問題1の解説

生徒数 n は無関係で，アンケートの数 m の大きさの配列を用意すればよい． 入力ファイルを読みながら，カウントしていく．ソートは，アンケートの集計をそのままソートするのではなく，その番号（インデックス）が出力に必要なの で，インデックスを保存する配列を別に用意して，間接的にソートする．回答数が同じ項目は，アンケートの番号順なので，安定ソートを用いるか，比較の条件 に入れてソートすることが重要である．m≦100 なので，**クイックソート** (quicksort) など使わなくても十分である．

安定ソート (stable sort) とは，「同じ値をもつデータのソート前の順序が，ソート後も変わらない」ようなソート・アルゴリズムの総称である． 例えばこの問題の場合には，アンケートの番号順に回答数を集計した配列を用意しておき，安定ソートによりソートすると，回答数が同じ項目はアンケートの番号順で並ぶようになる．

安定ソートの例としては，アルゴリズムが単純で実装が容易な**挿入ソート** (insertion sort) がある． 挿入ソートの基本的な考え方は，すでに並んでいるデータ a[0], ..., a[i−1] に対して，a[i] を左から順に比較していき，挿入場所（例えば、a[j] の直後に挿入すべきだったとする）がわかったら，その後ろの部分列 a[j + 1], a[j + 2], ..., a[i − 1] をずらして，空いた場所に a[i] を挿入する，というものである．

問題 2 の解説

与えられた文字列に問題の操作を 1 回施す方法だけを解説しよう. これを n 回繰り返せばよい.

まず, 文字列の左端の文字に注目し, 2 文字目以降を検査して行きながら, 現在注目している文字と同じときはカウントを増加させ, 異なるときは現在のカウントと注目している文字を新しい文字列に 書き加え, 注目している文字をそのとき読んだ文字に変更し, カウントを 1 に戻す. この作業を文字列の右端まで行う. あらかじめ文字列の右端に数字以外の文字, 例えば空白文字を **番兵** (centinel) として加えておけば, 文字列を左端から右端まで 1 度読むだけで操作は終了する.

与えられた文字列 st = "$s_0\ s_1\ \cdots\ s_L$" ($s_0, s_1, \cdots, s_L$ それぞれは 1 文字)に上述の操作を施す関数は次のようになる.

```
function 文字列の操作
begin
  st = st + " " ;   /* st の末尾に空白文字をつける. + は文字列の連結を表す */
  result = "" ;     /* result は操作を施した文字列(最初は空の文字列) */
  ch = s0 ;         /* ch は現在注目している文字 */
  count = 1 ;
  for i = 1 to L + 1
  begin
    if si = ch then
      count = count + 1 ;
    else
    begin
      result = result + toString (count) + toString (ch) ;
        /* toString は整数や文字を文字列に変換する関数 */
      ch = si+1 ;
      count = 1 ;
    end
  end
```

```
    return result ;
  end
```

最後に，プログラミング言語に依存することであるが，式

```
result = result + toString (count) + toString (ch)
```

を実現するには，C 言語では

```
sprintf(result, "%s%d%c", result, count, ch)
```

Java では

```
result = result + count + ch
```

とすればよい．

問題 3 の解説

この問題は **再帰** (recursion) を使って解いてくれることを期待して(再帰を理解しているかどうかを見るために)出題した.

問題文の絵を眺めていると, 直ぐに次のことに気付くであろう. 総数 n 個の正方形を指定されているような辞書式順序で, 一番左に m 個以下の正方形が積まれているように並べる並べ方を F(n, m) で表すことにすると, 次のような漸化式(再帰方程式)が得られる:

F(0, m) = 何もしない

F(n, m) = [m 個の正方形を縦に積む] F(n − m, m) ;

[m − 1 個の正方形を縦に積む] F(n − m + 1, m − 1) ;

・・・

[m − i 個の正方形を縦に積む] F(n − m + i, m − i) ;

・・・

[1 個の正方形を置く] F(n − 1, 1) ;

これをそのまま再帰的なプログラムとして書けばよい([ ] の部分では出力を行う).

このような 2 変数の再帰を思い付けば問題ないが, 以下では, n だけに注目した再帰しか思いつかなかった場合についても考えてみよう. 求める並び方を求める関数を f(n) とする. f(n) では, 上記で [ ] の部分を実行するのと同じ順序で, 正方形の並べ方を記録に残しておいて, f(0) のときにその記録を出力する(実は, この方法は本質的に上述の方法と同じものである). この "記録" を行うためには **スタック** (stack) を用いる. 下記のプログラムでは, スタックを配列 S で表し, i をプッシュすることを push(S, i) で, S のトップの要素をポップすることを pop(S) で表している:

```
関数 f(n)   /* 総計 n 個の正方形を積む */
begin
  if (n＞0) then
  begin
    for i = n downto 1 do
    begin
      if (i≦S[top]) then    /* S[top] はスタック S のトップ */
      begin
```

```
        push(S, i) ;   f(n − i) ;
          /* 直前に積んだ正方形の個数 S[top] 以下の正方形を，残り総数 n − i 個積む */
        pop(S) ;   /* S[top] を最左とする場合を終了したので S からポップする */
      end
    end
  end
  else
  begin
    if (S が空でない) then
    begin
      print(S) ;    /* スタック S の内容を逆順に(底からトップへ向かって)出力する */
    end
  end
end
```

上述の 2 つの再帰的アルゴリズムのいずれにおいても再帰には重複する計算がないので，実行時間は n に比例する時間しかかからない．あるPC上で実行してみた結果によると，解の個数と，それを求めるのにかかった時間は以下の通りであった(解の個数は漸化 式で求められる)：

n = 80　 5 秒　 15796497

n = 90　 2 分　 56634173

n = 100　8 分　190569292

ということは，実行にはさほど時間はかからないものの，すでに n = 80 で出力ファイルのサイズが 15 MB 以上になり，馬鹿でかいものとなってしまうので，出題では n＜30 とした．

問題 4 の解説

この問題は今のところ効率的に解く方法が知られていない有名な問題の1つである. したがって, すべての可能性をしらみつぶしに試していく方法を用いるしかない.

この問題の場合, 最も長い鎖を求めるために, **深さ優先探索** (DFS: depth-first search) と呼ばれる再帰的な方法で解くことが, 一般的な解法である. 深さ優先探索とは, リングから紐を順序良くたどって行くときに, 各リングにあるどれか1つの紐を選び, 先へ先へと行けるところまで行き, 行き止まりに達するかまたはすでにたどったリングに達したときには1つ手前のリングまで戻って別の紐を選んでいく, という方法である. このとき, 同じ経路を何度もたどらないように記録を残しておけば(すなわち, すでに通った紐に印を付けることにより, わかるようにしておけば), この方法で全体をくまなく探索することができる.

解を求めるのにかかる時間はばらつきが大きく一定しないが, 最も単純で記述しやすい方法であるため, 広く用いられている.

この深さ優先探索を実現する方法の1つに**再帰呼出し** (recursive call) がある. 再帰呼出しとは, 自分自身を呼び出すことである. しかしながら, 常に自分自身を呼び出すわけではない. そうでないと永遠に終了しなくなってしまう. そこで, 自分自身の呼び出しを終了するための条件を与えることが重要である. この問題の場合, あるリングからいくつかの紐がたどれるので, その先のリングについて探索する部分で自分自身を呼び出していくことになる. そして, 行き止まりに達するか, またはすでにたどったリングに達したときが自分自身の呼び出しを終了するときである.

再帰呼出しは一般的なデータ構造である**スタック**と関係が深い. 内部的には再帰呼出しはスタックを用いて実現されることが多いため, 自分のプログラム中でスタックを用いることで再帰呼出しを行わないようにすることもできる. これは高速化につながることもあるが, プログラムが複雑化することにもなり, どちらが良いかは問題の性質等による.

すでにこれまでの解説の中に何回もスタックが登場しているが, それはそれだけスタックが有用なデータ構造であるということである. 繰り返しになるが, スタックとは, 最後に入力したデータが先に出力される (LIFO: Last In First Out) というデータ構造である. 本を机の上に積み上げるような構造になっており, 本を積むときは一番上に積み(**プッシュ** (push) という), 本を取り出すときは一番上にある本から順次取り出していく(**ポップ** (pop) という)しかないようなものである.

深さ優先探索ではすべての可能性をしらみつぶしに試していくことになるが, 実際には, 探索の途中で引き返すことができるので, 純粋な総当りよりは効率が良くなる. ある解を求めるときに, 可能性のある手順を順に試していき, その手順では解が求められないと判明した時点で一つ前の状態に戻って別の手順を試す, という方法のことを**バックトラッキング** (backtracking) という(すでに別の解説でも述べた).

問題 5 の解説

この問題に対する単純なアルゴリズムは，平面が頂点の座標が整数であるような 1 辺の長さが 1 の正方形のタイルからなっていると考えるやり方である．長方形を1つ入力するごとに，その長方形に含まれる各タイルの位置に印を付け，すべての長方形に対 してそのような処理を終えてから，印の付いている正方形の個数を数えれば求める面積が得られる．さらに，こうして印が付けられたタイルそれぞれについて， その上下左右に隣接している 4 つのタイルに印が付いているか否かを調べることにより，出来上がった図形の周囲の長さも求めることができる． なぜなら，印が付いていないタイルに隣接する辺は，出来あがった図形の周上にあるので，そのような辺の本数を数えれば周の長さを知ることができるからであ る．しかし，この方法では多くの入力データでタイムオーバーになってしまう（そのように入力データを作成した）．

そこで，出来上がる図形 A を与えられたタイルを何枚か縦に張り合わせた x 軸方向の長さが 1 の縦長の長方形の集まりと考えてこの問題を解く方法を考えてみよう． 言い換えると，図形を間隔が 1 の y 軸に平行な直線で切り分けて考える． 以下，左下の座標が (a_x, a_y) で右上の座標が (b_x, b_y) である長方形を $(a_x, a_y)\times(b_x, b_y)$ と表し，図形 A に対して，直線 $x = x_0$ と $x = x_0 + 1$ に挟まれた部分を $A[x_0]$ と書くことにする．一般には，$A[x_0]$ は y 軸方向の長さが 1 の長方形ではなく，それらいくつかの和になっていることに注意しよう．

i 番目に入力された長方形を $S_i = (x_{i1}, y_{i1})\times(x_{i2}, y_{i2})$ とする（i = 1, ･･･, ,N）． $A = S_1 \cup S_2 \cup \cdots \cup S_N$ について（∪ は和を表す）

$$A[x_0] = S_1[x_0] \cup S_2[x_0] \cup \cdots \cup S_N[x_0]$$

が成り立つ．$x_{i1} \leqq x_0 < x_{i2}$ のとき $S_i[x_0] = (x_0, y_{i1})\times(x_0 + 1, y_{i2})$ であり，それ以外のとき空である．$A_i = S_1 \cup S_2 \cup \cdots \cup S_i$ に対して $A_i[x_0]$ が

$$A_i[x_0] = (x_0, a_1)\times(x_0 + 1, b_1) \cup (x_0, a_2)\times(x_0 + 1, b_2) \cup \cdots \cup (x_0, a_m)\times(x_0 + 1, b_m),$$
$$a_1 < b_1 < a_2 < b_2 < \cdots < a_m < b_m$$

と表されていれば，$A_{i+1}[x_0]$ を求めるのは多少ややこしいが，それほど難しくはない．例えば，$a_s < x_{(i+1)\,1} < b_s$ かつ $b_t < x_{(i+1)\,2} < a_{t+1}$ ならば

$$\begin{aligned} A_{i+1}[x_0] = \{ &(x_0, a_1)\times(x_0 + 1, b_1) \cup (x_0, a_2)\times(x_0 + 1, b_2) \cup \cdots \cup (x_0, a_{s-1})\times(x_0 + 1, b_{s-1}) \} \\ &\cup (x_0, a_s)\times(x_0 + 1, x_{(i+2)\,2}) \\ &\cup \{ (x_0, a_{t+1})\times(x_0 + 1, b_{t+1}) \cup (x_0, a_{t+2})\times(x_0 + 1, b_{t+2}) \cup \cdots \cup (x_0, a_m)\times(x_0 + 1, b_m) \} \end{aligned}$$

になる．したがって，必要な全ての座標について $A[x_0]$ を計算することは可能である．さらに， 入力 S_1, ･･･, S_N をあらかじめ都合の良い順序に並べなおしておけば，さらに効率良く $A[x_0]$ を計算することが出来る．

$$A_i[x_0] = (x_0, a_1)\times(x_0 + 1, b_1) \cup (x_0, a_2)\times(x_0 + 1, b_2) \cup \cdots \cup (x_0, a_m)\times(x_0 + 1, b_m)$$

の面積は

$(b_1 - a_1) \cup (b_2 - a_2) \cup \cdots \cup (b_m - a_m)$

である. こうして, $A[x_0] = A_N[x_0]$ から A の面積を求めることができる:

A の面積 =(A[0] の面積) ∪ (A[1] の面積) ∪ ･･･ ∪ (A[maxX −1] の面積)　（maxX は長方形の x 座標の最大値）

さらに $A[x_0]$ に含まれる A の周囲で x 軸に平行な辺の長さは 2k であり, y 軸に平行な辺は両隣り $A[x_0 - 1]$, $A[x_0 + 1]$ と共通の部分以外の辺であり, これも k に比例した時間で計算することができる.

さて, $A[x_0]$ をプログラム上で実装するためには **線形リスト** (linear list) を用いるとメモリ使用量に関しても実行時間に関しても効率が良い(A[0], A[1], ..., A[maxX − 1] それぞれを線形リストで表す必要があるので, それらへのポインタを要素とする配列を使う). 線形リストとは, 同じタイプの要素を並べた列

$\langle x_1, x_2, \ldots, x_n \rangle$　（各 x_i は同じ型のデータで, この線形リストの要素という）

のことであり, これらがポインタで

root → x_1 → x_2 → ･･･ → x_n　（root はリストの先頭を指すポインタ）

のようにリンクされているものである. したがって, x_i にアクセスするには $x_1, x_2, \ldots, x_i$ の順にポインタに沿ってたどるしかない. ただし, 線形リストを使うと, 必要なデータしか保持しないので, メモリが効率よく使えるだけでなく, データへの総 アクセス時間が結果的に短くなることが多い. この問題でも, 最初に述べた方法では時間内に答が計算できない(計算に何時間もかかる)ような入力データに対 しても, 後で述べた方法を線形リストを使って実装すると十分短い時間内で計算が終わるように入力データを作った. ただし, この方法でも, 特殊なデータに対 してはかなり実行時間がかかってしまう場合があることを付記しておく.