OI Templates
v1.1.0

Sing Yin Secondary School

2022-8-12

Contents

1 C++ Syntax

1.1 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;
}
```

1.2 String Functions

To construct a string with a specific character of length `n`:

```
string str(n, chr);
```

To obtain a substring `t` from index `i` to `j` inclusively:

```
string t = s.substr(i, j - i + 1);
```

To find if a character exists in a string `str`:

```
if (str.find(chr) != string::npos) {
    cout << "Yes\n";
} else {
    cout << "No\n";
}
```

To access front element / back element of a vector:

```
vec.front();
vec.back();
```

To insert an element `x` in position `idx` into a vector:

```
vec.insert(vec.begin() + idx, x);
```

1.3 Regular Expression

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 $O(\log n)$

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);
}
```

2.4 Matrix

Matrices can be useful in solving linear recurrences.

```cpp
struct matrix {
    int row_size, col_size;
    vector<vector<long long>> container;
    matrix(int n, int m) : row_size(n), col_size(m), container(n, vector<long long>(m))
        {};
    matrix(vector<vector<long long>> mtx) : row_size(mtx.size()), col_size(mtx.size() ==
         0 ? 0 : mtx[0].size()), container(mtx) {};

    matrix& operator *= (const matrix& rhs) {
        if (col_size != rhs.row_size) {
            cerr << "This two matrices cannot be multiplied together.\n";
            exit(EXIT_FAILURE);
        }

        vector<vector<long long>> new_container(row_size, vector<long long>(rhs.col_size
            ));
        int common_side = col_size;
        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < rhs.col_size; j++) {
                long long result = 0;
                for (int k = 0; k < common_side; k++) {
                    result += container[i][k] * rhs.container[k][j];
                }
                new_container[i][j] = result;
            }
        }

        container = new_container;
        return *this;
    }

    friend matrix operator * (matrix lhs, const matrix& rhs) {
        lhs *= rhs;
        return lhs;
    }

    matrix& operator += (const matrix& rhs) {
        if (col_size != rhs.col_size || row_size != rhs.row_size) {
            cerr << "This two matrices cannot be added together.\n";
            exit(EXIT_FAILURE);
        }

        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < col_size; i++) {
                container[i][j] += rhs.container[i][j];
            }
        }

        return *this;
    }

    matrix operator + (matrix lhs, const matrix& rhs) {
        return lhs + rhs;
    }

    matrix& operator %= (const int mod) {
```

```cpp
        for (int i = 0; i < row_size; i++) {
            for (int j = 0; j < col_size; j++) {
                container[i][j] %= mod;
            }
        }
        return *this;
    }

    friend matrix operator % (matrix lhs, const int mod) {
        lhs %= mod;
        return lhs;
    }
};

matrix power(matrix n, long long p, long long mod) {
    matrix curr = n;
    matrix result(vector<vector<long long>>({{1, 0}, {0, 1}}));
    while (p > 0) {
        if (p & 1) {
            result *= curr;
            result %= mod;
        }
        p >>= 1;
        curr *= curr;
        curr %= mod;
    }
    return result;
}
```

2.5 Extended Euclidean Algorithm

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> parent; // ancestor
    vector<int> minn, maxx;
    vector<int> count;

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

        disjoint_set_union(int length) {
            parent.resize(length + 1);
            iota(parent.begin(), parent.end(), 0);
            count.resize(length + 1, 1);
        }

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

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

            if (rootx != rooty) {
                parent[rooty] = rootx;
                count[rootx] += count[rooty];
            }
        }
};
```

3.2 Binary Indexed Tree / Fenwick Tree

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

public:
    fenwick(int length) {
        bit.resize(length + 1);
        this->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

3.3.1 Basic Segment Tree

```
template <class T>
class segment_tree {
private:
    int n;
    vector<T> tree;
    T operation(T left, T right) {
        return left + right;
    }

    void pushup(int node) {
        tree[node] = operation(tree[node + node], tree[node + node + 1]);
    }

    void build(const vector<T>& v, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = v[lo];
        } else {
            int mi = lo + (hi - lo) / 2;
            build(v, curr + curr, lo, mi);
            build(v, curr + curr + 1, mi + 1, hi);
            pushup(curr);
        }
    }

    void set(int idx, int val, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = val;
        } else {
            int mi = lo + (hi - lo) / 2;
            if (idx <= mi) {
                set(idx, val, curr + curr, lo, mi);
            } else {
                set(idx, val, curr + curr + 1, mi + 1, hi);
            }
            pushup(curr);
        }
    }

    T query(int l, int r, int curr, int lo, int hi) { // l, r is the target range!
        if (lo == l && hi == r) {
            return tree[curr];
        } else {
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                return query(l, r, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                return query(l, r, curr + curr + 1, mi + 1, hi);
            } else {
                return operation(query(l, mi, curr + curr, lo, mi), query(mi + 1, r,
                    curr + curr + 1, mi + 1, hi));
```

```
            }
        }
    }

public:
    segment_tree(const vector<T>& v) : n(v.size()), tree(4 * v.size()) {
        build(v, 1, 0, n - 1);
    }

    void set(int idx, T val) {
        set(idx, val, 1, 0, n - 1);
    }

    T query(int l, int r) {
        return query(l, r, 1, 0, n - 1);
    }
};
```

3.3.2 Online Segment with Maximum Sum

```
class segment_tree {
    int _size = 0;
    int power2size = 1;
    vector<long long> v;
    vector<long long> pref, suf, sum;

    public:
        segment_tree(int length) {
            _size = length;
            v.resize(4 * _size, 0);
            pref.resize(4 * _size);
            suf.resize(4 * _size);
            sum.resize(4 * _size);

            length--;
            while (length > 0) power2size <<= 1, length >>= 1;
        }

        void set(int l, int r, int pos, int idx, long long x, int p2s) {
            if (inRange(pos, pos + 1, l, r) == 2) {
                int mid = p2s >> 1;
                if (r - l != 1) {
                    set(l, l + mid, pos, 2 * idx + 1, x, mid);
                    set(l + mid, r, pos, 2 * idx + 2, x, mid);
                    v[idx] = max(max(v[2 * idx + 1], v[2 * idx + 2]), suf[2 * idx + 1] +
                        pref[2 * idx + 2]);
                    pref[idx] = max(pref[2 * idx + 1], sum[2 * idx + 1] + pref[2 * idx +
                        2]);
                    suf[idx] = max(suf[2 * idx + 2], sum[2 * idx + 2] + suf[2 * idx +
                       1]);
                    sum[idx] = sum[2 * idx + 1] + sum[2 * idx + 2];
                } else {
                    v[idx] = max(0LL, x);
                    pref[idx] = max(0LL, x);
                    suf[idx] = max(0LL, x);
                    sum[idx] = x;
                }
            }
```

```
        }

        void set(int pos, long long x) {
            set(0, _size, pos, 0, x, power2size);
        }

        // current left, current right, target left, target right
        long long ans(int l, int r, int lt, int rt, int idx, int p2s) {
            int checker = inRange(l, r, lt, rt);
            int mid = p2s >> 1;
            if (checker == 0) return 0;
            else if (checker == 1) return max(ans(l, l + mid, lt, rt, 2 * idx + 1, mid),
                ans(l + mid, r, lt, rt, 2 * idx + 2, mid));
            else return v[idx];
        }

        // [l, r)
        long long ans(int l, int r) {
            return ans(0, _size, l, r, 0, power2size);
        }

        int size() {
            return _size;
        }

        friend istream& operator >> (istream& is, segment_tree& obj) {
            for (int i = 0; i < obj.size(); i++) {
                int x;
                is >> x;
                obj.set(i, x);
            }
            return is;
        }

    private:
        // 0 = completely out of range, 1 = partially in range, 2 = completely in range
        // current segment, target segment
        int inRange(int l, int r, int lt, int rt) {
            if (r <= lt || l >= rt) return 0;
            else if (l >= lt && r <= rt) return 2;
            else return 1;
        }
};
```

3.3.3 Lazy Segment Tree

```
template <class T>
class segment_tree {
private:
    int n;
    vector<T> tree, diff; // tree = real value, diff = modification value
    vector<bool> lazy; // lazy = is the current node lazied

    T operation(T left, T right) {
        return min(left, right);
    }

    void pushup(int node) {
```

```
        if (lazy[node]) {
            // un-lazy itself
            lazy[node] = false;
            // lazy its children
            tree[node + node] = tree[node + node + 1] = diff[node];
            diff[node + node] = diff[node + node + 1] = diff[node];
            lazy[node + node] = lazy[node + node + 1] = true;
        }
        tree[node] = operation(tree[node + node], tree[node + node + 1]);
    }

    void build(const vector<T>& v, int curr, int lo, int hi) {
        if (lo == hi) {
            tree[curr] = v[lo];
        } else {
            int mi = lo + (hi - lo) / 2;
            build(v, curr + curr, lo, mi);
            build(v, curr + curr + 1, mi + 1, hi);
            pushup(curr);
        }
    }

    void set(int l, int r, T val, int curr, int lo, int hi) {
        if (lo == l && hi == r) {
            tree[curr] = val;
            diff[curr] = val;
            lazy[curr] = true;
        } else {
            pushup(curr);
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                set(l, r, val, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                set(l, r, val, curr + curr + 1, mi + 1, hi);
            } else {
                set(l, mi, val, curr + curr, lo, mi);
                set(mi + 1, r, val, curr + curr + 1, mi + 1, hi);
            }
            pushup(curr);
        }
    }

    T query(int l, int r, int curr, int lo, int hi) {
        if (lo == l && hi == r) {
            return tree[curr];
        } else {
            pushup(curr);
            int mi = lo + (hi - lo) / 2;
            if (r <= mi) {
                return query(l, r, curr + curr, lo, mi);
            } else if (l >= mi + 1) {
                return query(l, r, curr + curr + 1, mi + 1, hi);
            } else {
                return operation(query(l, mi, curr + curr, lo, mi), query(mi + 1, r,
                    curr + curr + 1, mi + 1, hi));
            }
        }
    }
public:
```

```
    segment_tree(const vector<T>& v) : n(v.size()), tree(4 * v.size()), diff(4 * v.size
        ()), lazy(4 * v.size()) {
        build(v, 1, 0, n - 1);
    }

    void set(int l, int r, T val) {
        set(l, r, val, 1, 0, n - 1);
        // cout << '\n';
    }

    T query(int l, int r) {
        return query(l, r, 1, 0, n - 1);
    }
};
```

3.4 Sparse Table

```
template <class T>
class sparse_table {
    int n, k;
    vector<vector<T>> table;

    T operation(T left, T right) {
        return left + right;
    }

    int log2_floor(unsigned long long i) {
        return i ? __builtin_clzll(1) - __builtin_clzll(i) : -1;
    }

    void build(const vector<T>& v) {
        // copy(v.begin(), v.end(), table[0]);
        for (int i = 0; i < n; i++) {
            table[0][i] = v[i];
        }
        for (int i = 1; i < k; i++) {
            for (int j = 0; j + (1 << i) <= n; j++) {
                table[i][j] = operation(table[i - 1][j], table[i - 1][j + (1 << (i - 1))
                    ]);
            }
        }
    }
public:
    sparse_table(const vector<T>& v) : n(v.size()), k(log2_floor(v.size()) + 1), table(k
        , vector<T>(v.size())) {
        build(v);
    }

    T query(int l, int r) {
        int size = r - l + 1;
        T ans = 0;
        for (int i = 0; i < k; i++) {
            if (size & 1) {
                ans = operation(ans, table[i][l]);
                l += 1 << i;
            }
            size >>= 1;
        }
```

```
        return ans;
    }
};
```

3.5 Min Heap / Max Heap

```
template <class T>
using max_heap = priority_queue<T, vector<T>>;

template <class T>
using min_heap = priority_queue<T, vector<T>, greater<T>>;
```

3.6 Linked List

3.6.1 Singly Linked List

To initiate a linked list:

```
list<int> lst(5, 0)
```

To insert element `x` to index `idx`:

```
auto it = lst.begin();
advance(it, idx);
lst.insert(it, x);
```

To erase element of index `idx`;

```
auto it = lst.begin();
advance(it, idx);
lst.erase(it);
```

3.6.2 Doubly Linked List

4 String Algorithms

4.1 Hashing

```
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 = bigmd(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

```
struct trie_node{
    int children[26]; // index of the next node
    int isWord = -1; // if it is a word
    int cnt = 0;
};

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

        void insert(string& s, int idx) {
            int currpos = 0;
            v[currpos].cnt++;
            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'];
                v[currpos].cnt++;

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

        void remove(string& s) {
            int currpos = 0;
            v[currpos].cnt--;
            for (int i = 0; i < s.size(); i++) {
                int next = v[currpos].children[s[i] - 'a'];
                currpos = next;
                v[currpos].cnt--;
            }
        }

        int traverse(string& p) {
            int currpos = 0;
            for (int i = 0; i < p.size(); i++) {
                int next = v[currpos].children[p[i] - 'a'];
```

```
                if (next == 0) return 0;
                currpos = next;
            }
            return v[currpos].cnt;
        }
};
```

4.3 KM(P)

5 Graph

5.1 Bipartite Graph

5.1.1 Bipartite Graph Checking

```
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;
}
```

5.1.2 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.

```
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;
}
```

5.2 Lowest Common Ancestor

```
class lowest_common_ancestor {
    vector<vector<int>> parent;
    vector<int> depth;
    void dfs(vector<vector<int>>& graph, int node, int steppies) {
        depth[node] = steppies;
        for (auto other : graph[node]) {
            parent[0][other] = node;
            dfs(graph, other, steppies + 1);
        }
    }

    public:
        lowest_common_ancestor(int n, int root, vector<vector<int>>& graph) {
            parent.resize(20, vector(n + 1, -1));
            depth.resize(n + 1);
            dfs(graph, root, 1);
            for (int i = 1; i < 20; i++) {
                for (int j = 0; j < n; j++) {
                    if (parent[i - 1][j] != -1) parent[i][j] = parent[i - 1][parent[i -
                        1][j]];
                }
            }
        }

        int lift(int node, int steppies) {
            for (int i = 19; i >= 0; i--) {
                if (steppies & (1 << i)) node = parent[i][node];
            }
            return node;
        }

        int query(int lhs, int rhs) {
            if (depth[lhs] > depth[rhs]) swap(lhs, rhs);
            int required_steppies = depth[rhs] - depth[lhs];
            rhs = lift(rhs, required_steppies);

            if (lhs == rhs) return lhs;

            for (int i = 19; i >= 0; i--) {
                if (parent[i][lhs] != parent[i][rhs]) {
                    lhs = parent[i][lhs], rhs = parent[i][rhs];
                }
            }

            return parent[0][lhs];
        }
};
```

5.3 Graph Traversal

5.3.1 Dijkstra's Algorithm

A single source weighted shortest path algorithm in $O(n \log n)$

```
void dijkstra(vector<int>& visited, int node){
    min_heap<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});
            }
        }
    }
}
```

5.3.2 Bellman-Ford Algorithm

A single source weighted shortest path algorithm which supports negative edges in $O(VE)$.

```
vector<long long> dist(n, inf * inf);
dist[root] = 0;

for (int t = 0; t < n - 1; t++) {
    for (int i = 0; i < n; i++) {
        for (pair<int, int> other : graph[i]) {
            int u = i, v = other.first;
            if (dist[u] != inf * inf && dist[v] > dist[u] + other.second) {
                dist[v] = dist[u] + other.second;
            }
        }
    }
}

// for checking negative cycles
for (int i = 0; i < n; i++) {
    for (pair<int, int> other : graph[i]) {
        int u = i, v = other.first;
        if (dist[u] != inf * inf && dist[v] > dist[u] + other.second) {
            cout << "NEGATIVE CYCLE\n";
            return 0;
        }
    }
}
```

5.3.3 Floyd-Warshall Algorithm

An all pairs weighted shortest path algorithm which supports negative edges in $O(V^3)$.

```
vector dist(n, vector(n, inf + inf));
for (int i = 0; i < n; i++) {
    dist[i][i] = 0;
}

for (int i = 0; i < m; i++) {
    long long u, v, w;
    cin >> u >> v >> w;
    dist[u][v] = w;
}
```

```cpp
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][k] != inf + inf && dist[k][j] != inf + inf) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}

// for checking negative cycles
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            if (dist[i][k] + dist[k][j] + dist[j][i] < 0) {
                cout << "NEGATIVE CYCLE\n";
                return 0;
            }
        }
    }
}
```

5.3.4 0-1 BFS

If the weights of the edges are either 0 or 1, we do not have to use a priority queue. Instead, we can use a deque. When we do insertion, if the newly added edge is 0, place it in front of the deque; otherwise, place it at the back of the deque.

```cpp
vector<int> dist(n, INF);
dist[s] = 0;
deque<int> dq;
dq.push_front(s);
while (!dq.empty()) {
    int node = dq.front();
    dq.pop_front();
    for (auto edge : graph[node]) {
        int other = edge.first;
        int weight = edge.second;
        if (dist[other] + w < dist[node]) {
            dust[other] = dist[node] + w;
            if (w == 1)
                q.push_back(other);
            else
                q.push_front(other);
        }
    }
}
```