size = 0;
19
20 // 伪构造函数 O(1)
21 // 为了方便，在结点类外面再定义一个伪构造函数
22 node *newnode(int x) { // 键值为x
23     *++ptr = node(x);
24     ptr -> ch[0] = ptr -> ch[1] = null;
25     return ptr;
26 }
27
28 // 插入键值 期望O(\Log n)
29 // 需要调用旋转
30 void insert(int x, node *&rt) { // rt为当前结点，建议调用
31     // 时传入root，下同
32     if (rt == null) {
33         rt = newnode(x);
34         return;
35     }
36
37     int d = x > rt -> key;
38     insert(x, rt -> ch[d]);
39     rt -> refresh();
40
41     if (rt -> ch[d] -> p < rt -> p)
42         rot(rt, d ^ 1);
43 }
44
45 // 删除一个键值 期望O(\Log n)
46 // 要求键值必须存在至少一个，否则会导致RE
47 // 需要调用旋转
48 void erase(int x, node *&rt) {
49     if (x == rt -> key) {
50         if (rt -> ch[0] != null && rt -> ch[1] != null) {
51             int d = rt -> ch[0] -> p < rt -> ch[1] -> p;
52             rot(rt, d);
53             erase(x, rt -> ch[d]);
54         }
55         else
56             rt = rt -> ch[rt -> ch[0] == null];
57     }
58     else
59         erase(x, rt -> ch[x > rt -> key]);
60
61     if (rt != null)
62         rt -> refresh();
63 }
64
65 // 求元素的排名(严格小于键值的个数 + 1) 期望O(\Log n)
66 // 非递归
67 int rank(int x, node *rt) {
68     int ans = 1, d;
69     while (rt != null) {
70         if ((d = x > rt -> key))
71             ans += rt -> ch[0] -> size + 1;
72         rt = rt -> ch[d];
73     }
74
75     return ans;
76 }
77
78 // 返回排名第k(从1开始)的键值对应的指针 期望O(\Log n)
79 // 非递归
80 node *kth(int x, node *rt) {
81     int d;
82     while (rt != null) {
83         if (x == rt -> ch[0] -> size + 1)
84             return rt;

```

```

85
86     if ((d = x > rt -> ch[0] -> size))
87     |   x -= rt -> ch[0] -> size + 1;
88
89     rt = rt -> ch[d];
90 }
91
92 return rt;
93 }
94
95 // 返回前驱(最大的比给定键值小的键值)对应的指针 期
96   ↳ 望 $O(\log n)$ 
97 // 非递归
98 node *pred(int x, node *rt) {
99     node *y = null;
100     int d;
101
102     while (rt != null) {
103         if ((d = x > rt -> key))
104         |   y = rt;
105
106         rt = rt -> ch[d];
107     }
108
109     return y;
110 }
111
112 // 返回后继最小的比给定键值大的键值对应的指针 期
113   ↳ 望 $O(\log n)$ 
114 // 非递归
115 node *succ(int x, node *rt) {
116     node *y = null;
117     int d;
118
119     while (rt != null) {
120         if ((d = x < rt -> key))
121         |   y = rt;
122
123         rt = rt -> ch[d ^ 1];
124     }
125
126     return y;
127 }
128
129 // 旋转(Treap版本)  $O(1)$ 
130 // 平衡树基础操作
131 // 要求对应儿子必须存在, 否则会导致后续各种莫名其妙的问
132   ↳ 题
133 void rot(node *&x, int d) { // x为被转下去的结点, 会被修
134   ↳ 改以维护树结构
135     node *y = x -> ch[d ^ 1];
136
137     x -> ch[d ^ 1] = y -> ch[d];
138     y -> ch[d] = x;
139
140     x -> refresh();
141     (x = y) -> refresh();
142 }

```

4.4 Splay

(参见LCT,除了splay()需要传一个点表示最终它的父亲,其他写法都和LCT相同)

4.5 树分治

4.5.1 动态树分治

```

1 // 为了减小常数, 这里采用bfs写法, 实测预处理比dfs快将近
2   ↳ 一半
3 // 以下以维护一个点到每个黑点的距离之和为例
4
5 // 全局数组定义
6 vector<int> G[maxn], W[maxn];
7 int size[maxn], son[maxn], q[maxn];
8 int p[maxn], depth[maxn], id[maxn][20], d[maxn][20]; //
9   ↳ id是对应层所在子树的根
10 int a[maxn], ca[maxn], b[maxn][20], cb[maxn][20]; // 维护
11   ↳ 距离和用的
12 bool vis[maxn], col[maxn];
13
14 // 建树 总计 $O(n \log n)$ 
15 // 需要调用找重心和预处理距离, 同时递归调用自身
16 void build(int x, int k, int s, int pr) { // 结点, 深度,
17   ↳ 连通块大小, 点分树上的父亲
18     x = getcenter(x, s);
19     vis[x] = true;
20     depth[x] = k;
21     p[x] = pr;
22
23     for (int i = 0; i < (int)G[x].size(); i++)
24     |   if (!vis[G[x][i]]) {
25     |       d[G[x][i]][k] = W[x][i];
26     |       p[G[x][i]] = x;
27     |
28     |       getdis(G[x][i], k, G[x][i]); // bfs每个子树, 预
29     |       ↳ 处理距离
30     |   }
31
32     for (int i = 0; i < (int)G[x].size(); i++)
33     |   if (!vis[G[x][i]])
34     |       build(G[x][i], k + 1, size[G[x][i]], x); //
35     |       ↳ 递归建树
36 }
37
38 // 找重心  $O(n)$ 
39 int getcenter(int x, int s) {
40     int head = 0, tail = 0;
41     q[tail++] = x;
42
43     while (head != tail) {
44         x = q[head++];
45         size[x] = 1; // 这里不需要清空, 因为以后要用的话
46         ↳ 一定会重新赋值
47         son[x] = 0;
48
49         for (int i = 0; i < (int)G[x].size(); i++)
50         |   if (!vis[G[x][i]] && G[x][i] != p[x]) {
51         |       p[G[x][i]] = x;
52         |       q[tail++] = G[x][i];
53         |   }
54     }
55
56     for (int i = tail - 1; i > 0; i--) {
57         x = q[i];
58         size[p[x]] += size[x];
59
60         if (size[x] > size[son[p[x]]])
61         |   son[p[x]] = x;
62     }
63
64     x = q[0];
65     while (son[x] && size[son[x]] * 2 >= s)
66     |   x = son[x];

```

```

60     return x;
61 }
62
63 // 预处理距离  $O(n)$ 
64 // 方便起见, 这里直接用了笨一点的方法,  $O(n \log n)$  全存下
65 // 来
66 void getdis(int x, int k, int rt) {
67     int head = 0, tail = 0;
68     q[tail++] = x;
69
70     while (head != tail) {
71         x = q[head++];
72         size[x] = 1;
73         id[x][k] = rt;
74
75         for (int i = 0; i < (int)G[x].size(); i++)
76             if (!vis[G[x][i]] && G[x][i] != p[x]) {
77                 p[G[x][i]] = x;
78                 d[G[x][i]][k] = d[x][k] + W[x][i];
79
80                 q[tail++] = G[x][i];
81             }
82     }
83
84     for (int i = tail - 1; i; i--)
85         size[p[q[i]]] += size[q[i]]; // 后面递归建树要用
86 // 到子问题大小
87
88 // 修改  $O(\log n)$ 
89 void modify(int x) {
90     if (col[x])
91         ca[x]--;
92     else
93         ca[x]++; // 记得先特判自己作为重心的那层
94
95     for (int u = p[x], k = depth[x] - 1; u; u = p[u],
96 // k--) {
97         if (col[x]) {
98             a[u] -= d[x][k];
99             ca[u]--;
100
101             b[id[x][k]][k] -= d[x][k];
102             cb[id[x][k]][k]--;
103         }
104         else {
105             a[u] += d[x][k];
106             ca[u]++;
107
108             b[id[x][k]][k] += d[x][k];
109             cb[id[x][k]][k]++;
110         }
111     }
112
113     col[x] ^= true;
114
115 // 询问  $O(\log n)$ 
116 int query(int x) {
117     int ans = a[x]; // 特判自己是重心的那层
118
119     for (int u = p[x], k = depth[x] - 1; u; u = p[u],
120 // k--)
121         ans += a[u] - b[id[x][k]][k] + d[x][k] * (ca[u] -
122 // cb[id[x][k]][k]);
123
124     return ans;
125 }

```

4.5.2 紫荆花之恋

```

1 #include<cstdio>
2 #include<cstring>
3 #include<algorithm>
4 #include<vector>
5 using namespace std;
6 const int maxn=100010;
7 const double alpha=0.7;
8 struct node{
9     static int randint(){
10         static int
11 // a=1213,b=97818217,p=998244353,x=751815431;
12         x=a*x+b;x%=p;
13         return x<0?(x+=p):x;
14     }
15     int data,size,p;
16     node *ch[2];
17     node(int d):data(d),size(1),p(randint()){
18         inline void refresh()
19 // {size=ch[0]->size+ch[1]->size+1;}
20 // }*null=new node(0),*root[maxn],*root1[maxn][50];
21 void addnode(int,int);
22 void rebuild(int,int,int,int);
23 void dfs_getcenter(int,int,int&);
24 void dfs_getdis(int,int,int,int);
25 void dfs_destroy(int,int);
26 void insert(int,node*&);
27 int order(int,node*&);
28 void destroy(node*&);
29 void rot(node*&,int);
30 vector<int>G[maxn],W[maxn];
31 int size[maxn]={0},siz[maxn][50]={0},son[maxn];
32 bool vis[maxn];
33 int depth[maxn],p[maxn],d[maxn][50],id[maxn][50];
34 int n,m,w[maxn],tmp;
35 long long ans=0;
36 int main(){
37     freopen("flowera.in","r",stdin);
38     freopen("flowera.out","w",stdout);
39     null->size=0;
40     null->ch[0]=null->ch[1]=null;
41     scanf("%d",&n);
42     fill(vis,vis+n+1,true);
43     fill(root,root+n+1,null);
44     for(int i=0;i<=n;i++)fill(root1[i],root1[i]+50,null);
45     scanf("%d%d",&w[1]);
46     insert(-w[1],root[1]);
47     size[1]=1;
48     printf("0\n");
49     for(int i=2;i<=n;i++){
50         scanf("%d%d",&p[i],&tmp,&w[i]);
51         p[i]^=(ans%(int)1e9);
52         G[i].push_back(p[i]);
53         W[i].push_back(tmp);
54         G[p[i]].push_back(i);
55         W[p[i]].push_back(tmp);
56         addnode(i,tmp);
57         printf("%lld\n",ans);
58     }
59     return 0;
60 }
61 void addnode(int x,int z){//wj-dj>=di-wi
62     depth[x]=depth[p[x]]+1;
63     size[x]=1;
64     insert(-w[x],root[x]);
65     int rt=0;
66     for(int u=p[x],k=depth[p[x]];u;u=p[u],k--){
67         if(u==p[x]){

```



```

66     id[x][k]=x;
67     d[x][k]=z;
68 }
69 else{
70     id[x][k]=id[p[x]][k];
71     d[x][k]=d[p[x]][k]+z;
72 }
73 ans+=order(w[x]-d[x][k],root[u]-order(w[x]-d[x]
    ↪ [k],root1[id[x][k]][k]));
74 insert(d[x][k]-w[x],root[u]);
75 insert(d[x][k]-w[x],root1[id[x][k]][k]);
76 size[u]++;
77 siz[id[x][k]][k]++;
78 if(siz[id[x][k]][k]>size[u]*alpha+5)rt=u;
79 }
80 id[x][depth[x]]=0;
81 d[x][depth[x]]=0;
82 if(rt){
83     dfs_destroy(rt,depth[rt]);
84     rebuild(rt,depth[rt],size[rt],p[rt]);
85 }
86 }
87 void rebuild(int x,int k,int s,int pr){
88     int u=0;
89     dfs_getcenter(x,s,u);
90     vis[x]=true;
91     p[x]=pr;
92     depth[x]=k;
93     size[x]=s;
94     d[x][k]=id[x][k]=0;
95     destroy(root[x]);
96     insert(-w[x],root[x]);
97     if(s<=1)return;
98     for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x][i]]){
99         p[G[x][i]]=0;
100         d[G[x][i]][k]=w[x][i];
101         siz[G[x][i]][k]=p[G[x][i]][i]=0;
102         destroy(root1[G[x][i]][k]);
103         dfs_getdis(G[x][i],x,G[x][i],k);
104     }
105     for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x]
    ↪ [i]])rebuild(G[x][i],k+1,size[G[x][i]],x);
106 }
107 void dfs_getcenter(int x,int s,int &u){
108     size[x]=1;
109     son[x]=0;
110     for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x]
    ↪ [i]]&&G[x][i]!=p[x]){
111         p[G[x][i]]=x;
112         dfs_getcenter(G[x][i],s,u);
113         size[x]+=size[G[x][i]];
114         if(size[G[x][i]]>size[son[x]])son[x]=G[x][i];
115     }
116     if(!u||max(s-size[x],size[son[x]])<max(s-size[u],size[son
    ↪ [u]]))u=x;
117 }
118 void dfs_getdis(int x,int u,int rt,int k){
119     insert(d[x][k]-w[x],root[u]);
120     insert(d[x][k]-w[x],root1[rt][k]);
121     id[x][k]=rt;
122     siz[rt][k]++;
123     size[x]=1;
124     for(int i=0;i<(int)G[x].size();i++)if(!vis[G[x]
    ↪ [i]]&&G[x][i]!=p[x]){
125         p[G[x][i]]=x;
126         d[G[x][i]][k]=d[x][k]+w[x][i];
127         dfs_getdis(G[x][i],u,rt,k);
128         size[x]+=size[G[x][i]];
129     }
130 }

```

```

131 void dfs_destroy(int x,int k){
132     vis[x]=false;
133     for(int i=0;i<(int)G[x].size();i++)if(depth[G[x]
    ↪ [i]]>=k&&G[x][i]!=p[x]){
134         p[G[x][i]]=x;
135         dfs_destroy(G[x][i],k);
136     }
137 }
138 void insert(int x,node *&rt){
139     if(rt==null){
140         rt=new node(x);
141         rt->ch[0]=rt->ch[1]=null;
142         return;
143     }
144     int d=x>rt->data;
145     insert(x,rt->ch[d]);
146     rt->refresh();
147     if(rt->ch[d]->p<rt->p)rot(rt,d^1);
148 }
149 int order(int x,node *rt){
150     int ans=0,d;
151     x++;
152     while(rt!=null){
153         if((d=x>rt->data))ans+=rt->ch[0]->size+1;
154         rt=rt->ch[d];
155     }
156     return ans;
157 }
158 void destroy(node *&x){
159     if(x==null)return;
160     destroy(x->ch[0]);
161     destroy(x->ch[1]);
162     delete x;
163     x=null;
164 }
165 void rot(node *&x,int d){
166     node *y=x->ch[d^1];
167     x->ch[d^1]=y->ch[d];
168     y->ch[d]=x;
169     x->refresh();
170     (x=y)->refresh();
171 }

```

4.6 LCT

4.6.1 不换根(弹飞绵羊)

```

1 #define isroot(x) ((x) != (x) -> p -> ch[0] && (x) != (x)
    ↪ -> p -> ch[1]) // 判断是不是Splay的根
2 #define dir(x) ((x) == (x) -> p -> ch[1]) // 判断它是它父
    ↪ 亲的左 / 右儿子
3
4 struct node { // 结点类定义
5     int size; // Splay的子树大小
6     node *ch[2], *p;
7
8     node() : size(1) {}
9     void refresh() {
10         size = ch[0] -> size + ch[1] -> size + 1;
11     } // 附加信息维护
12 } null[maxn];
13
14 // 在主函数开头加上这句初始化
15 null -> size = 0;
16
17 // 初始化结点
18 void initialize(node *x) {
19     x -> ch[0] = x -> ch[1] = x -> p = null;

```

```

20 }
21
22 // Access 均摊 $O(\log n)$ 
23 // LCT核心操作, 把结点到根的路径打通, 顺便把与重儿子的连
    ↳ 边变成轻边
24 // 需要调用splay
25 node *access(node *x) {
26     node *y = null;
27
28     while (x != null) {
29         splay(x);
30
31         x -> ch[1] = y;
32         (y = x) -> refresh();
33
34         x = x -> p;
35     }
36
37     return y;
38 }
39
40 // Link 均摊 $O(\log n)$ 
41 // 把x的父亲设为y
42 // 要求x必须为所在树的根节点, 否则会导致后续各种莫名其妙的
    ↳ 问题
43 // 需要调用splay
44 void link(node *x, node *y) {
45     splay(x);
46     x -> p = y;
47 }
48
49 // Cut 均摊 $O(\log n)$ 
50 // 把x与其父亲的连边断开
51 // x可以是所在树的根节点, 这时此操作没有任何实质效果
52 // 需要调用access和splay
53 void cut(node *x) {
54     access(x);
55     splay(x);
56
57     x -> ch[0] -> p = null;
58     x -> ch[0] = null;
59
60     x -> refresh();
61 }
62
63 // Splay 均摊 $O(\log n)$ 
64 // 需要调用旋转
65 void splay(node *x) {
66     while (!isroot(x)) {
67         if (isroot(x -> p)) {
68             rot(x -> p, dir(x) ^ 1);
69             break;
70         }
71
72         if (dir(x) == dir(x -> p))
73             rot(x -> p -> p, dir(x -> p) ^ 1);
74         else
75             rot(x -> p, dir(x) ^ 1);
76         rot(x -> p, dir(x) ^ 1);
77     }
78 }
79
80 // 旋转(LCT版本)  $O(1)$ 
81 // 平衡树基本操作
82 // 要求对应儿子必须存在, 否则会导致后续各种莫名其妙的问
    ↳ 题
83 void rot(node *x, int d) {
84     node *y = x -> ch[d ^ 1];
85
86     y -> p = x -> p;
87     if (!isroot(x))

```

```

88     x -> p -> ch[dir(x)] = y;
89
90     if ((x -> ch[d ^ 1] = y -> ch[d]) != null)
91         y -> ch[d] -> p = x;
92     (y -> ch[d] = x) -> p = y;
93
94     x -> refresh();
95     y -> refresh();
96 }

```

4.6.2 换根/维护生成树

```

1 #define isroot(x) ((x) -> p == null || ((x) -> p -> ch[0]
    ↳ != (x) && (x) -> p -> ch[1] != (x)))
2 #define dir(x) ((x) == (x) -> p -> ch[1])
3
4 using namespace std;
5
6 const int maxn = 200005;
7
8 struct node{
9     int key, mx, pos;
10     bool rev;
11     node *ch[2], *p;
12
13     node(int key = 0): key(key), mx(key), pos(-1),
        ↳ rev(false) {}
14
15     void pushdown() {
16         if (!rev)
17             return;
18
19         ch[0] -> rev ^= true;
20         ch[1] -> rev ^= true;
21         swap(ch[0], ch[1]);
22
23         if (pos != -1)
24             pos ^= 1;
25
26         rev = false;
27     }
28
29     void refresh() {
30         mx = key;
31         pos = -1;
32         if (ch[0] -> mx > mx) {
33             mx = ch[0] -> mx;
34             pos = 0;
35         }
36         if (ch[1] -> mx > mx) {
37             mx = ch[1] -> mx;
38             pos = 1;
39         }
40     }
41 } null[maxn * 2];
42
43 void init(node *x, int k) {
44     x -> ch[0] = x -> ch[1] = x -> p = null;
45     x -> key = x -> mx = k;
46 }
47
48 void rot(node *x, int d) {
49     node *y = x -> ch[d ^ 1];
50     if ((x -> ch[d ^ 1] = y -> ch[d]) != null)
51         y -> ch[d] -> p = x;
52
53     y -> p = x -> p;
54     if (!isroot(x))

```

```

55 |     x -> p -> ch[dir(x)] = y;
56 |
57 |     (y -> ch[d] = x) -> p = y;
58 |
59 |     x -> refresh();
60 |     y -> refresh();
61 | }
62 |
63 | void splay(node *x) {
64 |     x -> pushdown();
65 |
66 |     while (!isroot(x)) {
67 |         if (!isroot(x -> p))
68 |             x -> p -> p -> pushdown();
69 |         x -> p -> pushdown();
70 |         x -> pushdown();
71 |
72 |         if (isroot(x -> p)) {
73 |             rot(x -> p, dir(x) ^ 1);
74 |             break;
75 |         }
76 |
77 |         if (dir(x) == dir(x -> p))
78 |             rot(x -> p -> p, dir(x -> p) ^ 1);
79 |         else
80 |             rot(x -> p, dir(x) ^ 1);
81 |
82 |         rot(x -> p, dir(x) ^ 1);
83 |     }
84 | }
85 |
86 | node *access(node *x) {
87 |     node *y = null;
88 |
89 |     while (x != null) {
90 |         splay(x);
91 |
92 |         x -> ch[1] = y;
93 |         (y = x) -> refresh();
94 |
95 |         x = x -> p;
96 |     }
97 |
98 |     return y;
99 | }
100 |
101 | void makeroot(node *x) {
102 |     access(x);
103 |     splay(x);
104 |     x -> rev ^= true;
105 | }
106 |
107 | void link(node *x, node *y) {
108 |     makeroot(x);
109 |     x -> p = y;
110 | }
111 |
112 | void cut(node *x, node *y) {
113 |     makeroot(x);
114 |     access(y);
115 |     splay(y);
116 |
117 |     y -> ch[0] -> p = null;
118 |     y -> ch[0] = null;
119 |     y -> refresh();
120 | }
121 |
122 | node *getroot(node *x) {
123 |     x = access(x);
124 |     while (x -> pushdown(), x -> ch[0] != null)

```

```

125 |     x = x -> ch[0];
126 |     splay(x);
127 |     return x;
128 | }
129 |
130 | node *getmax(node *x, node *y) {
131 |     makeroot(x);
132 |     x = access(y);
133 |
134 |     while (x -> pushdown(), x -> pos != -1)
135 |         x = x -> ch[x -> pos];
136 |     splay(x);
137 |
138 |     return x;
139 | }
140 |
141 | // 以下为主函数示例
142 | for (int i = 1; i <= m; i++) {
143 |     init(null + n + i, w[i]);
144 |     if (getroot(null + u[i]) != getroot(null + v[i])) {
145 |         ans[q + 1] -= k;
146 |         ans[q + 1] += w[i];
147 |
148 |         link(null + u[i], null + n + i);
149 |         link(null + v[i], null + n + i);
150 |         vis[i] = true;
151 |     }
152 |     else {
153 |         int ii = getmax(null + u[i], null + v[i]) - null
154 |             -> n;
155 |         if (w[i] >= w[ii])
156 |             continue;
157 |
158 |         cut(null + u[ii], null + n + ii);
159 |         cut(null + v[ii], null + n + ii);
160 |
161 |         link(null + u[i], null + n + i);
162 |         link(null + v[i], null + n + i);
163 |
164 |         ans[q + 1] -= w[ii];
165 |         ans[q + 1] += w[i];
166 |     }

```

4.6.3 维护子树信息

```

1 // 这个东西虽然只需要抄板子但还是极其难写，常数极其巨大，
  ↳ 没必要的时候就不要用
2 // 如果维护子树最小值就需要套一个可删除的堆来维护，复杂
  ↳ 度会变成  $O(n \log^2 n)$ 
3 // 注意由于这道题与边权有关，需要边权拆点变点权
4
5 // 宏定义
6 #define isroot(x) ((x) -> p == null || ((x) != (x) -> p
  ↳ -> ch[0] && (x) != (x) -> p -> ch[1]))
7 #define dir(x) ((x) == (x) -> p -> ch[1])
8
9 // 节点类定义
10 struct node { // 以维护子树中黑点到根距离和为例
11     int w, chain_cnt, tree_cnt;
12     long long sum, suml, sumr, tree_sum; // 由于换根需要
  ↳ 子树反转，需要维护两个方向的信息
13     bool rev, col;
14     node *ch[2], *p;
15
16     node() : w(0), chain_cnt(0),
  ↳ tree_cnt(0), sum(0), suml(0), sumr(0),
  ↳ tree_sum(0), rev(false), col(false) {}
17
18     inline void pushdown() {

```

```

20     if(!rev)
21     |     return;
22
23     ch[0]->rev ^= true;
24     ch[1]->rev ^= true;
25     swap(ch[0], ch[1]);
26     swap(suml, sumr);
27
28     rev = false;
29 }
30
31 inline void refresh() { // 如果不想这样特判
32     ↪ 就pushdown一下
33     // pushdown();
34
35     sum = ch[0] -> sum + ch[1] -> sum + w;
36     suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
37     ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
38     ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
39     ↪ (ch[0] -> sum + w) + tree_sum;
40
41     sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
42     ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
43     ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
44     ↪ (ch[1] -> sum + w) + tree_sum;
45
46     chain_cnt = ch[0] -> chain_cnt + ch[1] ->
47     ↪ chain_cnt + tree_cnt;
48 }
49 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
50
51 // 封装构造函数
52 node *newnode(int w) {
53     node *x = nodes.front(); // 因为有删边加边, 可以用一
54     ↪ 个队列维护可用结点
55     nodes.pop();
56     initialize(x);
57     x -> w = w;
58     x -> refresh();
59     return x;
60 }
61
62 // 封装初始化函数
63 // 记得在进行操作之前对所有结点调用一遍
64 inline void initialize(node *x) {
65     *x = node();
66     x -> ch[0] = x -> ch[1] = x -> p = null;
67 }
68
69 // 注意一下在Access的同时更新子树信息的方法
70 node *access(node *x) {
71     node *y = null;
72
73     while (x != null) {
74         splay(x);
75
76         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
77         ↪ chain_cnt;
78         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
79         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
80         x -> ch[1] = y;
81
82         (y = x) -> refresh();
83         x = x -> p;
84     }
85
86     return y;
87 }
88
89 // 找到一个点所在连通块的根
90 // 对比原版没有变化
91 node *getroot(node *x) {
92     if(!rev)
93     |     return;
94
95     ch[0]->rev ^= true;
96     ch[1]->rev ^= true;
97     swap(ch[0], ch[1]);
98     swap(suml, sumr);
99
100    rev = false;
101 }
102
103 inline void refresh() { // 如果不想这样特判
104     ↪ 就pushdown一下
105     // pushdown();
106
107    sum = ch[0] -> sum + ch[1] -> sum + w;
108    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
109    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
110    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
111    ↪ (ch[0] -> sum + w) + tree_sum;
112
113    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
114    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
115    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
116    ↪ (ch[1] -> sum + w) + tree_sum;
117
118    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
119    ↪ chain_cnt + tree_cnt;
120 }
121 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
122
123 // 封装构造函数
124 node *newnode(int w) {
125     node *x = nodes.front(); // 因为有删边加边, 可以用一
126     ↪ 个队列维护可用结点
127     nodes.pop();
128     initialize(x);
129     x -> w = w;
130     x -> refresh();
131     return x;
132 }
133
134 // 封装初始化函数
135 // 记得在进行操作之前对所有结点调用一遍
136 inline void initialize(node *x) {
137     *x = node();
138     x -> ch[0] = x -> ch[1] = x -> p = null;
139 }
140
141 // 注意一下在Access的同时更新子树信息的方法
142 node *access(node *x) {
143     node *y = null;
144
145     while (x != null) {
146         splay(x);
147
148         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
149         ↪ chain_cnt;
150         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
151         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
152         x -> ch[1] = y;
153
154         (y = x) -> refresh();
155         x = x -> p;
156     }
157
158     return y;
159 }
160
161 // 找到一个点所在连通块的根
162 // 对比原版没有变化
163 node *getroot(node *x) {
164     if(!rev)
165     |     return;
166
167     ch[0]->rev ^= true;
168     ch[1]->rev ^= true;
169     swap(ch[0], ch[1]);
170     swap(suml, sumr);
171
172    rev = false;
173 }
174
175 inline void refresh() { // 如果不想这样特判
176     ↪ 就pushdown一下
177     // pushdown();
178
179    sum = ch[0] -> sum + ch[1] -> sum + w;
180    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
181    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
182    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
183    ↪ (ch[0] -> sum + w) + tree_sum;
184
185    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
186    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
187    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
188    ↪ (ch[1] -> sum + w) + tree_sum;
189
190    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
191    ↪ chain_cnt + tree_cnt;
192 }
193 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
194
195 // 封装构造函数
196 node *newnode(int w) {
197     node *x = nodes.front(); // 因为有删边加边, 可以用一
198     ↪ 个队列维护可用结点
199     nodes.pop();
200     initialize(x);
201     x -> w = w;
202     x -> refresh();
203     return x;
204 }
205
206 // 封装初始化函数
207 // 记得在进行操作之前对所有结点调用一遍
208 inline void initialize(node *x) {
209     *x = node();
210     x -> ch[0] = x -> ch[1] = x -> p = null;
211 }
212
213 // 注意一下在Access的同时更新子树信息的方法
214 node *access(node *x) {
215     node *y = null;
216
217     while (x != null) {
218         splay(x);
219
220         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
221         ↪ chain_cnt;
222         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
223         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
224         x -> ch[1] = y;
225
226         (y = x) -> refresh();
227         x = x -> p;
228     }
229
230     return y;
231 }
232
233 // 找到一个点所在连通块的根
234 // 对比原版没有变化
235 node *getroot(node *x) {
236     if(!rev)
237     |     return;
238
239     ch[0]->rev ^= true;
240     ch[1]->rev ^= true;
241     swap(ch[0], ch[1]);
242     swap(suml, sumr);
243
244    rev = false;
245 }
246
247 inline void refresh() { // 如果不想这样特判
248     ↪ 就pushdown一下
249     // pushdown();
250
251    sum = ch[0] -> sum + ch[1] -> sum + w;
252    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
253    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
254    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
255    ↪ (ch[0] -> sum + w) + tree_sum;
256
257    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
258    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
259    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
260    ↪ (ch[1] -> sum + w) + tree_sum;
261
262    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
263    ↪ chain_cnt + tree_cnt;
264 }
265 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
266
267 // 封装构造函数
268 node *newnode(int w) {
269     node *x = nodes.front(); // 因为有删边加边, 可以用一
270     ↪ 个队列维护可用结点
271     nodes.pop();
272     initialize(x);
273     x -> w = w;
274     x -> refresh();
275     return x;
276 }
277
278 // 封装初始化函数
279 // 记得在进行操作之前对所有结点调用一遍
280 inline void initialize(node *x) {
281     *x = node();
282     x -> ch[0] = x -> ch[1] = x -> p = null;
283 }
284
285 // 注意一下在Access的同时更新子树信息的方法
286 node *access(node *x) {
287     node *y = null;
288
289     while (x != null) {
290         splay(x);
291
292         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
293         ↪ chain_cnt;
294         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
295         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
296         x -> ch[1] = y;
297
298         (y = x) -> refresh();
299         x = x -> p;
300     }
301
302     return y;
303 }
304
305 // 找到一个点所在连通块的根
306 // 对比原版没有变化
307 node *getroot(node *x) {
308     if(!rev)
309     |     return;
310
311     ch[0]->rev ^= true;
312     ch[1]->rev ^= true;
313     swap(ch[0], ch[1]);
314     swap(suml, sumr);
315
316    rev = false;
317 }
318
319 inline void refresh() { // 如果不想这样特判
320     ↪ 就pushdown一下
321     // pushdown();
322
323    sum = ch[0] -> sum + ch[1] -> sum + w;
324    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
325    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
326    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
327    ↪ (ch[0] -> sum + w) + tree_sum;
328
329    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
330    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
331    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
332    ↪ (ch[1] -> sum + w) + tree_sum;
333
334    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
335    ↪ chain_cnt + tree_cnt;
336 }
337 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
338
339 // 封装构造函数
340 node *newnode(int w) {
341     node *x = nodes.front(); // 因为有删边加边, 可以用一
342     ↪ 个队列维护可用结点
343     nodes.pop();
344     initialize(x);
345     x -> w = w;
346     x -> refresh();
347     return x;
348 }
349
350 // 封装初始化函数
351 // 记得在进行操作之前对所有结点调用一遍
352 inline void initialize(node *x) {
353     *x = node();
354     x -> ch[0] = x -> ch[1] = x -> p = null;
355 }
356
357 // 注意一下在Access的同时更新子树信息的方法
358 node *access(node *x) {
359     node *y = null;
360
361     while (x != null) {
362         splay(x);
363
364         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
365         ↪ chain_cnt;
366         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
367         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
368         x -> ch[1] = y;
369
370         (y = x) -> refresh();
371         x = x -> p;
372     }
373
374     return y;
375 }
376
377 // 找到一个点所在连通块的根
378 // 对比原版没有变化
379 node *getroot(node *x) {
380     if(!rev)
381     |     return;
382
383     ch[0]->rev ^= true;
384     ch[1]->rev ^= true;
385     swap(ch[0], ch[1]);
386     swap(suml, sumr);
387
388    rev = false;
389 }
390
391 inline void refresh() { // 如果不想这样特判
392     ↪ 就pushdown一下
393     // pushdown();
394
395    sum = ch[0] -> sum + ch[1] -> sum + w;
396    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
397    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
398    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
399    ↪ (ch[0] -> sum + w) + tree_sum;
400
401    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
402    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
403    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
404    ↪ (ch[1] -> sum + w) + tree_sum;
405
406    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
407    ↪ chain_cnt + tree_cnt;
408 }
409 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
410
411 // 封装构造函数
412 node *newnode(int w) {
413     node *x = nodes.front(); // 因为有删边加边, 可以用一
414     ↪ 个队列维护可用结点
415     nodes.pop();
416     initialize(x);
417     x -> w = w;
418     x -> refresh();
419     return x;
420 }
421
422 // 封装初始化函数
423 // 记得在进行操作之前对所有结点调用一遍
424 inline void initialize(node *x) {
425     *x = node();
426     x -> ch[0] = x -> ch[1] = x -> p = null;
427 }
428
429 // 注意一下在Access的同时更新子树信息的方法
430 node *access(node *x) {
431     node *y = null;
432
433     while (x != null) {
434         splay(x);
435
436         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
437         ↪ chain_cnt;
438         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
439         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
440         x -> ch[1] = y;
441
442         (y = x) -> refresh();
443         x = x -> p;
444     }
445
446     return y;
447 }
448
449 // 找到一个点所在连通块的根
450 // 对比原版没有变化
451 node *getroot(node *x) {
452     if(!rev)
453     |     return;
454
455     ch[0]->rev ^= true;
456     ch[1]->rev ^= true;
457     swap(ch[0], ch[1]);
458     swap(suml, sumr);
459
460    rev = false;
461 }
462
463 inline void refresh() { // 如果不想这样特判
464     ↪ 就pushdown一下
465     // pushdown();
466
467    sum = ch[0] -> sum + ch[1] -> sum + w;
468    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
469    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
470    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
471    ↪ (ch[0] -> sum + w) + tree_sum;
472
473    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
474    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
475    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
476    ↪ (ch[1] -> sum + w) + tree_sum;
477
478    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
479    ↪ chain_cnt + tree_cnt;
480 }
481 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
482
483 // 封装构造函数
484 node *newnode(int w) {
485     node *x = nodes.front(); // 因为有删边加边, 可以用一
486     ↪ 个队列维护可用结点
487     nodes.pop();
488     initialize(x);
489     x -> w = w;
490     x -> refresh();
491     return x;
492 }
493
494 // 封装初始化函数
495 // 记得在进行操作之前对所有结点调用一遍
496 inline void initialize(node *x) {
497     *x = node();
498     x -> ch[0] = x -> ch[1] = x -> p = null;
499 }
500
501 // 注意一下在Access的同时更新子树信息的方法
502 node *access(node *x) {
503     node *y = null;
504
505     while (x != null) {
506         splay(x);
507
508         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
509         ↪ chain_cnt;
510         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
511         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
512         x -> ch[1] = y;
513
514         (y = x) -> refresh();
515         x = x -> p;
516     }
517
518     return y;
519 }
520
521 // 找到一个点所在连通块的根
522 // 对比原版没有变化
523 node *getroot(node *x) {
524     if(!rev)
525     |     return;
526
527     ch[0]->rev ^= true;
528     ch[1]->rev ^= true;
529     swap(ch[0], ch[1]);
530     swap(suml, sumr);
531
532    rev = false;
533 }
534
535 inline void refresh() { // 如果不想这样特判
536     ↪ 就pushdown一下
537     // pushdown();
538
539    sum = ch[0] -> sum + ch[1] -> sum + w;
540    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
541    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
542    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
543    ↪ (ch[0] -> sum + w) + tree_sum;
544
545    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
546    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
547    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
548    ↪ (ch[1] -> sum + w) + tree_sum;
549
550    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
551    ↪ chain_cnt + tree_cnt;
552 }
553 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
554
555 // 封装构造函数
556 node *newnode(int w) {
557     node *x = nodes.front(); // 因为有删边加边, 可以用一
558     ↪ 个队列维护可用结点
559     nodes.pop();
560     initialize(x);
561     x -> w = w;
562     x -> refresh();
563     return x;
564 }
565
566 // 封装初始化函数
567 // 记得在进行操作之前对所有结点调用一遍
568 inline void initialize(node *x) {
569     *x = node();
570     x -> ch[0] = x -> ch[1] = x -> p = null;
571 }
572
573 // 注意一下在Access的同时更新子树信息的方法
574 node *access(node *x) {
575     node *y = null;
576
577     while (x != null) {
578         splay(x);
579
580         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
581         ↪ chain_cnt;
582         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
583         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
584         x -> ch[1] = y;
585
586         (y = x) -> refresh();
587         x = x -> p;
588     }
589
590     return y;
591 }
592
593 // 找到一个点所在连通块的根
594 // 对比原版没有变化
595 node *getroot(node *x) {
596     if(!rev)
597     |     return;
598
599     ch[0]->rev ^= true;
600     ch[1]->rev ^= true;
601     swap(ch[0], ch[1]);
602     swap(suml, sumr);
603
604    rev = false;
605 }
606
607 inline void refresh() { // 如果不想这样特判
608     ↪ 就pushdown一下
609     // pushdown();
610
611    sum = ch[0] -> sum + ch[1] -> sum + w;
612    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
613    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
614    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
615    ↪ (ch[0] -> sum + w) + tree_sum;
616
617    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
618    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
619    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
620    ↪ (ch[1] -> sum + w) + tree_sum;
621
622    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
623    ↪ chain_cnt + tree_cnt;
624 }
625 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
626
627 // 封装构造函数
628 node *newnode(int w) {
629     node *x = nodes.front(); // 因为有删边加边, 可以用一
630     ↪ 个队列维护可用结点
631     nodes.pop();
632     initialize(x);
633     x -> w = w;
634     x -> refresh();
635     return x;
636 }
637
638 // 封装初始化函数
639 // 记得在进行操作之前对所有结点调用一遍
640 inline void initialize(node *x) {
641     *x = node();
642     x -> ch[0] = x -> ch[1] = x -> p = null;
643 }
644
645 // 注意一下在Access的同时更新子树信息的方法
646 node *access(node *x) {
647     node *y = null;
648
649     while (x != null) {
650         splay(x);
651
652         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
653         ↪ chain_cnt;
654         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
655         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
656         x -> ch[1] = y;
657
658         (y = x) -> refresh();
659         x = x -> p;
660     }
661
662     return y;
663 }
664
665 // 找到一个点所在连通块的根
666 // 对比原版没有变化
667 node *getroot(node *x) {
668     if(!rev)
669     |     return;
670
671     ch[0]->rev ^= true;
672     ch[1]->rev ^= true;
673     swap(ch[0], ch[1]);
674     swap(suml, sumr);
675
676    rev = false;
677 }
678
679 inline void refresh() { // 如果不想这样特判
680     ↪ 就pushdown一下
681     // pushdown();
682
683    sum = ch[0] -> sum + ch[1] -> sum + w;
684    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
685    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
686    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
687    ↪ (ch[0] -> sum + w) + tree_sum;
688
689    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
690    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
691    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
692    ↪ (ch[1] -> sum + w) + tree_sum;
693
694    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
695    ↪ chain_cnt + tree_cnt;
696 }
697 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
698
699 // 封装构造函数
700 node *newnode(int w) {
701     node *x = nodes.front(); // 因为有删边加边, 可以用一
702     ↪ 个队列维护可用结点
703     nodes.pop();
704     initialize(x);
705     x -> w = w;
706     x -> refresh();
707     return x;
708 }
709
710 // 封装初始化函数
711 // 记得在进行操作之前对所有结点调用一遍
712 inline void initialize(node *x) {
713     *x = node();
714     x -> ch[0] = x -> ch[1] = x -> p = null;
715 }
716
717 // 注意一下在Access的同时更新子树信息的方法
718 node *access(node *x) {
719     node *y = null;
720
721     while (x != null) {
722         splay(x);
723
724         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
725         ↪ chain_cnt;
726         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
727         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
728         x -> ch[1] = y;
729
730         (y = x) -> refresh();
731         x = x -> p;
732     }
733
734     return y;
735 }
736
737 // 找到一个点所在连通块的根
738 // 对比原版没有变化
739 node *getroot(node *x) {
740     if(!rev)
741     |     return;
742
743     ch[0]->rev ^= true;
744     ch[1]->rev ^= true;
745     swap(ch[0], ch[1]);
746     swap(suml, sumr);
747
748    rev = false;
749 }
750
751 inline void refresh() { // 如果不想这样特判
752     ↪ 就pushdown一下
753     // pushdown();
754
755    sum = ch[0] -> sum + ch[1] -> sum + w;
756    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
757    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
758    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
759    ↪ (ch[0] -> sum + w) + tree_sum;
760
761    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
762    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
763    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
764    ↪ (ch[1] -> sum + w) + tree_sum;
765
766    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
767    ↪ chain_cnt + tree_cnt;
768 }
769 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
770
771 // 封装构造函数
772 node *newnode(int w) {
773     node *x = nodes.front(); // 因为有删边加边, 可以用一
774     ↪ 个队列维护可用结点
775     nodes.pop();
776     initialize(x);
777     x -> w = w;
778     x -> refresh();
779     return x;
780 }
781
782 // 封装初始化函数
783 // 记得在进行操作之前对所有结点调用一遍
784 inline void initialize(node *x) {
785     *x = node();
786     x -> ch[0] = x -> ch[1] = x -> p = null;
787 }
788
789 // 注意一下在Access的同时更新子树信息的方法
790 node *access(node *x) {
791     node *y = null;
792
793     while (x != null) {
794         splay(x);
795
796         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
797         ↪ chain_cnt;
798         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
799         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
800         x -> ch[1] = y;
801
802         (y = x) -> refresh();
803         x = x -> p;
804     }
805
806     return y;
807 }
808
809 // 找到一个点所在连通块的根
810 // 对比原版没有变化
811 node *getroot(node *x) {
812     if(!rev)
813     |     return;
814
815     ch[0]->rev ^= true;
816     ch[1]->rev ^= true;
817     swap(ch[0], ch[1]);
818     swap(suml, sumr);
819
820    rev = false;
821 }
822
823 inline void refresh() { // 如果不想这样特判
824     ↪ 就pushdown一下
825     // pushdown();
826
827    sum = ch[0] -> sum + ch[1] -> sum + w;
828    suml = (ch[0] -> rev ? ch[0] -> sumr : ch[0] ->
829    ↪ suml) + (ch[1] -> rev ? ch[1] -> sumr : ch[1]
830    ↪ -> suml) + (tree_cnt + ch[1] -> chain_cnt) *
831    ↪ (ch[0] -> sum + w) + tree_sum;
832
833    sumr = (ch[0] -> rev ? ch[0] -> suml : ch[0] ->
834    ↪ sumr) + (ch[1] -> rev ? ch[1] -> suml : ch[1]
835    ↪ -> sumr) + (tree_cnt + ch[0] -> chain_cnt) *
836    ↪ (ch[1] -> sum + w) + tree_sum;
837
838    chain_cnt = ch[0] -> chain_cnt + ch[1] ->
839    ↪ chain_cnt + tree_cnt;
840 }
841 } null[maxn * 2]; // 如果没有边权变点权就不用乘2了
842
843 // 封装构造函数
844 node *newnode(int w) {
845     node *x = nodes.front(); // 因为有删边加边, 可以用一
846     ↪ 个队列维护可用结点
847     nodes.pop();
848     initialize(x);
849     x -> w = w;
850     x -> refresh();
851     return x;
852 }
853
854 // 封装初始化函数
855 // 记得在进行操作之前对所有结点调用一遍
856 inline void initialize(node *x) {
857     *x = node();
858     x -> ch[0] = x -> ch[1] = x -> p = null;
859 }
860
861 // 注意一下在Access的同时更新子树信息的方法
862 node *access(node *x) {
863     node *y = null;
864
865     while (x != null) {
866         splay(x);
867
868         x -> tree_cnt += x -> ch[1] -> chain_cnt - y ->
869         ↪ chain_cnt;
870         x -> tree_sum += (x -> ch[1] -> rev ? x -> ch[1] ->
871         ↪ sumr : x -> ch[1] -> suml) - y -> suml;
872         x -> ch[1] = y;
873
874         (y = x) -> refresh();
875         x = x -> p;
876     }
877
878     return y;
879 }
880
881 // 找到一个点所在连通块的根
882 // 对比原版没有变化
883 node *getroot(node *x) {
884     if(!rev)
885     |     return;
886
887     ch[0]->rev ^= true;
888     ch[1]->rev ^= true;
889     swap(ch[0], ch[1]);
890     swap(suml, sumr);
891
892    rev = false;
893 }
894
895 inline void refresh() { // 如果不想这样特判
89
```

```

149     }
150 }
151
152 // 旋转函数
153 // 对比原版没有变化
154 void rot(node *x, int d) {
155     node *y = x -> ch[d ^ 1];
156
157     if ((x -> ch[d ^ 1] = y -> ch[d]) != null)
158         y -> ch[d] -> p = x;
159
160     y -> p = x -> p;
161     if (!isroot(x))
162         x -> p -> ch[dir(x)] = y;
163
164     (y -> ch[d] = x) -> p = y;
165
166     x -> refresh();
167     y -> refresh();
168 }

```

4.6.4 模板题:动态QTREE4(询问树上相距最远点)

```

1 #include<bits/stdc++.h>
2 #include<ext/pb_ds/assoc_container.hpp>
3 #include<ext/pb_ds/tree_policy.hpp>
4 #include<ext/pb_ds/priority_queue.hpp>
5
6 #define isroot(x) ((x)->p==null||((x)!=(x)->p-
   ↳ >ch[0]&&(x)!=(x)->p->ch[1]))
7 #define dir(x) ((x)==(x)->p->ch[1])
8
9 using namespace std;
10 using namespace __gnu_pbds;
11
12 const int maxn=100010;
13 const long long INF=1000000000000000001;
14
15 struct binary_heap{
16     __gnu_pbds::priority_queue<long long,less<long
   ↳ long>,binary_heap_tag>q1,q2;
17     binary_heap(){
18         void push(long long x){if(x>(-INF)>>2)q1.push(x);}
19         void erase(long long x){if(x>(-INF)>>2)q2.push(x);}
20         long long top(){
21             if(empty())return -INF;
22             while(!q2.empty()&&q1.top()==q2.top()){
23                 q1.pop();
24                 q2.pop();
25             }
26             return q1.top();
27         }
28         long long top2(){
29             if(size(<2)return -INF;
30             long long a=top();
31             erase(a);
32             long long b=top();
33             push(a);
34             return a+b;
35         }
36         int size(){return q1.size()-q2.size();}
37         bool empty(){return q1.size()==q2.size();}
38 }heap;//全局堆维护每条链的最大子段和
39 struct node{
40     long long sum,maxsum,prefix,suffix;
41     int key;
42     binary_heap heap;//每个点的堆存的是它的子树中到它的
   ↳ 最远距离b如果是黑点的话还会包括自己
43     node *ch[2],*p;
44     bool rev;
45     node(int k=0):sum(k),maxsum(-INF),prefix(-INF),

```

```

46     suffix(-INF),key(k),rev(false){}
47     inline void pushdown(){
48         if(!rev)return;
49         ch[0]->rev^=true;
50         ch[1]->rev^=true;
51         swap(ch[0],ch[1]);
52         swap(prefix,suffix);
53         rev=false;
54     }
55     inline void refresh(){
56         pushdown();
57         ch[0]->pushdown();
58         ch[1]->pushdown();
59         sum=ch[0]->sum+ch[1]->sum+key;
60         prefix=max(ch[0]->prefix,
61             ch[0]->sum+key+ch[1]->prefix);
62         suffix=max(ch[1]->suffix,
63             ch[1]->sum+key+ch[0]->suffix);
64         maxsum=max(max(ch[0]->maxsum,ch[1]->maxsum),
65             ch[0]->suffix+key+ch[1]->prefix);
66         if(!heap.empty()){
67             prefix=max(prefix,
68                 ch[0]->sum+key+heap.top());
69             suffix=max(suffix,
70                 ch[1]->sum+key+heap.top());
71             maxsum=max(maxsum,max(ch[0]->suffix,
72                 ch[1]->prefix)+key+heap.top());
73             if(heap.size(>1){
74                 maxsum=max(maxsum,heap.top2()+key);
75             }
76         }
77     }
78 }null[maxn<<1],*ptr=null;
79 void addedge(int,int,int);
80 void deledge(int,int);
81 void modify(int,int,int);
82 void modify_color(int);
83 node *newnode(int);
84 node *access(node*);
85 void makeroot(node*);
86 void link(node*,node*);
87 void cut(node*,node*);
88 void splay(node*);
89 void rot(node*,int);
90 queue<node*>freenodes;
91 tree<pair<int,int>,node*>mp;
92 bool col[maxn]={false};
93 char c;
94 int n,m,k,x,y,z;
95 int main(){
96     null->ch[0]=null->ch[1]=null->p=null;
97     scanf("%d%d%d",&n,&m,&k);
98     for(int i=1;i<=n;i++){
99         newnode(0);
100     }
101     heap.push(0);
102     while(k--){
103         scanf("%d",&x);
104         col[x]=true;
105         null[x].heap.push(0);
106     }
107     for(int i=1;i<n;i++){
108         scanf("%d%d%d",&x,&y,&z);
109         if(x>y)swap(x,y);
110         addedge(x,y,z);
111     }
112     while(m--){
113         scanf(" %c%d",&c,&x);
114         if(c=='A'){
115             scanf("%d",&y);
116             if(x>y)swap(x,y);
117             deledge(x,y);

```

```

118     }
119     else if(c=='B'){
120         scanf("%d%d",&y,&z);
121         if(x>y)swap(x,y);
122         addedge(x,y,z);
123     }
124     else if(c=='C'){
125         scanf("%d%d",&y,&z);
126         if(x>y)swap(x,y);
127         modify(x,y,z);
128     }
129     else modify_color(x);
130     printf("%lld\n",(heap.top()>0?heap.top():-1));
131 }
132 return 0;
133 }
134 void addedge(int x,int y,int z){
135     node *tmp;
136     if(freenodes.empty())tmp=newnode(z);
137     else{
138         tmp=freenodes.front();
139         freenodes.pop();
140         *tmp=node(z);
141     }
142     tmp->ch[0]=tmp->ch[1]=tmp->p=null;
143     heap.push(tmp->maxsum);
144     link(tmp,null+x);
145     link(tmp,null+y);
146     mp[make_pair(x,y)]=tmp;
147 }
148 void deledge(int x,int y){
149     node *tmp=mp[make_pair(x,y)];
150     cut(tmp,null+x);
151     cut(tmp,null+y);
152     freenodes.push(tmp);
153     heap.erase(tmp->maxsum);
154     mp.erase(make_pair(x,y));
155 }
156 void modify(int x,int y,int z){
157     node *tmp=mp[make_pair(x,y)];
158     makeroot(tmp);
159     tmp->pushdown();
160     heap.erase(tmp->maxsum);
161     tmp->key=z;
162     tmp->refresh();
163     heap.push(tmp->maxsum);
164 }
165 void modify_color(int x){
166     makeroot(null+x);
167     col[x]^=true;
168     if(col[x])null[x].heap.push(0);
169     else null[x].heap.erase(0);
170     heap.erase(null[x].maxsum);
171     null[x].refresh();
172     heap.push(null[x].maxsum);
173 }
174 node *newnode(int k){
175     *(++ptr)=node(k);
176     ptr->ch[0]=ptr->ch[1]=ptr->p=null;
177     return ptr;
178 }
179 node *access(node *x){
180     splay(x);
181     heap.erase(x->maxsum);
182     x->refresh();
183     if(x->ch[1]!=null){
184         x->ch[1]->pushdown();
185         x->heap.push(x->ch[1]->prefix);
186         x->refresh();
187         heap.push(x->ch[1]->maxsum);
188     }
189     x->ch[1]=null;

```

```

190     x->refresh();
191     node *y=x;
192     x=x->p;
193     while(x!=null){
194         splay(x);
195         heap.erase(x->maxsum);
196         if(x->ch[1]!=null){
197             x->ch[1]->pushdown();
198             x->heap.push(x->ch[1]->prefix);
199             heap.push(x->ch[1]->maxsum);
200         }
201         x->heap.erase(y->prefix);
202         x->ch[1]=y;
203         (y=x)->refresh();
204         x=x->p;
205     }
206     heap.push(y->maxsum);
207     return y;
208 }
209 void makeroot(node *x){
210     access(x);
211     splay(x);
212     x->rev^=true;
213 }
214 void link(node *x,node *y){//新添一条虚边维护y对应的堆
215     makeroot(x);
216     makeroot(y);
217     x->pushdown();
218     x->p=y;
219     heap.erase(y->maxsum);
220     y->heap.push(x->prefix);
221     y->refresh();
222     heap.push(y->maxsum);
223 }
224 void cut(node *x,node *y){//断开一条实边一条链变成两条
    //链需要维护全局堆
225     makeroot(x);
226     access(y);
227     splay(y);
228     heap.erase(y->maxsum);
229     heap.push(y->ch[0]->maxsum);
230     y->ch[0]->p=null;
231     y->ch[0]=null;
232     y->refresh();
233     heap.push(y->maxsum);
234 }
235 void splay(node *x){
236     x->pushdown();
237     while(!isroot(x)){
238         if(!isroot(x->p))
239             x->p->p->pushdown();
240         x->p->pushdown();
241         x->pushdown();
242         if(isroot(x->p)){
243             rot(x->p,dir(x)^1);
244             break;
245         }
246         if(dir(x)==dir(x->p))
247             rot(x->p->p,dir(x->p)^1);
248         else rot(x->p,dir(x)^1);
249         rot(x->p,dir(x)^1);
250     }
251 }
252 void rot(node *x,int d){
253     node *y=x->ch[d^1];
254     if((x->ch[d^1]=y->ch[d])!=null)
255         y->ch[d]->p=x;
256     y->p=x->p;
257     if(!isroot(x))
258         x->p->ch[dir(x)]=y;
259     (y->ch[d]=x)->p=y;
260     x->refresh();

```



```

261     y->refresh();
262 }

```

4.7 虚树

```

1  #include<cstdio>
2  #include<cstring>
3  #include<algorithm>
4  #include<vector>
5  using namespace std;
6  const int maxn=100005;
7  struct Tree{
8      vector<int>G[maxn],W[maxn];
9      int p[maxn],d[maxn],size[maxn],mn[maxn],mx[maxn];
10     bool col[maxn];
11     long long ans_sum;
12     int ans_min,ans_max;
13     void add(int x,int y,int z){
14         G[x].push_back(y);
15         W[x].push_back(z);
16     }
17     void dfs(int x){
18         size[x]=col[x];
19         mx[x]=(col[x]?d[x]:-0x3f3f3f3f);
20         mn[x]=(col[x]?d[x]:0x3f3f3f3f);
21         for(int i=0;i<(int)G[x].size();i++){
22             d[G[x][i]]=d[x]+W[x][i];
23             dfs(G[x][i]);
24             ans_sum+=(long long)size[x]*size[G[x]
25                 ↳ [i]]*d[x];
26             ans_max=max(ans_max,mx[x]+mx[G[x]
27                 ↳ [i]]-(d[x]<<1));
28             ans_min=min(ans_min,mn[x]+mn[G[x]
29                 ↳ [i]]-(d[x]<<1));
30             size[x]+=size[G[x][i]];
31             mx[x]=max(mx[x],mx[G[x][i]]);
32             mn[x]=min(mn[x],mn[G[x][i]]);
33         }
34     }
35     void clear(int x){
36         G[x].clear();
37         W[x].clear();
38         col[x]=false;
39     }
40     void solve(int rt){
41         ans_sum=0;
42         ans_max=1<<31;
43         ans_min=(~0u)>>1;
44         dfs(rt);
45         ans_sum<=1;
46     }
47 }virtree;
48 void dfs(int);
49 int LCA(int,int);
50 vector<int>G[maxn];
51 int f[maxn][20],d[maxn],dfn[maxn],tim=0;
52 bool cmp(int x,int y){return dfn[x]<dfn[y];}
53 int n,m,lgn=0,a[maxn],s[maxn],v[maxn];
54 int main(){
55     scanf("%d",&n);
56     for(int i=1,x,y;i<n;i++){
57         scanf("%d%d",&x,&y);
58         G[x].push_back(y);
59         G[y].push_back(x);
60     }
61     G[n+1].push_back(1);
62     dfs(n+1);
63     for(int i=1;i<=n+1;i++)G[i].clear();

```

```

61     lgn--;
62     for(int j=1;j<=lgn;j++)for(int i=1;i<=n;i++)f[i]
63         ↳ [j]=f[f[i][j-1]][j-1];
64     scanf("%d",&m);
65     while(m--){
66         int k;
67         scanf("%d",&k);
68         for(int i=1;i<=k;i++)scanf("%d",&a[i]);
69         sort(a+1,a+k+1,cmp);
70         int top=0,cnt=0;
71         s[++top]=v[++cnt]=n+1;
72         long long ans=0;
73         for(int i=1;i<=k;i++){
74             virtree.col[a[i]]=true;
75             ans+=d[a[i]]-1;
76             int u=LCA(a[i],s[top]);
77             if(s[top]!=u){
78                 while(top>1&&d[s[top-1]]>=d[u]){
79                     virtree.add(s[top-1],s[top],d[s[top]]-d[s[top-1]]);
80                     top--;
81                 }
82                 if(s[top]!=u){
83                     virtree.add(u,s[top],d[s[top]]-d[u]);
84                     s[top]=v[++cnt]=u;
85                 }
86             }
87             s[++top]=a[i];
88         }
89         for(int
90             ↳ i=top-1;i;i--)virtree.add(s[i],s[i+1],d[s[i+1]]-d[s[i]]);
91         virtree.solve(n+1);
92         ans*=k-1;
93         printf("%lld %d
94             ↳ %d\n",ans-virtree.ans_sum,virtree.ans_min,virtree.ans_max);
95         for(int i=1;i<=k;i++)virtree.clear(a[i]);
96         for(int i=1;i<=cnt;i++)virtree.clear(v[i]);
97     }
98     return 0;
99 }
100 void dfs(int x){
101     dfn[x]=++tim;
102     d[x]=d[f[x][0]]+1;
103     while((1<<lgn)<d[x])lgn++;
104     for(int i=0;i<(int)G[x].size();i++){
105         if(G[x][i]!=f[x][0]){
106             f[G[x][i]][0]=x;
107             dfs(G[x][i]);
108         }
109     }
110 }
111 int LCA(int x,int y){
112     if(d[x]!=d[y]){
113         if(d[x]<d[y])swap(x,y);
114         for(int
115             ↳ i=lgn;i>=0;i--)if(((d[x]-d[y])>>i)&1)x=f[x][i];
116         return x;
117     }
118     if(x==y)return x;
119     for(int i=lgn;i>=0;i--)if(f[x][i]!=f[y][i]){
120         x=f[x][i];
121         y=f[y][i];
122     }
123     return f[x][0];
124 }

```


4.8 长链剖分

```

1 // 顾名思义, 长链剖分是取最深的儿子作为重儿子
2
3 // O(n)维护以深度为下标的子树信息
4 vector<int> G[maxn], v[maxn];
5 int n, p[maxn], h[maxn], son[maxn], ans[maxn];
6
7 // 原题题意: 求每个点的子树中与它距离是几的点最多, 相同的
8 //   ↳ 取最大深度
9 //   ↳ 由于vector只能在后面加入元素, 为了写代码方便, 这里反
10 //   ↳ 过来存
11 void dfs(int x) {
12     h[x] = 1;
13
14     for (int y : G[x])
15         if (y != p[x]){
16             p[y] = x;
17             dfs(y);
18
19             if (h[y] > h[son[x]])
20                 son[x] = y;
21         }
22
23     if (!son[x]) {
24         v[x].push_back(1);
25         ans[x] = 0;
26         return;
27     }
28
29     h[x] = h[son[x]] + 1;
30     swap(v[x], v[son[x]]);
31
32     if (v[x][ans[son[x]]] == 1)
33         ans[x] = h[x] - 1;
34     else
35         ans[x] = ans[son[x]];
36
37     v[x].push_back(1);
38
39     int mx = v[x][ans[x]];
40     for (int y : G[x])
41         if (y != p[x] && y != son[x]) {
42             for (int j = 1; j <= h[y]; j++) {
43                 v[x][h[x] - j - 1] += v[y][h[y] - j];
44
45                 int t = v[x][h[x] - j - 1];
46                 if (t > mx || (t == mx && h[x] - j - 1 >
47                     ↳ ans[x])) {
48                     mx = t;
49                     ans[x] = h[x] - j - 1;
50                 }
51             }
52             v[y].clear();
53         }
54 }

```

4.9 梯子剖分

```

1 // 在线求一个点的第k祖先  $O(n \log n) - O(1)$ 
2 // 理论基础: 任意一个点x的k级祖先y所在长链长度一定  $\geq k$ 
3
4 // 全局数组定义
5 vector<int> G[maxn], v[maxn];
6 int d[maxn], mxd[maxn], son[maxn], top[maxn], len[maxn];
7 int f[19][maxn], log_tbl[maxn];
8
9 // 在主函数中两遍dfs之后加上如下预处理

```

```

10 log_tbl[0] = -1;
11 for (int i = 1; i <= n; i++)
12     log_tbl[i] = log_tbl[i / 2] + 1;
13 for (int j = 1; (1 << j) < n; j++)
14     for (int i = 1; i <= n; i++)
15         f[j][i] = f[j - 1][f[j - 1][i]];
16
17 // 第一遍dfs, 用于计算深度和找出重儿子
18 void dfs1(int x) {
19     mxd[x] = d[x];
20
21     for (int y : G[x])
22         if (y != f[0][x]){
23             f[0][y] = x;
24             d[y] = d[x] + 1;
25
26             dfs1(y);
27
28             mxd[x] = max(mxd[x], mxd[y]);
29             if (mxd[y] > mxd[son[x]])
30                 son[x] = y;
31         }
32 }
33
34 // 第二遍dfs, 用于进行剖分和预处理梯子剖分(每条链向上延
   ↳ 伸一倍)数组
35 void dfs2(int x) {
36     top[x] = (x == son[f[0][x]] ? top[f[0][x]] : x);
37
38     for (int y : G[x])
39         if (y != f[0][x])
40             dfs2(y);
41
42     if (top[x] == x) {
43         int u = x;
44         while (top[son[u]] == x)
45             u = son[u];
46
47         len[x] = d[u] - d[x];
48         for (int i = 0; i < len[x]; i++, u = f[0][u])
49             v[x].push_back(u);
50
51         u = x;
52         for (int i = 0; i < len[x] && u; i++, u = f[0][
   ↳ u])
53             v[x].push_back(u);
54     }
55 }
56
57 // 在线询问x的k级祖先 O(1)
58 // 不存在时返回0
59 int query(int x, int k) {
60     if (!k)
61         return x;
62     if (k > d[x])
63         return 0;
64
65     x = f[log_tbl[k]][x];
66     k ^= 1 << log_tbl[k];
67     return v[top[x]][d[top[x]] + len[top[x]] - d[x] + k];
68 }

```

4.10 左偏树

(参见k短路)

4.11 常见根号思路

通用

- 出现次数大于 \sqrt{n} 的数不会超过 \sqrt{n} 个
- 对于带修改问题, 如果不方便分治或者二进制分组, 可以考虑对操作分块, 每次查询时暴力最后的 \sqrt{n} 个修改并更正答案
- 根号分治: 如果分治时每个子问题需要 $O(N)$ (N 是全局问题的大小), 而规模较小的子问题可以 $O(n^2)$ 解决, 则可以使用根号分治
 - 规模大于 \sqrt{n} 的子问题用 $O(N)$ 的方法解决, 规模小于 \sqrt{n} 的子问题用 $O(n^2)$ 暴力
 - 规模大于 \sqrt{n} 的子问题最多只有 \sqrt{n} 个
 - 规模不大于 \sqrt{n} 的子问题大小的平方和也必定不会超过 $n\sqrt{n}$
- 如果输入规模之和不大于 n (例如给定多个小字符串与大字符串进行询问), 那么规模超过 \sqrt{n} 的问题最多只有 \sqrt{n} 个

```

16         fail[i + 1] = 0;
17     }
18 }
19
20 int KMP() {
21     int cnt = 0, j = 0;
22
23     for (int i = 0; i < n; i++) {
24         while (j && s[i] != t[j])
25             j = fail[j];
26
27         if (s[i] == t[j])
28             j++;
29         if (j == m)
30             cnt++;
31     }
32
33     return cnt;
34 }

```

序列

- 某些维护序列的问题可以用分块/块状链表维护
- 对于静态区间询问问题, 如果可以快速将左/右端点移动一位, 可以考虑莫队
 - 如果强制在线可以分块预处理, 但是一般空间需要 $n\sqrt{n}$
 - * 例题: 询问区间中有几种数出现次数恰好为 k , 强制在线
 - 如果带修改可以试着想一想带修莫队, 但是复杂度高达 $n^{\frac{5}{3}}$
- 线段树可以解决的问题也可以用分块来做到 $O(1)$ 询问或是 $O(1)$ 修改, 具体要看哪种操作更多

树

- 与序列类似, 树上也有树分块和树上莫队
 - 树上带修莫队很麻烦, 常数也大, 最好不要先考虑
 - 树分块不要想当然
- 树分治也可以套根号分治, 道理是一样的

字符串

- 循环节长度大于 \sqrt{n} 的子串最多只有 $O(n)$ 个, 如果是极长子串则只有 $O(\sqrt{n})$ 个

5. 字符串

5.1 KMP

```

1 char s[maxn], t[maxn];
2 int fail[maxn];
3 int n, m;
4
5 void init() {
6     // memset(fail, 0, sizeof(fail));
7
8     for (int i = 1; i < m; i++) {
9         int j = fail[i];
10        while (j && t[i] != t[j])
11            j = fail[j];
12
13        if (t[i] == t[j])
14            fail[i + 1] = j + 1;
15        else

```

5.1.1 ex-KMP

```

1 //全局变量与数组定义
2 char s[maxn], t[maxn];
3 int n, m, a[maxn];
4
5 // 主过程  $O(n + m)$ 
6 // 把t的每个后缀与s的LCP输出到a中, s的后缀和自己的LCP存
7 // 在nx中
8 //  $\theta$ -based, s的长度是m, t的长度是n
9 void exKMP(const char *s, const char *t, int *a) {
10     static int nx[maxn];
11
12     memset(nx, 0, sizeof(nx));
13
14     int j = 0;
15     while (j + 1 < m && s[j] == s[j + 1])
16         j++;
17     nx[1] = j;
18
19     for (int i = 2, k = 1; i < m; i++) {
20         int pos = k + nx[k], len = nx[i - k];
21
22         if (i + len < pos)
23             nx[i] = len;
24         else {
25             j = max(pos - i, 0);
26             while (i + j < m && s[j] == s[i + j])
27                 j++;
28
29             nx[i] = j;
30             k = i;
31         }
32     }
33
34     j = 0;
35     while (j < n && j < m && s[j] == t[j])
36         j++;
37     a[0] = j;
38
39     for (int i = 1, k = 0; i < n; i++) {
40         int pos = k + a[k], len = nx[i - k];
41         if (i + len < pos)
42             a[i] = len;
43         else {
44             j = max(pos - i, 0);
45             while (j < m && i + j < n && s[j] == t[i + j])
46                 j++;

```

```

47     a[i] = j;
48     k = i;
49 }
50 }
51 }

```

5.2 AC自动机

```

1 // Aho-Corasick Automata AC自动机
2 // By AntiLeaf
3 // 通过题目 P3881 Divljak
4
5
6 // 全局变量与数组定义
7 int ch[maxm][26] = {{0}}, f[maxm][26] = {{0}}, q[maxm] =
    ↳ {0}, sum[maxm] = {0}, cnt = 0;
8
9
10 // 在字典树中插入一个字符串 O(n)
11 int insert(const char *c) {
12     int x = 0;
13     while (*c) {
14         if (!ch[x][*c - 'a'])
15             ch[x][*c - 'a'] = ++cnt;
16         x = ch[x][*c++ - 'a'];
17     }
18     return x;
19 }
20
21
22 // 建AC自动机 O(n*sigma)
23 void getfail() {
24     int x, head = 0, tail = 0;
25
26     for (int c = 0; c < 26; c++)
27         if (ch[0][c])
28             q[tail++] = ch[0][c]; // 把根节点的儿子加入队
                ↳ 列
29
30     while (head != tail) {
31         x = q[head++];
32
33         G[f[x][0]].push_back(x);
34         fill(f[x] + 1, f[x] + 26, cnt + 1);
35
36         for (int c = 0; c < 26; c++) {
37             if (ch[x][c]) {
38                 int y = f[x][0];
39
40                 while (y && !ch[y][c])
41                     y = f[y][0];
42
43                 f[ch[x][c][0]] = ch[y][c];
44                 q[tail++] = ch[x][c];
45             }
46             else
47                 ch[x][c] = ch[f[x][0]][c];
48         }
49         fill(f[0], f[0] + 26, cnt + 1);
50     }
51 }

```

5.3 后缀数组

5.3.1 SA-IS

```

1 // 注意求完的SA有效位只有1~n, 但它是0-based, 如果其他部
    ↳ 分是1-based记得+1再用
2
3 constexpr int maxn = 100005, l_type = 0, s_type = 1;

```

```

4
5 // 判断一个字符是否为LMS字符
6 bool is_lms(int *tp, int x) {
7     return x > 0 && tp[x] == s_type && tp[x - 1] ==
    ↳ l_type;
8 }
9
10 // 判断两个LMS子串是否相同
11 bool equal_substr(int *s, int x, int y, int *tp) {
12     do {
13         if (s[x] != s[y])
14             return false;
15         x++;
16         y++;
17     } while (!is_lms(tp, x) && !is_lms(tp, y));
18
19     return s[x] == s[y];
20 }
21
22 // 诱导排序(从*型诱导到L型, 从L型诱导到S型)
23 // 调用之前应将*型按要求放入SA中
24 void induced_sort(int *s, int *sa, int *tp, int *buc, int
    ↳ *lbuc, int *sbuc, int n, int m) {
25     for (int i = 0; i <= n; i++)
26         if (sa[i] > 0 && tp[sa[i] - 1] == l_type)
27             sa[lbuc[s[sa[i] - 1]]++] = sa[i] - 1;
28
29     for (int i = 1; i <= m; i++)
30         sbuc[i] = buc[i] - 1;
31
32     for (int i = n; ~i; i--)
33         if (sa[i] > 0 && tp[sa[i] - 1] == s_type)
34             sa[sbuc[s[sa[i] - 1]]--] = sa[i] - 1;
35 }
36
37 // s是输入字符串, n是字符串的长度, m是字符集的大小
38 int *sais(int *s, int len, int m) {
39     int n = len - 1;
40
41     int *tp = new int[n + 1];
42     int *pos = new int[n + 1];
43     int *name = new int[n + 1];
44     int *sa = new int[n + 1];
45     int *buc = new int[m + 1];
46     int *lbuc = new int[m + 1];
47     int *sbuc = new int[m + 1];
48
49     memset(buc, 0, sizeof(int) * (m + 1));
50
51     for (int i = 0; i <= n; i++)
52         buc[s[i]]++;
53
54     for (int i = 1; i <= m; i++) {
55         buc[i] += buc[i - 1];
56
57         lbuc[i] = buc[i - 1];
58         sbuc[i] = buc[i] - 1;
59     }
60
61     tp[n] = s_type;
62     for (int i = n - 1; ~i; i--) {
63         if (s[i] < s[i + 1])
64             tp[i] = s_type;
65         else if (s[i] > s[i + 1])
66             tp[i] = l_type;
67         else
68             tp[i] = tp[i + 1];
69     }
70 }

```

```

71  int cnt = 0;
72  for (int i = 1; i <= n; i++)
73      if (tp[i] == s_type && tp[i - 1] == l_type)
74          pos[cnt++] = i;
75
76  memset(sa, -1, sizeof(int) * (n + 1));
77  for (int i = 0; i < cnt; i++)
78      sa[sbuc[s[pos[i]]]--] = pos[i];
79  induced_sort(s, sa, tp, buc, lbuc, sbuc, n, m);
80
81  memset(name, -1, sizeof(int) * (n + 1));
82  int lastx = -1, namecnt = 1;
83  bool flag = false;
84
85  for (int i = 1; i <= n; i++) {
86      int x = sa[i];
87
88      if (is_lms(tp, x)) {
89          if (lastx >= 0 && !equal_substr(s, x, lastx,
90              ↳ tp))
91              namecnt++;
92
93          if (lastx >= 0 && namecnt == name[lastx])
94              flag = true;
95
96          name[x] = namecnt;
97          lastx = x;
98      }
99      name[n] = 0;
100
101  int *t = new int[cnt];
102  int p = 0;
103  for (int i = 0; i <= n; i++)
104      if (name[i] >= 0)
105          t[p++] = name[i];
106
107  int *tsa;
108  if (!flag) {
109      tsa = new int[cnt];
110
111      for (int i = 0; i < cnt; i++)
112          tsa[t[i]] = i;
113  }
114  else
115      tsa = sais(t, cnt, namecnt);
116
117  lbuc[0] = sbuc[0] = 0;
118  for (int i = 1; i <= m; i++) {
119      lbuc[i] = buc[i - 1];
120      sbuc[i] = buc[i] - 1;
121  }
122
123  memset(sa, -1, sizeof(int) * (n + 1));
124  for (int i = cnt - 1; ~i; i--)
125      sa[sbuc[s[pos[tsa[i]]]--] = pos[tsa[i]];
126  induced_sort(s, sa, tp, buc, lbuc, sbuc, n, m);
127
128  return sa;
129 }
130
131 // O(n)求height数组, 注意是sa[i]与sa[i - 1]的LCP
132 void get_height(int *s, int *sa, int *rnk, int *height,
133     ↳ int n) {
134     for (int i = 0; i <= n; i++)
135         rnk[sa[i]] = i;
136
137     int k = 0;
138     for (int i = 0; i <= n; i++) {

```

```

138         if (!rnk[i])
139             continue;
140
141         if (k)
142             k--;
143
144         while (s[sa[rnk[i]] + k] == s[sa[rnk[i] - 1] +
145             ↳ k])
146             k++;
147
148         height[rnk[i]] = k;
149     }
150
151     char str[maxn];
152     int n, s[maxn], sa[maxn], rnk[maxn], height[maxn];
153
154     // 方便起见附上主函数
155     int main() {
156         scanf("%s", str);
157         n = strlen(str);
158         str[n] = '$';
159
160         for (int i = 0; i <= n; i++)
161             s[i] = str[i];
162
163         memcpy(sa, sais(s, n + 1, 256), sizeof(int) * (n +
164             ↳ 1));
165
166         get_height(s, sa, rnk, height, n);
167
168         return 0;
169     }

```

5.3.2 SAMSA

```

1  bool vis[maxn * 2];
2  char s[maxn];
3  int n, id[maxn * 2], ch[maxn * 2][26], height[maxn], tim
4      ↳ = 0;
5
6  void dfs(int x) {
7      if (id[x]) {
8          height[tim++] = val[last];
9          sa[tim] = id[x];
10
11          last = x;
12      }
13
14      for (int c = 0; c < 26; c++)
15          if (ch[x][c])
16              dfs(ch[x][c]);
17
18      last = par[x];
19  }
20
21 int main() {
22     last = ++cnt;
23
24     scanf("%s", s + 1);
25     n = strlen(s + 1);
26
27     for (int i = n; i > 0; i--) {
28         expand(s[i] - 'a');
29         id[last] = i;
30     }
31
32     vis[1] = true;
33     for (int i = 1; i <= cnt; i++)

```

```

33     if (id[i])
34         for (int x = i, pos = n; x && !vis[x]; x =
            ↳ par[x]) {
35             vis[x] = true;
36             pos -= val[x] - val[par[x]];
37             ch[par[x]][s[pos + 1] - 'a'] = x;
38         }
39
40     dfs(1);
41
42     for (int i = 1; i <= n; i++) {
43         if (i > 1)
44             printf(" ");
45         printf("%d", sa[i]); // 1-based
46     }
47     printf("\n");
48
49     for (int i = 1; i < n; i++) {
50         if (i > 1)
51             printf(" ");
52         printf("%d", height[i]);
53     }
54     printf("\n");
55
56     return 0;
57 }

```

5.4 后缀自动机

(广义后缀自动机复杂度就是 $O(n|\Sigma|)$, 也没法做到更低了)

```

1 // 在字符集比较小的时候可以直接开go数组, 否则需要用map或
    ↳ 者哈希表替换
2 // 注意!!!结点数要开成串长的两倍
3
4 // 全局变量与数组定义
5 int last, val[maxn], par[maxn], go[maxn][26], cnt;
6 int c[maxn], q[maxn]; // 用来桶排序
7
8 // 在主函数开头加上这句初始化
9 last = cnt = 1;
10
11 // 以下是按val进行桶排序的代码
12 for (int i = 1; i <= cnt; i++)
13     c[val[i] + 1]++;
14 for (int i = 1; i <= n; i++)
15     c[i] += c[i - 1]; // 这里n是串长
16 for (int i = 1; i <= cnt; i++)
17     q[++c[val[i]]] = i;
18
19 //加入一个字符 均摊O(1)
20 void extend(int c) {
21     int p = last, np = ++cnt;
22     val[np] = val[p] + 1;
23
24     while (p && !go[p][c]) {
25         go[p][c] = np;
26         p = par[p];
27     }
28
29     if (!p)
30         par[np] = 1;
31     else {
32         int q = go[p][c];
33
34         if (val[q] == val[p] + 1)
35             par[np] = q;
36         else {
37             int nq = ++cnt;
38             val[nq] = val[p] + 1;

```

```

39         memcpy(go[nq], go[q], sizeof(go[q]));
40
41         par[nq] = par[q];
42         par[np] = par[q] = nq;
43
44         while (p && go[p][c] == q){
45             go[p][c] = nq;
46             p = par[p];
47         }
48     }
49 }
50
51 last = np;
52 }

```

5.5 回文树

```

1 // 定理: 一个字符串本质不同的回文子串个数是O(n)的
2 // 注意回文树只需要开一倍结点, 另外结点编号也是一个可用
    ↳ 的bfs序
3
4 // 全局数组定义
5 int val[maxn], par[maxn], go[maxn][26], last, cnt;
6 char s[maxn];
7
8 // 重要!在主函数最前面一定要加上以下初始化
9 par[0] = cnt = 1;
10 val[1] = -1;
11 // 这个初始化和广义回文树不一样, 写普通题可以用, 广义回
    ↳ 文树就不要乱搞了
12
13 // extend函数 均摊O(1)
14 // 向后扩展一个字符
15 // 传入对应下标
16 void extend(int n) {
17     int p = last, c = s[n] - 'a';
18     while (s[n - val[p] - 1] != s[n])
19         p = par[p];
20
21     if (!go[p][c]) {
22         int q = ++cnt, now = p;
23         val[q] = val[p] + 2;
24
25         do
26             p = par[p];
27         while (s[n - val[p] - 1] != s[n]);
28
29         par[q] = go[p][c];
30         last = go[now][c] = q;
31     }
32     else
33         last = go[p][c];
34
35     // a[last]++;
36 }

```

5.5.1 广义回文树

(代码是梯子剖分的版本, 压力不大的题目换成直接倍增就好了, 常数只差不到一倍)

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 constexpr int maxn = 1000005, mod = 1000000007;
6

```

```

7  int val[maxn], par[maxn], go[maxn][26], fail[maxn][26],
    ↪ pam_last[maxn], pam_cnt;
8  int weight[maxn], pow_26[maxn];
9
10 int trie[maxn][26], trie_cnt, d[maxn], mxd[maxn],
    ↪ son[maxn], top[maxn], len[maxn], sum[maxn];
11 char chr[maxn];
12 int f[25][maxn], log_tbl[maxn];
13 vector<int> v[maxn];
14
15 vector<int> queries[maxn];
16
17 char str[maxn];
18 int n, m, ans[maxn];
19
20 int add(int x, int c) {
21     if (!trie[x][c]) {
22         trie[x][c] = ++trie_cnt;
23         f[0][trie[x][c]] = x;
24         chr[trie[x][c]] = c + 'a';
25     }
26
27     return trie[x][c];
28 }
29
30 int del(int x) {
31     return f[0][x];
32 }
33
34 void dfs1(int x) {
35     mxd[x] = d[x] = d[f[0][x]] + 1;
36
37     for (int i = 0; i < 26; i++)
38         if (trie[x][i]) {
39             int y = trie[x][i];
40
41             dfs1(y);
42
43             mxd[x] = max(mxd[x], mxd[y]);
44             if (mxd[y] > mxd[son[x]])
45                 son[x] = y;
46         }
47 }
48
49 void dfs2(int x) {
50     if (x == son[f[0][x]])
51         top[x] = top[f[0][x]];
52     else
53         top[x] = x;
54
55     for (int i = 0; i < 26; i++)
56         if (trie[x][i]) {
57             int y = trie[x][i];
58             dfs2(y);
59         }
60
61     if (top[x] == x) {
62         int u = x;
63         while (top[son[u]] == x)
64             u = son[u];
65
66         len[x] = d[u] - d[x];
67
68         for (int i = 0; i < len[x]; i++) {
69             v[x].push_back(u);
70             u = f[0][u];
71         }
72
73         u = x;
74         for (int i = 0; i < len[x]; i++) { // 梯子剖分, 要
            ↪ 延长一倍
75             v[x].push_back(u);
76             u = f[0][u];
77         }
78     }
79 }
80
81 int get_anc(int x, int k) {
82     if (!k)
83         return x;
84     if (k > d[x])
85         return 0;
86
87     x = f[log_tbl[k]][x];
88     k ^= 1 << log_tbl[k];
89
90     return v[top[x]][d[top[x]] + len[top[x]] - d[x] + k];
91 }
92
93 char get_char(int x, int k) { // 查询x前面k个的字符是哪个
94     return chr[get_anc(x, k)];
95 }
96
97 int getfail(int x, int p) {
98     if (get_char(x, val[p] + 1) == chr[x])
99         return p;
100     return fail[p][chr[x] - 'a'];
101 }
102
103 int extend(int x) {
104
105     int p = pam_last[f[0][x]], c = chr[x] - 'a';
106
107     p = getfail(x, p);
108
109     int new_last;
110
111     if (!go[p][c]) {
112         int q = ++pam_cnt, now = p;
113         val[q] = val[p] + 2;
114
115         p = getfail(x, par[p]);
116
117         par[q] = go[p][c];
118         new_last = go[now][c] = q;
119
120         for (int i = 0; i < 26; i++)
121             fail[q][i] = fail[par[q]][i];
122
123         if (get_char(x, val[par[q]]) >= 'a')
124             fail[q][get_char(x, val[par[q]]) - 'a'] =
                ↪ par[q];
125
126         if (val[q] <= n)
127             weight[q] = (weight[par[q]] + (long long)(n -
                ↪ val[q] + 1) * pow_26[n - val[q]]) % mod;
128         else
129             weight[q] = weight[par[q]];
130     }
131     else
132         new_last = go[p][c];
133
134     pam_last[x] = new_last;
135
136     return weight[pam_last[x]];
137 }
138
139 void bfs() {
140
141     queue<int> q;
142
143     q.push(1);
144
145     while (!q.empty()) {

```

```

146     int x = q.front();
147     q.pop();
148
149     sum[x] = sum[f[0][x]];
150     if (x > 1)
151         sum[x] = (sum[x] + extend(x)) % mod;
152
153     for (int i : queries[x])
154         ans[i] = sum[x];
155
156     for (int i = 0; i < 26; i++)
157         if (trie[x][i])
158             q.push(trie[x][i]);
159 }
160
161 }
162
163 int main() {
164
165     pow_26[0] = 1;
166     log_tbl[0] = -1;
167
168     for (int i = 1; i <= 1000000; i++) {
169         pow_26[i] = 26ll * pow_26[i - 1] % mod;
170         log_tbl[i] = log_tbl[i / 2] + 1;
171     }
172
173     int T;
174     scanf("%d", &T);
175
176     while (T--) {
177         scanf("%d%d%s", &n, &m, str);
178
179         trie_cnt = 1;
180         chr[1] = '#';
181
182         int last = 1;
183         for (char *c = str; *c; c++)
184             last = add(last, *c - 'a');
185
186         queries[last].push_back(0);
187
188         for (int i = 1; i <= m; i++) {
189             int op;
190             scanf("%d", &op);
191
192             if (op == 1) {
193                 char c;
194                 scanf("%c", &c);
195
196                 last = add(last, c - 'a');
197             }
198             else
199                 last = del(last);
200
201             queries[last].push_back(i);
202         }
203
204         dfs1(1);
205         dfs2(1);
206
207         for (int j = 1; j <= log_tbl[trie_cnt]; j++)
208             for (int i = 1; i <= trie_cnt; i++)
209                 f[j][i] = f[j - 1][f[j - 1][i]];
210
211         par[0] = pam_cnt = 1;
212
213
214         for (int i = 0; i < 26; i++)
215             fail[0][i] = fail[1][i] = 1;
216
217         val[1] = -1;

```

```

218         pam_last[1] = 1;
219
220         bfs();
221
222         for (int i = 0; i <= m; i++)
223             printf("%d\n", ans[i]);
224
225         for (int j = 0; j <= log_tbl[trie_cnt]; j++)
226             memset(f[j], 0, sizeof(f[j]));
227
228         for (int i = 1; i <= trie_cnt; i++) {
229             chr[i] = 0;
230             d[i] = mxd[i] = son[i] = top[i] = len[i] =
                ↳ pam_last[i] = sum[i] = 0;
231             v[i].clear();
232             queries[i].clear();
233
234             memset(trie[i], 0, sizeof(trie[i]));
235         }
236         trie_cnt = 0;
237
238         for (int i = 0; i <= pam_cnt; i++) {
239             val[i] = par[i] = weight[i];
240
241             memset(go[i], 0, sizeof(go[i]));
242             memset(fail[i], 0, sizeof(fail[i]));
243         }
244         pam_cnt = 0;
245
246     }
247
248     return 0;
249 }

```

5.6 Manacher马拉车

```

1 //n为串长,回文半径输出到p数组中
2 //数组要开串长的两倍
3 void manacher(const char *t, int n) {
4     static char s[maxn * 2];
5
6     for (int i = n; i; i--)
7         s[i * 2] = t[i];
8     for (int i = 0; i <= n; i++)
9         s[i * 2 + 1] = '#';
10
11     s[0] = '$';
12     s[(n + 1) * 2] = '\0';
13     n = n * 2 + 1;
14
15     int mx = 0, j = 0;
16
17     for (int i = 1; i <= n; i++) {
18         p[i] = (mx > i ? min(p[j * 2 - i], mx - i) : 1);
19         while (s[i - p[i]] == s[i + p[i]])
20             p[i]++;
21
22         if (i + p[i] > mx) {
23             mx = i + p[i];
24             j = i;
25         }
26     }
27 }

```

5.7 字符串原理

KMP和AC自动机的fail指针存储的都是它在串或者字典树上的最长后缀, 因此要判断两个前缀是否互为后缀时可以直接用fail指针判断. 当然它不能做子串问题, 也不能做最长公共后缀.

后缀数组利用的主要是LCP长度可以按照字典序做RMQ的性质, 与某个串的后缀长度 \geq 某个值的后缀形成一个区间。另外一个比较好用的性质是本质不同的子串个数 = 所有子串数 - 字典序相邻的串的height。

后缀自动机实际上可以接受的是所有后缀, 如果把中间状态也算上的话就是所有子串。它的fail指针代表的也是当前串的后缀, 不过注意每个状态可以代表很多状态, 只要右端点在right集合中且长度处在 $(val_{par_p}, val_p]$ 中的串都被它代表。

后缀自动机的fail树也就是反串的后缀树。每个结点代表的串和后缀自动机同理, 两个串的后缀长度也就是他们在后缀树上的LCA。

6. 动态规划

6.1 决策单调性 $O(n \log n)$

```
1 int a[maxn], q[maxn], p[maxn], g[maxn]; // 存左端点, 右端
   ↳ 点就是下一个左端点 - 1
2
3 long long f[maxn], s[maxn];
4
5 int n, m;
6
7 long long calc(int l, int r) {
8     if (r < l)
9         return 0;
10
11     int mid = (l + r) / 2;
12     if ((r - l + 1) % 2 == 0)
13         return (s[r] - s[mid]) - (s[mid] - s[l - 1]);
14     else
15         return (s[r] - s[mid]) - (s[mid - 1] - s[l - 1]);
16 }
17
18 int solve(long long tmp) {
19     memset(f, 63, sizeof(f));
20     f[0] = 0;
21
22     int head = 1, tail = 0;
23
24     for (int i = 1; i <= n; i++) {
25         f[i] = calc(1, i);
26         g[i] = 1;
27
28         while (head < tail && p[head + 1] <= i)
29             head++;
30         if (head <= tail) {
31             if (f[q[head]] + calc(q[head] + 1, i) < f[i])
32                 ↳ {
33                     f[i] = f[q[head]] + calc(q[head] + 1, i);
34                     g[i] = g[q[head]] + 1;
35                 }
36             while (head < tail && p[head + 1] <= i + 1)
37                 head++;
38             if (head <= tail)
39                 p[head] = i + 1;
40         }
41         f[i] += tmp;
42
43     int r = n;
44
45     while (head <= tail) {
46         if (f[q[tail]] + calc(q[tail] + 1, p[tail]) >
47             ↳ f[i] + calc(i + 1, p[tail])) {
48             r = p[tail] - 1;
49             tail--;
50         }
51         else if (f[q[tail]] + calc(q[tail] + 1, r) <=
52             ↳ f[i] + calc(i + 1, r)) {
53             if (r < n) {
```

```
51         q[++tail] = i;
52         p[tail] = r + 1;
53     }
54     break;
55 }
56 else {
57     int L = p[tail], R = r;
58     while (L < R) {
59         int M = (L + R) / 2;
60
61         if (f[q[tail]] + calc(q[tail] + 1, M)
62             ↳ <= f[i] + calc(i + 1, M))
63             L = M + 1;
64         else
65             R = M;
66     }
67
68     q[++tail] = i;
69     p[tail] = L;
70
71     break;
72 }
73 if (head > tail) {
74     q[++tail] = i;
75     p[tail] = i + 1;
76 }
77 }
78
79 return g[n];
80 }
```

7. Miscellaneous

7.1 $O(1)$ 快速乘

```
1 // Long double 快速乘
2 // 在两数直接相乘会爆Long Long时才有必要使用
3 // 常数比直接Long Long乘法 + 取模大很多, 非必要不建议
   ↳ 使用
4 long long mul(long long a, long long b, long long p) {
5     a %= p;
6     b %= p;
7     return ((a * b - p * (long long)((long double)a / p *
8         ↳ b + 0.5)) % p + p) % p;
9 }
10 // 指令集快速乘
11 // 试机记得测试能不能过编译
12 inline long long mul(const long long a, const long long
13     ↳ b, const long long p) {
14     long long ans;
15     __asm__ __volatile__ ("tmulq %%rbx\n\tdivq %%rcx\n"
16         ↳ : "=d"(ans) : "a"(a), "b"(b), "c"(p));
17     return ans;
18 }
```

7.2 $O(n^2)$ 高精度

```
1 // 注意如果只需要正数运算的话
2 // 可以只抄英文名的运算函数
3 // 按需自取
4 // 乘法 $O(n^2)$ 除法 $O(10 * n^2)$ 
5
6 const int maxn = 1005;
7
8 struct big_decimal {
9     int a[maxn];
```

```

10     bool negative;
11
12     big_decimal() {
13         memset(a, 0, sizeof(a));
14         negative = false;
15     }
16
17     big_decimal(long long x) {
18         memset(a, 0, sizeof(a));
19         negative = false;
20
21         if (x < 0) {
22             negative = true;
23             x = -x;
24         }
25
26         while (x) {
27             a[++a[0]] = x % 10;
28             x /= 10;
29         }
30     }
31
32     big_decimal(string s) {
33         memset(a, 0, sizeof(a));
34         negative = false;
35
36         if (s == "")
37             return;
38
39         if (s[0] == '-') {
40             negative = true;
41             s = s.substr(1);
42         }
43         a[0] = s.size();
44         for (int i = 1; i <= a[0]; i++)
45             a[i] = s[a[0] - i] - '0';
46
47         while (a[0] && !a[a[0]])
48             a[0]--;
49     }
50
51     void input() {
52         string s;
53         cin >> s;
54         *this = s;
55     }
56
57     string str() const {
58         if (!a[0])
59             return "0";
60
61         string s;
62         if (negative)
63             s = "-";
64
65         for (int i = a[0]; i; i--)
66             s.push_back('0' + a[i]);
67
68         return s;
69     }
70
71     operator string () const {
72         return str();
73     }
74
75     big_decimal operator - () const {
76         big_decimal o = *this;
77         if (a[0])
78             o.negative ^= true;
79
80         return o;
81     }
82
83     friend big_decimal abs(const big_decimal &u) {
84         big_decimal o = u;
85         o.negative = false;
86         return o;
87     }
88
89     big_decimal &operator <= (int k) {
90         a[0] += k;
91
92         for (int i = a[0]; i > k; i--)
93             a[i] = a[i - k];
94
95         for(int i = k; i; i--)
96             a[i] = 0;
97
98         return *this;
99     }
100
101     friend big_decimal operator << (const big_decimal &u,
102         ↪ int k) {
103         big_decimal o = u;
104         return o <= k;
105     }
106
107     big_decimal &operator >= (int k) {
108         if (a[0] < k)
109             return *this = big_decimal(0);
110
111         a[0] -= k;
112         for (int i = 1; i <= a[0]; i++)
113             a[i] = a[i + k];
114
115         for (int i = a[0] + 1; i <= a[0] + k; i++)
116             a[i] = 0;
117
118         return *this;
119     }
120
121     friend big_decimal operator >> (const big_decimal &u,
122         ↪ int k) {
123         big_decimal o = u;
124         return o >>= k;
125     }
126
127     friend int cmp(const big_decimal &u, const
128         ↪ big_decimal &v) {
129         if (u.negative || v.negative) {
130             if (u.negative && v.negative)
131                 return -cmp(-u, -v);
132
133             if (u.negative)
134                 return -1;
135
136             if (v.negative)
137                 return 1;
138         }
139
140         if (u.a[0] != v.a[0])
141             return u.a[0] < v.a[0] ? -1 : 1;
142
143         for (int i = u.a[0]; i; i--)
144             if (u.a[i] != v.a[i])
145                 return u.a[i] < v.a[i] ? -1 : 1;
146
147         return 0;

```



```

270     }
271
272     return decimal_plus(u, v);
273 }
274
275 friend big_decimal operator - (const big_decimal &u,
    ↪ const big_decimal &v) {
276     if (u.negative || v.negative) {
277         if (u.negative && v.negative)
278             return -decimal_minus(-u, -v);
279
280         if (u.negative)
281             return -decimal_plus(-u, v);
282
283         if (v.negative)
284             return decimal_plus(u, -v);
285     }
286
287     return decimal_minus(u, v);
288 }
289
290 friend big_decimal operator * (const big_decimal &u,
    ↪ const big_decimal &v) {
291     if (u.negative || v.negative) {
292         big_decimal o = decimal_multi(abs(u),
    ↪ abs(v));
293
294         if (u.negative ^ v.negative)
295             return -o;
296         return o;
297     }
298
299     return decimal_multi(u, v);
300 }
301
302 big_decimal operator * (long long x) const {
303     if (x >= 10)
304         return *this * big_decimal(x);
305
306     if (negative)
307         return -(*this * x);
308
309     big_decimal o;
310
311     o.a[0] = a[0];
312
313     for (int i = 1; i <= a[0]; i++) {
314         o.a[i] += a[i] * x;
315
316         if (o.a[i] >= 10) {
317             o.a[i + 1] += o.a[i] / 10;
318             o.a[i] %= 10;
319         }
320     }
321
322     if (o.a[a[0] + 1])
323         o.a[0]++;
324
325     return o;
326 }
327
328 friend pair<big_decimal, big_decimal>
    ↪ decimal_div(const big_decimal &u, const
    ↪ big_decimal &v) {
329     if (u.negative || v.negative) {
330         pair<big_decimal, big_decimal> o =
    ↪ decimal_div(abs(u), abs(v));
331
332         if (u.negative ^ v.negative)
333             return make_pair(-o.first, -o.second);

```

```

334         return o;
335     }
336
337     return decimal_divide(u, v);
338 }
339
340 friend big_decimal operator / (const big_decimal &u,
    ↪ const big_decimal &v) { // v不能是0
341     if (u.negative || v.negative) {
342         big_decimal o = abs(u) / abs(v);
343
344         if (u.negative ^ v.negative)
345             return -o;
346         return o;
347     }
348
349     return decimal_divide(u, v).first;
350 }
351
352 friend big_decimal operator % (const big_decimal &u,
    ↪ const big_decimal &v) {
353     if (u.negative || v.negative) {
354         big_decimal o = abs(u) % abs(v);
355
356         if (u.negative ^ v.negative)
357             return -o;
358         return o;
359     }
360
361     return decimal_divide(u, v).second;
362 }
363 };

```

7.3 笛卡尔树

```

1 int s[maxn], root, lc[maxn], rc[maxn];
2
3 int top = 0;
4 s[++top] = root = 1;
5 for (int i = 2; i <= n; i++) {
6     s[top + 1] = 0;
7     while (a[i] < a[s[top]]) // 小根笛卡尔树
8         top--;
9
10    if (top)
11        rc[s[top]] = i;
12    else
13        root = i;
14
15    lc[i] = s[top + 1];
16    s[++top] = i;
17 }

```

7.4 常用NTT素数及原根

| $p = r \times 2^k + 1$ | r | k | 最小原根 |
|------------------------|------|-----|-----------|
| 104857601 | 25 | 22 | 3 |
| 167772161 | 5 | 25 | 3 |
| 469762049 | 7 | 26 | 3 |
| 985661441 | 235 | 22 | 3 |
| 998244353 | 119 | 23 | 3 |
| 1004535809 | 479 | 21 | 3 |
| 1005060097* | 1917 | 19 | 5 |
| 2013265921 | 15 | 27 | 31 |
| 2281701377 | 17 | 27 | 3 |
| 31525197391593473 | 7 | 52 | 3 |
| 180143985094819841 | 5 | 55 | 6 |
| 1945555039024054273 | 27 | 56 | 5 |
| 4179340454199820289 | 29 | 57 | 3 |

*注: 1005060097有点危险, 在变化长度大于 $524288 = 2^{19}$ 时不可用.

7.5 xorshift

```

1 ull k1, k2;
2 const int mod = 100000000;
3 ull xorShift128Plus() {
4     ull k3 = k1, k4 = k2;
5     k1 = k4;
6     k3 ^= (k3 << 23);
7     k2 = k3 ^ k4 ^ (k3 >> 17) ^ (k4 >> 26);
8     return k2 + k4;
9 }
10 void gen(ull _k1, ull _k2) {
11     k1 = _k1, k2 = _k2;
12     int x = xorShift128Plus() % threshold + 1;
13     // do sth
14 }
15
16 uint32_t xor128(void) {
17     static uint32_t x = 123456789;
18     static uint32_t y = 362436069;
19     static uint32_t z = 521288629;
20     static uint32_t w = 88675123;
21     uint32_t t;
22
23     t = x ^ (x << 11);
24     x = y; y = z; z = w;
25     return w = w ^ (w >> 19) ^ (t ^ (t >> 8));
26 }
27
```

7.6 枚举子集

(注意这是 $t \neq 0$ 的写法, 如果可以等于0需要在循环里手动break)

```

1 for (int t = s; t; (--t) &= s) {
2     // do something
3 }

```

7.7 STL

7.7.1 vector

- `vector(int nSize)`: 创建一个vector, 元素个数为nSize
- `vector(int nSize, const T &value)`: 创建一个vector, 元素个数为nSize, 且值均为value
- `vector(begin, end)`: 复制[begin, end)区间内另一个数组的元素到vector中

- `void assign(int n, const T &x)`: 设置向量中前n个元素的值为x
- `void assign(const_iterator first, const_iterator last)`: 向量中[first, last)中元素设置成当前向量元素

7.7.2 list

- `assign()` 给list赋值
- `back()` 返回最后一个元素
- `begin()` 返回指向第一个元素的迭代器
- `clear()` 删除所有元素
- `empty()` 如果list是空的则返回true
- `end()` 返回末尾的迭代器
- `erase()` 删除一个元素
- `front()` 返回第一个元素
- `insert()` 插入一个元素到list中
- `max_size()` 返回list能容纳的最大元素数量
- `merge()` 合并两个list
- `pop_back()` 删除最后一个元素
- `pop_front()` 删除第一个元素
- `push_back()` 在list的末尾添加一个元素
- `push_front()` 在list的头部添加一个元素
- `rbegin()` 返回指向第一个元素的逆向迭代器
- `remove()` 从list删除元素
- `remove_if()` 按指定条件删除元素
- `rend()` 指向list末尾的逆向迭代器
- `resize()` 改变list的大小
- `reverse()` 把list的元素倒转
- `size()` 返回list中的元素个数
- `sort()` 给list排序
- `splice()` 合并两个list
- `swap()` 交换两个list
- `unique()` 删除list中重复的元素

7.8 pb_ds

7.8.1 哈希表

```

1 #include<ext/pb_ds/assoc_container.hpp>
2 #include<ext/pb_ds/hash_policy.hpp>
3 using namespace __gnu_pbds;
4
5 cc_hash_table<string, int> mp1; // 拉链法
6 gp_hash_table<string, int> mp2; // 查探法(快一些)

```

7.8.2 堆

默认也是大根堆, 和std::priority_queue保持一致.

```
1 #include<ext/pb_ds/priority_queue.hpp>
2 using namespace __gnu_pbds;
3
4 __gnu_pbds::priority_queue<int> q;
5 __gnu_pbds::priority_queue<int, greater<int>,
  ↳ pairing_heap_tag> pq;
```

效率参考:

- * 共有五种操作: push、pop、modify、erase、join
- * pairing_heap_tag: push和join为 $O(1)$, 其余为均摊 $\Theta(\log n)$
- * binary_heap_tag: 只支持push和pop, 均为均摊 $\Theta(\log n)$
- * binomial_heap_tag: push为均摊 $O(1)$, 其余为 $\Theta(\log n)$
- * rc_binomial_heap_tag: push为 $O(1)$, 其余为 $\Theta(\log n)$
- * thin_heap_tag: push为 $O(1)$, 不支持join, 其余为 $\Theta(\log n)$; 果只有increase_key, 那么modify为均摊 $O(1)$
- * “不支持”不是不能用, 而是用起来很慢 [csdn.net/TRiddle](https://www.csdn.net/TRiddle)

常用操作:

- push(): 向堆中压入一个元素, 返回迭代器
- pop(): 将堆顶元素弹出
- top(): 返回堆顶元素
- size(): 返回元素个数
- empty(): 返回是否非空
- modify(point_iterator, const key): 把迭代器位置的 key 修改为传入的 key
- erase(point_iterator): 把迭代器位置的键值从堆中删除
- join(__gnu_pbds::priority_queue &other): 把 other 合并到 *this, 并把 other 清空

7.8.3 平衡树

```
1 #include <ext/pb_ds/tree_policy.hpp>
2 #include <ext/pb_ds/assoc_container.hpp>
3 using namespace __gnu_pbds;
4
5 tree<int, null_type, less<int>, rb_tree_tag,
  ↳ tree_order_statistics_node_update> t;
6
7 // rb_tree_tag 红黑树(还有splay_tree_tag和ov_tree_tag, 后
  ↳ 者不知道是什么)
```

注意第五个参数要填tree_order_statistics_node_update才能使用排名操作.

- insert(x): 向树中插入一个元素x, 返回pair<point_iterator, bool>
- erase(x): 从树中删除一个元素/迭代器x, 返回一个 bool 表明是否删除成功
- order_of_key(x): 返回x的排名, 0-based
- find_by_order(x): 返回排名(0-based)所对应元素的迭代器
- lower_bound(x) / upper_bound(x): 返回第一个 \geq 或者 $>$ x的元素的迭代器

- join(x): 将x树并入当前树, 前提是两棵树的类型一样, 并且二者值域不能重叠, x树会被删除
- split(x,b): 分裂成两部分, 小于等于x的属于当前树, 其余的属于b树
- empty(): 返回是否为空
- size(): 返回大小

(注意平衡树不支持多重值, 如果需要多重值, 可以再开一个unordered_map来记录值出现的次数, 将x<<32后加上出现的次数后插入. 注意此时应该为long long类型.)

7.9 rope

```
1 #include <ext/rope>
2 using namespace __gnu_cxx;
3
4 push_back(x); // 在末尾添加x
5 insert(pos, x); // 在pos插入x, 自然支持整个char数组的一次
  ↳ 插入
6 erase(pos, x); // 从pos开始删除x个
7 copy(pos, len, x); // 从pos开始到pos + len为止的部分, 赋
  ↳ 值给x
8 replace(pos, x); // 从pos开始换成x
9 substr(pos, x); // 提取pos开始x个
10 at(x) / [x]; // 访问第x个元素
```

7.10 编译选项

- -O2 -g -std=c++11: 狗都知道
- -Wall -Wextra -Wconversion: 更多警告
- -fsanitize=(address/undefined): 检查有符号整数溢出(算ub)/数组越界
注意无符号类型溢出不算ub

8. 注意事项

8.1 常见下毒手法

- 高精度高低位搞反了吗
- 线性筛抄对了吗
- sort比较函数是不是比了个寂寞
- 该取模的地方都取模了吗
- 边界情况(+1-1之类的)有没有想清楚
- 特判是否有必要, 确定写对了吗

8.2 场外相关

- 安顿好之后查一下附近的咖啡店, 打印店, 便利店之类的位置, 以备不时之需
- 热身赛记得检查一下编译注意事项中的代码能否过编译, 还有熟悉比赛场地, 清楚洗手间在哪儿, 测试打印机(如果可以)
- 比赛前至少要翻一遍板子, 尤其要看原理与例题
- 比赛前一两天不要摸鱼, 要早睡, 有条件最好洗个澡; 比赛当天不要起太晚, 维持好的状态

- 赛前记得买咖啡,最好直接安排三人份,记得要咖啡因比较足的;如果主办方允许,就带些巧克力之类的高热量零食
- 入场之后记得检查机器,尤其要逐个检查键盘按键有没有坏的;如果可以的话,调一下gedit设置
- 开赛之前调整好心态,比赛而已,不必心急.

8.3 做题策略与心态调节

- 拿到题后立刻按照商量好的顺序读题, 前半小时最好跳过题意太复杂的题(除非被过穿了)
- 签到题写完不要激动, 稍微检查一下最可能的下毒点再交, 避免无谓的罚时
一两行的那种傻逼题就算了
- 读完题及时输出题意, 一方面避免重复读题, 一方面也可以让队友有一个初步印象, 方便之后决定开题顺序
- 如果不能确定题意就不要贸然输出甚至上机, 尤其是签到题, 因为样例一般都很弱
- 一个题如果卡了很久又有其他题可以写, 那不妨先放掉写更容易的题, 不要在一棵树上吊死
不要被一两道题搞得心态爆炸, 一方面急也没有意义, 一方面你很可能真的离AC就差一步

- 榜是不会骗人的, 一个题如果被不少人过了就说明这个题很可能并没有那么难;如果不是有十足的把握就不要轻易开没什么人交的题;另外不要忘记最后一小时会封榜
- 想不出题/找不出毒自然容易犯困, 一定不要放任自己昏昏欲睡, 最好去洗手间冷静一下, 没有条件就站起来踱步
- 思考的时候不要挂机, 一定要在草稿纸上画一画, 最好说出声来最不容易断掉思路
- 出完算法一定要check一下样例和一些trivial的情况, 不然容易写了半天发现写了个假算法
- 上机前有时间就提前给需要思考怎么写的地方打草稿, 不要浪费机时
- 查毒时如果最难的地方反复check也没有问题, 就从头到脚仔仔细细查一遍, 不要放过任何细节, 即使是并查集和sort这种东西也不能想当然
- 后半场如果时间不充裕就不要冒险开难题, 除非真的无事可做
如果是没写过的东西也不要轻举妄动, 在有其他好写的题的时候就等一会再说
- 大多数时候都要听队长安排, 虽然不一定最正确但可以保持组织性
- 任何时候都不要着急, 着急不能解决问题, 不要当喆国王
- 输了游戏, 还有人生;赢了游戏, 还有人生.