充電器が足りなくて

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

■問題

今回の問題は，2000 年 11 月につくばで行われたアジア地区予選の問題 D，“Pump up Batteries” である．問題文は以下から入手できる．

http://icpc.score.is.tsukuba.ac.jp/tsukuba-problems/d.html

まず問題の概要を説明しよう．ある人数（nguard とする）のガードが仕事についている．それぞれのガードはバッテリーで稼働する機器を携帯しており，ある時間それを使用すると，事務所に戻って充電しなければならない（最初に仕事を開始するときには充電完了となっている）．この仕事／充電のパターンはガードごとに定められていて，たとえば，

２ １ ３ ２

というパターンは，2 分仕事，1 分充電，3 分仕事，2 分充電という合計 8 分のサイクルを限りなく繰り返すことを示している．

具合の悪いことに充電器は 1 台しかないので，複数のガードが充電に戻ってきたときには，充電の順番を待つために到着順に列を作る．同じ時刻に複数のガードが列に並ぶ際には，識別番号の若いガードが先に並ぶことになっている（ガードには，0 から nguard-1 までの整数が識別番号として振られている）．以上をまとめると，ガードの状態には，仕事中，充電中，充電待ちの 3 つがあることになる．充電待ちをしているガードは充電器が空くのを待っているだけであり，時間を無駄に使っている．

この問題は，ガードの動作のシミュレーションを行うことによって，このような無駄な時間の総和を求めるものである．その際，入力データとしてはガードの人数，シミュレーション期間，それぞれのガードの仕事／充電パターンが与えられる．

入力データの形式は，最初の行にガードの人数とシミュレーション期間があり，そのあとにガードごとに仕事／充電のパターンが並んだものを 1 セットとし，これを何セットか繰り返して最後に 2 つの 0 を含む行で終了する．仕事／充電のパターンは偶数個の整数の並びで，最後に 0 が目印として付いている．

出力としては，入力データセットのそれぞれについて，無駄な時間の総和を 1 行に 1 つずつ印刷することが求められている．

具体例をあげよう．

```
3 25
3 1 2 1 4 1 0
1 1 0
2 1 3 2 0
```

このセットによるガードの振る舞いをシミュレートしてみると，次のようになる．

```
          0         10        20
          |    |    |    |    |    |
guard 0: ***.**.****.***.**-.****.
guard 1: *.*-.*-.*-.*.*.*.*--.*.*-
guard 2: **.***--..**-.***..**.***
```

横軸は時刻を示す．* はガードが仕事中であることを示し，. は充電中，- は充電待ちであることを示す．この図における - の総数が，求める「無駄時間」の総和であり，それは 10 となる．

なお，データの規模に関する制限は以下のとおりである．

ガード人数の上限 100
シミュレーション期間の上限 10080 分（1 週間）
仕事／充電パターンの長さの上限 50
仕事／充電パターンの個々の値の上限 1440 分

■入力データの保持

入力データをひとセット読み込んで保持するために，以下の変数と手続きを使う．

```
#define MAXDURATION 10080
#define MAXGUARD 100
#define MAXPATTERN 50

/* duty/charge のパターン */
int pattern[MAXGUARD][MAXPATTERN+1];
/* 各ガードの動作ステップ */
int patindex[MAXGUARD];

/* ガード i の動作に必要な時間 */
#define TIME(i) pattern[i][patindex[i]]

/* ガードの数 */
int nguard;
/* シミュレーションの終了時刻 */
int duration;
/* 累積の無駄時間 */
int idle;

/* データ読み込み */
void read_data(void)
{
  int i, j;

  scanf("%d", &nguard);
  scanf("%d", &duration);
  for(i=0; i<nguard; i++){
    for(j=0; ; j++){
      scanf("%d", &pattern[i][j]);
      if(pattern[i][j] == 0) break;
    }
    patindex[i] = 0;
  }
}
```

配列 patindex[NGUARD] は，各ガードが自分のパターンのどこを実行中かを示す添字を保持する．シミュレーションの開始時にはパターンの先頭から実行するから，初期値は 0 とする . また，パターンは充電と仕事が交互に現れるから，この値の偶奇を見ることで仕事中か充電中かを判定できる．

■時間駆動シミュレーション

この問題のように , 時間軸に沿ってシミュレーションを行うには，2 つの方式がある．1 つは時間駆動 (time-driven) 方式であり，もう 1 つは次章で取り上げるイベント駆動 (event-driven) 方式である．

時間駆動方式においては，システム全体の時刻が離散的な値をとることを仮定しており，その一刻みごとに各構成要素の振る舞いを計算していく．

単純で分かりやすい方式ではあるのだが，もちろん欠点もある．大多数の時刻には何も起こらず，ごくまれに何かの事象が起きるような問題では，不必要な計算を大量に行ってしまうのである．

この問題では，時刻は分を単位として離散化されており，その上限は 10080 と定められている．さらに構成要素たるガードは 100 人を上限とされているため，要素の考慮回数はそれらの積 (約 100 万) でおさえることができ，個々の考慮に要する時間が十分短ければ，この時間駆動方式で実用的な解を得ることができる．

各時刻における処理内容は以下のようになる：

- 仕事中のガードの各々に対して，仕事期間が終了したかをチェックし，終了していれば，充電のための待ち行列に移動する．その際，待ち行列が空であればただちに充電中に移行する．
- 充電中のガードに対して，充電の完了をチェックし，充電が完了していれば，仕事中へ移行する．さらに，充電待ちのガードがいればその先頭を充電中に移行する．

仕事中の各ガードと，充電中のガードは現在の作業完了時刻を配列 finish[NGUARD] に保持する．充電待ちのガードについては，この値は実際に充電に着手するときに設定することにした．また，現在時刻は大域変数 clock に保持する．

充電の待ち行列は，先入れ先出しの原則に従うので，リングバッファを用いたキュー構造とする．リングバッファへの 2 つの添字 head と tail がその挿入位置と取り出し位置を示す．ここで，リングバッファの容量を考える．キューに入るガードの数は，0 (全ガードが仕事中) から nguard (1 人が充電中で，残りはすべて充電待ち) までをとり得る可能性がある．リングバッファのサイズを nguard にしておくとこれらの両極端はどちらも head と tail が一致してしまうので，これらを判別するためには，リングバッファのサイズを 1 つ大きくしておくのがよい．

キューの操作としては，挿入 (enque)，先頭要素の検査 (peekque)，先頭要素の削除 (deque) を用意した．

以上をまとめた宣言は以下の通り：

```
int clock;

int finish[MAXGUARD];

int queue[MAXGUARD+1];
int head, tail;

void enque(int g)
{
```

```
  queue[head] = g;
  head++;
  if(head == MAXGUARD+1)
    head = 0;
}

void deque(void)
{
  tail++;
  if(tail == MAXGUARD+1)
    tail = 0;
}

int peekque(void)
{
  if(head == tail) return -1;
  else return queue[tail];
}
```

ここまで用意ができれば，あとは各時刻で行う処理を記述すればよい．

```
/* on duty のガードの処理
   終了予定時刻になっているものは全員，
   番号順に charging キューに移す
*/
void task1(void)
{
  int i;

  for(i=0; i<nguard; i++){
    if((patindex[i] & 1) == 0){ /* on duty */
      if(finish[i] <= clock){
        /* duty finished */
        enque(i);
        patindex[i]++;
        if(peekque() == i){
          /* no one waiting */
          finish[i] = clock + TIME(i);
        }
      }
    }
  }
}

/* charging キューの処理
   先頭のガードを on duty に戻し，
   duty 終了時刻を設定．
   2番目のガードがいれば，
   それの charging 終了時刻も設定
*/
void task2(void)
{
  int g, i;
  g = peekque();
  if(g >= 0){    /* g is charging */
    if(finish[g] <= clock){
      /* charge finished */
      deque();
      patindex[g]++;
      if(TIME(g) == 0){
        patindex[g] = 0;
      }
      finish[g] = clock + TIME(g);
      g = peekque();
      if(g >= 0){
        finish[g] = clock + TIME(g);
      }
    }
  }
  i = head - tail;
  if(i < 0) i += MAXGUARD+1;
  if(i > 0){
    idle += i-1;
  }
}
```

充電待ちのキューに対する処理の最後で，充電待ちのガードの数，つまり（キューの長さ -1）を idle に累計している．最後に，main 関数は以下の通り：

```
int main(int ac, char **av)
{
  int i;

  while(1){
    read_data();
    if(nguard == 0) break;
    idle = 0;
    head = 0;
    tail = 0;
    for(i=0; i<nguard; i++)
      finish[i] = pattern[i][0];

    /* 時刻を1ずつ進めてシミュレーション */
    for(clock=0; clock<duration; clock++){
      task1();
      task2();
    }
    printf("%d\n", idle);
  }
  return 0;
}
```

■イベント駆動シミュレーション

前章で紹介した時間駆動方式では，事象が起ころうと起こるまいと地道にシミュレーションを進めていった．これとは異なり，事象の時刻順リストを保持して，そのリストの各要素の時刻においてだけシミュレーションを行う方法がある．これがイベント駆動方式であり，時刻の離散化が密である場合（たとえばこの問題で時刻の単位が分でなくマイクロ秒であったケース）に計算時間を減らすことができる．

ここで計算量についてふれておく．シミュレーションの期間を n, ガードの人数を m とすると，時間駆動方式の計算量は $O(nm)$ となる．これに対して，イベ

ント駆動方式では，一般には，発生する事象の数は最悪の場合は nm となり，それをリストに挿入するコストは工夫をしたとしても $O(\log m)$ 必要であるから，全体としては $O(nm\log m)$ かかることになる．しかしこの問題についていえば，発生する事象の数はそれほど大きくならない．充電器が 1 台しかないことが幸いして，各時刻にたかだか 1 人しか仕事に戻ることができないのである．このことは事象総数が n でおさえられることを意味し，したがって全体の計算量は $O(n \log m)$ となり，時間駆動方式に対して優位となる．

ただし，この問題についていえば，前章で述べた通り全時刻のシミュレーションを行うことが可能なので，イベント駆動方式で行う意義は薄い（コンテストの場では，問題に与えられたデータ値の上限を見て，単純な時間駆動方式で十分解けると見抜くことが大切である）．しかし，シミュレーション記述方式の紹介という観点から，プログラムを紹介することにした．

まず，事象として何を取り上げるかだが，仕事の完了と充電の完了がそれにあたる．どちらも開始時刻が定まれば完了時刻も定まる．また，あるガードについてみると，仕事完了待ちと充電完了待ちは同時には起こり得ないから，事象のデータ構造をわざわざ作る必要はなく，ガードそのものを用いることができる．

事象のリストは 1 本にしておく方が先頭要素の取り出しなどに便利であるが，この問題の場合には充電完了事象に対する処理の一部に，充電待ちの先頭のガードを充電状態に移行させる処理が含まれるので，仕事中リストと充電リストの 2 つに分けることとした．

仕事中リストは仕事の完了時刻順となっており，新たなガードをリストに挿入する際には，その順序関係に従った場所に挿入される．なお，同時刻に完了するガードは，その識別番号順に並べておく．これは最初に述べた充電待ちの列での制約を実現するためである．

充電リストは到着順のリスト（つまりキュー）となる．先頭が充電中で，充電完了時刻を保持している．2 番目以降は充電待ちで，リストに加わった時刻を保持している．

仕事中リストは途中への挿入が必要となるので，単純なキューではなくプライオリティキューとなる．その実現には，計算量の問題がないという前述の議論を踏まえてプログラムの単純化の観点からリンクリストとする．1 人のガードはリストの片方のみに含まれていて 2 つのリストは排他的であるから，リンクのフィールドは共用できる．先頭を示す変数だけをそれぞれ別に用意すればよい．さらに，充電リストについては末尾を示す変数も用意する．

```
/* リンクリスト用のフィールド */
int next[MAXGUARD];
/* ガードが次に何かする予定時刻 */
int time[MAXGUARD];

/* on duty のガードのリスト先頭 */
int d_head;
/* charge 中のガードのリストの先頭と末尾 */
int c_head, c_tail;

/* 現在時刻 */
int clock;

#define NIL (-1)

/* i 番のガードを duty リストに挿入する
   挿入位置は，duty 終了予定時刻の順
   ただし，同時刻の場合には，識別番号の順
*/
void insert_d(int i){
  int *ip;
  time[i] = clock + TIME(i);
  ip = &d_head;
  while((*ip != NIL) &&
        ((time[*ip] < time[i]) ||
         ((time[*ip] == time[i])
          && (*ip < i)))){
    ip = &next[*ip];
  }
  next[i] = *ip;
  *ip = i;
}

/* i 番のガードをchargingリストの末尾に追加
   time フィールドには，
     先頭の場合には charge 終了予定時刻
     2番目以降の場合には リストへの追加時刻
     (つまり現在時刻)
*/
void insert_c(int i){
  patindex[i]++;
  next[i] = NIL;
  if(c_head == NIL){
    c_head = i;
    idle += (clock - time[i]);
    time[i] = clock + TIME(i);
  } else {
    time[i] = clock;
    next[c_tail] = i;
  }
  c_tail = i;
}
/* duty リストの処理
   先頭のガードを charging リストへ移動する
*/
void process_d(void){
  int n = next[d_head];
  insert_c(d_head);
  d_head = n;
}
```

```
/* charging リストの処理
   先頭のガードを duty リストへ移動する
   2番目のガードがいれば，その終了予定時刻を
   設定する
*/
void process_c(void){
  int n;
  patindex[c_head]++;
  if(TIME(c_head) == 0){
    patindex[c_head] = 0;
  }
  n = next[c_head];
  insert_d(c_head);
  if(n != NIL){
    idle += (clock - time[n]);
    time[n] = clock + TIME(n);
  }
  c_head = n;
}

#define INFINITY 99999999
#define MIN(x,y) (((x)>(y))?(y):(x))

int headtime(int head)
{
  if(head == NIL) return INFINITY;
  else return time[head];
}

/* 主要な処理のループ */
void task1(void)
{
  int c_time, d_time;
  while(1){
    /* 両リストの先頭の終了予定時刻を取得 */
    c_time = headtime(c_head);
    d_time = headtime(d_head);
    /* それらのうちの早い時刻を選び */
    clock = MIN(c_time, d_time);
    /* シミュレーション期間を外れたら終了 */
    if(clock > duration) break;
    /* それぞれのリストの処理 */
    if(clock == c_time) process_c();
    if(clock == d_time) process_d();
  }
}

int main(int ac, char **av)
{
  int i;

  while(1){
    read_data();
    if(nguard == 0) break;
    idle = 0;
    clock = 0;
    d_head = NIL;
    c_head = NIL;
    /* 全ガードを duty リストに登録 */
    for(i = 0; i<nguard; i++){
      insert_d(i);
    }
    /* 主処理 */
    task1();
    /* シミュレーションが終了した時点で
       charging リストに残っているガードが
       それまでに費やした待ち時間も累積する */
    if(c_head != NIL){
      for(i=next[c_head];i!=NIL;
          i=next[i]){
        idle += duration - time[i];
      }
    }
    printf("%d\n", idle);
  }
  return 0;
}
```

イベント駆動方式では時刻がとびとびに進むので，シミュレーションの終了のタイミングとしては，与えられたシミュレーション期間 duration 内の事象だけを扱わなければならない．

特に，最後の事象が終了時刻より早い場合には注意が必要である．その間には特別な事象が起きないが，この問題で求められている「充電待ちの時間」はその間についても考慮しなければならないのである．上のプログラムでは，main の最後で，充電リストに残っているガードについてこの追加処理を行っている．

問題文に含まれている 2 つの例においてはこのような状況はなく，シミュレーション終了のタイミングにちょうど事象が起き，さらに充電リストにはガードが残らないようになっているが，審判団の用意した入力データではその点もしっかりチェックするようになっており，問題が比較的簡単なわりにはやや意地の悪いところをのぞかせている．

（平成 16 年 1 月 13 日受付）