NJIT ICPC Team Notebook (2023)

Contents

1 Contest

1.1 Template

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

using ll = long long;
using pi = pair<int,int>;
using pl = pair<ll,ll>;
using vi = vector<int>;
using vll = vector<ll>;
using iii = tuple<int, int, int>;

#define PB push_back
#define MP make_pair
#define LSOne(S) ((S) & -(S))
#define sz(x) int((x).size())
#define all(x) begin(x), end(x)
#define tcT template<class T

#define FOR(i,a,b) for(int i=(a),_b=(b); i<=_b; i++)
#define FORD(i,a,b) for(int i=(a),_b=(b); i>=_b; i--)
#define REP(i,a) for(int i=0,_a=(a); i<_a; i++)
#define DEBUG(x) { cout << #x << " = "; cout << (x) << endl; }
#define PR(a,n) { cout << #a << " = "; FOR(_,1,n) cout << a[_] << ' '; cout << endl; }
#define PR0(a,n) { cout << #a << " = "; REP(_,n) cout << a[_] << ' '; cout << endl; }

// const int dx[4]{1,0,-1,0}, dy[4]{0,1,0,-1}; // for every grid problem!!

tcT> using pqg = priority_queue<T,vector<T>,greater<T>>; // minheap

tcT> bool ckmin(T& a, const T& b) {
        return b < a ? a = b, 1 : 0; } // set a = min(a,b)
tcT> bool ckmax(T& a, const T& b) {
        return a < b ? a = b, 1 : 0; } // set a = max(a,b)

tcT> T gcd(T a, T b){ T r; while (b != 0) { r = a % b; a = b; b = r; } return a;}
tcT> T lcm(T a, T b) { return a / gcd(a, b) * b; }

ll cdiv(ll a, ll b) { return a/b+((a^b)>0&&a%b); } // divide a by b rounded up
ll fdiv(ll a, ll b) { return a/b-((a^b)<0&&a%b); } // divide a by b rounded down

tcT> T gcd(T a, T b){ T r; while (b != 0) { r = a % b; a = b; b = r; } return a;}
tcT> T lcm(T a, T b) { return a / gcd(a, b) * b; }

// bitwise ops
constexpr int pct(int x) { return __builtin_popcount(x); } // # of bits set
constexpr int bits(int x) { // assert(x >= 0);
        return x == 0 ? 0 : 31-__builtin_clz(x); } // floor(log2(x))
constexpr int p2(int x) { return 1<<x; }
constexpr int msk2(int x) { return p2(x)-1; }

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);


    return 0;
}
```

1.2 .Bashrc

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
        -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>

g++ -std=c++17 -O2 name.cpp -o name -Wall

open ~/.zshrc and copy:
co() { g++ -std=c++17 -O2 -o "${1%.*}" $1 -Wall; }
run() { co $1 && ./${1%.*} & fg; }

run name.cpp // compile and run
```

1.3 Troubleshoot

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
```

```
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

1.4 Technical Note

```
Submission :
All program source code files and/or test data files which you create must be located in or
beneath your home directory. Your home directory will normally be named /home/team
but this may vary between sites so check with your site coordinator. You may create
subdirectories beneath your home directory.

If your program exits with a non-zero exit code, it will be judged as a Run Time Error.
-----------
Compiling :
1. For C++:
compileg++ progname.cpp (compiling)
To execute a C/C++ program after compiling it as above, type the command : ./a.out

2. For Java:
compilejava Progname.java
runjava Progname

3. For python3
compilepython3 progname.py
runpython3 progname.py
----------
IDE: They can be accessed using the Applications Programming menu.
Other editors: They can be accessed using the Applications Accessories menu.
```

2 Data Structures

2.1 DSU

```cpp
/**
 * Description: Disjoint-set data structure.
 * Time: O(alpha(N)), almost constant
 */

struct DSU {
    vi lab;
    DSU(int n) { lab = vi(n, -1); }

    // get representive component (uses path compression)
    int get(int x) { return lab[x] < 0 ? x : lab[x] = get(lab[x]); }

    bool same_set(int a, int b) { return get(a) == get(b); }

    bool unite(int x, int y) {  // union by size
                x = get(x), y = get(y);
                if (x == y) return false;
                if (lab[x] > lab[y]) swap(x, y);
                lab[x] += lab[y];
                lab[y] = x;
                return true;
        }
};

int main() {
    DSU dsu(node_num);
    return 0;
}
```

2.2 Fenwick Tree

```cpp
/**
 * Description: Computes partial sums a[0] + a[1] + ... + a[pos - 1],
 * and updates single elements a[i],
 * taking the difference between the old and new value.
 * Time: Both operations are $O(log N)$.
 */

// 0 based index
template<
    typename T
> struct FT {
    FT(int _n) : n(_n), f(_n + 1) {}

    // a[u] += val
    void update(int u, T val) {
        assert(0 <= u && u < n);
        ++u;
        for (; u <= n; u += u & -u) {
            f[u] += val;
        }
    }

    // return a[0] + .. + a[u-1]
    T get(int u) const {
        assert(0 <= u && u <= n);
        T res = 0;
        for (; u > 0; u -= u & -u) {
            res += f[u];
        }
        return res;
    }

    // return a[l] + .. + a[r-1]
    T get(int l, int r) const {
        assert(0 <= l && l <= r && r <= n);
        if (l == r) return 0;  // empty
        return get(r) - get(l);
    }

    void reset() {
        std::fill(f.begin(), f.end(), T(0));
    }

    int lower_bound(T sum) {// min pos st sum of [0, pos] >= sum
                // Returns n if no sum is >= sum, or -1 if empty sum is.
                if (sum <= 0) return -1;
                int pos = 0;
                for (int pw = 1 << 25; pw; pw >>= 1) {
                        if (pos + pw <= sz(f) && f[pos + pw-1] < sum)
                                pos += pw, sum -= f[pos-1];
                }
                return pos;
        }

    int n;
    vector<T> f;
};

int main () {
    FT<ll> ft(n);
}
```

2.3 Lazy Segment Tree

```cpp
/**
 * Description: Segment Tree with lazy propagation
 */

// everything should be 1-based index

// 1 x y val: Increase all elements in [x, y] by val
// 2 x y: max value in [x, y]

struct SegTree {
    int n;  // size of A
    vi A, st, lazy;

    SegTree(int sz) : n(sz), st(4 * n), lazy(4 * n, 0) {}

    SegTree(const vi& initialA, int a_size): SegTree(a_size) {
        A = initialA;
        build(1, 1, n);
    }

    void build(int id, int l, int r) { // O(n)
        if (l == r) {
            st[id] = A[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(2 * id, l, mid);
        build(2 * id + 1, mid + 1, r);
        st[id] = max(st[2 * id], st[2 * id + 1]);
    }

    // propagate
    void fix(int id, int l, int r) {
        if (!lazy[id]) return;
        st[id] += lazy[id]; // careful about how to update info from lazy to node
```

```
        if (l != r){ // if not a leaf then propagate downwards
            lazy[2 * id] += lazy[id];
            lazy[2 * id + 1] += lazy[id];
        }
        else {
            A[l] += lazy[id];
        }

        lazy[id] = 0; // erase lazy flag
    }

    void update(int id, int l, int r, int u, int v, int val) { // O(log n)
        fix(id, l, r);
        if (l > v || r < u) return;
        if (l >= u && r <= v) {
            lazy[id] += val;
            fix(id, l, r);
            return;
        }
        int mid = (l + r) >> 1;
        update(2 * id, l, mid, u, v, val);
        update(2 * id + 1, mid + 1, r, u, v, val);
        st[id] = max(st[2 * id], st[2 * id + 1]);
    }

    long long get(int id, int l, int r, int u, int v) { // O(log n)
        fix(id, l, r);
        if (l >  v || r <  u) return -inf;
        if (l >= u && r <= v) return st[id];

        int mid = (l + r) >> 1;
        long long get1 = get(2 * id, l, mid, u, v);
        long long get2 = get(2 * id + 1, mid + 1, r, u, v);
        return max(get1, get2);
    }

    void update(int u, int v, int val) { update(1, 1, n, u, v, val); }
    long long get(int u, int v) { return get(1, 1, n, u, v); }
};

vll A(n + 1, 0);
SegTree st(A, n);
```

2.4 Trie

```
/**
 * Description: Prefix tree
 * Time: Both operations are O(string length).
 */

struct Trie {
    struct Node {
        Node* child[26];
        int exist, cnt;

        Node() {
            for (int i = 0; i < 26; i++) child[i] = NULL;
            exist = cnt = 0;
        }
    };

    int cur;
    Node* root;
    Trie() : cur(0) {
        root = new Node();
    };

    void add_string(string s) {
        Node* p = root;
        for (auto f : s) {
            int c = f - 'a';
            if (p->child[c] == NULL) p->child[c] = new Node();
            p = p->child[c];
            p->cnt++;
        }
        p->exist++;
    }

    bool delete_string_recursive(Node* p, string& s, int i) {
        if (i != (int)s.size()) {
            int c = s[i] - 'a';
            bool isChildDeleted = delete_string_recursive(p->child[c], s, i + 1);
            if (isChildDeleted) p->child[c] = NULL;
        }
        else p->exist--;

        if (p != root) {
            p->cnt--;
            if (p->cnt == 0) {
                delete(p);
                return true;
            }
        }
        return false;
    }

    void delete_string(string s) {
        if (find_string(s) == false) return;

        delete_string_recursive(root, s, 0);
    }

    bool find_string(string s) {
        Node* p = root;
        for (auto f : s) {
            int c = f - 'a';
            if (p->child[c] == NULL) return false;
            p = p->child[c];
        }
        return (p->exist != 0);
    }

    void dfs(Node* p, string& cur_string, vector<string>& res) {
        for (int i = 0; i < 26; i++) {
            if (p->child[i] != NULL) {
                cur_string += char(i + 'a');
                dfs(p->child[i], cur_string, res);
                cur_string.pop_back();
            }
        }
    }

    vector<string> sorted_strings() {
        vector<string> res;
        string current_string = "";
        dfs(0, current_string, res);
        return res;
    }
};
```

2.5 Sparse Table

```
/**
 * Description: Sparse Table. LG la so lon nhat thoa 2^LG < N.
 * vi du: N = 10^5 thi LG = 16 vi 2^16 = 65536
 */


// a is 1-based index
int a[N], st[LG + 1][N];
void preprocess() {
    for (int i = 1; i <= n; ++i) st[0][i] = a[i];
    for (int j = 1; j <= LG; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            st[j][i] = min(st[j - 1][i], st[j - 1][i + (1 << (j - 1))]);
}
int queryMin(int l, int r) {
    int k = __lg(r - l + 1);
    return min(st[k][l], st[k][r - (1 << k) + 1]);
}

// Preprocessing: O(NlogN)
// query: O(1)
// __lg(x) = floor(log_2(x)) while log2(x) return double
```

2.6 Small To Large Merging

```
/**
 * Description: Small-to-large merging technique to speed up the process of merging to data structure
 */

/*
Note that swap exchanges two sets in O(1) time. Thus, merging a smaller set of size m into
the larger one of size n takes O(mlogn) time. We can also merge other standard library data structures
such as std::map or std:unordered_map in the same way. However, std::swap does not always
run in O(1) time. For example, swapping std::array s takes time linear in the sum of the sizes
of the arrays, and the same goes for GCC policy-based data structures such as
__gnu_pbds::tree or __gnu_pbds::gp_hash_table. To swap two policy-based data structures a and b
in O(1), use a.swap(b) instead. Note that for standard library data structures, swap(a,b)
is equivalent to a.swap(b) .
*/
```

```
// Always merge smaller set to larger set
if (a.size() < b.size()) swap(a, b);
for (int x : b) a.insert(x);
```

3 Graph

3.1 Dijkstra

```
/**
 * Description: Dijkstra algorithm for finding SSSP
 * Time: O(Mlog N)
 */

void dijkstra(int n, int s, vector<vector<Edge>> &E, vector<long long> &D, vector<int> &trace) {
    D.resize(n, INF);
    trace.resize(n, -1);
    vector<bool> P(n, 0);

    D[s] = 0;
    priority_queue<pi, vector<pi>, greater<pi>> pq;
    pq.push({0, s});
    while (!pq.empty()) {
        int u = pq.top().second;
        int du = pq.top().first;
        pq.pop();
        if (du != D[u]) continue; // node is already processed or equivalently processed[u] = true
        for (auto e : E[u]) {
            int v = e.v;
            long long w = e.w;

            if (D[v] > D[u] + w) {
                D[v] = D[u] + w;
                pq.push({v, D[v]});
                trace[v] = u;
            }
        }
    }
}

/*
Trick: To find all shortest paths of all nodes to a node A. Simply reverse
the orginal graph and then run dijkstra from node A. Useful when finding shortest path
given the condition that an edge is fixed.
*/
```

3.2 BellmanFord

```
/**
 * Description: Bellman-Frod
 * Time: O(MN)
 */

struct Edge {
    int u, v, w;
    Edge(int u = 0, int v = 0, int w = 0) {
        this->u = u;
        this->v = v;
        this->w = w;
    }
};

vi d(maxn, INF);
vi path(maxn, -1);
vector<Edge> g;
int n, m;

bool BellmanFord(int s) {
    int u, v, w;
    d[s] = 0;
    for (int i = 1; i <= n - 1; i++) {
        for (int j = 0; j < m; j++) {
            u = g[j].u;
            v = g[j].v;
            w = g[j].w;
            if (d[u] != INF && d[u] + w < d[v]) {
                d[v] = d[u] + w;
                path[v] = u;
            }
        }

    // check if graph contains a negative cycle
    for (int i = 0; i < m; i++) {
        u = g[i].u; v = g[i].v; w = g[i].w;
        if (d[u] != INF && d[u] + w < d[v]) {
            return false;
        }
    }
    return true;
}

vector<int> trace_path(int s, int t) {
    if (t != s && path[t] == -1) return vi(0);

    vi res;
    while (t != -1) {
        res.push_back(t);
        t = path[t];
    }
    reverse(res.begin(), res.end());
    return res;
}

bool findNegativeCycle(vector<int> &negCycle) {
    // after running Bellman-ford n - 1 times
    int negStart = -1;
    for (auto E: g) {
        int u = E.u;
        int v = E.v;
        int w = E.w;
        if (d[u] != INF && d[v] > d[u] + w) {
            d[v] = -INF;
            path[v] = u;
            negStart = v;
        }
    }

    if (negStart == -1) return false; // no negative cycle

    int u = negStart;
    // To get the vertices that are guaranteed to lie in a negative cycle
    for (int i = 0; i < n; i++) {
        u = path[u];
    }

    negCycle = vector<int>(1, u);
    for (int v = path[u]; v != u; v = path[u]) {
        negCycle.push_back(v); // truy vet mot dong
    }
    reverse(negCycle.begin(), negCycle.end());

    return true;
}

int main() {
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        g.push_back(Edge(u, v, w));
    }

    int s = 0;
    bool res = BellmanFord(s);

    if (!res) {
        // if there is a negative cycle, identify all nodes that does not exist shortest path
        for (int i = 0; i < n; i++) {
            for (auto E : g) {
                int u = E.u;
                int v = E.v;
                int w = E.w;
                if (d[u] != INF && d[v] > d[u] + w) {
                    d[v] = -INF; path[v] = u;
                }
            }
        }
    }

    return 0;
}
```

3.3 Floyd Warshall

```
/**
 * Description: Floyd warshall
```

```
 * Time: O(N^3)
 */

void init_trace(vector<vector<int>> &trace) {
    int n = trace.size();
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            trace[u][v] = u;
        }
    }
}

bool floydWarshall(int n, vector<vector<long long>> &w, vector<vector<long long>> &D, vector<vector<
    int>> &trace) {
    D = w; // if d[i][i] = 0, d[i][j] = INF if no edge
    init_trace(trace); // neu can do duong di

    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                // if the graph has negative weight edges, it is better to write the Floyd-Warshall
                      algorithm in the following way, so that it does not perform transitions using
                      paths that don't exist.
                if (D[i][k] < INF && D[k][j] < INF) {
                    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
                    trace[i][j] = trace[k][j];

                    // make sure that D[i][j] wont be overflow when graph contains negative cycles
                    D[i][j] = max(D[i][j], -INF);
                }
            }
        }
    }

    // check if graph contain negative cycle
    for (int i = 0; i < n; i++) {
        if (D[i][i] < 0) return false;
    }
    return true;
}

vector<int> trace_path(vector<vector<int>> &trace, int u, int v) {
    vector<int> path;
    while (v != u) { // truy vet nguoc tu v ve u
        path.push_back(v);
        v = trace[u][v];
    }
    path.push_back(u);

    reverse(path.begin(), path.end()); // can reverse vi duong di tu v nguoc ve u

    return path;
}
```

3.4 BFS

```
/**
 * Description: BFS
 * Time: O(V + E)
 */

void BFS(int s) {
    queue<int> q;
    visited[s] = true;
    q.push(s);

    while (!q.empty()){
        int u = q.front();
        q.pop();
        for (auto v : g[u]){
            if (!visited[v]){
                visited[v] = true;
                d[v] = d[u] + 1;
                path[v] = u;
                q.push(v);
            }
        }
    }
}

// Tracing
void printPath(int u) {
    if (!visited[u]) cout << "No path!";
    else {
        vi path;
        for (int v = u; v != -1; v = path[v]){
            path.push_back(v);
        }
        reverse(path.begin(), path.end());
        cout << "Path: ";
        for (auto v: path) cout << v << " ";
    }
}
```

3.5 BFS 0-1

```
/**
 * Description: 0-1 BFS, find shortest path in 0-1 weighted graph.
 * Time: better than Dijkstra
 */

// 0-1 BFS could be use to find the minimum of edges that is needed to
// be reversed in direction to make the path 1->N possible

int n, m;
const int inf = 1e9;
const int maxn = 1e5 + 7;
vector<pi> g[maxn];
int d[maxn];

void sssp(int s) {
    fill_n(d, n + 1, inf);
    deque<int> q;
    q.push_back(s);
    d[s] = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop_front();

        for (auto e: g[u]) {
            int v = e.second;
            int uv = e.first;
            if (d[v] > d[u] + uv) {
                d[v] = d[u] + uv;
                if (uv == 1) {
                    q.push_back(v);
                }
                else {
                    q.push_front(v);
                }
            }
        }
    }
}
```

3.6 Kruskal

```
/**
 * Description: Kruskal Algo
 * Time: if the graph is densed, use Prim for better performance
 */

struct DSU {
    vi lab;
    DSU(int n) { lab = vi(n, -1); }

    int get(int x) { return lab[x] < 0 ? x : lab[x] = get(lab[x]); }

    bool same_set(int a, int b) { return get(a) == get(b); }

    bool unite(int x, int y) {
                x = get(x), y = get(y);
                if (x == y) return false;
                if (lab[x] > lab[y]) swap(x, y);
                lab[x] += lab[y];
                lab[y] = x;
                return true;
        }
};

ll kruskal(int n, vector<pair<ll, pi>> Edges) {
    DSU d(n);
    ll res = 0; int taken = 0;

    sort(Edges.begin(), Edges.end());

    for (auto edge: Edges) {
        int u = edge.second.first;
        int v = edge.second.second;
        if (d.unite(u, v)) {
            res += edge.first;
```

```
            taken++;
        }
        if (taken == n - 1) break;
    }
    if (taken != n - 1) return -1;
    return res;
}
```

3.7 Prim

```
/**
 * Description: Prim Algo
 * Time: O(ElogV)
 * Status:
 */

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

vector<vii> AL;                                 // the graph stored in AL
vi taken;                                       // to avoid cycle
priority_queue<ii> pq;                          // to select shorter edges
// C++ STL priority_queue is a max heap, we use -ve sign to reverse order

void process(int u) { // set u as taken and enqueue neighbors of u
  taken[u] = 1;
  for (auto &[v, w] : AL[u])
    if (!taken[v])
      pq.push({-w, -v});                        // sort by non-dec weight
}                                               // then by inc id

int main() {
    int V, E; scanf("%d %d", &V, &E);
    AL.assign(V, vii());
    for (int i = 0; i < E; ++i) {
        int u, v, w; scanf("%d %d %d", &u, &v, &w);  // read as (u, v, w)
        AL[u].emplace_back(v, w);
        AL[v].emplace_back(u, w);
    }
    taken.assign(V, 0);                         // no vertex is taken
    process(0);                                 // take+process vertex 0
    int mst_cost = 0, num_taken = 0;            // no edge has been taken
    while (!pq.empty()) {                       // up to O(E)
        auto [w, u] = pq.top(); pq.pop();         // C++17 style
        w = -w; u = -u;                           // negate to reverse order
        if (taken[u]) continue;                   // already taken, skipped
        mst_cost += w;                            // add w of this edge
        process(u);                               // take+process vertex u
        ++num_taken;                              // 1 more edge is taken
        if (num_taken == V-1) break;              // optimization
    }
    printf("MST cost = %d (Prim's)\n", mst_cost);

}
```

3.8 MST variants

```
/**
 * Description: Some variants of MST problems
 */
```

Minimum Spanning Subgraph of MST problem. Some edges in the given graph have already been fixed and must be taken as part of the solution. For Kruskal's algorithm, we first take into account all the fixed edges and their costs. Then, we continue running Kruskal's algorithm on the remaining free edges until we have a spanning subgraph (or spanning tree). For Prim's algorithm, we give higher priorities to these fixed edges so that we will always take them and their costs.

Second-Best Spanning Tree of MST problem, We can see that the second best ST is actually the MST with just two edges difference. One edge is taken out from the MST and another chord edge is added into the MST. Next, for each edge in the MST (there are at most V-1 edges in the MST), temporarily flag it so that it cannot be chosen, then try to find the MST again in O(E) but now excluding that flagged edge. Note that we do not have to re-sort the edges at this point. The best spanning tree found after this process is the second best ST.