# 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 2-1）
# 解説

廣田悠輔
y-hirota@u-fukui.ac.jp

## 目次



## 1 問題 A：簡単な制約付き整数方程式

**解説**

変数は $A, B, C, D$ の 4 個しかなく，それらの取りうる値はいずれも $-M, -M+1, ..., M$ に限定される．したがって，$A, B, C, D$ 取りうる値の組み合わせの数は

$$(2M+1)^4$$

である．例えば，$M = 1$ のとき，取りうる値の組み合わせは

- $A = -1, B = -1, C = -1, D = -1$
- $A = -1, B = -1, C = -1, D = 0$
- $A = -1, B = -1, C = -1, D = 1$
- $A = -1, B = -1, C = 0, D = -1$
- $A = -1, B = -1, C = 0, D = 0$
- $A = -1, B = -1, C = 0, D = 1$
- $A = -1, B = 0, C = -1, D = -1$

$\vdots$

- $A = 1, B = 1, C = 0, D = 1$
- $A = 1, B = 1, C = 1, D = -1$
- $A = 1, B = 1, C = 1, D = 0$
- $A = 1, B = 1, C = 1, D = 1$

の 81 通りとなる．制約

$$1 \leq M \leq 10$$

が与えられているので，$A, B, C, D$ 取りうる値の組み合わせの数は最大でも $21^4$ $(= 194481)$ でしかない．したがって，すべての組み合わせについて方程式の変数に値を代入して等式が成り立つかどうかを調べ，成り立つケースを数え上げれば良い．

すべての組み合わせを網羅する簡単な方法には

- 四重ループにより各変数がとりうる値を網羅する方法，
- 再帰による深さ優先探索

がある．いずれを使っても良い．ただし，前者のような他重ループによる網羅は本問題のように変数が高々数個に固定されている場合には容易に使用できるが，例えば数十個の変数を持つ場合や，変数の数が入力依存である場合には使用困難であることに注意せよ．そのような場合には，深さ優先探索を使用することになる．

回答例（4 重ループ）

```
1  #include <stdio.h>
2
3  int n_solutions(int n, int m) {
4    int a, b, c, d;
5    int count = 0;
6    for (a = -m; a <= m; ++a) {
7      for (b = -m; b <= m; ++b) {
8        for (c = -m; c <= m; ++c) {
9          for (d = -m; d <= m; ++d) {
10           if (a * b + c * d * 2 + 1 == n) {
11             count++;
12           }
13         }
14       }
15     }
16   }
17   return count;
18 }
19
20 int main(void) {
```

```
21    int n, m;
22    scanf("%d", &n);
23    scanf("%d", &m);
24
25    printf("%d\n", n_solutions(n, m));
26
27    return 0;
28  }
```

回答例（再帰）

```
1   #include <stdio.h>
2
3   int n_solutions_recursive(int n, int m, int *x, int depth) {
4     int counter = 0;
5
6     if (depth == 4) {
7       if (x[0] * x[1] + x[2] * x[3] * 2 + 1 == n) {
8         return 1;
9       } else {
10        return 0;
11      }
12    }
13
14    for (x[depth] = -m; x[depth] <= m; ++x[depth]) {
15      counter += n_solutions_recursive(n, m, x, depth + 1);
16    }
17    return counter;
18  }
19
20  int main(void) {
21    int n, m;
22    int x[4];
23    scanf("%d", &n);
24    scanf("%d", &m);
25
26    printf("%d\n", n_solutions_recursive(n, m, x, 0));
27
28    return 0;
29  }
```

## 2　問題 B：フクイ国の両替

**解説**

所持金 $N$ フクをすべてを硬貨の枚数が最小になるように両替したときの，1 フク硬貨，10 フク硬貨，40 フク硬貨，50 フク硬貨，200 フク硬貨の枚数をそれぞれ $a^{(1)}$，$a^{(10)}$，$a^{(40)}$，$a^{(50)}$，$a^{(200)}$ とおく．このときの 1 フク硬貨の枚数が $a^{(1)} \geq 10$ であると仮定する．すると，別の硬貨の組み合わせ「1 フク硬貨 $a^{(1)} - 10$ 枚，10 フク硬貨 $a^{(10)} + 1$ 枚，40 フク硬貨，50 フク硬貨，200 フク硬貨がそれぞれ $a^{(40)}$ 枚，$a^{(50)}$ 枚，$a^{(200)}$ 枚」は，その合計金額が $N$ フクかつ硬貨合計枚数が元の両替の合計枚数より少なくなる．このことは，元の両替の硬貨合計枚数が最小であることに矛盾する．よって，$a^{(1)} \leq 9$ である．同様の議論により，10 フク硬貨，40 フク硬貨，50 フク硬貨の枚数について $a^{(10)} \leq 3$，$a^{(40)} \leq 4$，$a^{(50)} \leq 3$ が成り立つ．一方，硬貨の合計金額が $N$ フクを超えないようにするため $200a^{(200)} \leq N$．したがって，硬貨の合計枚数が最小になる両替を行ったときの各硬貨の枚数は以下の条件を満たす．

$$\begin{aligned} 0 \leq a^{(1)} \leq 9, \quad 0 \leq a^{(10)} \leq 3, \quad 0 \leq a^{(40)} \leq 4, \\ 0 \leq a^{(50)} \leq 3, \quad 0 \leq a^{(200)} \leq \lfloor N/200 \rfloor. \end{aligned} \tag{1}$$

ただし，$\lfloor a \rfloor$ は $a$ 以下の最大の整数（$a$ の床関数値）である．本問題では $N$ は最大で $10^7$ であるので，このような範囲の $a^{(1)}$，$a^{(10)}$，$a^{(40)}$，$a^{(50)}$，$a^{(200)}$ の組み合わせの数は最大 $4N = 4 \times 10^7$ である．したがって，(1) の範囲を全探索する五重ループにより，それぞれの組み合わせについて合計金額が $N$ フクとなるか，$N$ フクとなる場合に硬貨の合計枚数が何枚になるか調べるプログラムを書けば問題に答えられる．

以下に示す追加の考察を行うことで，探索すべき範囲をさらに削減することができる．

- 1 フク硬貨以外をどのように組み合わせても 1 フクから 9 フクの金額を作ることはできない．一方，$0 \leq a^{(1)} \leq 9$ である．したがって，硬貨の枚数が最小となる組み合わせにおける 1 フク硬貨の枚数は $a^{(1)} = N \bmod 10$ と他の硬貨とは独立して決定できる．ただし，$a \bmod b$ は $a$ を $b$ で割った余りである．
- 範囲 (1) のもとで 10 フク硬貨，40 フク硬貨，50 フク硬貨のみを組み合わせて作ることができる 200 フク以上の金額は，200 フク，210 フク，220 フク，...，340 フクの 15 通りである．これらの金額は，200 フク硬貨を 1 枚含む組み合わせによって，より少ない硬貨の合計枚数で作ることができる．したがって，硬貨の枚数が最小となる組み合わせにおける 200 フク硬貨の枚数は $a^{(200)} = \lfloor N/200 \rfloor$ と他の硬貨とは独立して決定できる．

したがって，1 フク硬貨，200 フク硬貨の枚数は上記の方法で独立して決定し，10 フク硬貨，40 フク硬貨，50 フク硬貨の枚数は (1) の範囲を調べる三重ループによって探索することにより，O(1) の計算量で問題を解くことができる．

**コメント**

日本で流通する硬貨（1 円，5 円，10 円，50 円，100 円および 500 円）の場合には貪欲法（高額の硬貨から順にできるだけ多く割り当てる方法）による両替が最小硬貨枚数となるが，フクイ国の硬貨の場合にはそうな

らない．例えば入力 2 のように $N = 485$ のとき，高額硬貨から順にできるだけ多く両替すると，

$$485 = 200 \times 2 + 50 \times 1 + 10 \times 3 + 1 \times 5$$

と 11 枚の硬貨が必要となる．一方

$$485 = 200 \times 2 + 40 \times 2 + 1 \times 5$$

と 50 フク硬貨を使わずに 40 フク硬貨を使った場合，硬貨の合計枚数は 8 枚となる（これは最小値である）．余裕があれば，どのような条件の下で貪欲法が最小硬貨枚数を与えるか考えられたい．

回答例

```c
#include <stdio.h>

int main(void) {
  int n, rest;
  int sum;
  int min_10_40_50;
  int i, j, k;

  scanf("%d", &n);

  sum = n / 200; // 200-Fuku coin
  rest = n % 200;

  min_10_40_50 = 9999; // Huge value
  for (i = 0; i <= 3; i++) { // Num. 10-Fuku
    for (j = 0; j <= 4; j++) { // Num. 40-Fuku
      for (k = 0; k <= 3; k++) { // Num. 50-Fuku
        if (i * 10 + j * 40 + k * 50 == (rest / 10) * 10) {
          if (i + j + k < min_10_40_50) {
            min_10_40_50 = i + j + k;
          }
        }
      }
    }
  }
  sum += min_10_40_50;

  rest %= 10;
  sum += rest; // 1-Fuku coin

```

```
31    printf("%d\n", sum);
32
33    return 0;
34  }
```

## 3 問題 C：出張費用最小化

**解説**

直接移動可能な都市間の移動費用を都市間の距離とみなして，最短経路問題として扱う．

最短経路問題を効率よく方法としてよく知られた Dijkstra 法によりある都市からある都市への最小移動費用を求めると，必要な計算量は $\mathrm{O}(N^2)$ から $\mathrm{O}((N+M)\log N)$ となる（実装法により変わる）．最小移動費用を求めるべき問合せは $L$ 回あるので，そのたびに Dijkstra 法により最小移動費用を求めると，全体の計算量は $\mathrm{O}(LN^2)$ から $\mathrm{O}(L(N+M)\log N)$ となる．$N, M, L$ はそれぞれ高々 $10^2, 10^4, 10^2$ であるので，この方法を使って制限時間内に回答可能である．

問合せ回数 $L$ の値がもっと大きく（例えば $L = 10^5$ であるような）制限時間にそれほど余裕がないという状況を考えよう．このような場合には，すべての都市間の最小移動費用をあらかじめ何らかの方法で調べておき，$L$ 回の問合せの際にはその結果を毎回参照するという方法が考えられる．最多経路問題のすべてのノード間の距離（この問題てはすべての都市間の最小移動費用）を調べる方法に Warshall-Floyd 法 [2, Sec. 2.5] がある．Warshall-Floyd 法を用いると，すべての都市間の最小移動費用が $\mathrm{O}(N^3)$ の計算量で求まる．以後は問い合わせがあるたびに結果を参照する（$\mathrm{O}(1)$ の操作）だけで最小移動費用を答えられる．したがって，全体の計算量は $\mathrm{O}(N^3 + L)$ となり，$L$ が大きな値である場合に計算量を抑えられる．

**回答例（Warshall-Floyd 法）**

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <limits.h>
4
5   int distance(int **a, int const i, int const j)
6   {
7     return (i > j) ? a[j][i] : a[i][j];
8   }
9
10  /* Find (i, j) shortest distances for all 0 <= i, j < n. */
11  void warshall_floyd(int const n, int **a)
12  {
13    int i, j, k;
14
15    for (k = 0; k < n; ++k) {
16      /* Update */
```

```
17      for (i = 0; i < n; ++i) {
18        for (j = i + 1; j < n; ++j) {
19          int const candidate = distance(a, i, k) + distance(a, k, j);
20          if (a[i][j] > candidate) {
21            a[i][j] = candidate;
22          }
23        }
24      }
25    }
26  }
27
28  int main(void)
29  {
30    int n, m, l;
31    int **a;
32    int i, j, k;
33
34    scanf("%d", &n);
35    scanf("%d", &m);
36    scanf("%d", &l);
37
38    a = malloc(sizeof(int *) * n);
39    for (i = 0; i < n; ++i) {
40      a[i] = malloc(sizeof(int) * n);
41    }
42
43    for (i = 0; i < n; ++i) {
44      for (j = 0; j < i; ++j) {
45        // Lower part elements are not used.
46        a[i][j] = -1;
47      }
48    }
49    for (i = 0; i < n; ++i) {
50      // Self
51      a[i][i] = 0;
52    }
53    for (i = 0; i < n; ++i) {
54      for (j = i + 1; j < n; ++j) {
55        // Default values (no direct path between i and j).
56        a[i][j] = INT_MAX / 4; // Divided by 4 to avoid overflow.
```

```
57      }
58    }
59
60    // Problem setting.
61    for (k = 0; k < m; ++k) {
62      int ii, jj, d;
63      scanf("%d␣%d␣%d", &ii, &jj, &d);
64      a[ii][jj] = d;
65    }
66
67    // Solve
68    warshall_floyd(n, a);
69
70    // Query
71    for (int k = 0; k < l; ++k) {
72      int ii, jj;
73      scanf("%d␣%d", &ii, &jj);
74      printf("%d\n", distance(a, ii, jj));
75    }
76
77    for (i = 0; i < n; ++i) {
78      free(a[i]);
79    }
80    free(a);
81    return EXIT_SUCCESS;
82  }
```

## 4　問題 D：最大の円

**解説**

線分が $n$ 本与えられるの条件を満たす最大半径は，図 1 のように線分が一本ずつ選んだけときの円の最大半径を調べ，その最小値をとることで求められる．一見して非常に複雑な問題に見えるが，実は一本の線分と一つの円だけがある場合を考えれば良いのである．

線分が一本だけ存在するときの円の最大半径の調べ方について述べる．このときの円の最大半径は円の中心から線分までの距離に一致する．円の中心から線分までの距離は，円の中心と線分の座標の位置関係により 3 つに場合分けして考える．

- 円の中心が線分の端点を通り線分に垂直となる 2 本の直線の間になく，線分の片側の端点（以下，$P_1$ と呼ぶ）により近い側にあるとき（図 2 の下側），円の中心と線分の距離は円の中心から $P_1$ の長さとなる．

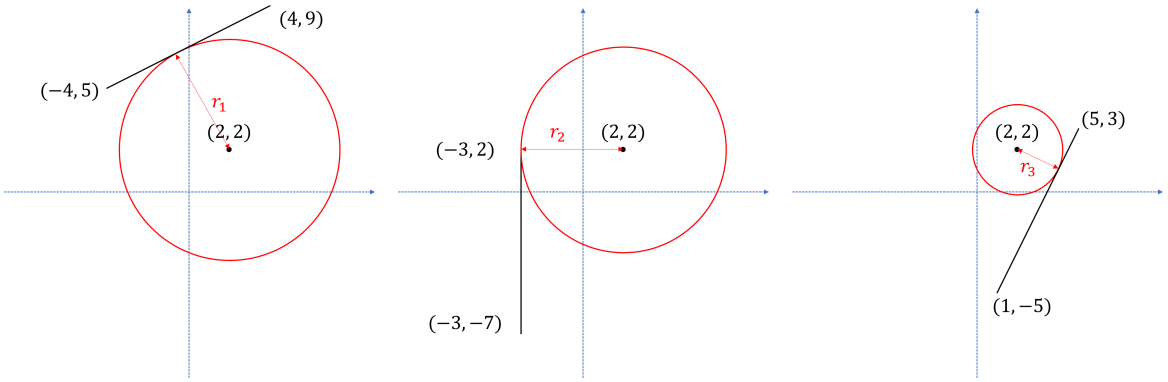


図 1　問題文の入力例 1 で線分を一本ずつ選んだときの最大半径の円（赤）.

- 円の中心が 2 本の直線の間になく，線分の端点のうち $P_1$ でない点（以下，$P_2$ と呼ぶ）により近い側にあるとき（図 2 の上側），円の中心と線分の距離は円の中心から $P_2$ の長さとなる.
- 円の中心が 2 本の直線の間（図 2 の 2 本の緑破線の間）にある場合，円の中心と線分の距離は円の中心から線分に向かう垂線の長さとなる.

あとは，それぞれの場合に分けて円の中心から線分の距離を求めれば良い（回答例 11–30 行目）. 具体的な場合分けの方法，距離の求め方は初等幾何の内容であるので省略する. 回答例では線分の片側の端点から円の中心に向かうベクトルおよび線分の同じ端点からもう一方の端点に向かうベクトルの内積や外積を用いて計算している.

なお，回答例に示すプログラムでは浮動小数点数を使用して計算を行っているが，扱うべき座標の最大値が 100 程度であるのに対して出力すべき値は小数点第 2 位までであるので，倍精度浮動小数点数（double 型）を用いれば精度は十分である.

ノート

- 本問題は比較的シンプルな問題なのでその必要はないが，複雑な幾何問題ではすべてを自分で実装するのではなくライブラリを用いるのが容易である. C++ であれば `boost::geometry`（C++ STL ではないので注意），Java であれば `java.awt.geom` パッケージなどが使用できる. 自分の目的に応じて適切なものを使用されたい.

回答例

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <float.h>
4  #include <math.h>
5
6  struct segment {
7    /* Line segment (x, y) -- (v, w) */
8    double x, y, v, w;
```

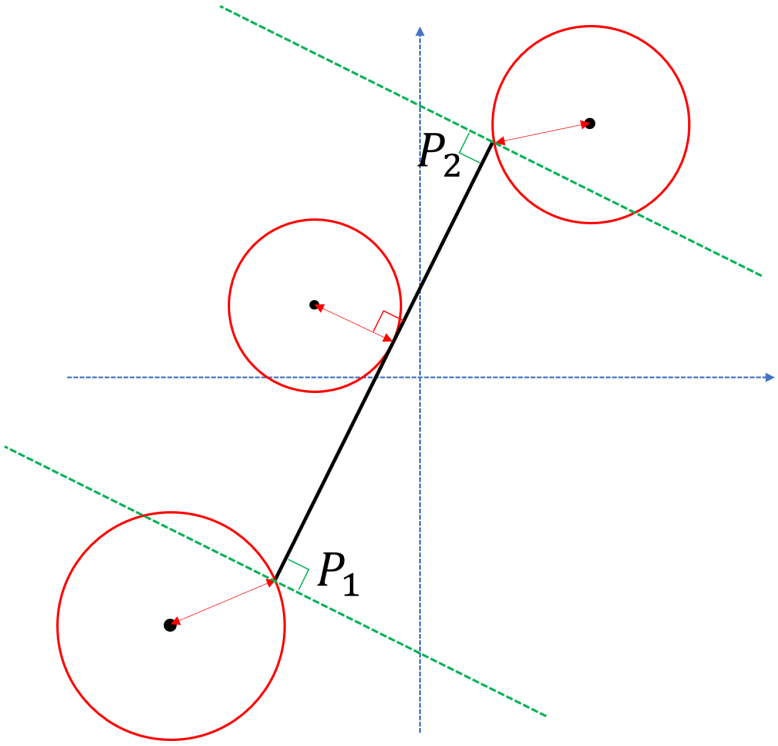


図 2　線分（黒線分）と円の中心（黒点）の位置と最大円の半径（両側赤矢印付き線分の長さ）の関係．緑破線は線分の端点を通り線分に垂直な直線．

```
 9 };
10
11 double segment_distance(struct segment const s, double const a, double
       const b)
12 {
13   double inner_prod1, inner_prod2, outer_prod;
14
15   /* If the nearest point is (x, y). */
16   inner_prod1 = (a - s.x) * (s.v - s.x) + (b - s.y) * (s.w - s.y);
17   if (inner_prod1 <= 0.0) {
18     return sqrt((s.x - a) * (s.x - a) + (s.y - b) * (s.y - b));
19   }
20
21   /* If the nearest point is (v, w). */
22   inner_prod2 = (a - s.v) * (s.x - s.v) + (b - s.w) * (s.y - s.w);
23   if (inner_prod2 <= 0.0) {
24     return sqrt((a - s.v) * (a - s.v) + (b - s.w) * (b - s.w));
25   }
26
```

```
27    /* Otherwise, the nearest point is on the line segment. */
28    outer_prod = (a - s.x) * (s.w - s.y) - (s.v - s.x) * (b - s.y);
29    return fabs(outer_prod) / sqrt((s.v - s.x) * (s.v - s.x) + (s.w - s.y
          ) * (s.w - s.y));
30  }
31
32  double solve(int const n, struct segment const * const s, double const
       a, double const b)
33  {
34    int i;
35    double min_distance = DBL_MAX;
36    for (i = 0; i < n; ++i) {
37      double const distance = segment_distance(s[i], a, b);
38      if (distance < min_distance) min_distance = distance;
39    }
40    return min_distance;
41  }
42
43  int main(void)
44  {
45    int n;
46    struct segment s[10];
47    double a, b;
48    int i;
49
50    scanf("%d", &n);
51    for (i = 0; i < n; ++i) {
52      scanf("%lf␣%lf␣%lf␣%lf", &s[i].x, &s[i].y, &s[i].v, &s[i].w);
53    }
54
55    scanf("%lf␣%lf", &a, &b);
56
57    printf("%.2f\n", solve(n, s, a, b));
58
59    return EXIT_SUCCESS;
60  }
```

# 5　問題 E：Keep Distance

**解説**

本問題は POJ 2456 “Agressive Cow”[1][*1]を参考に作成された類題である．

本問題を解く前に，まず「全員の距離を $L$ 以上離して $M$ 人全員を着席させられるか」というより簡単な問題を考える．例えば，

```
**#**#********#**#**#**#**
```

という配置の椅子に対して，6 以上の距離を開けて 4 人が着席可能かとった具合である．この可否は以下の手順により判断可能である．

1. 最初の椅子に 1 名を着席させる．
2. 現在着席している最も右側の椅子よりも右にあり，現在着席している最も右側の椅子から $L$ 以上離れたもっとも左側にある椅子に 1 名を着席させる．
3. 全員が着席するか，着席可能な椅子がなくなるまで手順 2 を繰り返す．
4. 全員が着席出来たら可能であり，着席可能な椅子がなくなったら不可能であると判断できる．

例えば，前述の例であれば，以下の `@` の位置に 3 人が着席させられるが 4 人目が着席可能な椅子が存在せず，そのような配置が不可能だとわかる．

```
**@**#********@**#**#**@*#*
```

この手順は椅子の位置（座標）をあらかじめ配列に格納しておけば $O(N)$ の計算量で実行可能である．

先述の簡単化された問題が解ければ，元の問題も容易に解ける．簡単化された問題「全員の距離を $L$ 以上離して $M$ 人全員を着席させられるか」の距離を $L = 1, 2, ...$ を順に増大させて解き，全員着席可能な最大の $L$ が求めるべき答えとなる．文字列の長さは $|S|$ であるので，調べるべき距離 $L$ の最大値は $|S| - 1$ である．したがって，この方法で問題を解いたときの計算量は $O(N|S|)$ となる．文字列の長さは $|S| \leq 1000$，椅子の数は $N \leq 1000$ という制約があるので，時間内に回答可能である．

参考までにより難しいケースとして，例えば文字列の長さが $|S| = 10^6$ のように大きな値で，前述の方法では実行時間に余裕がない場合を考える．前述の方法では，$L$ の値を $1, 2, ...$ と順に増大させ，逐一その $L$ の値で着席が可能か否かを判定していた．しかしながら，この部分は二分探索を使えば探索回数（判定回数）を $O(\log |S|)$ 回に削減できる．例えば，$L = 1$ で着席可能，$L = |S| - 1$ で着席不能であったとする．このとき，着席可能な $L$ の最大値は $1 \leq L \leq |S| - 1$ の範囲に存在する．次にその中間値である $L = |S|/2$ で着席可能であるかを調べ，着席可能ならば着席可能な $L$ の最大値は $|S|/2 \leq L \leq |S| - 1$ の範囲に存在し，着席不可能ならば着席可能な $L$ の最大値は $1 \leq L \leq |S|/2$ の範囲に存在するとわかる．今度は新しい範囲の中間値を $L$ として着席可能であるかを調べ，同様に最大値の存在範囲を半分に絞り込む．これを $\log_2(|S| - 1)$ 繰り返せば着席可能な最大の $L$ に辿りづける．結果として，この方法で問題を解いたときの計算量は $O(N \log |S|)$ となる．

**回答例（二分探索）**

[*1] 日本語訳が [2, Sec. 3.1] に掲載されている．

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

int num_available_seat(int const n, int const k, int const * const pos)
{
  int i;
  int num_available = 1; /* Number of available seats */
  int last_pos = pos[0];
  for (i = 1; i < n; ++i) {
    if (pos[i] - last_pos >= k) {
      last_pos = pos[i];
      num_available++;
    }
  }
  return num_available;
}

int find_max_interval(int const n, int const m, int const * const pos)
{
  /* Binary search search. */
  int left, right, mid;

  left = 1; /* Feasible interval */
  right = pos[n - 1] - pos[0] + 1; /* Infeasible interval */
  for (mid = (left + right) / 2; mid != left; mid = (left + right) / 2)
      {
    if (num_available_seat(n, mid, pos) >= m) {
      /* Interval mid is feasible. */
      left = mid;
    } else {
      /* Interval mid is infeasible. */
      right = mid;
    }
  }
  return mid;
}

#define MAX_STRING_LENGTH (1000)
```

```
39
40 int main(void)
41 {
42   int m, n;
43   char s[MAX_STRING_LENGTH + 2]; /* The main string, '\n', and '\0' */
44   int pos[MAX_STRING_LENGTH]; /* Only first n elements are used. */
45   int i;
46   int result;
47
48   fgets(s, MAX_STRING_LENGTH + 2, stdin);
49   scanf("%d", &m);
50
51   n = 0;
52   for (i = 0; s[i] != '\0'; ++i) {
53     if (s[i] == '#') {
54       /* n-th chair is placed on the position i. */
55       pos[n] = i;
56       n++;
57     }
58   }
59
60   /* Solve */
61   result = find_max_interval(n, m, pos);
62
63   printf("%d\n", result);
64
65   return EXIT_SUCCESS;
66 }
```

## 参考文献


[1] PKU JudgeOnline, "Aggressive cows", http://poj.org/problem?id=2456 (accessed on 2022-05-22).

[2] 秋葉拓哉，岩田陽一，北川宜稔，「プログラミングコンテストチャレンジブック: 問題解決のアルゴリズム活用力とコーディングテクニックを鍛える」，第 2 版，マイナビ出版，2012.