```
ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);
```

memset设置最值:

```
// 设置最小值
memset(a, 128, sizeof(a));
// 设置最大值（2139062143，即0x7f7f7f7f）
memset(a, 127, sizeof(a));
```

算法基础

前缀和

一维前缀和

```
struct NumArray
{
    int nums[n];
    int preSum[n + 1];
    NumArray()
    {
        for (int i = 1; i <= n; i++)
            preSum[i] = preSum[i - 1] + nums[i - 1];
    }
    // 下标从1开始
    int sumRange(int left, int right)
    {
        return preSum[right + 1] - preSum[left];
    }
};
```

二维前缀和

```
struct NumMatrix
{
    int preSum[m + 1][n + 1];
    int matrix[m][n];
    NumMatrix()
```

```
    {
        for (int i = 1; i <= m; i++)
            for (int j = 1; j <= n; j++)
                preSum[i][j] = preSum[i - 1][j] + preSum[i][j - 1] + matrix[i - 1]
[j - 1] - preSum[i - 1][j - 1];
    }
    int sumRegion(int x1, int y1, int x2, int y2)
    {
        return preSum[x2 + 1][y2 + 1] - preSum[x1][y2 + 1] - preSum[x2 + 1][y1] +
preSum[x1][y1];
    }
};
```

差分

```
struct NumArray
{
    int nums[n];
    int diff[n];
    NumArray()
    {
        diff[0] = nums[0];
        for (int i = 1; i < n; i++)
            diff[i] = nums[i] - nums[i - 1];
    }
};
```

滑动窗口

```
string slideWindow(string s, string t)
{
    unordered_map<char, int> need;
    unordered_map<char, int> window;
    for (char c : t)
        need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    int start = 0, len = INT_MAX;
    while (right < s.length())
    {
        char c = s[right];
```

```
        ++right;
        // 更新右窗口
        if (need.count(c))
        {
            window[c]++;
            if (window[c] == need[c])
                ++valid;
        }
        // 判断左侧是否需要收缩
        while (valid == need.size())
        {
            if (right - left < len)
            {
                start = left;
                len = right - left;
            }
            // 将移出的字符
            char d = s[left];
            ++left;
            if (need.count(d))
            {
                if (window[d] == need[d])
                    --valid;
                window[d]--;
            }
        }
    }
    return len == INT_MAX ? "" : s.substr(start, len);
}
```

二分搜索

基本二分搜索

```
int binarySearch(vector<int>& nums, int target)
{
    int left = 0, right = nums.size() - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
```

```
        else if (nums[mid] > target)
            right = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

寻找左边界

```
int left_bound(vector<int>& nums, int target)
{
    int left = 0, right = 0;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (nums[left] < target)
            left = mid + 1;
        else if (nums[left] > target)
            right = mid - 1;
        else
            // 锁定左边界
            right = mid - 1;
    }
    // target大于所有数
    if (left == nums.size())
        return -1;
    return nums[left] == target ? left : -1;
}
```

寻找右边界

```
int right_bound(vector<int>& nums, int target)
{
    int left = 0, right = 0;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid - 1;
```

```
        else if (nums[mid] == target)
            // 锁定右边界
            left = mid + 1;
    }
    if (left - 1 < 0)  return -1;
    return nums[left - 1] == target ? (left - 1) : -1;
}
```

数据结构

单调栈

```
int main()
{
    int n;
    cin >> n;
    vector<int> nums(n), res(n);

    for (int i = 0; i < n; i++)
        cin >> v[i];
    stack<int> stk;
    for (int i = n - 1; i >= 0; i--)
    {
        while (!stk.empty() && s.peek() <= nums[i])
        {
            stk.pop();
        }
        res[i] = s.empty() ? -1 : s.peek();
        stk.push(nums[i]);
    }
    for (int i = 0; i < n; i++)
        cout << res[i] << " ";

    return 0;
}
```

单调队列

```
struct MonotonicQueue
{
    deque<int> q;
```

```
    void push(int n)
    {
        while (!q.empty() && q.back() < n)
        {
            q.pop_back();
        }
        q.push_back(n);
    }
    int getMax() { return q.front(); }
    // 在队头删除元素n
    void pop(int n)
    {
        if (n == q.front())  q.pop_front();
    }
};
```

二叉树

构造二叉树

前序+中序

```
TreeNode* build(int preStart, int preEnd, int inStart, int inEnd)
{
    if (inStart > inEnd)
        return nullptr;
    int rootVal = preorder[preStart];
    int index = valToIndex[rootVal];
    int leftSize = index - inStart;
    TreeNode* root = new TreeNode(rootVal);
    root->left = build(preStart + 1, preStart + leftSize, inStart, index - 1);
    root->right = build(preStart + leftSize + 1, preEnd, index + 1, inEnd);
}
```

中序+后序

```
TreeNode* build(int inStart, int inEnd, int postStart, int postEnd)
{
    if (inStart > inEnd)
        return nullptr;
    int rootVal = postorder[postEnd];
    int index = valToIndex[rootVal];
    int leftSize = index - inStart;
    TreeNode* root = new TreeNode(rootVal);
    root->left = build(inStart, index - 1, postStart, postStart + leftSize - 1);
    root->right = build(index + 1, inEnd, postStart + leftSize, postEnd - 1);
    return root;
}
```

堆

大根堆

```
class MaxPQ
{
private:
    int* pq;
    int size = 0;
public:
    MaxPQ(int cap)
    {
        pq = new int[cap + 1];
    }
    int max()
    {
        return pq[1];
    }
    void insert(int e)
    {
        ++size;
        pq[size] = e;
        swim(size);
    }
    int delMax()
    {
        int maxE = pq[1];
        swap(pq[1], pq[size]);
        --size;
```

```
        sink(1);
        return maxE;
    }
private:
    int parent(int root)
    {
        return root / 2;
    }
    int left(int root)
    {
        return root * 2;
    }
    int right(int root)
    {
        return root * 2 + 1;
    }
    void swim(int x)
    {
        while (x > 1 && pq[parent(x)] < pq[x])
        {
            swap(pq[parent(x)], pq[x]);
            x = parent(x);
        }
    }
    void sink(int x)
    {
        while (left(x) <= size)
        {
            int max = left(x);
            if (right(x) <= size && pq[max] < pq[right(x)])
                max = right(x);
            if (pq[max] < pq[x])  break;
            swap(pq[max], pq[x]);
            x = max;
        }
    }
};
```

小根堆

```
class MinPQ
{
private:
    int* pq;
    int size = 0;
public:
    MinPQ(int cap)
    {
        pq = new int[cap + 1];
    }
    int min()
    {
        return pq[1];
    }
    void insert(int e)
    {
        ++size;
        pq[size] = e;
        swim(size);
    }
    int delMin()
    {
        int minE = pq[1];
        swap(pq[1], pq[size]);
        --size;
        sink(1);
        return minE;
    }
private:
    int parent(int root)
    {
        return root / 2;
    }
    int left(int root)
    {
        return root * 2;
    }
    int right(int root)
    {
```

```
        return root * 2 + 1;
    }
    void swim(int x)
    {
        while (x > 1 && pq[parent(x)] > pq[x])
        {
            swap(pq[parent(x)], pq[x]);
            x = parent(x);
        }
    }
    void sink(int x)
    {
        while (left(x) <= size)
        {
            int min = left(x);
            if (right(x) <= size && pq[min] > pq[right(x)])
                min = right(x);
            if (pq[min] > pq[x])  break;
            swap(pq[min], pq[x]);
            x = max;
        }
    }
};
```

图论

并查集

```
int parent[505];
void init(int n)
{
    for (int i = 1; i <= n; i++)
        parent[i] = i;
}
int find(int x)
{
    if (parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
}
void unite(int x, int y)
```

```
{
    int rootx = find(x);
    int rooty = find(y);
    if (rootx == rooty)
        return;
    parent[rooty] = rootx;
}
bool is_connected(int x, int y)
{
    return find(x) == find(y);
}
```

最短路

Dijkstra

```
struct Dijkstra
{
    vector<int> dist(n);
    vector<vector<pair<int, int>>> g(n);

    Dijkstra(int n)
    {
        dist.resize(n + 5, INT_MAX);
        g.resize(n + 5);
    }
    // 添加有向边
    void addEdge(int from, int to, int value)
    {
        g[from].push_back({to, value});
    }
    // 获取从start到其他点的最短路径长度
    void getDist(int start)
    {
        dist[start] = 0;
        // 路径长度, 当前节点
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> q;
        q.push({0, start});
        while (!q.empty())
        {
            auto [u, t] = q.top();
            q.pop();
```

```
            if (u > dist[t]) continue;
            for (auto [v, w] : g[t])
            {
                if (u + w < dist[v])
                {
                    dist[v] = u + w;
                    q.push({u + w, v});
                }
            }
        }
    }
};

int main()
{
    int n, m, s, t;
    cin >> n >> m >> s >> t;
    Dijkstra D(n);
    while (m--)
    {
        int u, v, w;
        cin >> u >> v >> w;
        D.addEdge(u, v, w);
        D.addEdge(v, u, w);
    }
    D.getDist(s);
    cout << D.dist[t] << endl;
    return 0;
}
```

最小生成树

Prim

```
struct MST
{
    vector<vector<int> > edges;
    vector<int> parent;
    int n;
    MST(int n)
    {
        this.n = n;
```

```
        for (int i = 0; i <= n + 5; i++)
            parent.push_back(i);
    }
    int find(int x)
    {
        if (x != parent[x])
            parent[x] = find(parent[x]);
        return x;
    }
    void add(int u, int v, int w)
    {
        edges.push_back({u, v, w});
    }
    int getMST()
    {
        int cnt = 0, res = 0;
        sort(edges.begin(), edges.end(), [](vector<int>& a, vector<int>& b)
            {
                return a[2] < b[2];
            });
        for (int i = 0; i < edges.size(); i++)
        {
            int u = edges[i][0];
            int v = edges[i][1];
            int w = edges[i][2];
            if (find(u) != find(v))
            {
                parent[find(u)] = parent(v);
                ++cnt;
                res += w;
                // 需要n - 1条边
                if (cnt == n - 1)
                    return res;
            }
        }
        // 无最小生成树
        return 0;
    }
};

int main()
```

```
{
    int n, m;
    cin >> n >> m;
    MST tree(n);
    while (m--)
    {
        int u, v, w;
        cin >> u >> v >> w;
        tree.add(u, v, w);
    }
    cout << tree.getMST() << res;
}
```

数学

GCD LCM

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

int lcm(int a, int b)
{
    return a * b / gcd(a, b);
}
```

快速幂

```
int pow(int a, int k)
{
    int res = 1;
    while (k)
    {
        if (k & 1)
            res = (long long) res * a % mod;
        a = (long long) a * a % mod;
        b >= 1;
    }
    return res % mod;
}
```

质数筛

埃氏筛

```cpp
// 时间复杂度O(nloglogn)
vector<int> Eratosthenes(int n)
{
    vector<int> prime;
    vector<bool> isPrime(n + 1, true);
    for (int i = 2; i <= n; i++)
    {
        if (isPrime[i])
        {
            prime.push_back(i);
            for (int j = 2; j <= n; j += i)  isPrime[j] = false;
        }
    }
    return prime;
}
```

欧拉筛

```cpp
// 时间复杂度O(n)
vector<int> Euler(int n)
{
    vector<int> prime;
    vector<bool> notPrime(n + 1);
    for (int i = 2; i <= n; i++)
    {
        if (!notPrime[i])  prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] <= n; j++)
        {
            notPrime[i * prime[j]] = true;
            if (i % prime[j] == 0)  break;
        }
    }
    return prime;
}
```