

Sale con dp y fritas



Contents

1	Template	1
2	Data structures	2
2.1	Disjoint Set	2
2.2	MinHeap	2
3	Grafos	3
3.1	BFS	3
3.2	DFS	4
3.3	Bellman-Ford	4
3.4	Dijkstra	5
3.5	Floyd-Warshall	5
3.6	Dantzig	6
3.7	Kruskal	6
3.8	Toposort	6
4	Searching	7
4.1	Binary search	7
4.2	Integer ternary search	7
4.3	Intervals	7
5	Other	9
5.1	Enumerar abecedario	9

1 Template

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  typedef long long ll;
5  typedef long double ld;
6
7  const ll UNDEFINED = -1;
8  const int MAX_N = 1e5 + 1;
9  const int MOD = 1e9 + 7;
10 const int INF = 1e9;
11 const ll LINF = 1e18;
12 const ll zero = 0;
13 const ld EPSILON = 1e-10;
14 const double PI = acos(-1.0);
15
16 #define pb push_back
17 #define fst first
18 #define snd second
19 #define esta(x,c) ((c).find(x) != (c).end()) // Devuelve true si x es
    un elemento de c.
20 #define all(c) (c).begin(),(c).end()
21
22 #define DBG(x) cerr << #x << " = " << (x) << endl
23 #define RAYA cerr << "-----" << endl
24
25 #define forn(i,n) for (int i=0;i<(int)(n);i++)
26 #define forsn(i,s,n) for (int i=(s);i<(int)(n);i++)
27 #define dforn(i,n) for(int i=(int)((n)-1);i>=0;i--)
28 #define dforsn(i,s,n) for(int i=(int)((n)-1);i>=(int)(s);i--)
29
30 // Show vector
31 template <typename T>
32 ostream & operator <<(ostream &os, const vector<T> &v) {
33     os << "[";
34     forn(i, v.size()) {
35         if (i > 0) os << ",";
36         os << v[i];
37     }
38     return os << "]";
39 }

```

```

40
41 // Show pair
42 template <typename T1, typename T2>
43 ostream & operator <<(ostream &os, const pair<T1, T2> &p) {
44     os << "{" << p.first << "," << p.second << "}";
45     return os;
46 }
47
48 // Show set
49 template <typename T>
50 ostream & operator <<(ostream &os, const set<T> &s) {
51     os << "{";
52     for(auto it = s.begin(); it != s.end(); it++){
53         if(it != s.begin()) os << ",";
54         os << *it;
55     }
56     return os << "}";
57 }
58
59 // ##### //
60
61 int main()
62 {
63     cin.tie(0);
64     cin.sync_with_stdio(0);
65
66     return 0;
67 }
    
```

2 Data structures

2.1 Disjoint Set

```

1 struct DisjointSet{
2     vector<ll> parent, rnk;
3     ll numOfComponents;
4
5     DisjointSet(ll n){
6         rnk.assign(n, 0);
7         for (ll i = 0; i < n; i++)
8             parent.push_back(i);
9         numOfComponents = n;
10    }
    
```

```

11
12 ll findSet(ll x){
13     if(parent[x]==x) return x;
14     parent[x] = findSet(parent[x]);
15     return parent[x];
16 }
17
18 void unionSet(ll x, ll y){
19     // Encontrar los representantes del conjunto.
20     x = findSet(x);
21     y = findSet(y);
22
23     // Si los conjuntos son disjuntos:
24     if (x != y){
25         // Pongo al que tiene menos rango por debajo del de mayor
26         // rango.
27         if (rnk[x] < rnk[y]){
28             parent[x] = y;
29         } else if (rnk[x] > rnk[y]){
30             parent[y] = x;
31         } else { // Si tienen el mismo rango, incremento del rango.
32             (rnk[x] == rnk[y])
33             parent[y] = x;
34             rnk[x]++;
35         }
36         numOfComponents--;
37     }
38 }
39
40 bool same(ll x, ll y){
41     return findSet(x) == findSet(y);
42 }
    
```

2.2 MinHeap

```

1 template <typename T>
2 using min_heap = priority_queue<T, vector<T>, greater<T>>;
3
4 // ejemplo:
5 // min_heap<pair<int, int>> q;
    
```

3 Grafos

3.1 BFS

```

1 // BFS Normal
2 void bfs(ll inicio, vector<vector<ll>> &ady, vector<bool> &vis, vector<
   ll> &parent){
3     queue<ll> q;
4     q.push(inicio);
5     vis[inicio] = true;
6     while(!q.empty()){
7         ll v = q.front(); // v es el vertice que estoy procesando
8         q.pop();
9         for (ll u : ady[v]){
10             if (!vis[u]){
11                 vis[u] = true;
12                 parent[u] = v;
13                 q.push(u);
14             }
15         }
16     }
17 }
18
19 // BFS que calcula los padres de los vertices
20 void calculatingParents(ll v, vector<vector<ll>> &adjList, vector<bool>
   &visited, vector<ll> &parents){
21     ll n = adjList.size();
22     visited[v] = true;
23     parents[v] = v;
24
25     queue<ll> q;
26     q.push(v);
27
28     while (!q.empty()){
29         ll u = q.front();
30         q.pop();
31
32         for (int i = 0; i < adjList[u].size(); i++){
33             ll w = adjList[u][i];
34             if (!visited[w]){
35                 parents[w] = u;
36                 visited[w] = true;
37                 q.push(w);

```

```

38     }
39 }
40 }
41 }
42
43 // BFS que chequea si un grafo es bipartito
44 bool isBipartite(ll v, vector<vector<ll>> &adjList, vector<ll> &teams){
45     bool res = true;
46     ll n = adjList.size();
47     teams[v] = 1;
48
49     queue<ll> q;
50     q.push(v);
51
52     while (!q.empty()){
53         ll u = q.front();
54         q.pop();
55
56         for (int i = 0; i < adjList[u].size(); i++){
57             ll w = adjList[u][i];
58             if (teams[w] == 0){ // If the node isn't painted, paint it.
59                 if (teams[u] == 1){
60                     teams[w] = 2;
61                 } else {
62                     teams[w] = 1;
63                 }
64                 q.push(w);
65             } else {
66                 if (teams[w] == teams[u]){ // If the node is painted, I
67                     // check that its color is different from the v.
68                     res = false;
69                     break;
70                 }
71             }
72         }
73     }
74     return res;
75 }
76
77 // BFS que calcula el numero de componentes conexas
78 ll numberOfConnectedComponents(vector<vector<ll>> &adjList){
79     ll n = adjList.size();

```

```

80     vector<bool> visited(n, false);
81     vector<ll> parents(n, UNDEFINED);
82     ll res = 0;
83
84     for (int i = 0; i < n; i++){
85         if (!visited[i]){
86             bfs(i, adjList, visited, parents);
87         }
88     }
89
90     return res;
91 }
92
93 // BFS que calcula el numero de vertices en una componente conexas
94 ll numberOfVerticesInConnectedComponent(ll v, vector<vector<ll>> &
    adjList){
95     ll n = adjList.size();
96     vector<bool> visited(n, false);
97     visited[v] = true;
98     ll res = 1;
99
100     queue<ll> q;
101     q.push(v);
102
103     while (!q.empty()){
104         ll w = q.front();
105         q.pop();
106
107         for (int i = 0; i < adjList[w].size(); i++){
108             ll u = adjList[w][i];
109
110             if (!visited[u]){
111                 q.push(u);
112                 visited[u] = true;
113                 res++;
114             }
115         }
116     }
117
118     return res;
119 }
    
```

3.2 DFS

```

1 // DFS simple
2 void dfs(ll v, vector<vector<ll>> &ady, vector<bool> &vis){
3     vis[v] = true;
4     for(ll u : ady[v]){
5         if (!vis[u]){
6             dfs(u, ady, vis);
7         }
8     }
9 }
10
11 // DFS que me dice si existe un ciclo en un grafo (no dirigido) y en
    caso de existir, retorna la back-edge
12 pair<bool, edge> hasCycle(int v, vector<vector<int>> &adjList, vector<
    bool> &visited, vector<int> &parents){
13     visited[v] = true;
14     edge e = {UNDEFINED, UNDEFINED};
15     pair<bool, edge> res = {false, e};
16
17     for (int i = 0; i < adjList[v].size(); i++){
18         int w = adjList[v][i];
19         if (!visited[w]){
20             parents[w] = v;
21             res = hasCycle(w, adjList, visited, parents);
22             if (res.first){
23                 break;
24             }
25         } else if (visited[w] && parents[v] != w && parents[v] !=
            UNDEFINED){
26             edge backEdge = {w,v};
27             return {true, backEdge};
28         }
29     }
30
31     return res;
32 }
    
```

3.3 Bellman-Ford

```

1 using peso = ll;
2 using indice_nodo = ll;
3 using nodo_pesado = pair<peso, indice_nodo>;
4
5 struct Edge {
    
```

```

6     ll a, b, cost;
7
8     Edge(ll desde, ll hasta, ll c) : a(desde), b(hasta), cost(c) {}
9 };
10
11 // #####
12
13 // Asume grafo representado como lista de aristas.
14
15 bool bellman_ford(ll n, indice_nodo inicio, vector<Edge> &edges, vector<
    peso> &dist){
16     // Devuelve true sii existe un ciclo de longitud negativa.
17     // Calcula SSSP en dist.
18
19     // Obtiene las distancias mas cortas desde inicio hacia todos.
20     dist[inicio] = 0;
21     for(ll i=0; i<n; i++){
22         for(Edge e : edges){
23             if(-LINF < dist[e.a] && dist[e.a] < LINF){
24                 dist[e.b] = min(dist[e.b], dist[e.a] + e.cost);
25             }
26         }
27     }
28
29     // Detectar ciclo de longitud negativa.
30     for(Edge e : edges){
31         if(-LINF < dist[e.a] && dist[e.a] < LINF){
32             if (dist[e.a] + e.cost < dist[e.b]){
33                 return true;
34             }
35         }
36     }
37     return false;
38 }
    
```

3.4 Dijkstra

```

1 using peso = ll;
2 using indice_nodo = ll;
3 using nodo_pesado = pair<peso, indice_nodo>;
4
5 // #####
6
    
```

```

7 // Devuelve el vector de distancias desde inicio al i-esimo vertice.
8 vector<ll> dijkstra(const indice_nodo inicio, const vector<vector<
    nodo_pesado>> &ady){
9     vector<ll> distancia(ady.size(), LINF);
10    // vector<ll> parent(ady.size(), UNDEFINED);
11    vector<bool> vis(ady.size(), false);
12
13    distancia[inicio] = 0;
14    set<nodo_pesado> q;
15
16    q.insert({0, inicio});
17
18    while(!q.empty()){
19        ll v = q.begin() -> second;
20        q.erase(q.begin());
21
22        if (vis[v]) {continue;}
23        vis[v] = true;
24        for(const auto& [longitud, u] : ady[v]){
25            // Longitud del camino de v hacia u.
26            // Relax:
27            if(distancia[v] + longitud < distancia[u]){
28                q.erase({distancia[u], u});
29                distancia[u] = distancia[v] + longitud;
30                // parent[u] = v;
31                q.insert({distancia[u], u});
32            }
33        }
34    }
35    return distancia;
36 }
    
```

3.5 Floyd-Warshall

```

1 // La matriz tuvo que ser inicializada como:
2 // d(s,v) = w(s,v) si existe
3 // d(s,v) = 0    si s = v
4 // d(s,v) = INF si no
5
6 void floyd_warshall (ll n, vector<vector<ll>> &matrizDist){
7     // matrizDist en la entrada era la matriz de distancias directas,
8     // luego del algoritmo queda calculada en ella APSP. (Distancias
9     // minimas i->j)
10 }
    
```

```

8   for (ll k = 0; k < n; ++k) {
9       for (ll i = 0; i < n; ++i) {
10          for (ll j = 0; j < n; ++j) {
11              if (matrizDist[i][k] < LINf && matrizDist[k][j] < LINf){
12                  matrizDist[i][j] = min(matrizDist[i][j], matrizDist[
13                      i][k] + matrizDist[k][j]);
14              }
15          }
16      }
17 }
    
```

3.6 Dantzig

```

1 void dantzig(ll n, vector<vector<peso>> &matrizDist){
2     // matrizDist en la entrada era la matriz de distancias diractas,
3     // luego del algoritmo queda calculada en ella APSP.
4     for(ll k=0; k<n; k++){
5         for(ll i=0; i<k; i++){
6             for(ll j=0; j<k; j++){
7                 matrizDist[i][k] = min(matrizDist[i][k], matrizDist[i][j]
8                     + matrizDist[j][k]);
9                 matrizDist[k][i] = min(matrizDist[k][i], matrizDist[k][j]
10                    + matrizDist[j][i]);
11             }
12         }
13         for(ll i=0; i<k; i++){
14             for(ll j=0; j<k; ++j){
15                 matrizDist[i][j] = min(matrizDist[i][j], matrizDist[i][k]
16                    + matrizDist[k][j]);
17             }
18         }
19     }
20 }
    
```

3.7 Kruskal

```

1 // Asume grafo representado como lista de aristas.
2
3 ll kruskal(ll n, vector<pair<ll, pair<ll, ll>>> &lista_edges){
4     // Devuelve el costo del AGM. En caso de que no sea conexo, devuelve
5     // -1.
6 }
    
```

```

5     sort(lista_edges.begin(), lista_edges.end());
6     DisjointSet dsu(n+1);
7     ll res = 0;
8     for(auto e : lista_edges){
9         ll peso = e.first;
10        ll x = (e.second).first;
11        ll y = (e.second).second;
12
13        if (dsu.findSet(x) != dsu.findSet(y)){
14            dsu.unionSet(x, y);
15            res += peso;
16            n--; // Para verificar luego si es posible visitar todos.
17        }
18    }
19
20    if(n!=1){res = -1;} // No era conexo.
21    return res;
22 }
    
```

3.8 Toposort

```

1 enum Color {WHITE, GREY, BLACK}; // Sin visitar / en proceso /
2 // Procesado.
3 // Devuelve true si encuentra un ciclo.
4 bool tdfs(ll v, const vector<vector<ll>> &ady, vector<Color> &color,
5     vector<ll> &orden){
6     color[v] = GREY;
7
8     for(auto u : ady[v]){
9         if(color[u] == GREY){ // Si encuentra un nodo en proceso, hay
10            un ciclo.
11            return true;
12        }
13        else if (color[u] == WHITE){ // Si encuentra un nodo no
14            visitado, realiza DFS.
15            if(tdfs(u, ady, color, orden)) return true;
16        }
17    }
18
19    orden.pb(v);
20    color[v] = BLACK;
21    return false;
22 }
    
```

```

19 }
20
21 // Devuelve true sii existe un ciclo en G.
22 // Si no existe ciclo, en orden queda almacenado un orden topologico de
   G.
23 bool toposort(const vector<vector<ll>> &ady, vector<ll> &orden){
24     vector<Color> color(ady.size(), WHITE);
25     orden.clear();
26
27     for(ll v=0; v < ady.size(); v++){
28         if (color[v] == WHITE){
29             if(tdfs(v, ady, color, orden)) return true;
30         }
31     }
32
33     reverse(orden.begin(), orden.end());
34     return false;
35 }

```

4 Searching

4.1 Binary search

```

1 // Asumiento que quiero hacer una busqueda binaria en el rango [0, n)
2 // Importante: chequear el indice porque en caso de que no exista
   elemento que cumple P(X) puede haber problemas
3
4 // Calcular extremo derecho que cumple P(X)
5 int l = -1; // extremo izquierdo del rango de busqueda -1
6 int r = n; // extremo derecho del rango de busqueda +1
7 while(r - l > 1) { // mientras que la distancia entre las fronteras sea
   >1 (es decir, mientras que no esten contiguas)
8     int mid = (l + r) / 2;
9     if(P(mid)) {
10         l = mid;
11     } else {
12         r = mid;
13     }
14 }
15 // l es el ultimo elemento que cumple P(X)
16
17 // Calcular extremo izquierdo que cumple P(X)
18 int l = -1;

```

```

19 int r = n;
20 while(r - l > 1) {
21     int mid = (l + r) / 2;
22     if(!P(mid)) {
23         l = mid;
24     } else {
25         r = mid;
26     }
27 }
28
29 // r es el primer elemento que cumple P(X)

```

4.2 Integer ternary search

```

1 // Busqueda ternaria entera sobre [lower, high].
2
3 ll l = lower; // extremo izquierdo del rango de busqueda.
4 ll r = high; // extremo derecho del rango de busqueda.
5 while(l < r) {
6     ll mid = (l + r) / 2;
7     if(f(mid) < f(mid+1)) { // (<) Busca el minimo | (>) Busca el
   maximo.
8         r = mid;
9     } else {
10         l = mid+1;
11     }
12 }
13
14 // Respuesta: f(l).

```

4.3 Intervals

```

1 // We suppose that the intervals are correct: p.first <= p.second
2 // The intervals describe this: [p.first, p.second]
3
4 // This function checks if the intervals p1 and p2 are disjoint
5 bool areDisjoint(pair<ll, ll> &p1, pair<ll, ll> &p2) {
6     ll startingTimeP1 = p1.first;
7     ll endingTimeP1 = p1.second;
8     ll startingTimeP2 = p2.first;
9     ll endingTimeP2 = p2.second;
10
11     // This option checks supposing that p1 starts before than p2
12     bool option1 = endingTimeP1 < startingTimeP2;

```

```

13 // This option checks supposing that p2 starts before than p1
14 bool option2 = endingTimeP2 < startingTimeP1;
15
16 return option1 || option2;
17 }
18
19 // This function checks if p1 is included in p2
20 bool isIncluded(pair<ll, ll> &p1, pair<ll, ll> &p2){
21     ll startingTimeP1 = p1.first;
22     ll endingTimeP1 = p1.second;
23     ll startingTimeP2 = p2.first;
24     ll endingTimeP2 = p2.second;
25
26     bool res = startingTimeP2 <= startingTimeP1 && endingTimeP1 <=
27         endingTimeP2;
28     return res;
29 }
30 bool belongsToTheInterval(pair<ll, ll> &p1, ll x){
31     ll startingTimeP1 = p1.first;
32     ll endingTimeP1 = p1.second;
33
34     return startingTimeP1 <= x && x <= endingTimeP1;
35 }
36
37 // This function helps me to sort the intervals to have first those
38 // which finish earlier. In case of a tie, I choose first the interval
39 // which starts earlier.
40 bool customCompare(pair<ll, ll> &p1, pair<ll, ll> &p2){
41     ll endingTimeP1 = p1.second;
42     ll endingTimeP2 = p2.second;
43
44     if (endingTimeP1 < endingTimeP2){
45         return true;
46     } else if (endingTimeP1 > endingTimeP2){
47         return false;
48     }
49
50     ll startingTimeP1 = p1.first;
51     ll startingTimeP2 = p2.first;
52
53     return startingTimeP1 < startingTimeP2;
54 }

```

```

53
54 // This function helps me to sort the intervals to have first those
55 // which start earlier. In case of a tie, I choose first the interval
56 // which ends later.
57 bool customCompare2(pair<ll, ll> &p1, pair<ll, ll> &p2){
58     ll startingTimeP1 = p1.first;
59     ll startingTimeP2 = p2.first;
60
61     if (startingTimeP1 < startingTimeP2){
62         return true;
63     } else if (startingTimeP1 > startingTimeP2){
64         return false;
65     }
66
67     ll endingTimeP1 = p1.second;
68     ll endingTimeP2 = p2.second;
69
70     return endingTimeP1 > endingTimeP2;
71 }
72
73 ll maximumNumberOfDisjointIntervals(vector<pair<ll, ll>> &intervals){
74     int n = intervals.size();
75     if (n == 0) return 0;
76     sort(all(intervals), customCompare);
77
78     pair<ll, ll> lastInterval = intervals[0];
79     int res = 1;
80
81     forsn(i, 1, n){
82         pair<ll, ll> currentInterval = intervals[i];
83
84         if (areDisjoint(currentInterval, lastInterval)){
85             lastInterval = currentInterval;
86             res++;
87         }
88     }
89
90     return res;
91 }
92
93 pair<ll, ll> redundantInterval(vector<pair<ll, ll>> &intervals){
94     int n = intervals.size();

```



```
94 pair<ll, ll> res = {UNDEFINED, UNDEFINED};
95
96 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         )){
110             pair<ll, ll> newInterval = {currentInterval.second + 1,
111             nextInterval.second};
112
113             if (isIncluded(newInterval, laterInterval)){
114                 res = nextInterval;
115                 break;
116             }
117         }
118     }
119
120     return res;
121 }
```

5 Other

5.1 Enumerar abecedario

```
1 // 'A' es 97 | 'a' es 65.
2 int transformarLetra(char x) {return (int)x - 97;}
3 char transformarNumero(int x) {return (char)(x + 97);}
```