🖥️ Resolução de Problemas

Grafos (Conceitos, Representações, DFS, BFS, Dijkstra)

Exercício 1 – Love Triangle

Resolução em Python:

```python
n = int(input())

# Lê a lista dos aviões que cada avião gosta
f = list(map(int, input().split()))

# Como o Python indexa a partir de 0, subtraímos 1 de cada elemento
# Ex: se o avião 1 gosta do avião 2, f[0] será 1
f = [x - 1 for x in f]

# Flag que indica se encontramos um triângulo amoroso
achou = False

# Percorre todos os aviões para verificar se existe um ciclo de 3 elementos
for i in range(n):
    a = f[i]      # Avião A gosta do avião B
    b = f[a]      # Avião B gosta do avião C
    c = f[b]      # Avião C gosta do avião D

    if c == i:    # Se avião D for o avião A, então temos um triângulo A → B → C → A
        achou = True
        break     # Já encontramos um triângulo, não precisa continuar

print("YES" if achou else "NO")
```

Exercício 2 – Party

Resolução em Python:

```python
import sys
sys.setrecursionlimit(10**6) # Evita erro de recursão profunda

n = int(input())
managers = []
for _ in range(n):
managers.append(int(input()))

# Cria o grafo de relações (gerente -> subordinado)
graph = [[] for _ in range(n)]
```

```
for i in range(n):
if managers[i] != -1:
graph[managers[i] - 1].append(i)

# DFS para encontrar a profundidade da árvore
def dfs(v):
max_depth = 1 # Começa com 1 (o próprio funcionário)
for neighbor in graph[v]:
max_depth = max(max_depth, 1 + dfs(neighbor))
return max_depth

# Encontrar todas as "raízes" (funcionários sem gerente)
max_chain = 0
for i in range(n):
if managers[i] == -1:
max_chain = max(max_chain, dfs(i))

print(max_chain)
```

Exercício 3 - Li Hua and Maze

Resolução em Python:

```
def contar_direcoes(x, y, n, m):
direcoes = 0
if x > 1:
direcoes += 1
if x < n:
direcoes += 1
if y > 1:
direcoes += 1
if y < m:
direcoes += 1
return direcoes



# Leitura de múltiplos casos de teste
t = int(input())
for _ in range(t):
n, m = map(int, input().split())
x1, y1, x2, y2 = map(int, input().split())

mov1 = contar_direcoes(x1, y1, n, m)
mov2 = contar_direcoes(x2, y2, n, m)

# A menor quantidade de direções define o número mínimo de obstáculos necessários
print(min(mov1, mov2))
```

Exercício 4 – Two Buttons

Resolução em Python:

```
from collections import deque

def bfs_min_clicks(n, m):
# Como os números podem dobrar, o limite máximo para visitar é 2 * max(n, m)
max_limit = 2 * max(n, m) + 2

# visited[i] = True se o número i já foi visitado
visited = [False] * max_limit

# dist[i] armazena o número de cliques necessários para chegar em i a partir de n
dist = [0] * max_limit

# Inicialização da fila da BFS com o número inicial n
queue = deque()
queue.append(n)
visited[n] = True

# Início da BFS
while queue:
curr = queue.popleft()

# Se chegar no número desejado, retorna a distância (número de cliques)
if curr == m:
return dist[curr]

# Geramos os dois possíveis estados seguintes
next_steps = []

# Botão vermelho: multiplica por 2
if curr * 2 < max_limit:
next_steps.append(curr * 2)

# Botão azul: subtrai 1 (se ainda for positivo)
if curr - 1 > 0:
next_steps.append(curr - 1)

# Para cada próximo número, se ainda não foi visitado, adicionamos à fila
for next_node in next_steps:
if not visited[next_node]:
visited[next_node] = True
dist[next_node] = dist[curr] + 1 # Um clique a mais
queue.append(next_node)
```

Exercício 5 - Cthulhu

Resolução em Python:

```python
n, m = map(int, input().split())  # Lê o número de vértices (n) e arestas (m)

# Cria a lista de adjacência para representar o grafo
adj = [[] for _ in range(n + 1)]

# Lê as m arestas e monta o grafo (não direcionado)
for _ in range(m):
    u, v = map(int, input().split())
    adj[u].append(v)
    adj[v].append(u)

# Vetor para marcar os vértices visitados durante a DFS
visited = [False] * (n + 1)

# Função de DFS para visitar todos os vértices conectados a 'u'
def dfs(u):
    visited[u] = True
    for v in adj[u]:
        if not visited[v]:
            dfs(v)

dfs(1)  # Começa a DFS a partir do vértice 1

# Verifica se o grafo é conectado e tem exatamente n arestas (formando um ciclo
simples)
if all(visited[1:]) and m == n:
    print("FHTAGN!")
else:
    print("NO")
```

Exercício 6 – Rook, Bishop and King

Resolução em Python:

```python
def min_moves(r1, c1, r2, c2):
    # Cálculo dos movimentos da torre (rook)
    if r1 == r2 and c1 == c2:
        rook_moves = 0  # Mesma posição
    elif r1 == r2 or c1 == c2:
        rook_moves = 1  # Mesma linha ou coluna
    else:
        rook_moves = 2  # Caso contrário, dois movimentos (linha + coluna)

    # Cálculo dos movimentos do bispo (bishop)
    if r1 == r2 and c1 == c2:
        bishop_moves = 0  # Mesma posição
```

```
    elif (r1 + c1) % 2 != (r2 + c2) % 2:
        bishop_moves = 0  # Casas de cores diferentes: impossível mover
    elif abs(r1 - r2) == abs(c1 - c2):
        bishop_moves = 1  # Mesma diagonal
    else:
        bishop_moves = 2  # Duas diagonais diferentes: precisa de dois lances

    # Cálculo dos movimentos do rei (king)
    if r1 == r2 and c1 == c2:
        king_moves = 0
    else:
        king_moves = max(abs(r1 - r2), abs(c1 - c2))  # Move um por vez, maior
distância define

    return rook_moves, bishop_moves, king_moves


# Entrada das posições inicial e final
r1, c1, r2, c2 = map(int, input().split())

# Chamada da função e exibição dos resultados
rook, bishop, king = min_moves(r1, c1, r2, c2)
print(f"{rook} {bishop} {king}")
```

Exercício 7 – Minimum spanning tree for each edge

Resolução em C++:

```
#include<bits/stdc++.h>
using namespace std;

// Macros para facilitar leitura e escrita
#define si(a) scanf("%d",&a)
#define sii(a,b) scanf("%d %d",&a,&b);
#define siii(a,b,c) scanf("%d %d %d",&a,&b,&c);

#define sl(a) scanf("%lld",&a)
#define sll(a,b) scanf("%lld %lld",&a,&b);
#define slll(a,b,c) scanf("%lld %lld %lld",&a,&b,&c);

#define outi(a) printf("%d\n",a)
#define outii(a,b) printf("%d %d\n",a,b)
#define outis(a) printf(" %d",a)

#define outl(a) printf("%lld\n",a)
#define outll(a,b) printf("%lld %lld\n",a,b)
#define outls(a) printf(" %lld",a)
```

```cpp
#define cel(n,k) ((n-1)/k+1)
#define sets(a) memset(a, -1, sizeof(a))
#define clr(a) memset(a, 0, sizeof(a))
#define fr(n) for(int i=0;i<n;i++)
#define fr1(n) for(int i=1;i<=n;i++)
#define frj(n) for(int j=0;j<n;j++)
#define frj1(n) for(int j=1;j<=n;j++)
#define pb push_back
#define all(v) v.begin(),v.end()
#define mp make_pair
#define ff first
#define ss second
#define INF 10000007
#define fastIO() ios_base::sync_with_stdio(false); cin.tie(NULL);

typedef long long i64;
typedef unsigned long long ull;
typedef pair<int,int> pii;
typedef pair<long long,long long> pll;

const int maxn= 2e5+5; // limite máximo de vértices

// Estrutura de uma aresta
struct edge {
    int u, v;
    i64 w;
    bool operator<(const edge& p) const {
        return w < p.w; // ordenação por peso (para Kruskal)
    }
};

// Variáveis globais
int vis[maxn],n,m,k, par[maxn], T[maxn], P[maxn][22], L[maxn], Dist[maxn];
i64 Maxi[maxn][22], Mini[maxn][22]; // tabelas para RMQ em LCA

vector<int> v[maxn];         // Lista de adjacência da MST
vector<edge> edges, eds;     // edges = todas as arestas; eds = backup
map<pii,i64> weight;         // armazena pesos das arestas da MST

// Limpa estruturas para múltiplos casos de teste
void reset(){
    fr1(n)  v[i].clear();
    edges.clear();
    clr(par);
}

// Union-Find com path compression
int finds(int r) {
```

```
    if (par[r]==r) return r;
    return par[r]= finds(par[r]);
}

// Algoritmo de Kruskal para construir a MST
i64 mst(int n) {
    sort(edges.begin(), edges.end()); // ordena por peso
    for (int i = 1; i <= n; i++) par[i] = i; // inicializa UF

    i64 count = 0, s = 0;
    for(edge e: edges){
        int x = finds(e.u);
        int y = finds(e.v);
        int w = e.w;

        if (x != y) {
            // adiciona a aresta na árvore
            v[e.u].pb(e.v);
            v[e.v].pb(e.u);

            // salva o peso da aresta nos dois sentidos
            weight[mp(e.u,e.v)] = w;
            weight[mp(e.v,e.u)] = w;

            par[x] = y; // união
            count++;
            s += e.w;
            if (count == n - 1)
                break;
        }
    }

    return s; // peso total da MST
}

// DFS para preencher pai (T), profundidade (L)
void dfs(int u, int p){
    T[u]= p;
    L[u]= (p > -1) ? L[p] + 1 : 0;

    for(int s: v[u]){
        if(s == T[u]) continue;
        dfs(s, u);
    }
}

// Inicializa estruturas para LCA com Binary Lifting
void init_LCA(int n){
```

```
    sets(P); // inicializa ancestrais com -1

    fr1(n){
        P[i][0]= T[i]; // pai direto

        if(P[i][0]!=-1) {
            Maxi[i][0] = weight[mp(i,T[i])]; // peso da aresta pai
            Mini[i][0] = weight[mp(i,T[i])];
        }
    }

    for(int j=1; (1<<j)<n; j++){
        fr1(n)
            if(P[i][j-1]!=-1) {
                P[i][j] = P[P[i][j - 1]][j - 1]; // ancestral de 2^j
                Maxi[i][j] = max(Maxi[i][j-1], Maxi[P[i][j-1]][j-1]);
                Mini[i][j] = min(Mini[i][j-1], Mini[P[i][j-1]][j-1]);
            }
    }
}

// Responde a consulta: maior aresta no caminho entre p e q na MST
i64 lca_query(int p, int q) {
    i64 tmp, log, i, maxi = 0, mini = 1e6 + 6;

    // garante que p esteja mais profundo
    if (L[p] < L[q])
        tmp = p, p = q, q = tmp;

    // calcula o maior log válido
    log = 1;
    while ((1 << (log + 1)) <= L[p]) log++;

    // sobe p até igualar a profundidade de q
    for (i = log; i >= 0; i--)
        if (L[p] - (1 << i) >= L[q]) {
            maxi = max(maxi, Maxi[p][i]);
            mini = min(mini, Mini[p][i]);
            p = P[p][i];
        }

    if (p == q) return maxi; // já são o mesmo nó

    // sobe os dois juntos até encontrar o LCA
    for (i = log; i >= 0; i--)
        if (P[p][i] != -1 && P[p][i] != P[q][i]) {
            maxi = max({maxi, Maxi[p][i], Maxi[q][i]});
            mini = min({mini, Mini[p][i], Mini[q][i]});
```

```
            p = P[p][i], q = P[q][i];
        }

    // checa último passo (pai de p e q)
    maxi = max({maxi, Maxi[p][0], Maxi[q][0]});
    mini = min({mini, Mini[p][0], Mini[q][0]});

    return maxi;
}

// Lê a entrada do grafo
void input(){
    sii(n,m);
    fr(m) {
        int x, y;
        i64 w;
        sii(x,y);  sl(w);
        edge e;
        e.u = x;
        e.v = y;
        e.w = w;
        edges.pb(e);
        eds.pb(e); // backup para pós-processamento
    }
}

// Função principal
int main() {
    reset();       // limpa estruturas
    input();       // lê dados

    i64 wet = mst(n); // cria a MST e armazena o peso total

    L[1] = 0;
    dfs(1,-1);        // monta a árvore e profundidades
    init_LCA(n);      // inicializa estrutura para LCA

    // Para cada aresta original, tenta substituí-la na MST
    for(edge e: eds) {
        int a = e.u, b = e.v;

        i64 ws = lca_query(a,b);        // maior peso no caminho a-b na MST
        i64 ans = wet - ws + e.w;       // nova MST simulada com essa aresta

        outl(ans); // imprime o peso da nova MST simulada
    }
}
```