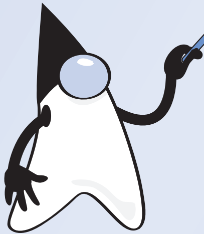
第6章

ソート

- 単純交換ソート（バブルソート）
- 単純選択ソート
- 単純挿入ソート
- シェルソート
- クイックソート
- マージソート
- ヒープソート
- 度数ソート



6-1 ソートとは

本章ではソートについて学習します。本節は、その導入です。

ソートとは

整列（*sorting*）すなわち**ソート**は、名前／学籍番号／身長といったキーとなる項目の値の大小関係に基づいて、データの集合を一定の順序に並べかえる作業です。

データをソートすれば**探索が容易になる**のは、いうまでもありません。もし辞書に収録されている何万語や何十万語にも及ぶ語句がアルファベットや五十音の順でソートされていなければ、目的とする語句を見つけるのは、事実上不可能です。

Fig.6-1 に示すように、キー値の小さいデータを先頭に並べるのが**昇順**（*ascending order*）のソート、その逆が**降順**（*descending order*）のソートです。

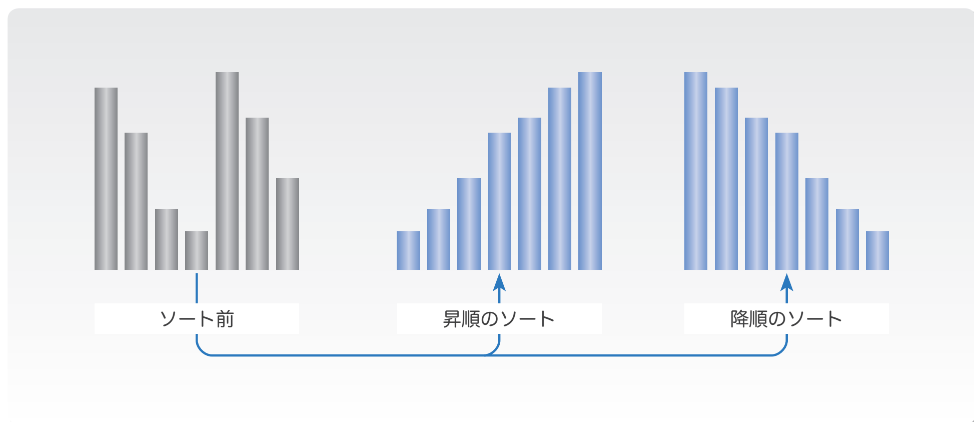


Fig.6-1　昇順ソートと降順ソート

ソートアルゴリズムの安定性

本章では、数多くのソートアルゴリズムから、代表的な八つを学習します。それらは、**安定な**（*stable*）ものと、安定でないものとに分けられます。

安定なソートのイメージを表した図を、**Fig.6-2** に示します。左の図に示しているのが、学籍番号順に並んだテストの点数の配列です（棒の高さが点数であり、棒中の 0 から 9 の数が学籍番号です）。点数をキーとしてソートすると、右の図のようになります。

同じ点数の学生は、ソート後も学籍番号の小さいものが前に、学籍番号の大きいものが後ろに位置しています（右図）。このように、**同一キーをもつ要素の順序関係が、ソート前後で必ず維持される**のが安定なソートです。

安定でないアルゴリズムを利用してソートを行うと、たまたま学籍番号順になることもありますが、それが保証されるわけではありません。

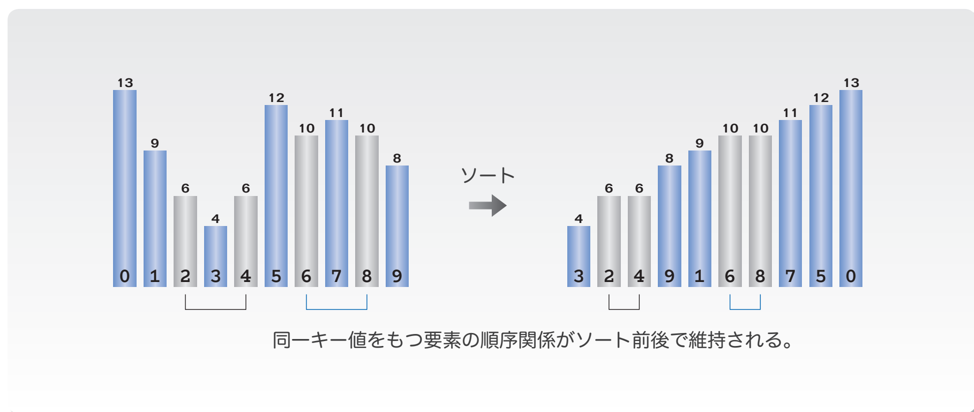


Fig.6-2 安定なソート

内部ソートと外部ソート

さて、最大で30枚のカードを並べられる机の上でトランプのカードをソートすることを想像してみてください。

もしカードが30枚以下であれば、すべてのカードを机において、一度に見渡しながら並べかえられます。しかし、カードが500枚といった感じで大量になると、机の上で一度に並べることはできません。別の大きな机を作業用に用意するなどして、並べかえを行うことになります。

このことは、プログラムにおいても同様です。以下に示すように、ソートアルゴリズムは**内部ソート**（*internal sorting*）と**外部ソート**（*external sorting*）の２種類に分類できます。

■ **内部ソート**
ソートの対象となるすべてのデータが、一つの配列上に格納できる場合に用いられるアルゴリズム。カードは、配列上に展開された要素に相当する。

■ **外部ソート**
ソートの対象となるデータが大量であり、一度に並べかえることができない場合に用いられるアルゴリズム。

外部ソートは、内部ソートの応用です。その実現には作業用ファイルなどが必要であり、アルゴリズムも複雑です。

本書で学習するアルゴリズムは、すべて内部ソートです。

ソートの考え方

ソートアルゴリズムの三大要素は、**交換・選択・挿入**の三つです。ほとんどのソートアルゴリズムは、これらの考え方を応用したものです。

6-2 単純交換ソート（バブルソート）

隣り合う二つの要素の大小関係を調べて、必要に応じて交換を繰り返すのが、本節で学習する**単純交換ソート**（*straight exchange sort*）です。

単純交換ソート（バブルソート）

次に示す数値の並びを例に、単純交換ソートの手順を理解していくことにします。

6	4	3	7	1	9	8

まず、末尾に位置する二つの要素 9 と 8 に着目します。もし昇順にソートするのであれば、先頭側（左側）の値は、末尾側（右側）の値と、同じか、あるいは小さくなければなりません。そこで、これらの二値を交換すると、並びは次のようになります。

6	4	3	7	1	8	9

引き続き、後ろから 2 番目と 3 番目の要素 1 と 8 に着目します。1 は 8 より小さいため、交換を行う必要はありません。

このように、隣り合う要素を比較して、必要ならば交換する作業を先頭要素まで続けていく様子を示したのが **Fig.6-3** です。

要素数 n の配列に対して $n - 1$ 回の比較・交換を行うと、**最小要素が先頭に移動します**。この一連の比較・交換の作業をパスと呼びます。

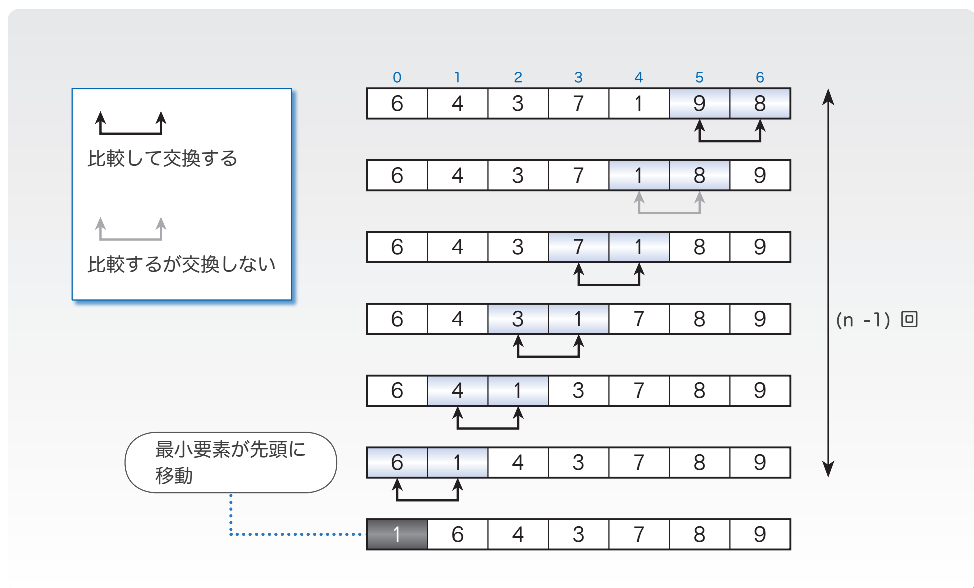


Fig.6-3 単純交換ソートにおける1回目のパス

引き続き、配列の２番目以降の要素に対して比較・交換のパスを行います。その様子を示したのが **Fig.6-4** です。

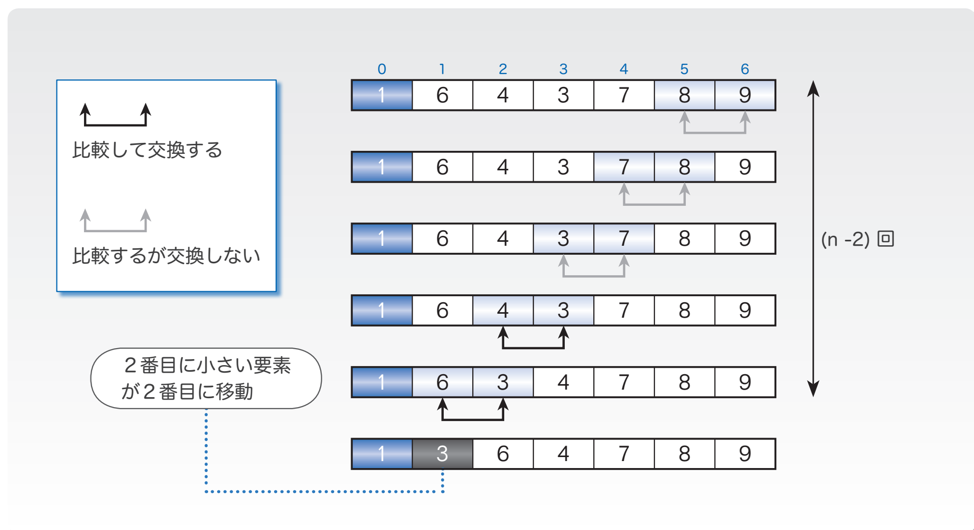


Fig.6-4　単純交換ソートにおける2回目のパス

このパスが完了すると、２番目に小さい要素である３が先頭から２番目の位置へと移動します。その結果、先頭２個の要素が **“ソート済み”** となります。

２パス目の比較回数は、１パス目より１回少ない $n - 2$ 回です。というのも、パスを行うたびにソートすべき要素が一つずつ減っていくからです。

パスを k 回行えば、先頭側 k 個の要素が **“ソート済み”** となります。したがって、パスを $(n - 1)$ 回行うと、全体のソートが完了です。

▶　行うパスの回数が、n 回ではなくて $n - 1$ 回でよいのは、先頭 $n - 1$ 個の要素がソート済みとなれば、最大要素が末尾に位置して全体がソート済みとなるからです。

*

液体中の気泡を想像してください。液体より軽い＝値の小さい気泡が、ブクブクと上にあがってきます。

そのイメージと似ているため、単純交換ソートは**バブルソート**（*bubble sort*）あるいは**泡立ちソート**とも呼ばれます。

単純交換ソートのプログラム

単純交換ソートのアルゴリズムを、プログラムとして実現しましょう。

変数 `i` の値を０から $n - 2$ までインクリメントしてパスを $n - 1$ 回行うことにすると、プログラムの概略は次のようになります（次ページ）。

```
for (int i = 0; i < n - 1; i++) {
    a[i], a[i + 1], …, a[n - 1]に対して、
    隣接する二つの要素を比較して先頭側が大きければ交換する作業を、
    末尾側から先頭側へ走査しながら行う。
}
```

ここで、比較のために着目する２要素のインデックスを、$j - 1$ と j とします。変数 j の値を、どのように変化させればよいのか、**Fig.6-5** を見ながら考えていきましょう。

走査は末尾側から行いますので、走査における j の開始値は、どのパスにおいても末尾要素のインデックスである $n - 1$ です。

走査の過程では、二つの要素 a[j - 1] と a[j] の値を比較して、前者のほうが大きければ交換します。これを先頭側に向かって行うのですから、j の値を一つずつデクリメントすることになります。

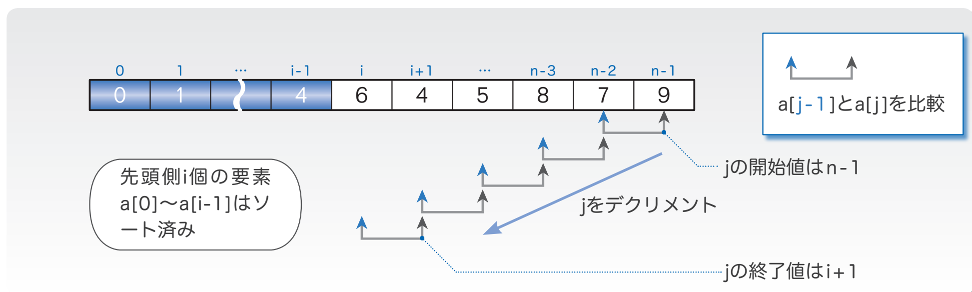


Fig.6-5　単純交換ソートにおけるi回目のパス

各パスにおいて、先頭 i 個の要素はソート済みであって、未ソート部は a[i] 〜 a[n-1] です。そのため、j のデクリメントは、その値が $i + 1$ になるまで行うことになります。

▶ 前ページまでに示した二つの図で確認してみましょう。以下のようになっています。

- i が 0 である１回目のパスでは、j の値が 1 になるまで繰り返す（**Fig.6-3**）。
- i が 1 である２回目のパスでは、j の値が 2 になるまで繰り返す（**Fig.6-4**）。

なお、比較する二つの要素の末尾側＝右側のインデックスを $i + 1$ になるまで減らしますから、先頭側＝左側のインデックスは i になるまで減らされることになります。

単純交換ソートを行うプログラムを **List 6-1** に示します。

＊

飛び越えた要素を一気に交換せず、隣り合う要素のみを交換するため、このソートアルゴリズムは**安定**です。

要素の**比較回数**は、１回目のパスでは $n - 1$、２回目のパスでは $n - 2$、… ですから、その合計は次のようになります。

$$(n - 1) + (n - 2) + \cdots + 1 = n(n - 1) / 2$$

ただし、実際に要素を**交換する回数**は、配列の要素の値によって左右されますので、その平均は比較回数の半分の $n(n - 1) / 4$ 回です。

▶ *swap* 内で移動（代入）が３回行われますから、移動回数の平均は $3n(n - 1) / 4$ 回です。

List 6-1　Chap06/BubbleSort.java

```
// 単純交換ソート

import java.util.Scanner;

class BubbleSort {

   //--- 配列の要素a[idx1]とa[idx2]の値を交換 ---//
   static void swap(int[] a, int idx1, int idx2) {
      int t = a[idx1]; a[idx1] = a[idx2]; a[idx2] = t;
   }

   //--- 単純交換ソート ---//
   static void bubbleSort(int[] a, int n) {
      for (int i = 0; i < n - 1; i++)
         for (int j = n - 1; j > i; j--)
            if (a[j - 1] > a[j])
               swap(a, j - 1, j);
   }

   public static void main(String[] args) {
      Scanner stdIn = new Scanner(System.in);

      System.out.println("単純交換ソート（バブルソート）");
      System.out.print("要素数：");
      int nx = stdIn.nextInt();
      int[] x = new int[nx];

      for (int i = 0; i < nx; i++) {
         System.out.print("x[" + i + "]：");
         x[i] = stdIn.nextInt();
      }

      bubbleSort(x, nx);              // 配列xを単純交換ソート

      System.out.println("昇順にソートしました。");
      for (int i = 0; i < nx; i++)
         System.out.println("x[" + i + "]＝" + x[i]);
   }
}
```

パス（内側のforループ部分）

実行例

```
単純交換ソート
 （バブルソート）
要素数：7
x[0]：22
x[1]：5
x[2]：11
x[3]：32
x[4]：120
x[5]：68
x[6]：70
昇順にソートしました。
x[0]＝5
x[1]＝11
x[2]＝22
x[3]＝32
x[4]＝68
x[5]＝70
x[6]＝120
```

▶　メソッド *swap* は、**List 2-6**（p.43）で作成したものと同じです。

□ 演習 6-1

各パスでの比較・交換の走査を末尾側ではなく先頭側から行ってもソートは可能である（各パスでは最大要素が末尾側へと移動することになる）。

そのように変更したプログラムを作成せよ。

□ 演習 6-2

右のように比較・交換の過程を詳細に表示するプログラムを作成せよ。

比較する二つの要素間には、交換を行う場合は '+' を、交換を行わない場合は '-' を表示すること。

また、ソート終了時に、比較回数と交換回数を表示すること。

```
パス1:
  6   4   3   7   1   9 + 8
  6   4   3   7   1 - 8   9
  6   4   3   7 + 1   8   9
  6   4   3 + 1   7   8   9
  6   4 + 1   3   7   8   9
  6 + 1   4   3   7   8   9
  1   6   4   3   7   8   9
パス2:
  1   6   4   3   7   8 - 9
   …中略…
比較は21回でした。
交換は8回でした。
```

アルゴリズムの改良（1）

先ほどは、2 番目に小さい要素を並べるまでの様子を示しました（p.185）。比較・交換の作業を続けていきましょう。**Fig.6-6** に示すのは、3 パス目の手続きです。パス終了時に、3 番目に小さい要素である 4 が 3 番目に位置します。

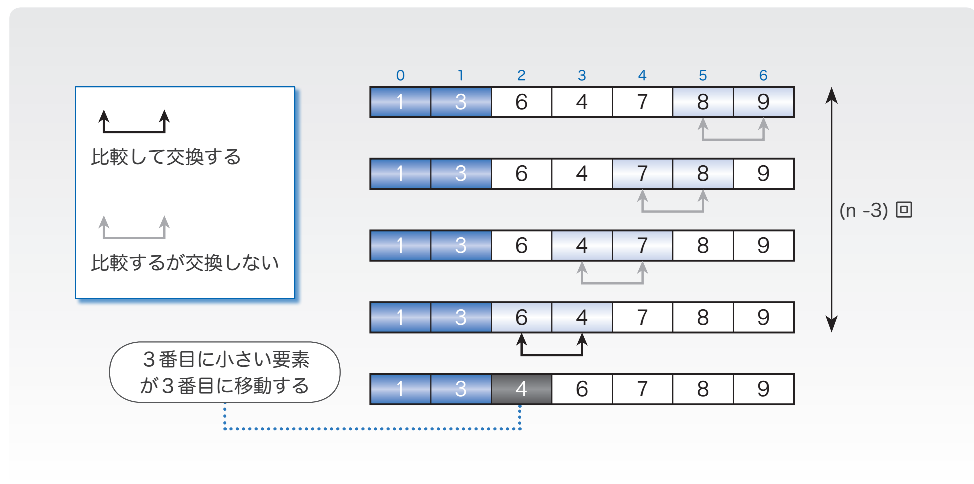


Fig.6-6　単純交換ソートにおける3回目のパス

次に行う 4 パス目の手続きを示したのが、**Fig.6-7** です。ここでは、要素の交換が一度も行われません。というのも、3 パス目で**ソートが完了している**からです。

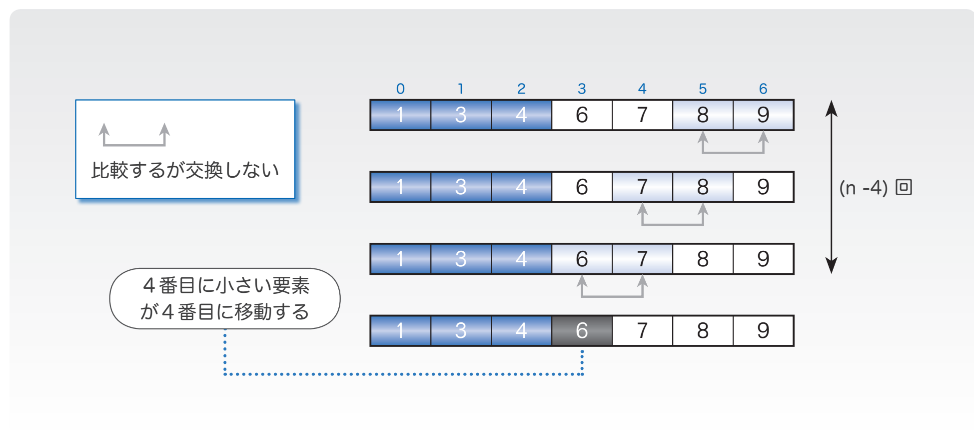


Fig.6-7　単純交換ソートにおける4回目のパス

ソートが完了すれば、それ以降のパスで交換が行われることはありません。

▶　ここには示しませんが、5 パス目と 6 パス目でも、要素の交換は行われません。

各パスにおいて要素の交換回数をカウントしておきます。その値が 0 であれば、すべての要素がソート済みであると判断でき、**それ以降のパスを省略できます**。

もしも、最初に与えられた配列が、たまたま以下のようにソート済みであったとします。

1	3	4	6	7	8	9

最初に行う 1 回目のパスで一度も交換が行われませんから、1 パス目が終了した時点でソート作業を終了できます。

この《打切り》を導入すると、もともとソート済みの配列や、それに近い状態の配列に対しては、行う必要のない比較が大幅に省略されることになり、ソートは短時間で完了します。

そのように改良した単純交換ソートのメソッドが **List 6-2** です。

List 6-2 Chap06/BubbleSort2.java

```
//--- 単純交換ソート（第２版：交換回数による打切り）---//
static void bubbleSort(int[] a, int n) {
    for (int i = 0; i < n - 1; i++) {
        int exchg = 0;                        // パスにおける交換回数
        for (int j = n - 1; j > i; j--)
            if (a[j - 1] > a[j]) {
                swap(a, j - 1, j);            ← パス
                exchg++;
            }
        if (exchg == 0) break;                // 交換が行われなかったら終了
    }
}
```

変数 *exchg* は、パスの開始直前に 0 にしておき、要素を交換するたびにインクリメントします。したがって、パスが終了した（内側の **for** 文による繰返しが完了した）時点での変数 *exchg* の値は、**そのパスにおける交換回数**となります。

パス終了時点での *exchg* の値が 0 であれば、配列全体がソート済みということですから、**break** 文によって外側の **for** 文を強制的に脱出して、メソッドの実行を終了します。

□ 演習 6-3

演習 6-2（p.187）と同様に、比較・交換の過程を詳細に表示するように、第２版のプログラムを書きかえよ。

アルゴリズムの改良（2）

次は、{1，3，9，4，7，8，6}というデータの並びに対して単純交換ソートを行ってみます。最初のパスにおける比較・交換の過程を示したのが **Fig.6-8** です。

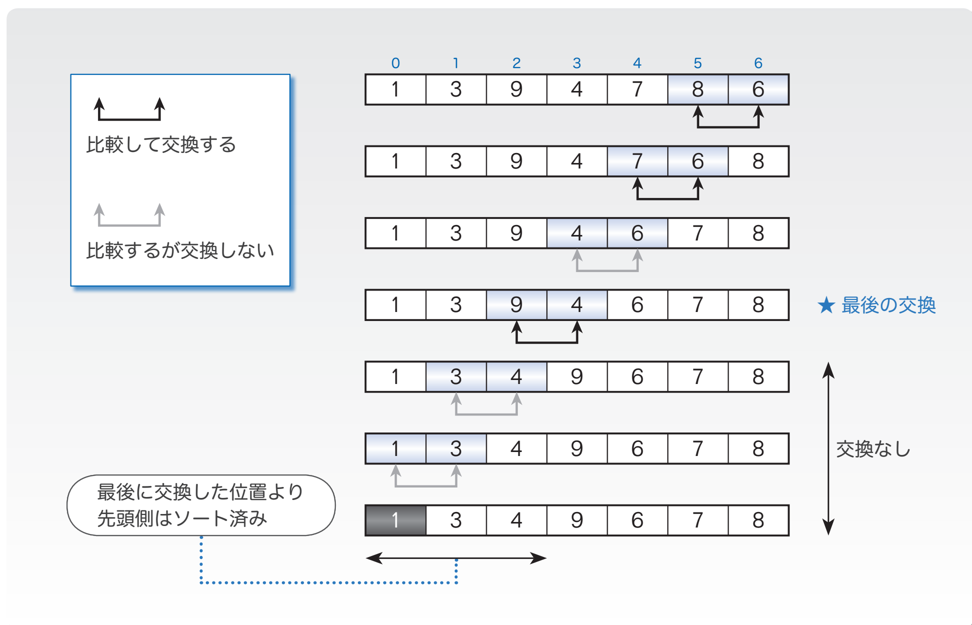


Fig.6-8　単純交換ソートにおける1回目のパス

★の交換が終了した時点で、先頭の3要素{1，3，4}がソート済みとなっています。

この例が示すように、一連の比較・交換を行うパスにおいて、ある時点以降に交換がなければ、それより先頭側はソート済みです。

そのため、2回目のパスは、先頭を除いた6要素ではなく、4要素に絞り込めます。すなわち、**Fig.6-9** に示すように、4要素のみを比較・交換の対象とすればいいのです。

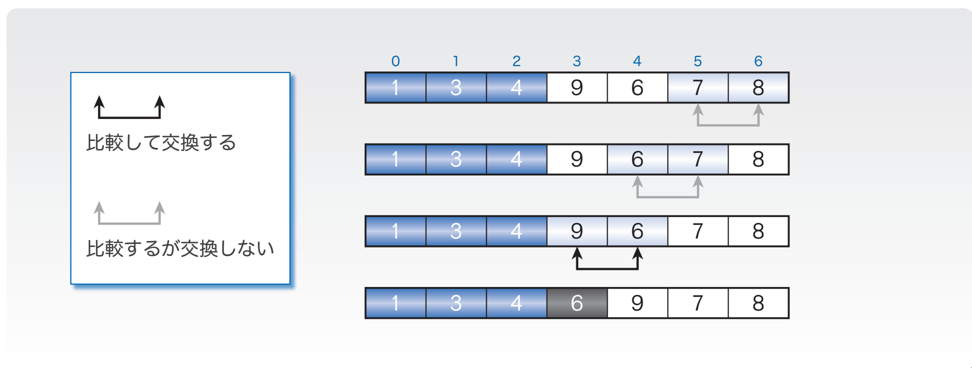


Fig.6-9　単純交換ソートにおける2回目のパス

このアイディアに基づいて改良したメソッドを **List 6-3** に示します。

List 6-3 Chap06/BubbleSort3.java

```
//--- 単純交換ソート（第３版：走査範囲を限定）---//
static void bubbleSort(int[] a, int n) {
    int k = 0;                          // a[k]より前はソート済み
    while (k < n - 1) {
        int last = n - 1;               // 最後に交換した位置
        for (int j = n - 1; j > k; j--)
            if (a[j - 1] > a[j]) {
                swap(a, j - 1, j);        ← パス
                last = j;
            }
        k = last;
    }
}
```

last は、各パスにおいて最後に交換した２要素の右側要素のインデックスを格納する変数です（交換を行うたびに、右側要素のインデックスの値を *last* に入れます）。

パスが終了した（**for** 文による繰返しが完了した）時点で、*last* の値を *k* に代入することによって、次に行われるパスの走査範囲を a[*k*] までに限定します（次のパスで最後に比較される要素が、a[*k*] と a[*k* + 1] になります）。

▶ **Fig.6-8** の例であれば、パス終了時の *last* の値は 3（9 と 4 を比較したときの右側のインデックス）となります。そのため、次に行われる２回目のパスでは、*j* の走査は 6, 5, 4 とデクリメントされながら３回行われます。

なお、メソッドの冒頭で *k* の値を 0 に初期化しているのは、第１回目のパスで先頭までの全要素を走査するためです。

□ 演習 6-4

演習 6-2（p.187）と同様に、比較・交換の過程を詳細に表示するように、第３版のプログラムを書きかえよ。

□ 演習 6-5

以下に示すデータの並びをソートすることを考えよう。

9 1 3 4 6 7 8

ほぼソート済みであるにもかかわらず、最大の要素 9 が先頭に位置しているために、第３版のアルゴリズムでも、ソート作業を早期に打ち切ることはできない。というのも、最大の要素 9 が、1 回のパスで一つずつしか後ろに移動しないためである。

そこで、奇数パスでは最小要素を先頭側に移動させ、偶数パスでは最大要素を末尾側に移動するというように、パスの走査方向を交互に変えると、このような並びを少ない比較回数でソートできる。バブルソートを改良したこのアルゴリズムは、**双方向バブルソート**（*bidirection bubble sort*）あるいは**シェーカーソート**（*shaker sort*）という名称で知られている。

第３版を改良して、双方向バブルソートを行うプログラムを作成せよ。

6-3 単純選択ソート

単純選択ソート（*straight selection sort*）は、最小要素を先頭に移動し、2 番目に小さい要素を先頭から 2 番目に移動する、といった作業を繰り返すアルゴリズムです。

単純選択ソート

次に示す並びのソートを考えましょう。まず最小の要素 1 に着目します。

6	4	8	3	1	9	7

これは、配列の先頭に位置すべきものですから、先頭要素 6 と交換します。そうすると、データの並びは次のようになります。

1	4	8	3	6	9	7

これで**最小の要素が先頭に位置する**ことになります。

引き続き、2 番目に小さい要素 3 に着目します。先頭から 2 番目の要素 4 と交換すると、次に示すように、2 番目の要素までのソートが完了します。

1	3	8	4	6	9	7

同様な作業を続けていく様子を示したのが **Fig.6-10** です。

未ソート部から最小の要素を選択して、未ソート部の先頭要素と交換する操作を繰り返します。

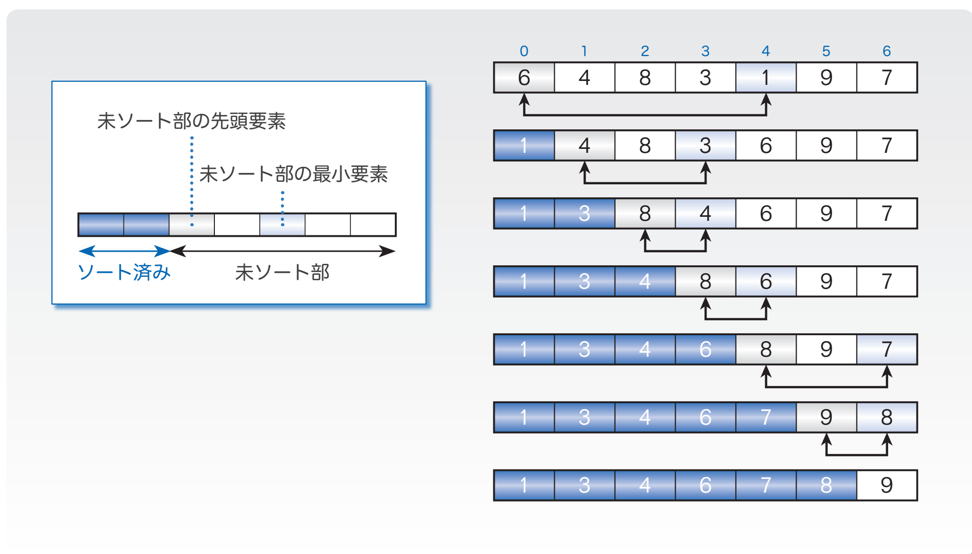


Fig.6-10 単純選択ソートの手順

交換の手順は、次のとおりです。

1. 未ソート部から最小のキーをもつ要素 a[*min*] を選択する。
2. a[*min*] と未ソート部の先頭要素を交換する。

これを $n - 1$ 回繰り返すと、未ソート部がなくなってソートは完了です。したがって、アルゴリズムの概略は、以下のようになります。

```
for (int i = 0; i < n - 1; i++) {
   min ← a[i], …, a[n - 1]で最小のキーをもつ要素のインデックス。
   a[i]とa[min]の値を交換する。
}
```

単純選択ソートを行うメソッドを **List 6-4** に示します。

List 6-4　Chap06/SelectionSort.java

```
//--- 単純選択ソート ---//
static void selectionSort(int[] a, int n) {
   for (int i = 0; i < n - 1; i++) {
      int min = i;          // 未ソート部の最小要素のインデックス
      for (int j = i + 1; j < n; j++)
         if (a[j] < a[min])
            min = j;
      swap(a, i, min);      // 未ソート部の先頭要素と最小要素を交換
   }
}
```

要素の値を比較する回数は $(n^2 - n) / 2$ です。

＊

このソートアルゴリズムは、離れた要素を交換することがあるため、**安定ではないこと**に注意しましょう。安定でないソートが行われる例が **Fig.6-11** です。値 3 の要素が二つあります（ソート前の先頭側を 3*、末尾側を 3** と表しています）が、これらの要素の順序は、ソート後には逆転しています。

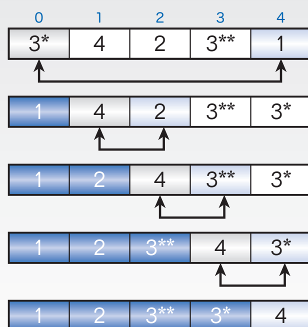


Fig.6-11　単純選択ソートが安定でないことを示す例

6-4 単純挿入ソート

シャトルソート（*shuttle sort*）とも呼ばれる**単純挿入ソート**（*straight insertion sort*）は、着目要素をそれより先頭側の適切な位置に“挿入する”作業を繰り返すアルゴリズムです。

単純挿入ソート

単純挿入ソートは、トランプのカードを並べるときに行う方法に似たアルゴリズムです。次に示すデータの並びを考えます。

6	4	1	7	3	9	8

まず2番目の要素4に着目します。これは、先頭の6よりも先頭側に位置すべきですから先頭に挿入します。これに伴って6を右にずらすと、次のようになります。

4	6	1	7	3	9	8

次に3番目の要素1に着目し、先頭に挿入します。以下、同様な作業を行っていきます。その様子を示したのが**Fig.6-12**です。

図に示すように、**目的列**と**原列**とから配列が構成されると考えると、

原列の先頭要素を、目的列内の適切な位置に挿入する。

という操作を $n - 1$ 回繰り返せばソートが完了することが分かります。

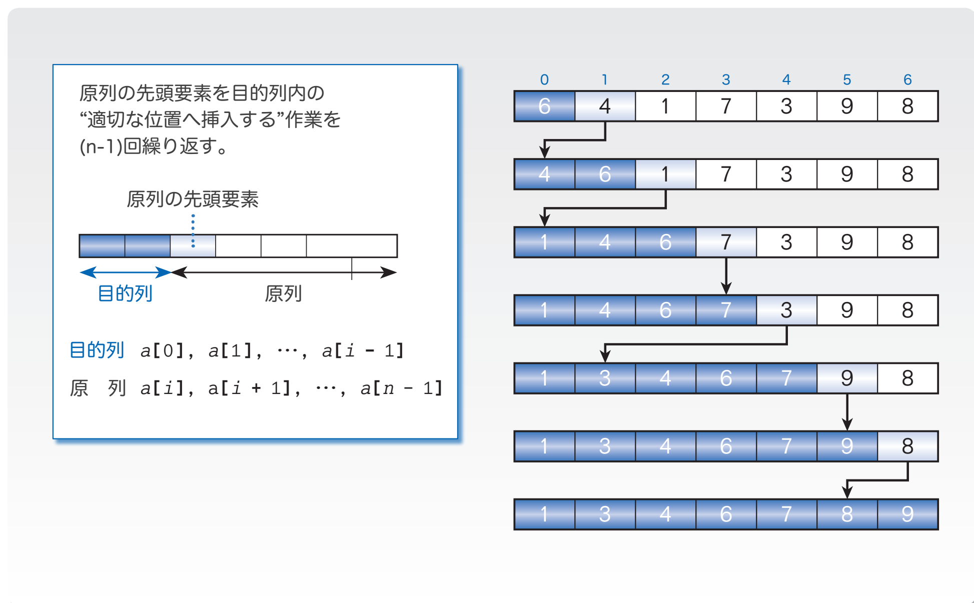


Fig.6-12 単純挿入ソートの手順

このとき、i を1，2，… とインクリメントしながら、インデックスが i である要素を取り出して、それを目的列内の適切な位置に挿入するのですから、アルゴリズムの概略は、以下のようになります。

```
for (int i = 1; i < n; i++) {
    tmp ← a[i]
    tmpをa[0], …, a[i - 1]の適切な位置に挿入する。
}
```

さて、Javaには配列の要素を《適切な位置に挿入する》という命令はありません。したがって、その実現には多少の工夫が必要です。

具体的な手続きの一例を **Fig.6-13** に示します。これは、値が3である要素を、それより先頭側の適切な位置に挿入する手順です。

左隣の要素が、現在着目している要素の値よりも大きい限り、その値を代入する作業を繰り返します。ただし、挿入する値以下の要素に出会うと、そこから先は走査していく必要はありませんので、そこに挿入する値を代入します。

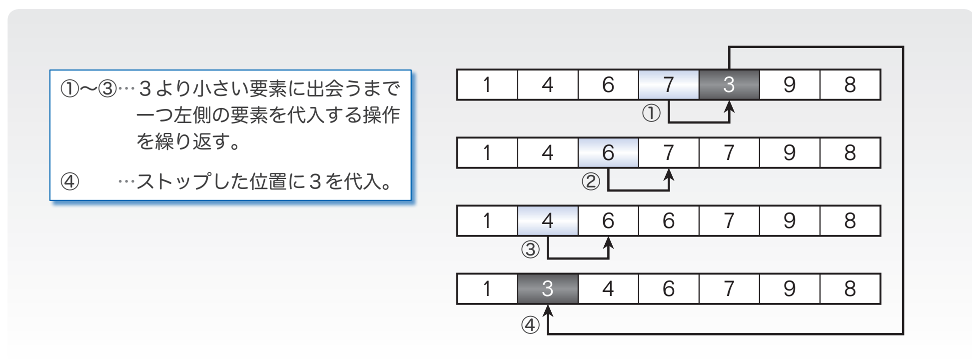


Fig.6-13　単純挿入ソートにおける《挿入》の手続き

すなわち、tmp に a[i] を代入し、繰返し制御用の変数 j に i - 1 を代入しておき、

> 1 目的列の左端に達した。
> 2 tmp と等しいか小さいキーをもった項目 a[j] が見つかった。

のいずれか一方が成立するまで j をデクリメントしながら代入操作を繰り返します。

ド・モルガンの法則（p.21）を用いると、以下に示す二つの条件の両方が成立している間繰り返すことになります。

> 1 j が0より大きい。
> 2 a[j - 1] の値が tmp より大きい。

そして、この走査が終了すると、その位置の要素 a[j] に、挿入すべき値である tmp を代入します。

単純挿入ソートを行うプログラムを **List 6-5** に示します。

List 6-5 Chap06/InsertionSort.java

```java
// 単純挿入ソート

import java.util.Scanner;

class InsertionSort {

	//--- 単純挿入ソート ---//
	static void insertionSort(int[] a, int n) {
		for (int i = 1; i < n; i++) {
			int j;
			int tmp = a[i];
			for (j = i; j > 0 && a[j - 1] > tmp; j--)
				a[j] = a[j - 1];
			a[j] = tmp;
		}
	}

	public static void main(String[] args) {
		Scanner stdIn = new Scanner(System.in);

		System.out.println("単純挿入ソート");
		System.out.print("要素数：");
		int nx = stdIn.nextInt();
		int[] x = new int[nx];

		for (int i = 0; i < nx; i++) {
			System.out.print("x[" + i + "]：");
			x[i] = stdIn.nextInt();
		}

		insertionSort(x, nx);			// 配列xを単純挿入ソート

		System.out.println("昇順にソートしました。");
		for (int i = 0; i < nx; i++)
			System.out.println("x[" + i + "]＝" + x[i]);
	}
}
```

実行例

```
単純挿入ソート
要素数：7
x[0]：22
x[1]：5
x[2]：11
x[3]：32
x[4]：120
x[5]：68
x[6]：70
昇順にソートしました。
x[0]＝5
x[1]＝11
x[2]＝22
x[3]＝32
x[4]＝68
x[5]＝70
x[6]＝120
```

飛び越えた要素の交換が行われることはありませんので、このソートアルゴリズムは**安定**です。

要素の比較回数と交換回数は、ともに $n^2 / 2$ です。

単純ソートの時間計算量

ここまで学習してきた三つの単純ソートの時間計算量は、いずれも $O(n^2)$ であり、非常に効率の悪いものです。

次節以降では、これらのソートを改良した、効率のよいアルゴリズムを学習します。

□ 演習 6-6

　演習 6-2（p.187）と同様に、比較・交換の過程を詳細に表示するように、単純選択ソートのプログラムを書きかえよ。
※本問は、前節の内容に関する演習問題である。

□ 演習 6-7

　演習 6-2（p.187）と同様に、比較・代入の過程を詳細に表示するように、単純挿入ソートのプログラムを書きかえよ。

□ 演習 6-8

　配列の先頭要素 `a[0]` が未使用でデータが `a[1]` 以降に格納されていれば、`a[0]` を番兵とすることによって、挿入処理の終了条件を緩和できる。
　このアイディアに基づいて単純挿入ソートを行うメソッドを作成せよ。

□ 演習 6-9

　配列の要素数が多くなると、要素の挿入に要する比較・代入のコストは無視できなくなる。目的列はソート済みであるため、挿入すべき位置は２分探索法によって調べることができる。そのように変更したプログラムを作成せよ。
　なお、このソート法は、**２分挿入ソート**（*binary insertion sort*）と呼ばれるアルゴリズムとして知られている。
※ただし、安定ではなくなることに注意せよ。