9 配列

複数のものに添字番号付きの名前を付けると、それらを系統的に表せることがある。例えば数学で、一般的な数列を $\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \ldots$ という風に表したりする。

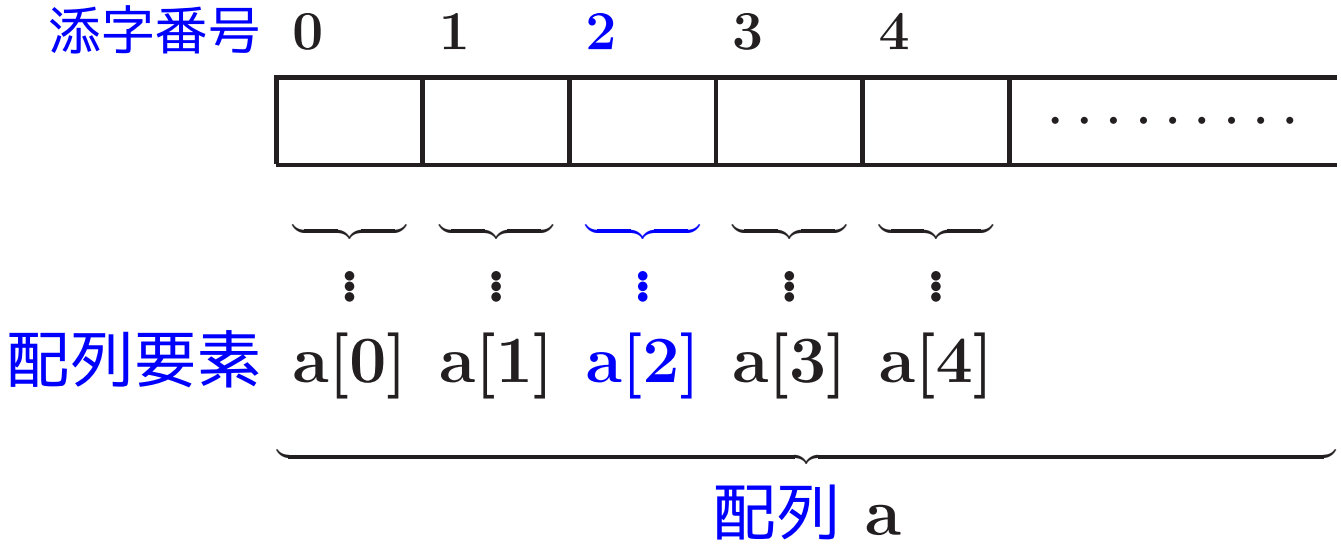
9–1 一次元配列を用いた計算

同じデータ型の領域を連続的に並べたものを一般に配列と呼び、その中の個々のデータ領域を配列要素と呼ぶ。

配列を使えば 大量の同種のデータを規則的に並べて格納し、その中の各々のデータに対して同じ処理を繰り返すことが簡単にできるので、大抵のプログラミング言語で配列が使えるようになっている。

特にC言語においては、

配列の先頭に位置する要素から順に 0, 1, 2, 3, ... という添字番号が各々の配列要素に割り振られ、配列 a の中の添字番号が k の配列要素は a[k] と表される。



例題9. 1 (平均と分散)　50個の実数データ $x_0, x_1, x_2, \cdots, x_{49}$ を読み込み、それらの平均 μ と分散 V を定義式

$$\mu = \frac{1}{50}(x_0 + x_1 + x_2 + \cdots + x_{49})$$

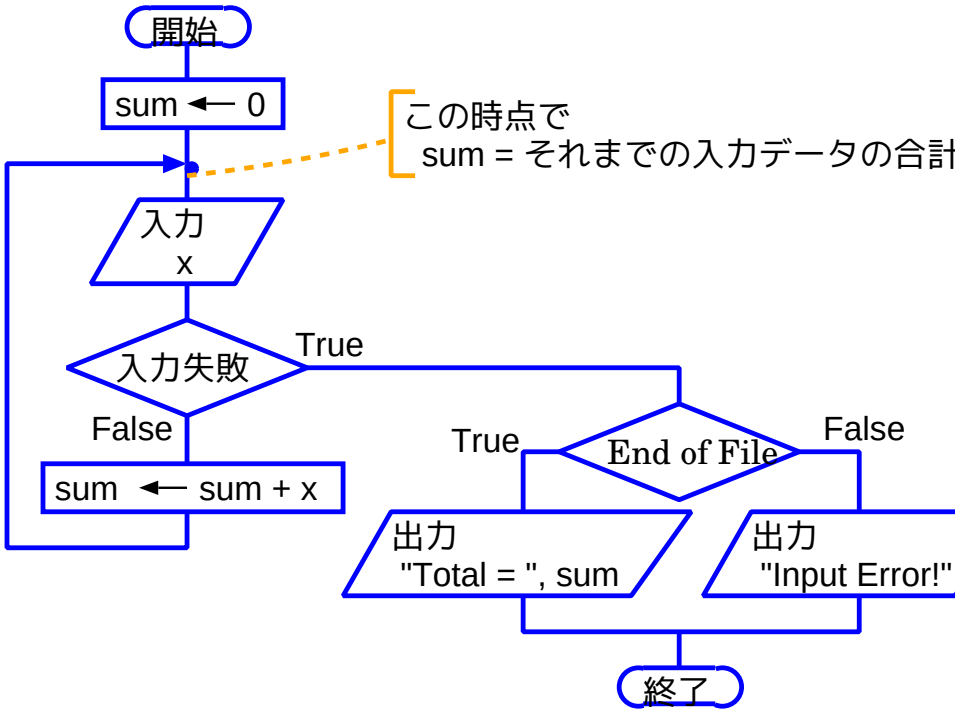
$$V = \frac{1}{50}\sum_{i=0}^{49}(x_i - \mu)^2$$

に従って求めて出力するCプログラムを作成せよ。

(考え方)　平均 μ を計算するだけなら、例題6.19で行ったように、読み込んだデータを保持する変数を1個だけ用意し、

- そこへのデータ読み込みと、
- 読み込んだデータを別の累算値を保持する変数に加える作業

を交互に繰り返せばよい。



$$\mu = \frac{1}{50}(\mathbf{x_0} + \mathbf{x_1} + \mathbf{x_2} + \cdots + \mathbf{x_{49}})$$
$$\mathbf{V} = \frac{1}{50}\sum_{\mathbf{i=0}}^{\mathbf{49}}(\mathbf{x_i} - \mu)^{\mathbf{2}}$$

しかし、
指定された式に従って分散 $\mathbf{V}$ を計算するなら 、
$\mathbf{V}$ の計算には平均 μ の計算結果が必要になるので、分散 $\mathbf{V}$ の計算で指定された累算をデータの読み込みと並行して行うわけにはいかない。

⇨ 読み込んだ50個の実数データ $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{49}}$ は
全て保持しておく必要がある。

⇨ これらのデータ保持に配列を用いる。

(プログラミング)

```
[motoki@x205a]$  nl average-variance.c   Enter
```

(注釈は省略)

```
 6  #include <stdio.h>

 7  int main(void)
 8  {
 9    int      i;
10    double   x[50], ave, var;

11    ave = 0.0;
12    for (i=0; i<50; ++i) {
13      scanf("%lf", &x[i]);
14      ave += x[i];
15    }
16    ave /= 50.0;
```

```
    17    var = 0.0;
    18    for (i=0; i<50; ++i)
    19      var += (x[i]-ave)*(x[i]-ave);
    20    var /= 50.0;

    21    printf("\nInput data:\n");
    22    for (i=0; i<50; i+=5)
    23      printf("%14.5e%14.5e%14.5e%14.5e%14.5e\n",
    24             x[i], x[i+1], x[i+2], x[i+3], x[i+4]);
    25    printf("\nAverage  = %14.6g\n"
    26           "Variance = %14.6g\n", ave, var);
    27    return 0;
    28  }
[motoki@x205a]$
```

```
[motoki@x205a]$  cat average-variance.data   Enter
1.0000  1.0001  1.0002  1.0003  1.0004
1.0005  1.0006  1.0007  1.0008  1.0009
1.0010  1.0011  1.0012  1.0013  1.0014
1.0015  1.0016  1.0017  1.0018  1.0019
1.0020  1.0021  1.0022  1.0023  1.0024
1.0025  1.0026  1.0027  1.0028  1.0029
1.0030  1.0031  1.0032  1.0033  1.0034
1.0035  1.0036  1.0037  1.0038  1.0039
1.0040  1.0041  1.0042  1.0043  1.0044
1.0045  1.0046  1.0047  1.0048  1.0049
[motoki@x205a]$  gcc average-variance.c   Enter
[motoki@x205a]$  ./a.out < average-variance.data   Enter

Input data:
   1.00000e+00   1.00010e+00   1.00020e+00   1.00030e+00   1.0
   1.00050e+00   1.00060e+00   1.00070e+00   1.00080e+00   1.0
```

```
   1.00100e+00   1.00110e+00   1.00120e+00   1.00130e+00   1.0
   1.00150e+00   1.00160e+00   1.00170e+00   1.00180e+00   1.0
   1.00200e+00   1.00210e+00   1.00220e+00   1.00230e+00   1.0
   1.00250e+00   1.00260e+00   1.00270e+00   1.00280e+00   1.0
   1.00300e+00   1.00310e+00   1.00320e+00   1.00330e+00   1.0
   1.00350e+00   1.00360e+00   1.00370e+00   1.00380e+00   1.0
   1.00400e+00   1.00410e+00   1.00420e+00   1.00430e+00   1.0
   1.00450e+00   1.00460e+00   1.00470e+00   1.00480e+00   1.0

Average  =         1.00245
Variance =     2.0825e-06
[motoki@x205a]$
```

9–2 整列化

コンピュータ内に蓄えられた(大量の)データをある指定された順序に並べ直す作業は整列化と呼ばれ、...

特に整列化が配列と密接な関係がある訳ではないが、配列に蓄えられたデータの処理をする代表的な問題として、次に整列化を考えよう。

例題9. 6 (バブル整列法)　50個の整数データを読み込み、それらを小さい順に出力するCプログラムを作成せよ。

(考え方)

50個の整数データは最初に全て配列 a[0]～a[49] に読み込む。

ここでは、実用的ではないが最も簡単な部類に属するバブル整列法を紹介しよう。

バブル整列法では、

大小順が逆になっている隣り同士の要素を交換する、

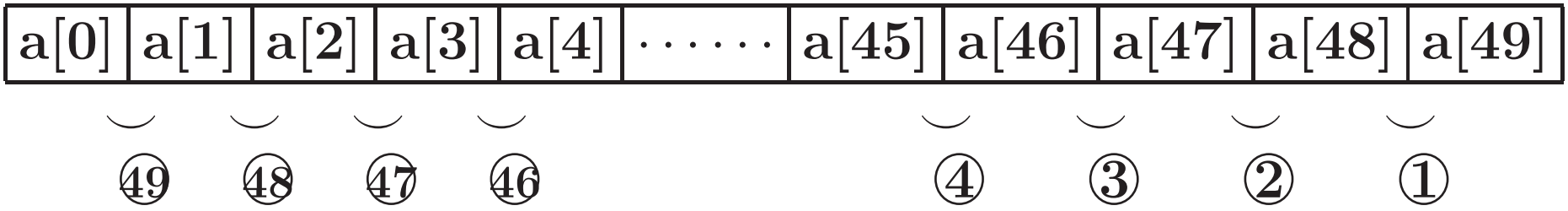
という操作を繰り返す。

まず 第1段階 として、

a[49] と a[48], a[48] と a[47], a[47] と a[46], ...

....., a[4] と a[3], a[3] と a[2], a[2] と a[1], a[1] と a[0]

という組に対して、順に「逆順になっていれば交換」という操作を行う。



これで、あたかも (軽い) 泡が水面に出て来るように、全体の中で最小の要素が一つずつ配列の先頭方向に移動し、最後に a[0] に最小要素が入る。

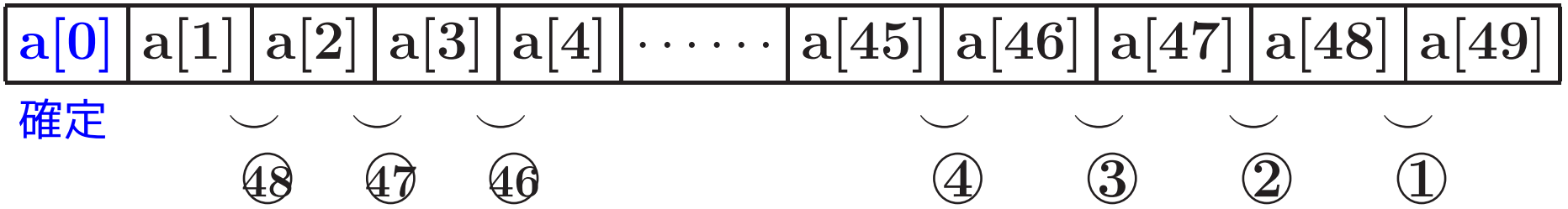
あとは a[1] ~ a[49] を小さい順に並べ直せば良いので、

次に 第2段階 として、

a[49] と a[48], a[48] と a[47], a[47] と a[46], ...

....., a[4] と a[3], a[3] と a[2], a[2] と a[1]

という組に対して、順に「逆順になっていれば交換」という操作を行う。



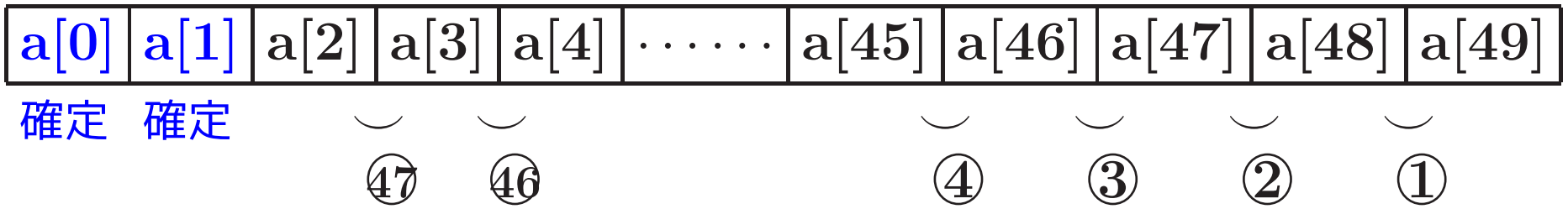
これで a[1] に2番目に小さな要素が入るので、

第3段階 としては、

a[49] と a[48], a[48] と a[47], a[47] と a[46], ...

....., a[4] と a[3], a[3] と a[2]

という組に対して、順に「逆順になっていれば交換」という操作を行う。

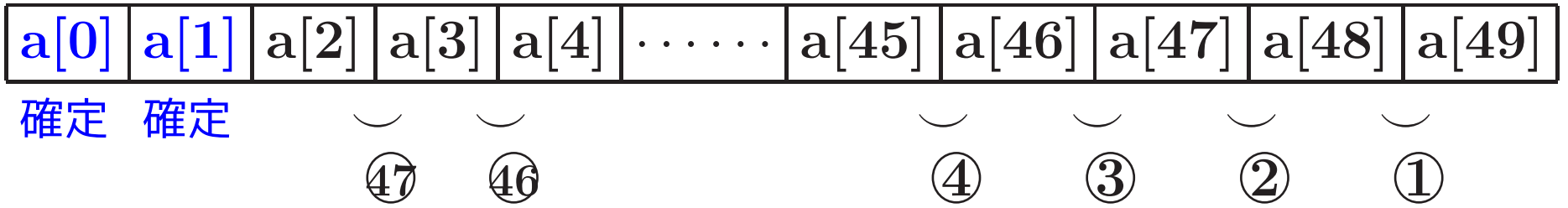


第3段階 としては、

a[49] と a[48], a[48] と a[47], a[47] と a[46], ...

....., a[4] と a[3], a[3] と a[2]

という組に対して、順に「逆順になっていれば交換」という操作を行う。



これで a[2] に3番目に小さな要素が入り、a[0] ~ a[2] の部分は最終的に確定した要素が入ったことになる。

以下 、

a[0] ~ a[49] の部分に最終的に確定した要素が入るまで、

この手順を続けるだけである。

(プログラミング)

```
[motoki@x205a]$  nl bubblesort.c   Enter
     1 /* 50個の整数データを読み込み、それらを小さい順に出力する *
     2 /* Cプログラム　(バブル整列法)

     3 #include <stdio.h>
     4 #define  SIZE  50

     5 int main(void)
     6 {
     7   int a[SIZE], temp, k, i;

     8   for (k=0; k<SIZE; k++)
     9     scanf("%d", &a[k]);
```

```
10    printf("整列前：");
11    for (k=0; k<SIZE; k++) {
12      if (k%5 == 0)
13        printf("\n");
14      printf("__%11d", a[k]);
15    }
16    printf("\n");

17    for (k=0; k<SIZE-1; k++) {
18      for (i=SIZE-1; i > k; i--)
19        if (a[i-1] > a[i]) {        /* a[i-1] と a[i]    */
20          temp   = a[i-1];        /* の大小を調べて、  */
21          a[i-1] = a[i];          /* 逆順なら交換する。*/
22          a[i]   = temp;
23        }
24    }
```

```
25    printf("整列後：");
26    for (k=0; k<SIZE; k++) {
27      if (k%5 == 0)
28        printf("\n");
29      printf("__%11d", a[k]);
30    }
31    printf("\n");
32    return 0;
33 }
```

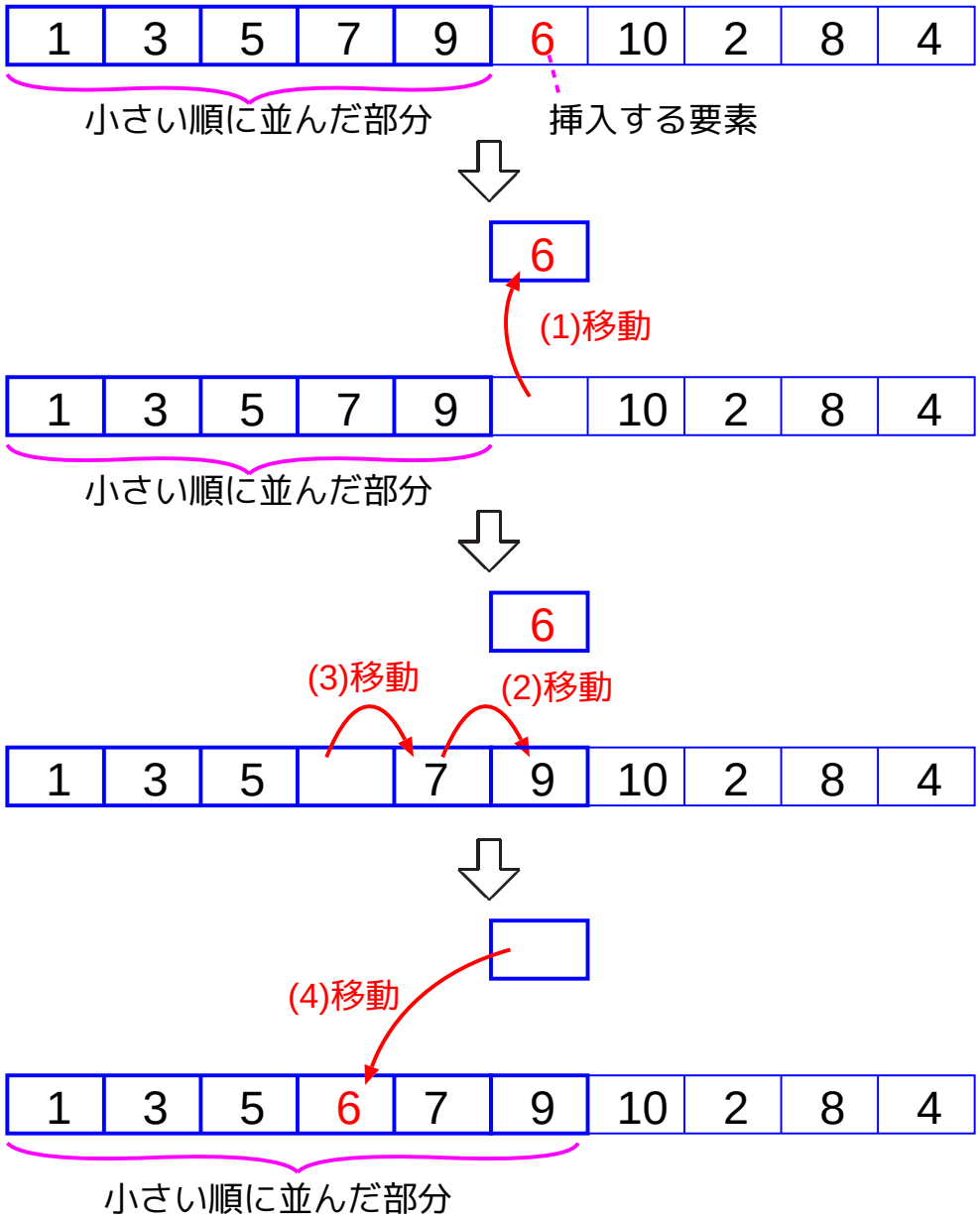
```
[motoki@x205a]$  cat bubblesort.data  Enter
-1234   1000   3456  -7890     11    0   3333   6666   9876   4860
    1      2      4      9      8    7      6      5      0      3
  -45    -95    333  11111   7890   34   5555    234    784  -7420
 5893  -3099  77777  88888    452   99    470   -999   -672    -13
-2222  12345  -6789  -9876     -5   -7   -400   5505   1980  12800
[motoki@x205a]$  gcc bubblesort.c  Enter
[motoki@x205a]$  ./a.out < bubblesort.data  Enter
```

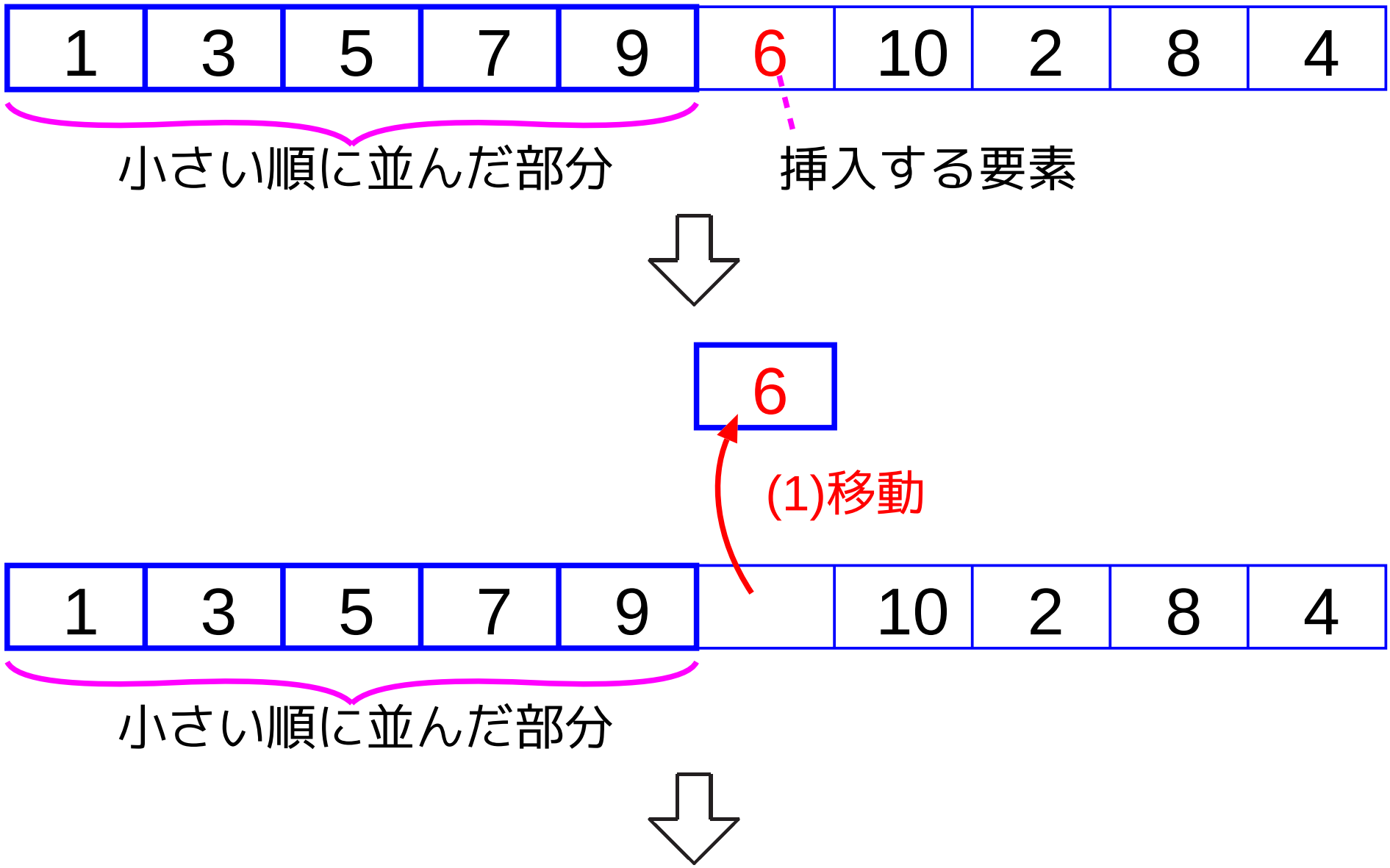
```
整列前：
      -1234        1000        3456       -7890
          0        3333        6666        9876
                          (途中省略)
         -7        -400        5505        1980
整列後：
      -9876       -7890       -7420       -6789
      -2222       -1234        -999        -672
        -95         -45         -13          -7
          0           0           1           2
          4           5           6           7
          9          11          34          99
        333         452         470         784
       1980        3333        3456        4860
       5555        5893        6666        7890
      11111       12345       12800       77777
[motoki@x205a]$
```

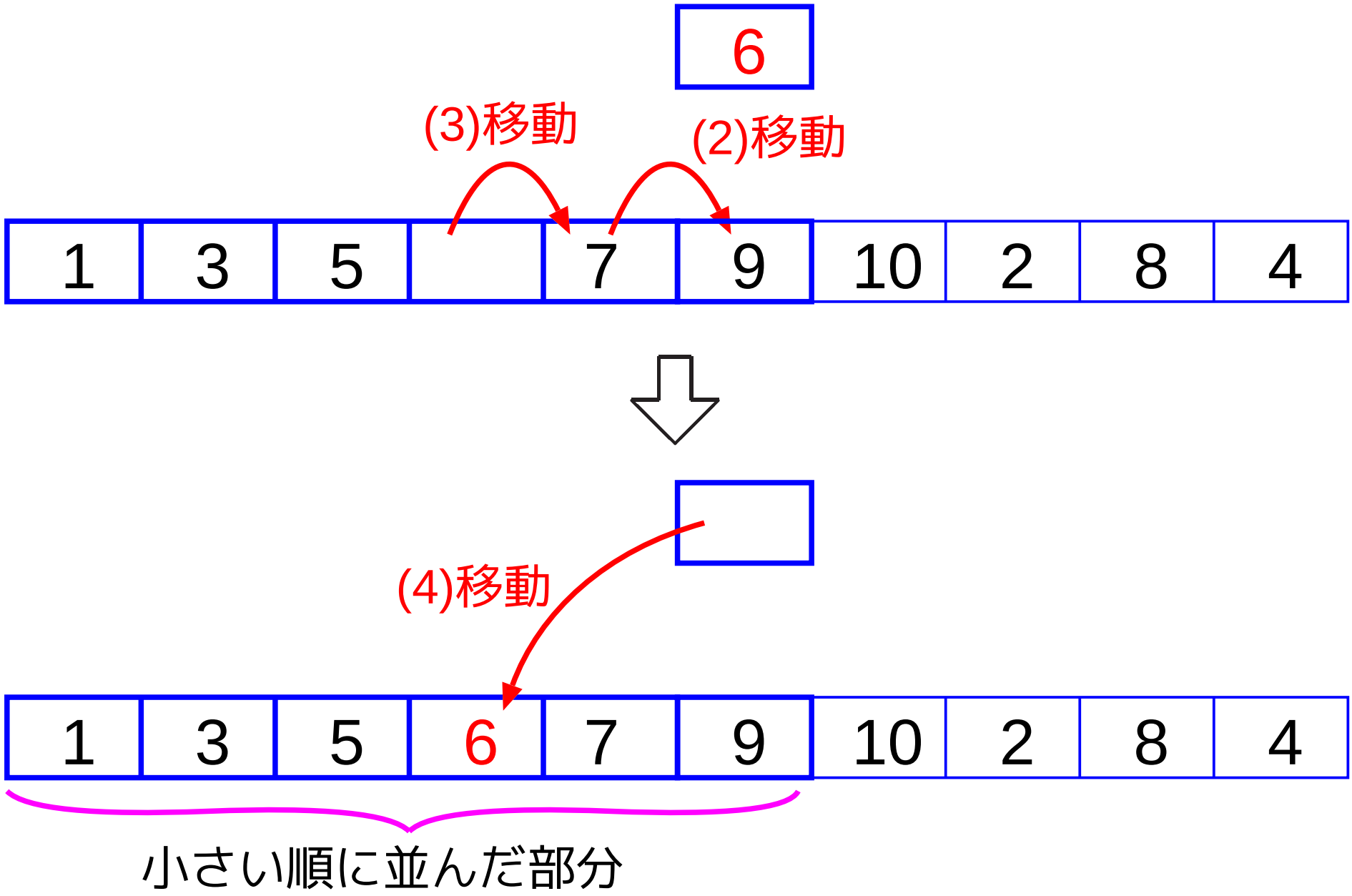
□演習9. 7 (整列化) 50個の整数データを読み込み、それらを小さい順に出力するCプログラムを作成せよ。 但し、ここでは、上の例題9.6 で紹介したバブル整列法を使うのではなく、自分の頭だけで整列化を行うアルゴリズムを考え出してみて下さい。

Hint どうすれば良いか分からない人のために、直感的に分かり易い整列化手法を2つ例示します。

(整列化手法1,挿入法)　小さい順に並んだデータ列に別のデータを然るべき場所に挿入する操作を繰り返す。すなわち、次の様な操作を繰り返す。

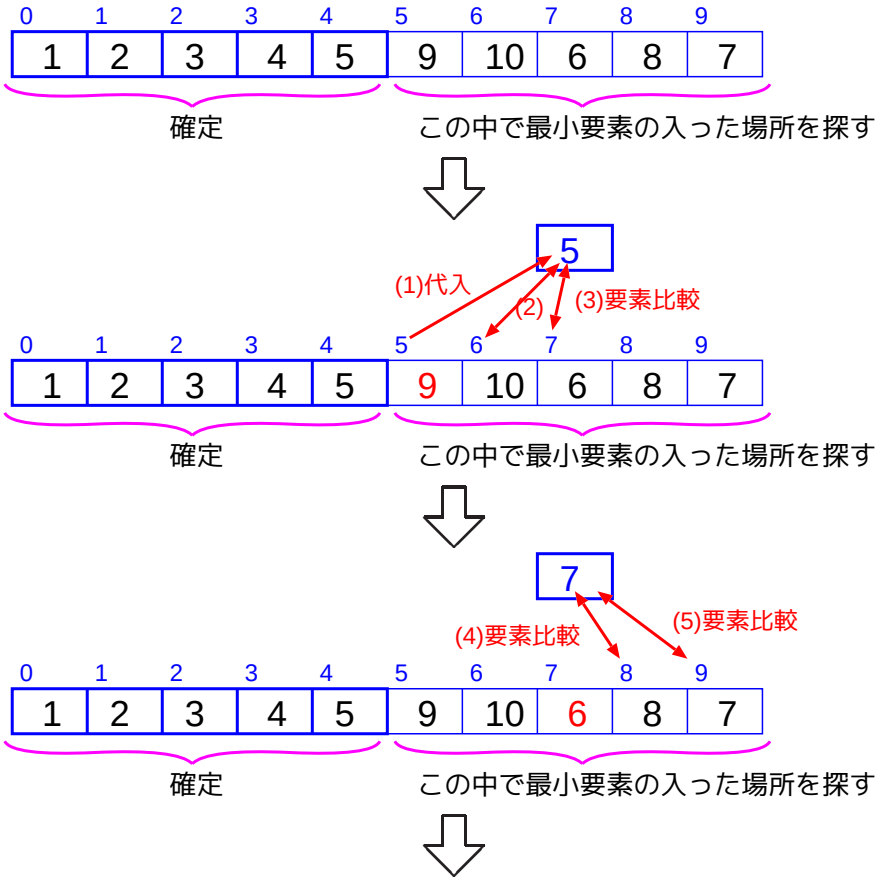


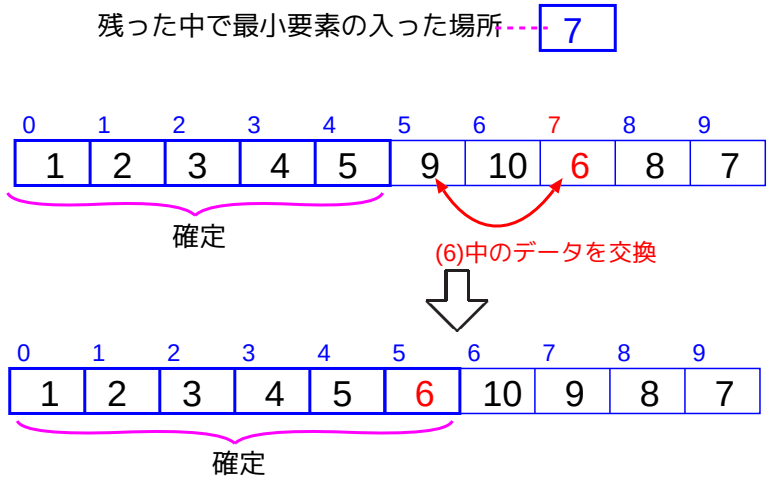

1
3
5
7
9
6
10
2
8
4
小さい順に並んだ部分
挿入する要素
6
(1)移動
1
3
5
7
9
10
2
8
4
小さい順に並んだ部分

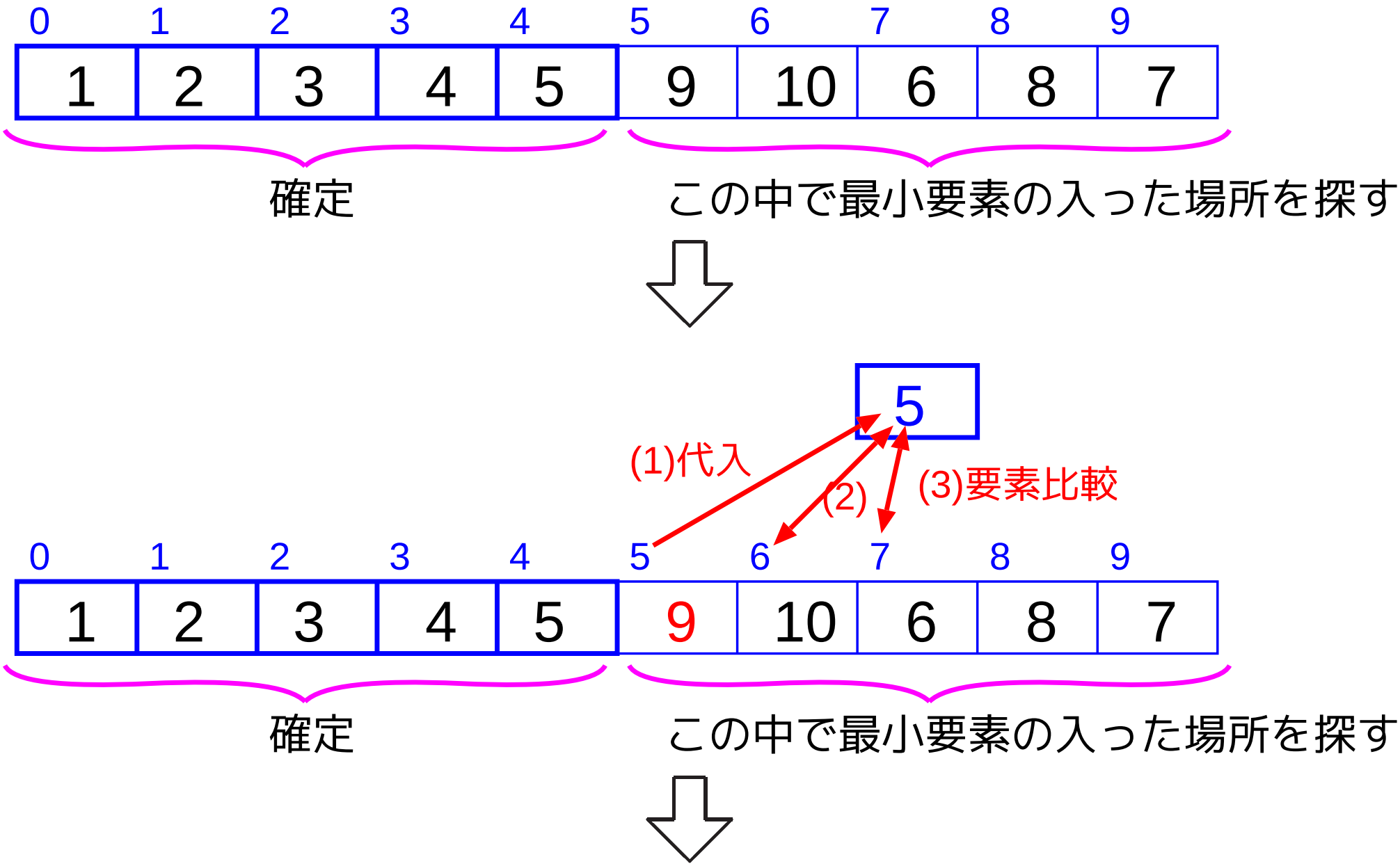

6
(3)移動
(2)移動
1
3
5
7
9
10
2
8
4
(4)移動
1
3
5
6
7
9
10
2
8
4
小さい順に並んだ部分

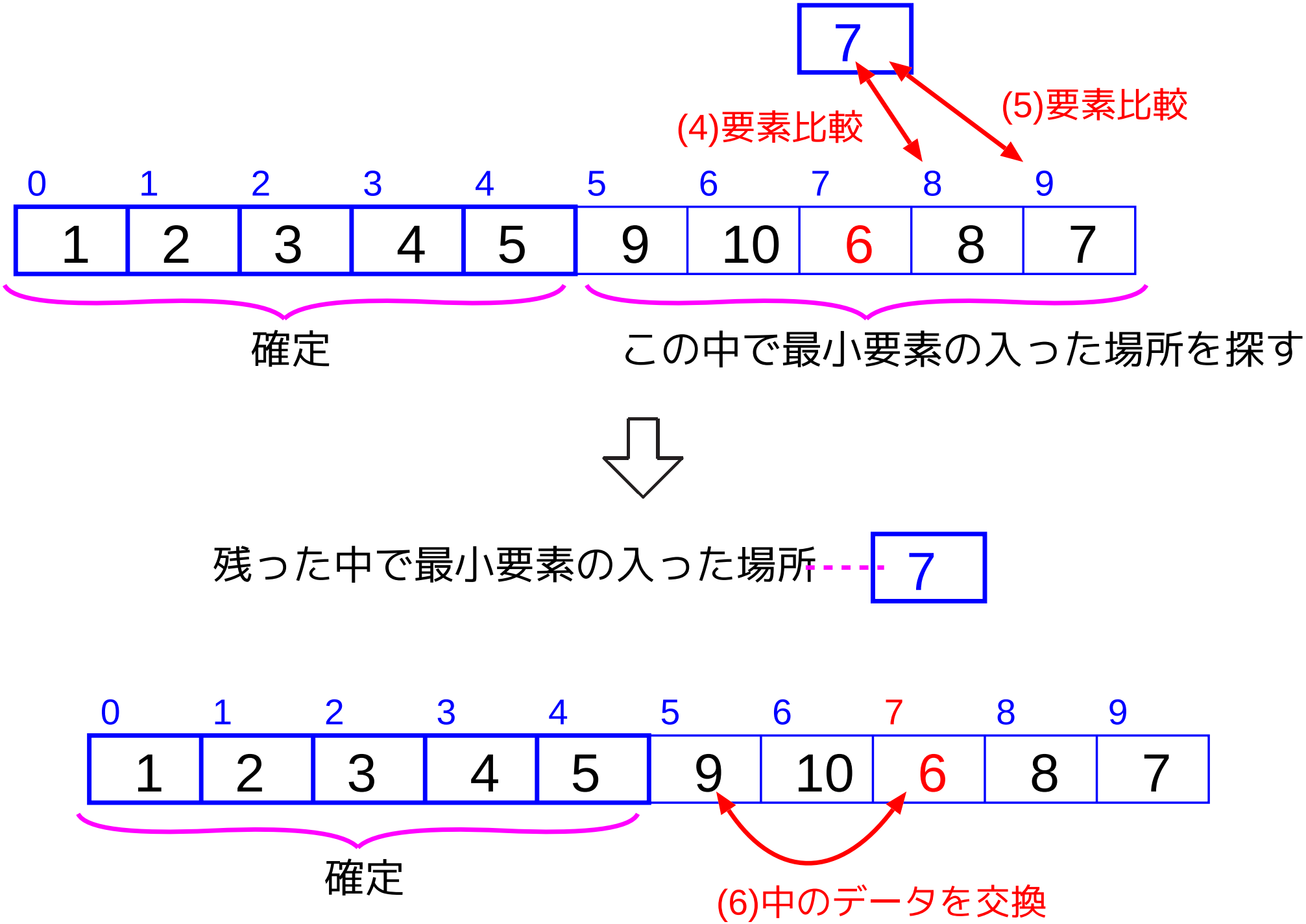
(整列化手法2,最小値選択法)　最小データ、2番目に小さいデータ、3番目に小さいデータ、...と順に見つけ出していき、見つけ出す度にそのデータを然るべき場所に移す。この方法における基本操作は次の2つ。

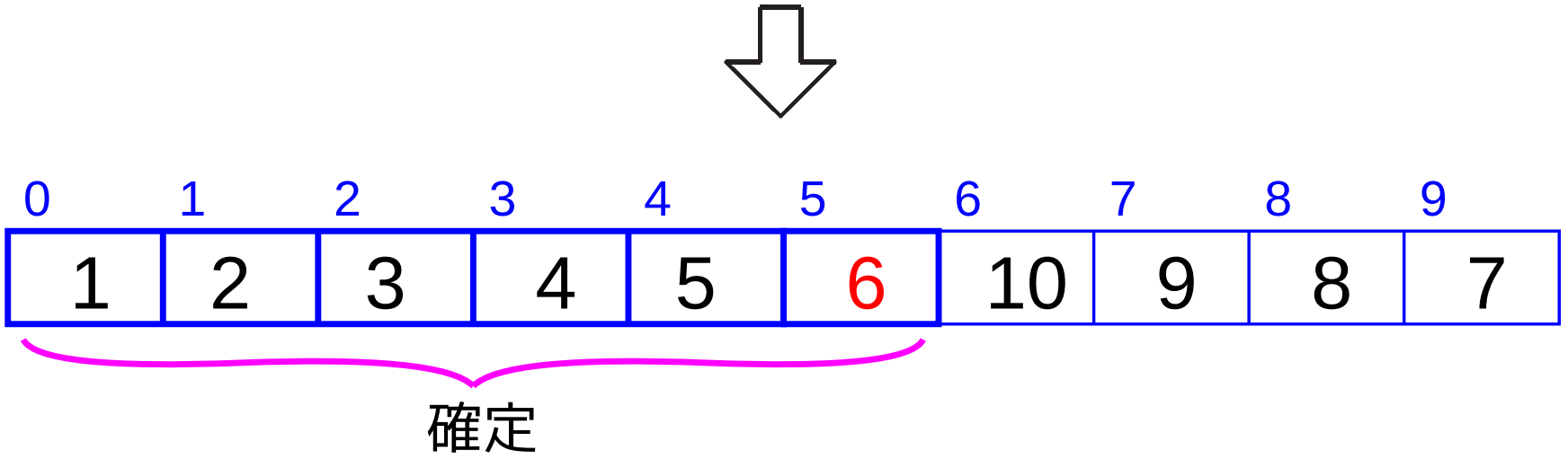
- 残ったデータ列の中で最小データの入った配列要素の番号を見つける。
  例題6.1(1)を参照
- 2つの配列要素に入っているデータを交換する。






0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 9 10 6 8 7
確定
この中で最小要素の入った場所を探す
5
(1)代入
(2)
(3)要素比較
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 9 10 6 8 7
確定
この中で最小要素の入った場所を探す


7
(4)要素比較
(5)要素比較
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 9 10 6 8 7
確定
この中で最小要素の入った場所を探す
残った中で最小要素の入った場所
7
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 9 10 6 8 7
確定
(6)中のデータを交換


0
1
2
3
4
5
6
7
8
9
1
2
3
4
5
6
10
9
8
7
確定

9–3 2次元配列

C言語においては添字番号の個数が複数の配列も用いることができる。

同じデータ型の領域を2次元格子状に並べたものを**2次元配列**、
3次元格子状に並べたものを**3次元配列**と呼び、
次元が2以上の配列を総称して**多次元配列**と呼ぶ。

どの次元も 0, 1, 2, 3, ... という添字番号が付けられ、一般にn次元配列 a の中の添字番号が $i_1, i_2, i_3, \ldots, i_n$ の配列要素は $a[i_1][i_2][i_3]\cdots[i_n]$ と表される。

添字番号	0	1	2	·····	k_2-1
0	a[0][0]	a[0][1]	a[0][2]	·····	a[0][k_2-1]
1	a[1][0]	a[1][1]	a[1][2]	·····	a[1][k_2-1]
2	a[2][0]	a[2][1]	a[2][2]	·····	a[2][k_2-1]
	⋮	⋮	⋮		⋮
k_1-1	a[k_1-1][0]	a[k_1-1][1]	a[k_1-1][2]	·····	a[k_1-1][k_2-1]

添字番号	0	1	2	.....	k_2-1
0	a[0][0]	a[0][1]	a[0][2]	.....	a[0][k_2-1]
1	a[1][0]	a[1][1]	a[1][2]	.....	a[1][k_2-1]
2	a[2][0]	a[2][1]	a[2][2]	.....	a[2][k_2-1]
	⋮	⋮	⋮		⋮
k_1-1	a[k_1-1][0]	a[k_1-1][1]	a[k_1-1][2]	.....	a[k_1-1][k_2-1]

補足：

C言語においては、1次元配列を1列に並べたものが2次元配列、2次元配列を1列に並べたものが3次元配列、..... となっており、内部では多次元配列は1 次元配列を入れ子にしたものとして処理される。

例えば、上に図示された2 次元配列においては、

- a[0][0], a[0][1], ..., a[0][k_2-1] の部分が　a[0]　という1次元配列、
- a[1][0], a[1][1], ..., a[1][k_2-1] の部分が　a[1]　という1次元配列、

.....

であり、同じ大きさの1次元配列を並べた a[0], a[1], a[2], ..., a[k_1-1] の部分(全体)が　a　という2次元配列になる。

例題9. 8 (行列の積)　3×3実数値行列 $A=[a_{ij}]$, $B=[b_{ij}]$ の要素を読み込み、行列の積 AB を計算してその要素を2次元状に見易く出力するCプログラムを作成せよ。

(考え方)

2つの行列 A, B とそれらの積 AB の結果は、各々3×3の2次元配列に保持すれば良い。

積 AB も3×3行列で、その i行, j列 要素は

$$AB\text{の}(i,j)\text{要素} = \sum_{k=1}^{3} a_{ik}b_{kj} = a_{i1}b_{1j}+a_{i2}b_{2j}+a_{i3}b_{3j}$$

と定められているので、積 AB の各要素をこの式の通りに計算して、結果を見易い形に出力するだけである。

(プログラミング)

```
[motoki@x205a]$  nl matrix-multiplication.c   Enter
   1 /* 3×3実数値行列 A=[a_ij], B=[b_ij] の要素を読み込み、*/
   2 /* 行列の積 AB を計算してその要素を2次元状に見易く出力する */
   3 /* Cプログラム

   4 #include <stdio.h>

   5 #define  N  (3)

   6 int main(void)
   7 {
   8   double a[N][N], b[N][N], productAB[N][N];
   9   int  i, j, k;
```

```
10    /* 行列 A,B の各要素を入力する */
11    for (i=0; i<N; i++) {
12      printf("行列 A の %d 行目の要素を順に入力して下さい:  "
13      for (j=0; j<N; j++)
14        scanf("%lf", &a[i][j]);
15    }
16    printf("\n");
17    for (i=0; i<N; i++) {
18      printf("行列 B の %d 行目の要素を順に入力して下さい:  "
19      for (j=0; j<N; j++)
20        scanf("%lf", &b[i][j]);
21    }
```

```
22    /* 行列の積 AB の各要素を計算して配列 productAB に記録 */
23    for (i=0; i<N; i++) {
24      for (j=0; j<N; j++) {
25        productAB[i][j] = 0.0;
26        for (k=0; k<N; k++)
27          productAB[i][j] += a[i][k]*b[k][j];
28      }
29    }

30    /* 行列の積 AB の結果を表示する */
31    printf("\n行列の積 AB の結果:\n");
32    for (i=0; i<N; i++) {
33      printf("    ");
34      for (j=0; j<N; j++)
35        printf("  %12.5g", productAB[i][j]);
36      printf("\n");
37    }
```

```
  38   return 0;
  39 }
[motoki@x205a]$  gcc matrix-multiplication.c  Enter
[motoki@x205a]$  ./a.out  Enter
行列 A の 0 行目の要素を順に入力して下さい:   1  2  3  Enter
行列 A の 1 行目の要素を順に入力して下さい:   4  5  6  Enter
行列 A の 2 行目の要素を順に入力して下さい:   7  8  9  Enter

行列 B の 0 行目の要素を順に入力して下さい:   -1  1  1  Enter
行列 B の 1 行目の要素を順に入力して下さい:   1 -1  1  Enter
行列 B の 2 行目の要素を順に入力して下さい:   1  1 -1  Enter

行列の積 AB の結果:
                4               2               0
                7               5               3
               10               8               6
[motoki@x205a]$
```

9–4 付録 配列のまとめ

配列について：

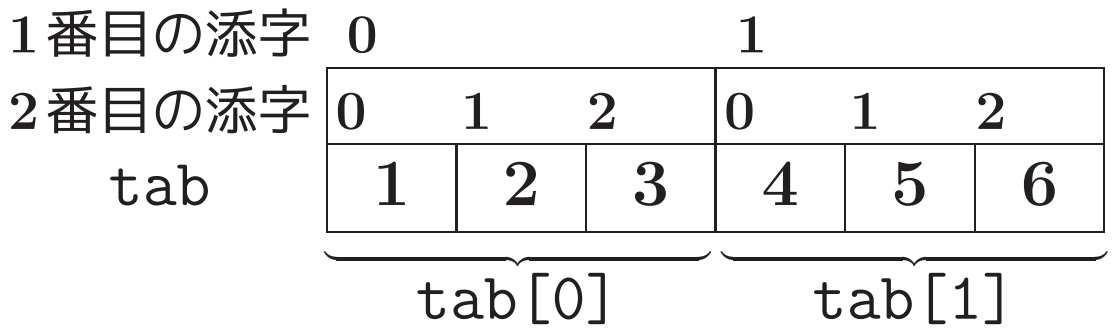
- 配列名が a、大きさが $\mathbf{k_1}\times\mathbf{k_2}\times\cdots\times\mathbf{k_n}$ の配列の宣言／領域確保は次の様に行う。

  データ型 $\mathbf{a[k_1][k_2]\cdots[k_n]}$

- 宣言時の配列の初期化は例えば次の様に行う。

```
int num[12]={31,28,31,30,31,30,31,31,30,31,30,31};

char tab[2][3]={{1,2,3}, {4,5,6}};
```



文字列について：

- **char** 型の配列を使う。

- 文字列の終わりの印として文字列の最後にヌル文字'\0'を置く。

  ⇨ (配列の大きさ) $\geq$ (文字列の長さ) $+ 1$ でなければならない。

0	1	2	3	4	5	6	
's'	't'	'r'	'i'	'n'	'g'	'\0'	$\cdots$

- 文字列を 2 重引用符で囲めば文字列定数になる。

- **char** 型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

```
char s[]="string";
```

(`char s[]={'s','t','r','i','n','g','\0'};` と同等。)