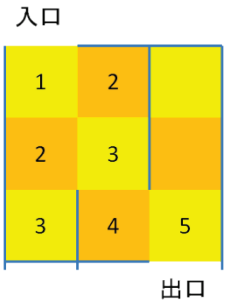
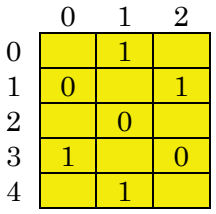
[ICPC2010 国内予選問題 B]　迷路と命ず



迷路に対する最短経路長を求める問題。迷路を表すマップ上で幅優先探索を行えばよい。例えば右図のような迷路では

1. 入口の部分に「1」のラベルをふる
2. マップ上で「１」のラベルが付いた部分に対して、その隣接位置へ移動可能(間に壁が無い)で、かつまだラベルが付いていない部分に「２」のラベルをふる
3. マップ上で「２」のラベルが付いた部分に対して、上記と同様の処理を繰り返し「３」のラベルをふる
4. このような操作を出口にラベルがふられるか、あるいは、新しいラベル付けが行われなくなるまで繰り返す
5. もし出口にラベルが付いていれば、その値が最短経路長である。出口にラベルが付いていなければ、入口から出口に至る経路が無いことを意味している。

この問題の壁のデータの与え方が変わっている。その意図をくみ取ってみよう。
壁のデータを文字の 2 次元配列で読み込んだとする。



壁データ	迷路のマップ
0 行 1 列の 1	(0,0)-(0,1)の間の壁
1 行 0 列の 0	(0,0)-(1,0)の間は壁無
1 行 2 列の 1	(0,1)-(1,1)の間の壁
2 行 1 列の 0	(1,0)-(1,1)の間は壁無
3 行 0 列の 1	(1,0)-(2,0)の間の壁
3 行 2 列の 0	(1,1)-(2,1)の間は壁無
4 行 1 列の 1	(2,0)-(2,1)の間の壁

これより、迷路のマップで(a,b)と(c,d)が隣接しているときに、その間に壁があるかどうかは、壁データの方で a+c 行 b+d 列の値を見ることで分かる。（値が 1 なら壁あり、値が 0 なら壁無）

空白を含む文字列を入力するには、gets関数を用いれば良い[1]が、gets関数を使用したプログラムを実行すると「warning: this program uses gets(), which is unsafe」といったバッファーオーバーフローに関する警告がでる環境が多くなっている。そのため、getsの代わりにfgets関数を用いて標準入力ファイルであるstdinから文字列を入力するようにすれば良い。

scanf と fgets の両者を用いる場合注意が必要である。この問題の場合、最初の行の入力である整数 w と h を scanf で読みこみ、それに続く 2×h－1 行を文字列として fgets で読

[1] scanf で文字列を入力する場合、空白文字を入力することは出来ない。

み込むとすると、scanf の後で 1 回 getchar()を行う必要がある（プログラム例参照)。これを行わないと、scanf 直後の fgets では空行(改行のみの行)が読み込まれてしまう。

プログラム例１　簡易版幅優先探索

```
// ACM-ICPC 2010 Japan Online Contest Problem B
// http://icpc2010.honiden.nii.ac.jp/domestic-contest/problems#section_B
//    ファイル名: b_1.c
//    コンパイル方法: cc -O2 b_1.c
//    実行方法: ./a.out < B0 > B0.result など
//    チェック方法: diff B0.ans B0.result など
//
// アルゴリズム概要
// キューを用いない簡易版の幅優先探索 (n = w * h とすると O(n*n))
// 壁のデータは文字列のまま配列 wall に格納
// (x,y)と(x+1,y)の間に壁有 <==> wall[2*y][2*x+1]=='1'
// (x,y)と(x-1,y)の間に壁有 <==> wall[2*y][2*x-1]=='1'
// (x,y)と(x,y+1)の間に壁有 <==> wall[2*y+1][2*x] == '1'
// (x,y)と(x,y-1)の間に壁有 <==> wall[2*y-1][2*x] == '1'
// (x,y)の隣接位置を(nx,ny)とすると
// (x,y)と(nx,ny)の間に壁有 <==> wall[y+ny][x+nx] == '1'

#include <stdio.h>

#define MAXWH 30         // 幅、高さの最大値
int map[MAXWH][MAXWH];   // 経路探索用の地図。入口からの最短距離を格納
char wall[2*MAXWH-1][2*MAXWH+2]; // 壁のデータ

// 隣接位置のオフセット　右,上,左,下
int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, -1, 0, 1};

int w, h;        // 迷路の幅と高さ

void bfs()       // 簡易版の幅優先探索 (n=w*h とすると O(n*n))
{
  int x,y, len,flag,d;
  int nx, ny;
  for(y=0; y<h; y++)              // 探索用マップの初期化
    for(x=0; x<w; x++)
      map[y][x] = 0;              // まだ入口からの最短距離は求まっていない
  map[0][0] = 1;                  // 入口は左上

  for(len=1; ; len++){
    flag = 0;
    for(y=0; y<h; y++){
      for(x=0; x<w; x++){
        if(map[y][x] != len) continue;    // 最短距離が len 以外(未満)ならスキップ
        // 入口から(x,y)までの最短距離は len
        for(d=0; d<4; d++){       // (x,y)に隣接する４方向をチェック(右、上、左、下)
          nx = x + dx[d];// nx は隣接場所の x 座標
          ny = y + dy[d];// ny は隣接場所の y 座標
```

```
          if(nx<0 || nx >= w || ny<0 || ny>=h) continue;  // 迷路の外へはでない
          if(map[ny][nx] > 0) continue;   // 既に最短距離が求まっているのでスキップ
          if(wall[y+ny][x+nx]=='1') continue;      // 進行方向は壁なのでスキップ
          map[ny][nx] = len+1;    // 新しい場所に来たので、最短距離(len+1)を格納
          if(ny==h-1  && nx==w-1){         // 迷路の出口なので
            printf("%d¥n",len+1); // 求まった最短距離をプリント
            return;
          }
          flag = 1;       // 新たに最短距離が求まった場所がある。
        }
      }
    }
    if(flag==0) break;    // 新たに最短距離が求まった場所がなければ探索終了
  }
  printf("0¥n"); // 出口に到達できなかった。
}

int main()
{
  int i;

  while(1){
    scanf("%d %d", &w, &h);        // 幅と高さの入力
    if(w==0 && h==0) break;        // 両方共0なら終了
    getchar();                     // wとhの行の行末の改行コードを読み飛ばす
    for(i=0; i< 2*h-1; i++){       // 壁データの読み込み
      fgets(wall[i],2*MAXWH+1,stdin);      // 文字列として読み込む
    }
    bfs();        // 幅優先探索により最短距離を求める。
  }
}
```

プログラム例２　キューを用いた幅優先探索

```
// ACM-ICPC 2010 Japan Online Contest Problem B
// http://icpc2010.honiden.nii.ac.jp/domestic-contest/problems#section_B
//    ファイル名: b_2.c
//    コンパイル方法: cc -O2 b_2.c
//    実行方法: ./a.out < B0 > B0.result など
//    チェック方法: diff B0.ans B0.result など
//
// アルゴリズム概要
// キューを用いた幅優先探索
// 壁のデータは文字列のまま配列 wall に格納
// (x,y)と(x+1,y)の間に壁有 <==> wall[2*y][2*x+1]=='1'
// (x,y)と(x-1,y)の間に壁有 <==> wall[2*y][2*x-1]=='1'
// (x,y)と(x,y+1)の間に壁有 <==> wall[2*y+1][2*x] == '1'
// (x,y)と(x,y-1)の間に壁有 <==> wall[2*y-1][2*x] == '1'
// (x,y)の隣接位置を(nx,ny)とすると
// (x,y)と(nx,ny)の間に壁有 <==> wall[y+ny][x+nx] == '1'

#include <stdio.h>
#include <stdlib.h>
```

```
#define MAXWH 30 // 幅、高さの最大値
int map[MAXWH][MAXWH];   // 経路探索用の地図。入口からの最短距離を格納
char wall[2*MAXWH-1][2*MAXWH+2]; // 壁のデータ

// 隣接位置のオフセット　右,上,左,下
int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, -1, 0, 1};

int w, h;        // 迷路の幅と高さ

// 線形リストによりキューを実現
typedef struct q_data_type {
  int x;
  int y;
  struct q_data_type *next;
} qd;            // キューに格納する要素のデーター型
qd *q_front;     // キューの先頭要素へのポインタ
qd *q_rear;      // キューの最後要素へのポインタ

void q_init()    // キューを空にする
{
  q_front = q_rear = NULL;
}

void enqueue(int x, int y)        // キューにデータ(x,y)を入れる。
{
  qd *qpt;
  qpt = (qd *)malloc(sizeof(qd)); // キューの最後に入れる要素の領域を動的に確保
  qpt->x = x;                     // その要素に x をセット
  qpt->y = y;                     // その要素に y をセット
  qpt->next = NULL;               // 次の要素は無い
  if(q_rear == NULL)              // キューが空の場合は
    q_front = q_rear = qpt;       // q_front, q_rear ともにこの要素へのポイント
  else {                  // キューが空では無い場合は
    q_rear->next = qpt;           // キューの最後にその要素を追加し
    q_rear = qpt;                 // 追加した要素をキューの最後のデータとする
  }
}

void dequeue()                    // キューの先頭の要素を取り除く
{
  qd *qpt;
  if(q_front == NULL) return;     // キューが空なら何もしない
  qpt = q_front;          // 現在のキューの先頭要素へのポインタを qpt に退避
  q_front = q_front->next;        // 先頭の次の要素(即ち先頭から２番目の要素)を
                                  // キューの先頭とする
  if(q_front == NULL) q_rear = NULL;      // キューが空になったら、q_rear を NULL に戻
す
  free(qpt);              // 取り除いた要素が使用していたメモリ領域を返却する
}

void bfs()       // キューを用いた真面目な幅優先探索(n=w*h とすると O(n))
```

```
{
  int x,y, len,flag,d;
  int nx, ny;
  for(y=0; y<h; y++)              // 探索用マップの初期化
    for(x=0; x<w; x++)
      map[y][x] = 0;              // まだ入口からの最短距離は求まっていない
  map[0][0] = 1;          // 入口は左上
  enqueue(0,0);

  while(q_front){                 // キューが空でない間以下を繰り返す
    x = q_front->x;               // キューの先頭の要素に格納されていた x 座標
    y = q_front->y;               // キューの先頭の要素に格納されていた y 座標
    len = map[y][x];              // その場所の入口からの最短距離
    dequeue();                    // キューから先頭要素を取り除く
    for(d=0; d<4; d++){           // (x,y)に隣接する４方向をチェック(右、上、左、下)
      nx = x + dx[d];             // nx は隣接場所の x 座標
      ny = y + dy[d];             // ny は隣接場所の y 座標
      if(nx<0 || nx >= w || ny<0 || ny>=h) continue;  // 迷路の外へはでない
      if(map[ny][nx] > 0) continue;       // 既に最短距離が求まっているのでスキップ
      if(wall[y+ny][x+nx]=='1') continue; // 進行方向は壁なのでスキップ
      map[ny][nx] = len+1;        // 新しい場所に来たので、最短距離(len+1)を格納
      enqueue(nx, ny);            // その場所の情報をキューに格納
      if(ny==h-1  && nx==w-1){    // 迷路の出口なので
        printf("%d¥n",len+1);     // 求まった最短距離をプリント
        return;
      }
    }
  }
  printf("0¥n"); // 出口に到達できなかった。
}

int main()
{
  int i;

  while(1){
    scanf("%d %d", &w, &h);       // 幅と高さの入力
    if(w==0 && h==0) break;       // 両方共０なら終了
    getchar();                    // w と h の行の行末の改行コードを読み飛ばす
    for(i=0; i< 2*h-1; i++){      // 壁データの読み込み
      fgets(wall[i],2*MAXWH+1,stdin);     // 文字列として読み込む
    }
    q_init();             // キューを初期化
    bfs();                // 幅優先探索により最短距離を求める
    while(q_front)        // キューに残っている要素を全て取り出す
      dequeue();
  }
}
```