Team Notebook

lyreloi

July 6, 2024

Contents

1 Data Structures

1.1 Doubly Linked List

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
    def get_data(self):
        return self.data
class Sentinel_DLL:
    def __init__(self):
        self.sentinel = Node(None)
        self.sentinel.next = self.sentinel
        self.sentinel.prev = self.sentinel
    def first_node(self):
        if self.sentinel.next == self.sentinel:
            return None
        else:
            return self.sentinel.next
    def insert_after(self, x, data):
        y = Node(data)
        y.prev = x
        y.next = x.next
        x.next = y
        y.next.prev = y
    def append(self, data):
        last_node = self.sentinel.prev
        self.insert_after(last_node, data)
    def prepend(self, data):
        self.insert_after(self.sentinel, data)
    def delete(self, x):
        x.prev.next = x.next
        x.next.prev = x.prev
    def find(self, data):
        self.sentinel.data = data
        x = self.first_node()
        while x.data != data:
            x = x.next
        self.sentinel.data = None
        if x == self.sentinel:
            return None
        else:
            return x
    def __str__(self):
        s = "["
        x = self.sentinel.next
        while x != self.sentinel:
            if type(x.data) == str:
                s += "'"
            s += str(x.data)
            if type(x.data) == str:
                s += "'"
            if x.next != self.sentinel:
                s += ", "
            x = x.next

        s += "]"
        return s
#test
llist = Sentinel_DLL()
llist.append(5)
llist.append(6)
llist.append(2)
llist.prepend(19)
print(llist)
#insert_after = insert a new node with data after node x
#append = insert new node at end of list
#prepend = insert a new node at the start of the list
#delete = delete node x
#find = finds x (note: O(n) )
```

1.2 Order Statistics Tree

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<pair<ll, int>, null_type, less<pair<ll, int>>,
    rb_tree_tag,
            tree_order_statistics_node_update>
    ordered_set;

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

    ordered_set s;
    // s.insert(2);
    // s.insert(3);
    // s.insert(5);
    // s.order_of_key(3); // index when 3 is inserted OR how
         many values are to the left of 3
    // s.find_by_order(0); // what is in index i
    // cout << s.order_of_key(3) << endl;
    // cout << s.order_of_key(4) << endl;
    s.insert({-2,2});
    s.insert({-1,1});
    s.insert({-1,3});
    cout << s.order_of_key({-1, 1}) << endl;

    return 0;
}
```

1.3 Segment Tree - Range Compression

```
struct CompressedST {
  int n;
  vector<ll> st, lazy;

  // compressed information
  vector<pair<ll,ll>> lr;
  map<ll, int> compress;

  CompressedST(vector<ll> &c) {
    int sz = c.size();
    for (int i = 0; i < sz-1; i++) {
      compress[c[i]] = lr.size();
      lr.push_back({c[i], c[i]});
      if (c[i]+1 <= c[i+1]-1)
        lr.push_back({c[i]+1, c[i+1]-1});
    }
    compress[c[sz-1]] = lr.size();
    lr.push_back({c[sz-1], c[sz-1]});
    n = lr.size();

    st.assign(4*n, 0);
    lazy.assign(4*n, 0);
  }

  void pull(int p) {
    st[p] = st[p<<1] + st[p<<1|1];
  }

  void push(int p, int i, int j) {
    if (lazy[p]) {
      st[p] += (lr[j].second-lr[i].first+1)*lazy[p];
      if (i != j) {
        lazy[p<<1] += lazy[p];
        lazy[p<<1|1] += lazy[p];
      }
      lazy[p] = 0;
    }
  }
```

```
  void update(int l, int r, ll v, int p, int i, int j) {
    push(p, i, j);
    if (l <= i && j <= r) {
      lazy[p] += v;
      push(p, i, j);
    }
    else if (j < l || r < i);
    else {
      int k = (i+j)/2;
      update(l, r, v, p<<1, i, k);
      update(l, r, v, p<<1|1, k+1, j);
      pull(p);
    }
  }

  ll query(int l, int r, int p, int i, int j) {
    push(p, i, j);
    if (l <= i && j <= r) return st[p];
    else if (j < l || r < i) return 0;
    else {
      int k = (i+j)/2;
      return query(l, r, p<<1, i, k)
        + query(l, r, p<<1|1, k+1, j);
    }
  }

  ll query(ll l, ll r) {
    return query(compress[l], compress[r], 1, 0, n-1);
  }
  void update(ll l, ll r, ll v) {
    update(compress[l], compress[r], v, 1, 0, n-1);
  }
};
```

1.4 Segment Tree - Range Update

```
struct segtree {
    int n, *vals, *deltas;
    segtree(vector<int> &ar) {
        n = ar.size();
        vals = new int[4*n];
        deltas = new int[4*n];
        build(ar, 1, 0, n-1);
    }

    void build(vector<int> &ar, int p, int i, int j) {
        deltas[p] = 0;
        if (i == j) {
            vals[p] = ar[i];
        }
        else {
            int k = (i + j) / 2;
            build(ar, p<<1, i, k);
            build(ar, p<<1|1, k+1, j);
            pull(p);
        }
    }

    void pull(int p) {
        vals[p] = vals[p<<1] + vals[p<<1|1];
    }

    void push(int p, int i, int j) {
        if (deltas[p]) {
            vals[p] += (j - i + 1) * deltas[p];
            if (i != j) {
                deltas[p<<1] += deltas[p];
                deltas[p<<1|1] += deltas[p];
            }
            deltas[p] = 0;
        }
    }

    // i, j starts at 0, n-1
    void update(int _i, int _j, int v, int p, int i, int j) {
        push(p, i, j);
        // query overlaps or equates i, j
        if (_i <= i && j <= _j) {
            deltas[p] += v;
            push(p, i, j);
        }
        // no overlap
        else if (_j < i || j < _i) {}
        else {
            int k = (i + j) / 2;
            update(_i, _j, v, p<<1, i, k);
            update(_i, _j, v, p<<1|1, k+1, j);
            pull(p);
        }
    }

    int query(int _i, int _j, int p, int i, int j) {
        push(p, i, j);
        if (_i <= i && j <= _j)
            return vals[p];
        else if (_j < i || j < _i)
            return 0;
        else {
            int k = (i + j) / 2;
            return query(_i, _j, p<<1, i, k) +
                   query(_i, _j, p<<1|1, k+1, j);
        }
    }

    void update(int _i, int _j, int v) {
        update(_i, _j, v, 1, 0, n-1);
    }

    int query(int _i, int _j) {
        return query(_i, _j, 1, 0, n-1);
    }
};
```

1.5 Union Find

```
class DisjointSet
{
    // put this in main()
    //vector<int> univ;
    //for (int i = 1; i <= n; i++) univ.push_back(i);
    //DisjointSet ds;
    //ds.makeSet(univ);

    unordered_map<int, int> parent;
    unordered_map<int, int> rank;
    unordered_map<int, int> members;

public:
    void makeSet(vector<int> const &universe)
    {
        for (int i: universe)
        {
            parent[i] = i;
            rank[i] = 0;
            members[i] = 1;
        }
    }

    int Find(int k)
    {
        if (parent[k] != k)
        {
            parent[k] = Find(parent[k]);
        }

        return parent[k];
    }
```

```
    void Union(int a, int b)
    {
        int x = Find(a);
        int y = Find(b);

        if (x == y) {
            return;
        }

        if (rank[x] > rank[y]) {
            parent[y] = x;
            members[x] += members[y];
        }
        else if (rank[x] < rank[y]) {
            parent[x] = y;
            members[y] += members[x];
        }
        else {
            parent[x] = y;
            rank[y]++;
            members[y] += members[x];
        }
    }

    int GetMembers(int a)
    {
        // get the number of members of the disjoint set
             where a is included
        int x = Find(a);
        return members[x];
    }
};
```

2 Graph Algorithms

2.1 Bellman-Ford

```
bool bellman(int s){
    dist[s] = 0;
    for (int i = 0; i < n-1; i++){
        for (int u = 1; u <= n; u++){
            for (auto& [v, w] : adj[u]){
                dist[v] = max(dist[v], dist[u] + w);
            }
        }
    }
    ll ans = dist[n];
    for (int u = 1; u <= n; u++){
        for (auto& [v, w] : adj[u]){
            dist[v] = max(dist[v], dist[u] + w);
            // if dist[v] changes, there's a cycle
        }
    }
    return ans == dist[n];
}
```

2.2 Binary Lifting

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;

void binary_lift(vector<vector<int>>& lift, int n, int l){
    // generates binary lift DS.
    // lift[u][steps] - resulting node after jumping 2^steps
         from node u
    // lift[u][0] for any u should be pre-computed already
    // n - no. of nodes. 1-indexed
    // l - 2^(l-1) is the max no. of steps

    for (int j = 1; j < l; j++){
        for (int u = 1; u <= n; u++){
            lift[u][j] = lift[lift[u][j-1]][j-1];
        }
    }
}

int jump(int u, int steps, vector<vector<int>>& lift){
    // jumps 'steps' steps from u. returns resulting node.
    int bit = 0;
    while (steps){
        if (steps & 1){
            u = lift[u][bit];
        }
        bit++;
        steps >>= 1;
    }
    return u;
}

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

    int n, l;
    int q;
    cin >> n >> q;
    l = 30; // log2(1e9) exclusive
    vector<vector<int>> lift(n+1, vector<int>(l));

    for (int u = 1; u <= n; u++){
        cin >> lift[u][0];
    }

    binary_lift(lift, n, l);

    while (q--){
        int u, steps;
        cin >> u >> steps;
        cout << jump(u, steps, lift) << "\n";
    }
    return 0;
}
```

2.3 Edmonds-Karp

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long int;

struct edge {
    size_t i; // index at edges
    int v;
    ll c, f; // directed to v, capacity, flow
    ll residue() { return c - f; }
};

struct flow_network {
    int n, s, t;
    vector<edge> edges; // even indeces are forward flows,
         e_i+1 are reverse flows.
    vector<vector<int>> adj; // stores index pointing in
         edges
    vector<int> parent;
    set<pair<int, int>> edge_cuts;
    set<int> A; // set of nodes that belongs to one side of
         the cut

    flow_network(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        parent.resize(n);
    }

    void add_edge(int u, int v, ll cap) {
        edges.push_back({edges.size(), v, cap, 0});
```

```
    adj[u].push_back((int)edges.size()-1);
    edges.push_back({edges.size(), u, 0, 0}); // reverse
    adj[v].push_back((int)edges.size()-1);
  }

  bool aug_path() {
    for (int i=0; i<n; i++) parent[i] = -1;
    parent[s] = s;
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (auto ind : adj[u]){
        edge& e = edges[ind];
        if (e.residue() > 0 && parent[e.v] == -1) {
          parent[e.v] = e.i;
          q.push(e.v);
        }
      }
    }
    return parent[t] != -1;
  };

  ll augment() {
    ll bottleneck = numeric_limits<ll>::max();
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      bottleneck = min(bottleneck, edges[parent[v]].
          residue());
    }
    for (int v = t; v != s; v = edges[parent[v] ^ 1].v) {
      edges[parent[v]].f += bottleneck;
      edges[parent[v] ^ 1].f -= bottleneck;
    }
    return bottleneck;
  }

  ll calc_max_flow() {
    ll flow = 0;
    while (aug_path()){
      flow += augment();
    }
    return flow;
  }

  void calc_edge_cuts() {
    queue<int> q;
    q.push(s);
    vector<int> vis(n, 0);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      A.insert(u);
      for (auto ind : adj[u]) {
        edge& e = edges[ind];
        if (ind % 2 == 0 && !vis[e.v] && e.residue() >
            0) {
          vis[e.v] = 1;
          q.push(e.v);
        }
      }
    }
    for (int u = 0; u < n; u++) {
      for (auto ind : adj[u]) {
        edge& e = edges[ind];
        int a = u, b = e.v;
        if (a > b) swap(a, b);

        if ((A.find(a) != A.end() && A.find(b) == A.
            end()) ||
          (A.find(a) == A.end() && A.find(b) != A.
              end())){
          edge_cuts.insert({a, b});
        }
      }
    }

  }
};

int main(){
  int n, m;
  cin >> n >> m;
  int s = 0, t = n-1;
  flow_network fn(n, s, t);
  for (int i = 0; i < m; i++) {
    int u, v;
    ll cap;
    cin >> u >> v >> cap;
    u--; v--;
    fn.add_edge(u, v, cap);
  }

  cout << fn.calc_max_flow() << endl;
}
```

2.4 Kruskal

```
void kruskal(vector<pair<ll, pair<ll, ll>>> &res){
  // res == minimum spanning tree vector
  // needs DisjointSet class
  DisJointSet ds;
  vector<int> univ;
  for (int i = 1; i <= n; i++)
    univ.push_back(i);
  ds.makeSet(univ);
  // edges == vector of edges, vector< weight , uv >
  // edges should be sorted.
  for (auto edge : edges){
    int u = edge.second.first;
    int v = edge.second.second;
    if (ds.hasCycle(u, v))
      continue;
    ds.Union(u, v);
    res.push_back(edge);
  }
}
```

2.5 Prim

```
void prim(int start, vector<pair<ll, pair<ll, ll>>> &res){
  // res == minimum spanning tree vector
  priority_queue<pair<ll, pair<ll, ll>>> pq;
  vector<bool> vis(n+1, false);
  vis[start] = true;

  for (auto &[v, w] : graph[start]){
    pq.push({w, {start, v}});
  }

  while (!pq.empty()){
    auto edge = pq.top();
    pq.pop();
    ll u = edge.second.second;
    if (vis[u]) continue;
    vis[u] = true;
    res.push_back(edge);
    for (auto &[v, w] : graph[u])
     if (!vis[v]) pq.push({w, {u, v}});
  }
}
```

2.6 Shortest Path Faster Algo

```
void spfa(int s){
  for (int u = 0; u <= n; u++){
```

```
        dist[u] = 1e18;
    }
    dist[s] = 0;
    queue<int> q;
    q.push(s);
    vis[s] = 1;
    while (!q.empty()){
        int u = q.front(); q.pop();
        vis[u] = 0;
        for (int i = 0; i < adj[u].size(); i++){
            int v = adj[u][i].first;
            int w = adj[u][i].second;
            if (dist[v] > dist[u] + w){
                dist[v] = dist[u] + w;
                if (!vis[v]){
                    q.push(v);
                    vis[v] = 1;
                }
            }
        }
    }
}
```

2.7 Tarjan

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;

const int MAXN = 1e5+10;
int n;
vector<vector<int>> adj;

int id = 0, sccCount = 0;
int ids[MAXN], low[MAXN], onStack[MAXN];
stack<int> st;

void dfs(int at){
    st.push(at);
    onStack[at] = 1;
    ids[at] = low[at] = id++;

    for (auto to : adj[at]){
        if (ids[to] == -1)
            dfs(to);
        if (onStack[to])
            low[at] = min(low[at], low[to]);
    }
    if (ids[at] == low[at]){
        while (!st.empty()){
            int node = st.top();
            st.pop();
            onStack[node] = 0;
            low[node] = ids[at];
            if (node == at)
                break;
        }
        sccCount++;
    }
}

void fixIndex(){
    map<int, int> old_new;
    int newi = 0;
    for (int i = 0; i < n; i++){
        if (old_new.find(low[i]) == old_new.end()){
            old_new[low[i]] = newi++;
        }
    }
    for (int i = 0; i < n; i++){
        low[i] = old_new[low[i]];
    }
}

void tarjan(){
    memset(ids, -1, sizeof(ids));
    for (int i = 0; i < n; i++){
        if (ids[i] == -1)
            dfs(i);
    }
    fixIndex();
}

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



    return 0;
}
```

3 Math

3.1 Sieve

```
MXN = 100000;
bool prime[MXN + 1];

void sieve()
{
    memset(prime, true, sizeof(prime));

    for (int p = 2; p * p <= MXN; p++) {
        if (prime[p] == true) {
            for (int i = p * p; i <= MXN; i += p)
                prime[i] = false;
        }
    }
}
```

4 z Miscellaneous

4.1 CPP Fast IO

```
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;

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

    return 0;
}
```

4.2 Stress Test

```
import random, subprocess

def generate():
    '''Insert generator here'''

solution = input("Solution file: ")
brutef = input("Bruteforce file: ")

passed = 0
while passed <= 1000:
    test_case = generate()
    with open('input.txt', mode='w') as f:
        print(test_case, file=f)
```

```
p1 = subprocess.run(
    f'python3 {brutef} < input.txt',
    check=True, shell=True, capture_output=True, text=
        True
)
p2 = subprocess.run(
    f'./{solution} < input.txt',
    check=True, shell=True, capture_output=True, text=
        True
)
if p1.stdout != p2.stdout:
    print('Failed!')
    print('Expected:', p1.stdout)
    print('Output:', p2.stdout)
    print("Test Case:\n" + test_case)
    break

passed += 1
print(f'{passed} cases passed')
```