OI Templates
v1.0.0

Sing Yin Secondary School

2022-8-12

Contents

1 General

1.1 Debugging

In general, there are two common ways to debug.

1. You can debug with debugger.
   (a) Press F5.
   (b) Select GDB.
   (c) Select g++.exe compiler.
2. You can debug with command line.
   (a) Open terminal.
   (b) Confirm your file name.
   (c) Type the following command: `g++ --std=c++[version] -Wall -Wextra -Wshadow -O2 -o [filename].exe [filename].cpp`
   `g++`: g++ command
   `--std=c++[version]`: Select your c++ version.
   `-Wall`: Turn on most warnings.
   `-Wextra`: Turn on the warnings that -Wall doesn't turn on.
   `-Wshadow`: Check shadow declaration of variables. Very useful for debugging.
   `-o [filename].exe`: The output name of your executable.
   `[filename].cpp`: Your source code file.
   (d) Run your code with typing `[filename].exe`.
   (e) You can redirect input / output / errorlog by doing `[filename].exe < [inputFile] > [outputFile] >2 [errorLog]`. You don't need to redirect every stream each time.

   Note: if you are using linux, you need to first type `chmod +x [filename].out` to grant the executing permission for that file. After that, type `./[filename].out` to run.

1.2 Operator Overloading

Operator overloading is a powerful method that allows users to overload operators to perform different operatorations. We can use this to overload input and outupt of `std::vector` and `std::pair`.

```
// pair IO
template <class T, class V>
istream& operator>>(istream& is, pair<T, V>& obj){
    is >> obj.first >> obj.second;
    return is;
}

template <class T, class V>
ostream& operator<<(ostream& os, const pair<T, V>& obj){
    os << obj.first << ' ' << obj.second;
    return os;
}

// vector IO
template <class T>
istream& operator>>(istream& is, vector<T>& obj){
    for (int i = 0; i < int(obj.size()); i++) is >> obj[i];
    return is;
}

template <class T>
ostream& operator<<(ostream& os, const vector<T>& obj){
    for (int i = 0; i < int(obj.size()); i++){
        os << obj[i]; if (i != int(obj.size()) - 1) os << ' ';
    }
    return os;
}
```

2 Mathematics

2.1 Basic Modular Arithmetic

Modular arithmetic is very important especially when dealing with counting problems. Modulear arithmetic is essential to solve them.
Here are some common identities.

- If $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$
- If $a \equiv b \pmod{n}$, and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$
- $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$
- $(a - b) \bmod m = (a \bmod m - b \bmod m) \bmod m$
- $(a \times b) \bmod m = (a \bmod m \times b \bmod m) \bmod m$

2.2 Binary Exponentiation

Using this method, we can calculate $a^b \bmod m$ in $\mathcal{O}(\log n)$

Iterative method:

```
long long bigmd(long long x, long long y, long long z) {
    long long ans = 1;
    while (y){
        if (y & 1) ans = ans * x % z;
        y >>= 1;
        x = x * x % z;
    }
    return ans;
}
```

Recursive method:

```
long long bigmd(int n, int p, int mod){
    if (p == 0) return 1;
    long long tt = bigmd(n, p / 2, mod);
    return (p & 1 ? (n % mod) : 1) * (tt * tt % mod);
}
```

2.3 Modular Inverse

Fermat's little theorem states that

$$a^p \equiv a \pmod{p}$$

where a is an integer and p is a prime number.
The formula can be also represented as

$$a^{p-1} \equiv 1 \pmod{p}$$

Normally, we call a number b as an inverse of a when $a \times b = 1$.
e.g. 0.5 is the inverse of 2.

Similarly, we call a number b as the modular inverse of a modulo n when $a \times b \equiv 1 \pmod{n}$

e.g. 4 is the inverse of 2 modulo 7. ($4 \times 2 \bmod 7 = 1$)

If we recall the above formula, we can come into a conclusion that the modular inverse of $a \bmod p$ is $a^{p-2} \bmod p$ (for prime number p only).

Code example:

```
// use the above bigmd function
long long modinv(long long a, long long p){
    return bigmd(a, p - 2, p);
}
```

3 Data Structures

3.1 Disjoint-set Union

This data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The structure is very flexible as you can add different operations to it.

```
class disjoint_set_union{
    vector<int> p; // ancestor
    vector<int> minn, maxx;
    vector<int> count;

    vector<bool> config;
    public:
        disjoint_set_union(int length){
            p.resize(length + 1);
            iota(p.begin(), p.end(), 0);
        }

        disjoint_set_union(int length, bool haveCount, bool haveMin, bool haveMax){
            p.resize(length + 1);
            iota(p.begin(), p.end(), 0);
            if (haveCount) count.resize(length + 1, 1);
            if (haveMin){
                minn.resize(length + 1);
                iota(minn.begin(), minn.end(), 0);
            }
            if (haveMax){
                maxx.resize(length + 1);
                iota(maxx.begin(), maxx.end(), 0);
            }

            config = {haveCount, haveMin, haveMax};
        }

        int find(int node){
            return p[node] = p[node] == node ? node : find(p[node]);
        }

        void joint(int x, int y){
            int rootx = find(x), rooty = find(y);

            if (rootx != rooty){
                p[rooty] = rootx;
                if (config.size() > 0) operation(rootx, rooty);
            }
        }

        tuple<int, int, int> result(int x){
            int root = find(x);
            return make_tuple(minn[root], maxx[root], count[root]);
        }

    private:
        void operation(int x, int y){
            if (config[0]) count[x] += count[y];
            if (config[1]) minn[x] = min(minn[x], minn[y]);
            if (config[2]) maxx[x] = max(maxx[x], maxx[y]);
        }
};
```

3.2 Binary Indexed Tree / Fenwick Tree

```
class fenwick{
    // must be one-based
    vector<long long> bit;
    int size = 0;

    public:
        binary_index_tree(int length){
            bit.resize(length + 1);
            size = length + 1;
        }

        void update(int pos, int val){
            for (; pos < size; pos += pos & (-pos)) bit[pos] += val;
        }

        long long query(int pos){
            long long ans = 0;
            for (;pos > 0; pos -= pos & (-pos)) ans += bit[pos];
            return ans;
        }
};
```

3.3 Segment Tree

(Provided by iDoItSaNdWiTcH)

```
struct segtree {
    long long size;
    vector<long long> seg;

    segtree(long long n) {
        size = 1;

        while (size < n) size *= 2;
        seg.assign(2 * size, 0);
    }

    long long sum(long long l, long long r, long long x, long long lx, long long rx) {

        if (r <= lx || rx <= l) {
            return 0;
        }
        if (l <= lx && rx <= r) {
            return seg[x];
        }

        long long m = (lx + rx) / 2;

        //change here
        return sum(l, r, 2 * x + 1, lx, m) + sum(l, r, 2 * x + 2, m, rx);
    }

    long long sum(long long l, long long r) {
        return sum(l, r, 0, 0, size);
    }

    void set(long long i, long long v, long long x, long long lx, long long rx) {
        if (rx - lx == 1) {
            seg[x] = v;
            return;
        }

        long long m = (lx + rx) / 2;

        if (i < m) {
            set(i, v, 2 * x + 1, lx, m);
        } else {
            set(i, v, 2 * x + 2, m, rx);
        }

        seg[x] = seg[2 * x + 1] + seg[2 * x + 2];
    }

    void set(long long i, long long v) {
        set(i, v, 0, 0, size);
    }
};
```

4 String Algorithms

4.1 Hashing

(Provided by chemistrying) (bugged)

```
struct custom_hash{
    int mod = 31;
    int mod2 = 1000003957;
    // 1000003957, 1000001957, 1000003469, 1000003283, 1000002431
    // 1000010611, 1000009739, 1000009567, 1000012253, 1000011421
    void prefixHash(vector<long long>& dest, string s){
        dest.resize(s.size());
        dest[0] = s[0] - 'a';

        for (int i = 1; i < s.size(); i++){
            dest[i] = dest[i - 1] * mod % mod2 + (s[i] - 'a');
            dest[i] %= mod2;
        }
    }

    void prefixHashSize(vector<long long>& dest, string s, int size){
        dest.resize(s.size());
        dest[0] = s[0];

        long long power = bigmod(mod, size, mod2);

        for (int i = 1; i < s.size(); i++){
            dest[i] = dest[i - 1] * mod % mod2 + (s[i]);

            if (i >= size){
                long long minus = (s[i - size]) * power % mod2;
                dest[i] -= minus;
            }

            if (dest[i] < 0){
                dest[i] += mod2;
            }

            dest[i] %= mod2;
        }
    }

    void suffixHash(vector<long long>& dest, string s){
        dest.resize(s.size());
        dest[s.size() - 1] = s[s.size() - 1] - 'a';

        for (int i = s.size() - 2; i >= 0; i--){
            dest[i] = dest[i + 1] * mod + (s[i] - 'a');
            dest[i] %= mod2;
        }
    }

    long long singlePrefixHash(string s){
        long long result = s[0] - 'a';
        for (int i = 1; i < s.size(); i++){
            result = result * mod + (s[i] - 'a');
            result %= mod2;
        }
        return result;
    }

    long long singleSuffixHash(string s){
        long long result = s[s.size() - 1] - 'a';
        for (int i = s.size() - 2; i >= 0; i--){
            result = result * mod + (s[i] - 'a');
            result %= mod2;
        }
        return result;
    }
}hasher;
```

4.2 Trie

(Provided by iDoItSaNdWiTcH)

```
struct Trie {
    int totalsz = 0;
    bool isend = 0;
    struct map<int, Trie*> children;
};

void tinsert(struct Trie* root, string s) {
    //find the bug in tinsert
    int len = s.length();

    Trie* cnt = root;
    for (int i = 0; i < len; i++) {
        if (cnt->children[s[i] - 'a'] == nullptr) {
            //create new tree
            cnt->children[s[i] - 'a'] = new Trie();
        }
        cnt = cnt->children[s[i] - 'a'];
    }
    cnt->isend = 1;
}
```

(Provided by chemistrying)

```
struct trie_node{
    int children[26]; // index of the next node
    bool isWord = 0; // if it is a word
};

class trie{
    vector<trie_node> v;
    int size = 1;
    public:
        trie(){
            v.resize(1);
        }

        void insert(string s){
            int currpos = 0;
            for (int i = 0; i < s.size(); i++){
                // create a new node if it doesn't exist
                if (v[currpos].children[s[i] - 'a'] == 0){
                    v[currpos].children[s[i] - 'a'] = size++;
                    v.push_back(trie_node());
                }
                currpos = v[currpos].children[s[i] - 'a'];

                // mark it as a word if it is the end of the loop
                if (i == s.size() - 1) v[currpos].isWord = true;
            }
        }
};
```

5 Graphs

5.1 Depth-first Search

```
void dfs(vector<int>& visited, int node, int steppies){
    visited[node] = steppies;

    for (auto a : graph[node]){
        if (!visited[a]) dfs(visited, a, steppies + 1);
    }
}
```

5.2 Breadth-first Search

```
void bfs(vector<int>& visited, int node){
    queue<pair<int, int>> q;
    q.push({1, node});
    visited[node] = 1;

    while (!q.empty()){
        pair<int, int> curr = q.front();
        for (auto a : graph[curr.second]){
            if (!visited[a]){
                visited[a] = curr.first + 1;
                q.push({curr.first + 1, a});
            }
        }
        q.pop();
    }
}
```

5.3 Dijkstra's Algorithm

```
void dijkstra(vector<int>& visited, int node){
    priority_queue<pair<int, int>, vector<pair<int ,int>>, greater<pair<int ,int>>> pq;
    pq.push({0, node});
    visited[node] = 0;

    while (!pq.empty()){
        pair<int, int> curr = pq.top();
        pq.pop();

        if (curr.first > visited[curr.second]) continue;

        for (auto a : graph[curr.second]){
            if (curr.first + a.second < visited[a.first]){
                visited[a.first] = curr.first + a.second;
                pq.push({curr.first + a.second, a.first});
            }
        }
    }
}
```

5.4 Depth-first Search with Backtracking

```
void dfs(vector<int>& visited, int node, int steppies, vector<int>& trking){
    visited[node] = steppies;

    for (auto a : graph[node]){
        if (!visited[a]){
            trking[a] = node;
            dfs(visited, a, steppies + 1, trking);
        }
    }
}
```

5.5 Breadth-first Search with Backtracking

```
void bfs(vector<int>& visited, int node, vector<int>& trking){
    queue<pair<int, int>> q;
    q.push({1, node});
    visited[node] = 1;

    while (!q.empty()){
        pair<int, int> curr = q.front();
        for (auto a : graph[curr.second]){
            if (!visited[a]){
                visited[a] = curr.first + 1, trking[a] = curr.second;
                q.push({curr.first + 1, a});
            }
        }
        q.pop();
    }
}
```

5.6 Dijkstra's Algorithm with Backtracking

```
void dijkstra(vector<int>& visited, int node, vector<int>& trking){
    priority_queue<pair<int, int>, vector<pair<int ,int>>, greater<pair<int ,int>>> pq;
    pq.push({0, node});
    visited[node] = 0;

    while (!pq.empty()){
        pair<int, int> curr = pq.top();
        pq.pop();

        if (curr.first > visited[curr.second]) continue;

        for (auto a : graph[curr.second]){
            if (curr.first + a.second < visited[a.first]){
                visited[a.first] = curr.first + a.second, trking[a.first] = curr.second;
                pq.push({curr.first + a.second, a.first});
            }
        }
    }
}
```

5.7 Retrieval of Backtracking

```
void backtrack(vector<int>& backtracking, int start, int end, vector<int>& dest){
    int curr = end;
    while (curr != -1){
        dest.push_back(curr);
        curr = backtracking[curr];
    }
    reverse(dest.begin(), dest.end());
}
```

5.8 Bipartite Graph Checking

Code Example (Provided by longhuenchan):

```
bool isBipartite(vector<bool>& visited, int node, vector<int>& color){
    visited[node] = true;

    for (auto a : graph[node]){
        if (!visited[a]){
            color[a] = 1 - color[node];
            return isBipartite(visited, a, color);
        }else if (color[a] == color[node]){
            return false;
        }
    }

    return true;
}
```

5.9 Maximum Bipartite Matching

Idea: We keep finding valid augmentation to graph until we can find none.

1. We start at a vertex A, and it connects to vertices $B_1, B_2, \cdots, B_k$.
2. We try to include edge (A, B_1) in our maximum matching. If B_1 is not the endpoint of any edge in the current matching configuration, we are done.
3. Otherwise, B1 has been matched to some other vertex, lets say C. We have to find if C can match to some other vertices (not B). Only if it is possible, we match A with B_1.
4. If A cannot match with B_1, try next connected vertex (B_2) and repeat from step 2, until you find one augmentation, or you declare that A cannot be included.

(Provided by longhuenchan)

```
bool dfs(vector<bool>& visited, int node){
    if (visited[node]) return false;

    visited[node] = true;
    for (auto to : graph[node]){
        if (matching[to] == -1 || dfs(visited, matching[to])){
            matching[to] = cur;
            return true;
        }
    }

    return false;
}

int maxmatch(vector<vector<int>>& graph){
    int cnt = 0;
    vector<bool> visited(graph.size());
    for (int i = 1; i < graph.size(); i++){
        fill(visited.begin(), visited.end(), false);
        cnt += dfs(i);
    }
    return cnt;
}
```

6 Tree

6.1 Lowest Common Ancestor

(Provided by iDoItSaNdWiTcH)

```
//build lca
queue<long long> q;
q.push(1);
for (long long i = 0; i <= n; i++) visited[i] = 0;

while (!q.empty()) {
    long long cur = q.front(); q.pop();

    visited[cur] = 1;

    for (auto n : g[cur]) {
        if (!visited[n]) {
            depth[n] = depth[cur] + 1;
            lca[n][0] = cur;

            q.push(n);
        }
    }
}

for (long long j = 1; j < 30; j++) {
    for (long long i = 1; i <= n; i++) {
        lca[i][j] = lca[lca[i][j - 1]][j - 1];
    }
}

long long qq; cin >> qq;

while (qq--) {
    long long u, v; cin >> u >> v;
    long long uu = u, vv = v;

    long long lcaa;

    //get lca
    if (depth[u] != depth[v]) {
        if (depth[u] < depth[v]) swap(u, v);

        //u is the lower one
        for (long long i = 30; i > -1; i--) {
            if (depth[u] - (1LL << i) >= depth[v]) {
                u = lca[u][i];
            }
        }
    }

    if (u != v) {
        for (long long i = 30; i > -1; i--) {
            if (lca[u][i] != lca[v][i]) {
                u = lca[u][i];
                v = lca[v][i];
            }
        }

        lcaa = lca[u][0];
    } else {
        lcaa = u;
    }
    cout << lcaa << endl;
}
```

6.2 Minimum Spanning Tree

Properties of the minimum spanning tree

- A minimum spanning tree of a graph is unique, if the weight of all the edges are distinct. Otherwise, there may be multiple minimum spanning trees. (Specific algorithms typically output one of the possible minimum spanning trees).
- Minimum spanning tree is also the tree with minimum product of weights of edges. (It can be easily proved by replacing the weights of all edges with their logarithms)
- In a minimum spanning tree of a graph, the maximum weight of an edge is the minimum possible from all possible spanning trees of that graph. (This follows from the validity of Kruskal's algorithm).
- The maximum spanning tree (spanning tree with the sum of weights of edges being maximum) of a graph can be obtained similarly to that of the minimum spanning tree, by changing the signs of the weights of all the edges to their opposite and then applying any of the minimum spanning tree algorithm.

(Quoted from cp-algorithms)

6.2.1 Kruskal with Disjoint-set Union

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int mst(vector<Edge>& edges, int n, vector<Edge>& result){
    int cost = 0;

    DSU dsu(n);

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

    for (Edge e : edges) {
        if (dsu.get(e.u) != dsu.get(e.v)) {
            cost += e.weight;
            result.push_back(e);
            dsu.unite(e.u, e.v);
        }
    }

    return cost;
}
```