Biblioteca da Maratona de Gabriel Pessoa :)

Contents

1 String Algorithms

1.1 String Alignment

```
int pd[ms][ms];

int edit_distance(string &a, string &b) {
    int n = a.size(), m = b.size();
    for(int i = 0; i <= n; i++) {
        pd[i][0] = i;
    }
    for(int j = 0; j <= m; j++) {
        pd[0][j] = j;
    }
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= m; j++) {
            int del = pd[i][j - 1] + 1;
            int ins = pd[i - 1][j] + 1;
            int mod = pd[i - 1][j - 1] + a[i - 1] != b[j - 1];
            pd[i][j] = min(del, min(ins, mod));
        }
    }
    return pd[n][m];
}
```

1.2 KMP

```
string p, t;
int b[ms], n, m;

void kmpPreprocess() {
    int i = 0, j = -1;
    b[0] = -1;
    while(i < m) {
        while(j >= 0 && p[i] != p[j]) j = b[j];
        b[++i] = ++j;
    }
}

void kmpSearch() {
    int i = 0, j = 0, ans = 0;
    while(i < n) {
        while(j >= 0 && t[i] != p[j]) j = b[j];
        i++; j++;
        if(j == m) {
            //ocorrencia aqui comecando em i - j
            ans++;
            j = borda[j];
        }
    }
}
```

1.3 Trie

```
int trie[ms][sigma], terminal[ms], z;

void init() {
    memset(trie[0], -1, sizeof trie[0]);
    z = 1;
}

int get_id(char c) {
    return c - 'a';
}

void insert(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            memset(trie[z], -1, sizeof trie[z]);
            trie[cur][id] = z++;
        }
        cur = trie[cur][id];
    }
    terminal[cur]++;
}

int count(string &p) {
    int cur = 0;
    for(int i = 0; i < p.size(); i++) {
        int id = get_id(p[i]);
        if(trie[cur][id] == -1) {
            return false;
        }
        cur = trie[cur][id];
    }
    return terminal[cur];
}
```

1.4 Aho-Corasick

```
// Construa a Trie do seu dicionario com o codigo acima

int fail[ms];

void buildFailure() {
    queue<int> q;
    q.push(0);
    while(!q.empty()) {
        int node = q.front();
        q.pop();
        for(int pos = 0; pos < sigma; pos++) {
            int &v = trie[node][pos];
            int f = max(0, trie[fail[node]][pos]);
            if(v == -1) {
                v = f;
            } else {
                fail[v] = f;
                q.push(v);
                // juntar as informacoes da borda para o V ja q um match em V implica um match na
                    borda
                terminal[v] += terminal[f];
            }
        }
```

```
    }
}

int search(string &txt) {
    int node = 0;
    int ans = 0;
    for(char c : txt) {
        int pos = get_id(c);
        node = trie[node][pos];
        // processar informacoes no no atual
       ans += terminal[node];
    }
    return ans;
}
```

2 Trees

2.1 BIT - Binary Indexed Tree

```
int arr[ms], bit[ms], n;

void update(int v, int idx) {
    while(idx <= n) {
        bit[idx] += v;
        idx += idx & -idx;
    }
}

int query(int idx) {
    int r = 0;
    while(idx > 0) {
        r += bit[idx];
        idx -= idx & -idx;
    }
    return r;
}
```

2.2 Iterative Segment Tree

```
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int p, int value) { // set value at position p
    for(t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) {
    int res = 0;
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) res += t[l++];
        if(r&1) res += t[--r];
    }
    return res;
}

// If is non-commutative
S query(int l, int r) {
  S resl, resr;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) resl = combine(resl, t[l++]);
    if (r&1) resr = combine(t[--r], resr);
  }
  return combine(resl, resr);
}
```

2.3 Iterative Segment Tree with Interval Updates

```
int n, t[2 * ms];

void build() {
    for(int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void update(int l, int r, int value) {
    for(l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if(l&1) t[l++] += value;
        if(r&1) t[--r] += value;
    }
}

int query(int p) {
    int res = 0;
    for(p += n; p > 0; p >>= 1) res += t[p];
    return res;
}

void push() { // push modifications to leafs
    for(int i = 1; i < n; i++) {
        t[i<<1] += t[i];
        t[i<<1|1] += t[i];
        t[i] = 0;
    }
}
```

2.4 Recursive Segment Tree

```
int arr[4 * ms], seg[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}

void update(int V, int I, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l > I || r < I) return;
    if(l == r) {
        arr[I] = V;
        seg[idx] = V;
        return;
    }
    update(V, I, left, l, mid); update(V, I, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

2.5 Segment Tree with Lazy Propagation

```
int arr[4 * ms],  seg[4 * ms], lazy[4 * ms], n;

void build(int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(l == r) {
        seg[idx] = arr[l];
        return;
    }
    build(left, l, mid); build(right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}

int query(int L, int R, int idx = 0, int l = 0, int r = n - 1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1);
        if(l < r) {
            lazy[left] += lazy[idx];
            lazy[right] += lazy[idx];
        }
        lazy[idx] = 0;
    }
    if(R < l || L > r) return 0;
    if(L <= l && r <= R) return seg[idx];
    return query(L, R, left, l, mid) + query(L, R, right, mid + 1, r);
}
```

```
void update(int V, int L, int R, int idx = 0, int l = 0, int r = n -1) {
    int mid = (l+r)/2, left = 2 * idx + 1, right = 2 * idx + 2;
    if(lazy[idx]) {
        seg[idx] += lazy[idx] * (r - l + 1);
        if(l < r) {
            lazy[left] += lazy[idx];
            lazy[right] += lazy[idx];
        }
        lazy[idx] = 0;
    }
    if(l > R || r < L) return;
    if(L <= l && r <= R) {
        seg[idx] += V * (r - l + 1);
        if(l < r) {
            lazy[left] += V;
            lazy[right] += V;
        }
        return;
    }
    update(V, L, R, left, l, mid); update(V, L, R, right, mid + 1, r);
    seg[idx] = seg[left] + seg[right];
}
```

3 Graph Algorithms

3.1 Graph Structure

```
const int ms = 1e3; // Quantidade maxima de vertices
const int me = 1e5; // Quantidade maxima de arestas

int adj[ms], to[me], ant[me], wt[me], z, n;

void clear() {
    memset(adj, -1, sizeof adj);
    z = 0;
}

void add(int u, int v, int w = 1) {
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = w;
    adj[u] = z++;
}
```

3.2 DFS / Toposort

```
int vis[ms], topo[ms], topoLen;

void dfs(int v) {
    if(vis[v]) return;
    vis[v] = true;
    for(int i = adj[v]; i > -1; i = ant[i]) dfs(to[i]);
    topo[topoLen++] = v;
}

void dfs() {
    memset(vis, 0, sizeof vis);
    for(int i = 0; i < n; i++) dfs(i);
    reverse(topo, topo + n);
}
```

3.3 BFS / Shortest Path in a Unweighted Graph

```
int dis[ms], q[ms], front, rear;

void bfs(int x) {
    memset(vis, 0, sizeof dis);
    dis[x] = 0; front = 0; rear = 0;
    q[rear++] = x;
    while(front < size) {
        int v = fila[front++];
        for(int i = adj[v]; i > -1; i = ant[i]) {
            if(vis[to[i]]) continue;
            vis[to[i]] = true;
            dis[to[i]] = dis[v] + 1;
            q[rear++] = to[i];
        }
    }
}
```

3.4 Dijkstra / Shortest Path in a Weighted Graph

```
typedef pair<int, int> ii;

int dis[ms];
priority_queue<ii, vector<ii>, greater<ii>> pq;

void dijkstra(int x) {
    memset(dis, 63, sizeof dis);
    dis[x] = 0;
    pq.push(ii(0, x));
    while(!pq.empty()) {
        ii x = pq.top(); pq.pop();
        v = x.second;
        if(x.first > dis[v]) continue;
        for(int i = adj[v]; i > -1; i = ant[i]) {
            if(dis[v]+wt[i] < dis[to[i]]) {
                dis[to[i]] = dis[v] + wt[i];
                pq.push(ii(dis[to[i]], to[i]));
            }
        }
    }
}
```

3.5 Dinic Max Flow

```
int copy_adj[ms], fila[ms], level[ms];

void clear() {
    memset(adj, -1, sizeof adj);
    z = 0;
}

int add(int u, int v, int k) {
    to[z] = v;
    ant[z] = adj[u];
    wt[z] = k;
    adj[u] = z++;
}

int bfs(int source, int sink) {
        memset(level, -1, sizeof level);
        level[source] = 0;
        int front = 0, size = 0, v;
        fila[size++] = source;
        while(front < size) {
                v = fila[front++];
                for(int i = adj[v]; i != -1; i = ant[i]) {
                        if(wt[i] && level[to[i]] == -1) {
                                level[to[i]] = level[v] + 1;
                                fila[size++] = to[i];
                        }
                }
        }
        return level[sink] != -1;
}

int dfs(int v, int sink, int flow) {
        if(v == sink) return flow;
        int f;
        for(int &i = copy_adj[v]; i != -1; i = ant[i]) {
                if(wt[i] && level[to[i]] == level[v] + 1 && (f = dfs(to[i], sink, min(flow, wt[i]))))
                        {
                        wt[i] -= f;
                        wt[i ^ 1] += f;
                        return f;
                }
        }
        return 0;
}

int maxflow(int source, int sink) {
        int ret = 0, flow;
        while(bfs(source, sink)) {
                memcpy(copy_adj, adj, sizeof adj);
                while((flow = dfs(source, sink, 1 << 30))) {
                        ret += flow;
```

```
            }
        }
        return ret;
    }
```

3.6 Articulations Points and Bridges

```
int idx[ms], art[ms], bridge[me], ind, child;

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            int temp = dfs(to[i], v);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par) low = min(low, idx[w]);
    }
    return low;
}

void artPointAndBridge() {
    ind = 0;
    memset(idx, -1, sizeof idx);
    memset(art, 0, sizeof art);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
        art[i] = child > 1;
    }
}
```

3.7 Biconnected Components

```
int idx[ms], bc[me], ind, nbc, child, st[me], top;

void generateBc(int edge) {
    while(st[--top] != edge) {
        bc[st[top]] = nbc;
    }
    bc[edge] = nbc++;
}

int dfs(int v, int par = -1) {
    int low = idx[v] = ind++;
    for(int i = adj[v]; i > -1; i = ant[i]) {
        if(idx[to[i]] == -1) {
            if(par == -1) child++;
            st[top++] = i;
            int temp = dfs(to[i], v);
            if(par == -1 && child > 1 || ~par && temp >= idx[v]) generateBc(i);
            if(temp >= idx[v]) art[v] = true;
            if(temp > idx[v]) bridge[i] = true;
            low = min(low, temp);
        } else if(to[i] != par && idx[to[i]] < low) {
            low = idx[to[i]];
            st[top++] = i;
        }
    }
    return low;
}

void biconnected() {
    ind = 0;
    nbc = 0;
    top = -1;
    memset(idx, -1, sizeof idx);
    for(int i = 0; i < n; i++) if(idx[i] == -1) {
        child = 0;
        dfs(i);
    }
}
```

3.8 SCC - Strongly Connected Components / 2SAT

```
int idx[ms], low[ms], ind, comp[ms], ncomp, st[ms], top;

int dfs(int v) {
    if(~idx[v]) return idx[v] ? idx[v] : ind;
    low[v] = idx[v] = idx++;
    st[top++] = v;
    onStack[v] = true;
    for(int w = adj[x]; ~w; w = ant[w]) {
        low[v] = min(low[v], dfs(to[w]));
    }
    if(low[v] == idx[v]) {
        while(top > -1) {
            int w = st[--top];
            idx[w] = 0;
            low[w] = low[v];
            comp[w] = ncomp;
        }
        ncomp++;
    }
    return low[v];
}

bool solveSat() {
    memset(idx, -1, sizeof idx);
    ind = 1; top = -1;
    for(int i = 0; i < n; i++) dfs(i);
    for(int i = 0; i < n; i++) if(low[i] == low[i^1]) return false;
    return true;
}

// Operacoes comuns de 2-sat
// ~v = "nao v"
#define trad(v) (v<0?((~v)*2)^1:v*2)
void addImp(int a, int b) { add(trad(a), trad(b)); }
void addOr(int a, int b) { addImp(~a, b); addImp(~b, a); }
void addEqual(int a, int b) { addOr(a, ~b); addOr(~a, b); }
void addDiff(int a, int b) { addEqual(a, ~b); }
// valoracao: value[v] = comp[trad(v)] < comp[trad(~v)]
```

3.9 LCA - Lowest Common Ancestor

```
int par[ms][mlg], lvl[ms];

void dfs(int v, int p, int l = 0) {
    lvl[v] = l;
    par[v][0] = p;
    for(int i = adj[v]; i > - 1; i = ant[i]) {
        if(to[i] != p) dfs(to[i], v, l + 1);
    }
}

void processAncestors(int root = 0) {
    dfs(root, root);
    for(int k = 1; k <= mlg; k++) {
        for(int i = 0; i < n; i++) {
            par[i][k] = par[par[i][k-1]][k-1];
        }
    }
}

int lca(int a, int b) {
    if(lvl[b] > lvl[a]) swap(a, b);
    for(int i = mlg; i >= 0; i--) {
        if(lvl[a] - (1 << i) <= lvl[b]) a = par[a][i];
    }
    if(a == b) return a;
    for(int i = mlg; i >= 0; i--) {
        if(par[a][i] != par[b][i]) a = par[a][i], b = par[b][i];
    }
    return par[a][0];
}
```

3.10 Floyd-Warshall / Shortest path between all pairs

```
const int inf = 0x3f3f3f3f;
int g[ms][ms], dis[ms][ms], n;

void clear() {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            g[i][j] = i == j ? 0 : inf;
        }
    }
```

```cpp
}

void add(int u, int v, int w) {
    g[u][v] = min(w, g[u][v]);
}

void floydWarshall() {
    memcpy(g, dis, sizeof g);
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
            }
        }
    }
}
```

3.11 Disjoint-Set / Union-Find

```cpp
int ds[ms], sz[ms], n;

void dsBuild() {
    for(int i = 0; i < n; i++) {
        ds[i] = i;
    }
}

int dsFind(int i) {
    while(ds[i] != i) {
        ds[i] = ds[ds[i]];
        i = ds[i];
    }
}

void dsUnion(int a, int b) {
    a = dsFind(a);
    b = dsFind(b);
    if(sz[a] < sz[b]) swap(a, b);
    sz[a] += sz[b];
    ds[b] = a;
}
```

3.12 Kruskal's MST - Minimum Spanning Tree

```cpp
// Usa a estrutura de Disjoint-Set acima

typedef pair<int, int> ii;
typedef pair<int, ii> iii;

iii e[me], mst[me];
int z, mstLen;

void add(int u, int v, int w) {
    e[z++] = iii(u, ii(v, w));
}

int kruskal() {
    int ans = 0;
    dsBuild();
    sort(e, e + z);
    for(auto i : e) {
        int u = i.second.first, v = i.second.second, w = i.first;
        if(dsFind(u) != dsFind(v)) {
            dsUnion(u, v);
            ans += w;
            mst[mstLen++] = i;
        }
    }
    return ans;
}
```

4 Miscellaneous

4.1 LIS - Longest Increasing Subsequence

```cpp
int arr[ms], lisArr[ms], n;
// int bef[ms], pos[ms];

int lis() {
    int len = 1;
    lisArr[0] = arr[0];
    // bef[0] = -1;
    for(int i = 1; i < n; i++) {
        // upper_bound se non-decreasing
        int x = lower_bound(lisArr, lisArr + len, arr[i]) - lisArr;
        len = max(len, x + 1);
        lisArr[x] = arr[i];
        // pos[x] = i;
        // bef[i] = x ? pos[x-1] : -1;
    }
    return len;
}

vi getLis() {
    int len = lis();
    vi ans;
    for(int i = pos[lisArr[len - 1]]; i >= 0; i = bef[i]) {
        ans.push_back(arr[i]);
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

4.2 Binary Search

```cpp
int smallestSolution() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(!ok(x+b)) x += b;
    }
    int k = x + 1;
}

int maximumValue() {
    int x = -1;
    for(int b = z; b >= 1; b /= 2) {
        while(f(x+b) < f(x+b+1)) x += b;
    }
    int k = x + 1;
}
```