# 算法模板完整版

> 包含所有算法模板的完整代码，适用于算法考试

## 📋 目录



---

## 数据结构

### 快速排序

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

int arr[10007];

// 分区函数：将数组分为小于等于和大于基准的两部分
int partition1(int l, int r, int m){
    if (l >= r) return 0;
    int index = l;
    int xindex = 0;  // 基准元素的位置
    int a = l;
    for (index = l; index <= r; index++){
        if (arr[index] <= arr[m]){
            if (arr[index] == arr[m]){
                xindex = a;  // 记录基准元素位置
            }
            swap(arr[index], arr[a]);
            a++;
        }
    }
    swap(arr[xindex], arr[a - 1]);  // 将基准放到正确位置
    return a - 1;
}

// 快速排序主函数
int quicksort1(int l, int r){
    if (l >= r) return 0;
    int x = l + random() % (r - l + 1);  // 随机选择基准
    int neopos = partition1(l, r, x);
```

```
    quicksort1(l, neopos - 1);
    quicksort1(neopos + 1, r);
    return 0;
}

int main(){
    int num;
    cin >> num;
    for (int i = 0; i < num; i++){
        cin >> arr[i];
    }
    quicksort1(0, num - 1);
    for (int i = 0; i < num; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

## 并查集

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

int* arr;

// 初始化：每个元素都是自己的父节点
void build(int n){
    for (int i = 0; i < n; i++){
        arr[i] = i;
    }
}

// 查找：路径压缩优化
int find(int x){
    if (arr[x] != x){
        arr[x] = find(arr[x]);  // 路径压缩
    }
    return arr[x];
}

// 合并：将两个集合合并
void merge(int a, int b){
    int fa = find(a);
    int fb = find(b);
    if (fa != fb){
        arr[fa] = fb;
    }
}

// 判断是否在同一集合
```

```
bool same(int a, int b){
    int fa = find(a);
    int fb = find(b);
    return fa == fb;
}

int main(){
    int n, m;
    cin >> n >> m;
    arr = new int[n];
    build(n);
    int op, a, b;
    for (int i = 0; i < m; i++){
        cin >> op >> a >> b;
        if (op == 1){
            merge(a - 1, b - 1);
        }
        else{
            if (same(a - 1, b - 1)){
                cout << "Y" << endl;
            }
            else{
                cout << "N" << endl;
            }
        }
    }
    delete[] arr;
    return 0;
}
```

## 线段树（区间修改查询）

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 100007;
ll arr[maxn], tree[maxn << 2], lazy[maxn << 2];

// 向上更新：合并左右子树信息
void pushup(int rt){
    tree[rt] = tree[rt << 1] + tree[rt << 1 | 1];
}

// 向下传递懒标记
void pushdown(int rt, int l, int r){
    if (lazy[rt]){
        int mid = (l + r) >> 1;
        lazy[rt << 1] += lazy[rt];
        lazy[rt << 1 | 1] += lazy[rt];
        tree[rt << 1] += lazy[rt] * (mid - l + 1);
        tree[rt << 1 | 1] += lazy[rt] * (r - mid);
```

```
        lazy[rt] = 0;
    }
}

// 建树
void build(int rt, int l, int r){
    if (l == r){
        tree[rt] = arr[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(rt << 1, l, mid);
    build(rt << 1 | 1, mid + 1, r);
    pushup(rt);
}

// 区间修改
void update(int rt, int l, int r, int L, int R, ll val){
    if (L <= l && r <= R){
        tree[rt] += val * (r - l + 1);
        lazy[rt] += val;
        return;
    }
    pushdown(rt, l, r);
    int mid = (l + r) >> 1;
    if (L <= mid) update(rt << 1, l, mid, L, R, val);
    if (R > mid) update(rt << 1 | 1, mid + 1, r, L, R, val);
    pushup(rt);
}

// 区间查询
ll query(int rt, int l, int r, int L, int R){
    if (L <= l && r <= R){
        return tree[rt];
    }
    pushdown(rt, l, r);
    int mid = (l + r) >> 1;
    ll sum = 0;
    if (L <= mid) sum += query(rt << 1, l, mid, L, R);
    if (R > mid) sum += query(rt << 1 | 1, mid + 1, r, L, R);
    return sum;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> arr[i];
    }
    build(1, 1, n);
    while (m--){
        int op, l, r;
```

```
        ll val;
        cin >> op;
        if (op == 1){
            cin >> l >> r >> val;
            update(1, 1, n, l, r, val);
        }
        else {
            cin >> l >> r;
            cout << query(1, 1, n, l, r) << endl;
        }
    }
    return 0;
}
```

## 树状数组

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 100007;
ll tree[maxn];
int n;

// 获取最低位的1
int lowbit(int x){
    return x & -x;
}

// 单点修改
void add(int x, ll val){
    while (x <= n){
        tree[x] += val;
        x += lowbit(x);
    }
}

// 前缀和查询
ll query(int x){
    ll sum = 0;
    while (x > 0){
        sum += tree[x];
        x -= lowbit(x);
    }
    return sum;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int m;
    cin >> n >> m;
```

```
    for (int i = 1; i <= n; i++){
        ll val;
        cin >> val;
        add(i, val);
    }
    while (m--){
        int op, l, r;
        ll val;
        cin >> op;
        if (op == 1){
            cin >> l >> val;
            add(l, val);
        }
        else {
            cin >> l >> r;
            cout << query(r) - query(l - 1) << endl;
        }
    }
    return 0;
}
```

## ST表（静态区间最值）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 100007;
int st[maxn][20], arr[maxn], lg[maxn];
int n, m;

// 初始化ST表
void init(){
    lg[1] = 0;
    for (int i = 2; i <= n; i++){
        lg[i] = lg[i >> 1] + 1;  // 预处理log2
    }
    for (int i = 1; i <= n; i++){
        st[i][0] = arr[i];  // 长度为1的区间
    }
    for (int j = 1; j <= lg[n]; j++){
        for (int i = 1; i + (1 << j) - 1 <= n; i++){
            st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
        }
    }
}

// 查询区间最值
int query(int l, int r){
    int k = lg[r - l + 1];
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
```

```
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> arr[i];
    }
    init();
    while (m--){
        int l, r;
        cin >> l >> r;
        cout << query(l, r) << endl;
    }
    return 0;
}
```

---

## 动态规划

### 01背包

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1007;
int dp[maxn], w[maxn], v[maxn];  // w:重量, v:价值

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;  // n:物品数, m:背包容量
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> w[i] >> v[i];
    }
    // dp[j]: 容量为j的背包能装的最大价值
    for (int i = 1; i <= n; i++){
        for (int j = m; j >= w[i]; j--){  // 倒序枚举，保证每个物品只用一次
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

### 完全背包

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1007;
```

```
int dp[maxn], w[maxn], v[maxn];  // w:重量, v:价值

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;  // n:物品数, m:背包容量
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> w[i] >> v[i];
    }
    // dp[j]: 容量为j的背包能装的最大价值
    for (int i = 1; i <= n; i++){
        for (int j = w[i]; j <= m; j++){  // 正序枚举, 允许重复使用
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

## 混合背包

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1007;
int dp[maxn], w[maxn], v[maxn], s[maxn];  // s:数量限制, -1表示无限

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> w[i] >> v[i] >> s[i];
    }
    for (int i = 1; i <= n; i++){
        if (s[i] == -1){
            // 完全背包: 正序
            for (int j = w[i]; j <= m; j++){
                dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
            }
        }
        else if (s[i] == 1){
            // 01背包: 倒序
            for (int j = m; j >= w[i]; j--){
                dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
            }
        }
        else{
            // 多重背包: 二进制优化
            int k = 1;
```

```
            while (k < s[i]){
                for (int j = m; j >= k * w[i]; j--){
                    dp[j] = max(dp[j], dp[j - k * w[i]] + k * v[i]);
                }
                s[i] -= k;
                k <<= 1;
            }
            for (int j = m; j >= s[i] * w[i]; j--){
                dp[j] = max(dp[j], dp[j - s[i] * w[i]] + s[i] * v[i]);
            }
        }
    }
    cout << dp[m] << endl;
    return 0;
}
```

LIS - DP版本（O(n²)）

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

int dp[1007];
int arr[1007];
int ans = 0;

int main(){
    int n;
    cin >> n;
    for (int i = 0; i < n; i++){
        cin >> arr[i];
    }
    // dp[i]: 以arr[i]结尾的最长上升子序列长度
    for (int i = 0; i < n; i++){
        dp[i] = 1;
        for (int j = 0; j < i; j++){
            if (arr[j] < arr[i]){  // 上升子序列
                dp[i] = max(dp[j] + 1, dp[i]);
            }
        }
        ans = max(ans, dp[i]);
    }
    cout << ans << endl;
    return 0;
}
```

LIS - 贪心版本（O(n log n)）

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

```cpp
typedef long long ll;
int cnt = 0, increase_cnt = 0, decrease_cnt = 0;
ll increase[100007], decrease[100007], arr[100007];

// 二分查找：找到第一个大于等于目标的位置
int bisearch_le(int l, int r, ll target){
    int mid = (l + r) >> 1;
    while (l <= r){
        if (increase[mid] >= target){
            r = mid - 1;
        }
        else {
            l = mid + 1;
        }
        mid = (l + r) >> 1;
    }
    return l;
}

// 二分查找：找到第一个严格比目标小的位置
int bisearch_gt(int l, int r, ll target){
    int mid = (l + r) >> 1;
    while (l <= r){
        if (decrease[mid] < target){
            r = mid - 1;
        }
        else {
            l = mid + 1;
        }
        mid = (l + r) >> 1;
    }
    return l;
}

int main(){
    ll num;
    while(~scanf("%lld", &num)){
        arr[cnt++] = num;
        int de = bisearch_gt(0, decrease_cnt - 1, num);
        int in = bisearch_le(0, increase_cnt - 1, num);
        if (de == decrease_cnt){
            decrease[decrease_cnt++] = num;
        }
        else {
            decrease[de] = num;
        }
        if (in == increase_cnt){
            increase[increase_cnt++] = num;
        }
        else {
            increase[in] = num;
        }
    }
    cout << decrease_cnt << endl << increase_cnt << endl;
```

```
    return 0;
}
```

区间DP

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 307;
ll dp[maxn][maxn], arr[maxn], sum[maxn];

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++){
        cin >> arr[i];
        sum[i] = sum[i - 1] + arr[i];  // 前缀和
    }
    memset(dp, 0x3f, sizeof(dp));
    for (int i = 1; i <= n; i++){
        dp[i][i] = 0;  // 单个元素合并代价为0
    }
    // dp[l][r]: 合并区间[l,r]的最小代价
    for (int len = 2; len <= n; len++){  // 枚举区间长度
        for (int l = 1; l + len - 1 <= n; l++){
            int r = l + len - 1;
            for (int k = l; k < r; k++){  // 枚举分割点
                dp[l][r] = min(dp[l][r], dp[l][k] + dp[k + 1][r] + sum[r]
- sum[l - 1]);
            }
        }
    }
    cout << dp[1][n] << endl;
    return 0;
}
```

数位DP

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll dp[20][20];
int digit[20];

// pos: 当前位置, pre: 前一位数字, limit: 是否达到上界, lead: 是否有前导0
ll dfs(int pos, int pre, bool limit, bool lead){
    if (pos == 0) return 1;
```

```
    if (!limit && !lead && dp[pos][pre] != -1) return dp[pos][pre];
    int up = limit ? digit[pos] : 9;
    ll res = 0;
    for (int i = 0; i <= up; i++){
        if (lead || abs(i - pre) >= 2){  // 条件：相邻数字差至少为2
            res += dfs(pos - 1, i, limit && (i == up), lead && (i == 0));
        }
    }
    if (!limit && !lead) dp[pos][pre] = res;
    return res;
}

ll solve(ll x){
    int len = 0;
    while (x){
        digit[++len] = x % 10;
        x /= 10;
    }
    memset(dp, -1, sizeof(dp));
    return dfs(len, 0, true, true);
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    ll a, b;
    cin >> a >> b;
    cout << solve(b) - solve(a - 1) << endl;
    return 0;
}
```

## 最长公共子序列（LCS）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1007;
int dp[maxn][maxn];
string s1, s2;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> s1 >> s2;
    int n = s1.length(), m = s2.length();
    // dp[i][j]: s1前i个字符和s2前j个字符的LCS长度
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= m; j++){
            if (s1[i - 1] == s2[j - 1]){
                dp[i][j] = dp[i - 1][j - 1] + 1;
            }
            else {
```

```
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    cout << dp[n][m] << endl;
    return 0;
}
```

---

# 图论

## Dijkstra（单源最短路，非负权）

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
int node[200007], edge[200007], weight[200007], nxt[200007], dist[200007],
visit[200007];
const int inf = 0x3f3f3f3f;

// 添加边：链式前向星
void addEdge(int i){
    int f, t, w;
    cin >> f >> t >> w;
    weight[i] = w;
    nxt[i] = node[f];
    edge[i] = t;
    node[f] = i;
}

priority_queue <pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

signed main(){
    int n, m, s;  // n:点数, m:边数, s:起点
    cin >> n >> m >> s;
    memset(dist, 0x3f, sizeof(dist));
    for (int i = 1; i <= m; i++){
        addEdge(i);
    }
    dist[s] = 0;
    pq.push({0, s});
    while (!pq.empty()){
        auto [d, cur] = pq.top();
        pq.pop();
        if (visit[cur]) continue;  // 已访问过，跳过
        visit[cur] = 1;
        // 松弛操作
        for (int e = node[cur]; e; e = nxt[e]){
            int t = edge[e];
            if (dist[t] > dist[cur] + weight[e]){
```

```
                dist[t] = dist[cur] + weight[e];
                pq.push({dist[t], t});
            }
        }
    }
    for (int i = 1; i <= n; i++){
        cout << dist[i] << " ";
    }
    cout << endl;
    return 0;
}
```

SPFA（单源最短路，可判负环）

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

int node[3007], edge[6007], nxt[6007], weight[6007], dist[3007];
queue<int> que;
int visit[3007], cnt[3007];  // cnt:记录入队次数，用于判负环
int n, m, c = 1;

void addEdge(int i){
    int f, t, w;
    cin >> f >> t >> w;
    nxt[c] = node[f];
    node[f] = c;
    edge[c] = t;
    weight[c] = w;
    c++;
    if (w >= 0){  // 无向边
        nxt[c] = node[t];
        node[t] = c;
        edge[c] = f;
        weight[c] = w;
        c++;
    }
}

bool spfa(){
    que.push(1), dist[1] = 0;
    while (!que.empty()){
        int cur = que.front();
        visit[cur] = 0, que.pop();
        for (int e = node[cur]; e; e = nxt[e]){
            int t = edge[e];
            if (dist[t] > dist[cur] + weight[e]){
                dist[t] = dist[cur] + weight[e];
                if (!visit[t]) {
                    cnt[t] = cnt[cur] + 1;
                    que.push(t), visit[t] = 1;
```

```
                    if (cnt[t] >= n) return true;  // 入队次数>=n，存在负环
                }
            }
        }
    }
    return false;
}

int main(){
    int t;
    cin >> t;
    while (t--){
        c = 1;
        memset(nxt, 0, sizeof(nxt));
        memset(node, 0, sizeof(node));
        memset(edge, 0, sizeof(edge));
        memset(dist, 0x3f, sizeof(dist));
        memset(cnt, 0, sizeof(cnt));
        memset(visit, 0, sizeof(visit));
        while (!que.empty()) que.pop();
        cin >> n >> m;
        for (int i = 1; i <= m; i++){
            addEdge(i);
        }
        if (spfa()){
            cout << "YES" << endl;  // 存在负环
        }
        else {
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

## 全源最短路（Johnson算法）

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int inf = 1e9;
int n, m;
ll node[10007], edge[10007], nxt[10007], weight[10007], nw[10007];
ll ecnt = 1, tcnt;
queue<ll> que;
priority_queue<pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll, ll>>>
pq;
ll visit[10007], dist1[10007], dist2[10007], cntc[10007];

void addEdge(ll f, ll t, ll w){
    nxt[ecnt] = node[f];
    weight[ecnt] = w;
```

```
    node[f] = ecnt;
    edge[ecnt] = t;
    ecnt++;
}

// SPFA判负环并计算势能
bool spfa(){
    que.push(0);
    memset(dist1, 0x3f, sizeof(dist1));
    memset(cntc, 0, sizeof(cntc));
    dist1[0] = 0;
    while (!que.empty()){
        int cur = que.front();
        que.pop(), visit[cur] = 0;
        for (ll e = node[cur]; e; e = nxt[e]){
            ll t = edge[e], w = weight[e];
            if (dist1[t] > dist1[cur] + w){
                dist1[t] = dist1[cur] + w;
                if (!visit[t]){
                    cntc[t] = cntc[cur] + 1;
                    que.push(t);
                    visit[t] = 1;
                    if (cntc[t] >= n + 1) return true;  // 负环
                }
            }
        }
    }
    return false;
}

// 重新赋权：w' = w + h[u] - h[v]
void processNw(){
    for (int i = 0; i <= n; i++){
        for (int e = node[i]; e; e = nxt[e]){
            ll t = edge[e];
            nw[e] = weight[e] + dist1[i] - dist1[t];
        }
    }
}

// Dijkstra求单源最短路
void dij(ll st){
    memset(dist2, 0x3f, sizeof(dist2));
    memset(visit, 0, sizeof(visit));
    while (!pq.empty()) pq.pop();
    dist2[st] = 0;
    pq.push({0, st});
    while (!pq.empty()){
        auto [d, cur] = pq.top();
        pq.pop();
        if (visit[cur]) continue;
        visit[cur] = 1;
        for (ll e = node[cur]; e; e = nxt[e]){
            ll t = edge[e], w = nw[e];
```

```
            if (dist2[t] > dist2[cur] + w){
                dist2[t] = dist2[cur] + w;
                pq.push({dist2[t], t});
            }
        }
    }
    ll sum = 0;
    for (int i = 1; i <= n; i++) {
        if (dist2[i] == 0x3f3f3f3f3f3f3f3fLL) {
            sum += (ll)i * inf;
        } else {
            ll actual_dist = dist2[i] - dist1[st] + dist1[i];  // 还原实际距
离
            sum += (ll)i * actual_dist;
        }
    }
    cout << sum << endl;
}

int main(){
    memset(node, 0, sizeof(node));
    memset(nxt, 0, sizeof(nxt));
    memset(edge, 0, sizeof(edge));
    memset(weight, 0, sizeof(weight));
    cin >> n >> m;
    for (int i = 1; i <= m; i++){
        int f, t, w;
        cin >> f >> t >> w;
        addEdge(f, t, w);
    }
    tcnt = ecnt;
    // 添加虚拟源点0
    for (int i = 1; i <= n; i++){
        addEdge(0, i, 0);
    }
    if (spfa()) {
        cout << -1 << endl;  // 存在负环
        return 0;
    }
    processNw();
    for (int i = 1; i <= n; i++){
        dij(i);
    }
    return 0;
}
```

## 无向图最小环（Floyd）

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

```
ll mp[207][207], dist[207][207];
const ll INF = 0x3f3f3f3f;

int main(){
    int n, m;
    cin >> n >> m;
    memset(mp, 0x3f, sizeof(mp));
    memset(dist, 0x3f, sizeof(dist));
    for (int i = 1; i <= n; i++){
        dist[i][i] = mp[i][i] = 0;
    }
    ll u, v, w;
    for (int i = 1; i <= m; i++){
        cin >> u >> v >> w;
        dist[u][v] = min(dist[u][v], w);
        dist[v][u] = min(dist[v][u], w);
        mp[u][v] = min(mp[u][v], w);
        mp[v][u] = min(mp[v][u], w);
    }
    ll ans = 0x3f3f3f3f;
    // Floyd找最小环：在更新dist前，检查经过k的环
    for (int k = 1; k <= n; k++){
        for (int i = 1; i < k; i++){
            for (int j = i + 1; j < k; j++){
                if (dist[i][j] < INF && mp[i][k] < INF && mp[k][j] < INF)
                    ans = min(ans, mp[i][k] + mp[k][j] + dist[i][j]);
            }
        }
        // Floyd更新最短路
        for (int i = 1; i <= n; i++){
            for (int j = 1; j <= n; j++){
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                dist[j][i] = dist[i][j];
            }
        }
    }
    if(ans == 0x3f3f3f3f) cout << "No solution." << endl;
    else cout << ans << endl;
    return 0;
}
```

## Kruskal（最小生成树）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 5007;
struct Edge{
    int u, v, w;
    bool operator < (const Edge& e) const {
        return w < e.w;
    }
```

```
} edges[maxn];
int parent[maxn];
int n, m;

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

int kruskal(){
    sort(edges, edges + m);
    for (int i = 1; i <= n; i++){
        parent[i] = i;
    }
    int ans = 0, cnt = 0;
    for (int i = 0; i < m; i++){
        int u = edges[i].u, v = edges[i].v, w = edges[i].w;
        int fu = find(u), fv = find(v);
        if (fu != fv){
            parent[fu] = fv;
            ans += w;
            cnt++;
            if (cnt == n - 1) break;
        }
    }
    return cnt == n - 1 ? ans : -1;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    for (int i = 0; i < m; i++){
        cin >> edges[i].u >> edges[i].v >> edges[i].w;
    }
    int ans = kruskal();
    if (ans == -1) cout << "orz" << endl;
    else cout << ans << endl;
    return 0;
}
```

Prim（最小生成树，堆优化）

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 5007;
int node[maxn], edge[maxn << 1], nxt[maxn << 1], weight[maxn << 1], cnt =
1;
```

```
int visit[maxn];
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;

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

int prim(int n){
    int ans = 0, tot = 0;
    pq.push({0, 1});
    while (!pq.empty() && tot < n){
        auto [w, u] = pq.top();
        pq.pop();
        if (visit[u]) continue;
        visit[u] = 1;
        ans += w;
        tot++;
        for (int e = node[u]; e; e = nxt[e]){
            int v = edge[e];
            if (!visit[v]){
                pq.push({weight[e], v});
            }
        }
    }
    return tot == n ? ans : -1;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= m; i++){
        int u, v, w;
        cin >> u >> v >> w;
        addEdge(u, v, w);
        addEdge(v, u, w);
    }
    int ans = prim(n);
    if (ans == -1) cout << "orz" << endl;
    else cout << ans << endl;
    return 0;
}
```

## 拓扑排序

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
int head[100007], to[100007], nxt[100007], weight[100007],
indegree[100007];
int visit[100007];
ll dist[100007];
int n, m;
const ll inf = 0x3f3f3f3f3f3f3f3f;
vector<int> ans;

void addEdge(int i){
    int u, v, w;
    cin >> u >> v >> w;
    nxt[i] = head[u];
    to[i] = v;
    weight[i] = w;
    head[u] = i;
    indegree[v]++;
}

void topo(){
    queue<int> que;
    for (int i = 1; i <= n; i++){
        if (indegree[i] == 0) que.push(i);
    }
    while (!que.empty()){
        int cur = que.front();
        que.pop();
        if (visit[cur]) continue;
        visit[cur] = 1;
        ans.push_back(cur);
        for (int e = head[cur]; e; e = nxt[e]){
            int t = to[e];
            if (--indegree[t] == 0) que.push(t);
        }
    }
}

// DAG上的最短路
void road(){
    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;
    for (auto x : ans){
        if (dist[x] == inf) continue;
        for (int e = head[x]; e; e = nxt[e]){
            int t = to[e], w = weight[e];
            dist[t] = min(dist[t], dist[x] + w);
        }
    }
}

int main(){
```

```
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    for (int i = 1; i <= m; i++){
        addEdge(i);
    }
    topo();
    for (auto x : ans){
        cout << x << " ";
    }
    cout << endl;
    road();
    for (int i = 1; i <= n; i++){
        cout << (dist[i] == inf ? -1 : dist[i]) << " ";
    }
    cout << endl;
    return 0;
}
```

## 二分图最大匹配（匈牙利算法）

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

ll node[50007], edge[50007], nxt[50007];
int match[50007], visit[50007];

void addEdge(int i){
    ll a, b;
    cin >> a >> b;
    nxt[i] = node[a];
    node[a] = i;
    edge[i] = b;
}

// 寻找增广路
int dfs(ll u){
    for (int e = node[u]; e; e = nxt[e]){
        int v = edge[e];
        if (visit[v]) continue;
        visit[v] = 1;
        if (!match[v] || dfs(match[v])){  // v未匹配或能找到增广路
            match[v] = u;
            return 1;
        }
    }
    return 0;
}

int main(){
```

```
    ios::sync_with_stdio(false);
    cin.tie(0);
    ll n, m, e;  // n:左部点数, m:右部点数, e:边数
    cin >> n >> m >> e;
    for (int i = 1; i <= e; i++){
        addEdge(i);
    }
    int ans = 0;
    for (int i = 1; i <= n; i++){
        memset(visit, 0, sizeof(visit));
        ans += dfs(i);
    }
    cout << ans << endl;
    return 0;
}
```

## 最大流（Edmonds-Karp）

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll inf = 1e11;
ll st, ed, n, m, cnt = 0;
ll node[10007], edge[10007], nxt[10007], weight[10007], mf[10007],
pre[10007];

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

// BFS找增广路
bool bfs(){
    memset(mf, 0, sizeof(mf));
    queue<ll> q;
    q.push(st);
    mf[st] = inf;
    while (!q.empty()){
        ll cur = q.front(); q.pop();
        for (ll e = node[cur]; e >= 0; e = nxt[e]){
            ll t = edge[e], w = weight[e];
            if (mf[t] == 0 && w > 0){
                mf[t] = min(mf[cur], w);
                pre[t] = e;
                q.push(t);
                if (t == ed) return true;
            }
        }
```

```
    }
    return false;
}

ll ek(){
    ll flow = 0;
    while (bfs()){
        int tmp = ed;
        // 沿增广路更新流量
        while (tmp != st){
            ll e = pre[tmp];
            weight[e] -= mf[ed];
            weight[e^1] += mf[ed];  // 反向边
            tmp = edge[e^1];
        }
        flow += mf[ed];
    }
    return flow;
}

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

## 最大流最小割（Dinic）

```
#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], h[4007], mf[4007], cur[4007], visit[4007],
a[4007], b[4007], we[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;
    return sum;
}

ll dinic(){
    ll flow = 0;
    while (bfs()){
        memcpy(cur, node, sizeof(cur));  // 重置当前弧
        flow += dfs(st, 1e10);
    }
    return flow;
}

// 最小割：从源点DFS标记可达点
void mincut(ll u){
    visit[u] = 1;
    for (ll e = node[u]; e >= 0; e = nxt[e]){
```

```
        ll t = edge[e];
        if (!visit[t] && weight[e]) mincut(t);
    }
}

int main(){
    memset(node, -1, sizeof(node));
    cin >> n >> m;
    st = 1, ed = n;
    for (int i = 0; i < m; i++){
        ll w;
        cin >> a[i] >> b[i] >> we[i];
        addEdge(a[i], b[i], we[i]);
        addEdge(b[i], a[i], 0);
    }
    cout << dinic() << " ";
    // 重新建图计算最小割边数
    cnt = 0;
    memset(node, -1, sizeof(node));
    for (int i = 0; i < m; i++){
        addEdge(a[i], b[i], we[i] * (m + 1) + 1);
        addEdge(b[i], a[i], 0);
    }
    cout << dinic() % (m + 1) << endl;
    return 0;
}
```

### 最小费用最大流

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const ll inf = 1e11;
ll st, ed, n, m, cnt = 0, flow = 0, cost = 0;
ll node[100007], edge[100007], nxt[100007], dist[100007],
weight[100007], c[100007], mf[100007], pre[100007], visit[100007];

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

// SPFA找费用最小的增广路
bool spfa(){
    memset(visit, 0, sizeof(visit));
    memset(mf, 0, sizeof(mf));
    memset(dist, 0x3f, sizeof(dist));
```

```
    queue<ll> que;
    que.push(st);
    dist[st] = 0, mf[st] = inf, visit[st] = 1;
    while (!que.empty()){
        int cur = que.front(); que.pop(); visit[cur] = 0;
        for (ll e = node[cur]; e >= 0; e = nxt[e]){
            ll t = edge[e], w = weight[e], limit = c[e];
            if (dist[t] > dist[cur] + w && limit){
                dist[t] = dist[cur] + w;
                mf[t] = min(mf[cur], limit);
                pre[t] = e;
                if (!visit[t]){
                    que.push(t);
                    visit[t] = 1;
                }
            }
        }
    }
    return mf[ed] > 0;
}

ll ek(){
    while (spfa()){
        int tmp = ed;
        while (tmp != st){
            ll e = pre[tmp];
            c[e] -= mf[ed];
            c[e^1] += mf[ed];
            tmp = edge[e^1];
        }
        flow += mf[ed];
        cost += mf[ed] * dist[ed];
    }
    return flow;
}

int main(){
    cin >> n >> m >> st >> ed;
    memset(node, -1, sizeof(node));
    for (int i = 0; i < m; i++){
        ll u, v, limit, w;
        cin >> u >> v >> limit >> w;
        addEdge(u, v, limit, w);
        addEdge(v, u, 0, -w);  // 反向边费用为负
    }
    ek();
    cout << flow << " " << cost << endl;
    return 0;
}
```

## Tarjan（强连通分量）

```cpp
#include <bits/stdc++.h>
using namespace std;
const int maxn = 100007;
int node[maxn], edge[maxn], nxt[maxn], cnt = 1;
int dfn[maxn], low[maxn], scc[maxn], scccnt = 0, idx = 0;
stack<int> stk;
bool instk[maxn];

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

void tarjan(int u){
    dfn[u] = low[u] = ++idx;
    stk.push(u);
    instk[u] = true;
    for (int e = node[u]; e; e = nxt[e]){
        int v = edge[e];
        if (!dfn[v]){
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }
        else if (instk[v]){
            low[u] = min(low[u], dfn[v]);
        }
    }
    // u是强连通分量的根
    if (dfn[u] == low[u]){
        scccnt++;
        int v;
        do {
            v = stk.top();
            stk.pop();
            instk[v] = false;
            scc[v] = scccnt;
        } while (v != u);
    }
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= m; i++){
        int u, v;
        cin >> u >> v;
        addEdge(u, v);
    }
    for (int i = 1; i <= n; i++){
```

```
        if (!dfn[i]){
            tarjan(i);
        }
    }
    cout << scccnt << endl;
    return 0;
}
```

## LCA（最近公共祖先，倍增）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 500007;
int node[maxn], edge[maxn << 1], nxt[maxn << 1], cnt = 1;
int depth[maxn], fa[maxn][20], lg[maxn];

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

// DFS预处理深度和父节点
void dfs(int u, int father){
    depth[u] = depth[father] + 1;
    fa[u][0] = father;
    for (int i = 1; i <= lg[depth[u]]; i++){
        fa[u][i] = fa[fa[u][i - 1]][i - 1];  // 倍增
    }
    for (int e = node[u]; e; e = nxt[e]){
        int v = edge[e];
        if (v != father){
            dfs(v, u);
        }
    }
}

int lca(int x, int y){
    if (depth[x] < depth[y]){
        swap(x, y);
    }
    // 将x提升到与y同一深度
    while (depth[x] > depth[y]){
        x = fa[x][lg[depth[x] - depth[y]]];
    }
    if (x == y){
        return x;
    }
    // 同时向上跳
    for (int i = lg[depth[x]]; i >= 0; i--){
```

```
            if (fa[x][i] != fa[y][i]){
                x = fa[x][i];
                y = fa[y][i];
            }
        }
        return fa[x][0];
    }

    int main(){
        ios::sync_with_stdio(false);
        cin.tie(0);
        int n, m, s;  // s:根节点
        cin >> n >> m >> s;
        lg[0] = -1;
        for (int i = 1; i <= n; i++){
            lg[i] = lg[i >> 1] + 1;
        }
        for (int i = 1; i < n; i++){
            int u, v;
            cin >> u >> v;
            addEdge(u, v);
            addEdge(v, u);
        }
        dfs(s, 0);
        while (m--){
            int x, y;
            cin >> x >> y;
            cout << lca(x, y) << endl;
        }
        return 0;
    }
```

---

# 字符串

## KMP

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

int nxt[1000007];
vector<int> ans;

// 计算next数组
void getNxt(string s){
    int len = s.length();
    nxt[0] = -1;
    nxt[1] = 0;
    int i = 2, cn = 0;  // cn:当前匹配长度
    while (i <= len){
        if (s[i - 1] == s[cn]){
```

```
            nxt[i++] = ++cn;
        }
        else if (cn > 0){
            cn = nxt[cn];  // 回退
        }
        else {
            nxt[i++] = 0;
        }
    }
}

// KMP匹配
int kmp(string s1, string s2){
    getNxt(s2);
    int len1 = s1.length(), len2 = s2.length(), x = 0, y = 0;
    while (x < len1 && y < len2){
        if (s1[x] == s2[y]){
            x++;
            y++;
        }
        else if (y == 0){
            x++;
        }
        else {
            y = nxt[y];  // 利用next数组跳转
        }
        if (y == len2) {
            cout << x - y + 1 << endl;  // 输出匹配位置
            y = nxt[y];  // 继续匹配
        }
    }
    return y == len2 ? x - y : -1;
}

int main(){
    string s1, s2;
    cin >> s1 >> s2;
    int len2 = s2.length(), len1 = s1.length();
    kmp(s1, s2);
    for (int i = 1; i <= len2; i++){
        cout << nxt[i] << " ";
    }
    return 0;
}
```

## Manacher（最长回文子串）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 11000007;
char s[maxn], str[maxn << 1];
```

```
int p[maxn << 1];

int manacher(){
    int len = strlen(s);
    // 插入分隔符，统一奇偶长度
    str[0] = '$';
    str[1] = '#';
    int cnt = 2;
    for (int i = 0; i < len; i++){
        str[cnt++] = s[i];
        str[cnt++] = '#';
    }
    str[cnt] = '\0';
    int mx = 0, id = 0, ans = 0;  // mx:最右边界, id:中心点
    for (int i = 1; i < cnt; i++){
        if (i < mx){
            p[i] = min(p[2 * id - i], mx - i);  // 利用对称性
        }
        else {
            p[i] = 1;
        }
        // 中心扩展
        while (str[i + p[i]] == str[i - p[i]]){
            p[i]++;
        }
        if (mx < i + p[i]){
            mx = i + p[i];
            id = i;
        }
        ans = max(ans, p[i] - 1);
    }
    return ans;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> s;
    cout << manacher() << endl;
    return 0;
}
```

## 字符串哈希

```
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long ull;
const int maxn = 1000007;
const ull base = 131;  // 哈希基数
ull h[maxn], p[maxn];
char s[maxn];
```

```
// 初始化哈希数组
void init(){
    p[0] = 1;
    int len = strlen(s + 1);
    for (int i = 1; i <= len; i++){
        h[i] = h[i - 1] * base + s[i];
        p[i] = p[i - 1] * base;
    }
}

// 获取子串哈希值
ull getHash(int l, int r){
    return h[r] - h[l - 1] * p[r - l + 1];
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> (s + 1);
    init();
    int m;
    cin >> m;
    while (m--){
        int l1, r1, l2, r2;
        cin >> l1 >> r1 >> l2 >> r2;
        if (getHash(l1, r1) == getHash(l2, r2)){
            cout << "Yes" << endl;
        }
        else {
            cout << "No" << endl;
        }
    }
    return 0;
}
```

---

## 数学

### 快速幂

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

// 快速幂：计算 a^b mod p
ll qpow(ll a, ll b, ll mod){
    ll res = 1;
    while (b){
        if (b & 1){
            res = res * a % mod;
```

```
        }
        a = a * a % mod;
        b >>= 1;
    }
    return res;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    ll a, b, p;
    cin >> a >> b >> p;
    cout << a << "^" << b << " mod " << p << "=" << qpow(a, b, p) << endl;
    return 0;
}
```

## 扩展欧几里得

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

// 扩展欧几里得：求解 ax + by = gcd(a,b)
ll exgcd(ll a, ll b, ll &x, ll &y){
    if (b == 0){
        x = 1, y = 0;
        return a;
    }
    ll d = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return d;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    while (t--){
        ll a, b, c;
        cin >> a >> b >> c;
        ll x, y;
        ll d = exgcd(a, b, x, y);
        if (c % d != 0){
            cout << -1 << endl;  // 无解
            continue;
        }
        x *= c / d;
        y *= c / d;
        ll dx = b / d, dy = a / d;
        ll xmin = (x % dx + dx) % dx;
```

```
        if (xmin == 0) xmin = dx;
        ll ymax = (c - a * xmin) / b;
        if (ymax <= 0){
            cout << xmin << " " << (y % dy + dy) % dy << endl;
        }
        else {
            ll ymin = (y % dy + dy) % dy;
            if (ymin == 0) ymin = dy;
            ll xmax = (c - b * ymin) / a;
            cout << (xmax - xmin) / dx + 1 << " " << xmin << " " << ymin
<< " " << xmax << " " << ymax << endl;
        }
    }
    return 0;
}
```

## 素数筛（欧拉筛）

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 10000007;
bool isPrime[maxn];
vector<int> primes;

// 欧拉筛：线性时间复杂度
void sieve(int n){
    memset(isPrime, true, sizeof(isPrime));
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i <= n; i++){
        if (isPrime[i]){
            primes.push_back(i);
        }
        for (int j = 0; j < primes.size() && i * primes[j] <= n; j++){
            isPrime[i * primes[j]] = false;
            if (i % primes[j] == 0){  // 关键：保证每个合数只被最小质因子筛一次
                break;
            }
        }
    }
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, q;
    cin >> n >> q;
    sieve(n);
    while (q--){
        int k;
        cin >> k;
        cout << primes[k - 1] << endl;
```

```
    }
    return 0;
}
```

### 组合数

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 2007;
const int mod = 1e9 + 7;
ll C[maxn][maxn];

// 预处理组合数: C(n,m) = C(n-1,m) + C(n-1,m-1)
void init(){
    for (int i = 0; i < maxn; i++){
        C[i][0] = 1;
        for (int j = 1; j <= i; j++){
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
        }
    }
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    init();
    int n;
    cin >> n;
    while (n--){
        int a, b;
        cin >> a >> b;
        cout << C[a][b] << endl;
    }
    return 0;
}
```

---

## 几何

### 凸包（Andrew算法）

```
#include <bits/stdc++.h>
using namespace std;
const double PI = acos(-1.0);
int n;
double a, b;

typedef struct Point
```

```
{
    double x, y;
    Point operator+(const Point &b) const
    {
        return {x + b.x, y + b.y};
    }
    Point operator-(const Point &b) const
    {
        return {x - b.x, y - b.y};
    }
    bool operator<(const Point &b) const
    {
        return x != b.x ? x < b.x : y < b.y;
    }
    Point rotate(double theta) const
    {
        double c = cos(theta);
        double s = sin(theta);
        return {x * c - y * s, x * s + y * c};
    }
} Point;

Point all[80007];
int cnt = 0;

// 叉积
double cross(Point a, Point b)
{
    return a.x * b.y - a.y * b.x;
}

// 三点叉积
double cross_product(Point a, Point b, Point c)
{
    return cross(b - a, c - a);
}

// 距离
double dis(Point a, Point b)
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

// Andrew算法求凸包
vector<Point> andrew(int sz)
{
    sort(all, all + sz, [](const Point &a, const Point &b)
         { return a < b; });
    vector<Point> hull;
    // 下凸包
    for (int i = 0; i < sz; i++)
    {
        while (hull.size() >= 2 && cross_product(hull[hull.size() - 2],
hull.back(), all[i]) <= 0)
```

```
        {
            hull.pop_back();
        }
        hull.push_back(all[i]);
    }
    int len = hull.size();
    // 上凸包
    for (int i = sz - 2; i >= 0; i--)
    {
        while (hull.size() > len && cross_product(hull[hull.size() - 2],
hull.back(), all[i]) <= 0)
        {
            hull.pop_back();
        }
        hull.push_back(all[i]);
    }
    return hull;
}

int main()
{
    int t;
    cin >> t;
    while (t--)
    {
        cin >> n;
        cnt = 0;
        double x, y, theta;
        for (int i = 0; i < n; i++){
            cin >> x >> y;
            Point center = {x, y};
            all[cnt++] = center;
        }
        vector<Point> ans = andrew(n);
        double pt = 0;
        for (int i = 0; i < ans.size() - 1; i++){
            pt += dis(ans[i], ans[i + 1]);
        }
        cout << fixed << setprecision(10) << pt << endl;
    }
    return 0;
}
```

---

# FFT

## 大数乘法（FFT）

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

```
struct Complex {
    double x, y; // 实部 x, 虚部 y
    Complex(double _x = 0, double _y = 0) : x(_x), y(_y) {}
    Complex operator + (const Complex& b) const {
        return Complex(x + b.x, y + b.y);
    }
    Complex operator - (const Complex& b) const {
        return Complex(x - b.x, y - b.y);
    }
    Complex operator * (const Complex& b) const {
        return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
    }
    Complex& operator += (const Complex& b) {
        x += b.x;
        y += b.y;
        return *this;
    }
    Complex& operator *= (const Complex& b) {
        double tmp_x = x * b.x - y * b.y;
        y = x * b.y + y * b.x;
        x = tmp_x;
        return *this;
    }
};

const double PI = acos(-1.0);

// FFT: 快速傅里叶变换
void fft(vector<Complex>& a, bool invert) {
    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 * PI / len * (invert ? -1 : 1);
        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];
                Complex v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
```

```
    if (invert) {
        for (int i = 0; i < n; i++) a[i].x /= n;
    }
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int t;
    cin >> t;
    string xx, yy;
    while (t--){
        cin >> xx >> yy;
        int al = xx.length(), bl = yy.length();
        int n = 1;
        while (n < al + bl + 2) n <<= 1;  // 扩展到2的幂次
        vector<Complex> fa(n), fb(n);
        // 将字符串转为复数（倒序存储）
        for (int i = 0; i < al; i++) {
            fa[i] = Complex(xx[al - 1 - i] - '0', 0);
        }
        for (int i = 0; i < bl; i++) {
            fb[i] = Complex(yy[bl - 1 - i] - '0', 0);
        }
        fft(fa, false), fft(fb, false);
        vector<int> result(n);
        for (int i = 0; i < n; i++) fa[i] *= fb[i];
        fft(fa, true);
        for (int i = 0; i < n; i++) {
            result[i] = round(fa[i].x);
        }
        // 处理进位
        for (int i = 0; i < n - 1; i++) {
            result[i + 1] += result[i] / 10;
            result[i] %= 10;
        }
        int pos = n - 1;
        while (pos > 0 && result[pos] == 0) pos--;
        for (; pos >= 0; pos--) {
            cout << result[pos];
        }
        cout << endl;
    }
    return 0;
}
```

## 多项式乘积（FFT）

```
#include <iostream>
#include <vector>
#include <complex>
```

```cpp
#include <cmath>
using namespace std;

using cd = complex<double>;
const double PI = acos(-1);
int offset = 0, cnt = 0;

// FFT递归实现
void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    if (n == 1) return;
    // 分治
    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2 * i];
        a1[i] = a[2 * i + 1];
    }
    fft(a0, invert);
    fft(a1, invert);
    // 合并
    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n / 2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n / 2] /= 2;
        }
        w *= wn;
    }
}

// 多项式乘法
vector<int> multiply(const vector<int>& a, const vector<int>& b) {
    int n = 1;
    while (n < a.size() + b.size()) n <<= 1;
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++) {
        fa[i] *= fb[i];
    }
    fft(fa, true);
    vector<int> result(n);
    cnt = n;
    for (int i = 0; i < n; i++) {
        result[i] = round(fa[i].real());
        if (result[i] == 0) offset++;
        else offset = 0;
    }
    return result;
```

```
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<int> a(n + 1);
    vector<int> b(m + 1);
    for (int i = 0; i <= n; i++){
        cin >> a[i];
    }
    for (int i = 0; i <= m; i++){
        cin >> b[i];
    }
    vector<int> res = multiply(a, b);
    for (int i = 0; i < cnt - offset; i++) {
        cout << res[i] << " ";
    }
    cout << endl;
    return 0;
}
```

---

# 其他

## 二分查找

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1000007;
int arr[maxn];

// 找到第一个大于等于target的位置
int bisearch_le(int l, int r, int target){
    int mid;
    while (l <= r){
        mid = (l + r) >> 1;
        if (arr[mid] >= target){
            r = mid - 1;
        }
        else {
            l = mid + 1;
        }
    }
    return l;
}

// 找到第一个大于target的位置
int bisearch_gt(int l, int r, int target){
    int mid;
    while (l <= r){
        mid = (l + r) >> 1;
```

```
        if (arr[mid] > target){
            r = mid - 1;
        }
        else {
            l = mid + 1;
        }
    }
    return l;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++){
        cin >> arr[i];
    }
    while (m--){
        int x;
        cin >> x;
        int pos1 = bisearch_le(1, n, x);
        int pos2 = bisearch_gt(1, n, x);
        if (arr[pos1] == x){
            cout << pos1 << " " << pos2 - 1 << endl;
        }
        else {
            cout << -1 << " " << -1 << endl;
        }
    }
    return 0;
}
```

## 三分查找

```
#include <bits/stdc++.h>
using namespace std;
const double eps = 1e-7;

// 单峰函数（需要根据题目修改）
double f(double x){
    return x * x * x - 3 * x * x - 3 * x - 1;
}

// 三分查找：找单峰函数的极值
double ternary_search(double l, double r){
    while (r - l > eps){
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        if (f(m1) < f(m2)){  // 找最小值，找最大值则改为>
            l = m1;
```

```
            }
            else {
                r = m2;
            }
        }
        return (l + r) / 2;
    }

    int main(){
        ios::sync_with_stdio(false);
        cin.tie(0);
        double l, r;
        cin >> l >> r;
        cout << fixed << setprecision(5) << ternary_search(l, r) << endl;
        return 0;
    }
```

---

## 📝 使用说明

1. **快速定位**：使用 Ctrl+F 搜索关键词
2. **代码风格**：所有模板使用统一的代码风格
    - `#include <bits/stdc++.h>`
    - `using namespace std;`
    - `typedef long long ll;`
    - 链式前向星存储图
3. **注意事项**：
    - 注意数组大小限制
    - 注意数据范围（int/long long）
    - 注意初始化操作
    - 注意输入输出优化

---

*算法模板完整版 - 适用于算法考试*