2020 年度 電気電子情報工学実験 II (b)
実践的・競技プログラミング 第 2 回コンテスト解説

廣田悠輔
y-hirota@u-fukui.ac.jp

2020 年 6 月 25 日

目次

1 問題 A：素数判定 1
2 問題 B：腹が減っては戦が出来ぬ 2
3 問題 C：制約付き整数方程式 4
4 問題 D：暗号文の復号 6
5 問題 E：三本の鎖 10

1 問題 A：素数判定

解説

整数 $N = 2, 3$ は素数である．整数 N $(N \geq 4)$ が素数であるか否かは，$2 \leq M \leq \sqrt{N}$ を満たす整数 M の中に N を割り切る M（すなわち N の約数である M）が存在するか否かにより判定できる．N を割り切る M が存在しなければ N は素数であり，一つでも存在すれば N は合成数である．したがって，`for` 文などにより，そのような M が存在するか否かを調べるプログラムを実装すれば良い．

割り切れるか否かを調べるべき M の範囲が $2 \leq M \leq N - 1$ ではなく，$2 \leq M \leq \sqrt{N}$ である理由について補足する．もし，$\sqrt{N} < M \leq N - 1$ を満たす M が N を割り切れるのであれば，$K = N/M$ もまた N を割り切る．$\sqrt{N} < M \leq N - 1$ であるので，K は $2 \leq K \leq \sqrt{N}$ を満たす．したがって，N を割り切る $2 \leq M \leq N - 1$ が存在するならば，N を割り切る $2 \leq M \leq \sqrt{N}$ が存在する．ゆえに，$2 \leq M \leq \sqrt{N}$ の範囲だけを調べれば十分であるとわかる．

割り切れるか否かを調べるべき M の範囲を $2 \leq M \leq \sqrt{N}$ とした場合，計算量は $\mathrm{O}(\sqrt{N})$ となる．一方，$2 \leq M \leq N - 1$ とした場合，計算量は $\mathrm{O}(N)$ となる．本問題では N について $N \leq 32767$ の制約があるので，後者の方法を実装した場合でも時間制限に十分間に合う．

正答例

```
#include <stdio.h>

int is_prime(int n) {
  int i;
  for (i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      return 0;
    }
  }
  return 1;
}

int main(void) {
  int n;
  scanf("%d", &n);

  if (is_prime(n)) {
    printf("Yes\n");
  } else {
    printf("No\n");
  }

  return 0;
}
```

2　問題 B：腹が減っては戦が出来ぬ

解説

条件 $P_i \leq A$ および $B \leq Q_i < C$ のもと，$\lfloor Q_i/P_i \rfloor$ の最大値を求める条件付き最大化問題を解けば良い．ただし，$\lfloor Q_i/P_i \rfloor$ は Q_i/P_i の小数点以下を切り捨てた値を意味する．このような条件付き最大化問題を解くプログラムは，$i = 1, 2, \ldots, N$ について順に P_i, Q_i に関する条件を満たしているかを判定し，条件を満たす場合には $\lfloor Q_i/P_i \rfloor$ の値を順次調べる線形探索の応用により作成できる．このときの計算量は $\mathrm{O}(N)$ であり，問題の制約より $1 \leq N \leq 10^2$ であるので，制限時間には十分に間に合う．

コメント

問題の易しさにもかかわらず非常に誤答が多かった．誤答の多くは，問題文中の「熱量（カロリー）が B cal 以上，C cal 未満であること」を，正しく $B \leq$ 熱量 $< C$ と解釈せず，$B \leq$ 熱量 $\leq C$ と誤解したもので

あった．

自然言語で記述された問題文（実際のソフトウェア開発現場であれば仕様書や設計書）の誤解（勘違い）による不具合は，一人でソフトウェア開発を行うときには露見しにくく危険である．多人数でソフトウェア開発を行う場合には，

- ペアプログラミングにより解釈の誤りを予防する，
- コードレビューにより指摘を受ける，
- 別の人がテストコードを書くことにより誤りを検出する（本体プログラムを書いた人自身がテストコードを書く場合はテストも誤った解釈に基づくので誤りを検出できないことに注意）

などの手段をとることで，危険を抑制できる場合がある．

正答例

```
1  #include <stdio.h>
2
3  int main(void) {
4    int n, a, b, c;
5    int found = 0;
6    int best_performance;
7    int i;
8
9    scanf("%d", &n);
10   scanf("%d␣%d␣%d", &a, &b, &c);
11   for (i = 0; i < n; ++i) {
12     int p, q;
13     scanf("%d␣%d", &p, &q);
14     if (p <= a && q >= b && q < c) {
15       if (found == 0) {
16         found = 1;
17         best_performance = q / p;
18       } else {
19         if (best_performance < q / p) {
20           best_performance = q / p;
21         }
22       }
23     }
24   }
25   if (found) {
26     printf("%d\n", best_performance);
27   } else {
```

```
28      printf("-1\n");
29    }
30
31    return 0;
32  }
```

3　問題 C：制約付き整数方程式

解説

基本的な考え方は第 1 回コンテストの問題 C「簡単な制約付き整数方程式」と同じである．A_i のとりうる値は最大でも 3 通り（0, 1, 2 のいずれか）であり，変数の個数は最大でも 10 個である．したがって，変数 $A_1, A_2, \ldots, A_M$ の取りうる組み合わせの数は最大でも 3^{10} $(= 59049)$ でしかない．ゆえに，本問題を解くには，すべての組み合わせについて等式が成り立つか否かを調べ，その個数を数え上げれば良い．

全探索の実装には，再帰による深さ優先探索を用いるのが容易である．この問題では変数の個数は可変であり，また変数の最大数も 10 とやや多い．したがって，第 1 回コンテストの問題 C のような多重ループによる実装はやや困難である（不可能ではないが煩雑で実装に時間を要す）．

方程式が成り立つ変数の組み合わせの数は図 1 のような $L + 1$ 分木を用いて表現できる．この木の根ノードが方程式が成り立つ変数の組み合わせの数（すなわち求めるべき値）を表す．根ノードの $L + 1$ 個の子ノードは，それぞれ A_1 の値のみを固定したときの（例えば図 1 では左から順に $A_1 = 0$, $A_1 = 1 (= L)$ と固定したときの）方程式が成り立つ変数の組み合わせの数を表す．根ノードの子ノードも同様に $L + 1$ 個の子ノードをもち，それぞれのノードが A_1 に加えて A_2 の値を固定したときの方程式が成り立つ変数の組み合わせの数を表す．同様の親子関係が M 個の変数すべてが固定されるまで繰り返され，葉ノードはそれぞれ，すべての変数を固定したときの方程式が成り立つ変数の組み合わせの数（全変数が固定されているので 0 または 1 のいずれか）を表す．このような親子関係にあるため，葉ノード以外の各ノードの値は，自身の子ノードの値の和となる．したがって，方程式が成り立つ変数の組み合わせの数を計算するには，葉ノードの値（すなわち具体的な変数の組み合わせについて方程式が成り立つか否か）を調べて，根ノードに向かって値を足し合わせて行くように再帰関数を実装すれば良い．

コメント

コンテスト参加者の中に，十重ループと分岐を駆使して正答を得た者がいた．競技プログラミングの観点からはどのように実装しても早く正答できれば良いので，このような回答であっても問題はない．しかしながら，通常のプログラミングでは保守性や拡張性が求められるので，再帰による探索も書けるようになっておくこと．

正答例

```
1  #include <stdio.h>
2
3  int n_solutions(int l, int m, int n, int *b, int partial_sum,
4                  int depth) {
```

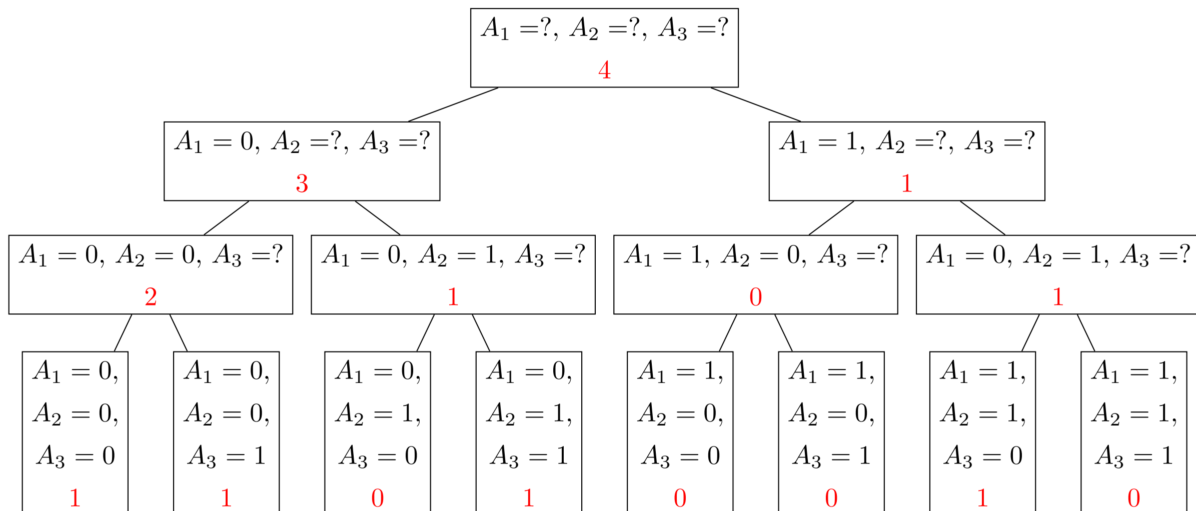

図 1　$L+1$ 分木による変数の組み合わせの表現（$L = 1, M = 3$ の例）．各ノード下部の赤色の数値はノードの値を意味する．

```
5     int i, count;
6
7     if (depth == m) {
8       if (partial_sum == n) {
9         return 1;
10      } else {
11        return 0;
12      }
13    }
14
15    count = 0;
16    for (i = 0; i <= l; ++i) {
17      count += n_solutions(l, m, n, b, partial_sum + i * b[depth],
18                           depth + 1);
19    }
20    return count;
21  }
22
23  #define M_MAX (10)
24  int main(void) {
25    int l, m, n;
26    int b[M_MAX];
27    int i;
28
```

```
29    scanf("%d␣%d␣%d", &l, &m, &n);
30    for (i = 0; i < m; ++i) {
31      scanf("%d", &b[i]);
32    }
33
34    printf("%d\n", n_solutions(l, m, n, b, 0, 0));
35
36    return 0;
37  }
```

4　問題 D：暗号文の復号

解説

本問題では，書いてある通りの復号操作をそのまま実装すれば正しい出力が得られる．しかしながら，実装法によっては実行時間制限を超過してしまうため，適切なデータ構造を選択することが重要となる．

文字列を C プログラム上で扱うには，

- char 型配列を使用する，
- char 型データを格納するリスト構造を使用する

という方法が考えられる．結論を先に書くと，本問題ではリスト構造を用いるべきである．本問題の復号操作では与えられた文字列に対して文字の消去，文字の挿入，部分文字列の交換の 3 つの操作を繰り返し行う必要がある．配列を用いて文字列を格納する場合，制限時間内に実行可能なプログラムを実装するのが難しくなる．

リスト構造を用いて文字列を格納する場合，文字の消去，文字の挿入，部分文字列の操作は，それぞれ以下の方法で実現できる．ただし，リストの先頭および末尾のノードへの参照は常に記憶され，リストを構成する各ノードは次のノードへの参照を持つ（ただし末尾ノードのみ NULL 値を参照する）ものとする．

- 文字列の消去は，リストの先頭から $B_i - 1$ 番目のノードを参照し，その $B_i - 1$ 番目の持つ次のノードへの参照を B_i 番目のノードから $B_i + 1$ 番目のノードに変更することにより実現される．この操作により，変更前の B_i 番目のノードはリストを構成するノードではなくなる．$B_i - 1$ 番目のノードをリストの先頭から順に探索するのには $B_i - 1$ 回のポインタ変数参照が，$B_i - 1$ 番目のノードの次のノードを変更するのに 1 回のポインタ変数書き換えが必要である．
- 文字列の挿入は，以下の手順により $B_i - 1$ 番目のノードと（変更前の）B_i 番目のノードの間に文字 _ を格納するノードが挿入されることで実現される．実現される．
  1. 文字 _ を格納するノードを新たに作成する．
  2. $B_i - 1$ 番目のノードを，リストの先頭から順にたどって参照する．
  3. $B_i - 1$ 番目のノードの持つ次のノードへの参照を作成したノードに変更する．
  4. 作成したノードの持つ次のノードへの参照を書き換え前の B_i 番目のノードに設定する．

  新しいノードの作成には O(1) 回の処理が，$B_i - 1$ 番目のノードをリストの先頭から順に探索するのには $B_i - 1$ 回のポインタ変数参照が，$B_i - 1$ 番目のノードおよび新しいノードの次のノードへの参照を

変更するには 2 回のポインタ変数書き換えが必要である．

- 部分文字列の交換は，以下の手順により実現される．
  1. $B_i - 1$ 番目のノードを，リストの先頭から順にたどって参照する．
  2. リストの末尾ノードへの参照を，変更前の $B_i - 1$ 番目のノードに変更する．
  3. リストの先頭ノードへの参照を，変更前の B_i 番目のノードに変更する．
  4. 変更前の $B_i - 1$ 番目のノードの持つ次のノードへの参照を，NULL 値に変更する．
  5. 変更前の末尾ノードの持つ次のノードへの参照を，変更前の 1 番目のノードに変更する．

  最初の操作には $B_i - 1$ 回のポインタ変数参照が，以降の操作には $\mathrm{O}(1)$ 回の処理が必要である．

文字の削除，文字の挿入，部分文字列の交換のいずれでも，計算量の大半は $B_i - 1$ 回のポインタ変数参照に費やされる．復号操作全体では，文字の削除，文字の挿入，部分文字列の交換の操作が合計 M 回行われる．したがって，復号操作全体でのポインタ変数参照回数は約 $\sum_{i=1}^{M} B_i$ 回となる．制約 $1 \leq M \leq 10^5$, $1 \leq B_i \leq 10^2$ $(1 \leq i \leq M)$ より，計算量は高々 10^7 である．したがって，リスト構造を用いるプログラムは制限時間内に余裕をもって実行可能である．

正答例

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

struct Node {
  char c;
  struct Node *next;
};

struct List {
  struct Node *first;
  struct Node *last;
};

void push_back(char c, struct List *li) {
  struct Node *no = malloc(sizeof(struct Node));
  no->c = c;
  no->next = NULL;

  if (li->first == NULL && li->last == NULL) {
    li->first = no;
    li->last = no;
  } else {
    (li->last)->next = no;
    li->last = no;
```

```
25    }
26  }
27
28  void print_list(struct List li) {
29    struct Node *p = li.first;
30    while (p != NULL) {
31      printf("%c", p->c);
32      p = p->next;
33    }
34    printf("\n");
35  }
36
37  void operation(int a, int b, struct List *li) {
38    int i;
39    struct Node *p, *current, *prev;
40    switch (a) {
41    case 1:
42      if (b == 1) {
43        current = li->first;
44        li->first = current->next;
45        free(current);
46      } else {
47        current = li->first;
48        for (i = 1; i < b; ++i) {
49          prev = current;
50          current = current->next;
51        }
52        prev->next = current->next;
53        if (current == li->last) {
54          li->last = prev;
55        }
56        free(current);
57      }
58      break;
59    case 2:
60      p = malloc(sizeof(struct Node));
61      p->c = '_';
62      if (b == 1) {
63        p->next = li->first;
64        li->first = p;
```

```
65      } else {
66        current = li->first;
67        for (i = 1; i < b; ++i) {
68          prev = current;
69          current = current->next;
70        }
71        p->next = current;
72        prev->next = p;
73      }
74      break;
75    case 3:
76      if (b == 1) {
77        return; // Do nothing
78      } else {
79        current = li->first;
80        for (i = 1; i < b; ++i) {
81          prev = current;
82          current = current->next;
83        }
84        li->last->next = li->first;
85        li->first = current;
86        li->last = prev;
87        li->last->next = NULL;
88      }
89      break;
90    default:
91      abort(); // Cannot happen.
92    }
93  }
94
95  int main(void) {
96    int n, m;
97    int i;
98    struct List li = {NULL, NULL};
99
100   scanf("%d␣%d", &n, &m);
101
102   while (fgetc(stdin) != '\n') {}
103   for (i = 0; i < n; ++i) {
104     char c = (char)fgetc(stdin);
```

```
105      push_back(c, &li);
106    }
107    while (fgetc(stdin) != '\n') {}
108
109    for (i = 0; i < m; ++i) {
110      int a, b;
111      scanf("%d␣%d", &a, &b);
112      operation(a, b, &li);
113    }
114    print_list(li);
115
116    return 0;
117  }
```

5　問題 E：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，問題 C と同様に再帰による深さ優先探索を用いれば良い．問題 C の場合と異なり，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（和ではなく）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．

解説で述べられた，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

```
1  #include <stdio.h>
2  #include <limits.h>
3
4  int dfs(int a, int b, int c, int p, int q, int n, int *d,
5          int depth, int *target_chain_id) {
6    int i;
7    int min_cost;
8
9    if (depth == n) {
10     // Check feasibility.
```

```
11      int sum_length[] = {0, 0, 0, 0};
12      int num_chains[] = {0, 0, 0, 0};
13      int cost;
14      for (i = 0; i < n; ++i) {
15        sum_length[target_chain_id[i]] += d[i];
16        num_chains[target_chain_id[i]]++;
17      }
18      if (a < sum_length[0] || b < sum_length[1] || c < sum_length[2]) {
19        // Infeasible: over sized
20        return INT_MAX;
21      }
22      for (i = 0; i < 3; ++i) {
23        if (num_chains[i] == 0) {
24          // No chain.
25          return INT_MAX;
26        }
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 }
```

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）.