质数

- 质数的判定

> **试除法O(sqrt(n)):**
>
> ```
> bool check(int x) {
>     if (x < 2) return false;
>     for (int i = 2; i <= x / i; i ++ )
>         if (x % i == 0) return false;
>     return true;
> }
> ```

- 分解质因数

> **试除法O(logn~sqrt(n))**
>
> ```
> void divi(int x) {
>     for (int i = 2; i <= x / i; i ++ ) {
>         if (x % i == 0) {
>             int s = 0;
>             while (x % i == 0) {
>                 s ++ ;
>                 x /= i;
>             }
>             cout << i << ' ' << s << endl;
>         }
>     }
>     if (x > 1) cout << x << ' ' << 1 << endl;
>     cout << endl;
> }
> ```

- 筛质数

> **埃氏筛O(nloglogn)**

```
void get_prime(int n) {
    for (int i = 2; i <= n; i ++ ) {
        if (!st[i]) prime[cnt ++ ] = i;
        for (int j = i + i; j <= n; j += i) st[j] = true;
    }
}
```

> **线性筛O(n)**
>
> 在10^7后比埃氏筛快一倍左右
>
> ```
> void get_prime(int n) {
>     for (int i = 2; i <= n; i ++ ) {
>         if (!st[i]) prime[cnt ++ ] = i;
>         for (int j = 0; prime[j] <= n / i; j ++ ) {
>             st[prime[j] * i] = true;
>             if (i % prime[j] == 0) break;
>         }
>     }
> }
> ```

约数

- 求约数

> **试除法**

```
void get_divi(int n) {
    vector<int> ans;

    for (int i = 1; i <= n / i; i ++ )
        if (n % i == 0) {
            ans.emplace_back(i);
            if (i != n / i) ans.emplace_back(n / i);
        }
    sort(ans.begin(), ans.end());
    for (auto& v: ans) cout << v << ' ';
    cout << endl;
}
```

- 约数个数

> **因式分解**
>
> $(a_1+1)\cdot(a_2+1)\cdot\ldots\ldots\cdot(a_k+1)$
>
> $N = P_1^{a_1}\cdot P_2^{a_2}\cdot\ldots\ldots\cdot P_k^{a_k}$
>
> $d = P_1^{b_1}\cdot P_2^{b_2}\cdot\ldots\ldots\cdot P_k^{b_k}$ (d 为 N 的一个约数)
>
> $0 \le b_i \le a_i$

```
unordered_map<int, int> primes;
while (n -- ) {
  int x;
  cin >> x;

  for (int i = 2; i <= x / i; i ++ )
    while (x % i == 0) {
      x /= i;
      primes[i] ++ ;
    }
  if (x > 1) primes[x] ++ ;
}
LL res = 1;
for (auto& v : primes) res = res * (v.second + 1) % p;
```

```
cout << res << endl;
```

- 约数之和

> $(P_1^0 + P_1^1 + \ldots + P_1^{a_1}) \cdot (P_2^0 + P_2^1 + \ldots + P_2^{a_2}) \cdot \ldots \cdot (P_k^0 + P_k^1 + \ldots + P_k^{a_k})$
>
> 乘法分配律展开
>
> ```
> unordered_map<int, int> primes;
> while (n -- ) {
>   int x;
>   cin >> x;
>
>   for (int i = 2; i <= x / i; i ++ )
>     while (x % i == 0) {
>       x /= i;
>       primes[i] ++ ;
>     }
>   if (x > 1) primes[x] ++ ;
> }
> LL res = 1;
> for (auto& v : primes) {
>   LL p = v.first, a = v.second;
>   LL t = 1;
>   while (a -- ) t = (p * t + 1) % mod;
>   res = res * t % mod;
> }
> cout << res << endl;
> ```

- 最大公约数

> **欧几里得算法（辗转相除法）**
>
> 最大公约数(a, b) = 最大公约数(b, a mod b)
>
> a mod b = a - a // b * b = a - c * b

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

欧拉函数

欧拉函数的定义

> 1— N 中与 N 互质的数的个数被称为欧拉函数，记为 $\phi(N)$
>
> 若在算数基本定理中，$N = P_1^{a_1} \cdot P_2^{a_2} \cdot \ldots \ldots \cdot P_k^{a_k}$
>
> $\phi(N) = N \times (1 - \frac{1}{p_1}) \times (1 - \frac{1}{p_2}) \times \ldots \times (1 - \frac{1}{p_k})$

> **容斥原理**
>
> 1. 从 1 ~ N 中去掉 $p_1, p_2, \ldots, p_k$ 的所有倍数
> $N - \frac{N}{p_1} - \frac{N}{p_2} - \ldots - \frac{N}{p_k}$
> 2. 加上所有 $p_i \times p_j$ 的倍数
> 3. 减去所有 $p_i \times p_j \times p_k$
> 4. ...

> $O(sqrt(n))$

```cpp
int euler(int x) {
    int cnt = x;
    for (int i = 2; i <= x / i; i ++ ) {
        if (x % i == 0) {
            cnt -= cnt / i;
            while (x % i == 0) x /= i;
        }
    }
    if (x > 1) cnt -= cnt / x;
    return cnt;
}
```

线性筛求欧拉函数$O(n)$

```cpp
void euler(int x) {
    phi[1] = 1; //  **
    for (int i = 2; i <= n; i ++ ) {
        if (!st[i]) {
            prime[cnt ++ ] = i;
            phi[i] = i - 1; //  **
        }
        for (int j = 0; prime[j] <= x / i; j ++ ) {
            st[i * prime[j]] = true;
            if (i % prime[j] == 0) { //  最小质因子
                phi[i * prime[j]] = phi[i] * prime[j]; //  **
                break;
            }
            phi[i * prime[j]] = phi[i] * (prime[j] - 1);
        }
    }
    LL res = 0; //  **
    for (int i = 1; i <= n; i ++ ) res += phi[i];
    cout << res << endl;

}
```

快速幂(欧拉降幂)

> $a^k mod p$
>
> $a^k = a^{2^{x_1}} \cdot a^{2^{x_2}} \cdot \ldots \cdot a^{2^{x_t}}$
>
> $k = 2^{x_1} + 2^{x_2} + \ldots + 2^{x_t}$
>
> $O(log_k)$
>
> ```
> LL qmi(int a, int b, int p) {
>     LL res = 1;
>     while (b) {
>         if (b & 1) res = 1LL * res * a % p;
>         a = 1LL * a * a % p;
>         b >>= 1;
>     }
>     return res;
> }
> ```
>
> - 求乘法逆元
>
>   乘法逆元的定义
>
>   > 若整数 b，m 互质，并且对于任意的整数 a，如果满足 b|a，则存在一个整数 x，使得 a/b≡a×x(mod m)，则称 x 为 b 的模 m 乘法逆元，记为 b−1(mod m)。
>   >
>   > b 存在乘法逆元的充要条件是 b 与模数 m 互质。当模数 m 为质数时，b^m−2即为 b 的乘法逆元。
>
>   ```
>   #include <iostream>
>
>   using namespace std;
>
>   typedef long long LL;
>
>   ```

```
int n;

LL qmi(int a, int b, int p) {
    LL res = 1;
    while (b) {
        if (b & 1) res = 1LL * res * a % p;
        a = 1LL * a * a % p;
        b >>= 1;
    }
    return res;
}

int main(){
    cin >> n;
    while (n -- ) {
        int a, p;
        cin >> a >> p;
        if (a % p) cout << qmi(a, p - 2, p) << endl;
        else cout << "impossible" << endl;
    }
    return 0;
}
```

线性同余方程

- 求出 x 满足 $a * x \equiv b \pmod{m}$

```
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;
}
```

```
void solve() {
    cin >> n;
    while (n -- ) {
        int a, b, m, x, y;
        cin >> a >> b >> m;
        int d = exgcd(a, m, x, y);
        if (b % d) cout << "impossible" << endl;
        else cout << (LL)x * (b / d) % m << endl;
    }
}
```

扩展欧几里得

> 求出一组x,y，使得a * x + b * y == gcd(a, b)
>
> ```
> void exgcd(int a, int b, int &x, int &y) {
>     if(!b) {
>         x = 1, y = 0;
>         return ;
>     }
>     exgcd(b, a % b, y, x);
>     y = y - a / b * x;
> }
> ```

- 线性同余方程

  > 求一个x，满足a * x == b(mod m)

思路： 通过扩展欧几里得算法的思想，将题目意思转换为a * x+m * y==gcd(a, m)求x，同时如果gcd无法被b整除，则说明无解。

```
#include<iostream>
#include<algorithm>


using namespace std;
```

```
typedef long long ll;

ll exgcd(ll a, ll b, ll &x, ll &y)
{
    if(!b)
    {
        x = 1, y = 0; // 边界，（或理解为递归的重点）
        return a;
    }
    int d = exgcd(b, a % b, y, x);// y与b对应，x与a对应
    y -= a / b * x;// 公式推导
    return d;
}

int main()
{
    int n;
    cin >> n;
    while(n -- )
    {
        ll a, b, m;
        cin >> a >> b >> m;
        ll x, y;
        ll t = exgcd(a, m, x, y);
        if(b % t) puts("impossible");
        else cout << x * (b / t) % m << endl;
    }
    return 0;
}
```

求组合数

> 给定a,b求C_a^b

公式：$C_a^b = C_{a-1}^{b-1} * C_{a-1}^{b}$

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 2010, mod = 1e9 + 7;

int c[N][N];

void init() {
    for(int i = 0; i < N; i++) {
        for(int j = 0; j <= i; j++) {
            if(!j) c[i][j] = 1;
            else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % mod;
        }
    }
}

int main()
{
    int n;
    init();

    cin >> n;
    while(n--) {
        int a, b;
        cin >> a >> b;
        cout << c[a][b] << endl;
    }
    return 0;
}
```

> 大数

思路： 快速幂求逆元

```

#include<iostream>
```

```cpp
#include<algorithm>

using namespace std;

typedef long long ll;

const int N = 1e5 + 10, mod = 1e9 + 7;

ll fact[N], infact[N]; // infact表示除数的逆元

ll qmi(ll a, ll k, ll p)
{
    ll res = 1;
    while(k)
    {
        if(k & 1) res = res * a % p;
        a = a * a % p;
        k >>= 1;
    }
    return res;
}

int main(){
    fact[0] = infact[0] = 1;
    for(int i = 1; i < N; i++)
    {
        fact[i] = fact[i - 1] * i % mod;
        infact[i] = infact[i - 1] * qmi(i, mod - 2, mod) % mod;
    }

    int n;
    cin >> n;
    while(n--)
    {
        ll a, b;

        cin >> a >> b;
```

```
        cout << fact[a] * infact[b] % mod * infact[a - b] % mod << endl;
    }
    return 0;
}
```

树状数组

```
int tr[N];

int lowbit(int x) { // 从后往前返回二进制中的第一个1
    return x & -x;
}

void add(int x, int k) { // 单点修改
    for(int i = x; i <= n; i += lowbit(i)) tr[i] += k;
}

int query(int x) { // 区间查询
    int res = 0;
    for(int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}
```

线段树

```
int w[N];

struct Node{
    int l, r;
    LL add, sum;
}tr[N * 4];

void push_up(int u)
{
```

```
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

void push_down(int u)
{
    if (tr[u].add == 0) return;
    tr[u << 1].add += tr[u].add;
    tr[u << 1 | 1].add += tr[u].add;
    tr[u << 1].sum += tr[u].add * (tr[u << 1].r - tr[u << 1].l +
1);
    tr[u << 1 | 1].sum += tr[u].add * (tr[u << 1 | 1].r - tr[u <<
1 | 1].l + 1);
    tr[u].add = 0;
}

void build(int u, int l, int r)
{
    tr[u] = {l, r, 0};
    if (l == r) tr[u].sum = w[r];
    else {
        int mid = l + r >> 1;
        build(u << 1, l, mid);
        build(u << 1 | 1, mid + 1, r);
        push_up(u);
    }
}

void modify(int u, int l, int r, int v)
{
    if (l <= tr[u].l && tr[u].r <= r) {
        tr[u].sum += v * (tr[u].r - tr[u].l + 1);
        tr[u].add += v;
    }else {
        push_down(u);
        int mid = tr[u].l + tr[u].r >> 1;

        if (l <= mid) modify(u << 1, l, r, v);
```

```
        if (mid < r ) modify(u << 1 | 1, l, r, v);
        push_up(u);
    }
}

LL query(int u, int l, int r)
{
    if (l <= tr[u].l && tr[u].r <= r) return tr[u].sum;
    push_down(u);
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum = 0;
    if (l <= mid) sum = query(u << 1, l, r);
    if (mid <  r) sum += query(u << 1 | 1, l, r);
    return sum;
}
```

dijkstra

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include<queue>
#include<vector>

using namespace std;

typedef pair<int,int> PII;

const int N = 150010;

int n,m;
int dist[N];
int h[N],e[N],w[N],ne[N],idx; //  稀疏图用邻接表
bool st[N];
```

```
void add(int a,int b,int c){ //  邻接表添边的模板
    e[idx] = b, w[idx] = c, ne[idx] = h[a] , h[a] = idx++;
}

int dijkstra(){
    memset(dist, 0x3f,sizeof dist);
    dist[1] = 0;
    priority_queue<PII,vector<PII>, greater<PII>> heap; //  优先队列\小根堆
    heap.push({0,1}); //  first 为距离, second 为点

    while(heap.size()){
        auto t = heap.top();
        heap.pop();

        int distance = t.first, ver = t.second ;
        if(st[ver]) continue; //  如果当前点属于已确定最小距离的点，说明当前点是冗余备份，没有必要再处理，直接 continue
        st[ver] = true; //  将当前点标记为已确定最小距离的点
        for(int i = h[ver]; i != -1; i = ne[i]){ //  邻接表遍历当前点能走到的所有点，并更新他们的最小距离
            int j = e[i];
            if(dist[j] > distance + w[i]){
                dist[j] = distance + w[i];
                heap.push({dist[j], j}); //  放入待处理中
            }
        }
    }
    if(dist[n] == 0x3f3f3f3f)return -1; //  如果没法从 1 走到 n , dist[n] 就不会被更新，初始值为 0x3f3f3f3f
    return dist[n];
}

int main(){
    cin >> n >> m;

    memset(h, -1, sizeof h); //  一定记得初始化头结点，很容易忘
```

```
    while(m--){
        int a,b,c;
        cin >> a >> b >> c;
        add(a,b,c);
    }
    cout << dijkstra() << endl;

    return 0;
}
```

Bellman-Ford

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;

const int N = 510, M = 100010;

struct Eg{
    int a,b,w;
}egs[M];

int n,m,k;
int dist[N], backup[N];

void bellman_ford(){
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for(int i = 0 ; i < k ; i ++){
        // 把dist复制到backup经行遍历松弛,不会产生串联
        memcpy(backup,dist,sizeof dist);

        for(int j = 0 ; j < m ; j++){
```

```
            int a = egs[j].a, b = egs[j].b, w = egs[j].w;
            //  (dist(a) +w(ab)) < dist(b)则说明存在到 b 的更短的路径,取最小
            dist[b] = min(dist[b], backup[a] + w);
        }
    }
}

int main(){
    cin >> n >> m >> k;
    for(int i = 0 ; i < m ; i++){
        int a,b,c;
        cin >> a >> b >> c;
        egs[i] = {a,b,c};
    }

    bellman_ford();

    if(dist[n] > 0x3f3f3f3f / 2)cout << "impossible" << endl;
    else cout << dist[n] << endl;

    return 0;
}
```

SPFA

```
#include <queue>
#include <iostream>
#include <cstring>

using namespace std;

const int N = 100010;



int n, m;
```

```
int h[N], e[N], ne[N], w[N], idx;
int d[N];
bool st[N]; //  用于标记点是否已在队列中

void add(int a, int b, int c) {
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
}

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    while (m -- ) {
        int a, b, c; cin >> a >> b >> c;
        add(a, b, c);
    }

    queue<int> q;
    q.emplace(1); //  将源点放入队列中
    memset(d, 0x3f, sizeof d);
    d[1] = 0;

    while (q.size()) {
        auto t = q.front();
        q.pop();

        st[t] = false; //  表示该点已经不在队列中

        for (int i = h[t]; i != -1; i = ne[i]) { // 枚举被更新过的点的出边，更新到它出边的点的最短距离
            int j = e[i];
            if (d[j] > d[t] + w[i]) { //  如果出边的点能被更新，那么再判断是否在队列中并且插入队列
                d[j] = d[t] + w[i];
                if (!st[j]) {
                    st[j] = true;

                    q.emplace(j);
```

```
                }
            }
        }
    }
    if (d[n] == 0x3f3f3f3f) cout << "impossible" << endl;
    else cout << d[n] << endl;

    return 0;
}
```

Floyd

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 210, INF = 1e9;

int n, m, q;
int d[N][N]; //  用邻接矩阵存储边

int main(){
    cin >> n >> m >> q;

    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ ) //  初始化邻接矩阵，由于可能存在负权边，但不存在负权回路，因此自环和重边的情况只要这么判断就可以了
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

    while (m -- ) {
        int a, b, c;
        cin >> a >> b >> c;

        d[a][b] = min(d[a][b], c);
```

```
    }

    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
            {
                //  由于负权边的存在，可能被更新丞一个略小于这个额被当做无穷大的数，导致无穷大被更新为一个数
                if (d[i][k] == INF || d[k][j] == INF) continue; // 这句话可以防止无穷大被更新
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
    while (q -- ) {
        int a, b; cin >> a >> b;
        if (d[a][b] == INF) cout << "impossible" << endl;
        else cout << d[a][b] << endl;
    }

    return 0;
}
```

Prim

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 510;

int n, m;
int w[N][N];
int d[N];
bool st[N];

```

```
int main(){
    cin >> n >> m;
    memset(w, 0x3f, sizeof w);
    for (int i = 0; i < m; i ++ ) {
        int a, b, c;
        cin >> a >> b >> c;
        w[a][b] = w[b][a] = min(w[a][b], c);
    }
    memset(d, 0x3f, sizeof d);
    int res = 0, cnt = 0;
    for (int i = 0; i < n; i ++ ) {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if ((t == -1 || d[t] > d[j]) && !st[j]) t = j;
        if (i && d[t] == 0x3f3f3f3f) break;
        if (i) {
            res += d[t];
            cnt ++ ;
        }
        st[t] = true;
        for (int j = 1; j <= n; j ++ ) d[j] = min(d[j], w[j][t]);
    }
    if (cnt >= n - 1) cout << res << endl;
    else cout << "impossible" << endl;

    return 0;
}
```

Kruskal

```
#include <iostream>
#include <algorithm>

using namespace std;
```

```
const int N = 100010, M = 200010;

int n, m;
int p[N];

struct Node{
    int a, b, w;

    bool operator<(const Node& W) const{
        return w < W.w;
    }
}edges[M];

int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int main(){
    cin >> n >> m;
    for (int i = 0; i < m; i ++ ) {
        int a, b, c;
        cin >> a >> b >> c;
        edges[i] = {a, b, c}; //  用结构体存储每条边
    }

    sort(edges, edges + m); // 按照权重从小到大排序

    for (int i = 1; i <= n; i ++ ) p[i] = i; //  并查集初始化祖宗节
点

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ ) { //  枚举每条边
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;
        a = find(a), b = find(b);

        if (a != b) { //  不连通
```

```
            p[a] = b; // 将该边加入集合中
            res += w; //  边的权重累加到最小生成树答案中
            cnt ++ ; //  集合中边总数 + 1
        }
    }

    if (cnt >= n - 1) cout << res << endl;
    else cout << "impossible" << endl;

    return 0;
}
```