拓扑排序模板（C++）

以下是两种常用的拓扑排序实现方法：Kahn算法（基于入度）和DFS算法。

方法一：Kahn算法（基于BFS）

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

vector<int> topologicalSort(int n, vector<vector<int>>& graph) {
    vector<int> inDegree(n, 0);
    vector<vector<int>> adj(n); // 邻接表

    // 构建图和入度表
    for (auto& edge : graph) {
        int u = edge[0], v = edge[1];
        adj[u].push_back(v);
        inDegree[v]++;
    }

    queue<int> q;
    // 将所有入度为0的节点加入队列
    for (int i = 0; i < n; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    vector<int> result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        // 减少所有邻居节点的入度
        for (int v : adj[u]) {
            inDegree[v]--;
            if (inDegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // 如果结果中的节点数不等于总节点数，说明存在环
    if (result.size() != n) {
        return {}; // 返回空数组表示有环，无法拓扑排序
    }
```

```
    return result;
}

int main() {
    // 示例：6个节点，边为[[5,2],[5,0],[4,0],[4,1],[2,3],[3,1]]
    int n = 6;
    vector<vector<int>> graph = {{5,2}, {5,0}, {4,0}, {4,1}, {2,3}, {3,1}};

    vector<int> result = topologicalSort(n, graph);

    if (result.empty()) {
        cout << "图中存在环，无法进行拓扑排序" << endl;
    } else {
        cout << "拓扑排序结果： ";
        for (int node : result) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

方法二：DFS算法

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

bool dfs(int u, vector<vector<int>>& adj, vector<int>& visited, stack<int>&
st) {
    visited[u] = 1; // 标记为正在访问

    for (int v : adj[u]) {
        if (visited[v] == 1) return false; // 发现环
        if (visited[v] == 0 && !dfs(v, adj, visited, st)) return false;
    }

    visited[u] = 2; // 标记为已访问
    st.push(u);
    return true;
}

vector<int> topologicalSortDFS(int n, vector<vector<int>>& graph) {
    vector<vector<int>> adj(n);
    for (auto& edge : graph) {
        int u = edge[0], v = edge[1];
        adj[u].push_back(v);
    }
```

```
    vector<int> visited(n, 0); // 0:未访问, 1:正在访问, 2:已访问
    stack<int> st;

    for (int i = 0; i < n; i++) {
        if (visited[i] == 0) {
            if (!dfs(i, adj, visited, st)) {
                return {}; // 存在环
            }
        }
    }

    vector<int> result;
    while (!st.empty()) {
        result.push_back(st.top());
        st.pop();
    }

    return result;
}

int main() {
    int n = 6;
    vector<vector<int>> graph = {{5,2}, {5,0}, {4,0}, {4,1}, {2,3}, {3,1}};

    vector<int> result = topologicalSortDFS(n, graph);

    if (result.empty()) {
        cout << "图中存在环，无法进行拓扑排序" << endl;
    } else {
        cout << "拓扑排序结果: ";
        for (int node : result) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

使用说明

1. **输入格式**:
   - `n`: 图中的节点数量
   - `graph`: 边的列表，每条边表示为`[u, v]`，表示从节点`u`指向节点`v`的有向边
2. **输出**:
   - 如果图是有向无环图(DAG)，返回一个拓扑排序序列
   - 如果图中存在环，返回空数组
3. **算法选择**:

- Kahn算法更直观，易于理解，且可以检测环
- DFS算法在某些情况下可能更高效，但实现稍复杂

4. **时间复杂度**：两种算法的时间复杂度均为O(V+E)，其中V是顶点数，E是边数

根据具体问题选择合适的算法实现拓扑排序。

ST表

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

class ST {
private:
    vector<vector<int>> st;
    vector<int> log;

public:
    ST(vector<int>& arr) {
        int n = arr.size();
        log.resize(n + 1);
        for (int i = 2; i <= n; i++) log[i] = log[i / 2] + 1;

        int k = log[n] + 1;
        st.resize(n, vector<int>(k));

        for (int i = 0; i < n; i++) st[i][0] = arr[i];
        for (int j = 1; j < k; j++)
            for (int i = 0; i + (1 << j) <= n; i++)
                st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j -
1]);
    }

    int query(int l, int r) {
        int j = log[r - l + 1];
        return max(st[l][j], st[r - (1 << j) + 1][j]);
    }
};

inline int read()
{
    int x=0,f=1;char ch=getchar();
    while (ch<'0'||ch>'9'){if (ch=='-') f=-1;ch=getchar();}
    while (ch>='0'&&ch<='9'){x=x*10+ch-48;ch=getchar();}
    return x*f;
}

signed main(){
    ios::sync_with_stdio(false);
    cout.tie(0);
```

```
    int n,m;
    n=read();
    m=read();
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        arr[i]=read();
    }
    ST st(arr);
    vector<int> ans;
    for(int i=0;i<m;i++){
        int l,r;
        l=read();
        r=read();
        ans.push_back(st.query(l-1,r-1));
    }
    for(auto i:ans){
        cout<<i<<"\n";
    }
    return 0;
}
```

树状数组

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

template<typename T>
class FenwickTree {
private:
    vector<T> tree;
    int n;

public:
    // 构造函数，初始化大小为n+1（树状数组下标从1开始）
    FenwickTree(int size) : n(size), tree(size + 1, 0) {}

    // 在位置pos增加value
    void update(int pos, T value) {
        for (; pos <= n; pos += pos & -pos) {
            tree[pos] += value;
        }
    }

    // 查询前缀和[1, pos]
    T query(int pos) {
        T sum = 0;
        for (; pos > 0; pos -= pos & -pos) {
            sum += tree[pos];
        }
```

```
        return sum;
    }

    // 查询区间和[l, r]
    T rangeQuery(int l, int r) {
        return query(r) - query(l - 1);
    }

    // 获取原始数组（仅供调试使用）
    vector<T> getOriginalArray() {
        vector<T> arr(n + 1, 0);
        for (int i = 1; i <= n; i++) {
            arr[i] = rangeQuery(i, i);
        }
        return arr;
    }
};

// 示例用法
int main() {
    vector<int> nums = {1, 2, 3, 4, 5};
    int n = nums.size();

    FenwickTree<int> fenwick(n);

    // 构建树状数组
    for (int i = 0; i < n; i++) {
        fenwick.update(i + 1, nums[i]);  // 注意：树状数组下标从1开始
    }

    // 测试查询
    cout << "前缀和[1,3]: " << fenwick.query(3) << endl;       // 输出: 6 (1+2+3)
    cout << "区间和[2,4]: " << fenwick.rangeQuery(2, 4) << endl; // 输出: 9 (2+3+4)

    // 测试更新
    fenwick.update(2, 3);  // 在位置2增加3
    cout << "更新后区间和[2,4]: " << fenwick.rangeQuery(2, 4) << endl; // 输出: 12 (5+3+4)

    return 0;
}
```

逆序对

归并排序

```
#include<bits/stdc++.h>
#define int long long
```

```cpp
using namespace std;

class SortWithSwapCount {
private:
    long long swapCount;  // 统计交换次数

    // 归并排序的合并过程
    void merge(vector<int>& arr, int left, int mid, int right) {
        vector<int> temp(right - left + 1);
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                // 当左半部分的元素大于右半部分的元素时
                // 需要将右半部分的元素移动到左半部分元素的前面
                // 这相当于进行了 (mid - i + 1) 次相邻交换
                swapCount += (mid - i + 1);
                temp[k++] = arr[j++];
            }
        }

        // 复制剩余元素
        while (i <= mid) {
            temp[k++] = arr[i++];
        }
        while (j <= right) {
            temp[k++] = arr[j++];
        }

        // 将临时数组复制回原数组
        for (int idx = 0; idx < k; idx++) {
            arr[left + idx] = temp[idx];
        }
    }

    // 归并排序递归函数
    void mergeSort(vector<int>& arr, int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }

public:
    SortWithSwapCount() : swapCount(0) {}

    // 排序并返回交换次数
    long long sortAndCountSwaps(vector<int>& arr) {
        swapCount = 0;
        if (arr.size() <= 1) return 0;
```

```
        mergeSort(arr, 0, arr.size() - 1);
        return swapCount;
    }
};

signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n;
    cin>>n;
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    SortWithSwapCount sorter;
    cout<<sorter.sortAndCountSwaps(arr)<<"\n";
    return 0;
}
```

树状数组

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

class SortWithSwapCount {
private:
    long long swapCount;  // 统计交换次数

    // 归并排序的合并过程
    void merge(vector<int>& arr, int left, int mid, int right) {
        vector<int> temp(right - left + 1);
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                // 当左半部分的元素大于右半部分的元素时
                // 需要将右半部分的元素移动到左半部分元素的前面
                // 这相当于进行了 (mid - i + 1) 次相邻交换
                swapCount += (mid - i + 1);
                temp[k++] = arr[j++];
            }
        }

        // 复制剩余元素
        while (i <= mid) {
```

```
            temp[k++] = arr[i++];
        }
        while (j <= right) {
            temp[k++] = arr[j++];
        }

        // 将临时数组复制回原数组
        for (int idx = 0; idx < k; idx++) {
            arr[left + idx] = temp[idx];
        }
    }

    // 归并排序递归函数
    void mergeSort(vector<int>& arr, int left, int right) {
        if (left >= right) return;

        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }

public:
    SortWithSwapCount() : swapCount(0) {}

    // 排序并返回交换次数
    long long sortAndCountSwaps(vector<int>& arr) {
        swapCount = 0;
        if (arr.size() <= 1) return 0;

        mergeSort(arr, 0, arr.size() - 1);
        return swapCount;
    }
};

signed main(){
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n;
    cin>>n;
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        cin>>arr[i];
    }
    SortWithSwapCount sorter;
    cout<<sorter.sortAndCountSwaps(arr)<<"\n";
    return 0;
}
```

归并排序

归并排序（递归+迭代）C++模板，适合算法竞赛使用：

```cpp
#include <vector>
#include <algorithm>
using namespace std;

// 递归版本归并排序
void merge_sort_recursive(vector<int>& arr, int l, int r) {
    if (l >= r) return;

    int mid = l + (r - l) / 2;
    merge_sort_recursive(arr, l, mid);
    merge_sort_recursive(arr, mid + 1, r);

    // 合并操作
    vector<int> temp(r - l + 1);
    int i = l, j = mid + 1, k = 0;

    while (i <= mid && j <= r) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= r) temp[k++] = arr[j++];

    for (int i = l, k = 0; i <= r; i++, k++) {
        arr[i] = temp[k];
    }
}

// 迭代版本归并排序（更推荐，避免递归开销）
void merge_sort_iterative(vector<int>& arr) {
    int n = arr.size();
    vector<int> temp(n);

    for (int seg = 1; seg < n; seg *= 2) {
        for (int l = 0; l < n; l += 2 * seg) {
            int mid = min(l + seg, n);
            int r = min(l + 2 * seg, n);

            int i = l, j = mid, k = l;
            while (i < mid && j < r) {
                if (arr[i] <= arr[j]) {
                    temp[k++] = arr[i++];
                } else {
                    temp[k++] = arr[j++];
                }
```

```
            }

            while (i < mid) temp[k++] = arr[i++];
            while (j < r) temp[k++] = arr[j++];

            for (k = l; k < r; k++) {
                arr[k] = temp[k];
            }
        }
    }
}

// 使用示例
int main() {
    vector<int> arr = {5, 2, 9, 1, 5, 6};

    // 递归版本调用
    // merge_sort_recursive(arr, 0, arr.size() - 1);

    // 迭代版本调用（推荐）
    merge_sort_iterative(arr);

    return 0;
}
```

模板特点：

1. **双版本提供**：包含递归和迭代两种实现
2. **边界处理**：严格处理数组边界，避免越界
3. **空间优化**：使用单个临时数组，减少内存分配
4. **稳定排序**：保持相等元素的相对顺序

复杂度分析：

- 时间复杂度：O(nlogn)
- 空间复杂度：O(n)

算法竞赛使用建议：

- 推荐使用迭代版本，避免递归栈溢出
- 可修改比较条件实现降序排序
- 可扩展为结构体/对象的排序
- 注意数组下标范围（0-indexed）

如果需要处理大数据量，建议使用迭代版本，递归深度过大会导致栈溢出。

求所有因数

分解素因数

```
vector<pair<int,int>> fact(int n){
    vector<pair<int,int>> ans;
    ans.push_back({1,0});
    for(auto p:primes){
        if(n==1)break;
        while(n%p==0){
            n/=p;
            if(ans.back().first!=p){
                ans.push_back({p,1});
            }else{
                ans.back().second++;
            }
        }
    }
    if(n!=1){
        ans.push_back({n,1});
    }
    return ans;
}
```

既然您已经有了分解质因数的板子，只需要生成所有因数的部分，这里提供几个简洁高效的实现：

方法一：递归实现（推荐）

```
// 递归生成所有因数
void generateFactors(const vector<pair<int, int>>& primeFactors, int index,
                     int current, vector<int>& result) {
    if (index == primeFactors.size()) {
        result.push_back(current);
        return;
    }

    int prime = primeFactors[index].first;
    int exponent = primeFactors[index].second;

    for (int i = 0; i <= exponent; i++) {
        generateFactors(primeFactors, index + 1, current, result);
        current *= prime;
    }
}

// 获取所有因数的主函数
vector<int> getAllFactors(const vector<pair<int, int>>& primeFactors) {
    vector<int> result;
    generateFactors(primeFactors, 0, 1, result);
    sort(result.begin(), result.end());
    return result;
}
```

方法二：迭代实现（更简洁）

```
vector<int> getAllFactors(const vector<pair<int, int>>& primeFactors) {
    vector<int> result = {1};

    for (auto& [prime, exp] : primeFactors) {
        int currentSize = result.size();
        int power = 1;

        for (int i = 1; i <= exp; i++) {
            power *= prime;
            for (int j = 0; j < currentSize; j++) {
                result.push_back(result[j] * power);
            }
        }
    }

    sort(result.begin(), result.end());
    return result;
}
```

方法三：极简版本（如果不需要排序）

```
vector<int> getAllFactors(const vector<pair<int, int>>& primeFactors) {
    vector<int> factors = {1};

    for (auto& p : primeFactors) {
        int prime = p.first, exp = p.second;
        vector<int> newFactors;
        int power = 1;

        for (int i = 0; i <= exp; i++) {
            for (int f : factors) {
                newFactors.push_back(f * power);
            }
            power *= prime;
        }
        factors = move(newFactors);
    }

    return factors;
}
```

完整使用示例：

```
// 假设您已经有了 primeFactors
vector<pair<int, int>> primeFactors = {{2, 2}, {3, 1}}; // 12 = 2² × 3¹
```

```
// 获取所有因数
vector<int> factors = getAllFactors(primeFactors);

// 输出结果
for (int factor : factors) {
    cout << factor << " ";
}
// 输出: 1 2 3 4 6 12
```

性能分析：

- **时间复杂度**：O(d(n))，其中 d(n) 是 n 的因数个数
- **空间复杂度**：O(d(n))
- **优势**：避免了从 1 到 n 的遍历，直接利用质因数分解结果高效生成

推荐使用方法二（迭代实现），因为它代码简洁，没有递归开销，且容易理解。

笛卡尔树

以下是一个C++的笛卡尔树板子，包含构建和基本操作：

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

struct TreeNode {
    int val;
    int index;  // 在原数组中的索引
    TreeNode* left;
    TreeNode* right;
    TreeNode(int v, int idx) : val(v), index(idx), left(nullptr),
right(nullptr) {}
};

class CartesianTree {
private:
    TreeNode* root;

    // 构建笛卡尔树
    TreeNode* build(const vector<int>& arr) {
        if (arr.empty()) return nullptr;

        stack<TreeNode*> stk;

        for (int i = 0; i < arr.size(); i++) {
            TreeNode* node = new TreeNode(arr[i], i);
            TreeNode* last = nullptr;
```

```
            // 弹出比当前节点值大的节点
            while (!stk.empty() && stk.top()->val > arr[i]) {
                last = stk.top();
                stk.pop();
            }

            // 如果栈不为空，当前节点作为栈顶节点的右孩子
            if (!stk.empty()) {
                stk.top()->right = node;
            }
            // 否则当前节点作为根节点
            else {
                root = node;
            }

            // 最后一个弹出的节点作为当前节点的左孩子
            node->left = last;
            stk.push(node);
        }

        return root;
    }

    void destroy(TreeNode* node) {
        if (!node) return;
        destroy(node->left);
        destroy(node->right);
        delete node;
    }

public:
    CartesianTree(const vector<int>& arr) {
        root = build(arr);
    }

    ~CartesianTree() {
        destroy(root);
    }

    // 获取根节点
    TreeNode* getRoot() {
        return root;
    }

    // 前序遍历（用于验证）
    void preorder(TreeNode* node) {
        if (!node) return;
        cout << "(" << node->val << ", idx:" << node->index << ") ";
        preorder(node->left);
        preorder(node->right);
    }

    // 中序遍历（应该得到原数组顺序）
    void inorder(TreeNode* node) {
```

```
        if (!node) return;
        inorder(node->left);
        cout << node->val << " ";
        inorder(node->right);
    }

    // 获取树的高度
    int height(TreeNode* node) {
        if (!node) return 0;
        return max(height(node->left), height(node->right)) + 1;
    }
};

// 使用示例
int main() {
    vector<int> arr = {9, 3, 7, 1, 8, 12, 10, 20, 15, 18, 5};

    CartesianTree tree(arr);

    cout << "前序遍历（笛卡尔树结构）: ";
    tree.preorder(tree.getRoot());
    cout << endl;

    cout << "中序遍历（原数组顺序）: ";
    tree.inorder(tree.getRoot());
    cout << endl;

    cout << "树的高度: " << tree.height(tree.getRoot()) << endl;

    return 0;
}
```

笛卡尔树的特点：

1. 中序遍历得到原数组
2. 满足堆性质（通常是最小堆）
3. 任意节点的值小于其子树中的所有节点

时间复杂度：

- 构建：O(n)
- 空间：O(n)

主要应用：

- 范围最小值查询(RMQ)
- 直方图最大矩形面积
- 序列分析等

这个板子提供了笛卡尔树的基本构建和遍历功能，你可以根据具体需求扩展其他操作。

李超线段树

/

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

const int N = 1e6 + 5;
const ll INF = 1e18;

// 李超线段树 - 维护最大值
struct MaxLiChaoTree {
    struct Line {
        ll k, b;
        Line() : k(0), b(-INF) {}
        Line(ll _k, ll _b) : k(_k), b(_b) {}
        ll operator()(ll x) const { return k * x + b; }
    };

    vector<Line> tree;
    int n;

    MaxLiChaoTree(int _n) : n(_n) {
        tree.resize(4 * n);
    }

    void insert(int node, int l, int r, Line line) {
        if (l == r) {
            if (line(l) > tree[node](l)) tree[node] = line;
            return;
        }

        int mid = (l + r) / 2;
        if (line(mid) > tree[node](mid)) {
            swap(tree[node], line);
        }

        if (line(l) > tree[node](l)) {
            insert(2 * node, l, mid, line);
        } else if (line(r) > tree[node](r)) {
            insert(2 * node + 1, mid + 1, r, line);
        }
    }

    void insert(ll k, ll b) {
        insert(1, 1, n, Line(k, b));
    }

    ll query(int node, int l, int r, int x) {
        if (l == r) return tree[node](x);

        int mid = (l + r) / 2;
        ll res = tree[node](x);
        if (x <= mid) {
```

```
            res = max(res, query(2 * node, l, mid, x));
        } else {
            res = max(res, query(2 * node + 1, mid + 1, r, x));
        }
        return res;
    }

    ll query(int x) {
        return query(1, 1, n, x);
    }
};

// 李超线段树 - 维护最小值
struct MinLiChaoTree {
    struct Line {
        ll k, b;
        Line() : k(0), b(INF) {}
        Line(ll _k, ll _b) : k(_k), b(_b) {}
        ll operator()(ll x) const { return k * x + b; }
    };

    vector<Line> tree;
    int n;

    MinLiChaoTree(int _n) : n(_n) {
        tree.resize(4 * n);
    }

    void insert(int node, int l, int r, Line line) {
        if (l == r) {
            if (line(l) < tree[node](l)) tree[node] = line;
            return;
        }

        int mid = (l + r) / 2;
        if (line(mid) < tree[node](mid)) {
            swap(tree[node], line);
        }

        if (line(l) < tree[node](l)) {
            insert(2 * node, l, mid, line);
        } else if (line(r) < tree[node](r)) {
            insert(2 * node + 1, mid + 1, r, line);
        }
    }

    void insert(ll k, ll b) {
        insert(1, 1, n, Line(k, b));
    }

    ll query(int node, int l, int r, int x) {
        if (l == r) return tree[node](x);

        int mid = (l + r) / 2;
```

```
        ll res = tree[node](x);
        if (x <= mid) {
            res = min(res, query(2 * node, l, mid, x));
        } else {
            res = min(res, query(2 * node + 1, mid + 1, r, x));
        }
        return res;
    }

    ll query(int x) {
        return query(1, 1, n, x);
    }
};

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    MaxLiChaoTree max_tree(n);
    MinLiChaoTree min_tree(n);

    while (m--) {
        int op;
        scanf("%d", &op);

        if (op == 0) {
            ll k, b;
            scanf("%lld%lld", &k, &b);
            max_tree.insert(k, b);
            min_tree.insert(k, b);
        } else {
            int x;
            scanf("%d", &x);
            ll max_val = max_tree.query(x);
            ll min_val = min_tree.query(x);
            printf("%lld %lld\n", max_val, min_val);
        }
    }

    return 0;
}
```