Sale con dp y fritas

Contents

1 Template

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

typedef long long ll;
typedef long double ld;

const ll UNDEFINED = -1;
const int MAX_N = 1e5 + 1;
const int MOD = 1e9 + 7;
const int INF = 1e9;
const ll LINF = 1e18;
const ll zero = 0;
const ld EPSILON = 1e-10;

#define pb push_back
#define fst first
#define snd second
#define esta(x,c) ((c).find(x) != (c).end())  // Devuelve true si x es un elemento de c.
#define all(c) (c).begin(),(c).end()

#define DBG(x) cerr << #x << " = " << (x) << endl
#define RAYA cerr << "----------" << endl

#define forn(i,n) for (int i=0;i<(int)(n);i++)
#define forsn(i,s,n) for (int i=(s);i<(int)(n);i++)
#define dforn(i,n) for(int i=(int)((n)-1);i>=0;i--)
#define dforsn(i,s,n) for(int i=(int)((n)-1);i>=(int)(s);i--)

// Show vector
template <typename T>
ostream & operator <<(ostream &os, const vector<T> &v) {
    os << "[";
    forn(i, v.size()) {
        if (i > 0) os << ",";
        os << v[i];
    }
    return os << "]";
}

```

```cpp
// Show pair
template <typename T1, typename T2>
ostream & operator <<(ostream &os, const pair<T1, T2> &p) {
    os << "{" << p.first << "," << p.second << "}";
    return os;
}

// Show set
template <typename T>
ostream & operator <<(ostream &os, const set<T> &s) {
    os << "{";
    for(auto it = s.begin(); it != s.end(); it++){
        if(it != s.begin()) os << ",";
        os << *it;
    }
    return os << "}";
}

// ################################################################ //

int main()
{

    return 0;
}
```

2 Data structures

2.1 Disjoint Set

```cpp
struct DisjointSet{
    vector<ll> parent, rnk;
    ll numOfComponents;

    DisjointSet(ll n){
        rnk.assign(n, 0);
        for (ll i = 0; i < n; i++)
            parent.push_back(i);
        numOfComponents = n;
    }

    ll findSet(ll x){
        if(parent[x]==x) return x;
        parent[x] = findSet(parent[x]);
        return parent[x];
    }

    void unionSet(ll x, ll y){
        // Encontrar los representantes del conjunto.
        x = findSet(x);
        y = findSet(y);

        // Si los conjuntos son disjuntos:
        if (x != y){
            // Pongo al que tiene menos rango por debajo del de mayor rango.
            if (rnk[x] < rnk[y]){
                parent[x] = y;
            } else if (rnk[x] > rnk[y]){
                parent[y] = x;
            } else {  // Si tienen el mismo rango, incremento del rango. (rnk[x] == rnk[y])
                parent[y] = x;
                rnk[x]++;
            }
            numOfComponents--;
        }
    }

    bool same(ll x, ll y){
        return findSet(x) == findSet(y);
    }

};
```

2.2 MinHeap

```cpp
template <typename T>
using min_heap = priority_queue<T, vector<T>, greater<T>>;

// ejemplo:
// min_heap<pair<int, int>> q;
```

3 Grafos

3.1 BFS

```cpp
// BFS Normal
void bfs(ll inicio, vector<vector<ll>> &ady, vector<bool> &vis, vector<ll> &parent){
    queue<ll> q;
    q.push(inicio);
    vis[inicio] = true;
    while(!q.empty()){
        ll v = q.front();  // v es el vertice que estoy procesando
        q.pop();
        for (ll u : ady[v]){
            if (!vis[u]){
                vis[u] = true;
                parent[u] = v;
                q.push(u);
            }
        }
    }
}

// BFS que calcula los padres de los vertices
void calculatingParents(ll v, vector<vector<ll>> &adjList, vector<bool> &visited, vector<ll> &parents){
    ll n = adjList.size();
    visited[v] = true;
    parents[v] = v;

    queue<ll> q;
    q.push(v);

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

        for (int i = 0; i < adjList[u].size(); i++){
            ll w = adjList[u][i];
            if (!visited[w]){
                parents[w] = u;
                visited[w] = true;
                q.push(w);
            }
        }
    }
}


// BFS que chequea si un grafo es bipartito
bool isBipartite(ll v, vector<vector<ll>> &adjList, vector<ll> &teams){
    bool res = true;
    ll n = adjList.size();
    teams[v] = 1;

    queue<ll> q;
    q.push(v);

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

        for (int i = 0; i < adjList[u].size(); i++){
            ll w = adjList[u][i];
            if (teams[w] == 0){ // If the node isn't painted, paint it.
                if (teams[u] == 1){
                    teams[w] = 2;
                } else {
                    teams[w] = 1;
                }
                q.push(w);
            } else {
                if (teams[w] == teams[u]){ // If the node is painted, I check that its color is different from the v.
                    res = false;
                    break;
                }
            }
        }
    }

    return res;
}

// BFS que calcula el numero de componentes conexas
ll numberOfConnectedComponents(vector<vector<ll>> &adjList){
    ll n = adjList.size();
    vector<bool> visited(n, false);
    vector<ll> parents(n, UNDEFINED);
    ll res = 0;

```

```
    for (int i = 0; i < n; i++){
        if (!visited[i]){
            bfs(i, adjList, visited, parents);
        }
    }

    return res;
}

// BFS que calcula el numero de vertices en una componente conexa
ll numberOfVerticesInConnectedComponent(ll v, vector<vector<ll>> &
    adjList){
    ll n = adjList.size();
    vector<bool> visited(n, false);
    visited[v] = true;
    ll res = 1;

    queue<ll> q;
    q.push(v);

    while (!q.empty()){
        ll w = q.front();
        q.pop();

        for (int i = 0; i < adjList[w].size(); i++){
            ll u = adjList[w][i];

            if (!visited[u]){
                q.push(u);
                visited[u] = true;
                res++;
            }
        }
    }

    return res;
}
```

3.2 DFS

```
// DFS simple
void dfs(ll v, vector<vector<ll>> &ady, vector<bool> &vis){
    vis[v] = true;
    for(ll u : ady[v]){
        if (!vis[u]){
            dfs(u, ady, vis);
        }
    }
}

// DFS que me dice si existe un ciclo en un grafo (no dirigido) y en
    caso de existir, retorna la back-edge
pair<bool, edge> hasCycle(int v, vector<vector<int>> &adjList, vector<
    bool> &visited, vector<int> &parents){
    visited[v] = true;
    edge e = {UNDEFINED, UNDEFINED};
    pair<bool, edge> res = {false, e};

    for (int i = 0; i < adjList[v].size(); i++){
        int w = adjList[v][i];
        if (!visited[w]){
            parents[w] = v;
            res = hasCycle(w, adjList, visited, parents);
            if (res.first){
                break;
            }
        } else if (visited[w] && parents[v] != w && parents[v] !=
            UNDEFINED){
            edge backEdge = {w,v};
            return {true, backEdge};
        }
    }

    return res;
}
```

3.3 Bellman-Ford

```
using peso = ll;
using indice_nodo = ll;
using nodo_pesado = pair<peso, indice_nodo>;

struct Edge {
    ll a, b, cost;

    Edge(ll desde, ll hasta, ll c) : a(desde), b(hasta), cost(c) {}
```

```
};

// ############################################################## //

// Asume grafo representado como lista de aristas.

bool bellman_ford(ll n, indice_nodo inicio, vector<Edge> &edges, vector<
    peso> &dist){
    // Devuelve true sii existe un ciclo de longitud negativa.
    // Calcula SSSP en dist.

    // Obtiene las distancias mas cortas desde inicio hacia todos.
    dist[inicio] = 0;
    for(ll i=0; i<n; i++){
        for(Edge e : edges){
            if(-LINF < dist[e.a] && dist[e.a] < LINF){
                dist[e.b] = min(dist[e.b], dist[e.a] + e.cost);
            }
        }
    }

    // Detectar ciclo de longitud negativa.
    for(Edge e : edges){
        if(-LINF < dist[e.a] && dist[e.a] < LINF){
            if (dist[e.a] + e.cost < dist[e.b]){
                return true;
            }
        }
    }
    return false;
}
```

3.4 Dijkstra

```
using peso = ll;
using indice_nodo = ll;
using nodo_pesado = pair<peso, indice_nodo>;

// ############################################################## //

// Devuelve el vector de distancias desde inicio al i-esimo vertice.
vector<ll> dijkstra(const indice_nodo inicio, const vector<vector<
    nodo_pesado>> &ady){
    vector<ll> distancia(ady.size(), LINF);
    // vector<ll> parent(ady.size(), UNDEFINED);
    vector<bool> vis(ady.size(), false);

    distancia[inicio] = 0;
    set<nodo_pesado> q;

    q.insert({0, inicio});

    while(!q.empty()){
        ll v = q.begin() -> second;
        q.erase(q.begin());

        if (vis[v]) {continue;}
        vis[v] = true;
        for(const auto& [longitud, u] : ady[v]){
            // Longitud del camino de v hacia u.
            // Relax:
            if(distancia[v] + longitud < distancia[u]){
                q.erase({distancia[u], u});
                distancia[u] = distancia[v] + longitud;
                // parent[u] = v;
                q.insert({distancia[u], u});
            }
        }
    }
    return distancia;
}
```

3.5 Floyd-Warshall

```
// La matriz tuvo que ser inicializada como:
// d(s,v) = w(s,v) si existe
// d(s,v) = 0   si s = v
// d(s,v) = INF si no

void floyd_warshall (ll n, vector<vector<ll>> &matrizDist){
    // matrizDist en la entrada era la matriz de distancias diractas,
        luego del algoritmo queda calculada en ella APSP. (Distancias
        minimas i->j)
    for (ll k = 0; k < n; ++k) {
        for (ll i = 0; i < n; ++i) {
            for (ll j = 0; j < n; ++j) {
```

```
                if (matrizDist[i][k] < LINF && matrizDist[k][j] < LINF){
                    matrizDist[i][j] = min(matrizDist[i][j], matrizDist[
                        i][k] + matrizDist[k][j]);
                }
            }
        }
    }
}
```

3.6 Dantzig

```
void dantzig(ll n, vector<vector<peso>> &matrizDist){
    // matrizDist en la entrada era la matriz de distancias diractas,
        luego del algoritmo queda calculada en ella APSP.
    for(ll k=0; k<n; k++){

        for(ll i=0; i<k; i++){
            for(ll j=0; j<k; j++){
                matrizDist[i][k] = min(matrizDist[i][k], matrizDist[i][j
                    ] + matrizDist[j][k]);
                matrizDist[k][i] = min(matrizDist[k][i], matrizDist[k][j
                    ] + matrizDist[j][i]);
            }
        }

        for(ll i=0; i<k; i++){
            for(ll j=0; j<k; ++j){
                matrizDist[i][j] = min(matrizDist[i][j], matrizDist[i][k
                    ] + matrizDist[k][j]);
            }
        }
    }
}
```

3.7 Kruskal

```
// Asume grafo representado como lista de aristas.

ll kruskal(ll n, vector<pair<ll, pair<ll, ll>>> &lista_edges){
    // Devuelve el costo del AGM. En caso de que no sea conexo, devuelve
        -1.
    sort(lista_edges.begin(),lista_edges.end());
    DisjointSet dsu(n+1);
    ll res = 0;
    for(auto e : lista_edges){
        ll peso = e.first;
        ll x = (e.second).first;
        ll y = (e.second).second;

        if (dsu.findSet(x) != dsu.findSet(y)){
            dsu.unionSet(x, y);
            res += peso;
            n--;  // Para verificar luego si es posible visitar todos.
        }
    }

    if(n!=1){res = -1;}  // No era conexo.
    return res;
}
```

3.8 Toposort

```
enum Color {WHITE, GREY, BLACK};  // Sin visitar / en proceso /
    Procesado.

// Devuelve true si encuentra un ciclo.
bool tdfs(ll v, const vector<vector<ll>> &ady, vector<Color> &color,
    vector<ll> &orden){
    color[v] = GREY;

    for(auto u : ady[v]){
        if(color[u] == GREY){  // Si encuentra un nodo en proceso, hay
            un ciclo.
            return true;
        }
        else if (color[u] == WHITE){  // Si encuentra un nodo no
            visitado, realiza DFS.
            if(tdfs(u, ady, color, orden)) return true;
        }
    }

    orden.pb(v);
    color[v] = BLACK;
    return false;
}

// Devuelve true sii existe un ciclo en G.
```

```cpp
// Si no existe ciclo, en orden queda almacenado un orden topologico de G.
bool toposort(const vector<vector<ll>> &ady, vector<ll> &orden){
    vector<Color> color(ady.size(), WHITE);
    orden.clear();

    for(ll v=0; v < ady.size(); v++){
        if (color[v] == WHITE){
            if(tdfs(v, ady, color, orden)) return true;
        }
    }

    reverse(orden.begin(), orden.end());
    return false;
}
```

4 Searching

4.1 Binary search

```cpp
// Asumiento que quiero hacer una busqueda binaria en el rango [0, n)
// Importante: chequear el indice porque en caso de que no exista elemento que cumple P(X) puede haber problemas

// Calcular extremo derecho que cumple P(X)
int l = -1; // extremo izquierdo del rango de busqueda -1
int r = n; // extremo derecho del rango de busqueda +1
while(r - l > 1) { // mientras que la distancia entre las fronteras sea >1 (es decir, mientras que no esten contiguas)
    int mid = (l + r) / 2;
    if(P(mid)) {
        l = mid;
    } else {
        r = mid;
    }
}
// l es el ultimo elemento que cumple P(X)

// Calcular extremo izquierdo que cumple P(X)
int l = -1;
int r = n;
while(r - l > 1) {
    int mid = (l + r) / 2;
    if(!P(mid)) {
        l = mid;
    } else {
        r = mid;
    }
}

// r es el primer elemento que cumple P(X)
```

4.2 Integer ternary search

```cpp
// Busqueda ternaria entera sobre [lower, high].

ll l = lower; // extremo izquierdo del rango de busqueda.
ll r = high; // extremo derecho del rango de busqueda.
while(l < r) {
    ll mid = (l + r) / 2;
    if(f(mid) < f(mid+1)) {  // (<) Busca el minimo | (>) Busca el maximo.
        r = mid;
    } else {
        l = mid+1;
    }
}

// Respuesta: f(l).
```

4.3 Intervals

```cpp
// We suppose that the intervals are correct: p.first <= p.second
// The intervals describe this: [p.first, p.second]

// This function checks if the intervals p1 and p2 are disjoint
bool areDisjoint(pair<ll, ll> &p1, pair<ll, ll> &p2) {
    ll startingTimeP1 = p1.first;
    ll endingTimeP1 = p1.second;
    ll startingTimeP2 = p2.first;
    ll endingTimeP2 = p2.second;

    // This option checks supposing that p1 starts before than p2
    bool option1 = endingTimeP1 < startingTimeP2;
    // This option checks supposing that p2 starts before than p1
    bool option2 = endingTimeP2 < startingTimeP1;

```

```
    return option1 || option2;
}

// This function checks if p1 is included in p2
bool isIncluded(pair<ll, ll> &p1, pair<ll, ll> &p2){
    ll startingTimeP1 = p1.first;
    ll endingTimeP1 = p1.second;
    ll startingTimeP2 = p2.first;
    ll endingTimeP2 = p2.second;

    bool res = startingTimeP2 <= startingTimeP1 && endingTimeP1 <=
        endingTimeP2;
    return res;
}

bool belongsToTheInterval(pair<ll, ll> &p1, ll x){
    ll startingTimeP1 = p1.first;
    ll endingTimeP1 = p1.second;

    return startingTimeP1 <= x && x <= endingTimeP1;
}

// This function helps me to sort the intervals to have first those
    which finish earlier. In case of a tie, I choose first the interval
    which starts earlier.
bool customCompare(pair<ll, ll> &p1, pair<ll, ll> &p2){
    ll endingTimeP1 = p1.second;
    ll endingTimeP2 = p2.second;

    if (endingTimeP1 < endingTimeP2){
        return true;
    } else if (endingTimeP1 > endingTimeP2){
        return false;
    }

    ll startingTimeP1 = p1.first;
    ll startingTimeP2 = p2.first;

    return startingTimeP1 < startingTimeP2;
}

// This function helps me to sort the intervals to have first those
    which start earlier. In case of a tie, I choose first the interval
    which ends later.
bool customCompare2(pair<ll,ll> &p1, pair<ll,ll> &p2){
    ll startingTimeP1 = p1.first;
    ll startingTimeP2 = p2.first;

    if (startingTimeP1 < startingTimeP2){
        return true;
    } else if (startingTimeP1 > startingTimeP2){
        return false;
    }

    ll endingTimeP1 = p1.second;
    ll endingTimeP2 = p2.second;

    return endingTimeP1 > endingTimeP2;
}

ll maximumNumberOfDisjointIntervals(vector<pair<ll, ll>> &intervals){
    int n = intervals.size();
    if (n == 0) return 0;
    sort(all(intervals), customCompare);

    pair<ll, ll> lastInterval = intervals[0];
    int res = 1;

    forsn(i,1,n){
        pair<ll, ll> currentInterval = intervals[i];

        if (areDisjoint(currentInterval,lastInterval)){
            lastInterval = currentInterval;
            res++;
        }
    }

    return res;
}


pair<ll, ll> redundantInterval(vector<pair<ll, ll>> &intervals){
    int n = intervals.size();
    pair<ll, ll> res = {UNDEFINED, UNDEFINED};

    sort(all(intervals), customCompare2);
```

```
97
98      forn(i, n-1){
99          pair<ll, ll> currentInterval = intervals[i];
100         pair<ll, ll> nextInterval = intervals[i+1];
101
102         if (isIncluded(nextInterval, currentInterval)){
103             res = nextInterval;
104             break;
105         } else if (i+2 <= n-1){
106             pair<ll, ll> laterInterval = intervals[i+2];
107
108             if (belongsToTheInterval(currentInterval, nextInterval.first
                    )){
109                 pair<ll,ll> newInterval = {currentInterval.second + 1,
                        nextInterval.second};
110
111                 if (isIncluded(newInterval, laterInterval)){
112                     res = nextInterval;
113                     break;
114                 }
115             }
116
117         }
118     }
119
120     return res;
121 }
```