Competitive Programming Library

Too bad to be Accepted

Contents

1 Dynamic Programming

1.1 Some dp patterns

Maximumu/Minimum path cost

```cpp
const int MAX = 21;
int grid[MAX][MAX];
int mem[MAX][MAX];
int n = 20;
bool valid(int r, int c){
  return r >= 0 && r < n && c >= 0 && c < n;
}

int maxPathSum(int r, int c){
    if(!valid(r,c)){
        return 0;
    }

    if(r == n-1 && c == n-1){
        return mem[r][c] = grid[r][c];
    }
    // available moves
    int path1 = maxPathSum(r+1,c);
    int path2 = maxPathSum(r,c+1);

    return grid[r][c] + max(path1,path2);
}
```

add operators between numbers to get max prod/sum

```cpp
// put +, -, between sequence of numbers such that the sum is divisible by
    k, and maximum as possible
const int MAX = 21;
long long mem[MAX][MAX];
const int n = 20;
int k = 4; // example
int v[20];
int fix(int a){
  return (a % k + k) % k;
}
long long tryAll(int pos, int mod){
```

```cpp
    long long &ret = mem[pos][mod];
    if(ret != -1){
        return ret;
    }
    if(pos == n){
        return ret = mod == 0;
    }
    if(tryAll(pos+1,fix(mod + v[pos])) || tryAll(pos+1,fix(mod-v[pos]))){
        return ret = 1;
    }
    return ret = 0;
}
```

pick choices with no two similar consecutive choices

```cpp
// pick minimum of choinces costs with no two similar consecutive choices
const int choices = 4;
const int n = 20;
int MAX = n;
int mem[MAX][choices];
const int OO = 1e6+1;
int minCost(int pos, int lastChoice){
    if(pos == n){
        return 0; // invalid move
    }
    int &ret = mem[pos][lastChoice];

    if(ret != -1){
        return ret;
    }

    ret = OO; // want to minimze
    // let choices are 0, 1, 2
    if(lastChoice != 0){
        ret = min(ret, minCost(pos+1,0));
    }
    if(lastChoice != 1){
        ret = min(ret, minCost(pos+1,1));
    }
    if(lastChoice != 2){
        ret = min(ret, minCost(pos+1,2));
    }
```

```
    return ret;
}
```

---

sum S and max/min Product

```cpp
int maxK;

ll mem[21][101]; // k, and s

// You are given an integer s and an integer k. Find k positive integers
    a1, a2, ..., ak
// such that their sum is equal to s and their product is the maximal
    possible. Return their product.

ll maxProd(int k, int rem)
{
 if(k == maxK){
  // base case
  if(rem == 0)
   return 1;
  return 0;
 }

 if(rem == 0) // invalid case
  return 0;

 ll &ret = mem[k][rem];

 if(ret != -1)
  return ret;

 ret = 0;

 for (int i = 1; i <= rem; ++i) {
  ll sol = maxProd(k+1, rem - i) * i;
  ret = max(ret, sol);
 }

 return ret;
}
```

2   Bit Manipulation

3 Algorithms

3.1 MO

MO Algorithm

```cpp
// MO        -> O(N+Q SQRT(N)) <= 10^5

const int N = 1e5+5, M = 1e5+5;
int n, m;
int nums[N], q_ans[M];

struct query {
    int idx, block_idx, l, r;

    query() = default;
    query(int _l, int _r, int _idx) {
        idx = _idx;
        r = _r - 1;
        l = _l - 1;
        block_idx = _l / sqrt(n);
    }

    bool operator <(const query & y) const {
        if(y.block_idx == block_idx) return r < y.r;
        return block_idx < y.block_idx;
    }
};

int freq[N], ans;

void add(int idx) {
    freq[nums[idx]]++;
    if (freq[nums[idx]] == 2) ans++;
}

void remove(int idx) {
    freq[nums[idx]]--;
    if (freq[nums[idx]] == 1) ans--;
}

cin >> n >> m;
for (int i = 0; i < n; ++i) cin >> nums[i];
```

```cpp
vector<query> Query(m);
for (int i = 0; i < m; ++i) {
    int l, r; cin >> l >> r;
    Query[i] = query(l, r, i);
}

sort(Query.begin(), Query.end());
int l0 = 1, r0 = 0;
for (int i = 0; i < m; ++i) {
    while (l0 < Query[i].l) remove(l0++);
    while (l0 > Query[i].l) add(--l0);
    while (r0 < Query[i].r) add(++r0);
    while (r0 > Query[i].r) remove(r0--);
    q_ans[Query[i].idx] = ans;
}
for (int i = 0; i < m; ++i) {
    cout << q_ans[i] << '\n';
}
```

4 Data Structures

4.1 Strings

4.1.1 Trie (Prefix Tree)

Basic Implementation

```cpp
#define MAX_CHAR 26

struct TrieNode {
    TrieNode *pTrieNode[MAX_CHAR]{};
    bool isWord;

    TrieNode() {
        isWord = false;
        fill(pTrieNode, pTrieNode + 26, (TrieNode *) NULL);
    }

    virtual ~TrieNode() = default;
};

class Trie {
private:
    TrieNode *root;
public:
    Trie() {
        root = new TrieNode();
    }

    virtual ~Trie() = default;

    TrieNode *getTrieNode() {
        return this->root;
    }

    void insert(const string &word) {
        TrieNode *current = root;
        for (char c: word) {
            int i = c - 'a';
            if (current->pTrieNode[i] == nullptr)
                current->pTrieNode[i] = new TrieNode();
            current = current->pTrieNode[i];
        }
        current->isWord = true;
    }

    bool search(const string &word) {
        TrieNode *current = root;
        int ch = 0;
        for (char c: word) {
            ch = c - 'a';
            if (current->pTrieNode[ch] == nullptr)
                return false;
            current = current->pTrieNode[ch];
        }
        return current->isWord;
    }

    bool startsWith(const string &prefix) {
        TrieNode *current = root;
        int ch = 0;
        for (char c: prefix) {
            ch = c - 'a';
            if (current->pTrieNode[ch] == nullptr)
                return false;
            current = current->pTrieNode[ch];
        }
        return true;
    }
};
```

4.2 Range Queries

4.2.1 Segment Tree

Basic Implementation

```cpp
struct Node {
    long long val;
};

struct SegTree {
private:
    const Node NEUTRAL = {INT_MIN};
```

```cpp
    static Node merge(const Node& x1, const Node& x2) {
        return {x1.val + x2.val};
    }


    void set(const int& idx, const int& val, int x, int lx, int rx) {
        if (rx - lx == 1) return void(values[x].val = val);

        int mid = (rx + lx) / 2;

        if (idx < mid)
            set(idx, val, 2 * x + 1, lx, mid);
        else
            set(idx, val, 2 * x + 2, mid, rx);

        values[x] = merge(values[2 * x + 1], values[2 * x + 2]);
    }

    Node query(const int& l, const int& r, int x, int lx, int rx) {
        if (lx >= r || l >= rx) return NEUTRAL;
        if (lx >= l && rx <= r) return values[x];

        int mid = (rx + lx) / 2;

        return merge(query(l, r, 2 * x + 1, lx, mid), query(l, r, 2 * x +
            2, mid, rx));
    }
    void build(vector<int> &a, int x, int lx, int rx) {
        if (rx - lx == 1) {
            if (lx < a.size()) {
                values[x].val = a[lx];
            }
            return;
        }
        int m = (lx + rx) / 2;
        build(a, 2 * x + 1, lx, m);
        build(a, 2 * x + 2, m, rx);
        values[x] = merge(values[2 * x + 1], values[2 * x + 2]);
    }

     void assign_range(int l, int r, int node, int lx, int rx, int time,
         int val) {
        if (lx > r || l > rx) return;
        if (lx >= l && rx <= r) {
            lazy[node] = {time, val};
            return;
        }
        int mid = (lx+rx) / 2;

        assign_range(l, r, 2*node+1, lx, mid, time, val);
        assign_range(l, r, 2*node+2, mid+1, rx, time, val);
    }

    pair<int, int> point_query(int lx, int rx, int node, int idx) {
        if(rx == lx) return lazy[node];
        int mid = (lx+rx) / 2;

        if(idx <= mid) {
            auto x = point_query(lx, mid, 2*node+1, idx);
            if(x.first > lazy[node].first) return x;
            return lazy[node];
        }
        auto x = point_query(mid+1, rx, 2*node+2, idx);
        if(x.first > lazy[node].first) return x;
        return lazy[node];
    }

public:
    int size{};
    vector<Node> values;

    void build(vector<int> &a) {
        build(a, 0, 0, size);
    }

    void init(int _size) {
        size = 1;
        while (size < _size) size *= 2;
        values.assign(2 * size, NEUTRAL);
    }

    void set(int idx, int val) {
        set(idx, val, 0, 0, size);
    }

    Node query(const int& l, const int& r) {
        return query(l, r, 0, 0, size);
    }
};
```

4.2.2   Fenwick Tree

Fenwick Tree

```cpp
struct Fenwick {
    // One Based
    vector<int> tree;

    explicit Fenwick(int n) {tree.assign(n + 5, {});}

    // Computes the prefix sum from [1, i], O(log(n))
    int query(int i) {
        int res = 0;
        while (i > 0) {
            res += tree[i];
            i &= ~(i & -i);
        }
        return res;
    }

    int query(int l, int r) {
        return query(r) - query(l-1);
    }

    // Get the value at index i
    int get(int i) {
        return query(i, i);
    }

    // Add 'v' to index 'i', O(log(n))
    void update(int i, int v) {
        while (i < tree.size()) {
            tree[i] += v;
            i += (i & -i);
        }
    }

    // Update range, Point query
    // To get(k) do prefix sum [1, k] and in insert update_range(i, i, a[i
        ])
    void update_range(int l, int r, int v) {
        update(l, v);
```

```cpp
        update(r+1, -v);
    }
};
```

4.2.3   Fenwick UpdateRange

BIT UpdateRange

```cpp
struct BITUpdateRange {
private:
    int n;
    vector<int> B1, B2;

    void add(vector<int> &b, int idx, int x) {
        while (idx <= n) {
            b[idx] += x;
            idx += idx & -idx;
        }
    }

    int sum(vector<int> &b, int idx) {
        int total = 0;
        while (idx > 0) {
            total += b[idx];
            idx &= ~(idx & -idx);
        }
        return total;
    }

    int prefix(int idx) {
        return sum(B1, idx) * idx - sum(B2, idx);
    }

public:
    explicit BITUpdateRange(int n) : n(n) {
        B1.assign(n + 1, {});
        B2.assign(n + 1, {});
    }

    void update(int l, int r, int x) {
        add(B1, l, x);
        add(B1, r + 1, -x);
        add(B2, l, x * (l - 1));
```

```
        add(B2, r + 1, -x * r);
    }

    int query(int i) {
        return prefix(i) - prefix(i - 1);
    }

    int query(int l, int r) {
        return prefix(r) - prefix(l - 1);
    }
};
```

4.2.4 2D BIT

2D BIT

```
struct BIT2D {
    int n, m;
    vector<vector<int>> bit;

    BIT2D(int n, int m) : n(n), m(m) {
        bit.assign(n + 2, vector<int>(m + 2));
    }

    void update(int x, int y, int val) {
        for (; x <= n; x += x & -x) {
            for (int i = y; i <= m; i += i & -i) {
                bit[x][i] += val;
            }
        }
    }

    int prefix(int x, int y) {
        int res = 0;
        for (; x > 0; x &= ~(x & -x)) {
            for (int i = y; i > 0; i &= ~(i & -i)) {
                res += bit[x][i];
            }
        }
        return res;
    }

    int query(int sx, int sy, int ex, int ey) {
```

```
        int ans = 0;
        ans += prefix(ex, ey);
        ans -= prefix(ex, sy - 1);
        ans -= prefix(sx - 1, ey);
        ans += prefix(sx - 1, sy - 1);
        return ans;
    }
};
```

4.2.5 Sparse Table

4.3 Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void erase_set(ordered_set &os, int v) {
    // Number of elements less than v
    int rank = os.order_of_key(v);

    auto it = os.find_by_order(rank);
    os.erase(it);
}
```

Ordered Set

5 Counting Principles

5.1 nCr

$$C(n, k) = \frac{n!}{(n-k)!k!} = \frac{n * (n-1) * (n-2) * \ldots * (n-k+1)}{k!}$$

5.1.1 Fast nCr

$$C(n, k) = \frac{n * (n-1) * (n-2) * \ldots * (n-k+1)}{1 * 2 * 3 * \ldots * k} = \prod_{i=0}^{k-1} \frac{n-i}{i+1} = \prod_{i=0}^{k-1} (n-i)(i+1)^{-1}$$

Fast nCr

```cpp
int nCr(const int& n, const int& r) {
    double res = 1;
    for (int i = 1; i <= r; ++i)
        res = res * (n - r + i) / i;
    return (int)(res + 0.01);
}
```

6 Graph Theory

6.1 Shortest Path algorithms

6.2 Dijkstra Algorithm

Dijkstra Implementation

```cpp
#define INF (1e18) // for int defined as ll

int n, m;
vector<vector<pair<int, int>>> adj;
vector<int> cost;
vector<int> parent;

void dijkstra(int startNode = 1) {
    priority_queue<pair<ll, int>, vector<pair<ll, int>>, greater<>> pq;

    cost[startNode] = 0;
    pq.emplace(0, startNode);

    while (!pq.empty()) {
        int u = pq.top().second;
        ll d = pq.top().first;
        pq.pop();

        if (d > cost[u]) continue;

        for (auto &p: adj[u]) {
            int v = p.first;
            int w = p.second;
            if (cost[v] > cost[u] + w) {
                cost[v] = cost[u] + w;
                parent[v] = u;
                pq.emplace(cost[v], v);
            }
        }
    }
}

void run_test_case(int testNum) {
    cin >> n >> m;

    adj.assign(n + 1, {});
```

```cpp
    cost.assign(n + 1, INF);
    parent.assign(n + 1, -1);

    while (m--) {
        // Read Edges
    }

    dijkstra();

    if (cost[n] == INF) {
        cout << -1 << el; // not connected {Depends on you use case}
        return;
    }

    stack<int> ans;
    for (int v = n; v != -1; v = parent[v]) ans.push(v);

    while (!ans.empty()) { // printing the path
        cout << ans.top() << ' ';
        ans.pop();
    }
    cout << el;
}
```

6.3 Floyd Warshal Algorithm

FloydWarshal Implementation

```cpp
int main() {
    int n, m; cin >> n >> m;
    vector <vector <int>> adj(n + 1, vector <int> (n + 1, 2e9));
    for (int i = 0; i < n; i++) adj[i][i] = 0;

    while(m--) {
        int u, v, w;
  cin >> u >> v >> w;
        adj[u][v] = min(adj[u][v], w);
        adj[v][u] = min(adj[v][u], w);
    }

    for (int mid = 1; mid <= n; mid++) {
        for (int start = 1; start <= n; start++) {
            for (int end = 1; end <= n; end++) {
                adj[start][end] = min(adj[start][end], adj[start][mid] +
                    adj[mid][end]);
            }
        }
    }

 return 0;
}
```

6.4 Bellman Ford Algorithm

BellmanFord Implementation

```cpp
 vector <vector <pair<int, int>>> &adj

vector <long long> BellmanFord(int src) {
    int n = (int)adj.size();
    vector <long long> dist(n, 2e18);

    dist[src] = 0;
    for (int it = 0; it < n-1; it++) {
        bool in = false;
        for (int i = 0; i < n; i++) { // iterate on the edges
            for (auto &[j, w] : adj[i]) {
                if (dist[j] > dist[i] + w) {
                    in = true;
                    dist[j] = dist[i] + w;
                }
            }
        }
        if (!in) return dist;
    }

    for (int i = 0; i < n; i++) {
        for (auto &[j, w] : adj[i]) {
            if (dist[j] > dist[i] + w) { //negative cycle
                return vector <long long> (n, -1); // or any flag
            }
        }
    }
}
```

```
        return dist;
}
```

6.5 Cycle Detection

6.5.1 DFS Implementation

DFS Implementation

```
// return true with number of nodes in the cycle, either odd cycle or even
bool cycle_detection(unordered_map<int, vector<int>> &graph, int source,
    int par, unordered_map<int,bool> vis, int c){
    if(vis[source]) return true;

    vis[source] = true;

    for(int v: graph[source]){
     if(v != par){
         c++;
         if(dfs(graph,v, source, vis, c)) return true;
     }
    }
    return false;
}
```

6.5.2 Another way for undirected graphs

Another way for undirected graphs

```
// this is true only for undirected graphs
bool dfs1(int cur, int par) {
    bool ret = false;
    vis[cur] = true;
    for (auto &i : adj[cur]) {
        if (!vis[i]) ret|=dfs1(i, cur);
        else if (par != i) ret = true;
    }
    return ret;
}
```

6.5.3 General Way

General Way

```
// general algorithm
vector <bool> cyc;
bool dfs(int cur, int par) {
    bool ret = false;
    vis[cur] = cyc[cur] = true;
    for (auto &i : adj[cur]) {
        if (par == i) continue;
        if (!vis[i]) ret|=dfs(i, cur);
        else if (cyc[i]) ret = true;
    }
    cyc[cur] = false;
    return ret;
}
```

6.5.4 DSU Implementation

DSU Implementation

```
#include <iostream>
#include <vector>

class UnionFind {
public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }

    int find(int u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    void unionSets(int u, int v) {
```

```cpp
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                ++rank[rootU];
            }
        }
    }

private:
    std::vector<int> parent;
    std::vector<int> rank;
};

bool detectCycle(const std::vector<std::pair<int, int>>& edges, int n) {
    UnionFind uf(n);

    for (const auto& edge : edges) {
        int u = edge.first;
        int v = edge.second;

        if (uf.find(u) == uf.find(v)) {
            return true;
        }
        uf.unionSets(u, v);
    }

    return false;
}

int main() {
    std::vector<std::pair<int, int>> edges = { {0, 1}, {1, 2}, {2, 3}, {3,
        0} };
    int n = 4; // Number of vertices

    if (detectCycle(edges, n)) {
        std::cout << "Cycle detected" << std::endl;
    } else {
        std::cout << "No cycle detected" << std::endl;
    }

    return 0;
}
```

7 Techniques

7.1 Coordinate Compression

```cpp
void coordinate_compress(vector<int> &x, int start=0, int
    step=1) {
 set unique(x.begin(), x.end());
 map<int, int> valPos;

 int idx=0;
 for (auto i: unique) {
  valPos[i] = start + idx * step;
  ++idx;
 }
 for(auto &i: x) i = valPos[i];

}
```

Coordinate Compression

8 Templates

8.1 MOD Template

```cpp
constexpr int MOD = 1e9+7; // must be a prime number

int add(int a, int b) {
    int res = a+b;
    if(res >= MOD) return res -= MOD;
}

int sub(int a, int b) {
    int res = a-b;
    if(res < 0) return res += MOD;
}

int power(int a, int e) {
    int res = 1;
    while(e) {if(e & 1) res = res * a % MOD; a = a * a % MOD;
    e >>= 1;}
    return res;
}

int inverse(int a) {
    return power(a, MOD-2);
}

int div(int a, int b) {
    return a * inverse(b) % MOD;
}
```

MOD Template

8.2 Macros

Macros

```cpp
#define getBit(n, k) (n >> k)
#define ON(n, idx) (n | (1ll << idx))
#define OFF(n, idx) (n & ~(1ll << idx))
#define toggle(n, idx) ((n) ^ (1ll<<(idx)))
```

```cpp
#define gray(n) (n ^ (n >> 1))
#define bitCount(x) (__builtin_popcountll(x))
#define uniq(x) x.resize(unique(x.begin(), x.end())-x.begin());

#define angle(a) (atan2((a).imag(), (a).real()))
//#define vec(a, b) ((b)-(a))
#define same(v1, v2) (dp(vec(v1,v2),vec(v1,v2)) < EPS)
#define dotProduct(a, b) ((conj(a)*(b)).real()) // a*b cos(T), if zero ->
    prep
#define crossProduct(a, b) ((conj(a)*(b)).imag()) // a*b sin(T), if zero
    -> parallel
//#define length(a) (hypot((a).imag(), (a).real()))
#define normalize(a) ((a)/length(a))
#define rotate0(v, ang) ((v)*exp(point(0,ang)))
#define rotateA(p, ang, about) (rotate0(vec(about,p),ang)+about)
#define reflect0(v, m) (conj((v)/(m))*(m))
#define ceil_i(a, b) (((ll)(a)+(ll)(b-1))/(ll)(b))
#define floor_i(a, b) (a/b)
#define round_i(a, b) ((a+(b/2))/b) // if a>0
#define round_m(a, b) ((a-(b/2))/b) // if a<0
#define round_multiple(n, m) round_i(n,m)*m // round to multiple if
    specified element

const double PI = acos(-1.0);

int dx[4] = {1, -1, 0, 0};
int dy[4] = {0, 0, 1, -1};

int dx[8] = {1, 1, -1, -1, 2, 2, -2, -2};
int dy[8] = {2, -2, 2, -2, 1, -1, 1, -1};
```