XCPC 算法模版 V2.0

galiyu

2025 年 4 月 3 日

目录

1 基础算法

1.1 二分

有些时候 check 函数中一些变量可能会超出 i64 的范围，为了防止这种情况发生，我们可以设置一个上界，这些变量运算时与上界取 min。

```
// created on 2024-8-23

// 整数二分(1)
int l = 0, r = n;
while (r > l){
    int mid = (l + r) / 2;
    if (check(mid)) r = mid;
    else l = mid + 1;
}
// 整数二分(2)
int l = 0, r = n;
while (r > l){
    int mid = (l + r + 1) / 2;
    if (check(mid)) l = mid;
    else r = mid - 1;
}
// 浮点二分
const double acc = 1e-6; // 设置精度
double l = 0, r = n;
while (r - l > acc){
    double mid = (l + r) / 2;
    if (check(mid)) r = mid;qwq
    else l = mid;
}
```

1.2 三分

通常是用于寻找某个函数的最值。

```
// created on 2024-8-23

//三分求f函数的最大值（定义域为整数）
int maximum_int(int L, int R) {
    while (R > L) {
        int m1 = (2 * L + R) / 3;
        int m2 = (2 * R + L + 2) / 3;
        if (f(m1) > f(m2)) R = m2 - 1;
        else L = m1 + 1;
    }
    return L; //f(L)为最大值
}
//三分求f函数的最小值（定义域为整数）
int minimun_int(int L, int R) {
    while (R > L) {
        int m1 = (2 * L + R) / 3;
        int m2 = (2 * R + L + 2) / 3;
        if (f(m1) < f(m2)) R = m2 - 1;
        else L = m1 + 1;
    }
    return L; //f(L)为最小值
}
//三分求f函数的最大值（定义域为实数）
const double eps = 1e-6;
double maximum_double(double L, double R) {
    while (R - L > eps) { // for i in range(100):
        double m1 = (2 * L + R) / 3;
        double m2 = (2 * R + L) / 3;
```

```
        if (f(m1) > f(m2)) R = m2;
        else L = m1;
    }
    return L; //f(L)为最大值
}
//三分求f函数的最小值（定义域为实数）
const double eps = 1e-6;
double minimun_double(double L, double R) {
    while (R - L > eps) { // for i in range(100):
        double m1 = (2 * L + R) / 3;
        double m2 = (2 * R + L) / 3;
        if (f(m1) < f(m2)) R = m2;
        else L = m1;
    }
    return L; //f(L)为最小值
}
```

1.3 前缀和与差分

1.3.1 二维前缀和

假定求数组 $a_{i,j}$ 的前缀和，前缀和数组为 $sum_{i,j}$，计算操作左下角标 (x_1, y_1)，右上角标 (x_2, y_2)。

1) 计算 sum 数组

$$sum_{i,j} = sum_{i-1,j} + sum_{i,j-1} - sum_{i-1,j-1} + a_{i,j}$$

2) 计算一段的前缀和

$$sum_{x1,y1} - sum_{x1,y2-1} - sum_{x2-1,y1} + sum_{x2-1,y2-1}$$

1.3.2 二维差分

假定差分数组为 $sum_{i,j}$, 操作左下角标 (x_1, y_1), 操作右上角标 (x_2, y_2)。

$$sum_{x_1,y_1} + = d, sum_{x_2+1,y_1} - = d, sum_{x_1,y_2+1} - = d, sum_{x_2+1,y_2+1} + = d$$

1.3.3 三维前缀和

假定求数组 $a_{i,j,k}$ 的前缀和，前缀和数组为 $sum_{i,j,k}$。

1) 计算 sum 数组

$$\begin{aligned} sum_{i,j,k} &= +sum_{i-1,j,k} + sum_{i,j-1,k} + sum_{i,j,k-1} \\ &= -sum_{i-1,j-1,k} - sum_{i-1,j,k-1} - sum_{i,j-1,k-1} \\ &= +sum_{i-1,j-1,k-1} + a_{i,j,k} \end{aligned}$$

2) 计算一段的前缀和 (咕咕咕)

1.3.4 三维差分

假定差分数组为 $sum_{i,j,k}$, 两点确定一个立方体, 操作左下角标 (x_1, y_1, z_1), 操作右上角标 (x_2, y_2, z_2)。

$$\begin{gathered} sum_{x_1,y_1,z_1} + = d, sum_{x_1,y_1,z_2+1} - = d, sum_{x_1,y_2+1,z_1} - = d \\ sum_{x_2+1,y_1,z_1} - = d, sum_{x_2+1,y_2+1,z_1} + = d, sum_{x_2+1,y_1,z_2+1} + = d \\ sum_{x_1,y2+1,z_2+1} + = d, sum_{x_2+1,y_2+1,z_2+1} - = d \end{gathered}$$

1.4 离散化

```
// created on 2024-8-23

```

```
// a[i] 为初始数组，下标范围为 [1, n]。
// len 为离散化后数组的有效长度。
// 离散化整个数组的同时求出离散化后本质不同数的个数。
sort(a + 1, a + 1 + n);
len = unique(a + 1, a + n + 1) - a - 1;

// vector 版本的离散化
vector<int> tmp(arr);
sort(tmp.begin(), tmp.end());
tmp.erase(unique(tmp.begin(), tmp.end()), tmp.end());
```

1.5 位运算及相关库函数

自制绝赞位运算函数, 虽然好像一般也没怎么用过。

```
// created on 2024-8-23

// 获取 a 的第 b 位，最低位编号为 0。
int getBit(int a, int b) {return (a >> b) & 1;}
// 将 a 的第 b 位设置为 0 ，最低位编号为 0。
int unsetBit(int a, int b) {return a & ~(1 << b);}
// 将 a 的第 b 位设置为 1 ，最低位编号为 0。
int setBit(int a, int b) {return a | (1 << b);}
// 将 a 的第 b 位取反 ，最低位编号为 0。
int flapBit(int a, int b) {return a ^ (1 << b);}
```

C++ 自带绝赞库函数, 时间复杂度 $O(1)$, 这些函数都可以在函数名末尾添加 ull 来使参数类型变为 ull (返回值仍然是 int 类型)。

```
// created on 2024-8-23

// 返回 x 的二进制末尾最后一个 1 的位置。
// 位置的编号从1开始(最低位编号为1)，当 x 为 0 时返回 0。
```

```
int __builtin_ffs(int x)
// 返回 x 的二进制的前导 0 的个数。
// 当 x 为 0 时，结果未定义。
int __builtin_clz(unsigned int x)
// 返回 x 的二进制末尾连续 0 的个数。
// 当 x 为 0 时，结果未定义。
int __builtin_ctz(unsigned int x)
// 当 x 的符号位为 0 时返回 x 的二进制的前导 0 的个数减一。
// 否则返回 x 的二进制的前导 1 的个数减一。
int __builtin_clrsb(int x)
// 返回 x 的二进制中 1 的个数。
int __builtin_popcount(unsigned int x)
// 判断 x 的二进制中 1 的个数的奇偶性。
int __builtin_parity(unsigned int x)
```

1.6 C++ 标准库

介绍一些常用板块。

1.6.1 查找后继

```
// created on 2024-8-23

auto it = lower_bound(a + 1, a + len + 1, x) - a;
// 查询 x 离散化后对应的编号。
// 防越界。
if (SZ(xxx) && it <SZ(xxx)) {
    xxx;
}
else continue;
```

1.6.2 判断非递减

```
// created on 2024-8-23

// a 数组 [start,end) 区间是否是非递减的。
// 返回值为 bool。
is_sorted(a + start, a + end);
```

1.6.3 数组打乱

随机的力量。

```
// created on 2024-8-23

// md 这个太长了渲染不出来
mt19937
rnd(chrono::steady_clock::now().time_since_epoch().count());
shuffle(ver.begin(), ver.end(), rnd);
```

1.6.4 批量递增赋值函数

```
// created on 2024-8-23

// 将a数组[start,end)区间赋值成“x, x+1, x+2, …”
iota(a + start, a + end, x);
```

1.6.5 全排列函数

next_permutation 算法，是按照字典序顺序输出的全排列。

prev_permutation 算法，是按照逆字典序顺序输出的全排列。

```
// created on 2024-8-23

do {
    for (auto it : a) cout << it << " ";
    cout << endl;
} while (next_permutation(a.begin(), a.end()));
```

1.6.6 数组元素累加

```
// created on 2024-8-23

// 将 a 数组 [start,end) 区间的元素进行累加
// 并输出累加和 +x 的值。
accumulate(a + start, a + end, x);
```

1.6.7 迭代器相关

```
// created on 2024-8-23

//构建一个UUU容器的正向迭代器，名字叫it。
UUU::iterator it;
//创建一个正向迭代器，++ 操作时指向下一个。
vector<int>::iterator it;
//创建一个反向迭代器，++ 操作时指向上一个。
vector<int>::reverse_iterator it;
```

1.6.8 部分数学库函数

时间复杂度应该都是 $O(logn)$。

```
// created on 2024-8-23

```

```
// 返回 2^x。
exp2(x)
// 2^k = x, 返回 k。
log2(x)
gcd(x,y)/lcm(x,y)
```

1.6.9 C++ Set 库

```
// created on 2024-8-23

// set<int> s;
// 删除所有值等于 xxx 的元素。
s.erase(xxx);
// 删除一个值等于 xxx 的元素。
s.erase(s.find(xxx));
// 删除第一个元素。
s.erase(s.begin());
```

1.6.10 C++ pb_ds 库

可以查询下标的 set, 美名曰 ordered_set, 使用 find_by_order 查询下标 (从 0 开始) 的数值。

```
// created on 2024-8-23

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

//less<pii> 为按小到大
typedef pair<int, int> pii;
#define ordered_set tree<pii, null_type, less<pii>,
```

```
rb_tree_tag,tree_order_statistics_node_update>
```

gp_hash_table 的声明, 卡常神器。

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/hash_policy.hpp>
using namespace __gnu_pbds;

gp_hash_table<long long,int> mp;
```

1.6.11 C++ Bitset

```
// created on 2024-8-23

// 如果输入的是01字符串，可以直接使用">>"读入
bitset<10> s;
cin >> s;
//使用只含01的字符串构造——bitset<容器长度>B（字符串）
string S; cin >> S;
bitset<32> B (S);
//使用整数构造（两种方式）
int x; cin >> x;
bitset<32> B1 (x);
bitset<32> B2 = x;
// 构造时，尖括号里的数字不能是变量
int x; cin >> x;
bitset<x> ans; // 错误构造
[] //随机访问
set(x) //将第x位置1，x省略时默认全部位置1
reset(x) //将第x位置0，x省略时默认全部位置0
flip(x) //将第x位取反，x省略时默认全部位取反
to_ullong() //重转换为ULL类型
```

```
to_string() //重转换为ULL类型
count() //返回1的个数
any() //判断是否至少有一个1
none() //判断是否全为0
bitset<23> B1("11101001"), B2("11101000");
cout << (B1 ^ B2) << "\n"; //按位异或
cout << (B1 | B2) << "\n"; //按位或
cout << (B1 & B2) << "\n"; //按位与
cout << (B1 == B2) << "\n"; //比较是否相等
cout << B1 << " " << B2 << "\n"; //你可以直接使用cout输出
auto pa = (ull*)&a; //把bitset作为一个数输出
cout << pa[0]; //输出0-63位作为数的结果
cout << pa[1]; //输出64-127位作为数的结果
```

1.7 高精度

```
// created on 2025-1-18

const int N=1010;
struct BigInt {
    int a[N];
    BigInt(int x=0):a{} {
        for (int i=0;x;i++) {
            a[i]=x%10,x/=10;
        }
    }
    BigInt &operator*=(int x) {
        for (int i=0;i<N;i++) {
            a[i]*=x;
        }
        for (int i=0;i<N-1;i++) {
```

```
            a[i+1]+=a[i]/10,a[i]%=10;
        }
        return *this;
    }
    BigInt &operator/=(int x) {
        for (int i=N-1;i>=0;i--) {
            if (i) a[i-1]+=a[i]%x*10;
            a[i]/=x;
        }
        return *this;
    }
    BigInt &operator+=(const BigInt &x) {
        for (int i=0;i<N;i++) {
            a[i]+=x.a[i];
            if (a[i]>=10) {
                a[i+1]+=1,a[i]-=10;
            }
        }
        return *this;
    }
};
std::ostream &operator<<(std::ostream &o,const BigInt &a) {
    int t=N-1;
    while (a.a[t]==0) {
        t--;
    }
    for (int i=t;i>=0;i--) {
        o<<a.a[i];
    }
    return o;
}
```

2 字符串算法

2.1 KMP

2.1.1 Normal

应用:

1. 在字符串中寻找子串。
2. 最小周期: 字符串长度 - f[字符串长度], 形如 acaca 中 ac 是一个合法周期。
3. 最小循环节: 区别于周期，当字符串长度 $(n\%(n-f[n]) == 0)$ 时，等于最小周期，否则为 n。形如 acac 中 ac 和 acac 是循环节, 而 aca 不是。

```cpp
// created on 2024-8-23

// kmp 原函数
typedef vector<int> vi;

vi kmp(const string s) {
        const int n = s.size();
        vi f(n + 1);
        for (int i = 1,j = 0;i < n;i++) {
                while (j and s[i] != s[j]) j = f[j];
                j += (s[i] == s[j]);
                f[i + 1] = j;
        }
        return f;
}
// 匹配字符串
string s = "ababab", t = "ab";
```

```
auto next = kmp(t);
for (int i = 0,j = 0;i < s.size();i++) {
    while (j and s[i] != t[j]) j = next[j];
    if (t[j] == s[i]) j++;
    if (j == t.size()) {
        cout << i << endl;
        j = next[j];
    }
}
```

2.1.2 KMPAutoMaton

除此之外 KMP 还有一种比较特殊的用法, KMP 自动机, 像是单字符串的 AC 自动机。

```
// created on 2024-9-12

// CF 1721E
// 给定字符串 S,以及 Q 个字符串 Ti,求把 S 分别与每个 Ti 拼接后
// 最靠右的 |Ti| 个前缀的最长 border 长度, 询问分别独立。
// |S|<=1E6, Q<=1E5, |Ti|<=10

// O(26n) 的时间复杂度增加模板串长度
string s,t;
int n,q;
signed main(){
    cin>>s;
    s=" "+s;
    int n=s.size()-1;
    vector<array<int,26>> ch(n+20);
    vector<int> fail(n+20);
    for (int i=0;i<=n;i++) {
```

```
        if (i>1) fail[i]=ch[fail[i-1]][s[i]-'a'];
        if (i<n) {
            for (int j=0;j<26;j++) {
                if (s[i+1]-'a'==j) ch[i][j]=i+1;
                else ch[i][j]=ch[fail[i]][j];
            }
        }
    }
    int q;
    cin>>q;
    s+=string(10,'*');
    rep(tc,1,q+1) {
        cin>>t;
        int m=(t).size();
        for (int i=n+1;i<=n+m;i++) s[i]=t[i-n-1];
        for (int i=n;i<=n+m;i++) {
            if (i>1) fail[i]=ch[fail[i-1]][s[i]-'a'];
            if (i<n+m) {
                for (int j=0;j<26;j++) {
                    if (s[i+1]-'a'==j) ch[i][j]=i+1;
                    else ch[i][j]=ch[fail[i]][j];
                }
            }
            if (i>n) cout<<fail[i]<<" ";
        }
        cout<<endl;
    }
}
```

2.2 EXKMP

```
// created on 2024-8-23

#define rep(i,a,n) for(int i=a;i<n;i++)
#define SZ(x) ((int)(x).size())
typedef vector<int> vi;

vi zFunction(const string& S) {
	vi z(SZ(S));
	int l=-1,r=-1;
	rep(i,1,SZ(S)) {
		z[i]=(i>=r?0:min(r-i,z[i-l]));
		while (i+z[i]<SZ(S)&&S[i+z[i]]==S[z[i]]) {
			z[i]++;
		}
		if (i+z[i]>r) {
			l=i,r=i+z[i];
		}
	}
	return z;
}
```

2.3 Hash

```
// created on 2025-1-18

#define rep(i,a,n) for(int i=a;i<n;i++)
#define SZ(s) ((int)s.size())
typedef long long ll;
typedef uint64_t ull;

struct H {
```

```
    ull x; H(ull x=0) : x(x) {}
    H operator+(H o) { return x + o.x + (x + o.x < x); }
    H operator-(H o) { return *this + ~o.x; }
    H operator*(H o) { auto m = (__uint128_t)x * o.x;
        return H((ull)m) + (ull)(m >> 64); }
    ull get() const { return x + !~x; }
    bool operator==(H o) const { return get() == o.get(); }
    bool operator<(H o) const { return get() < o.get(); }
};
// (order ~ 3e9; random also ok)
static const H C = (ll)1e11+3;

struct HashInterval {
    vector<H> ha,pw;
    HashInterval(string& str):ha(SZ(str)+1),pw(ha) {
        pw[0]=1;
        rep(i,0,SZ(str))
            ha[i+1]=ha[i]*C+str[i],
            pw[i+1]=pw[i]*C;
    }
    H getHash(int a,int b) { // hash [a, b)
        return ha[b]-ha[a]*pw[b-a];
    }
};
// abcd
// get(3,4) * v[2] + get(1,2) == cdab
// get(1,2) * v[2] + get(3,4) == abcd
vector<H> getHashes(string& str,int length) {
    if (SZ(str)<length) return {};
    H h=0,pw=1;
    rep(i,0,length) {
```

```
        h=h*C+str[i],pw=pw*C;
    }
    vector<H> ret={h};
    rep(i,length,SZ(str)) {
        ret.push_back(h=h*C+str[i]-pw*str[i-length]);
    }
    return ret;
}
H hashString(string& s){
    H h{};
    for(char c:s) h=h*C+c;return h;
}
signed main() {
    string s="abcabc",t="abc";
    HashInterval Q(s),P(t);
    cout<<Q.getHash(0,3).x<<" ";
    cout<<P.getHash(0,3).x<<endl;
}
```

2.4 Manacher

```
// created on 2025-1-18

#define rep(i,a,n) for(int i=a;i<n;i++)
#define SZ(x) ((int)(x).size())
typedef vector<int> vi;

// O(N)
array<vi,2> manacher(const string& s) {
    int n=SZ(s);
    array<vi,2> p={vi(n+1), vi(n)};
```

```
    rep(z,0,2) for (int i=0,l=0,r=0;i<n;i++) {
        int t=r-i+!z;
        if (i<r) p[z][i]=min(t,p[z][l+t]);
        int L=i-p[z][i],R=i+p[z][i]-!z;
        while (L>=1 && R+1<n && s[L-1] == s[R+1]) {
            p[z][i]++;
            L--;
            R++;
        }
        if (R>r) l=L,r=R;
    }
    return p;
}
```

2.5 Trie

```
// created on 2024-8-23

struct Trie{
    const int N=1e6+10;
    int nex[N][26],cnt;
    bool ok[N];
    void insert(string s) {
        int p=0;
        for (int i=0;i<SZ(s);i++) {
            int c=s[i]-'a';
            if (!nex[p][c]) nex[p][c]=++cnt;
            p=nex[p][c];
        }
        ok[p]=1;
    }
```

```
    bool find(string s) {
        int p=0;
        for (int i=0;i<SZ(s);i++) {
            int c=s[i]-'a';
            if (!nex[p][c]) return 0;
            p=nex[p][c];
        }
        return ok[p];
    }
}trie;
```

可持久化 01 Trie 树：给你长度为 n 的数组 a，以及 q 次询问，询问参数 l,r,x，a 数组区间 $[l,r]$ 范围内，最大的 $a_i \bigoplus x$ 是多少。

```
// created on 2025-3-27
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

const int N=200010;
int n,m,idx,cnt;
int rt[N],ch[N*33][2],siz[N*33];

void insert(int v){
    rt[++idx]=++cnt; //新根开点
    int x=rt[idx-1]; //旧版
    int y=rt[idx];   //新版
    for(int i=31;i>=0;i--){
    int j=v>>i&1;
    ch[y][!j]=ch[x][!j]; //异位继承
    ch[y][j]=++cnt;      //新位开点
```

```
        x=ch[x][j];
        y=ch[y][j];
        siz[y]=siz[x]+1;       //新位多 1
        }
}
int query(int x,int y,int v){
        int ans=0;
        for(int i=31;i>=0;i--){
        int j=v>>i&1;
        if(siz[ch[y][!j]]>siz[ch[x][!j]]) {
            x=ch[x][!j];
            y=ch[y][!j];
            ans+=(1<<i);
        }
        else {
            x=ch[x][j];
            y=ch[y][j];
        }
        }
        return ans;
}
void clear() {
        for (int i=0;i<idx+10;i++) rt[i]=0;
        for (int i=0;i<cnt+10;i++) ch[i][0]=ch[i][1]=siz[i]=0;
        idx=cnt=0;
}
int main(){
        std::ios::sync_with_stdio(0);
        std::cin.tie(0);
        int t;
        cin>>t;
```

```
    while (t--) {
        clear();
        cin>>n>>m;
        int s=0;
        for (int i=1;i<=n;i++) {
            int x;
            cin>>x;
            insert(x);
        }
        while (m--) {
            int l,r,x;
            cin>>l>>r>>x;
            cout<<query(rt[l-1],rt[r],x)<<endl;
        }
    }
}
```

2.6 ACAutoMaton

```
// created on 2024-8-23

const int N=1e6+10;
struct ACAutoMaton {
    const int ALPHABET=26;
    ll ch[N][ALPHABET],link[N];
    ll tot,rt;
    ll cnt[N];
    void init() {
        tot=rt=0;
        memset(ch,0,sizeof ch);
        memset(link,0,sizeof link);
```

```
        memset(cnt,0,sizeof cnt);
    }
    void add(const string s,int id) {
        if (!rt) rt=++tot;
        int p=rt;
        for (auto c : s) {
            int x=(c-'a');
            p=ch[p][x]=(ch[p][x]?ch[p][x]:++tot);
        }
        cnt[p]++;
    }
    void build() {
        queue<int> q;
        for (int i=0;i<ALPHABET;i++) {
            if (ch[1][i]) {
                link[ch[1][i]]=1;
                q.push(ch[1][i]);
            } else ch[1][i]=1;
        }
        while (!q.empty()) {
            int x=q.front();
            q.pop();
            for (int i=0;i<ALPHABET;i++) {
                int y=ch[x][i];
                if (y) {
                    link[y]=ch[link[x]][i];
                    q.push(y);
                } else ch[x][i]=ch[link[x]][i];
            }
        }
    }
```

```
};
```

结合二进制分组算法的 ACAM, 以 logn 时间复杂度为代价做到在线插入。

```
// created on 2024-8-23

// 其中包括 AC 自动机合并
// 其实就是 把 AC 自动机的 trie 树部分给合并
// 然后再把 link 在 build 一遍

struct AhoCorasick {
    static constexpr int ALPHABET = 26;
    int tire[N][ALPHABET],next[N][ALPHABET],link[N];
    int rt[N],size[N],top,tot;
    int ed[N],cnt[N];
    void build(int root) {
        queue<int> q;
        for (int i=0;i<ALPHABET;i++) {
            if(tire[root][i]) {
                link[next[root][i]=tire[root][i]]=root;
                q.push(next[root][i]);
            }
            else {
                next[root][i]=root;
            }
        }
        while (q.size()) {
            int top=q.front();
            q.pop();
            for (int i=0;i<ALPHABET;i++) {
                if (tire[top][i]) {
                    next[top][i]=tire[top][i];
```

```
                link[next[top][i]]=next[link[top]][i];
                q.push(next[top][i]);
            }
            else next[top][i]=next[link[top]][i];
        }
        cnt[top]=ed[top]+cnt[link[top]];
    }
}
int merge(int a,int b) {
    if (!a||!b) return a+b;
    ed[a]+=ed[b];
    for (int i=0;i<ALPHABET;i++)
    tire[a][i]=merge(tire[a][i],tire[b][i]);
    return a;
}
void add(string a,int val) {
    rt[++top]=++tot;
    size[top]=1;
    int now=rt[top];
    for (auto c : a){
        if(!tire[now][c-'a']) tire[now][c-'a']=++tot;
        now=tire[now][c-'a'];
    }
    ed[now]+=val;
    while (size[top]==size[top-1]){
        --top;
        rt[top]=merge(rt[top],rt[top+1]);
        size[top]+=size[top+1];
    }
    build(rt[top]);
}
```

```
    int ask(string a) {
        int res=0;
        for (int i=1;i<=top;i++) {
            int now=rt[i];
            for (auto c : a) {
                now=next[now][c-'a'];
                res+=cnt[now];
            }
        }
        return res;
    }
};
```

2.7 SuffixArray

```
// created on 2024-8-23

struct SuffixArray {
    int n;
    vector<int> sa,rk,lc;
    SuffixArray(const string &s) {
        n=s.length();
        sa.resize(n);
        lc.resize(n-1);
        rk.resize(n);
        iota(sa.begin(),sa.end(),0);
        sort(sa.begin(),sa.end(),[&](int a,int b) {
            return s[a]<s[b];
        });
        rk[sa[0]]=0;
        for (int i=1;i<n;++i) {
```

```
            rk[sa[i]]=rk[sa[i-1]]+(s[sa[i]]!=s[sa[i-1]]);
        }
        int k=1;
        vector<int> tmp,cnt(n);
        tmp.reserve(n);
        while (rk[sa[n-1]]<n-1) {
            tmp.clear();
            for (int i=0;i<k;++i)
            tmp.push_back(n-k+i);
            for (auto i : sa) {
                if (i>=k) {
                    tmp.push_back(i - k);
                }
            }
            fill(cnt.begin(),cnt.end(),0);
            for (int i=0;i<n;++i) ++cnt[rk[i]];
            for (int i=1;i<n;++i) cnt[i]+=cnt[i-1];
            for (int i=n-1;i>=0;--i) {
                sa[--cnt[rk[tmp[i]]]]=tmp[i];
            }
            swap(rk,tmp);
            rk[sa[0]]=0;
            for (int i=1;i<n;++i) {
                rk[sa[i]]=rk[sa[i-1]]
                +(tmp[sa[i-1]]<tmp[sa[i]]||sa[i-1]+k==n
                ||tmp[sa[i-1]+k]<tmp[sa[i]+k]);
            }
            k*=2;
        }
        for (int i=0,j=0;i<n;++i) {
            if (rk[i]==0) {
```

```
                j=0;
                continue;
            }
            for (j-=j>0;i+j<n&&
            sa[rk[i]-1]+j<n&&s[i+j]==s[sa[rk[i]-1]+j];) {
                ++j;
            }
            lc[rk[i]-1]=j;
        }
    }
};
```

2.8 SuffixAutoMaton

```
// created on 2024-8-23

//must #define int long long if you use the self_test
struct SAM {
    static const int MAXN=1000010,MAXS=28;
    int tot=1,last=1;
    int link[MAXN<<1],ch[MAXN<<1][MAXS];
    int len[MAXN<<1],endpos[MAXN<<1];
    void clear() {
        for (int i=0;i<=tot;i++) {
            link[i]=len[i]=endpos[i]=0;
            for (int k=0;k<MAXS;k++) {
                ch[i][k]=0;
            }
        }
        tot=1;last=1;
    }
```

```
    // extend a char,usual as [1-26]
    void extend(int w) {
        int p=++tot,x=last,r,q;
        endpos[p]=1;
        for (len[last=p]=len[x]+1;
        x&&!ch[x][w];x=link[x]) {
            ch[x][w]=p;
        }
        if (!x) link[p]=1;
        else if (len[x]+1==len[q=ch[x][w]]) link[p]=q;
        else {
            link[r=++tot]=link[q];
            memcpy(ch[r],ch[q],sizeof ch[r]);
            len[r]=len[x]+1;
            link[p]=link[q]=r;
            for (;x&&ch[x][w]==q;x=link[x]) {
                ch[x][w]=r;
            }
        }
    }
    // attention! memory of vector
    vector<int> p[MAXN<<1];
    void dfs(int u) {
        int v;
        for (int i=0;i<p[u].size();i++) {
            v=p[u][i];
            dfs(v);
            endpos[u]+=endpos[v];
        }
    }
    void get_endpos() {
```

```
        for (int i=1;i<=tot;i++) p[i].clear();
        for (int i=2;i<=tot;i++) {
            p[link[i]].push_back(i);
        }
        dfs(1);
        for (int i=1;i<=tot;i++) p[i].clear();
    }
    // test template is already and right
    // self_test will clear predate
    static const int STC = 998244353;
    void self_test() {
        clear();
        for (int i=1;i<=1000;i++) extend(i*i%26+1);
        int tmp=107*last+301*tot;
        for (int i=1;i<=tot;i++){
            tmp=(tmp*33+link[i]*101+len[i]*97) % STC;
            for (int k=1;k<MAXS;k++) {
                tmp=(tmp+k*ch[i][k])%STC;
            }
        }
        // stage1
        // check build parent tree
        assert("stage 1" && tmp == 393281314);
        tmp=0; get_endpos();
        for (int i=1;i<=tot;i++) {
            tmp=(tmp*33+endpos[i])%STC;
        }
        // stage2
        // check endpos's mean
        // maybe error if modify it.
        assert("stage 2" && tmp == 178417668);
```

```
        cout<<"Self Test Passed.";
        cout<<Remember to delete this function's use.";
        cout<<endl;
        clear();
    }
    void debug_print() {
        for (int i=1;i<=tot;i++) {
            cout<<"node:"<<i<<" father:";
            cout<<link[i]<<" endpos:";
            cout<<endpos[i]<<"len:"<<len[i]<<endl;
        }
    }
    ll work() {
        // Do Something
        // Example luogu P3804
        ll ans=0;
        get_endpos();
        for (int i=1;i<=tot;i++) if (endpos[i]>=2) {
            ans=max(ans,(ll)endpos[i]*len[i]);
        }
        return ans;
    }
};
```

2.9 PalindromeAutomaton

```
// created on 2024-8-23

// 1. 本质不同的回文串个数: idx - 2;
// 2. 回文子串出现次数;
// 对于一个字符串 s
```

```
// 它的本质不同回文子串个数最多只有 |s| 个。
// 那么回文树的时间复杂度为 O(|s|)。

struct PalindromeAutomaton {
    constexpr static int N=5e5+10;
    int tr[N][26],fail[N],len[N];
    int cntNodes,last;
    int cnt[N];
    string s;
    PalindromeAutomaton(string s) {
        memset(tr,0,sizeof tr);
        memset(fail,0,sizeof fail);
        len[0]=0,fail[0]=1;
        len[1]=-1,fail[1]=0;
        cntNodes=1;
        last=0;
        this->s=s;
    }
    void insert(char c,int i) {
        int u=get_fail(last,i);
        if (!tr[u][c-'a']) {
            int v=++cntNodes;
            fail[v]=tr[get_fail(fail[u],i)][c-'a'];
            tr[u][c-'a']=v;
            len[v]=len[u]+2;
            cnt[v]=cnt[fail[v]]+1;
        }
        last=tr[u][c-'a'];
    }
    int get_fail(int u,int i) {
        while (i-len[u]-1<=-1||s[i-len[u]-1]!=s[i]) {
```

```
            u=fail[u];
        }
        return u;
    }
};
```

2.10 Other

2.10.1 最长公共子序列

```
// created on 2024-8-23

// 求最长公共子序列 LCS
// n <= 1e5

const int INF=0x7ffffff;
int n,a[maxn],b[maxn],f[maxn],p[maxn];
int main(){
    cin >> n;
    for (int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        p[a[i]]=i; //将第二个序列中的元素映射到第一个中
    }
    for (int i=1;i<=n;i++){
        scanf("%d",&b[i]);
        f[i]=INF;
    }
    int len=0;
    f[0]=0;
    for (int i=1;i<=n;i++){
        if (p[b[i]]>f[len]) f[++len]=p[b[i]];
```

```
        else {
            int l=0,r=len;
            while (l<r) {
                int mid=(l+r)>>1;
                if (f[mid]>p[b[i]]) r=mid;
                else l=mid+1;
            }
            f[l]=min(f[l],p[b[i]]);
        }
    }
    cout<<len<<"\n";
    return 0;
}
```

3 常用数据结构

3.1 Stack

这里仅演示单调栈喵。

```
// created on 2024-8-23

int a[N],nge[N];
stack<int> sta;
a[n]=INF;
for(int i=0;i<=n;++i) {
    while(!sta.empty()&&a[i]>a[sta.top()]){
        nge[sta.top()]=i;
        sta.pop();
    }
    sta.push(i);
}
```

3.2 UnionFind

```
// created on 2025-1-18

struct DSU {
        vi f,siz;
        DSU() {}
        DSU(int n) {init(n);}
        void init(int n) {
                f.resize(n);
                siz.assign(n,1);
                iota(f.begin(),f.end(),0);
        }
```

```
	int find(int x) {
		while (x!=f[x]) {
			x=f[x]=f[f[x]];
		}
		return x;
	}
	bool merge(int x,int y) {
		x=find(x),y=find(y);
		if (x==y) return false;
		siz[x]+=siz[y];
		f[y]=x;
		return true;
	}
};
```

3.3 STtable

```
// created on 2025-1-18

const int N=2e5+10,LOGN=19;
template<class T>
struct STTable {
    int n;
    T f[LOGN+1][N],g[LOGN+1][N];
    void init(vector<T> a) {
        n=SZ(a);
        assert(n<(1<<LOGN)&&n<N);
        rep(i,0,n) f[0][i]=g[0][i]=a[i];
        rep(j,1,LOGN) for (int i=0;i+(1<<j)-1<n;i++) {
            f[j][i]=min(f[j-1][i],f[j-1][i+(1<<(j-1))]);
            g[j][i]=max(g[j-1][i],g[j-1][i+(1<<(j-1))]);
```

```cpp
        }
    }
    T querymin(int l,int r) {
        assert(l<=r);
        int len=31-__builtin_clz(r-l+1);
        return min(f[len][l],f[len][r-(1<<len)+1]);
    }
    T querymax(int l,int r) {
        assert(l<=r);
        int len=31-__builtin_clz(r-l+1);
        return max(g[len][l],g[len][r-(1<<len)+1]);
    }
};
STTable<int> f;
```

3.4 Fenwick

3.4.1 Normal

```cpp
// created on 2025-1-18

template <typename T>
struct Fenwick {
    int n;
    vector<T> a;
    Fenwick(int n_=0) {
        init(n_);
    }
    void init(int n_) {
        n=n_;
        a.assign(n,T{});
```

```
    }
    void add(int x,const T &v) { // 注意下标自动抬1
        for (int i=x+1;i<=n;i+=(i&-i)) {
            a[i-1]=a[i-1]+v;
        }
    }
    T sum(int x) { // 查值[1-(x-1)]
        T ans{};
        for (int i=x;i>0;i-=(i&-i)) {
            ans=ans+a[i-1];
        }
        return ans;
    }
};
```

3.4.2 Fenwick2D

考虑到有时候出题人比较毒瘤, 赛时我可能铸币, 可能连二维树状数组都写炸, 所以这里贴另一套, 但码风不统一就有点。

```
struct FT {
    vector<ll> s;
    FT(int n):s(n) {}
    void update(int pos,ll dif) { // a[pos]+=dif
        for (;pos<sz(s);pos|=pos+1) s[pos]+=dif;
    }
    ll query(int pos) { // sum of values in [0, pos)
        ll res=0;
        for (;pos>0;pos&=pos-1) res+=s[pos-1];
        return res;
    }
    // min pos st sum of [0, pos] >= sum
```

```
    // Returns n if no sum is >= sum
    // or -1 if empty sum is.
    int lower_bound(ll sum) {
        if (sum<=0) return -1;
        int pos=0;
        for (int pw=1<<25;pw;pw>>=1) {
            if (pos+pw<=sz(s)&&s[pos+pw-1]<sum) {
                pos+=pw, sum-=s[pos-1];
            }
        }
        return pos;
    }
};
struct FT2 {
    vector<vi> ys; vector<FT> ft;
    FT2(int limx) : ys(limx) {}
    void fakeUpdate(int x,int y) {
        for (;x<sz(ys);x|=x+1) ys[x].push_back(y);
    }
    void init() {
        for (vi& v : ys) {
            sort(all(v)), ft.emplace_back(sz(v));
        }
    }
    int ind(int x,int y) {
        return (lower_bound(all(ys[x]),y)-ys[x].begin());
    }
    void update(int x,int y,ll dif) {
        for (;x<sz(ys);x|=x+1) {
            ft[x].update(ind(x,y),dif);
        }
```

```
    }
    ll query(int x,int y) {
        ll sum=0;
        for (;x;x&=x-1) {
            sum+=ft[x-1].query(ind(x-1, y));
        }
        return sum;
    }
};
```

4 基础数学

4.1 线性代数

为什么会有这个板块, 因为线性代数是我爹。无数次被线性代数斩于马下。

4.1.1 行列式基础

矩阵 A 的行列式记为 $det(A)$, 也可以记为 $|A|$。对于一个 n 阶的方块矩阵可直观的定义为 $\det(A)=\sum_{\sigma\in S_n}\operatorname{sgn}(\sigma)\prod_{i=1}^{n}a_{i,\sigma(i)}$。

那么这个该如何使用呢, 对于一个 3×3 矩阵有:

$$A=\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

对于 $n=3$, 对称群有 6 个排列。举例子计算前三个排列:
1) $\sigma_1=(1,2,3)$, 这个排列没有逆序对, $\operatorname{sgn}(\sigma_1)=1$。

$$\prod_{i=1}^{3}a_{i,\sigma_1(i)}=a\times e\times i$$

2) $\sigma_2=(1,3,2)$, 这个排列有一个逆序对, $\operatorname{sgn}(\sigma_2)=-1$。

$$\prod_{i=1}^{3}a_{i,\sigma_2(i)}=a\times f\times h$$

3) $\sigma_3=(2,1,3)$, 这个排列有一个逆序对, $\operatorname{sgn}(\sigma_3)=-1$。

$$\prod_{i=1}^{3}a_{i,\sigma_3(i)}=b\times d\times i$$

结果为: $det(A)=aei+bfg+cdh-ceg-bdi-afh$。

其他三个不计算了, 反正就两点:
1) 排列有 k 个逆序对, 贡献就是 $(-1)^k$。

2) 如果 $\sigma = (1,2,3)$, $\prod_{i=1}^{3} a_{i,\sigma_1(i)}$ 对应乘的元素是 (1,1),(2,2),(3,3)。

这个计算 $det(A)$ 的时间复杂度是 $O(n! * n)$ 的。

4.1.2 行列式性质

1) 单位矩阵 I 的行列式为 1, 单位矩阵如下:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2) 交换矩阵的两行, 矩阵变号。
3) 若矩阵某一行乘以 k, 行列式也乘以 k。
4) 用矩阵的一行加上另一行的倍数, 行列式不变。
5) 有某两行一样的矩阵, 行列式是 0。
6) 矩阵运算满足下所示:

$$\begin{bmatrix} a+a' & b+b' \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} a' & b' \\ c & d \end{bmatrix}$$

7) 矩阵乘法满足结合律, 不满足交换律。

4.1.3 行列式四则运算

1) 矩阵加/减法 (需要两个行列数都相同的矩阵):

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

2) 矩阵乘矩阵 (满足第一个矩阵的列数与第二个矩阵的行数相同):
第一个矩阵 a 为 $x \times y$, 第二个矩阵 b 为 $y \times z$, 得到的矩阵 c 大小为 $x \times z$。
计算过程为: $c_{i,j} = \sum_{k=1}^{y} a_{i,k} \times b_{k,j}$。

3) 矩阵乘数字: 就乘就完事了。

4) 矩阵除法, 大概用不到就不写了。

4.1.4 消元求行列式

通过消元得到上三角矩阵 A, 其 $det(A)$ 为:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & 2 \end{bmatrix} = 2 \times 1 \times 2 = 4$$

这个过程我们可以通过 $O(n^2)$ 的高斯消元来实现, 这样就会快很多。详细代码见后面的高斯消元部分。

4.1.5 行列式的秩

矩阵的行阶梯形的非零行个数称为矩阵的秩, 记做 $r(A)$。反正如果一个矩阵的秩不是满的, 那么这个矩阵的行列式就为 0。

矩阵的行秩与列秩相等。

4.1.6 积和式

矩阵 A 的积和式记作 $pre(A)$, 直观的定义为 $pre(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$。

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

这个矩阵的积和式为: $pre(A) = aei + bfg + cdh + ceg + bdi + afh$。

有一个结论 $pre(A) \equiv det(A)(mod\ 2)$。

4.2 常见恒等式

1) 朱世杰恒等式: $\sum_{i=1}^{n} \binom{i}{j} = \binom{n+1}{j+1}$。

2) 组合数的递推式: $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$。

3) 组合数的定义式: $\binom{n}{m} = \frac{n!}{m! \times (n-m)!}$。

4.3 常见不等式

4.4 最大公约数

4.4.1 欧几里得算法

```cpp
// created on 2025-3-23

int gcd(int a, int b) {
    return b > 0 ? gcd(b, a % b) : a;
}
```

4.4.2 cpp 特有的奇奇怪怪的欧几里得算法

```cpp
// created on 24-8-23

int diff;

int gcd(int a,int b){
    int az=__builtin_ctz(a);
    int bz=__builtin_ctz(b);
    int z=(az>bz?bz:az,diff);
    b>>=bz;
    while (a) {
        a>>=az;
        diff=b-a;
        az=__builtin_ctz(diff);
        if (a<b) b=a;
```

```
        a=(diff<0?-diff:diff);
    }
    return b<<z;
}
```

4.4.3 扩展欧几里得算法

对于 $Ax + By = C$ 这个二元不定方程来说, 如果 C 不能被 $gcd(A, B)$ 整除那么这个玩意就没有整数解 (裴蜀定理)。

反之就存在, 拿下面这个东西乱搞反正能搞出一组特解 x_1。

x 的通解为: $x_1 + k * \frac{B}{gcd(A,B)}$。

```
// created on 23-8-24

int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
```

5 基础图论

5.1 几种经典距离

设 A 点坐标为 (x_1, y_1), B 点坐标为 (x_2, y_2)。

5.1.1 哈曼顿距离

A 与 B 的哈曼顿距离为 $|x_1 - x_2| + |y_1 - y_2|$, 或:

$max(\{x_1 - x_2 + y_1 - y_2, x_1 - x_2 + y_2 - y_1, x_2 - x_1 + y_1 - y_2, x_2 - x_1 + y_2 - y_1\})$

5.2 拓扑排序

```cpp
// created on 24-8-23
const int N = 1e5 + 10;
int din[N];
vector<int> e[N], tp;
bool toposort(int n){
    queue<int> q;
    for (int uu = 1; uu <= n; uu ++ ) {
        if (din[uu] == 0) q.push(uu);
    }
    while (q.size()){
        int x = q.front();
        q.pop();
        tp.push_back(x);
        for (auto y : e[x]){
            if (-- din[y] == 0) q.push(y);
        }
    }
    return tp.size() == n;
}
```

5.3 Dijkstra

$O(mlogm)$ 不支持负数权边。

```
// created on 24-8-23
const int N=2e5+10;
struct edge{
    int v, w;
};
struct node{
    ll dis, u;
    bool operator>(const node& a) const {
        return dis > a.dis;
    }
};
vector<edge> e[N];
ll dis[N], vis[N];
priority_queue<node,vector<node>,greater<node>> q;
inline void dijkstra(int n, int s){
    memset(dis, 63, sizeof(dis));
    dis[s] = 0;
    q.push({0, s});
    while (!q.empty()){
        int u = q.top().u;
        q.pop();
        if (vis[u]) continue;
        vis[u] = 1;
        for (auto ed : e[u]){
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w){
                dis[v] = dis[u] + w;
                q.push({dis[v], v});
```

```
                }
            }
        }
}
```

5.4 Bellman_Ford

$O(nm)$, 可以处理负边权, 如果返回 true 就是有负环。

```
// created on 24-8-23
const int N = 2e5 + 10;
struct edge{
    int v, w;
};
vector<edge> e[N];
int d[N];
bool bellmanford(int s, int n){
    memset(d, inf, sizeof d);
    d[s] = 0;
    bool flag;
    for (int uu = 1; uu <= n; uu ++ ){
        flag = false;
        for (int u = 1; u <= n; u ++ ){
            if (d[u] == inf) continue;
            for (auto ed : e[u]){
                int v = ed.v, w = ed.w;
                if (d[v] > d[u] + w){
                    d[v] = d[u] + w;
                    flag = true;
                }
            }
        }
```

```
            if (!flag) break;
        }
        return flag;
}
```

5.5 SPFA

其实就是优化之后的 BellmanFord。但是时常被卡成 Bellmanford 的时间复杂度。相同的如果返回 true 就是有负环。

```
// created on 24-8-23
const int N = 2e5+10;
struct edge{
        int v, w;
};
vector<edge> e[N];
int d[N], cnt[N], vis[N];
queue<int> q;
inline bool spfa(int s, int n){
        memset(d, inf, sizeof d);
        d[s] = 0, vis[s] = 1;
        q.push(s);
        while (q.size()){
            int u = q.front();
            q.pop();
            vis[u] = 0;
            for (auto ed : e[u]){
                int v = ed.v, w = ed.w;
                if (d[v] > d[u] + w){
                    d[v] = d[u] + w;
                    cnt[v] = cnt[u] + 1;
                    if (cnt[v] >= n) return true;
```

```
                if (!vis[v]) q.push(v), vis[v] = 1;
            }
        }
    }
    return false;
}
```

5.6 Floyd

时间复杂度为 $O(N^3)$ 的全源最短路。

```
// created on 24-8-23
const int N=500;
int d[N][N];
inline void floyd() {
    for (int k = 1; k <= n; k ++ ) {
        for (int i = 1; i <= n; i ++ ) {
            for (int j = 1; j <= n; j ++ ) {
                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
            }
        }
    }
}
```

5.7 Johnson

时间复杂度为 $O(nmlogm)$ 的全源最短路。

```
// created on 24-8-23
const int N=3e4+10;
int n,m;
ll h[N],d[N],vis[N],cnt[N],ans;
```

```
struct edge {
    int v,w;
};
vector<edge> e[N];
// spfa 做一个以虚构点为单源的最短路
inline void spfa() {
    queue<int> q;
    memset(h,63,sizeof h);
    memset(vis,false,sizeof vis);
    h[0]=0,vis[0]=1;
    q.push(0);
    while (SZ(q)) {
        int u=q.front();
        q.pop();
        vis[u]=0;
        for (auto ed:e[u]) {
            int v=ed.v,w=ed.w;
            if (h[v]>h[u]+w) {
                h[v]=h[u]+w;
                cnt[v]=cnt[u]+1;
                if (cnt[v]>n) {
                    cout<<"-1\n";
                    exit(0);
                }
                if (!vis[v]) q.push(v),vis[v]=1;
            }
        }
    }
}
// 接下来所有的边都是正的了
// 可以跑 n 次堆优化的 dijkstra 以求出全源头最短路
```

```
inline void dijkstra(int s) {
    priority_queue<pii> q;
    rep(uu,1,n+1) d[uu]=inf;
    memset(vis,0,sizeof vis);
    d[s]=0;
    q.push({0,s});
    while (SZ(q)) {
        int u=q.top().se;
        q.pop();
        if (vis[u]) continue;
        vis[u]=1;
        for (auto ed:e[u]) {
            int v=ed.v,w=ed.w;
            if (d[v]>d[u]+w) {
                d[v]=d[u]+w;
            if (!vis[v]) q.push({-d[v],v});
            }
        }
    }
}
signed main(){
    ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    cin>>n>>m;
    rep(uu,1,m+1) {
        int u,v,w;
        cin>>u>>v>>w;
        e[u].pb({v,w});
    }
    // 建立一条虚点，以及其他的虚边
    rep(uu,1,n+1) e[0].pb({uu,0});
    // 跑一边 spfa 求出
```

```
    spfa();
    // 重构边的权值，边权值变为 w+hu-hv
    // hu 表示上面spfa求出的u到虚点的最短路距离
    rep(uu,1,n+1) {
        for (auto &ed:e[uu]) {
            ed.w+=h[uu]-h[ed.v];
        }
    }
    rep(uu,1,n+1) {
        // 实际上这个操作就可以求出全源最短路了
        dijkstra(uu);
        ans=0;
        rep(vv,1,n+1) {
            if (d[vv]==inf) ans+=(ll)vv*inf;
            else ans+=(ll)vv*(d[vv]+h[vv]-h[uu]);
        }
        cout<<ans<<endl;
    }
}
```

5.8 Prim

5.9 kruskal

5.10 矩阵树定理

5.10.1 无权图

给出一个无向无权图, 设 A 为邻接矩阵, D 为度数矩阵 (D[i][i] 为 i 的度数, 其他无值), 则基尔霍夫矩阵为 $K = D - A$, 然后令 K' 为 K 去掉第 k 行和第 k 列的结果 (k 任意), $det(K')$ 即为所求。

5.10.2 有权图

设 G 是一个带权无向连通图, 边权为 $w_{i,j}$, 其基尔霍夫矩阵 K 定义为: 度数矩阵 D − 邻接矩阵 A。然后也要变换为 K' 并求 $det(K')$。

度数矩阵 D: D_{ii} 表示与顶点 i 相连的边的权值和。
邻接矩阵 A: $A_{ij} = w_{ij}$ 仅当 i 与 j 之间有边。

6 基础树论

7 计算几何

8 动态规划模版

9 博弈论

大概下面所有的 k 都代表自然数。

9.1 Bash 博弈

有一堆 n 个物品, 两个人轮流从这里面取东西, 每次最少取一个东西, 最多取 m 个物品。

1) 最后取光者胜, 当 $n\%(m+1) == 0$ 时, 后手必胜, 否则先手必胜。

2) 最后取光者输, 当 $n = (m+1) * k + 1$ 时, 后手必胜, 否则先手必胜。

9.2 EX Bash 博弈

有一堆 n 个物品, 两个人轮流从这里面取东西, 每人可以取走 x 个石子 $(a \leq x \leq b)$。若最后剩余物品的个数小于 a 个, 则不能再取。拿到最后一颗石头者获胜。

1) $n = k * (a + b)$ 时, 后手必胜。

2) $n = k * (a + b) + R_1$ $(0 \leq R_1 < a)$ 时, 后手必胜。

3) $n = k * (a + b) + R_2$ $(a \leq R_2 \leq b)$ 时, 先手必胜。

4) $n = k * (a + b) + R_3$ $(b < R_3 < a + b)$ 时, 先手必胜。

9.3 Nim 博弈

有 n 堆石头, 给出每一堆的石头数量, 两名玩家轮流行动, 每人每次任选一堆, 拿走正整数颗石头, 拿到最后一颗石头的人获胜。几个特点: 不能跨堆拿, 不能不拿石子。

记初始情况下各堆石子的数量 $(A_1, A_2, A_3, ..., A_n)$, 定义尼姆和为 $Sum_N = A_1 \bigoplus A_2 \bigoplus A_3 \bigoplus ... \bigoplus A_n$。当 $Sum_N = 0$ 时先手必败, 反之先手必胜。

胜利的策略就是在取走棋子后, 使尼姆和为 0。只要取走棋子前, 尼姆和不为 0, 一定有办法取走部分棋子使尼姆和为 0。另一个游戏者无论怎么拿, 取走棋子后尼姆和都不会为 0。以此策略，只要在取棋子时照策略进行，一定会胜利。

具体取法为先计算尼姆和, 在对每一堆石子计算 $A_i \bigoplus Sum_N$, 记为 X_i。若得到的值 $X_i < A_i$, X_i 即为一个可行解, 即剩下 X_i 颗石头, 取走 $A_i - X_i$ 颗石头。(这里是小于号因为至少要取走一颗石头)

10 杂项算法和一些小工具

10.1 CPU Checker

```cpp
// created on 2025-03-16

#include <stdint.h>
#include <iostream>
#include <cpuid.h>
#define u32 uint32_t
static void cpuid(u32 func,u32 sub,u32 data[4]) {
    __cpuid_count(func,sub,
    data[0],data[1],data[2],data[3]);
}
int main() {
    u32 data[4];
    char str[48];
    for (int i=0;i<3;++i) {
        cpuid(0x80000002+i,0,data);
        for (int j=0;j<4;++j) {
            reinterpret_cast<u32*>(str)[i*4+j]=data[j];
        }
    }
    std::cout << str;
}
```