补充模板

最大流（Dinic算法 - 精简版）

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll inf = 1e11;
ll node[4007], edge[4007], nxt[4007], weight[4007], d[4007], cur[4007];
ll n, m, st, ed, cnt = 0;

void addEdge(ll f, ll t, ll w){
    nxt[cnt] = node[f];
    edge[cnt] = t;
    node[f] = cnt;
    weight[cnt] = w;
    cnt++;
}

// BFS构建分层图
bool bfs() {
    memset(d, -1, sizeof(d));
    queue<ll> q;
    q.push(st);
    d[st] = 0;
    while (!q.empty()) {
        ll u = q.front(); q.pop();
        for (ll e = node[u]; e != -1; e = nxt[e]) {
            ll v = edge[e];
            if (d[v] == -1 && weight[e] > 0) {
                d[v] = d[u] + 1;
                q.push(v);
                if (v == ed) return true;
            }
        }
    }
    return false;
}

// DFS多路增广
ll dfs(ll u, ll flow) {
    if (u == ed) return flow;
    ll sum = 0;
    for (ll &e = cur[u]; e != -1; e = nxt[e]) {  // 当前弧优化
        ll v = edge[e];
        if (d[v] == d[u] + 1 && weight[e] > 0) {
            ll f = dfs(v, min(flow, weight[e]));
            weight[e] -= f;
            weight[e ^ 1] += f;
            sum += f;
```

```
            flow -= f;
            if (flow == 0) break;
        }
    }
    if (sum == 0) d[u] = -1;  // 优化：标记无法到达汇点
    return sum;
}

ll dinic(){
    ll flow = 0;
    while (bfs()){
        memcpy(cur, node, sizeof(cur));  // 重置当前弧
        flow += dfs(st, inf);
    }
    return flow;
}

int main(){
    memset(node, -1, sizeof(node));
    cin >> n >> m >> st >> ed;
    for (int i = 0; i < m; i++){
        ll u, v, w;
        cin >> u >> v >> w;
        addEdge(u, v, w);
        addEdge(v, u, 0);  // 反向边
    }
    cout << dinic() << endl;
    return 0;
}
```

优化说明：

- 移除了不必要的变量（h, mf, visit, a, b, we）
- 移除了mincut函数（如需要可单独添加）
- 简化了main函数，直接输入st和ed
- 保留了核心功能：BFS分层 + DFS多路增广 + 当前弧优化

FFT（多项式类封装版）

```
#include <bits/stdc++.h>
using namespace std;

// 紧凑版复数类
struct Complex {
    double x, y;
    Complex(double x=0, double y=0):x(x),y(y){}
    Complex operator+(const Complex& b) const { return {x+b.x, y+b.y}; }
    Complex operator-(const Complex& b) const { return {x-b.x, y-b.y}; }
    Complex operator*(const Complex& b) const { return {x*b.x-y*b.y,
x*b.y+y*b.x}; }
```

```
};

// FFT 核心
void fft(vector<Complex>& a, int inv) {
    int n = a.size();
    // 位逆序置换
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    // 迭代FFT
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * M_PI / len * inv;
        Complex wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            Complex w(1);
            for (int j = 0; j < len / 2; j++) {
                Complex u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v, a[i+j+len/2] = u - v, w = w * wlen;
            }
        }
    }
    if (inv == -1) for (auto& x : a) x.x /= n;
}

// 多项式类封装
struct Poly : vector<long long> {
    using vector<long long>::vector; // 继承构造函数

    // 加法重载: a = a + b
    Poly operator+(const Poly& b) const {
        Poly res = *this;
        if (res.size() < b.size()) res.resize(b.size());
        for (int i = 0; i < b.size(); i++) res[i] += b[i];
        return res;
    }
    void operator+=(const Poly& b) { *this = *this + b; }

    // 乘法重载 (FFT): a = a * b
    Poly operator*(const Poly& b) const {
        int n = size(), m = b.size();
        int L = 1; while (L < n + m) L <<= 1;
        vector<Complex> fa(L), fb(L);
        for (int i = 0; i < n; i++) fa[i].x = (*this)[i];
        for (int i = 0; i < m; i++) fb[i].x = b[i];

        fft(fa, 1); fft(fb, 1);
        for (int i = 0; i < L; i++) fa[i] = fa[i] * fb[i];
        fft(fa, -1);

        Poly res(n + m - 1);
        for (int i = 0; i < n + m - 1; i++) res[i] = round(fa[i].x);
```

```
        return res;
    }
};

int main() {
    int n, m;
    cin >> n >> m;
    Poly A(n + 1), B(m + 1);
    for (int i = 0; i <= n; i++) cin >> A[i];
    for (int i = 0; i <= m; i++) cin >> B[i];

    // 像基本类型一样使用
    Poly C = A * B;
    Poly D = A + A; // 示例加法

    for (long long x : C) cout << x << " ";
    return 0;
}
```

特点：

- 使用多项式类封装，代码更简洁
- 支持运算符重载，使用更直观
- FFT实现紧凑高效
- 支持多项式的加法和乘法运算

使用说明

Dinic算法

- **复杂度**：O(n²m)，实际运行很快
- **适用场景**：最大流问题
- **关键优化**：当前弧优化、分层图
- **注意**：建图时记得添加反向边，初始容量为0

FFT多项式

- **复杂度**：O(n log n)
- **适用场景**：多项式乘法、大数乘法
- **特点**：封装成类，使用方便
- **注意**：结果需要四舍五入，注意精度问题