ACM ICPC2013 国内予選　問題 D 素数洞穴

- 洞穴数は最大で 1000000 なので、1000×1000 の 2 次元配列で表現可能であるが、上下左右に洞穴が無い行と列を加えることにし、1002×1002 の 2 次元配列を使用
- 2 次元配列の要素は、洞穴番号、調査可能素数洞穴数、最終調査素数洞穴番号を要素として持つ構造体を使用
- １番洞穴は(500,500)の位置に置く
- 洞穴の渦巻状配置の作成
  - 右 1 上 1 左 2 下 2 右 3 上 3 左 4 下 4 右 5 上 5 左 6 下 6 ‥‥
- 移動可能な下位の３つの洞穴の調査可能素数洞穴数の最大値を np とし、np を与える洞穴の中の最終素数洞穴番号の最大値を lp とする。
  - 素数洞穴　→　調査可能素数洞穴数 = np + 1
    np = 0 なら最終調査素数洞穴番号 = この洞穴の番号
    np > 0 なら lp
  - 非素数洞穴　→　調査可能素数洞穴数 = np
    最終調査素数洞穴番号 = lp
- アルゴリズム：
  - 動的計画法：　最下層の洞穴から上層に向けて最初の洞穴が現れるまで上記を計算
  - 深さ優先探索：　最初の洞穴から深さ優先探索(post order)で再帰的に上記を計算

【動的計画法によるプログラム例】

```
// ACM ICPC2013 Domestic Problem D
// http://sparth.u-aizu.ac.jp/icpc2013/d_problems.php?lang=jp#section_D
// Filename:        pd_dp.c
// Compile:         cc -O2 pd_dp.c
// Execution:        ./a.out < D0 > D0.result
// Check:   diff D0.result D0.ans
// Algorithm:       dynamic programming

#include <stdio.h>
#define MAX_M 1000000
#define DIV_MM 1000          // DIV_MM = ceiling(sqrt(MAX_M))
#define MAP_SIZE 1002        // MAP_SIZE = ceiling(sqrt(MAX_M))+2
                             // マップの一辺の大きさ。洞穴の周囲も含む。
#define CENTER 500           // １番の洞穴の位置
```

```
typedef struct map_type {
   int id;   // 洞穴の番号
   int np;  // ここから始めて調査可能な素数洞穴の個数
   int lp;   // 上記を実現する経路で最後に通過した素数洞穴番号の最大値
} map_t;

map_t map[MAP_SIZE][MAP_SIZE];      // 洞穴のマップ
int is_prime[MAX_M+1];                    // 素数表。is_prime[i]=1 <--> i は素数

// 洞穴番号をふるための移動方向   0→右, 1→上, 2→左, 3→下
int dx[ ] = {1, 0, -1, 0};
int dy[ ] = {0, -1, 0, 1};
int m, n;   // m = 洞穴の個数、  n = 最初の洞穴番号
int nx, ny;// 最初の洞穴の位置
int min_x, max_x, min_y, max_y;          // マップ: 幅の最小値,最大値,高さの最小値,最大値

void init_map(void)  // 洞穴のマップを初期化する
{
   int i,j;
   for(i=0; i<MAP_SIZE; i++)
      for(j=0; j<MAP_SIZE; j++){
         map[i][j].id = map[i][j].np = map[i][j].lp = 0;
      }
}

// マップの作成
void gen_map(void)
{
   int x, y, len, dir, i,j,id;
   init_map();                          // マップの初期化
   x = y = CENTER;  // 1 番の洞穴の位置
   min_x = max_x = min_y = max_y = CENTER;
   map[y][x].id = 1;
   if(n == 1){                          // 最初に入る洞穴が 1 番ならその位置を nx,ny にセット
      nx = x; ny = y;
   }
   len = 1; dir = 0; id = 2;
   while(id <= m){     // id が洞穴の総数 m に達するまで以下を繰り返す
      for(i=0; i<2; i++){                    // 同じ長さの移動を 2 回繰り返す
         for(j=0; j<len; j++){               // dir 方向に長さ len 移動
            x += dx[dir]; y += dy[dir];
            if(x > max_x) max_x = x;         // max_x の更新
            if(x < min_x) min_x = x;         // min_x の更新
            if(y > max_y) max_y = y;         // max_y の更新
            if(y < min_y) min_y = y;         // min_y の更新
            if(id == n){                     // 最初に入る洞穴なら、位置を nx,ny にセット
               nx = x; ny = y;
            }
            map[y][x].id = id++;  // 洞穴番号をセット
            if(id > m) break;
         }
```

```
        dir = (dir+1)%4;                    // 進行方向を変える
        if(id > m) break;
      }
      len++;
    }
}

// 素数表の作成(エラトステネスのふるい)
// is_prime[i] = 1 <--> i は素数,  is_prime[i] = 0 <--> i は非素数
void gen_prime()
{
    int i,j;
    for(i=0; i<= MAX_M; i++) is_prime[i] = 1;          // とりあえず全て素数とする
    is_prime[0] = is_prime[1] = 0;                     // 0, 1 は非素数
    for(i=2; i<=DIV_MM; i++){                  // DIV_MM までの約数かどうかのチェックで十分
      if(is_prime[i]){                         // i が素数のとき
        for(j = 2*i; j<=MAX_M; j += i)         // i の倍数は非素数
            is_prime[j] = 0;
      }
    }
}

// 動的プログラミングで下層の洞穴から上層に向けて解を求める
map_t dp()
{
    map_t d;
    int x, y, np, lp, dx;
    for(y = max_y; y >= min_y; y--){           // 最下層の洞穴から上層に向けて
      for(x=min_x; x <= max_x; x++){           // その層の各洞穴に対して
        if(map[y][x].id == 0) continue;        // 洞穴でなければ何もしない
        // 移動可能の下層の３つの洞穴の中で、
        // 調査可能な洞穴数の最大値と最後の素数洞穴を求める
        np = lp = 0;
        for(dx=-1; dx<=1; dx++){// 移動可能な下層の３つの洞穴に対して
            if(map[y+1][x+dx].np > np){
              np = map[y+1][x+dx].np;          // 素数洞穴数の最大値を求める
              lp = map[y+1][x+dx].lp;          // その時の最後の素数洞穴番号の最大値を求める
            }
            else if(map[y+1][x+dx].np == np){          // 素数洞穴数が同じ場合は
              if(map[y+1][x+dx].lp > lp)
                lp = map[y+1][x+dx].lp;        // 最後の素数洞穴番号の最大値を求める
            }
        }
        map[y][x].np = np;          // 下層の洞穴から調査可能な素数洞穴数
        map[y][x].lp = lp;          // その時の最後の素数洞穴番号の最大値
        if(is_prime[map[y][x].id]){             // ここが素数洞穴の場合
            if(map[y][x].np == 0)               // 下に調査可能な素数洞穴がなければ
              map[y][x].lp = map[y][x].id;  // ここが最後の素数洞穴になる
            map[y][x].np++;                     // 調査可能な素数洞穴数を増やす
        }
        if(y==ny && x==nx) break;           // ここが最初の洞穴なら終了
      }
```

```
    if(y==ny && x==nx) break;                    // ここが最初の洞穴なら終了
  }
  return map[ny][nx];
}

int main()
{
  int x, y, dx, np, lp;
  map_t d;
  gen_prime();                                   // 素数表を作成
  while(1){
    scanf("%d %d¥n", &m, &n);                    // m, n を入力
    if(m==0 && n==0) break;  // 共に 0 なら終了
    gen_map();                                   // マップの作成
    d = dp();
    printf("%d %d¥n", d.np, d.lp);
  }
}
```

【深さ優先探索に基づくプログラム例】

```
// ACM ICPC2013 Domestic Problem D
// http://sparth.u-aizu.ac.jp/icpc2013/d_problems.php?lang=jp#section_D
// Filename:          pd_dfs.c
// Compile:           cc -O2 pd_dfs.c
// Execution:          ./a.out < D0 > D0.result
// Check:   diff D0.result D0.ans
// ALgorithm:         depth first search

#include <stdio.h>
#define MAX_M 1000000
#define DIV_MM 1000          // DIV_MM = ceiling(sqrt(MAX_M))
#define MAP_SIZE 1002        // MAP_SIZE = ceiling(sqrt(MAX_M))+2
                             // マップの一辺の大きさ。洞穴の周囲も含む。
#define CENTER 500           //  1 番の洞穴の位置

typedef struct map_type {
  int id;   // 洞穴の番号
  int np;  // ここから始めて調査可能な素数洞穴の個数
  int lp;   // 上記を実現する経路で最後に通過した素数洞穴番号の最大値
} map_t;

map_t map[MAP_SIZE][MAP_SIZE];      // 洞穴のマップ
int is_prime[MAX_M+1];              // 素数表。is_prime[i]=1 <--> i は素数

// 洞穴番号をふるための移動方向   0→右, 1→上, 2→左, 3→下
int dx[ ] = {1, 0, -1, 0};
int dy[ ] = {0, -1, 0, 1};
int m, n;  // m =  洞穴の個数、  n =  最初の洞穴番号
int nx, ny;// 最初の洞穴の位置
int min_x, max_x, min_y, max_y;          // マップ: 幅の最小値,最大値,高さの最小値,最大値

void init_map(void)  // 洞穴のマップを初期化する
```

```
{
   int i,j;
   for(i=0; i<MAP_SIZE; i++)
      for(j=0; j<MAP_SIZE; j++){
         map[i][j].id = map[i][j].lp = 0;
         map[i][j].np = -1;
      }
}

// マップの作成
void gen_map(void)
{
   int x, y, len, dir, i,j,id;
   init_map();                    // マップの初期化
   x = y = CENTER;  // 1 番の洞穴の位置
   min_x = max_x = min_y = max_y = CENTER;
   map[y][x].id = 1;
   if(n == 1){                    // 最初に入る洞穴が 1 番ならその位置を nx,ny にセット
      nx = x; ny = y;
   }
   len = 1; dir = 0; id = 2;
   while(id <= m){     // id が洞穴の総数 m に達するまで以下を繰り返す
      for(i=0; i<2; i++){                    // 同じ長さの移動を 2 回繰り返す
         for(j=0; j<len; j++){         // dir 方向に長さ len 移動
               x += dx[dir]; y += dy[dir];
               if(x > max_x) max_x = x;          // max_x の更新
               if(x < min_x) min_x = x;          // min_x の更新
               if(y > max_y) max_y = y;          // max_y の更新
               if(y < min_y) min_y = y;          // min_y の更新
               if(id == n){                      // 最初に入る洞穴なら、位置を nx,ny にセット
                  nx = x; ny = y;
               }
               map[y][x].id = id++;  // 洞穴番号をセット
               if(id > m) break;
         }
         dir = (dir+1)%4;                       // 進行方向を変える
         if(id > m) break;
      }
      len++;
   }
}

// 素数表の作成(エラトステネスのふるい)
// is_prime[i] = 1 <--> i は素数,   is_prime[i] = 0 <--> i は非素数
void gen_prime()
{
   int i,j;
   for(i=0; i<= MAX_M; i++) is_prime[i] = 1;          // とりあえず全て素数とする
   is_prime[0] = is_prime[1] = 0;                     // 0, 1 は非素数
   for(i=2; i<=DIV_MM; i++){              // DIV_MM までの約数かどうかのチェックで十分
      if(is_prime[i]){                    // i が素数のとき
         for(j = 2*i; j<=MAX_M; j += i)       // i の倍数は非素数
               is_prime[j] = 0;
```

```
    }
  }
}

// 深さ優先探索で求める
map_t dfs(int x, int y)
{
  map_t d;
  int id, np, lp, dx;
  if(map[y][x].id == 0){                    // 洞穴ではない
    d.id = d.np = d.lp = 0;
  }
  else if(map[y][x].np >= 0){               // 既に計算済み
    d =map[y][x];
  }
  else {
    np = lp = 0;
    id = map[y][x].id;
    for(dx=-1; dx <= 1; dx++){  // 移動可能な 3 つの下層の洞穴に対して
      d = dfs(x+dx, y+1);       // 素数洞穴数と最終素数洞穴番号を求める
      if(d.np > np){            // 素数洞穴数がより大きければ
          np = d.np;            // 素数洞穴数と
          lp = d.lp;            // 最終素数洞穴番号を更新
      }
      else if(d.np == np){      // 素数洞穴数が同じ場合は
          if(d.lp > lp) lp = d.lp;// 最終素数洞穴番号の大きい方を採用
      }
    }
    if(is_prime[id]){           // ここが素数洞穴の場合
      if(np == 0)               // ここから下層に調査可能な素数洞穴がなければ
          lp = id;              // この素数洞穴が最終素数洞穴となる
      np++;                             // 調査可能素数洞穴数を 1 増やす
    }
    d.id = id; d.np = np; d.lp = lp;
    map[y][x] = d;
  }
  return d;
}

int main()
{
  int x, y, dx, np, lp;
  map_t d;
  gen_prime();                          // 素数表を作成
  while(1){
    scanf("%d %d¥n", &m, &n);           // m, n を入力
    if(m==0 && n==0) break;  // 共に 0 なら終了
    gen_map();                          // マップの作成
    d = dfs(nx,ny);
    printf("%d %d¥n", d.np, d.lp);
  }
}
```