Competitive Programming Contest Notes

Vid Furlan

Revision: 0.0

Index

1.Time Complexity

Input	**Required Time Complexity**
$n < 10$	$O(n!)$
$n < 20$	$O(2^n)$
$n < 500$	$O(n^3)$
$n < 5000$	$O(n^2)$
$n < 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(\log n)$ or $O(1)$

2.Graphs

2.1.Dijkstra's Algorithm O($n + m \log n$)

Dijkstra's Algorithm is used to find the shortest path from a single node to all the other nodes in a weighted graph.

```
vector<long long> dist(n, LLONG_MAX);
using T = pair<long long, int>; // {distance, node}
priority_queue<T, vector<T>, greater<T>> pq;

int start = 0;
dist[start] = 0;
pq.push({0, start});

while (!pq.empty()) {
    const auto [cdist, node] = pq.top();
    pq.pop();
    if (cdist != dist[node]) { continue; }
    for (const pair<int, int> &i : neighbors[node]) {
        // If we can reach a neighbouring node faster,
        // we update its minimum distance
        if (cdist + i.second < dist[i.first]) {
            dist[i.first] = cdist + i.second;
            pq.push({dist[i.first], i.first});
        }
    }
}
```

2.2.Floyd-Warshall Algorithm O(n^3)

Floyd-Warshall Algorithm is used to find the shortest path between all pairs of nodes in a weighted graph.

```
// Min distance is initialized to INF or edge weights
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            long long new_dist = min_dist[i][k] + min_dist[k][j];
            if (new_dist < min_dist[i][j]) {
                min_dist[i][j] = min_dist[j][i] = new_dist;
            }
        }
    }
}
```

2.3.Cycle Detection O(m)

Cycle Detection is used to find cycles in a graph using DFS.

```
int n;
vector<vector<int>> adj;
vector<bool> visited;
vector<int> parent;
int cycle_start, cycle_end;
bool dfs(int v, int par) { // passing vertex and its parent vertex
    visited[v] = true;
    for (int u : adj[v]) {
        if (u == par) continue; // skipping edge to parent vertex
        if (visited[u]) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
        parent[u] = v;
        if (dfs(u, parent[u]))
            return true;
    }
    return false;
}
void find_cycle() {
    visited.assign(n, false);
    parent.assign(n, -1);
    cycle_start = -1;
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dfs(v, parent[v]))
            break;
    }
    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        cout << "Cycle found: ";
        for (int v : cycle) cout << v << " ";
        cout << endl;
    }
}
```

2.4.Finding Bridges $\mathbf{O}(n + m)$

Finding Bridges: An edge in an undirected graph is a bridge if removing it disconnects the graph.

```
void IS_BRIDGE(int v,int to); // some function to process the found bridge
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    bool parent_skipped = false;
    for (int to : adj[v]) {
        if (to == p && !parent_skipped) {
            parent_skipped = true;
            continue;
        }
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs(i);
    }
}
```

3.Math

3.1.Sieve of Eratosthenes O($n \log \log n$)

Sieve of Eratosthenes is used to find all prime numbers up to a given limit.

```
int n;
vector<bool> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
}
```

4.Modular Arithmetic

4.1.Modular Exponentiation O($\log m$)

Modular Exponentiation is used to calculate $a^b \% mod$.

```
int modPow(int a, int b, int m){
    if (b == 0) {
        return 1;
    }
    if ((b & 1) == 0) {
        int x = fastPower(a, b>>1, m);
        x = x % m;
        return (x*x) % m;
    }
    return (a * modPow(a, b-1, m)) % m;
}
```

4.2.Modular Inverse O($\log m$)

Modular Inverse calculates $a^{-1} \% mod$.

```
int inv(int a) {
    return a <= 1 ? a : m - (long long)(m/a) * inv(m % a) % m;
}
```

5.Data Structures

5.1.Union Find O($\alpha(n)$)

Union Find is used to find connected components in a graph.

```
class DisjointSets {
    private:
    vector<int> parents;
    vector<int> sizes;

    public:
    DisjointSets(int size) : parents(size), sizes(size, 1) {
        for (int i = 0; i < size; i++) { parents[i] = i; }
    }

    // @return the "representative" node in x's component
    int find(int x) {
        return parents[x] == x ?
        x : (parents[x] = find(parents[x]));
    }

    // @return whether the merge changed connectivity
    bool unite(int x, int y) {
        int x_root = find(x);
        int y_root = find(y);
        if (x_root == y_root) { return false; }
        if (sizes[x_root] < sizes[y_root]) { swap(x_root, y_root); }
        sizes[x_root] += sizes[y_root];
        parents[y_root] = x_root;
        return true;
    }
    // @return whether x and y are in the same connected component
    bool connected(int x, int y) { return find(x) == find(y); }
};
```