Contents

1 Graphs 1
- 1.1 Dijkstra 1
- 1.2 Warshall 1
- 1.3 SPFA 2
- 1.4 MST - Kruskal 2
- 1.5 SCC - Tarjan 3
- 1.6 Topological Sort - Tarjan . 3
- 1.7 Articulation points and Bridges 3
- 1.8 Bipartite checking 4
- 1.9 MCBM - Augment 4
- 1.10 Hopcroft 5
- 1.11 Minimum vertex cover - MVC 5
- 1.12 Max Flow - Edmonds 6
- 1.13 Max Flow - Dinic 7
- 1.14 Min Cost - Max Flow (Edmonds + SPFA) 8
- 1.15 Euler tour 9

2 Trees 9
- 2.1 Diameter 9
- 2.2 Create from degree array . 9

3 DP 10
- 3.1 LCS 10
- 3.2 LIS - NlgN 10
- 3.3 MAX 2D 10
- 3.4 Count change 11

4 Data Structures 11
- 4.1 Max SegTree (Simple) 11
- 4.2 Sum SegTree (Lazy) 12
- 4.3 ETT in SegTree 12
- 4.4 Trie (Complete) 13

5 String 14
- 5.1 KMP 14

6 Misc 15
- 6.1 MOS - Distinct elements . . 15
- 6.2 Primes generator 16
- 6.3 Inversions vector nlgn . . 16

7 Theorems 17

8 Template 17

1 Graphs

1.1 Dijkstra

```
/*
 * O(ElgV)
 *
 * Setar vet_adj, V e E.
 * resposta: vetor dist
 *
 */

#define INF 100000000000000

int V, E;

/* pair com o id do vizinho e a distancia ate ele */
vector<list<pair<int, ll>>> vet_adj;
vll dist;


ll sssp(int source, int target){

  dist.clear();
    dist.resize(V, INF);
    dist[source] = 0;

    priority_queue<pair<ll, int>, vector<pair<ll, int>>,
     greater<pair<ll, int>>> pq;
    pq.push(mp(0, source));

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

        if(dist[u] < d_u) continue;

        for(auto &pv :  vet_adj[u]){
            int v = pv.first;
            ll u_v = pv.second;

            if(d_u + u_v < dist[v]){
                dist[v] = d_u + u_v;
                pq.push(mp(dist[v], v));
            }
        }
    }

    return dist[target];
}
```

1.2 Warshall

```
/*
 * O(V^3)
 *
 * Setar V e adj_matrix.
 *
 * Adj_matrix com custo de i -> i = 0 e vertices sem
    aresta com custo INF
 *
 */

#define INF 1000000000

int V;
vector<vi> adj_matrix;

void floyd () {

  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
```

```
      for (int j = 0; j < V; j++)
            adj_matrix[i][j] = min(adj_matrix[i][j],
    adj_matrix[i][k] + adj_matrix[k][j]);
}
```

1.3 SPFA

```
/*
 * Nao testado
 *
 * Setar V e vet_adj
 * A resposta eh dada no vetor global dist
 *
 * */


vi dist;
int V;
vector<vii> vet_adj;

void SPFA(int source) {

        dist.assign(V, INF);
        dist[source] = 0;

        vi in_queue(V, 0);
        queue<int> q;

        q.push(source);
        in_queue[source] = 1;


        while (!q.empty()) {

            int u = q.front();
            q.pop();
            in_queue[u] = 0;

            for (auto &edge : vet_adj[u]) {

                int v = edge.first;
                int u2v = edge.second;

                if (dist[u] + u2v < dist[v]) {

                    dist[v] = dist[u] + u2v; // relax

                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = 1;
                    }

                }
            }
        }
}
```

1.4 MST - Kruskal

```
/*
 * O (ElgV)
 *
 * Setar UF com os vertices iniciais.
 * Criar edge_list e ordena-la.
 *
 */


class UnionFind {
private:
  vi rep, rank, set_sz;
  int n_sets;
public:
    UnionFind(int N) {
      set_sz.assign(N, 1);
      n_sets = N;
      rank.assign(N, 0);
      rep.assign(N, 0);
      for (int i = 0; i < N; i++) {
        rep[i] = i;
      }
    }

    int find_set(int i) {

      if (rep[i] == i) {
        return i;
      }

      rep[i] = find_set(rep[i]);

      return rep[i];

    }

    bool is_same_set(int i, int j) {
      return find_set(i) == find_set(j);
    }

    void union_by_rank(int i, int j) {
      if (!is_same_set(i, j)) {

        n_sets--;
        int rep_i = find_set(i);
        int rep_j = find_set(j);


        if (rank[rep_i] > rank[rep_j]) {

          rep[rep_j] = rep_i;
          set_sz[rep_i] += set_sz[rep_j];

        } else {

          rep[rep_i] = rep_j;
          set_sz[rep_j] += set_sz[rep_i];

          if (rank[rep_i] == rank[rep_j]) {
            rank[rep_j]++;
          }

        }
      }
    }

    int sets_count() {
      return n_sets;
    }

};

struct Edge{
  int u, v;
  int weight;

  bool operator < (Edge const& other) const {
        return this->weight < other.weight;
    }
};

/* USAGE: */

int main() {
  int mst_cost = 0;
  UnionFind UF(V);
  auto it = edge_list.begin();  // already sorted
    while (it != edge_list.end() && UF.sets_count() > 1)
    {
```

```
        Edge curr_edge = *(it++);
    if (!UF.is_same_set(curr_edge.u, curr_edge.v)) {
      mst_cost += curr_edge.weight;
      UF.union_by_rank(curr_edge.u, curr_edge.v);
    }
  }
}
```

1.5 SCC - Tarjan

```
/*
 *  O (V + E)
 *
 *  Setar V e vet_adj
 *  Resposta : vetor SCC contem as componentes.
 */

int V;
vector<set<int>> vet_adj;

void SCC_aux(int u, int &dfs_time, vi &visited, vi &
    visit_order, vi &discovery_time, vi &
    lowest_discovery_reachable, vector<vi> &SCCs) {

    discovery_time[u] = ++dfs_time;
    lowest_discovery_reachable[u] = discovery_time[u];

    visit_order.push_back(u);

    visited[u] = 1;

    for (auto v : vet_adj[u]) {

        if (discovery_time[v] == 0) {
            SCC_aux(v, dfs_time, visited, visit_order,
    discovery_time, lowest_discovery_reachable, SCCs);
        }

        if (visited[v]) {
            lowest_discovery_reachable[u] = min(
    lowest_discovery_reachable[u],
    lowest_discovery_reachable[v]);
        }
    }


    if (discovery_time[u] == lowest_discovery_reachable[u
    ]) {

        vi new_scc;
        SCCs.pb(new_scc);
        int v;

        do {
            v = visit_order.back();
            visit_order.pop_back();
            visited[v] = 0;
            SCCs.back().pb(v);
        } while (u != v);

    }

}

void SCC(vector<vi> &SCCs) {

    SCCs.clear(); // Para recuperar de fato o cada SCC

    vi visit_order;

    vi discovery_time(V, 0);
    vi lowest_discovery_reachable(V, 0);

    vi visited(V, 0);
    int dfs_time = 0;

    for (int i = 0; i < V; ++i) {
        if (discovery_time[i] == 0) {
            SCC_aux(i, dfs_time, visited, visit_order,
    discovery_time, lowest_discovery_reachable, SCCs);
        }
    }

}
```

1.6 Topological Sort - Tarjan

```
/*
 * O (V + E)
 *
 * Setar V e vet_adj;
 * resposta dada em sorted_vertices
 *
*/

vector<vi> vet_adj;
vi visited;
vi sorted_vertices;
int V;

void aux_dfs(int root){

    visited[root] = 1;

    for(auto &v : vet_adj[root]){
        if(!visited[v])
            aux_dfs(v);
    }

    sorted_vertices.push_back(root);
}

void topological_sort(){

  visited.clear();
  visited.resize(V, 0);

  sorted_vertices.clear();
  sorted_vertices.reserve(V);

    for (size_t i = 0; i < V; i++) {
        if(!visited[i]){
            aux_dfs(i);
        }
    }

    reverse(all(sorted_vertices));

}
```

1.7 Articulation points and Bridges

```
/*
  Setar V e vet_adj.
  Alterar funcao para retornar as bridges
*/

int V;
vector<set<int>> vet_adj;
// int bridge_edge[][]

void aux_AP_and_bridges(int u, int &dfs_time, int
    root_dfs, vi &discovery_time, vi &
    lowest_discovery_reachable,
                        vi &articulation_vertex, vi &
    parent) {
```

```
    lowest_discovery_reachable[u] = ++dfs_time;
    discovery_time[u] = lowest_discovery_reachable[u];

    int root_dfs_children = 0;

    for (auto v : vet_adj[u]) {
        if (discovery_time[v] == 0) {
            parent[v] = u;

            if (u == root_dfs) {  // Tratando caso raiz
    do DFS
                root_dfs_children++;

                if (root_dfs_children > 1) {
                    articulation_vertex[u] = 1;
                }
            }

            aux_AP_and_bridges(v, dfs_time, root_dfs,
    discovery_time, lowest_discovery_reachable,
                               articulation_vertex,
    parent);

            if (u != root_dfs &&
    lowest_discovery_reachable[v] >= discovery_time[u])
    {
                articulation_vertex[u] = 1;
            }

            // FOR bridge
            //  if (lowest_discovery_reachable[v] >
    discovery_time[u])
            //    bridge_edge[u][v] = briged_edge[v][u] =
     true;

            lowest_discovery_reachable[u] = min(
    lowest_discovery_reachable[u],
    lowest_discovery_reachable[v]);
        } else if (v != parent[u]) {
            lowest_discovery_reachable[u] = min(
    lowest_discovery_reachable[u], discovery_time[v]);
        }
    }
}

void AP_and_bridges(vi &articulation_vertex) {

    articulation_vertex.clear();
    articulation_vertex.resize(V);

    vi discovery_time(V, 0);
    vi lowest_discovery_reachable(V);
    vi parent(V, 0);
    int dfs_time = 0;

    aux_AP_and_bridges(0, dfs_time, 0, discovery_time,
                       lowest_discovery_reachable,
    articulation_vertex, parent);
}
```

1.8 Bipartite checking

```
/*
 * O (V + E)
 * Setar vet_adj e V.
 *Esse algoritmo apenas conta (minimizando) o LeftSide de
    um grafo bipartido (se for possivel, possible =
    false).
*/

vi color(V, -1);
bool possible = true;
int res = 0;
for (int i = 0; possible && i < V; ++i) {

    if (color[i] == -1) {
        int one_count = 0, zero_count = 0;

        queue<int> q;

        color[i] = 0;
        zero_count = 1;
        q.push(i);

        while (!q.empty()) {
            int u = q.front();
            q.pop();


            for (auto &v : vet_adj[u]) {
                if (color[v] == color[u]) {
                    possible = false;
                    break;
                } else if (color[v] == -1) {
                    color[v] = !(bool)color[u];

                    if (color[v] == 0) {
                        ++zero_count;
                    } else {
                        ++one_count;
                    }

                    q.push(v);
                }
            }

            if (!possible) { break; }
        }

        res += min(one_count, zero_count);

        if (min(one_count, zero_count) == 0) {
            res += max(one_count, zero_count);
        }
    }

}
```

1.9 MCBM - Augment

```
/*
 * O(VE)
 *
 * Setar vet_adj, V e V_left.
 * Resposta pode ser encontrada no vetor match.
 *
 * */

vi match;
vector<vi> vet_adj;
vi visited;

bool aug(int u) {

    if (visited[u])
    return false;

    visited[u] = 1;

  for (auto &v : vet_adj[u]) {

        if (match[v] == -1 || aug(match[v])) {
            match[v] = u;
            return true;
        }
    }
```

```
  return false;
}


int MCBM() {

  int MCBM = 0;
    match.assign(V, -1);              // V is the number of
     vertices in bipartite graph
  for (int l = 0; l < V_left; l++) {               //
    Vleft = size of the left set
        visited.assign(V_left, 0);
      MCBM += aug(l);
  }

  return MCBM;
}
```

1.10 Hopcroft

```
/*
 *  O(sqrt(V)E)
 *
 *  Setar V_left e V_right
 *  Nao usar o vertice 0, pois esse sera o vertice dummy
    do algoritmo
 *  Inserir arestas no grafo sempre da esquerda para a
    direita.
*/

#define INF 1000000000

vector<vi> vet_adj;
int V_left, V_right;

vi match;
vi dist;

bool bfs() {

    queue<int> q;
    dist.assign(V_left + V_right + 1, INF);

    // comeca um BFS a partir de todo vertice livre (i.e.
     p[u] == 0) da esquerda
    for (int i = 1; i <= V_left; i++) {
        if (match[i] == 0) {
            dist[i] = 0;
            q.push(i);
        }
    }

    while(!q.empty()) {

        int u = q.front();
        q.pop();

        if (u == 0) { return true; } // cheguei no dummy,
     significa que cheguei em alguem na direita sem
    match

        for(auto &v : vet_adj[u]) {

            if (dist[match[v]] == INF) { // perceba que
    usamos o vetor match para descobrir caminhos
    alternados

                q.push(match[v]);
                dist[match[v]] = dist[u] + 1;

            }
        }

    }
    return false;
}

bool dfs(int u) {

    if (u == 0) { return true; } // cheguei no dummy,
    significa que cheguei em alguem na direita sem match


    for(auto &v : vet_adj[u]) {

        if (dist[u] + 1 == dist[match[v]]) {
            if (dfs(match[v])) {

                match[u] = v;
                match[v] = u;

                return true;
            }
        }

    }


    // Se chegou aqui, o vertice u nao tem mais caminhos
    para oferecer, entao ja invalidamos ele
    dist[u] = INF;
    return false;
}

int hopcroft() {

    match.assign(V_left + V_right + 1, 0);
    int matching = 0;
    while (bfs()) {

        for (int i = 1; i <= V_left; i++) {

            if (match[i] == 0) {

                if (dfs(i)) {
                    matching++;
                }

            }

        }

    }

    return matching;
}
```

1.11 Minimum vertex cover - MVC

```
/*
 * O(V + E)  // Mas o algoritmo aumentador eh O(VE)
 *
 * Setar o vetor match com algoritmo aumentador
 *
 * MVC = Vertices a esquerda nao visitados + vertices a
    direita visitados durante um
 * DFS alternado em um MCBM
*/

#define LEFT_TYPE 0
#define RIGHT_TYPE 1

vector<vi> vet_adj;
vi match;
vi matched; // will track left vertex that did not
    matched
vi visited;
```

```
int V_left, V_right, N;

void alternate_dfs_aux(int u, int type) {

    if (type == LEFT_TYPE) {

        for (auto v : vet_adj[u]) {
            if (!visited[v]) {
                visited[v] = 1;
                alternate_dfs_aux(v, RIGHT_TYPE);
            }
        }

    } else {

        if (!visited[match[u]]) {
            visited[match[u]] = 1;
            alternate_dfs_aux(match[u], LEFT_TYPE);
        }

    }

}

void alternate_dfs() {

    visited.assign(V_left + V_right, 0);

    for (int i = 0; i < V_left; ++i) {
        if (!matched[i]) {
            visited[i] = 1;
            alternate_dfs_aux(i, LEFT_TYPE);
        }
    }
}

vi min_vertex_cover() {

    match.assign(V_left + V_right, -1);
    for (int l = 0; l < V_left; l++) {
        visited.assign(V_left, 0);
        aug(l);
    }

    matched.assign(V_left, 0);
    for (int u : match) {
        if (u != - 1) {
            matched[u] = 1;
        }
    }

    alternate_dfs();

    vi result;

    for (size_t i = 0; i < V_left; i++) {
        if (!visited[i]) {
            result.pb(i);
        }
    }

    for (size_t i = 0; i < V_right; i++) {
        if (visited[i + V_left]) {
            result.pb(i + V_left);
        }
    }

    return result;
}
```

1.12 Max Flow - Edmonds

```
/*
 * min(O(VE^2), O(flow*E))
 *
 * Setar V
 * Resetar/Setar edge_list no tamanho de V
 * Usar put_edge para ligar os vertices
 * */

#define INF 1000000000

struct Edge{
    int dest;
    ll capacity;
    int cancel_edge; // id da reverse edge associada

    Edge(int x, ll y, int z) : dest(x), capacity(y),
    cancel_edge(z){}
};

vector<vector<Edge>> edge_list;
int V;

void put_edge(int u, int v, ll capacity) {

    edge_list[u].push_back(Edge(v, capacity, edge_list[v
    ].size()));
    edge_list[v].push_back(Edge(u, 0, edge_list[u].size()
     - 1));

}

void put_edge_undirected(int u, int v, ll capacity) {

    edge_list[u].push_back(Edge(v, capacity, edge_list[v
    ].size()));
    edge_list[v].push_back(Edge(u, capacity, edge_list[u
    ].size() - 1));

}

ll augment(int v, vi &prev_vertex, vi &prev_edge, ll
    min_edge) {

    if (prev_vertex[v] == -1) {

        return min_edge;

    } else {

        int u = prev_vertex[v];
        Edge &edge = edge_list[u][prev_edge[v]];

        ll curr_flow = augment(u, prev_vertex, prev_edge,
     min(min_edge, edge.capacity)); // recursive

        edge.capacity -= curr_flow;
        edge_list[v][edge.cancel_edge].capacity +=
    curr_flow;

        return curr_flow;

    }

}

ll max_flow(int source, int target) {

    ll max_flow = 0;

    while (true) {

        vi dist(V, INF);
        queue<int> q;
        vi prev_vertex(V, -1);
        vi prev_edge(V, -1);

        dist[source] = 0;
        q.push(source);
```

```cpp
        while (!q.empty()) {
            int u = q.front();
            q.pop();

            if (u == target) { break; }

            for (int i = 0; i < edge_list[u].size(); ++i)
     {
                auto &edge = edge_list[u][i];

                if (edge.capacity > 0 && dist[edge.dest]
    == INF) {

                    dist[edge.dest] = dist[u] + 1;
                    q.push(edge.dest);

                    prev_vertex[edge.dest] = u;
                    prev_edge[edge.dest] = i;
                }
            }
        }

        if (dist[target] != INF) {
            max_flow += augment(target, prev_vertex,
    prev_edge, INF);
        } else {
            break;
        }
    }

    return max_flow;

}
```

1.13 Max Flow - Dinic

```cpp
/*
 * min(O(V^2E), O(flow*E))
 *
 * Setar V
 * Resetar/Setar edge_list no tamanho de V
 * Usar put_edge para ligar os vertices
 * */

#define INF 1000000000

struct edge{
    int dest;
    int capacity;
    int cancel_edge; // id da reverse edge associada

    edge(int x, int y, int z) : dest(x), capacity(y),
    cancel_edge(z){}
};

int V;
vi next_neighbor;
vi dist;
vector<vector<edge>> edge_list;

void put_edge(int u, int v, int capacity)
{
    edge_list[u].push_back(edge(v, capacity, edge_list[v
    ].size()));
    edge_list[v].push_back(edge(u, 0, edge_list[u].size()
     - 1));
}

void put_edge_undirected(int u, int v, int capacity)
{
    edge_list[u].push_back(edge(v, capacity, edge_list[v
    ].size()));
    edge_list[v].push_back(edge(u, capacity, edge_list[u
    ].size() - 1));
}

bool bfs(int source, int target) {

    queue<int> q;
    q.push(source);

    dist.assign(V, INF);
    dist[source] = 0;

    while(!q.empty()) {

        int u = q.front();
        q.pop();

        // se a bfs chega no sorvedouro podemos retornar
    porque os vertices que nao estao no menor caminho
    para o sorvedouro nao importam
        if (u == target) { return true; }

        for(auto &e : edge_list[u]) {

            if(e.capacity > 0 && dist[e.dest] == INF) {
    // percorre todas as arestas que ainda podem passar
    fluxo

                dist[e.dest] = dist[u] + 1;
                q.push(e.dest);

            }
        }
    }


    return false;
}

int dfs(int u, int flow, int target)
{
    if (u == target) {
        return flow; // encontramos um caminho aumentante
    }

    for (int &i = next_neighbor[u]; i < edge_list[u].size
    (); i++) { //ignora arestas ja percorridas

        edge &e = edge_list[u][i];

        if (dist[u] + 1 == dist[e.dest] && e.capacity >
    0) { // so queremos as arestas que fazem parte de um
     caminho minimo e podem passar fluxo

            int rec_flow = dfs(e.dest, min(flow, e.
    capacity), target);

            if (rec_flow == 0) { continue; }

            e.capacity -= rec_flow; // Passa fluxo pelo
    caminho aumentante encontrado.
            edge_list[e.dest][e.cancel_edge].capacity +=
    rec_flow; // Essa linha nao afeta as proximas
    iteracoes da dfs porque a aresta reversa nao esta em
     um caminho minimo.

            return rec_flow;
        }

    }


    dist[u] = INF; // Se chegou aqui, esgotou-se as
    opcoes para esse vertice, vamos marca-lo como inutil
    return 0;
}
```

```
long long dinic(int source, int target) {

    ll flow = 0;
  V = edge_list.size();

    while (bfs(source, target)) {

        next_neighbor.assign(V, 0);
        while (int path_flow = dfs(source, INF, target))
    {
            flow += path_flow;

        }

    }

    return flow;
}
```

1.14 Min Cost - Max Flow (Edmonds + SPFA)

```
/*
 *
 * O(kVE^2) ???
 *
 * Carece de maiores testes
 *
 * Setar V
 * Resetar/Setar edge_list no tamanho de V
 * Usar put_edge para ligar os vertices
 *
 *
 * Esse algoritmo tambem funciona para redes de transporte (vertices com demandas), mas lembre-se:
 *      * Se a soma das demandas nÃčo for zero, crie um vertice dummy que a zere
 *      * Crie um source e um sink, O source eh ligado a todo vertice de demanda positiva (e fornece a mesma para eles)
 *        e o sink eh ligado a todo vertice de demanda negativa (e suga a mesma deles)
 *
 * */


#define INF 1000000000

struct Edge{
    int dest;
    int capacity;
    int cost;
    int cancel_edge; // id da reverse edge associada

    Edge(int x, int y, int c, int z) : dest(x), capacity(y), cost(c), cancel_edge(z){}
};

vector<vector<Edge>> edge_list;
int V;

void put_edge(int u, int v, int capacity, int cost) {

    edge_list[u].push_back(Edge(v, capacity, cost, edge_list[v].size()));
    edge_list[v].push_back(Edge(u, 0, -cost, edge_list[u].size() - 1));
}

int augment(int v, vi &parent, vi &prev_edge, int minEdge) {

    if (parent[v] == -1) {
        return minEdge;
    } else {

        int u = parent[v];
        Edge &edge = edge_list[u][prev_edge[v]];

        int curr_flow = augment(u, parent, prev_edge, min(minEdge, edge.capacity));

        edge.capacity -= curr_flow;
        edge_list[v][edge.cancel_edge].capacity += curr_flow;

        return curr_flow;

    }
}

int max_flow(int source, int target) {

    int max_flow = 0;

    int source_flow = 0;
    for (auto &edge : edge_list[source]) {
        source_flow += edge.capacity;
    }

    while (true) {

        vi dist(V, INF);
        dist[source] = 0;
        vi parent(V, -1);
        vi prev_edge(V, -1);
        vi in_queue(V, 0);

        queue<int> q;
        q.push(source);
        in_queue[source] = 1;

        //SPFA
        while (!q.empty()) {

            int u = q.front();
            q.pop();
            in_queue[u] = 0;

            for (int e = 0; e < edge_list[u].size(); ++e)
 {

                auto &edge  = edge_list[u][e];
                int v = edge.dest;

                if (edge.capacity > 0 && dist[u] + edge.cost < dist[v]) {

                    dist[v] = dist[u] + edge.cost; // relax
                    parent[v] = u;
                    prev_edge[v] = e;

                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = 1;
                    }

                }
            }
        }


        if (dist[target] != INF) {
            max_flow += augment(target, parent, prev_edge, INF);
        } else {
            break;
```

```
        }
    }

    return max_flow;

}
```

1.15 Euler tour

```
/*
 * O(V + E)
 * Setar vet_adj, escolher u inicial como vertice de grau
     impar.
 *
 * Eh importante checar se o grafo eh conexo e se comtem
    0 ou 2 vertices de grau impar.
 *
 */

bool checkParity()
bool checkConnectivity()

void euler_tour (int u, list<int>::iterator it) {

    while (!vet_adj[u].empty()) {
        int v = *vet_adj[u].begin();
        vet_adj[u].erase(find(all(vet_adj[u]), v));
        vet_adj[v].erase(find(all(vet_adj[v]), u));
        euler_tour(v, tour_list.insert(it, v));
    }

}
```

2 Trees

2.1 Diameter

```
/*
 * O (V + E)
 * Setar V e vet_adj
*/

#define INF 1000000000
int V;
vector<vi> vet_adj;

pii bfs_max_dist_and_index(int source) {

    vi dist(V, INF);
    stack<int> stack;

    stack.push(source);
    dist[source] = 0;

  int max_dist = 0;
    int max_dist_index = 0;

    while (!stack.empty()) {
        int u = stack.top();
        stack.pop();

        for (auto &v : vet_adj[u]) {
            if (dist[v] == INF) {
                dist[v] = dist[u] + 1;
                stack.push(v);

                if (dist[v] > max_dist) {
                    max_dist = dist[v];
                    max_dist_index = v;
                }
            }
        }
    }

  return mp(max_dist, max_dist_index);
}

int get_diameter() {

    int max_dist_index = bfs_max_dist_and_index(0).second
    ;
  return bfs_max_dist_and_index(max_dis_index).first;

}
```

2.2 Create from degree array

```
/*
  O (V)

  Parametros:
    * vetor de graus
    * vet_adj que sera a saida da funcao
    * curr_idx = 0 [indice no vetor de degree]
    * diff_one_pos = posicao do primeiro numero diferente
     de 1

  A ideia aqui eh que qualquer vetor de grau que contenha
     N elementos e tenha
  uma soma total de 2n - 2 (Total de graus de uma arvore
    de tamanho N), pode
  ter uma arvore construida respeitando seus graus. Isso
    pode ser provado facil-
  mente por inducao matematica no numero de vÃľrtices. Da
     prova deriva o algoritmo.
*/
```

```
void gen_tree(vii &degree, vector<vi> &vet_adj, int
    curr_idx, int diff_one_pos) {

    if (curr_idx == vet_adj.size() - 2) {
        // Base: two vertices, always possible
        vet_adj[degree[curr_idx].second].push_back(degree
    [curr_idx + 1].second);
        return;
    }

    vet_adj[degree[curr_idx].second].push_back(degree[
    diff_one_pos].second);

    --degree[diff_one_pos].first;
    if (degree[diff_one_pos].first == 1) {
        ++diff_one_pos;
    }

    gen_tree(degree, vet_adj, curr_idx + 1, diff_one_pos)
    ;

}
```

3 DP

3.1 LCS

```
/* Retorna apenas o tamanho da LCS */
int lcs(string &str1, string &str2) {

  int sz1 = str1.size();
  int sz2 = str2.size();

  int dp[sz1+1][sz2+1];
  memset(dp, 0, sizeof(dp));

  for (int i = 1; i<=m; i++) {
    for (int j = 1; j<=n; j++) {

      if (X[i - 1] == Y[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
      }

    }
  }

   return dp[sz1][sz2];
}
```

3.2 LIS - NlgN

```
void get_LIS(vi &values, vi &res) {

    int sz = values.size();
    res.resize(sz);

    vi ends_list(sz);
    int max_size = 0;
    for (size_t i = 0; i < sz; ++i) {
        int pos = distance(ends_list.begin(), lower_bound
    (ends_list.begin(), ends_list.begin() + max_size,
    values[i]));

        ends_list[pos] = values[i];

  if (pos == max_size)
            max_size = pos + 1;

  res[i] = max_size;
    }

}
```

3.3 MAX 2D

```
        vector<vi> acu_mat(N, vi(N));

        int num;
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                cin >> num;
                if(i == 0 && j == 0){
                    acu_mat[i][j] = num;
                }else if(i == 0){
                    acu_mat[i][j] = num + acu_mat[i][j
    -1];
                }else if(j == 0){
                    acu_mat[i][j] = num + acu_mat[i-1][j
    ];
                }else{
                    acu_mat[i][j] = num + acu_mat[i][j-1]
     + acu_mat[i-1][j] - acu_mat[i-1][j-1];
                }
            }
```

```
        }

        int maxi = numeric_limits<int>::min();

        for (int i_0 = 0; i_0 < N; ++i_0) {
            for (int j_0 = 0; j_0 < N; ++j_0) {
                for (int i_f = i_0; i_f < N; ++i_f) {
                    for (int j_f = j_0; j_f < N; ++j_f) {

                        int curr_val = acu_mat[i_f][j_f];

                        if(j_0 > 0) curr_val -= acu_mat[
    i_f][j_0-1];
                        if(i_0 > 0) curr_val -= acu_mat[
    i_0-1][j_f];
                        if(j_0 > 0 && i_0 > 0) curr_val
    += acu_mat[i_0-1][j_0-1];

                        maxi = max(maxi, curr_val);
                    }
                }
            }
        }

        cout << maxi << endl;

    }
```

3.4 Count change

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
  value[x] = INF;
  for (auto c : coins) {
    if (x-c >= 0 && value[x-c]+1 < value[x]) {
      value[x] = value[x-c]+1;
      first[x] = c;
    }
  }
}
```

4 Data Structures

4.1 Max SegTree (Simple)

```
#define LEFT_NODE(i) i+i+1
#define RIGHT_NODE(i) i+i+2
#define MID(seg) (seg.first+seg.second)/2
#define LEFT_SEG(seg) mp(seg.first, MID(seg))
#define RIGHT_SEG(seg) mp(MID(seg) + 1, seg.second)

class SegTree{

private:

    vi max_tree;
    int sz;

    void build(int *vet, pii curr_seg, int node){

        if(curr_seg.first == curr_seg.second){
            max_tree[node] = vet[curr_seg.first];
            return;
        }

        build(vet, LEFT_SEG(curr_seg), LEFT_NODE(node));
        build(vet, RIGHT_SEG(curr_seg), RIGHT_NODE(node))
    ;

        max_tree[node] = max(max_tree[LEFT_NODE(node)],
    max_tree[RIGHT_NODE(node)]);
    }

    void update_aux(pii curr_seg, int index, int value,
    int node){

        if (curr_seg.first == curr_seg.second) {
            max_tree[node] = value;
            return;
        }

        if (index <= MID(curr_seg)) {
            update_aux(LEFT_SEG(curr_seg), index, value,
    LEFT_NODE(node));
        } else {
            update_aux(RIGHT_SEG(curr_seg), index, value,
     RIGHT_NODE(node));
        }

        max_tree[node] = max(max_tree[LEFT_NODE(node)],
    max_tree[RIGHT_NODE(node)]);
    }

    int query_aux(pii curr_seg, pii target_seg, int node)
    {

        if(curr_seg.second < target_seg.first || curr_seg
    .first > target_seg.second){
            return 0;
        }

        if(curr_seg.first >= target_seg.first && curr_seg
    .second <= target_seg.second){
            return max_tree[node];
        }

        return max(query_aux(LEFT_SEG(curr_seg),
    target_seg, LEFT_NODE(node))
                   , query_aux(RIGHT_SEG(curr_seg),
    target_seg, RIGHT_NODE(node)));
    }

public:

    SegTree(int *vet, int size){
```

```
        sz = size;
        max_tree.resize(4 * sz);
        build(vet, mp(0, sz - 1), 0);
    }

    int query(pii target_seg){
        return query_aux(mp(0, sz - 1), target_seg, 0);
    }

    void update(int index, int value) {
        update_aux(mp(0, sz - 1), index, value, 0);
    }
};
```

4.2 Sum SegTree (Lazy)

```
#define LEFT_NODE(node) 2*node+1
#define RIGHT_NODE(node) 2*node+2
#define MID(seg) (seg.first+seg.second)/2
#define LEFT_SEG(seg) make_pair(seg.first, MID(seg))
#define RIGHT_SEG(seg) make_pair(MID(seg) + 1, seg.second
    )

class SegTree{
private:

    vector<long long int> tree;
    vector<long long int> lazy_tree;
    int sz;

    void lazy_update(pii curr_seg, int node){

        if(lazy_tree[node] != 0){

            int seg_sz = curr_seg.second - curr_seg.first
     + 1;
            long long int value = lazy_tree[node];

            tree[node] += seg_sz*value;

            if(curr_seg.first != curr_seg.second){
                /* Propagate, not a leaf */
                lazy_tree[LEFT_NODE(node)] += value;
                lazy_tree[RIGHT_NODE(node)] += value;
            }

            lazy_tree[node] = 0;
        }
    }


    void update_aux(pii curr_seg, pii target_seg, long
    long int value, int node){

        lazy_update(curr_seg, node);

        if(curr_seg.second < target_seg.first || curr_seg
    .first > target_seg.second){
            /* Disjoint */
            return;
        }

        if(curr_seg.first >= target_seg.first && curr_seg
    .second <= target_seg.second){
            /* Within */
            lazy_tree[node] = value;
            lazy_update(curr_seg, node);
            return;
        }

        /*Overlap*/
        update_aux(LEFT_SEG(curr_seg), target_seg, value,
     LEFT_NODE(node));
        update_aux(RIGHT_SEG(curr_seg), target_seg, value
    , RIGHT_NODE(node));

        tree[node] = tree[LEFT_NODE(node)] + tree[
    RIGHT_NODE(node)];

    }

    long long int value_aux(pii curr_seg, pii target_seg,
     int node){

        if(curr_seg.second < target_seg.first || curr_seg
    .first > target_seg.second){
            /* Disjoint */
            return 0;
        }

        lazy_update(curr_seg, node);

        if(curr_seg.first >= target_seg.first && curr_seg
    .second <= target_seg.second){
            /* Within */
            return tree[node];
        }

        return value_aux(LEFT_SEG(curr_seg), target_seg,
    LEFT_NODE(node))
                + value_aux(RIGHT_SEG(curr_seg),
    target_seg, RIGHT_NODE(node));

    }

public:

    SegTree(int size){
        sz = size;
        tree.resize(4*sz, 0);
        lazy_tree.resize(4*sz, 0);
    }

    void update(pii target_seg, long long int value){
        update_aux(make_pair(0, sz-1), target_seg, value,
     0);
    }

    long long int value(pii target_seg){
        return value_aux(make_pair(0, sz - 1), target_seg
    , 0);
    }
};
```

4.3 ETT in SegTree

```
vector<vi> vet_adj;

vector<pii> dict_id_to_range;
int dfs_curr_time;
vi tree_on_vector;
vi original_tree;

void dfs(int node) {

    tree_on_vector.push_back(original_tree[node]);
    ++dfs_curr_time;
    dict_id_to_range[node].first = dfs_curr_time;
    for (auto &v : vet_adj[node]) {
        dfs(v);
    }
    dict_id_to_range[node].second = dfs_curr_time;

}

void create_dict_id_to_range() {

    dfs_curr_time = -1;
```

```
    dfs(0);
}


int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);


    int sz, query_count;
    cin >> sz >> query_count;

    original_tree.clear();
    original_tree.resize(sz);

    vet_adj.clear();
    vet_adj.resize(sz);

    tree_on_vector.clear();

    dict_id_to_range.clear();
    dict_id_to_range.resize(sz);

    /*
        Read/ Create original tree
    */

    create_dict_id_to_range();


    /*
        Create SegTree using tree_on_vector vector
    */

    /*
        Read/process Querys
    */
    // USAGE:
    sgtree.update(dict_id_to_range[id]);
    sgtree.query(dict_id_to_range[id])



    return 0;
}
```

4.4 Trie (Complete)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bits/stdc++.h"

using namespace std;

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')


struct TrieNode {
    bool end_word;
    TrieNode *children[ALPHABET_SIZE];

    TrieNode() {
        this->end_word = false;

        for(int i = 0; i < ALPHABET_SIZE; i++) {
            this->children[i] = nullptr;
        }
    }

    bool empty() {

        for(int i = 0; i < ALPHABET_SIZE; i++) {
            if (this->children[i]) {
                return false;
            }
        }

        return true;
    }

};

class Trie{
    TrieNode *root;
    int count;

private:

    bool delete_aux(TrieNode *curr_node, const string &
    key, int level) {

        // Base case
        if (level == key.size()) {

            if (curr_node->end_word) { // Verifica se
    achou a chave

                curr_node->end_word = false;

                // Se o nÃş Ãľ vazio, podemos/devemos
    apaga-lo
                if (curr_node->empty()) {
                    return true;
                }

                return false;
            }
        } else {
            int index = INDEX(key[level]);

            if (delete_aux(curr_node->children[index],
    key, level + 1)) {

                // Se o proximo nÃş foi marcado para ser
    apagado, nos apagamos
                delete curr_node->children[index];
                curr_node->children[index] = nullptr;

                // Verificamos se o no atual deve ser
    apagado e propagamos
                return !curr_node->end_word && curr_node
    ->empty();
            }
        }

    return false;
}

public:
    Trie() {
        this->root = new TrieNode;
        this->count = 0;
    }

    void insert(const string &key) {

        int length = key.size();
        TrieNode *curr_node;

        this->count++;
        curr_node = this->root;

        for(int level = 0; level < length; level++) {
            int index = INDEX(key[level]);

            if (curr_node->children[index]) {
                // Ja existe no para esse prefixo
```

```
            curr_node = curr_node->children[index];
        } else {
            // Criamos um no novo para esse prefixo e
 continuamos nele
            curr_node->children[index] = new TrieNode
;
            curr_node = curr_node->children[index];
        }
    }

    // O ultimo no Ãľ marcado como fim de palavra
    curr_node->end_word = true;
}

bool search(const string &key) {

    int length = key.size();
    TrieNode *curr_node;

    curr_node = this->root;

    for(int level = 0; level < length; level++) {
        int index = INDEX(key[level]);

        if (curr_node->children[index]) {
            curr_node = curr_node->children[index];
        } else {
            return false;
        }
    }

    return curr_node->end_word;
}

void delete_key(const string &key) {
    if( key.size() > 0 ) {
        delete_aux(this->root, key, 0);
    }
}

};


int main()
{

    vector<string> keys = {"she", "sells", "sea", "shore"
, "the", "by", "sheer"};
    Trie trie;

    for (auto &s : keys) {
        trie.insert(s);
    }

    for (auto &s : keys) {
        printf("%s %s\n", s.c_str(), trie.search(s) ? "
    Present in trie" : "Not present in trie");
    }

    return 0;
}
```

5 String

5.1 KMP

```
/*
  Funcionamento do preprocess_lps:

Para melhor compreender o funcionamento veja a foto (pic1
    ) em anexo a essa pasta.
Basicamente os dois intervalos em azul dizem respeito ao
    maior prefixo&sufixo do subproblema
anterior, i.e, do vetor no intervalo [0.. i - 1]. Agora a
     pergunta que o algoritmo deve fazer
Ãľ: "SerÃą que eu posso expandir esses dois intervalos (
    em azul) para a direita adicionando o
novo elemento (*i)?" Para isso basta verificar se a
    direita de ambos intervalos coincidem, i.e., se pat[
    i] == pat[len].
  Caso pat[i] != pat[len], entÃčo nÃčo podemos
    simplesmente extender nosso prefixo&sufixo. Temos
    que recuar para o segundo maior prefixo&sufixo do
    nosso subproblema. Para isso uma abordagem ingenua
    seria simplesmente fazer len = len - 1 e recomecar o
     loop (pois assim diminuimos em 1 o tamanho do nosso
     prefixo&sufixo, descobrindo o segundo maior).
    Contudo, nem sempre existe um prefixo&sufixo que
    tenha exatamente 1 a menos de tamanho.
    Exemplo: ABAB
    Aqui vemos que AB Ãľ o prefixo&sufixo mas depois dele
     nÃčo temos nenhum com tamanho 1.
Por isso, utilizamos a linha len = lps[len - 1], pois se
    existir um sufixo de tamanho len - 1, entao lps[len
    - 1] = len - 1. Caso contrÃąrio, ele nos darÃą o
    maior prefixo&sufixo menor que esse tamanho.


*/

void preprocess_lps(const char *pat, int M, int *lps) {
    int len = 0;

    lps[0] = 0;

    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else  {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Printa os indices no txt onde o pat foi encontrado
void KMP(const char *pat, const char *txt) {
    int M = strlen(pat);
    int N = strlen(txt);

    int lps[M];
    preprocess_lps(pat, M, lps);

    int i = 0;
    int j  = 0;
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
```

```
        }

        if (j == M) {
            printf("%d\n", i-j);
            j = lps[j - 1];
        } else if (i < N && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}
```

6 Misc

6.1 MOS - Distinct elements

```
int blocks;

struct Query{
    int L, R;
    int order;
};

bool queryComparator(Query &q1, Query &q2) {

    if (q1.L / blocks == q2.L / blocks) {
        return q1.R < q2.R;
    }

    return q1.L / blocks < q2.L / blocks;
}

void queryResults(int a[], int n, Query q[], int m, vi &
    results) {

    blocks = sqrt(n);

    sort(q, q + m, queryComparator);

    int currL = 0, currR = 0;
    vi freq(1000001, 0);

    int distinct_count = 1;
    freq[a[0]] = 1;

    results.resize(m);

    for (size_t i = 0; i < m; i++) {
        int L = q[i].L, R = q[i].R;


        while (currL > L) {
            if (freq[a[currL - 1]] == 0) {
                distinct_count++;
            }
            ++freq[a[currL - 1]];
            --currL;

        }

        while (currR < R) {
            if (freq[a[currR + 1]] == 0) {
                distinct_count++;
            }
            ++freq[a[currR + 1]];
            ++currR;
        }

        while (currR > R) {
            --freq[a[currR]];
            if (freq[a[currR]] == 0) {
                --distinct_count;
            }
            --currR;
        }

        while (currL < L) {
            --freq[a[currL]];
            if (freq[a[currL]] == 0) {
                --distinct_count;
            }
            ++currL;
        }

        results[q[i].order] = distinct_count;
    }
```

```
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    int n, m;

    cin >> n;
    int a[n];
    for (size_t i = 0; i < n; i++) {
        cin >> a[i];
    }

    cin >> m;
    Query q[m];
    for (size_t i = 0; i < m; i++) {
        Query new_query;
        cin >> new_query.L >> new_query.R;
        --new_query.L, --new_query.R;
        new_query.order = i;
        q[i] = new_query;
    }

    vi results;
    queryResults(a, n, q, m, results);

    for (auto &res : results) {
        printf("%d\n", res);
    }

    return 0;
}
```

6.2 Primes generator

```
vi primes;

bool is_prime(int num) {

    if (num % 2 == 0) { return false; }

    int sqr = sqrt(num);
    for (int i = 3; i <= sqr; ++i) {
        if (num % i == 0) {
            return false;
        }
    }

    return true;
}

void calc_primes() {

    primes.push_back(2);
    for (int i = 3; i < 1000; ++i) {
        if (is_prime(i)) {
            primes.push_back(i);
        }
    }
}
```

6.3 Inversions vector nlgn

```
/*
  Aqui eu calculo o numero de inversoes durante o merge
    sort. Essa funcao realmente
  ordena o vetor, entao tome cuidado para protege-lo.
  ATENCAO: Essa funcao apenas foi testada para numeros
    distintos, cheque melhor caso
  nao seja o caso.

  Complexidade O(nlgn)
*/

int count_inversions(vi &vet, int begin, int end) {

    if (begin == end) {
        return 0;
    }

    int mid = (begin+end)/2;
    int left_inv  = count_inversions(vet, begin, mid);
    int right_inv = count_inversions(vet, mid + 1, end);

    vi aux_vet;
    int total_inv = left_inv + right_inv;

    int left_pos = begin;
    int right_pos = mid + 1;

    while (left_pos <= (begin+end)/2 && right_pos <= end)
     {
        if (vet[right_pos] < vet[left_pos]) {
            total_inv += mid - begin + 1 - (left_pos -
    begin);
            aux_vet.push_back(vet[right_pos]);
            ++right_pos;
        } else {
            aux_vet.push_back(vet[left_pos]);
            ++left_pos;
        }
    }

    while (left_pos <= (begin+end)/2) {
        aux_vet.push_back(vet[left_pos]);
        ++left_pos;
    }

    while (right_pos <= end) {
        aux_vet.push_back(vet[right_pos]);
        ++right_pos;
    }

    for (size_t i = begin; i <= end; i++) {
        vet[i] = aux_vet[i - begin];
    }

    return total_inv;
}
```

7 Theorems

Max Independent Set (MIS) = N - MCBM

8 Template

```
bool debug = true;

//<editor-fold desc="GUAXINIM TEMPLATE">
/********   All Required Header Files ********/
#include "bits/stdc++.h"
using namespace std;

#define all(container) container.begin(), container.end()
#define mp(i,j) make_pair(i,j)
#define space " "
#define pb push_back

typedef pair<int,int> pii;
typedef long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;
typedef vector<pii> vii;


/// Debug Start
template<class T1> void deb(T1 e1)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << endl;
    }
}
template<class T1,class T2> void deb(T1 e1, T2 e2)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << space << e2 << endl;
    }
}
template<class T1,class T2,class T3> void deb(T1 e1, T2
    e2, T3 e3)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << space << e2 << space << e3 << endl;
    }
}
template<class T1,class T2,class T3,class T4> void deb(T1
     e1, T2 e2, T3 e3, T4 e4)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << space << e2 << space << e3 << space
     << e4 << endl;
    }
}
template<class T1,class T2,class T3,class T4,class T5>
    void deb(T1 e1, T2 e2, T3 e3, T4 e4, T5 e5)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << space << e2 << space << e3 << space
     << e4 << space << e5 << endl;
    }
}
template<class T1,class T2,class T3,class T4,class T5,
    class T6> void deb(T1 e1, T2 e2, T3 e3, T4 e4 ,T5 e5
    , T6 e6)
{
    if(debug) {
        cout << "[DEBUG]";
        cout << e1 << space << e2 << space << e3 << space
     << e4 << space << e5 << space << e6 << endl;
    }
}


template<typename T>
```

```
void print_vector_debug(const T& t) {

    if(debug) {
        cout << "[DEBUG] VECTOR:";
        for (auto i = t.cbegin(); i != t.cend(); ++i) {
            if ((i + 1) != t.cend()) {
                cout << *i << " ";
            } else {
                cout << *i << endl;
            }
        }
    }
}

template<typename T>
void print_array_debug(const T arr, int size){

    if(debug) {
        cout << "[DEBUG] VECTOR:";
        for (int i = 0; i < size; ++i) {
            cout << arr[i] << space;
        }
        cout << endl;
    }
}

template<typename T>
void print_2Darray_debug(const T arr, int rows, int cols)
    {

    if(debug) {
        cout << "[DEBUG] Matrix:" << endl;
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                cout << arr[i][j] << space;
            }
            cout << endl;
        }
        cout << endl;
    }
}

template<typename T>
void print_matrix_debug(const T& t) {
    if(debug) {
        cout << "[DEBUG] MATRIX:" << endl;
        for (auto i = t.cbegin(); i != t.cend(); ++i) {
            for(auto j = i->cbegin(); j != i->cend(); ++j
    ){
                cout << *j << " ";
            }
            cout << endl;
        }
    }
}
//</editor-fold>


int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(0);


    return 0;
}
```