電気電子情報工学実験 II (b)
実践的・競技プログラミング 練習コンテスト（Contest 3-1）
解説

廣田悠輔
y-hirota@u-fukui.ac.jp

目次

1 問題 A：一夜漬け

解説

参考書を読む区間の開始ページを S，終了ページを E とおくと，問題は以下のように読み替えられる．

次の値を求めよ．

$$\min_{1 \le S \le E \le N} E - S + 1$$
$$\text{where} \quad \sum_{i=S}^{E} A_i \ \ge \ P.$$

このような問題を解くシンプルな方法は，$1 \le S \le E \le N$ を満たす S, E の全組み合わせ約 $N^2/2$ 個を調べることである．しかしながら，本問題では最大で $N = 10^5$ が与えられる可能性があるため，個々の組み合わせをどのように高速に調べたとしても制限時間超過となってしまう．

上述のシンプルな方法よりも効率的な探索方法にしゃくとり法がある．しゃくとり法は，Algorithm 1 のように，$S = 1, 2, \ldots, N$ について，それぞれの S で条件を満たし $E - S + 1$ が最小になる E_S を探索し，最後に $E_S - S + 1$ $(S = 1, 2, \ldots, N)$ の中で最小となるを求めるというのが基本的な考え方となる．このような大枠は前述のシンプルな方法に類似しているが，しゃくとり法では，Algorithm 1 の 2 行目の E_S の探索の際，過去の探索結果の情報を使って探索範囲を絞り込むことで探索回数を大きく削減し，効率的な探索を実現している．

具体的な処理の流れについて，図 1 に示す $N = 10$, $P = 7$ の例を使って説明する．まず $S = 1$ の場合に，

Algorithm 1 しゃくとり法の概略

1: **for** $S = 1, 2, \ldots, N$ **do**
2: Find $E_S \leftarrow \min_E(E - S + 1)$ where $\sum_{i=S}^{E} A_i \leq P$.
3: **end for**
4: Output $\min_{S=1,2,\ldots,N} E_S - S + 1$.

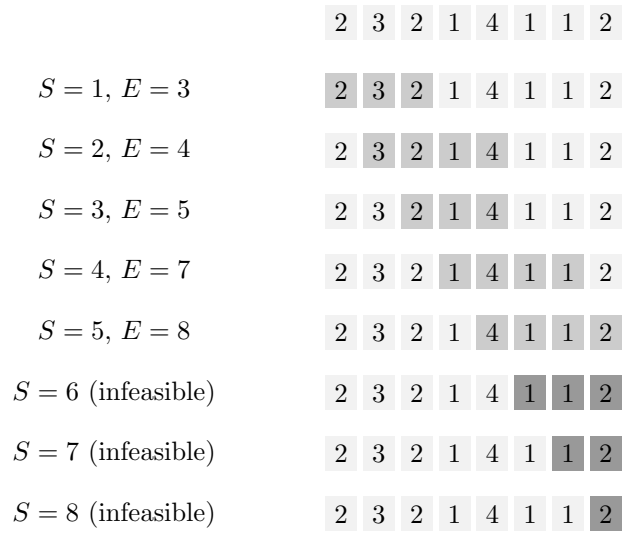

図 1　しゃくとり法の振る舞い（$N = 8, P = 7$ の例）．各行における色付きのマス目の数値はページごとの項目数を示す．

ページ数最小で条件を満たす区間（すなわち条件を満たす最小の E）を求める．これには単に $E = 1, 2, \ldots$ と順に E の値を増やしていき，初めて

$$\sum_{i=S(=1)}^{E} A_i \geq P$$

を満たす E を見つければ良い（一度条件を満たす E をみつければ，それより大きな値については探索不要）．図 1 の例では $S = 1$ のとき，$E = 3$ ではじめて項目数和が条件を満たす．次に $S = 2$ の場合を考えるが，ここからは一つ前の結果（$S = 2$ の場合には $S = 1$ の結果）を利用することで，無駄な探索を回避する．$S = 1$ では $E = 3$ のときに初めて条件を満たしのたであるのだから，$S = 2$ では $E < 3$ では条件を満たすことは絶対にない（各ページの項目数が非負であることに注意せよ）．したがって，$S = 2$ では $E = 3, 4, \ldots$ のみを順に調べていけば良いことになる．また，$S = 2$ から $E = 3$ の範囲の項目数和は，$S = 1$ から $E = 3$ の範囲の項目数和から A_1 を引いた値であるので，開始時の項目数和の和についても $S = 1$ での探索情報を再利用できる．以下，$S = 3, 4, \ldots$ の場合も同様であり，直前の探索結果を利用することで，探索範囲の絞り込み，項目数和の情報の再利用が可能となる．最終的には，S がある大きな値となったところで，E を最大値である N としたところで条件を満たさなくなるなるので，以後の探索は打ち切ることができる．例えば，図 1 では $S \geq 6$ から先は条件を満たさないので，探索不要である．

しゃくとり法の全体像を Algorithm 2 に示す．このアルゴリズムを見ればわかるとおり，しゃくとり法の計算量は $\mathrm{O}(N)$ となる（S, E' がそれぞれ高々 N 回しか変化していないことに注意）．

Algorithm 2 しゃくとり法

1: Set $E' \leftarrow 1$ (variable, end page number plus one).
2: Set $Q \leftarrow 0$ (variable, sum of the items in the pages $[S, E')$).
3: Set $M \leftarrow$ arbitrary value larger than N (variable, min. pages).
4: **for** $S = 1, 2, \ldots, N$ **do**
5: **while** $E' \leq N$ and $Q < P$ **do**
6: $Q \leftarrow Q + A_E$
7: $E' \leftarrow E' + 1$
8: **end while**
9: **if** $Q < P$ **then**
10: **break**
11: **end if**
12: $M \leftarrow \min(M, E' - S)$
13: $Q \leftarrow Q - A_S$
14: **end for**
15: **if** $M \leq$ N **then**
16: Output M.
17: **else**
18: Output -1 (no satisfactory solutions).
19: **end if**

ノート

- 本問題に対するしゃくとり法の適用は，すべてのページの項目数が非負の値であることに依存している．このような条件が成り立たない問題にはしゃくとり法が適用できないことに注意せよ．
- 解説中のアルゴリズムと，ソースコードではページ番号の開始の値が異なることに注意せよ．解説中のページ番号の自然言語の習慣に慣習に従い 1 ページから開始しているが，ソースコード中では配列の扱いやすさのために 0 ページを開始番号としている．

回答例（しゃくとり法による $\mathrm{O}(n)$ の方法）

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5   int n, p;
6   int *a;
7   int i;
8   int start, end;
9   int sum, min_pages;
```

```
10
11    scanf("%d␣%d", &n, &p);
12    a = malloc(sizeof(int) * n);
13    for (i = 0; i < n; ++i) {
14      scanf("%d", &a[i]);
15    }
16
17    start = 0;
18    end = 0;
19    sum = 0;
20    min_pages = n + 1;
21    while (1) {
22      while (end < n && sum < p) {
23        sum += a[end];
24        end++;
25      }
26      if (sum < p) {
27        break;
28      }
29      if (end - start < min_pages) {
30        min_pages = end - start;
31      }
32      sum -= a[start];
33      start++;
34    }
35
36    if (min_pages == n + 1) {
37      // No solution
38      printf("%d\n", -1);
39    } else {
40      printf("%d\n", min_pages);
41    }
42
43    free(a);
44    return 0;
45  }
```

正しい答えを出力するが制限時間超過になる例（累積和と $\mathrm{O}(N^2)$ 回の探索の組み合わせ）

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

```c
#include <stdlib.h>

int main(void) {
  int n, p;
  int *psum;
  int i;
  int start, end, min_pages;

  scanf("%d␣%d", &n, &p);
  psum = malloc(sizeof(int) * (n + 1));
  psum[0] = 0;
  for (i = 0; i < n; ++i) {
    int a;
    scanf("%d", &a);
    psum[i + 1] = psum[i] + a;
  }

  if (psum[n] < p) {
    // No solution.
    printf("-1\n");
    return 0;
  }

  min_pages = n;
  for (start = 0; start < n; ++start) {
    for (end = start + 1; end <= n; ++end) {
      if (psum[end] - psum[start] >= p && end - start < min_pages) {
        min_pages = end - start;
      }
    }
  }

  printf("%d\n", min_pages);

  free(psum);
  return 0;
}
```

2　問題 B：三本の鎖

解説

入力 1 を例に目的の鎖を作成する具体的な方法を考えてみよう．入力 1 では目的の鎖の長さは 3 本とも 10 であり，所有する 5 本の鎖の長さはそれぞれ 2, 3, 4, 5, 6 である．3 本の目的の鎖は，例えば以下のような方法で作成できる．

1 本目の鎖　長さ 2，3 の鎖をつなげて新たに長さ 5 の鎖を作る．続けて，この鎖に長さ 4 の鎖をつなげて新たに長さ 9 の鎖を作る．さらに，この鎖を 1 延伸する操作を行い，長さ 10 の鎖を作る．
2 本目の鎖　長さ 5 の鎖に対して，1 延伸する操作を 5 回行い，長さ 10 の鎖を作る．
3 本目の鎖　長さ 6 の鎖に対して，1 延伸する操作を 4 回行い，長さ 10 の鎖を作る．

このとき目的の鎖を作るのにかかった費用は，それぞれ

1 本目の鎖　$2P + Q$（2 回の接続，1 回の延伸），
2 本目の鎖　$5Q$（0 回の接続，5 回の延伸），
3 本目の鎖　$4Q$（0 回の接続，4 回の延伸），

となる．ところで，1 本目の鎖は以下のように作ることもできる．

1 本目の鎖　長さ 2，4 の鎖をつなげて新たに長さ 6 の鎖を作る．続けて，この鎖に長さ 3 の鎖をつなげて新たに長さ 9 の鎖を作る．さらに，この鎖を 1 延伸する操作を行い，長さ 10 の鎖を作る．

あるいは，

1 本目の鎖　長さ 2 の鎖を 1 延伸する操作を行い，長さ 3 の鎖を作る．続けて，長さ 4 の鎖をつなげて新たに長さ 7 の鎖を作る．さらに，この鎖に長さ 3 の鎖をつなげて新たに長さ 10 の鎖を作る．

というようにも作成できる．

上記の具体例から，いずれの方法で目的の鎖の 1 本目を作った場合でも鎖の作成には 2 回の接続と 1 回の延伸が必要であり，かかる費用は変わっていないことが察知できる．上記の具体例に限らず，鎖の作成にかかる費用（すなわち接続や延伸の回数）は，

- どのような順番でつないでも変わらない，
- いつ，どの鎖を延伸しても変わらない

という性質がある．したがって，目的の鎖の作成にかかる費用は，元となった所有する鎖の本数と長さが分かればその接続や延伸の順番に関係なく求められる．

以上の考察から，所有する N 本の鎖が，それぞれ

- 目的の鎖（1 本目）に使われる，
- 目的の鎖（2 本目）に使われる，
- 目的の鎖（3 本目）に使われる，
- 目的の鎖の作成には使われない

のいずれかとなる組み合わせ 4^N ケースについて，鎖の作成可能性と作成可能な場合の合計費用を調べれば，目的の鎖の作成に必要な最小費用を求められる．問題の制約より N の最大値は 11 であるので，調べるべき組合せの数は高々 $4^{11} \simeq 4.2 \times 10^6$ である．

全探索の実装は再帰による深さ優先探索を用いれば良い．葉ノードは具体的な組み合わせの費用を値とし，親ノードの値は自身のもつ子ノードの値の最小値となる．

コメント

本問題は AtCoder Beginner Contest にて過去に出題された問題 [1] を元に作成された．

具体例から得られた洞察を元に問題を読み替えてよく知られた解き方（本問題ではツリーの全探索）が適用できるようにするテクニックは競技プログラミングにおいて重要である．このようなテクニックは通常のプログラミングにおいても役立つものであるので，是非とも使いこなせるように努められたい．

正答例

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

int dfs(int a, int b, int c, int p, int q, int n, int *d,
        int depth, int *target_chain_id) {
  int i;
  int min_cost;

  if (depth == n) {
    // Check feasibility.
    int sum_length[] = {0, 0, 0, 0};
    int num_chains[] = {0, 0, 0, 0};
    int cost;
    for (i = 0; i < n; ++i) {
      sum_length[target_chain_id[i]] += d[i];
      num_chains[target_chain_id[i]]++;
    }
    if (a < sum_length[0] || b < sum_length[1] || c < sum_length[2]) {
      // Infeasible: over sized
      return INT_MAX;
    }
    for (i = 0; i < 3; ++i) {
      if (num_chains[i] == 0) {
        // No chain.
        return INT_MAX;
      }
```

```
27      }
28
29      // Calculate the cost.
30      cost = 0;
31      for (i = 0; i < 3; ++i) {
32        cost += (num_chains[i] - 1) * p;
33      }
34      cost += (a - sum_length[0]) * q;
35      cost += (b - sum_length[1]) * q;
36      cost += (c - sum_length[2]) * q;
37      return cost;
38    }
39
40    min_cost = INT_MAX;
41    for (i = 0; i < 4; ++i) {
42      int cost;
43      target_chain_id[depth] = i;
44      cost = dfs(a, b, c, p, q, n, d, depth + 1, target_chain_id);
45      if (min_cost > cost) {
46        min_cost = cost;
47      }
48    }
49    return min_cost;
50  }
51
52  #define N_MAX (11)
53  int main(void) {
54    int a, b, c;
55    int p, q;
56    int n;
57    int d[N_MAX];
58    int target_chain_id[N_MAX];
59    int i;
60    int cost;
61
62    scanf("%d␣%d␣%d", &a, &b, &c);
63    scanf("%d␣%d", &p, &q);
64    scanf("%d", &n);
65    for (i = 0; i < n; ++i) {
66      scanf("%d", &d[i]);
```

```
67    }
68
69    cost = dfs(a, b, c, p, q, n, d, 0, target_chain_id);
70    if (cost == INT_MAX) {
71      printf("-1\n");
72    } else {
73      printf("%d\n", cost);
74    }
75
76    return 0;
77  }
```

3 問題 C：カードシャッフル

解説

目的の出力を得る最も簡単な方法は，Algorithm 3 に示すように，カード交換の様子をプログラム上でシミュレーションしてしまうことである．しかしながら，このような方法の計算量は $\mathrm{O}(ML)$（出力部分を考慮したより正確な評価は $\mathrm{O}(ML+N)$）であり，本問題の M, L の最大値はそれぞれ 10^5, 10^6 であるため，制限時間内に出力を得ることは極めて困難である．

続いて，ML 回のカード交換を回避する方法を考える．毎回のカード交換を互換と考え，その積を取れば L 回の交換に相当する置換 σ が計算できる．したがって，最初に与えられたカードの数値の並びに対して置換 σ に対応するように並び替える処理を M 回繰り返せば目的の出力が得られる．アルゴリズムを Algorithm 4 に示す．例えば，$N=4$，$L=3$，$(A_{1,1},A_{1,2})=(2,3)$，$(A_{2,1},A_{2,2})=(1,3)$，$(A_{3,1},A_{3,2})=(2,4)$ である場合，L 回の交換に相当する置換 σ は，

$$\sigma=\sigma_{2,4}\sigma_{1,3}\sigma_{2,3}=\begin{pmatrix}1&2&3&4\\2&4&1&3\end{pmatrix}$$

となる．ただし，$\sigma_{p,q}$ は p と q を交換する互換

$$\sigma_{p,q}=\begin{pmatrix}1&2&\cdots&p-1&p&p+1&\cdots&q-1&q&q+1&\cdots&N\\1&2&\cdots&p-1&q&p+1&\cdots&q-1&p&q+1&\cdots&N\end{pmatrix}$$

である．さらに，$M=3$，最初のカードの並び順を $(C_1,C_2,C_3,C_4)=(11,22,33,44)$ としたとき，カードは $(11,22,33,44)\to(22,44,11,33)\to(44,33,22,11)\to(33,11,44,22)$ と並び替えられる．このアルゴリズムの計算量は，置換 σ を求めるのに $\mathrm{O}(L)$，M 回の置換の適用に $\mathrm{O}(MN)$ の計算量が必要であり，全体の計算量は $\mathrm{O}(MN+L)$ となる．本問題の M, L, N の最大値はそれぞれ 10^5, 10^6, 10^4 であるので，制限時間内に確実に出力を得るのは難しい（良いデータ構造を選択して実装すればかろうじて間に合うかもしれない）．

より少ない計算量で目的の出力を得る方法に，ダブリングを応用するものがある．M は重複のない 2 のべき乗である K 個の自然数 $t_1,t_2,\ldots,t_K$ により

$$M=\sum_{i=1}^{K}t_i$$

Algorithm 3 カード交換の様子のシミュレーションによる方法

1: **for** $i = 1, 2, \ldots, M$ **do**
2: **for** $j = 1, 2, \ldots, L$ **do**
3: Swap the $A_{j,1}$-th card for the $A_{j,2}$-th card.
4: **end for**
5: **end for**
6: Output the card numbers in the current order.

Algorithm 4 L 回のカード交換に相当する置換を計算する方法

1: $\sigma \leftarrow \begin{pmatrix} 1 & 2 & \cdots & N \\ 1 & 2 & \cdots & N \end{pmatrix}$ (an identity permutation)
2: **for** $j = 1, 2, \ldots, L$ **do**
3: $\sigma \leftarrow \sigma_{A_{j,1}, A_{j,2}} \sigma$ where $\sigma_{p,q}$ is a transposition that swaps p and q.
4: **end for**
5: **for** $i = 1, 2, \ldots, M$ **do**
6: Apply the permutation σ to the number sequence.
7: **end for**
8: Output the card numbers in the current order.

と表される（$K \leq \log M$）．このとき，M 回の置換の積は

$$\sigma^M = \sigma^{t_K} \sigma^{t_{K-1}} \cdots \sigma^{t_1} \tag{1}$$

と表現することができる．例えば，$M = 13$ の場合，$M = 8 + 4 + 1$ であるので，

$$\sigma^M = \sigma^8 \sigma^4 \sigma$$

と表現できる．置換 σ の 2 乗，4 乗，8 乗，... は，

$$\sigma^{2i} = \sigma^i \sigma^i \quad (i = 1, 2, \ldots)$$

を順に計算することで求められる．したがって，置換の 2 乗，4 乗，8 乗，...をあらかじめ計算しておき，最初に与えられたカードの数値の並びに対して (1) の右辺に現れるものを順に適用すれば，目的の出力が得られる（Algorithm 5）．このアルゴリズムは，σ を求めるのに $\mathrm{O}(L)$（2–4 行目），σ^2, σ^4, σ^8 ... を求めるのに $\mathrm{O}(N \log M)$（5–7 行目），σ, σ^2, σ^4 ... から必要なものを適用するのに $\mathrm{O}(N \log M)$（8–10 行目）の計算量が必要である．したがって，全体の計算量は $\mathrm{O}(N \log M + L)$ となり，制限時間内に余裕をもって問題を解くことができる．

回答例（ダブリングによる方法）

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void permute_int_array(int n, int *c, int *permutation, int *iwork) {
```

Algorithm 5 ダブリングによる方法

1: $\sigma \leftarrow \begin{pmatrix} 1 & 2 & \cdots & N \\ 1 & 2 & \cdots & N \end{pmatrix}$ (an identity permutation)
2: **for** $j = 1, 2, \ldots, L$ **do**
3: $\quad \sigma \leftarrow \sigma_{A_{j,1},A_{j,2}}\sigma$ where $\sigma_{p,q}$ is a transposition that swaps p and q.
4: **end for**
5: **for** $i = 2, 4, 8, \ldots, \lfloor 2^{\log_2 M} \rfloor$ **do**
6: $\quad \sigma^i \leftarrow \sigma^{i/2}\sigma^{i/2}$
7: **end for**
8: **for** $p = 1, 2, \ldots, K$ **do**
9: $\quad$ Apply the permutation σ^{t_p} for the current number sequence.
10: **end for**
11: Output the card numbers in the current order.

```
  int i;
  for (i = 0; i < n; ++i) {
    iwork[i] = c[permutation[i]];
  }
  for (i = 0; i < n; ++i) {
    c[i] = iwork[i];
  }
}

void swap_int(int *i, int *j) {
  int tmp = *i;
  *i = *j;
  *j = tmp;
}

int main(void) {
  int m, l;
  int *a1, *a2;
  int n;
  int *c, *iwork;
  int **permutation;
  int p_max;
  int i, j;

  // Read input data.
```

```
30  scanf("%d␣%d", &m, &l);
31  a1 = malloc(sizeof(int) * l);
32  a2 = malloc(sizeof(int) * l);
33  for (i = 0; i < l; ++i) {
34    scanf("%d␣%d", &a1[i], &a2[i]);
35
36    // Use 0-origin index.
37    a1[i]--;
38    a2[i]--;
39  }
40  scanf("%d", &n);
41  c = malloc(sizeof(int) * n);
42  for (i = 0; i < n; ++i) {
43    scanf("%d", &c[i]);
44  }
45
46  // 1, 2, 4, ..., 2^(p_max - 1)-step permutation storage.
47  p_max = 1;
48  for (i = 2; i <= m; i *= 2) {
49    p_max++;
50  }
51  permutation = malloc(sizeof(int *) * p_max);
52  for (i = 0; i < p_max; ++i) {
53    permutation[i] = malloc(sizeof(int) * n);
54  }
55
56  // Calculate the 1-step permutation.
57  for (i = 0; i < n; ++i) {
58    permutation[0][i] = i;
59  }
60  for (i = 0; i < l; ++i) {
61    swap_int(&permutation[0][a1[i]], &permutation[0][a2[i]]);
62  }
63
64  // Calculate the 2, 4, 8, ..., 2^(p_max - 1)-step permutations.
65  for (i = 1; i < p_max; ++i) {
66    for (j = 0; j < n; ++j) {
67      permutation[i][j] = permutation[i - 1][permutation[i - 1][j]];
68    }
69  }
```

```
70
71    // Apply the 1, 2, 4, 8, ..., 2^(p_max - 1)-step permutations if
          needed.
72    iwork = malloc(sizeof(int) * n);
73    for (i = m, j = 0; i >= 1; i /= 2, j++) {
74      if (i % 2 == 1) {
75        permute_int_array(n, c, permutation[j], iwork);
76      }
77    }
78
79    // Print the results.
80    for (i = 0; i < n; ++i) {
81      printf("%d\n", c[i]);
82    }
83
84    free(iwork);
85    free(permutation);
86    free(c);
87    free(a2);
88    free(a1);
89    return 0;
90  }
```

参考文献

[1] AtCoder Beginner Contest 119，問題 C「Synthetic Kadomatsu」，https://atcoder.jp/contests/abc119/tasks/abc119_c （2020 年 6 月 21 日に閲覧）.