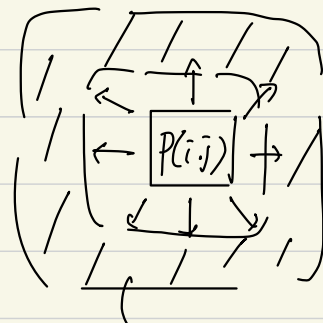


H37 ①

(1) 以下便宜上  $N \times N$  の面行列の  $i$  行,  $j$  列目の要素を  $P(i, j)$  と表記し  
 $P(i, j)$  に隣接 (4方向を含む) する要素のインデックスのペアを保存したリストを  
 $adj(i, j)$  とする。

また,  $P(i, j)$  を一度訪問したかを  
 保存する 2次元面行列を  $visited[N][N]$  とする。  
 (内部は全て false で初期化されているものとする)



この時、セル  $(s_x, s_y)$  が与えられた時  $(s_x, s_y)$  を含む連結領域を  $O(N^2)$   
 で計算する。深さ優先探索を応用したアルゴリズムの疑似コードを以下に示す。  
 (なお、適宜コード中に // (コメントアウト) を挿入している)

```
void dfs ( cx, cy ) {
    assert visited [cx][cy] = true; // visited [cx][cy] は訪問済み
    for [ adj_x, adj_y ] in adj [cx][cy] {
        if ( not visited [adj_x][adj_y] & 未渡が (1より値以上) ) {
            visited [adj_x][adj_y] = true; // 隣接頂点のうち未訪問の頂点
            dfs ( adj_x, adj_y );           // 未渡が (1より値以上の頂点
        }                                  // を発見したら、移動。
    }
}

// ( cx, cy ) は訪問済みにして初期化する
visited [cx][cy] = true;
dfs ( cx, cy );

// ( cx, cy ) との連結領域を調べるため、visited [i][j] の true の個数を計算
int area = 0;
for i { 0... N-1 } for j { 0... N-1 } {
```


```
if (visited[i][j]) area++;
```

↓

// 始点  $(x, y)$  への連結領域の個数を出力.

```
printf ("始点への連結領域 : %d", area);
```

上記のアルゴリズムが  $O(N^2)$  で動作することを示す.

全ての頂点は  $N^2$  であり、かつ、1度訪問した頂点に対して再度訪問を行うことはないので、高々  $O(N)$  回の再帰及び、隣接要素へのアクセスしか行わない. 以上より、全体の処理も  $O(N^2)$  のオーダーで発生しない. 

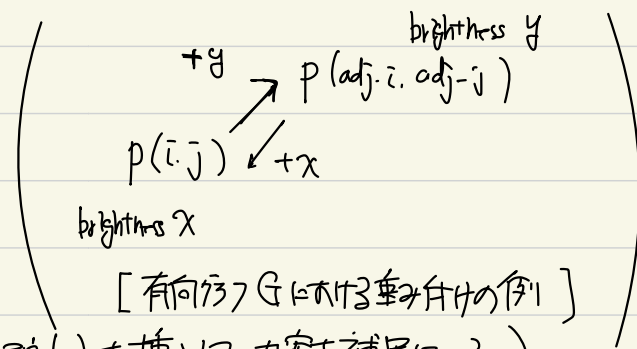
(2)  $p(i, j)$  に隣接する要素  $p(\text{adj}_i, \text{adj}_j)$  に移動する際に、マス  $p(\text{adj}_i, \text{adj}_j)$  の輝度に等しいだけのコストをもった有向グラフ  $G$  として、配列  $P$  をとらえる.

始点を  $(sx, sy)$ 、終点を  $(gx, gy)$  とした時.

$(sx, sy)$  から  $(gx, gy)$  への最短経路を求める問題に帰着される. 以下最短経路を計算する.

dijkstra法を用いた  $O(N^2 \log N)$  のアルゴリズムの

疑似コードを示す. (なお、適宜コード中に // (コメントアウト) を挿入して内容を補足している)



```
dist[0..n-1][0..n-1] = ∞; // dist[i][j] = (sx, sy) ~ (i, j) の最短経路コスト
par[0..n-1][0..n-1] = -1; // par[i][j] = (sx, sy) ~ (i, j) の最短経路における (i, j) の親点 (存在しないなら -1)
```

```
heap = [(0, (sx, sy))]; // (最短コスト, 1点の位置情報) を含める優先度付きキュー.
dist[sx][sy] = 0; // 最短コストの小さいものから順に取り出せる構造を付
```

```
while (heap が空でない) {
```

```
    cur_dist, (cx, cy) = heap から取り出した要素
```

```
    if (dist[cx][cy] != cur_dist) { continue; } // 最新情報でない場合、無駄なので枝刈り
```

```
for [ adjx, adjy ] in adj [ cx ] [ cy ] {
```

```
    new_dist = cur_dist + p [ adjx ] [ adjy ] . prghtness ; //隣接頂点への移動コストを計算
```

```
    if ( new_dist < dist [ adjx ] [ adjy ] ) {
```

```
        dist [ adjx ] [ adjy ] = new_dist ; //最短距離が更新された場合、最短経路の
```

```
        par [ adjx ] [ adjy ] = ( cx , cy ) ; //親頂点も更新した後、更新後データをheapに追加
```

```
        heap ( { dist [ adjx ] [ adjy ] , ( adjx , adjy ) } ) を加える。
```

```
}
```

```
int min_cost = dist [ gx ] [ gy ] ;
```

```
printf ( "最小移動コスト: %d", min_cost ) ; //最短経路の更新
```

次に、最短経路も復元する際は  $par$  配列の情報を用いて、 $(gx, gy)$  から  $(sx, sy)$

にいたるまで  $par [ cx ] [ cy ] \leftarrow (cx, cy)$  の遷移をたどればよい。

最短経路は同じ頂点を含まず、高々  $O(N^2)$  のコストしかたどらねるので、土記は  $O(N^2)$  で終了。

また、dijkstra 法自体は、与った頂点数、辺数を  $(V, E)$  とした時  $O(V \log E)$  程度で

終了し、 $V = N^2$ ,  $E = O(N^2)$  がこの場合成り立つため、 $O(N^2 \log N^2) = O(N \log N)$

でdijkstra法は動作する。(優先度付きキュー(ここでは素朴なデータ構造を用いた)を用いた)

