ケーブルマスタ

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

■問題

今回取り上げるのは，2001 年 11 月にロシアのサンクトペテルブルクで開催された Northeastern Europe Programming Contest の問題 C, "Cable master" である．問題は以下より入手可能である：http://icpc.baylor.edu/past/icpc2002/regionals/NEERC01/problems.html

真に公平なプログラミングコンテストを実施するために，各参加チームのコンピュータとネットワークハブとを等しい長さのケーブルで接続したい，と問題文は始まる．参加チームの間にはできるだけ距離をおきたいので，ケーブルは長いほどよい．発注を受けた業者は，手持ちのケーブル在庫の集まりから，等しい長さのできるだけ長いケーブルを，指定された本数だけ切り出さなくてはならない．納入すべき本数，在庫のケーブルの本数とそれぞれの長さが与えられたとして，上の条件を満たすケーブルの最大長を答えるのが問題である．

入力データは，最初の行に在庫ケーブルの本数 N と納入すべきケーブルの本数 K があり，そのあと在庫ケーブルそれぞれの長さが 1 行ずつ続く．

長さはセンチメートルを精度として，メートルで表現されている（小数点以下 2 桁が必ずつく）．

出力は納入するケーブルの最大長をメートル単位，小数点以下 2 桁で答える．1cm 以上のケーブルを納入できなければ失敗として 0.00 を出力する．

入力データに課せられた制約は以下のとおり：

- 在庫ケーブルの本数について $1 \leq N \leq 10000$
- 納入ケーブルの本数について $1 \leq K \leq 10000$
- 在庫ケーブルの長さは 1m 以上，100km 以下

問題文にあげられた例は以下のとおり：

入力：

```
4 11
8.02
7.43
4.57
5.39
```

出力：

```
2.00
```

この例では，4 本の在庫から 11 本のケーブルを納品することになるが，それぞれの在庫からのケーブルの切り出し数は順に 4, 3, 2, 2 とするのが最善である．

■入力データの保持

プログラムの中では，ケーブルの長さはセンチメートルを単位とする整数で表現する．ケーブルの最大長は 10^7cm であるから，32 ビットの整数で十分表現可能である．

各種の定数の定義と入力データを保持するコードは以下のとおりである：

```
#define MAXN 10000       /* 在庫の最大本数 */
#define MAXK 10000       /* 納入する最大本数 */
#define MINLEN 100       /* 在庫の最小長 */
#define MAXLEN 10000000 /* 在庫の最大長 */

int N;            /* 在庫の本数 */
int K;            /* 納入する本数 */
int stock[MAXN]; /* 各在庫の長さ */
```

データ読み込みの手続きは以下のとおり（本来はデータ形式や値の範囲のチェックなどが必要であるが，ここでは省略してある）．

```
void read_data(void)
{
  int i;
  double d;

  scanf("%d%d", &N, &K);
  for(i=0; i<N; i++){
    scanf("%lf", &d);
    /* 整数値に丸める */
    stock[i] = lrint(d * 100);
  }
}
```

main 関数は以下のとおり．ここで，task は最大長を求める関数であり，これからこの関数を作成する．

```
int main(void)
{
  int task(void);
  read_data();
  printf("%.2f\n", task()/100.0);
  return 0;
}
```

なお，2 つの整数の最大値と最小値を求める関数 MAX, MIN もしかるべく定義してあるとする．

■動的計画法

まず，最も単純に，可能な組合せをすべて試みることを考えよう．在庫ケーブルそれぞれから切り出すケーブルの本数を制御することになる．本当に何も考えなければ，在庫ケーブルそれぞれから 0 ～ K 本を切り出す可能性があるので $O((K+1)^N)$ の計算量となる．合計が K 本という条件があるので，そこまでのことはないが，いずれにせよ計算するには無理な時間が必要である．

この条件のもとですべての組合せを試みるには，以下のように再帰的に考えればよい．

> 0 番から s 番までの在庫を使って p 本のケーブルを作るときの最大可能長は以下の $(p+1)$ 通りのうち最大のものとなる：
> 0 番から $(s-1)$ 番までの在庫から i 本切り出し，s 番から残りの $(p-i)$ 本を切り出す
> (ここで $0 \le i \le p$)

このことをプログラムとして表現すると以下のようになる．なお，以下のコードでは在庫ケーブルの番号は 0 から $N-1$ まで，切り出す本数は 1 から K までとする．

```
/* 0番からs番までの在庫を使って
   p本のケーブルを作るときの
   最大可能長さを求める */
int exh(int s, int p)
{
  /* (1) s番だけで全部まかなう */
  int max = stock[s]/p;
  if(s > 0){
    int i;
    for(i=1; i<p; i++){
      /* (2) s-1番以前の在庫からi本とり，
         s番で残りの(p-i)本をまかなう */
      max = MAX(max,
                MIN(exh(s-1, i),
                    stock[s]/(p-i)));
    }
    /* (3) s番は一切使わない */
    max = MAX(max, exh(s-1, p));
  }
  return max;
}
```

さて，このプログラムは，exh(s,p) を s の小さい方から順次求めて表に格納していくことで，同一の引数の再帰呼び出しを一度だけで済ませることができる．おなじみとなった動的計画法である．

コードは以下の通り：

```
/* longest[s][p] は
   0番からs番までのストックを使って，
   p本のケーブルを作るときの
   最大可能長さ */
int longest[MAXN][MAXK+1];

/* 0番からs番までの在庫を使って
   p本のケーブルを作るときの
   最大可能長さを求める */
int one_step(int s, int p)
{
  int max = stock[s] / p;
  if(s > 0){
    int i;
    for(i=1; i<p; i++){
      max = MAX(max,
                MIN(longest[s-1][i],
                    stock[s]/(p-i)));
    }
    max = MAX(max, longest[s-1][p]);
  }
  return max;
}

int task(void)
{
  int s, p;

  for(s=0; s<N; s++){
    for(p=1; p<=K; p++){
      longest[s][p] = one_step(s, p);
    }
```

```
  }
  return longest[N-1][K];
}
```

計算量を考察しよう．まずメモリについていえば，表 longest が $N \times K$ のオーダで必要になり，問題の制限条件から，100 メガエントリが上限となる．今日のコンピュータでもやや大き過ぎるが，幸いなことにこれを大幅に縮小する方法がある．

longest[s][p] を計算するときには longest[s-1][*] だけが必要であって，s-2 以前は使わない．したがって，この配列の第 1 添字の上限を MAXN ではなく 2 として，交互にそれを使うようにすればよい．コードでいえば，配列参照の際に longest[s][p] のかわりに longest[s % 2][p] とするだけである．これで，空間計算量は $O(K)$ に下げることができた．

一方，時間計算量の方は芳しくない．表のエントリを 1 つ埋めるために $O(K)$ の手間がかかるので，全体として $O(N K^2)$ の計算量となってしまう．問題の制限条件に照らすと 1 テラのオーダとなり，これは今日の計算機でも手に余る時間である．

というわけで，いつものように動的計画法で正解かと思いきや，そうはならなかった．

■二分探索

ここで考え方を変えて，切り出すケーブルの長さを与えたときに，何本とれるかを考察しよう（*available*(*len*) と表記しよう）．その計算は簡単で，すべての在庫ケーブルの長さを *len* で割って，整数の商を合計するだけである．

関数 *available* が，引数に対して単調に減少する（正確には非増加）ことは容易に見てとれる．また，引数となる，欲しいケーブルの長さの上限は在庫ケーブルの最大長を超えることはない．

これらを考え合わせると，切り出すケーブルの長さについて二分探索をすればよいことが分かる．

```
/* 長さlenのケーブルが全部で何本とれるか */
int available(int len)
{
  int i;
  int r = 0;
  for(i=0; i<N; i++){
    r += stock[i] / len;
  }
  return r;
}
```

```
int task(void)
{
  int hi, lo, mid;

  hi = MAXLEN+1;
  lo = 0;

  while(hi > lo+1){
    mid = (hi+lo)/2;
    if(available(mid) < K){
      hi = mid;
    } else {
      lo = mid;
    }
  }

  return lo;
}
```

関数 available の計算は $O(N)$ で済むので，この方法は解の大きさを E としたとき $O(N \log E)$ ということになる．今回の解の精度は，ケーブルの最大長が 10^7cm であったから，$\log_2 10^7 < 24$ となり，繰り返し回数はたかだか 24 回で済む．

少し注意が必要なのは，整数について二分法を適用するときの終了条件の選び方である．区間の上端と下端のいずれもが解となり得るように区間を選ぶ──言い替えると対象とする区間を両側とも閉区間（つまり $[lo, hi]$）とする──と，終了条件は lo=hi とせざるを得ない．しかし，lo と hi が 1 だけ違った場合に中点 m は除算の切り捨てによって lo と等しくなるため，それ以上区間が小さくならず停止しなくなってしまう．

区間の上端を解の候補としない──大きい方を開区間とする（つまり $[lo, hi)$）──と，この問題は解消する．その場合の終了条件は $hi = lo+1$ となり，これなら必ず停止する．

■第 3 の方式

最後に，これまでとはまったく異なる解法を紹介する．これは本連載の担当者の 1 人である石畑氏が提案した方法である．

前述の動的計画法が 1 本ずつ在庫ケーブルを検討対象に加えていくアプローチだったのに対して，こちらは最初からすべての在庫ケーブルを対象としておいて，そのかわり切り出すケーブルの本数を 1 本から順次増やしていくのである．各在庫ごとに「現在までにそのケーブルから何本切り出したか」を保持しておき，それを順次更新することにする．

切り出すべきケーブルが1本であれば，当然最も長い在庫をそのまま使えばよい．2本切り出すには，最長の在庫から2本切り出すか，あるいは最長と次点の在庫から1本ずつ切り出すかのどちらかとなる．

一般的には，k本切り出すための最適な方法が分かっているとき，もう1本切り出すには，各在庫について，「現在の切り出し本数に加えてもう1本余計に切り出すとしたときに得られる長さ」を計算して，それが最長になるような在庫から切り出すことにする．こうして，以下のプログラムを得る．

```
/* ある在庫からの現在の切り出し本数 */
int npiece[MAXN];

int task(void)
{
  int p, s, imax;
  int max;
  /* 切り出し本数についてのループ */
  for(p=1; p<=K; p++){
    max = 0;
    /* 吟味する在庫についてのループ */
    for(s=0; s<N; s++){
      if(stock[s]/(npiece[s]+1) > max){
        max = stock[s]/(npiece[s]+1);
        imax = s;
      }
    }
    npiece[imax]++;
  }
  return max;
}
```

このプログラムの時間計算量は$O(NK)$であり，この問題の解法としては十分である．ただ，今のままだと，計算量は二分探索方式の方が小さく，$K = N = 10000$のとき，24万対1億くらいである．しかし，この方式の計算量をさらに下げることも可能である．

N本の在庫から次のベストを選ぶ際に，ヒープのデータ構造を用いると全体として$O(K \log N)$で済むので，二分探索方式との勝負は24万対14万と接戦に持ち込むことができる（定数倍の差はあるにせよ）．

今回は紙数に余裕があるので，ヒープによるコードも示しておこう．ヒープには在庫の番号が収めてあり，その条件は根に近いものほど「もう1本切り出した場合に長くとれる」ようにしてある．ヒープの根から1つ在庫を選んではそれから1本切り出す，という処理をK回繰り返す．

```
/* ある在庫からの現在の切り出し本数 */
int npiece[MAXN];
/* ヒープ用の領域 */
int heap[MAXN];

/* 在庫 n からもう1本切り出したときの長さ */
int next_len(int n)
{
  return stock[n]/(npiece[n]+1);
}

/* 在庫 ni のほうが nj より長くとれる? */
int gt(int ni, int nj){
  return (next_len(ni) > next_len(nj));
}

/* ヒープ内での交換 */
void exchange(int hi, int hj){
  int tmp;
  tmp = heap[hi];
  heap[hi] = heap[hj];
  heap[hj] = tmp;
}

/* ヒープ hi番から小さい要素を降ろす */
void down(int hi){
  int hl = hi*2+1;
  int hr = hl+1;
  int larger;     /* 左右の子の大きい方 */

  if(hl < N){
    if((hr < N) &&
       gt(heap[hr], heap[hl]))
      larger = hr;
    else larger = hl;
    if(gt(heap[larger], heap[hi])){
      exchange(hi, larger);
      down(larger);
    }
  }
}

void init_heap(void)
{
  int n;
  for (n=N-1; n>=0; n--) {
    heap[n]=n;
    down(n);
  }
}

/* 現在のベストの在庫から1本多くとり，
   ヒープを再構成する */
int get_max_and_update(void)
{
  int result = next_len(heap[0]);
  npiece[heap[0]]++;
  down(0);
  return result;
}

int task(void)
```

```
{
  int p;
  int max;

  init_heap();
  /* 切り出し本数についてのループ */
  for(p=1; p<=K; p++){
    max = get_max_and_update();
  }
  return max;
}
```

上のプログラムで，最初に在庫をヒープに格納するのが関数 init_heap である．初期のバージョンではこの関数は以下のようになっていた：

```
void init_heap(void)
{
  int n;
  for(n=0; n<N; n++){
    heap[n] = n;
    up(n);
  }
}
```

関数 up の定義は省略するが，注目点から上に向けてヒープ中の要素を交換していくものである．その計算量は，注目点の深さに比例する．この初期バージョンでは，要素を 1 つヒープに挿入するたびに up が呼ばれるので，$O(N \log N)$ の計算量を要する．

これに対して，現行バージョンの init_heap が呼び出している down は注目点から下に向けて交換していくので，全体としての計算量は $O(N)$ で済む．なぜそうなるかというと，全体の半分はヒープの葉であるから手数は 1，残りの 1/4 の手数は 2，その残りの 1/8 は手数が 3... というように，N チームが参加したトーナメント戦における全試合回数（の 2 倍）とほぼ同様になるからである．

■比例代表選挙との対応

さて，我が国の比例代表選挙において，各政党の得票数をもとに議席配分を求める方法として，ドント方式（d'Hondt method）なるものが採用されていることはご存知であろう．これはベルギーの法学者，数学者であった Viktor d'Hondt が 1878 年に発表した議席配分の方法で，たとえば文献 1）の説明には，「政党ごとの得票数をその政党がこれまでに得た議席数 +1 で割った値，すなわち $x[i]/(y[i]+1)$ が最大な政党に次の議席を与える，という方法で計算する」とある．

本稿の「ケーブルの各在庫」を「各政党」と読み替え，それぞれの在庫の長さを各政党の得票数と読み替えれば，これはまさしく本稿の「第 3 の方式」にほかならない．

つまり，ドント方式とは，当選者がいずれも等しい得票で当選したと考えた場合に，最も死票が少なくなるような議席配分方法—（1）—であったのだった．

筆者は，しかし，本稿のもともとの問題（Cable master）を読んだ時点では，比例代表の議席配分問題とはまったく結び付けることができなかった．なぜそうなったかを自分なりに考察してみると，新聞などでよく見るドント方式の解説は「各政党の総得票数を 1, 2, 3..., と整数で割り，その値〈つまり，割った答え〉の大きい順に議席を割り振る比例配分の方法」のように，いわば「手順（アルゴリズム）」の説明であって，そのアルゴリズムが達成しようとする目標（上の（1））の記述にはなっていないところに原因があるように思える．

対象が選挙であるから，手順の記述の正確さの方が目標の記述より重要であるのは確かであるが，一般社会においては，達成すべき目標—つまり問題—と，それを解くアルゴリズムがきちんと区別されていないことを示唆しているようにも思う．

■さいごに

今回の問題では，3 つの解法を提示した．

動的計画法による第 1 の解法は，通常のコンテストの問題であれば正解となり得るものであるが，計算量の点で残念ながら採用できない．

第 2（二分探索）と第 3（ドント方式）はいずれも正解といえる．どちらが望ましいか，という点では議論が分かれるところであろう．

二分探索は，単調性のある問題に一般に適用できる点で応用性に優れていると思う．

一方，ドント方式は，有効桁数の制限がなくとも利用できるところが優れているし，なによりも問題に対する考察に根ざしたシンプルなアルゴリズムである点がよい．実際，選挙の比例代表の配分アルゴリズムが，もし二分探索によって記述されていたとしたら，これほど広く採用されるに至ったか怪しまれるのである．

参考文献

1）有沢　誠：種々の比例代表制アルゴリズムの比較について，第 31 回プログラミングシンポジウム報告集，pp.75-82 (1990).

（平成 16 年 6 月 11 日受付）