

Standard Code Library

ONGLU

North Eastern University

August 2021

Contents

初始化	3
数据结构	3
轻重链剖分	3
二维树状数组	3
平衡树	4
可持久化数据结构	5
可持久化 Trie	5
主席树（静态第 k 小）	6
数学	7
多项式	7
结论	7
拉格朗日插值法	7
FFT 快速傅里叶变换	8
NTT 快速数论变换	8
FWT 快速沃尔什变换	9
群论	10
结论	10
图论	11
树论	11
树的直径	11
求 LCA	12
树上启发式合并	13
图论	13
第 k 短路	13
二分图匹配	14
结论	14
匈牙利算法	14
KM 算法二分图最大权匹配	15
网络流	15
Dinic 算法	15
EK 算法费用流	16
Dinic 算法费用流	16
连通性算法	17
Tarjan 强连通分量	17
点双连通	18
边双连通	19
2-SAT	19
计算几何	20
字符串	20
字符串哈希	20
Trie	21
KMP 算法	22
manacher 算法	22
AC 自动机	23
杂项	24
int128	24

`_builtin_` 内建函数

1. `__builtin_popcount(unsigned int n)` 该函数是判断 `n` 的二进制中有多少个 1
 2. `__builtin_parity(unsigned int n)` 该函数是判断 `n` 的二进制中 1 的个数的奇偶性
 3. `__builtin_ffs(unsigned int n)` 该函数判断 `n` 的二进制末尾最后一个 1 的位置，从一开始
 4. `__builtin_ctz(unsigned int n)` 该函数判断 `n` 的二进制末尾后面 0 的个数，当 `n` 为 0 时，和 `n` 的类型有关
 5. `__builtin_clz (unsigned int x)` 返回前导的 0 的个数 24
- tips: 24

初始化

数据结构

轻重链剖分

```
1 void dfs1(int x, int pre) {
2     siz[x] = 1; mson[x] = 0;
3     dth[x] = dth[pre] + 1;
4     fa[x] = pre;
5     for(auto y : son[x]) if(y != pre) {
6         dfs1(y, x);
7         siz[x] += siz[y];
8         if(!mson[x] || siz[y] > siz[mson[x]])
9             mson[x] = y;
10    }
11 }
12 void dfs2(int x, int pre, int ntp) {
13     id[x] = ++idcnt;
14     ltp[x] = ntp;
15     if(mson[x]) dfs2(mson[x], x, ntp);
16     for(auto y : son[x]) {
17         if(y == mson[x] || y == pre) continue;
18         dfs2(y, x, y);
19     }
20 }
21 void link_modify(int x, int y, int z) {
22     z %= mod;
23     while(ltp[x] != ltp[y]) {
24         dth[ltp[x]] < dth[ltp[y]] && (x ^= y ^= x ^= y);
25         modify(1, n, id[ltp[x]], id[x], 1, z);
26         x = fa[ltp[x]];
27     }
28     dth[x] < dth[y] && (x ^= y ^= x ^= y);
29     modify(1, n, id[y], id[x], 1, z);
30 }
31 }
32 int link_query(int x, int y) {
33     int ans = 0;
34     while(ltp[x] != ltp[y]) {
35         dth[ltp[x]] < dth[ltp[y]] && (x ^= y ^= x ^= y);
36         ans = (1ll * ans + query(1, n, id[ltp[x]], id[x], 1)) % mod;
37         x = fa[ltp[x]];
38     }
39     dth[x] < dth[y] && (x ^= y ^= x ^= y);
40     ans = (1ll * ans + query(1, n, id[y], id[x], 1)) % mod;
41     return ans;
42 }
```

二维树状数组

- 矩阵修改，矩阵查询

查询前缀和公式：

令 $d[i][j]$ 为差分数组，定义 $d[i][j] = a[i][j] - (a[i-1][j] - a[i][j-1] - a[i-1][j])$

$$\sum_{i=1}^x \sum_{j=1}^y a[i][j] = (x+1) * (y+1) * d[i][j] - (y+1) * i * d[i][j] + d[i][j] * i * j$$

```
1 void modify(int x, int y, int v) {
2     for(int rx = x; rx <= n; rx += rx & -rx) {
3         for(int ry = y; ry <= m; ry += ry & -ry) {
4             tree[rx][ry][0] += v;
5             tree[rx][ry][1] += v * x;
6             tree[rx][ry][2] += v * y;
7             tree[rx][ry][3] += v * x * y;
8         }
9     }
10 }
```

```

11 void range_modify(int x, int y, int xx, int yy, int v) {
12     modify(xx + 1, yy + 1, v);
13     modify(x, yy + 1, -v);
14     modify(xx + 1, y, -v);
15     modify(x, y, v);
16 }
17 int query(int x, int y) {
18     int ans = 0;
19     for(int rx = x; rx; rx -= rx & -rx) {
20         for(int ry = y; ry; ry -= ry & -ry) {
21             ans += (x + 1) * (y + 1) * tree[rx][ry][0]
22                 - tree[rx][ry][1] * (y + 1) - tree[rx][ry][2] * (x + 1)
23                 + tree[rx][ry][3];
24         }
25     }
26     return ans;
27 }
28 int range_query(int x, int y, int xx, int yy) {
29     return query(xx, yy) + query(x - 1, y - 1)
30         - query(x - 1, yy) - query(xx, y - 1);
31 }

```

平衡树

- luogu P3369 【模板】普通平衡树

```

1  #define val(x) tree[x].val
2  #define cnt(x) tree[x].cnt
3  #define siz(x) tree[x].siz
4  #define fa(x) tree[x].fa
5  #define son(x, k) tree[x].ch[k]
6  struct Tree {
7      struct node {
8          int val, cnt, siz, fa, ch[2];
9      } tree[N];
10     int root, tot;
11     int chk(int x) {
12         return son(fa(x), 1) == x;
13     }
14     void update(int x) {
15         siz(x) = siz(son(x, 0)) + siz(son(x, 1)) + cnt(x);
16     }
17     void rotate(int x) {
18         int y = fa(x), z = fa(y), k = chk(x), w = son(x, k ^ 1);
19         son(y, k) = w; fa(w) = y;
20         son(z, chk(y)) = x; fa(x) = z;
21         son(x, k ^ 1) = y; fa(y) = x;
22         update(y); update(x);
23     }
24     void splay(int x, int goal = 0) {
25         while(fa(x) != goal) {
26             int y = fa(x), z = fa(y);
27             if(z != goal) {
28                 //双旋
29                 if(chk(y) == chk(x)) rotate(y);
30                 else rotate(x);
31             }
32             rotate(x);
33         }
34         if(!goal) root = x;
35     }
36     int New(int x, int pre) {
37         tot++;
38         if(pre) son(pre, x > val(pre)) = tot;
39         val(tot) = x; fa(tot) = pre;
40         siz(tot) = cnt(tot) = 1;
41         son(tot, 0) = son(tot, 1) = 0;
42         return tot;
43     }

```

```

44 void Insert(int x) {
45     int cur = root, p = 0;
46     while(cur && val(cur) != x) {
47         p = cur;
48         cur = son(cur, x > val(cur));
49     }
50     if(cur) cnt(cur)++;
51     else cur = New(x, p);
52     splay(cur);
53 }
54 void Find(int x) {
55     if(!root) return ;
56     int cur = root;
57     while(val(cur) != x && son(cur, x > val(cur)))
58         cur = son(cur, x > val(cur));
59     splay(cur);
60 }
61 int Pre(int x) {
62     Find(x);
63     if(val(root) < x) return root;
64     int cur = son(root, 0);
65     while(son(cur, 1))
66         cur = son(cur, 1);
67     return cur;
68 }
69 int Succ(int x) {
70     Find(x);
71     if(val(root) > x) return root;
72     int cur = son(root, 1);
73     while(son(cur, 0))
74         cur = son(cur, 0);
75     return cur;
76 }
77 void Del(int x) {
78     int lst = Pre(x), nxt = Succ(x);
79     splay(lst); splay(nxt, lst);
80     int cur = son(nxt, 0);
81     if(cnt(cur) > 1) cnt(cur)--, splay(cur);
82     else son(nxt, 0) = 0, splay(nxt);
83 }
84 int Kth(int k) {
85     int cur = root;
86     while(1) {
87         if(son(cur, 0) && siz(son(cur, 0)) >= k) cur = son(cur, 0);
88         else if(siz(son(cur, 0)) + cnt(cur) >= k) return cur;
89         else k -= siz(son(cur, 0)) + cnt(cur), cur = son(cur, 1);
90     }
91 }
92 } T;

```

可持久化数据结构

可持久化 Trie

```

1 namespace Trie {
2     struct node {
3         int ch[2], ed, siz;
4     } tree[N * 40];
5     int tot = 0;
6     int _new() {
7         tot++;
8         tree[tot].ch[0] = 0;
9         tree[tot].ch[1] = 0;
10        tree[tot].ed = tree[tot].siz = 0;
11        return tot;
12    }
13    void init() {
14        tot = 0;
15        rt[0] = _new();
16    }
17    int Insert(int x, int t, int i = 15) {

```

```

18     int u = _new(), f = (x >> i) & 1;
19     tree[u] = tree[t];
20     if(i == -1) {
21         ed(u)++;
22         siz(u)++;
23         return u;
24     }
25     son(u, f) = Insert(x, son(t, f), i - 1);
26     siz(u) = siz(son(u, 0)) + siz(son(u, 1));
27     return u;
28 }
29 void print(int u, int now) {
30     if(u == 0) return;
31     for(int i = 1; i <= ed(u); i++) printf("%d ", now);
32     if(son(u, 0)) print(son(u, 0), now * 2);
33     if(son(u, 1)) print(son(u, 1), now * 2 + 1);
34 }
35 int query(int u1, int u2, int x, int i = 15, int now = 0) {
36     if(i == -1) return now;
37     int f = (x >> i) & 1;
38     if(siz(son(u1, f ^ 1)) - siz(son(u2, f ^ 1)) > 0)
39         return query(son(u1, f ^ 1), son(u2, f ^ 1), x, i - 1, now * 2 + (f ^ 1));
40     else return query(son(u1, f), son(u2, f), x, i - 1, now * 2 + (f));
41 }
42 }

```

主席树（静态第 k 小）

建立权值树，那么 $[l, r]$ 的区间权值树就是第 r 个版本减去第 $l - 1$ 个版本的树。

```

1  #include <iostream>
2  #include <cstdio>
3  #include <algorithm>
4  #include <cmath>
5  #include <assert.h>
6  #define Mid ((l + r) / 2)
7  #define lson (rt << 1)
8  #define rson (rt << 1 | 1)
9  using namespace std;
10 int read() {
11     char c; int num, f = 1;
12     while(c = getchar(), !isdigit(c)) if(c == '-') f = -1; num = c - '0';
13     while(c = getchar(), isdigit(c)) num = num * 10 + c - '0';
14     return f * num;
15 }
16 const int N = 1e7 + 1009;
17 const int M = 2e5 + 1009;
18 struct node {
19     int ls, rs, v;
20 } tree[N];
21 int tb;
22 int n, m, tot, a[M], b[M], rt[M];
23 int _new(int ls, int rs, int v) {
24     tree[++tot].ls = ls;
25     tree[tot].rs = rs;
26     tree[tot].v = v;
27     return tot;
28 }
29 void update(int rt) {
30     tree[rt].v = tree[tree[rt].ls].v + tree[tree[rt].rs].v;
31 }
32 int build(int l, int r) {
33     if(l == r) return _new(0, 0, 0);
34     int x = _new(build(l, Mid), build(Mid + 1, r), 0);
35     update(x);
36     return x;
37 }
38 int add(int l, int r, int p, int rt, int v) {
39     int x = ++tot;
40     tree[x] = tree[rt];
41     if(l == r) {

```

```

42     tree[x].v += v;
43     return x;
44 }
45 if(p <= Mid) tree[x].ls = add(l, Mid, p, tree[x].ls, v);
46 else tree[x].rs = add(Mid + 1, r, p, tree[x].rs, v);
47 update(x);
48 return x;
49 }
50 int query(int l, int r, int rt1, int rt2, int k) {
51     if(l == r) return l;
52     if(k <= tree[tree[rt1].ls].v - tree[tree[rt2].ls].v) return query(l, Mid, tree[rt1].ls, tree[rt2].ls, k);
53     else return query(Mid + 1, r, tree[rt1].rs, tree[rt2].rs, k - (tree[tree[rt1].ls].v - tree[tree[rt2].ls].v));
54 }
55 void Debug(int l, int r, int rt) {
56     printf("%d %d %d\n", l, r, tree[rt].v);
57     if(l == r) return ;
58     Debug(l, Mid, tree[rt].ls);
59     Debug(Mid + 1, r, tree[rt].rs);
60 }
61 signed main()
62 {
63     n = read(); m = read();
64     for(int i = 1; i <= n; i++) a[i] = b[i] = read();
65     sort(b + 1, b + 1 + n);
66     tb = unique(b + 1, b + 1 + n) - b - 1;
67     rt[0] = build(1, tb);
68     for(int i = 1; i <= n; i++) {
69         rt[i] = add(1, tb, lower_bound(b + 1, b + 1 + tb, a[i]) - b, rt[i - 1], 1);
70     }
71     for(int i = 1; i <= m; i++) {
72         int l, r, k;
73         l = read(); r = read(); k = read();
74         assert(r - l + 1 >= k);
75         printf("%d\n", b[query(1, tb, rt[r], rt[l - 1], k)]);
76     }
77     return 0;
78 }

```

数学

多项式

结论

1. 自然数幂之和 $s(n) = \sum_{i=0}^n i^k$ 是关于 n 的 $k + 1$ 次多项式

拉格朗日插值法

令拉格朗日函数

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

注意到这个函数有一些性质：

1. 次数为 n
 2. 在 $x = x_i$ 位置值为 1, $x = x_j (j \neq i)$ 位置值为 0
- 于是可以凑出唯一的多项式表达式为：

$$f(x) = \sum_{i=0}^n y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

如果要取模的话得求逆元，逆元先求好分母再一起求即可。

```

1 int interpolation(int *x, int *y, int n) {
2     int f = 0;
3     for(int i = 1; i <= n; i++) {

```



```

4         int s1 = 1, s2 = 1;
5         for(int j = 1; j <= n; j++) {
6             if(i != j) {
7                 s1 = 1ll * s1 * (k - x[j] + mod) % mod;
8                 s2 = 1ll * s2 * (x[i] - x[j] + mod) % mod;
9             }
10        }
11        f = (f + 1ll * y[i] * s1 % mod * inv(s2) % mod) % mod;
12    }
13    return f;
14 }

```

FFT 快速傅里叶变换

FFT 的想法是把第 k 号位置变成 $f(\omega_n^k)$, 注意到 $\omega_n^k = -\omega_n^{k+n/2}$, 于是可以进行变换。

几条公式:

$$\omega_n^n = 1$$

$$\omega_n^k = \omega_{2n}^{2k}$$

$$\omega_{2n}^{k+n} = -\omega_{2n}^k$$

蝴蝶变换: 相邻的位置为二进制的 reverse

DFT 变换公式 ($DFT(f)$ 为矩阵):

令

$$G(x) = a_0 + a_2x + a_4x^2 + \dots$$

$$H(x) = a_1 + a_3x + a_5x^3 + \dots$$

则有

$$f(x) = G(x^2) + x \times H(x^2)$$

$$DFT(f(\omega_n^k)) = DFT(G(\omega_{n/2}^k) + \omega_n^k \times DFT(H(\omega_{n/2}^k)))$$

$$DFT(f(\omega_n^{k+n/2})) = DFT(G(\omega_{n/2}^k) - \omega_n^k \times DFT(H(\omega_{n/2}^k)))$$

$DFT(G(\omega_{n/2}^k), DFT(H(\omega_{n/2}^k)))$ 可递归计算

NTT 快速数论变换

NTT 使用原根代替复数进行运算。

原根 g 的重要性质: $g^t \equiv k \pmod n, t \in [0, n-2], k$ 遍取 $1 \sim n-1$

原根存在的充要条件是: 模数 $n = 2, 4, p^\alpha, 2p^\alpha (p \text{ 为奇质数})$ 。

对于一个质数 $p = qn + 1 (n = 2^m)$, 原根满足性质 $g^{qn} \equiv 1 \pmod p$ 。

它满足和复数近似的性质, 我们把 q 看成复数中的 2π , 就可以套用 FFT 实现 NTT 了。

$$g_n^n \equiv 1, g_n^n \equiv -1$$

通常取

$$p = 1004535809 = 7 \times 479 \times 2^{21} + 1, g = 3$$

$$p = 998244353 = 7 \times 17 \times 2^{23} + 1, g = 3$$

```

1  const int P = 998244353, G = 3, Gi = 332748118;
2  struct Complex {double x, y;};
3  Complex operator+(const Complex &a, const Complex &b) {return (Complex) {a.x + b.x, a.y + b.y};}
4  Complex operator-(const Complex &a, const Complex &b) {return (Complex) {a.x - b.x, a.y - b.y};}
5  Complex operator*(const Complex &a, const Complex &b) {return (Complex) {a.x * b.x - a.y * b.y, a.x * b.y + a.y *
    ↪ b.x};}
6  namespace Polynomial {
7      const double Pi = acos(-1.0);

```

```

8   int rev[N];
9   template <typename T>
10  void change(T *y, int n) {
11      for(int i = 0; i < n; i++) {
12          rev[i] = (rev[i >> 1] >> 1) | ((i & 1) ? (n >> 1) : 0);
13      }
14      for(int i = 0; i < n; i++)
15          if(i < rev[i])
16              swap(y[i], y[rev[i]]);
17  }
18  void FFT(Complex *A, int n, int type) {
19      //type = 1 DFT
20      //type = -1 IDFT
21      //确保 n 是 2 的幂次
22      change(A, n);
23      for(int m = 1; m < n; m <= 1) {
24          Complex Wn = (Complex) {cos(Pi / m), type * sin(Pi / m)};
25          for(int i = 0; i < n; i += 2 * m) {
26              Complex w = (Complex) {1.0, 0};
27              for(int j = 0; j < m; j++, w = w * Wn) {
28                  Complex x = A[i + j], y = w * A[i + j + m];
29                  A[i + j] = x + y;
30                  A[i + j + m] = x - y;
31              }
32          }
33      }
34      if(type == -1) {
35          for(int i = 0; i < n; i++)
36              A[i].x = A[i].x / n;
37      }
38  }
39  void NTT(int *A, int n, int type) {
40      //type = 1 DFT
41      //type = -1 IDFT
42      change(A, n);
43      for(int m = 1; m < n; m <= 1) {
44          int Wn = qpow(type == 1 ? G : Gi, (P - 1) / (m <= 1));
45          for(int i = 0; i < n; i += 2 * m) {
46              int w = 1;
47              for(int j = 0; j < m; j++, w = 1ll * w * Wn % P) {
48                  int x = A[i + j], y = 1ll * w * A[i + j + m] % P;
49                  A[i + j] = (x + y) % P;
50                  A[i + j + m] = (x - y + P) % P;
51              }
52          }
53      }
54      if(type == -1) {
55          int inv = qpow(n, P - 2);
56          for(int i = 0; i < n; i++)
57              A[i] = 1ll * A[i] * inv % P;
58      }
59  }
60
61  }
62  //以下代码加在主函数内
63  limit = 1;
64  while(limit <= n + m) limit <= 1;
65  Polynomial :: FFT(A, limit, 1);
66  Polynomial :: FFT(B, limit, 1);
67  for(int i = 0; i < limit; i++) A[i] = A[i] * B[i];
68  Polynomial :: FFT(A, limit, -1);

```

FWT 快速沃尔什变换

FWT 用于计算下列多项式

$$C[k] = \sum_{i \oplus j = k} A[i] \times B[j]$$

先通过 FWT 将 A, B 变为 $FWT(A), FWT(B)$, 这样有 $FWT(C) = FWT(A) \times FWT(B)$ 。

当然位运算符不同的时候对应的变换形式也需要改变。

$a \in S, b \in S$ 可以表示为 $a|b \in S$

与卷积

当 $\oplus = \text{and}$ 的时候

$$FWT(A) = (FWT(A_0) + FWT(A_1), FWT(A_1))$$

$$FWT(A) = A(\text{长度为 } 1)$$

$$IFWT(A) = (IFWT(A_0) - IFWT(A_1), IFWT(A_1))$$

或卷积

当 $\oplus = \text{or}$ 的时候

$$FWT(A) = (FWT(A_0), FWT(A_0) + FWT(A_1))$$

$$FWT(A) = A(\text{长度为 } 1)$$

$$IFWT(A) = (IFWT(A_0), IFWT(A_1) - IFWT(A_0))$$

异或卷积

当 $\oplus = \text{xor}$ 的时候

$$FWT(A) = (FWT(A_0) + FWT(A_1), FWT(A_0) - FWT(A_1))$$

$$FWT(A) = A(\text{长度为 } 1)$$

$$IFWT(A) = (\frac{IFWT(A_0) + IFWT(A_1)}{2}, \frac{IFWT(A_0) - IFWT(A_1)}{2})$$

```

1 namespace Polynomial {
2     void FWT_or(int *A, int n, int type) {
3         for(int m = 1; m < n; m <= 1) {
4             for(int i = 0; i < n; i += 2 * m) {
5                 for(int j = 0; j < m; j++) {
6                     A[i + j + m] = (1ll * A[i + j + m] + A[i + j] * type + mod) % mod;
7                 }
8             }
9         }
10    }
11    void FWT_and(int *A, int n, int type) {
12        for(int m = 1; m < n; m <= 1) {
13            for(int i = 0; i < n; i += 2 * m) {
14                for(int j = 0; j < m; j++) {
15                    A[i + j] = (1ll * A[i + j + m] * type + A[i + j] + mod) % mod;
16                }
17            }
18        }
19    }
20    void FWT_xor(int *A, int n, int type) {
21        int inv_2 = Pow(2, mod - 2);
22        for(int m = 1; m < n; m <= 1) {
23            for(int i = 0; i < n; i += 2 * m) {
24                for(int j = 0; j < m; j++) {
25                    int x = A[i + j], y = A[i + j + m];
26                    A[i + j] = (1ll * x + y) * (type == 1 ? 1 : inv_2) % mod;
27                    A[i + j + m] = (1ll * x - y + mod) * (type == 1 ? 1 : inv_2) % mod;
28                }
29            }
30        }
31    }
32 }

```

群论

结论

1. 子群检验法: 群 G 是群 H 的子群的充分必要条件: 对于所有元素 h, g , 只需检查 $g^{-1} \cdot h \in H$ 。

图论

树论

树的直径

模板: POJ - 1985

- 两遍 DFS

```
1 void dfs(int x, int fa) {
2     for(int i = 0; i < E[x].size(); i++) {
3         int y = E[x][i].ver;
4         int w = E[x][i].val;
5         if(y == fa) continue;
6         d[y] = d[x] + w;
7         if(d[y] > d[c]) c = y;
8         dfs(y, x);
9     }
10 }
11 signed main()
12 {
13     n = read();
14     for(int i = 1; i < n; i++) {
15         int x = read(), y = read(), w = read();
16         E[x].push_back((Edge) {y, w});
17         E[y].push_back((Edge) {x, w});
18     }
19     dfs(1, 0);
20     d[c] = 0;
21     dfs(c, 0);
22     printf("%d\n", d[c]);
23     return 0;
24 }
```

- 树形 DP

```
1 void dfs(int x, int fa) {
2     d1[x] = d2[x] = 0;
3     for(int i = 0; i < E[x].size(); i++) {
4         int y = E[x][i].ver;
5         int w = E[x][i].val;
6         if(y == fa) continue;
7         dfs(y, x);
8         int t = d1[y] + w;
9         if(t > d1[x]) {
10             d2[x] = d1[x];
11             d1[x] = t;
12         } else if(t > d2[x]) {
13             d2[x] = t;
14         }
15     }
16     d = max(d, d1[x] + d2[x]);
17 }
18 signed main()
19 {
20     n = read();
21     for(int i = 1; i < n; i++) {
22         int x = read(), y = read(), w = read();
23         E[x].push_back((Edge) {y, w});
24         E[y].push_back((Edge) {x, w});
25     }
26     dfs(1, 0);
27     printf("%d\n", d);
28     return 0;
29 }
```

求 LCA

● 树链剖分

```
1 namespace Tree {
2     int siz[N], mson[N], ltp[N], fa[N], dth[N];
3     vector<int> son[N];
4     void dfs1(int x, int pre) {
5         siz[x] = 1;
6         mson[x] = 0;
7         fa[x] = pre;
8         dth[x] = dth[pre] + 1;
9         for(auto y : son[x]) if(y != pre) {
10             dfs1(y, x);
11             if(mson[x] == 0 || siz[y] > siz[mson[x]]) mson[x] = y;
12         }
13     }
14     void dfs2(int x, int pre, int tp) {
15         ltp[x] = tp;
16         if(mson[x]) dfs2(mson[x], x, tp);
17         for(auto y : son[x]) if(y != pre && y != mson[x]) {
18             dfs2(y, x, y);
19         }
20     }
21     void init() {
22         dfs1(1, 0);
23         dfs2(1, 0, 1);
24     }
25     int LCA(int x, int y) {
26         while(ltp[x] != ltp[y]) {
27             if(dth[ltp[x]] > dth[ltp[y]]) x = fa[ltp[x]];
28             else y = fa[ltp[y]];
29         }
30         return dth[y] > dth[x] ? x : y;
31     }
32 }
```

● 倍增

```
1 namespace Tree {
2     vector<int> son[N];
3     int root, fa[N][31], dth[N];
4     void dfs(int x, int pre) {
5         fa[x][0] = pre;
6         dth[x] = dth[pre] + 1;
7         for(int i = 1; i <= 30; i++)
8             fa[x][i] = fa[fa[x][i-1]][i-1];
9         for(auto y : son[x]) if(y != pre)
10             dfs(y, x);
11     }
12     void init() {
13         dfs(root, 0);
14     }
15     int LCA(int x, int y) {
16         if(dth[x] > dth[y]) swap(x, y);
17         for(int i = 30; ~i; i--)
18             if(dth[fa[y][i]] >= dth[x])
19                 y = fa[y][i];
20         if(x == y) return x;
21         for(int i = 30; ~i; i--)
22             if(fa[y][i] != fa[x][i]) {
23                 x = fa[x][i];
24                 y = fa[y][i];
25             }
26         return fa[x][0];
27     }
28 }
```

树上启发式合并

长春站的痛.jpg

- 先递归计算轻儿子的答案
- 计算重儿子的答案，并且保留重儿子的状态数组
- 把其他所有轻儿子的答案加到状态数组中，更新当前点的答案

```
1 void dfs1(int x, int pre) {
2     siz[x] = 1;
3     mson[x] = 0;
4     for(auto y : son[x]) if(y != pre) {
5         dfs1(y, x);
6         siz[x] += siz[y];
7         if(!mson[x] || siz[y] > siz[mson[x]]) mson[x] = y;
8     }
9 }
10 void add(int x, int pre, int v) {
11     cnt[col[x]] += v;
12     if(cnt[col[x]] > Mx) Mx = cnt[col[x]], sum = col[x];
13     else if(cnt[col[x]] == Mx) sum += col[x];
14     for(auto y : son[x]) {
15         if(y == pre || y == Son) continue;
16         add(y, x, v);
17     }
18 }
19 void dfs2(int x, int pre, int keep) {
20     for(auto y : son[x]) {
21         if(y == pre || y == mson[x]) continue;
22         dfs2(y, x, 0);
23     }
24     if(mson[x]) dfs2(mson[x], x, 1), Son = mson[x];
25     add(x, pre, 1); Son = 0;
26     ans[x] = sum;
27     if(!keep) add(x, pre, -1), sum = 0, Mx = 0;
28 }
29 }
```

图论

第 k 短路

模板: HDU-6351

估值函数: $h(x) = f(x) + g(x)$, 其中 $f(x)$ 为从起点到现在的距离, $g(x)$ 为起点到当前点的最短路。

```
1 bool operator<(const node &a, const node &b) {
2     return a.f + a.g > b.f + b.g;
3 }
4 priority_queue<node> q;
5 signed main()
6 {
7     n = read(); m = read();
8     for(int i = 1; i <= m; i++) {
9         int x, y, w;
10        x = read(); y = read(); w = read();
11        E[x].push_back((Edge) {y, w});
12        re[y].push_back((Edge) {x, w});
13    }
14    s = read(); t = read(); k = read();
15    memset(dis, 0x3f, sizeof(dis)); dis[t] = 0;
16    q.push((node) {t, 0, 0});
17    while(q.size()) {
18        int x = q.top().x, d = q.top().f;
19        q.pop();
20        if(dis[x] < d) continue;
21        for(int i = 0; i < re[x].size(); i++) {
22            int y = re[x][i].y, w = re[x][i].w;
23            if(dis[y] > dis[x] + w) {
24                dis[y] = dis[x] + w;
25                q.push((node) {y, dis[y], 0});
26            }
27        }
28    }
29 }
```

```

26     }
27 }
28 }
29 for(int i = 1; i <= n; i++) cnt[i] = k;
30 cnt[s]++;
31 q.push((node) {s, 0, dis[s]});
32 while(q.size()) {
33     int x = q.top().x, f = q.top().f, g = q.top().g;
34     q.pop();
35     if(cnt[x] == 0) continue;
36     cnt[x]--;
37     if(x == t && cnt[x] == 0) {
38         printf("%lld\n", f);
39         return 0;
40     }
41     for(int i = 0; i < E[x].size(); i++) {
42         int y = E[x][i].y, w = E[x][i].w;
43         q.push((node) {y, f + w, dis[y]});
44     }
45 }
46 printf("-1\n");
47 return 0;
48 }

```

二分图匹配

结论

最大匹配数：最大匹配的匹配边的数目

最小点/边覆盖数：选取最少的点/边，使任意一条边至少有一个点被选择 / 点至少连有一条边。

最大独立数：选取最多的点，使任意所选两点均不相连

最小路径覆盖数：对于一个 DAG（有向无环图），选取最少条路径，使得每个顶点属于且仅属于一条路径。路径长可以为 0（即单个点）。

1. 最大匹配数 = 最小点覆盖数（这是 Konig 定理）
2. 最大匹配数 = 最大独立数
3. 最小路径覆盖数 = 顶点数 - 最大匹配数
4. 原图的最大团 = 补图的最大独立集原图的最大独立集 = 补图的最大团
5. 最小边覆盖 = 顶点数 - 最大匹配数

在一般图中：

最小不相交路径覆盖：每个点拆点为 $2x - 1, 2x$ ，那么一条边 (x, y) ，则连边 $(2x - 1, 2y)$ ，答案是 $n - maxmatch$

最小可相交路径覆盖：跑一遍传递闭包，按传递闭包上的边建边之后转化为最小不相交路径覆盖。

二分图最大匹配的必须边：

在完备匹配中：

匹配边从左到右方向，非匹配边从右到左方向，则一条边为必须边当且仅当边在最大匹配中，并且边所连的两个点不在同一个强连通分量中。

在非完备匹配中：

匈牙利算法

```

1 int dfs(int x) {
2     for(int i = head[x]; i; i = nxt[i]) {
3         int y = ver[i];
4         if(vis[y]) continue;
5         vis[y] = 1;
6         if(!match[y] || dfs(match[y])) {
7             match[y] = x;
8             return true;
9         }
10    }
11    return false;

```

```

12 }
13 for(int i = 1; i <= n; i++) {
14     memset(vis, 0, sizeof(vis));
15     if(dfs(i)) ans++;
16 }

```

KM 算法二分图最大权匹配

KM 算法只支持二分图最大权完美匹配，若图不一定存在完美匹配，注意补 0 边和补点。

KM 算法引入了顶标的概念，用 $la[x]$ 和 $lb[x]$ 分别保存两侧点的顶标，顶标必须满足大于所有边。

每次对每个点进行循环匹配，匹配中统计一个 $delta$ 表示最小的权值使得一条边可以加入。

然后修改顶标再继续匹配。

```

1 int la[N], lb[N], va[N], vb[N], delta, match[N], g[N][N], n;
2 int dfs(int x) {
3     va[x] = 1;
4     for(int y = 1; y <= n; y++) {
5         if(!vb[y]) {
6             if(la[x] + lb[y] - g[x][y] == 0) {
7                 vb[y] = 1;
8                 if(!match[y] || dfs(match[y])) {
9                     match[y] = x;
10                    return true;
11                }
12            } else delta = min(delta, la[x] + lb[y] - g[x][y]);
13        }
14    }
15    return false;
16 }
17 void work() {
18     for(int i = 1; i <= n; i++)
19         for(int j = 1; j <= n; j++)
20             g[i][j] = read();
21     memset(match, 0, sizeof(match));
22     for(int i = 1; i <= n; i++) {
23         la[i] = g[i][1];
24         lb[i] = 0;
25         for(int j = 2; j <= n; j++)
26             la[i] = max(la[i], g[i][j]);
27     }
28     for(int i = 1; i <= n; i++) {
29         while(true) {
30             memset(va, 0, sizeof(va));
31             memset(vb, 0, sizeof(vb));
32             delta = 0x3f3f3f3f;
33             if(dfs(i)) break;
34             for(int j = 1; j <= n; j++) {
35                 if(va[j]) la[j] -= delta;
36                 if(vb[j]) lb[j] += delta;
37             }
38         }
39     }
40     long long ans = 0;
41     for(int i = 1; i <= n; i++)
42         ans += g[match[i]][i];
43     printf("%lld\n", ans);
44 }

```

网络流

Dinic 算法

```

1 const int inf = 0x3f3f3f3f;
2 int bfs() {
3     memset(d, 0, sizeof(int) * (t + 10)); d[s] = 1;
4     while(q.size()) q.pop(); q.push(s);
5     while(q.size()) {
6         int x = q.front(); q.pop();
7         for(int i = head[x]; i; i = nxt[i]) {

```



```

8         if(d[ver[i]]) continue;
9         if(edge[i] <= 0) continue;
10        d[ver[i]] = d[x] + 1;
11        q.push(ver[i]);
12    }
13    }
14    return d[t];
15 }
16 int dinic(int x, int flow) {
17     if(x == t) return flow;
18     int k, res = flow;
19     for(int i = head[x]; i && res; i = nxt[i]) {
20         if(d[ver[i]] != d[x] + 1 || edge[i] <= 0) continue;
21         k = dinic(ver[i], min(res, edge[i]));
22         if(k == 0) d[ver[i]] = 0;
23         edge[i] -= k;
24         edge[i ^ 1] += k;
25         res -= k;
26     }
27     return flow - res;
28 }

```

EK 算法费用流

```

1 //反向边 cost 为负数, 容量为 0
2 int SPFA() {
3     queue<int> q; q.push(s);
4     memset(dis, 0x3f, sizeof(dis)); dis[s] = 0;
5     memset(vis, 0, sizeof(vis)); vis[s] = 1;
6     q.push(s); flow[s] = 0x3f3f3f3f;
7     while(q.size()) {
8         int x = q.front();
9         vis[x] = 0; q.pop();
10        for(int i = head[x]; i; i = nxt[i]) {
11            if(edge[i] <= 0) continue;
12            if(dis[ver[i]] > dis[x] + cost[i]) {
13                dis[ver[i]] = dis[x] + cost[i];
14                pre[ver[i]] = i;
15                flow[ver[i]] = min(flow[x], edge[i]);
16                if(!vis[ver[i]]) {
17                    q.push(ver[i]);
18                    vis[ver[i]] = 1;
19                }
20            }
21        }
22    }
23    return dis[t] != 0x3f3f3f3f;
24 }
25 void update() {
26     int x = t;
27     while(x != s) {
28         int i = pre[x];
29         edge[i] -= flow[t];
30         edge[i ^ 1] += flow[t];
31         x = ver[i ^ 1];
32     }
33     maxflow += flow[t];
34     minncost += dis[t] * flow[t];
35 }

```

Dinic 算法费用流

```

1 int SPFA() {
2     while(q.size()) q.pop(); q.push(s);
3     memset(d, 0x3f, sizeof(int) * (n + 10)); d[s] = 0;
4     memset(vis, 0, sizeof(int) * (n + 10)); vis[s] = 1;
5     while(q.size()) {
6         int x = q.front(); q.pop();
7         vis[x] = 0;
8         for(int i = head[x]; i; i = nxt[i]) {

```

```

9         if(edge[i] <= 0) continue;
10        if(d[ver[i]] > d[x] + cost[i]) {
11            d[ver[i]] = d[x] + cost[i];
12            if(!vis[ver[i]]) {
13                vis[ver[i]] = 1;
14                q.push(ver[i]);
15            }
16        }
17    }
18 }
19 return d[t] != 0x3f3f3f3f;
20 }
21 int dinic(int x, int flow) {
22     if(x == t) return flow;
23     vis[x] = 1;
24     int k, res = flow;
25     for(int i = head[x]; i && res; i = nxt[i]) {
26         if(vis[ver[i]]) continue;
27         if(d[ver[i]] != d[x] + cost[i] || edge[i] <= 0) continue;
28         k = dinic(ver[i], min(edge[i], res));
29         if(!k) d[ver[i]] = -1;
30         edge[i] -= k;
31         edge[i ^ 1] += k;
32         res -= k;
33         mincost += cost[i] * k;
34     }
35     vis[x] = 0;
36     return flow - res;
37 }

```

连通性算法

Tarjan 强连通分量

$dfn[x]$: dfs 序。

$low[x]$: 追溯值, 指 x 的子树内部, 通过一条非树边能到达的最小的 dfn 值。

如果 $dfn[x] == low[x]$, 当前栈中, x 以后的元素为一个强连通。

```

1 void tarjan(int x) {
2     low[x] = dfn[x] = ++dfncnt;
3     s[++t] = x; vis[x] = 1;
4     for(int i = head[x]; i; i = nxt[i]) {
5         if(!dfn[ver[i]]) {
6             tarjan(ver[i]);
7             low[x] = min(low[x], low[ver[i]]);
8         } else if(vis[ver[i]]) {
9             low[x] = min(low[x], dfn[ver[i]]);
10        }
11    }
12    if(dfn[x] == low[x]) {
13        int z = -1;
14        ++sc;
15        while(z != x) {
16            scc[s[t]] = sc;
17            siz[sc]++;
18            vis[s[t]] = 0;
19            z = s[t];
20            t--;
21        }
22    }
23 }
24 //从任意点开始跑, 但是注意如果图不连通, 需要每个点跑一次
25 for(int i = 1; i <= n; i++)
26     if(!dfn[i])
27         tarjan(i);

```

点双连通

Tarjan 割点判定

```
1  int cut[N];
2  namespace v_dcc {
3      int root, low[N], dfn[N], dfntot;
4      void tarjan(int x) {
5          low[x] = dfn[x] = ++dfntot;
6          int flag = 0;
7          for(int i = head[x]; i; i = nxt[i]) {
8              int y = ver[i];
9              if(!dfn[y]) {
10                 tarjan(y);
11                 low[x] = min(low[x], low[y]);
12                 if(low[y] >= dfn[x]) {
13                     flag++;
14                     if(x != root || flag > 1) cut[x] = 1;
15                 }
16             } else low[x] = min(low[x], dfn[y]);
17         }
18     }
19     void getcut() {
20         for(int i = 1; i <= n; i++)
21             if(!dfn[i])
22                 tarjan(root = i);
23     }
24 }
25 }
```

求点双连通分量

点双连通分量比较复杂，一个点可能存在于多个点双连通分量当中，一个点删除与搜索树中的儿子节点断开时，不能在栈中弹掉父亲点，但是父亲点属于儿子的 v-dcc。

```
1  int cut[N];
2  vector<int> dcc[N];
3  namespace v_dcc {
4      int s[N], t, root;
5      int es[N], et;
6      void tarjan(int x) {
7          dfn[x] = low[x] = ++dfntot;
8          s[++t] = x;
9          if(x == root && head[x] == 0) {
10             dcc[++dc].clear();
11             dcc[dc].push_back(x);
12             return;
13         }
14         int flag = 0;
15         for(int i = head[x]; i; i = nxt[i]) {
16             int y = ver[i];
17             if(!dfn[y]) {
18                 tarjan(y);
19                 low[x] = min(low[x], low[y]);
20                 if(low[y] >= dfn[x]) {
21                     flag++;
22                     if(x != root || flag > 1) cut[x] = true;
23                     dcc[++dc].clear();
24                     int z = -1;
25                     while(z != y) {
26                         z = s[t--];
27                         dcc[dc].push_back(z);
28                     }
29                     dcc[dc].push_back(x);
30                 }
31             } else low[x] = min(low[x], dfn[y]);
32         }
33     }
34     void get_cut() {
35         for(int i = 1; i <= n; i++)
36             if(!dfn[i])
```

```

37         tarjan(root = i);
38     }
39 }

```

边双连通

搜索树上的点 x ，若它的一个儿子 y ，满足严格大于号 $low[y] > dfn[x]$ ，那么这条边就是桥。

注意由于会有重边，不能仅仅考虑他的父亲编号，而应该记录入边编号。

```

1  namespace e_dcc {
2      int low[N], dfn[N], dfntot;
3      vector<int> E[N];
4      void tarjan(int x, int in_edge) {
5          low[x] = dfn[x] = ++dfntot;
6          for(int i = head[x]; i; i = nxt[i]) {
7              int y = ver[i];
8              if(!dfn[y]) {
9                  tarjan(y, i);
10                 low[x] = min(low[x], low[y]);
11                 if(low[y] > dfn[x])
12                     bridge[i] = bridge[i ^ 1] = true;
13             } else if(i != (in_edge ^ 1))
14                 //注意运算优先级
15                 low[x] = min(low[x], dfn[y]);
16         }
17     }
18     void getbridge() {
19         for(int i = 1; i <= n; i++)
20             if(!dfn[i])
21                 tarjan(i, 0);
22     }
23     void dfs(int x) {
24         dcc[x] = dc;
25         for(int i = head[x]; i; i = nxt[i]) {
26             if(!dcc[ver[i]] && !bridge[i]) {
27                 dfs(ver[i]);
28             }
29         }
30     }
31     void getdcc() {
32         for(int i = 1; i <= n; i++) {
33             if(!dcc[i]) {
34                 ++dc;
35                 dfs(i);
36             }
37         }
38     }
39     void getgraphic() {
40         for(int x = 1; x <= n; x++) {
41             for(int i = head[x]; i; i = nxt[i]) {
42                 if(dcc[ver[i]] != dcc[x]) {
43                     E[dcc[x]].push_back(dcc[ver[i]]);
44                     E[dcc[ver[i]]].push_back(dcc[x]);
45                 }
46             }
47         }
48     }
49 }

```

2-SAT

2-SAT 用于解决每个变量的 01 取值问题，用于判断是否存在一种不冲突取值方法。

建边方法：假如选了 A 之后， B 的取值**确定**，那么就 A 的这个取值向 B 的这个取值建边，否则不要建边。

判定方法：如果， $\exists A$ ，使得 A 和 $\neg A$ 在同一个强连通分量里面，说明不存在一种合法取值，否则存在。

输出方案：自底向上确定每个变量的取值，由于 tarjan 求解强连通分量是自底向上，所以编号比较小的强连通是位于 DAG 底部的。

基于 tarjan 的方案输出就变得十分简单了，只要判断一个点和对立节点哪个 scc 的编号小就行了。

例如: $A \rightarrow B \rightarrow C$, 那么 C 的编号最小。

```
1  for(int i = 1; i <= m; i++) {
2      int x = read() + 1, y = read() + 1;
3      int w = read();
4      char c[10];
5      scanf("%s", c + 1);
6      if(c[1] == 'A') {
7          if(w) {
8              add(2 * x - 0, 2 * x - 1);
9              add(2 * y - 0, 2 * y - 1);
10         } else {
11             add(2 * x - 1, 2 * y - 0);
12             add(2 * y - 1, 2 * x - 0);
13         }
14     }
15     if(c[1] == 'O') {
16         if(w) {
17             add(2 * x - 0, 2 * y - 1);
18             add(2 * y - 0, 2 * x - 1);
19         } else {
20             add(2 * x - 1, 2 * x - 0);
21             add(2 * y - 1, 2 * y - 0);
22         }
23     }
24     if(c[1] == 'X') {
25         if(w) {
26             add(2 * x - 0, 2 * y - 1);
27             add(2 * x - 1, 2 * y - 0);
28             add(2 * y - 0, 2 * x - 1);
29             add(2 * y - 1, 2 * x - 0);
30         } else {
31             add(2 * x - 0, 2 * y - 0);
32             add(2 * x - 1, 2 * y - 1);
33             add(2 * y - 0, 2 * x - 0);
34             add(2 * y - 1, 2 * x - 1);
35         }
36     }
37 }
38 for(int i = 1; i <= 2 * n; i++)
39     if(!dfn[i])
40         tarjan(i);
41 for(int i = 1; i <= n; i++) {
42     if(scc[2 * i - 0] == scc[2 * i - 1]) {
43         printf("NO\n");
44         return 0;
45     }
46 }
47 printf("YES\n");
48 //2 * x - a -> 2 * y - b 的边表示, 假如 x 取值为 a, 那么 y 的取值必须为 b
49
50 //输出方案
51 for(int i = 2; i <= 2 * n; i += 2) {
52     if(scc[i - 0] == scc[i - 1]) {
53         printf("NO\n");
54         return 0;
55     } else ans[(i + 1) / 2] = scc[i - 1] < scc[i - 0];
56 }
```

计算几何

字符串

字符串哈希

```
1  namespace String {
2      const int x = 135;
3      const int p1 = 1e9 + 7, p2 = 1e9 + 9;
4      ull xp1[N], xp2[N], xp[N];
```

```

5 void init_xp() {
6     xp1[0] = xp2[0] = xp[0] = 1;
7     for(int i = 1; i < N; i++) {
8         xp1[i] = xp1[i - 1] * x % p1;
9         xp2[i] = xp2[i - 1] * x % p2;
10        xp[i] = xp[i - 1] * x;
11    }
12 }
13 struct HashString {
14     char s[N];
15     int length, subsize;
16     bool sorted;
17     ull h[N], hl[N];
18     ull init(const char *t) {
19         if(xp[0] != 1) init_xp();
20         length = strlen(t);
21         strcpy(s, t);
22         ull res1 = 0, res2 = 0;
23         h[length] = 0;
24         for(int j = length - 1; j >= 0; j--) {
25             #ifdef ENABLE_DOUBLE_HASH
26                 res1 = (res1 * x + s[j]) % p1;
27                 res2 = (res2 * x + s[j]) % p2;
28                 h[j] = (res1 << 32) | res2;
29             #else
30                 res1 = res1 * x + s[j];
31                 h[j] = res1;
32             #endif
33         }
34         return h[0];
35     }
36     //获取子串哈希, 左闭右开
37     ull get_substring_hash(int left, int right) {
38         int len = right - left;
39         #ifdef ENABLE_DOUBLE_HASH
40             unsigned int mask32 = ~(0u);
41             ull left1 = h[left] >> 32, right1 = h[right] >> 32;
42             ull left2 = h[left] & mask32, right2 = h[right] & mask32;
43             return (((left1 - right1 * xp1[len] % p1 + p1) % p1) << 32) |
44                 (((left2 - right2 * xp2[len] % p2 + p2) % p2));
45         #else
46             return h[left] - h[right] * xp[len];
47         #endif
48     }
49     void get_all_subs_hash(int sublen) {
50         subsize = length - sublen + 1;
51         for (int i = 0; i < subsize; ++i)
52             hl[i] = get_substring_hash(i, i + sublen);
53         sorted = 0;
54     }
55
56     void sort_substring_hash() {
57         sort(hl, hl + subsize);
58         sorted = 1;
59     }
60
61     bool match(ull key) const {
62         if (!sorted) assert (0);
63         if (!subsize) return false;
64         return binary_search(hl, hl + subsize, key);
65     }
66 };
67 }

```

Trie

```

1 namespace trie {
2     int t[N][26], sz, ed[N];
3     int _new() {
4         sz++;
5         memset(t[sz], 0, sizeof(t[sz]));

```

```

6     return sz;
7 }
8 void init() {
9     sz = 0;
10    _new();
11    memset(ed, 0, sizeof(ed));
12 }
13 void Insert(char *s, int n) {
14     int u = 1;
15     for(int i = 0; i < n; i++) {
16         int c = s[i] - 'a';
17         if(!t[u][c]) t[u][c] = _new();
18         u = t[u][c];
19     }
20     ed[u]++;
21 }
22 int find(char *s, int n) {
23     int u = 1;
24     for(int i = 0; i < n; i++) {
25         int c = s[i] - 'a';
26         if(!t[u][c]) return -1;
27         u = t[u][c];
28     }
29     return u;
30 }
31 }

```

KMP 算法

```

1 namespace KMP {
2     void get_next(char *t, int m, int *nxt) {
3         int j = nxt[0] = 0;
4         for(int i = 1; i < m; i++) {
5             while(j && t[i] != t[j]) j = nxt[j - 1];
6             nxt[i] = j += (t[i] == t[j]);
7         }
8     }
9     vector<int> find(char *t, int m, int *nxt, char *s, int n) {
10        vector<int> ans;
11        int j = 0;
12        for(int i = 0; i < n; i++) {
13            while(j && s[i] != t[j]) j = nxt[j - 1];
14            j += s[i] == t[j];
15            if(j == m) {
16                ans.push_back(i - m + 1);
17                j = nxt[j - 1];
18            }
19        }
20        return ans;
21    }
22 }

```

manacher 算法

```

1 namespace manacher {
2     char s[N];
3     int p[N], len;
4     void getp(string tmp) {
5         len = 0;
6         for(auto x : tmp) {
7             s[len++] = '#';
8             s[len++] = x;
9         }
10        s[len++] = '#';
11        memset(p, 0, sizeof(int) * (len + 10));
12        int c = 0, r = 0;
13        for(int i = 0; i < len; i++) {
14            if(i <= r) p[i] = min(p[2 * c - i], r - i);
15            else p[i] = 1;
16            while(i - p[i] >= 0 && i + p[i] < len && s[i - p[i]] == s[i + p[i]])

```

```

17         p[i]++;
18         if(i + p[i] - 1 > r) {
19             r = i + p[i] - 1;
20             c = i;
21         }
22     }
23     for(int i = 0; i < len; i++) p[i]--;
24 }
25 void getp(char *tmp, int n) {
26     len = 0;
27     for(int i = 0; i < n; i++) {
28         s[len++] = '#';
29         s[len++] = tmp[i];
30     }
31     s[len++] = '#';
32     memset(p, 0, sizeof(int) * (len + 10));
33     int c = 0, r = 0;
34     for(int i = 0; i < len; i++) {
35         if(i <= r) p[i] = min(p[2 * c - i], r - i);
36         else p[i] = 1;
37         while(i - p[i] >= 0 && i + p[i] < len && s[i - p[i]] == s[i + p[i]])
38             p[i]++;
39         if(i + p[i] - 1 > r) {
40             r = i + p[i] - 1;
41             c = i;
42         }
43     }
44     for(int i = 0; i < len; i++) p[i]--;
45 }
46 int getlen() {
47     return *max_element(p, p + len);
48 }
49 int getlen(string s) {
50     getp(s);
51     return getlen();
52 }
53 }

```

AC 自动机

```

1 struct ac_automaton {
2     int t[N][26], danger[N], tot, fail[N];
3     int dp[N][N];
4     void init() {
5         tot = -1;
6         _new();
7     }
8     int _new() {
9         tot++;
10        memset(t[tot], 0, sizeof(t[tot]));
11        danger[tot] = 0;
12        fail[tot] = 0;
13        return tot;
14    }
15    void Insert(const char *s) {
16        int u = 0;
17        for(int i = 0; s[i]; i++) {
18            if(!t[u][mp[s[i]]]) t[u][s[i] - 'a'] = _new();
19            u = t[u][mp[s[i]]];
20        }
21        danger[u] = 1;
22    }
23    void build() {
24        queue<int> q;
25        for(int i = 0; i < 26; i++) {
26            if(t[0][i]) {
27                fail[i] = 0;
28                q.push(t[0][i]);
29            }
30        }
31        while(q.size()) {

```



```

32         int u = q.front(); q.pop();
33         danger[u] |= danger[fail[u]];
34         for(int i = 0; i < 26; i++) {
35             if(t[u][i]) {
36                 fail[t[u][i]] = t[fail[u]][i];
37                 q.push(t[u][i]);
38             } else t[u][i] = t[fail[u]][i];
39         }
40     }
41 }
42 int query(const char *s) {
43     memset(dp, 0x3f, sizeof(dp));
44     int n = strlen(s);
45     dp[0][0] = 0;
46     for(int i = 0; i < n; i++) {
47         for(int j = 0; j <= tot; j++) if(!danger[j]) {
48             for(int k = 0; k < 26; k++) if(!danger[t[j][k]]) {
49                 dp[i + 1][t[j][k]] = min(dp[i + 1][t[j][k]], dp[i][j] + (s[i] - 'a' != k));
50             }
51         }
52     }
53     int ans = 0x3f3f3f3f;
54     for(int i = 0; i <= tot; i++) if(!danger[i]) {
55         ans = min(ans, dp[n][i]);
56     }
57     return ans == 0x3f3f3f3f ? -1 : ans;
58 }
59 };

```

杂项

int128

```

1  typedef __uint128_t u128;
2  inline u128 read() {
3      static char buf[100];
4      scanf("%s", buf);
5      // std::cin >> buf;
6      u128 res = 0;
7      for(int i = 0; buf[i]; ++i) {
8          res = res << 4 | (buf[i] <= '9' ? buf[i] - '0' : buf[i] - 'a' + 10);
9      }
10     return res;
11 }
12 inline void output(u128 res) {
13     if(res >= 16)
14         output(res / 16);
15     putchar(res % 16 >= 10 ? 'a' + res % 16 - 10 : '0' + res % 16);
16     //std::cout.put(res % 16 >= 10 ? 'a' + res % 16 - 10 : '0' + res % 16);
17 }

```

__builtin_ 内建函数

1. `__builtin_popcount(unsigned int n)` 该函数是判断 n 的二进制中有多少个 1
2. `__builtin_parity(unsigned int n)` 该函数是判断 n 的二进制中 1 的个数的奇偶性
3. `__builtin_ffs(unsigned int n)` 该函数判断 n 的二进制末尾最后一个 1 的位置，从一开始
4. `__builtin_ctz(unsigned int n)` 该函数判断 n 的二进制末尾后面 0 的个数，当 n 为 0 时，和 n 的类型有关
5. `__builtin_clz(unsigned int x)` 返回前导的 0 的个数

tips:

- 如果使用 sort 比较两个函数，不能出现 $a < b$ 和 $a > b$ 同时为真的情况，否则会运行错误。
- 多组数据清空线段树的时候，不要忘记清空全部数组（比如说 lazytag 数组）。
- 注意树的深度和节点到根的距离是两个不同的东西，深度是点数，距离是边长，如果求 LCA 时用距离算会出错。
- 连通性专题：注意判断 $dfn[x]$ 和 $low[y]$ 的关系时是否不小心两个都达成 low 了