蜂の巣バケツで水を汲む

川合 慧（東京大学 総合文化研究科）
kawai@graco.c.u-tokyo.ac.jp

プログラミングの問題には，見るからに難しそうなもの，少し考えれば分かるような気がするもの，コツが分かればあっけなく解けるもの，など，いろいろなものがある．特にやっかいなのは，見かけはやさしそうであるが具体的なプログラムを作ってみるとやたらと複雑になる種類の問題である．この「見かけ」は，多くの場合人間の先入観によることが多く，その中でも図形的な要素が寄与していることが多い．今回はその種の問題を取り上げる．

■蜂の巣バケツ

問題は South Pacific Regionals 2001 の第 6 問，"Basalt Buckets" である．**図 -1** に，問題文に出ていた図を示す．図の平行四辺形状に並んでいる 6 角形は，実は 6 角柱を真上から見たもので，中に書いてある数字は 6 角柱の上面の高さである．また，図 -1 は高さ 0 の 6 角柱で周りを囲まれているものとする．

いま仮に，7 本の 6 角柱が「正 6 角形状」にまとめられていて，真中の角柱の高さが周りの 6 個の角柱の高さのどれよりも低いとすれば，そこに水を溜めることができる（**図 -2**）．水が溜められるのは 1 つの 6 角柱だけとは限らない．図 -1 の中央左に縦に並んでいる 2 個（高さ 5 と 4）については，その周りを囲んでいる角柱の高さがどれも 5 より大きいので，やはり水を溜めることができる．図 -1 の中央右の 3 個（高さ 1，2，4）についても同様である．このことが題名の Buckets の由来である．そして問題では，これらの平行四辺形状に並んだ角柱の高さ（非負の整数）の集合が与えられたとき，そこに溜められる水の量が求められている．

もう 1 つの単語 Basalt はもちろん玄武岩の意味である．火山から噴出した溶岩が比較的にゆっくりと冷却する場合，内部に大きさの揃った対流の渦が整然と並んで，結果として 6 角柱にまとまって固まることがある．柱状節理と呼ばれるこの現象は世界のあちらこちらで見られるが，北アイルランドの海岸にある Giant's Causeway が有名である．メートル単位の大きさの 6 角柱が広大なスペースに並んでいるのは，写真で見ても壮観である．この地形は 1986 年に世界遺産に登録されているらしい．この地形に海の波が押し寄せると，それが引いたあとのあちこちに水が取り残される．これがこの問題のモチーフである，と問題文には書いてある．柱状節理は日本でも各地にあり，兵庫県豊岡市の玄武洞公園が有名である．日光の華厳の滝の岩肌にも見られる．

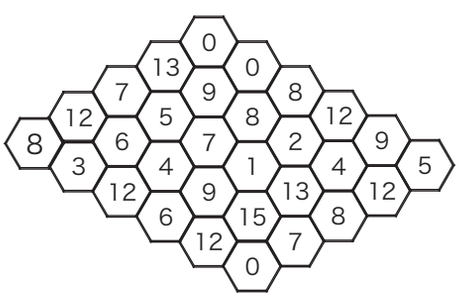

図 -1　問題文に示された例

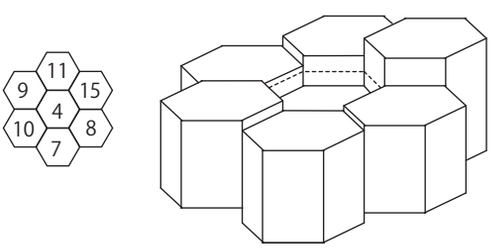

図 -2　バケツの構成
（左）高さの配置，（右）見取り図
中央に深さ 3（7-4）の水が溜まる

■問題の簡略化

プログラミングを知らない人には（多分）やさしそうに見えるであろうこの問題は，実は結構な量のプログラムを必要とする．ここではまず手慣らしとして，これを 1 次元に簡略化してみよう．問題は**図 -3** に示す「棒グラフのような図形」が与えられたとき，そこに「溜められる」水の量を求めることである．

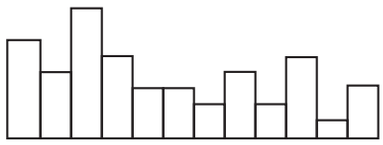

図-3　1次元問題

■湧出し法

もともとの状況では水は上からかかるのであるが，ちょっと考え方を変えて，角柱，ではなかった棒グラフの頂点から水が湧き出るものと考えよう．その水がこぼれずに溜まるためには，湧き出た場所より高い場所が左右両方に存在する必要がある．この「高い場所」は，一度低くなってから現れてもよい（**図 -4**（a））．かといってどんどん範囲を広げてゆくと，結局は元の場所には水が溜まらないこともあり得る（**図 -4**（b））．

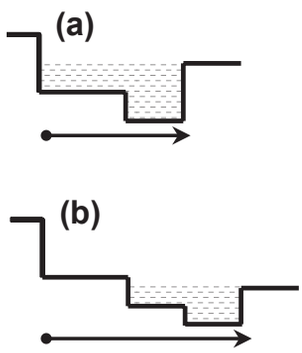

図-4　水が溜まる場所の探索

そこで処理を単純にするために，同じ高さのいくつかの棒が並んでいて，その両脇にそれより高い棒がある部分に着目する．これを基本バケツと呼ぼう．基本バケツが見つかったら，それを水で満たす．水の表面の高さは，両脇の高さの低い方である．そしてその水の表面を，新たな「底」とみなして処理を続ける．基本バケツが見つからなくなったら処理を終了する．

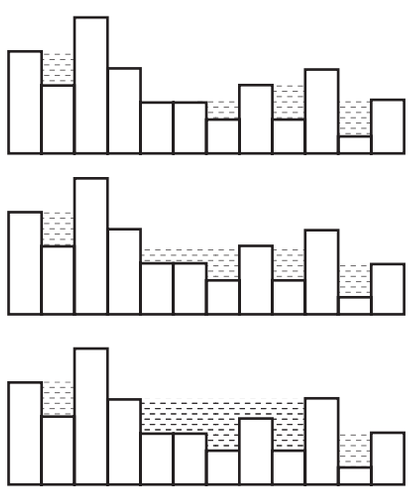

図-5　湧出し法による処理過程（1次元）

プログラムの例を示す．言語は Java であるが Java らしい機能は使っていない．データ入力は省いて，あらかじめ配列に書いてあるものとする．

```
class BB1a {
  static int[] h = {0,6,4,8,5,3,3,2,4,2,5,
                    1,3,0};
  static int quantity = 0;

  public static void main(String argv[ ]) {
    int i, j, k, w;
    boolean yet = true;

    while( yet ) {
      yet = false;
      i = 1;
      while( i < h.length-1 ) {
                        // 左から走査
        j = i+1;
        if( h[i-1] > h[i] ) {
                        // 左壁を検出
          while( h[j] == h[i] ) j++;
                        // 底を右へ
            if( h[j] > h[i] ) {
                        // 右壁を検出
              yet = true;
                        // バケツ検出
              if( h[i-1] < h[j] )
                        // 水面の高さ
                w = h[i-1];
              else
                w = h[j];
              quantity +=  (w-h[i])*(j-i);
                        // 水量
              for(k = i; k<j; k++)
                h[k] = w;  // 水で満たす
            }
          }
          i = j;      // さらに右へ
        }
      }
      System.out.println("Quantity  =  "
                         +quantity);
    }
}
```

例とした「地形」についての処理過程を**図 -5** に示す．

すぐ分かる通り，このプログラムでは同じ場所を何回も処理する．一番ひどそうな状況は**図 -6** のようなもので，深さ 1 の水を何回も重ねてゆくことになる．

■洪水法

湧出し法はいかにも能率が悪いと感じられる方は，すぐに「上から順に満たしてゆく」方法を思いつかれるであろう．まず全世界を大洪水よろしく水浸しにしておいて，だんだんと水面を下げてゆく．そのうちに，最も高い場所が島として現れ，次に現れた島との間に水が満たされる．

ここで，だんだんと下がってゆく水面には，高い場所から順に棒の頭が見えてくるはずであるから，最初に高さでソートして，最高の壁と 2 番目の壁の間，2 番目の壁と 3 番目の壁の間，という具合いに水を溜めてゆけばよい，であろうか．実はこのやり方は

(A) 水没した壁は以後考慮してはいけない（**図 -7**(a)）
(B) より高い壁をまたいでは水を溜められない（**図 -7** (b)）

という理由によりうまくゆかない．

最も高い壁 (P_1) とその次に高い壁 (P_2) の間に水を溜めた状態を考えよう（**図 -8**）．すると理由 (B) により，これ以降は左右別々に処理することが必要となる．今，図 -8 のように P_1 の方が左側にあるとすると，P_1 より左側と P_2 より右側は，独立に処理することになる．P_1 より左側の処理は，その範囲で最も高い壁 (P_3) を見つけ，P_1 と P_3 の間に水を溜める．さらに P_3 より左側の範囲での最も高い壁 (P_4) を見つけ，P_3 と P_4 の間に水を溜める．これを繰り返してゆく．P_2 より右側も同様である．

ふたたびプログラムの例を示す．一度左（右）を処理し出したら，あとは左へ左へ（右へ右へ）と同じ処理を繰り返してゆく．プログラムでは 2 つの再帰メソッドを使っているが，反復への変換，および 2 つのメソッドの統合は比較的やさしい．max は単なる最大値を求めるメソッドである．

```
class BB1b {
  static int[] h = {0,6,4,8,5,3,3,2,4,2,5,
                    1,3,0};
  static int quantity = 0;

  public static void main(String argv[ ]) {
    int mm = max(1, h.length-2);
                      // 全体の最大値
    fillleft(0, mm);
                      // 左側の処理
    fillright(mm,h.length-1);
                      // 右側の処理
    System.out.println("Quantity  =  "
                       +quantity);
  }

  static void fillleft(int left, int right)
{
    int i, nextmax;    // 左側の処理

    if( left < right-1 ) {
      nextmax = max(left,right-1);
                      // 左側の最大値
      for(i = nextmax+1; i<right; i++)
        quantity +=  h[nextmax]-h[i];
                      // 水量計算
      fillleft(left, nextmax);
                      // さらに左へ
    }
  }

  static void fillright(int left,
                        int right) {
    int i, nextmax;    // 右側の処理

    if( left < right-1 ) {
      nextmax = max(left+1,right);
      for(i = left+1; i<nextmax; i++)
        quantity +=  h[nextmax]-h[i];
      fillright(nextmax, right);
    }
  }

  static int max(int low, int high) {
```

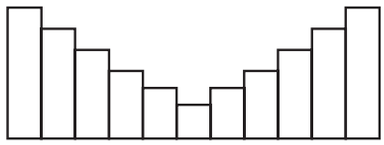

図-6　湧出し法が苦手な状況

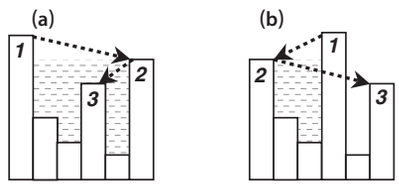

図-7　洪水法での壁の水没 (a) と水溜りの分離 (b)

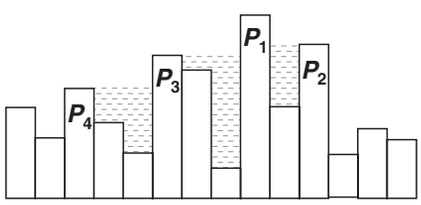

図-8　洪水法での処理の進行

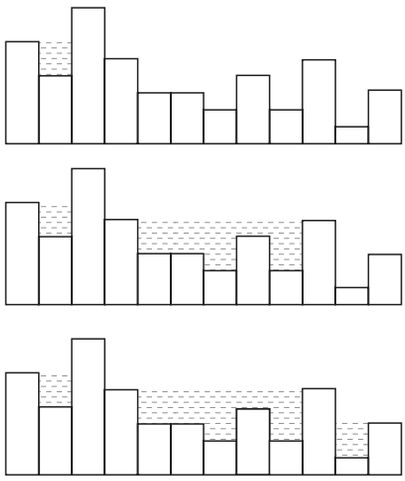

図-9　洪水法による処理過程（1次元）

```
        省略
  }
}
```

再び，例とした「地形」についての処理過程を**図-9**に示す．いちいち最大値を求め直しているのが非効率に見えるが，最初に全体をソートしておいても，前述の水没現象への対処と，対象としている範囲の中での最大値を特定するのが，案外手間のかかる処理となる．もちろん，棒の数が多くなってきた場合には，考慮すべき点である．

■いざ2次元へ

手慣らしが終わったところで，いよいよ本問題にとりかかろう．1次元で考えていた水を溜める2個の壁は，2次元では環状になった6角柱の連なりに相当する．原文ではwell（井戸）と呼んでいる．この「連なった」から「隣接関係」や「内外判定」を想起し，「込み入っていそうだ」と感じられれば，若干のプログラミングセンスを自認してよい．

利用するデータ構造としては，第一感として2次元配列があろう．**図-10**のように座標系を設定して，そのx座標とy座標とで各6角柱を指定する．なお，以後6角柱を「セル」と呼ぶことにする．

1つのセルに必要なデータを検討しよう．まず，そのセルのxおよびy座標値がある．(x, y) からセルが指定できるので原理的には不要であるが，あるほうがプログラム的には楽な場合が多い．セル（6角柱）の高さも必要である．さらに，以下のような情報を各セルに置いておくことにする．

- 壁ID…そのセルが属する壁のID．属さなければゼロ．
- 井戸情報…そのセルの「井戸度」を示す情報．
- 周囲情報…周囲のセルの高さの最小値，など．

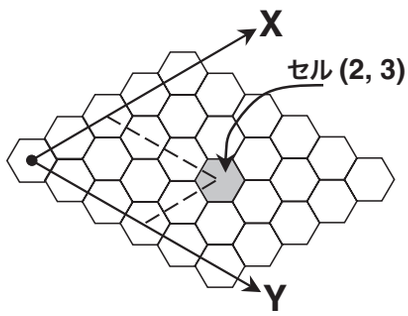

図-10　6角柱の座標系

要点は，必要なデータはなるべく各セルに書いておいて，ローカルな処理をやりやすくすることである．

```
class cell {
  int x,y,h;
  int well, low, wall, done;
  public cell(int xx, int yy, int hh) {
    x=xx; y=yy; h=hh;
    well=0; done=0;
  }
}
```

以下のコードでは

```
static cell[][] net
       = new cell[xsize][ysize];
```

と定義している．ここで，最も外側にあるとされる高さゼロのセルも，このデータnetには含ませている．

また，隣接する6個のセルの相対位置を，反時計回りに表すデータも用意する．

```
static int[][] nei = {{0,1},{1,0},{1,-1},
                      {0,-1},{-1,0},{-1,1}};
```

この方向データを用いて，「あるセルのある方向の隣接セル」を求めるメソッドを作っておく．

```
static cell neib(cell here, int dir) {
    return net[here.x + nei[dir%6][0]]
              [here.y + nei[dir%6][1]];
  }
```

6以上の方向についても処理できるように，%6（6で割った余り）を使用している．

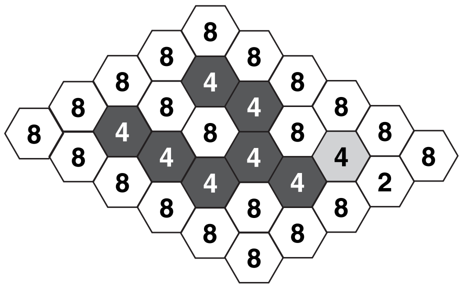

図-11　バケツの崩壊
高さ4のセルの連なりは，最も右のセルが候補では
ないために，全体が崩壊する

■まず湧出し法

1 次元の手順に沿って見てゆこう．基本バケツ（底が一定の高さで両脇が高くなっている場所）に相当するのは，「相互に隣接し高さが等しいセルの集まりで，その高さ以上のセルに囲まれているもの」である．これも基本バケツと呼ぼう．

基本バケツが見つかったら，それを水で満たす．水の表面の高さは，周りのセルの一番低い（しかし基本バケツの底よりは高い）高さである．水の表面を新たな「底」とみなして処理を続け，基本バケツが見つからなくなったら終了することは，1 次元の場合と同様である．

基本バケツをいきなり探しにゆくのは，かなり難しい．大きさ 1 の基本バケツであれば，「周囲のすべてが自分より高い」で見つけられるが，大きさ 2 以上の基本バケツを扱うには，「自分と同じ高さの隣接セル」まで考慮する必要がある．この処理は，隣接する同じ高さのセルすべてについて繰り返さなければならないが，どこかで「壁が崩れて」いると，辿ってきたセルばかりではなく，これから辿るはずであったすべてのセルもバケツではなくなる（**図 -11**）．

ここでは，まず基本バケツの要素となり得るセルを全部調べておいてから，基本バケツの検出を行うことにする．処理は以下の 5 段階に分かれる．

1）すべてのセルについて，自分の周囲の高さが自分の高さ以上であれば，「基本バケツの候補」の印をつける．ここで，図 -1 の最外周のセルは，さらにその外側の高さ 0 のセルに接しているので，基本バケツの一部にはなり得ないことに注意しよう．

```
static void findwellunit() {
  int x, y;
  cell here;
  for(x = 2; x<xsize-2; x++)
                      // 最外周は除く
    for(y = 2; y<ysize-2; y++) {
      here = net[x][y];
      if( lowest(here) >= here.h )
        here.well = candidate;
        //candidate は基本バケツの候補を示す定数
      else
        here.well = 0;
    }
}
```

ここでメソッド lowest は，周囲の高さの最小値を返すが，同時に「自分より高い値の最小値」も求め，here.low に書き込んでおく．この値は，バケツであると分かったときに溜められる水の量を求めるのに用いる．

2）すべてのセルについて，「基本バケツの候補」であり，かつ「まだ基本バケツとして登録されていない」ならば，「基本バケツチェック」を行う．このチェックが OK なら，「基本バケツ設定」を行う．

```
static void wellcheck() {
  int x, y;
  cell here;
  int donevalue = donemark;
                      // バケツ ID を初期化
  for(y = 2; y<ysize-2; y++)
    for(x = 2; x<xsize-2; x++) {
      here = net[x][y];
      if( (here.well == candidate) &&
          (here.done < donevalue)    ) {
        donevalue++;  // バケツ ID を更新
        wallheight = highvalue;
                  //wallheight（外部に定義）に
                  // 最小値を求めるための初期化
                  //highvalue はどのセルの高さ
                  // よりも大きな値
        if( wellcheck1(here,donevalue) )
          setiswell(here);
      }
    }
}
```

3）「基本バケツチェック」では，（再帰的に）周囲に同じ高さのセルで「基本バケツの候補」ではないものがないことを確かめる．なければ本物のバケツとなる．あれば（そこで壁が崩れているので）バケツにはならない．このチェックを行いながら，基本バケツ「かもしれない」セルの集まりの，周囲のセルの高さの最低値を求めておく．

```
static boolean wellcheck1(cell here,
                              int dval) {
```

```
  int dir;
  cell there;
  boolean truewell = true;

  here.done = dval;
  if( here.low < wallheight )
    wallheight = here.low;
                        // 最小値更新
  for(dir = 0; dir<6; dir++) {
                        // 周囲を調査
    if(truewell) {
      there = neib(here, dir);
      if( (there.h == here.h) &&
          (there.done < donemark))
        if( there.well  ==  candidate )
          truewell
            = wellcheck1( there, dval );
        else
          truewell = false;
    }
  }
  return truewell;
}
```

4)「基本バケツ設定」では，バケツに属するすべてのセルについて，well 欄に定数 iswell を，wall 欄に wallheight を，それぞれ書き込む．後者は処理の過程で求まった「周囲の高さの最低値」である．

```
static void setiswell(cell here) {
  int dir, hh = here.h;
  cell there;

  here.well = iswell;
  here.wall = wallheight;
  for(dir=0; dir<6; dir++) {
    there = neib(here, dir);
    if( (there.h == hh) &&
        (there.well != iswell) )
      setiswell(there);
  }
}
```

5）最後に，すべてのセルについて，バケツに属しているのなら，そこに溜められる水の量を累積する．

```
static boolean fillwell() {
  int x, y;
  cell here;

  boolean exist = false;
  for(x=2; x<xsize-2; x++)
    for(y=2; y<ysize-2; y++) {
      here = net[x][y];
      if( here.well == iswell ) {
        exist = true;
        quantity += here.wall-here.h;
                  // そこに溜まる量を追加
        here.h = here.wall;
```

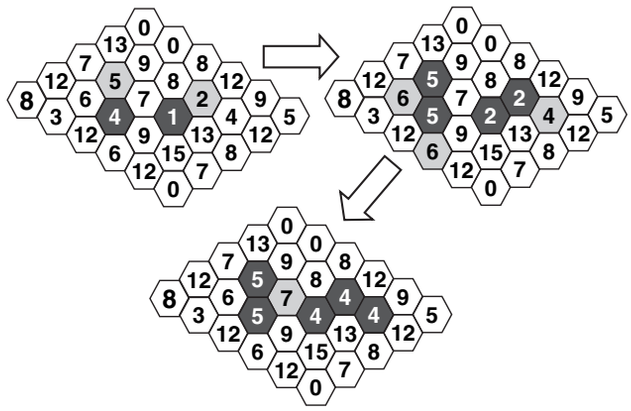

図-12　湧出し法による処理過程（2次元）

```
      }
    }
  return exist;
}
```

主プログラムは次のようになる．

```
public static void main(String argv[] ) {
  makenetwork(); // セルの配列の初期設定
  while(yet) {
    findwellunit();
    wellcheck();
    yet = fillwell();
  }
  System.out.println("Quantity = "
                     +quantity);
}
```

問題例についての処理過程を**図 -12** に示す．白抜き文字のセルは基本バケツ，影をつけたセルは基本バケツの周囲のセルで一番低いものを示す．

■洪水法

最後に 2 次元の洪水法をやってみよう．この方法の要点は，壁となるセルの連なりの構成法である．水面が下がってくると，高いセルから順番に水面に顔を出してくるが，やがてそれらが集まりだす．寄り添うように集まる場合は，単に壁が厚くなるだけである．独立にできてきた 2 つの壁が合流することもある．そして，同一の壁の違う部分が接続された瞬間が，壁の中に水が取り残される状況であり，ここで水が溜められる．

この方法のキーポイントは 1 つのセルの周囲の状況の把握である．水面に顔を出そうとしている 1 つのセルが，その周囲の 6 個のセルとどのような関係にあるかを計算する必要がある．周囲の状況に応じた場合分けは以下の通り複雑である．

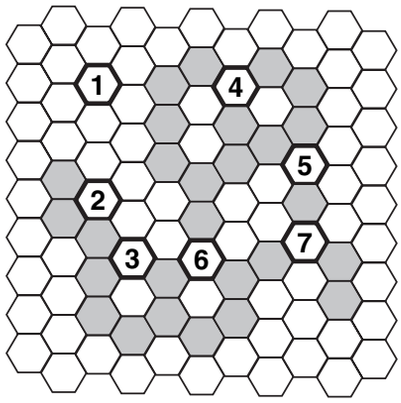

図-13　出現するセルの周囲状況

1）周囲にまだ「顔出しセル」がない
　⇒そのセルが新しい島となる
2）一塊となった壁がある
　⇒その壁が伸びる／厚くなる
3）2つに分かれてはいるが同じ壁の一部がある
　⇒その壁で井戸が1つできる
4）3つに分かれてはいるが同じ壁の一部がある
　⇒その壁で井戸が2つできる
5）2種類の壁が2つに分かれて存在する
　⇒その2種類の壁が合体する
6）2種類の壁が3つに分かれて存在する
　⇒壁が合体し，井戸が1つできる
7）3種類の壁が3つに分かれて存在する
　⇒3つの壁がすべて合体する

図-13 に上記の場合分けのそれぞれの例を示す．領域の単位が6角形であり，「周囲」が6個しかないので，場合分けは以上で尽くされる．また，「水面をだんだんと下げる」ように処理しているため，今考えているセルが既存の井戸の中にはない（あれば水没している）ことが保証されている．場合3）や4）に注意して欲しい．

この場合分けをやるための準備として，あるセルの周囲を調べ，何種類の壁があるか，と，それらがいくつの部分に分かれているかを求めるメソッドを用意する（makecell6．詳細は略）．洪水法の処理の性質として，すでにくっつき合っている複数のセルは同じ壁に属することが保証されている．

1つのセルは以下のようにしておく．

```
class cell {
  int x, y, h;
  int wallname;  // 壁ID
  int done;      // 水没処理用
  public cell(int xx, int yy, int hh) {
    x=xx; y=yy; h=hh;
    wallname=0; done=0;
  }
}
```

全体としては，セル全体をあらかじめ高さの順にソートしておいて，高いセルから順番に，上記の場合分けにしたがって処理してゆく．

```
for(col=0; col<celllist.length; col++){
  here = celllist[col];
          // 高さ順にソートされているものとする
  if( here.done == 0 && here.h > 0 ) {
    here.done = 1;
    makecell6(here, neibcell);
          // 周囲を調査し結果をneibcellに．
    topwall = neibcell.c[0].wallname;
          // 周囲の壁の1つ
    if( topwall>0 )
      attach(here, topwall);
          //既存の壁
    else
      attach(here, ++wallnameseq);
          //新しい島

    switch (neibcell.wallnum*10+
            neibcell.cluster     ) {
       // wallnum：壁の種類数（0～3）
       // cluster：集合している部分の数（0～3）
    case 00:// 新しい島ができる
             break;
    case 11:// 壁が伸びる／厚くなる
             break;
    case 12:// 井戸が1つ形成される
             processwell(here, 1);
             break;
    case 13:// 井戸が2つ形成される
             processwell(here, 2);
             break;
    case 22:// 2つの壁の合流
             joinwalls(here);
             break;
    case 23:// 壁が合流．井戸が1つ形成される
             joinwalls(here);
             processwell(here, 1);
             break;
    case 33:// 3つの壁の合流
             joinwalls(here);
             break;
    }
  }
}
```

ここでメソッドjoinwalls(cell here)は，セルhereから始めて，すでに他の壁とされているセルのwallnameに，here.wallnameを書き込む．これで壁が合流したことになる．

■内外判定

メソッド processwell は，現在のセル here に隣接する「壁でないセル」について，それが壁の内側であれば，そちらに溜められる水量を求める．ここで問題となるのは，壁のどちら側が内側であるかの決定である．図形が直線や曲線で構成されている場合には，調べたい点から「無限遠点」，あるいは外部と分かっている点に直線を引き，図形との交点の奇偶性を調べればよいが，直線が図形と接する場合には処理が厄介となる．現在取り組んでいる問題では，「図形」は 6 角形の集まりであり，頻繁に「接触状態」が出現してしまう．

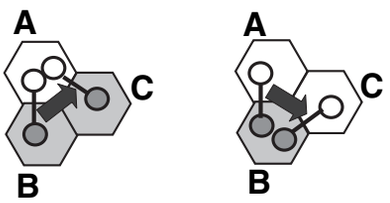

図-14　壁の辿り方
AとBに対してCを調べ，壁なら左のように，
壁でないなら右のように，それぞれ進む

ここでは「壁に触れながら一周する」方法を採用する．そのためには，調べるセル（壁ではない）と壁に属するセルのペアを作り，状況に応じてペアを更新してゆく（**図 -14**）．元のペアに戻るまでの「回転量」を求めれば，内側であるか外側であるかが決定できる．内側を一周するついでに，壁の高さの最小値が求められる．

```
static int isinside(cell p, int k) {
  cell emp=p,         // 調べるセル（図 -14 の A）
       wal=neib(p, k), // 壁のセル（図 -14 の B）
       nextcell;       // 次のセル（図 -14 の C）
  int rotation=0,      // ＋ 60 度ごとに＋ 1
      direction=k,     // 最初の進行方向
      lowest = highvalue;
                  // 最小値を求めるための初期化
  while( emp!=p || wal!=neib(p, k)
                  || lowest==highvalue ) {
    if( wal.h < lowest )
      lowest = wal.h;        // 最小値の更新
    nextcell=neib(emp, direction+1);
    if( iswall(nextcell) ) { // 左 60 度回転
      rotation++;
      wal=nextcell;
      direction = (direction+1)%6;
    } else {                 // 右 60 度回転
      rotation--;
      emp=nextcell;
      direction = (direction+5)%6;
    }
  }
```

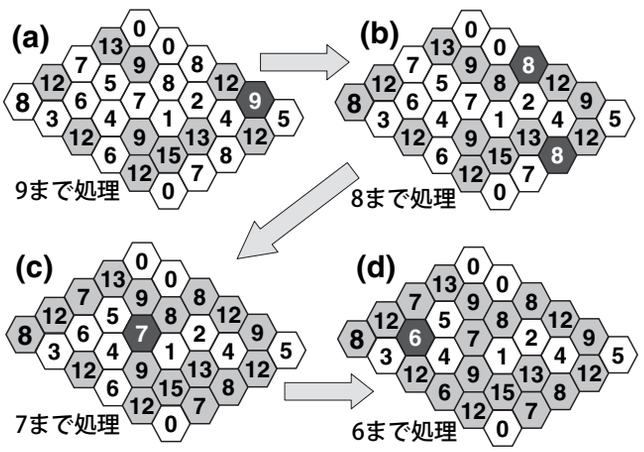

図-15　洪水法による処理過程（2次元）

```
  if(rotation == +6) // 反時計回りに 360 度回転
    return lowest;
  else
    return -1;
}
```

ここの処理では，1 次元の場合にはやらなかった水没現象への対処を行っている．

例とした問題での処理過程を**図 -15** に示す．(a) では 6 個の独立した壁のうち 2 つが右側の「9」によって合流する．(b) では「8」によりさらに合流が進み，壁は 3 個になっている．(c) は中央の「7」により井戸ができる瞬間である．(d) では左方の「6」によって，さらに井戸が作られる．

■取り組みの実際

元の問題とその簡略版のそれぞれについて，2 種類の方法によるプログラミングを紹介した．図形が絡む問題の「いやらしさ」の一端をお伝えできたかと思う．

実際のコンテストでは，参加が 87 チーム，この問題の解答の提出が 29，正答が 9 となっている．同一チームが何回提出したかは分からないが，細かな不具合が大量に発生するたぐいのプログラミングであり，何度も提出したチームが多かったものと推測される．また，微妙な状況の処理をチェックするためのテストデータの作りにくい問題でもある．

（平成 16 年 8 月 18 日受付）