ICPC Library | Ufes

Contents

1 Divide and Conquer

1.1 Bisection Method

```
// Bisection Method

// Very useful for finding roots of a function
// With 100 repetitions the value already converges.
// This implementation only works if the function in the range [lo, hi] has
    some
// zero.

// F(x)
// ^     F(lo)
// |   *
// |   | *
// |   |  *
// |   |   *  Goal
// |--------O------------------------> x
// |         *             |
// |           *           *
// |             *       *   F(hi)
// |               * * *

double bisection(double lo, double hi) {
    for (int i = 0; i < 100; i++) {
        double mid = (lo + hi) / 2;
        double F   = f(mid);  // Declare a function
        if (F > 0)
            lo = mid;
        else
            hi = mid;
    }
    return lo;
}
```

1.2 Ternary Search

```
// Ternary search
// Very useful for finding max/min values between interval
// The function on the interval must be unimodal (only 1 maximum)

// F(x)
// ^                Goal
// |               o
// |            *     *
// |          *         *
// |        *             *
// |--------*----------------*-----------> x
// |   |   *                   *  |
// |   | *                        * F(r)
// |   * F(l)

double ternary_search(double l, double r) {
    double eps = 1e-9;
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);
        double f2 = f(m2);
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l);  // Return the maximum of f(x) in [l, r]
}
```

1.3 Binary Search

```
// Binary Search

int binary_search(vector<int> arr, int target) {
    int left  = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

2 Sorting

2.1 Merge Sort

```
// Merge sort with inversion counter

int merge(int *arr, int *aux, int lo, int hi, int mid) {
    int inv = 0;
    for (int k = lo; k <= hi; k++) aux[k] = arr[k];
```

```
    int i = lo;
    int j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid)
            arr[k] = aux[j++];
        else if (j > hi)
            arr[k] = aux[i++];
        else if (aux[j] < aux[i]) {
            arr[k] = aux[j++];
            inv += mid + 1 - i;
        } else
            arr[k] = aux[i++];
    }
    return inv;
}

int mergesort(int *arr, int *aux, int lo, int hi) {
    int inv = 0;
    if (lo >= hi) return inv;
    int mid = lo + (hi - lo) / 2;
    inv += mergesort(arr, aux, lo, mid);
    inv += mergesort(arr, aux, mid + 1, hi);
    inv += merge(arr, aux, lo, hi, mid);
    return inv;
}
```

3 Graph Algorithms

3.1 DFS

```
// Depth first search

int         V;
vector<vi> adj;
bool        vis[VMAX];
vi          topsort;  // Topological Sort.
                      // Only works in directed acyclic graph.

void dfs(int s) {
    vis[s] = true;
    for (auto a : adj[s]) {
        if (!vis[a]) {
            dfs(a);
        }
    }
    topsort.push_back(s);  // Only works in DAG.
}
```

3.2 Strongly Connected Components

```
// Tarjan's Algorithm
// Finding strongly connected components (Directed Graph)

int         V;
vector<vi> adj;
vi          dfslow, dfsnum;
bool        vis[VMAX];
int         SCC, TIME;
stack<int> aux;

void tarjan_dfs(int s) {
    dfslow[s] = dfsnum[s] = ++TIME;
    aux.push(s);
    vis[s] = true;
    for (auto a : adj[s]) {
        if (!dfsnum[a]) tarjan_dfs(a);
        if (vis[a]) dfslow[s] = min(dfslow[s], dfslow[a]);
    }
    if (dfslow[s] == dfsnum[s]) {
        SCC += 1;
        while (1) {
            int v = aux.top();
            aux.pop();
            vis[v] = 0;
            if (s == v) break;
        }
    }
}

void scc() {
    aux    = stack<int>();
    dfslow = vi(V, 0);
    dfsnum = vi(V, 0);
    memset(vis, false, sizeof(vis));
    TIME = SCC = 0;
    for (int i = 0; i < V; i++) {
        if (!dfsnum[i]) tarjan_dfs(i);
    }
}
```

3.3 Edmonds-Karp

```
// Edmonds-Karp Algorithm
// Min-Cut/Max-Flow problem

int         V;
vector<vi> adj;
vector<vi> capacity;

int bfs(int s, int t, vi& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});
    while (!q.empty()) {
        int cur  = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({next, new_flow});
            }
        }
```

```
    }
    return 0;
}

int maxflow(int s, int t) {
    int flow = 0;
    vi  parent(V);
    int new_flow;
    while ((new_flow = bfs(s, t, parent))) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
```