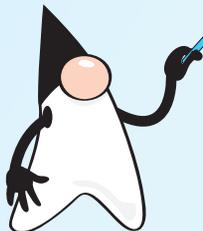


# 第6章

## 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
- Arrays.sortによるソート
- 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、データを一定の順序で並びかえるソートアルゴリズム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びかえる作業です。

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。

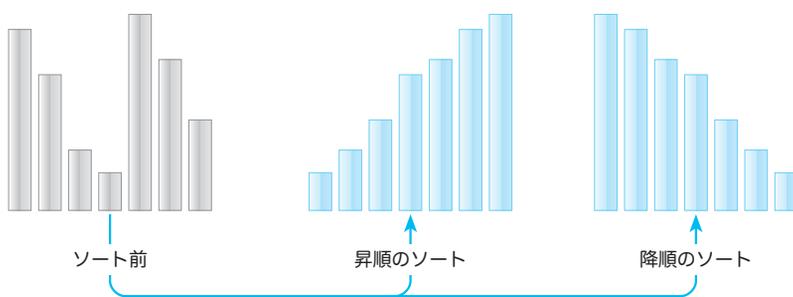


Fig.6-1 昇順ソートと降順ソート

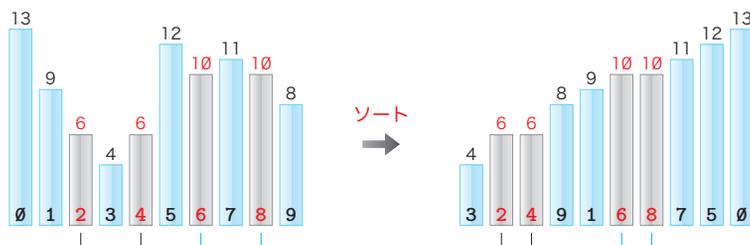
## ■ ソートアルゴリズムの安定性

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。それらのアルゴリズムは、**安定な** (stable) ものと、そう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のが、Fig.6-2 です。左の図に示しているのは、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、棒中の0から9の数値が学籍番号です。

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、学籍番号の小さいものが前に位置し、学籍番号の大きいものが後ろに位置しています。このように、同一キーをもつ要素の順序がソート前後で維持されるのが、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。



同一キー値をもつ要素の順序がソート前後で維持される。

Fig.6-2 安定なソート

### ■ 内部ソートと外部ソート

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考えましょう。

もしカードが30枚以下であれば、すべてのカードを机に置いて、一度に見渡しながら作業を行えます。しかし、カードが500枚といった感じで大量になると、机の上にすべてのカードは並べられません。そのため、大きな机を別に用意するなどして、作業を行うこととなります。

プログラムも同様です。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種類に分類されます。

#### ■ 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るアルゴリズムです。カードは、配列上に展開された要素に相当します。

#### ■ 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るアルゴリズムです。

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、これらの要素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** (*straight exchange sort*) の手順を理解していきましょう。

## 6

ソ  
ー  
ト

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。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、交換の必要はありません。

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、先頭要素まで続けた様子を示したのが **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。

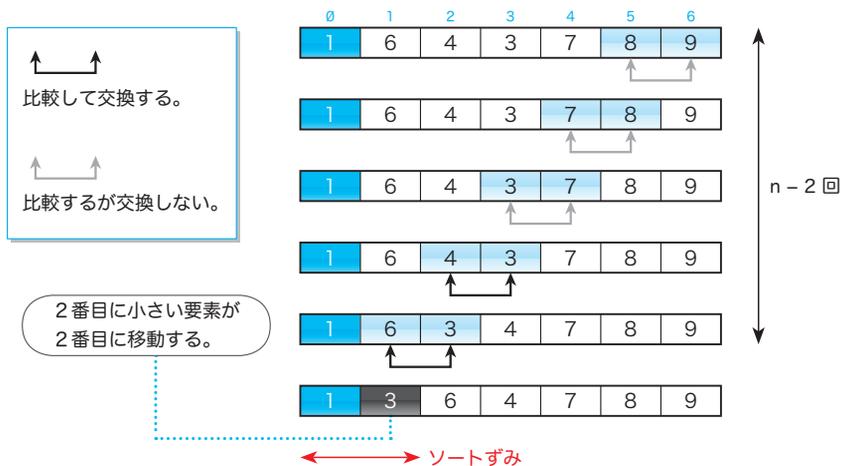


Fig.6-4 単純交換ソートにおける2回目のパス

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 $k$ 回行くと、先頭側 $k$ 個の要素がソート済みとなることが分かりました。全体のソートを完了させるために必要なパスは $n - 1$ 回です。

- ▶ 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ましょう。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。

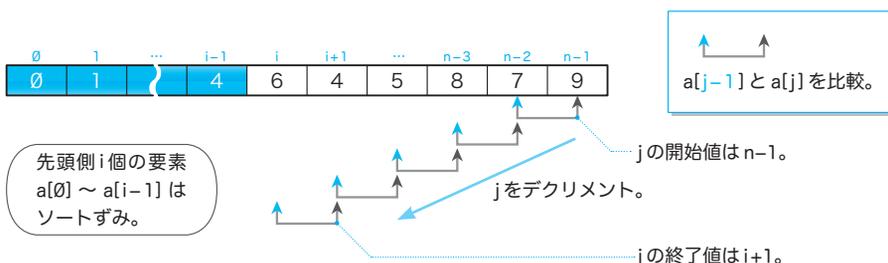
変数 $i$ の値を0から $n - 2$ までインクリメントしてパスを $n - 1$ 回行うことにすると、プログラムの概略は次のようになります（次ページ）。

```
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。

走査は配列の末尾から先頭に向かって行うため、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、2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、 $j$ の値は一つずつデクリメントしていきます。



**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 $a[i] \sim a[n-1]$ です。そのため、 $j$ のデクリメントは、その値が $i + 1$ になるまで行います。

- ▶ 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2要素の末尾側=右側のインデックスを $i + 1$ になるまでデクリメントしますから、先頭側=左側のインデックスは $i$ になるまでデクリメントされます。

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 $n - 2$ 、 $\dots$  ですから、その合計は次のようになります。

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

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。その平均値は、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.print("x[" + i + "]=" + x[i]);
    }
}
```

## 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
```

パス

▶ 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

単純交換ソートの各パスでの比較・交換の走査を、末尾側ではなく先頭側から行っても、ソートは行える (各パスでは最大要素が末尾側へと移動する)。

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しながら単純交換ソートを行うプログラムを作成せよ。

比較する2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。
```

## 6-2

## ■ アルゴリズムの改良 (1)

Fig.6-4 (p.185) では、2 番目に小さい要素を並べるまでの様子を示しました。比較・交換の作業を続けましょう。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。

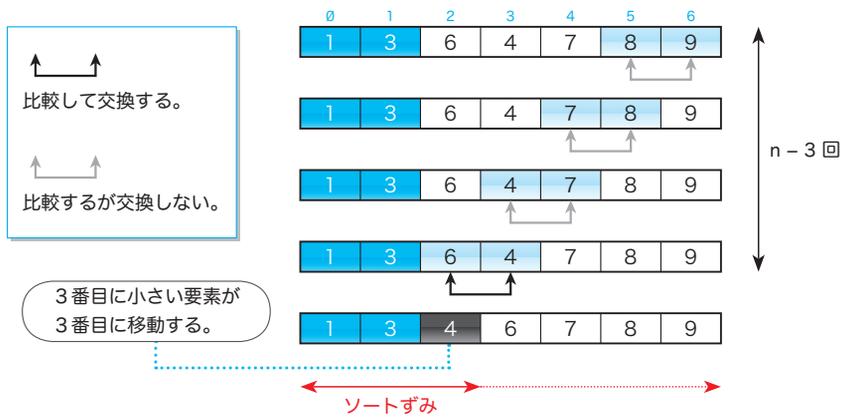


Fig.6-6 単純交換ソートにおける 3 回目のパス

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。

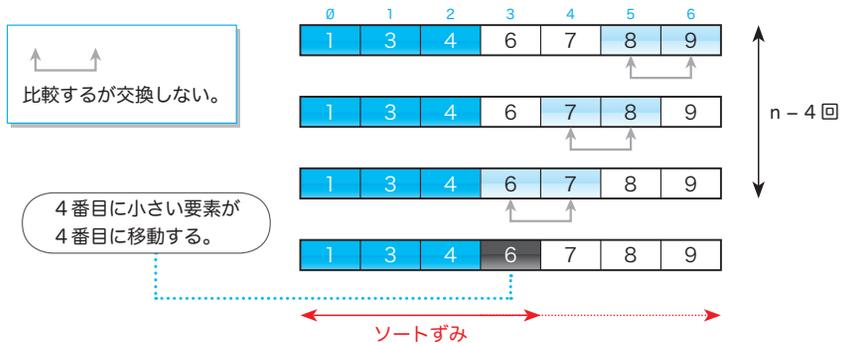


Fig.6-7 単純交換ソートにおける 4 回目のパス

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。図は省略しますが、5 パス目と 6 パス目でも、要素の交換は行われません。

あるパスにおける要素の交換回数が0であれば、すべての要素がソート済みですから、それ以降のパスは不要であって、ソート作業は打ち切れます。

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だったとします。

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ないため、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《打ち切り》を導入すると、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ため、短時間でソートが完了します。

\*

そのように改良した第2版のメソッドが、**List 6-2** のメソッド `bubbleSort` です。

**List 6-2**

Chap06/BubbleSort2.java

```

//--- 単純交換ソート (第2版: 交換回数による打ち切り) ---//
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);
                exchg++;
            }
        if (exchg == 0) break; // 交換が行われなかったら終了
    }
}

```

← **パス**

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。そのため、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、ソート完了と判断できるため、`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。

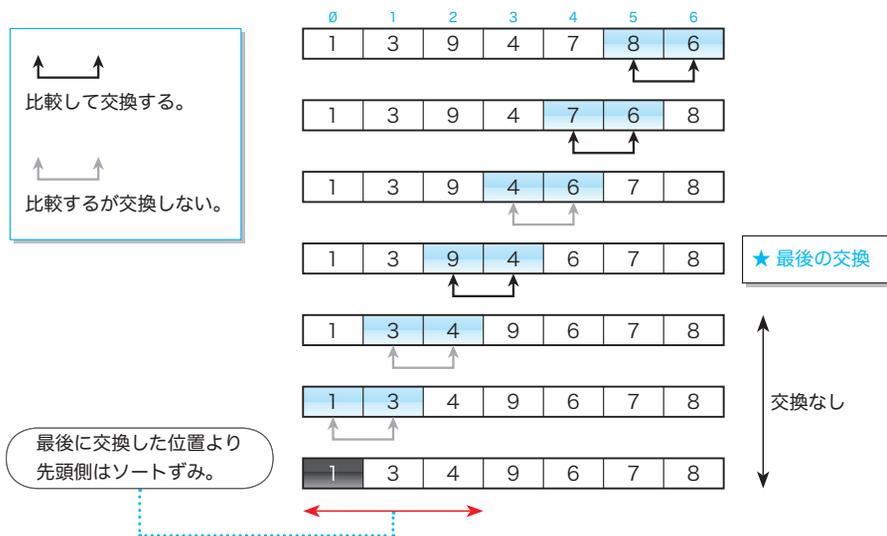


Fig.6-8 単純交換ソートにおける1回目のパス

★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要素に絞られます。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。

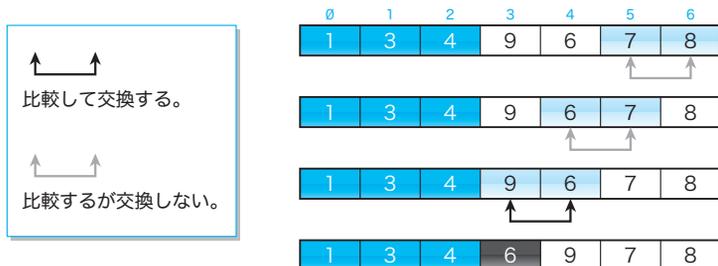


Fig.6-9 単純交換ソートにおける2回目のパス

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);
                last = j;
            }
        k = last;
    }
}

```



`last` は、各パスで最後に交換した 2 要素の右側要素のインデックスを格納するための変数です。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 2 要素は `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 を比較した際の右側の要素のインデックスである 3) です。そのため、次に行われる 2 回目のパス (**Fig.6-9**) 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 版を書きかえたプログラムを作成せよ。

#### 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、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いできない。というのも、先頭に位置している最大の要素 9 が、1 回のパスで一つずつしか後方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、このような並びのソートを少ない比較回数で行える。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例に、**単純選択ソート** (*straight selection sort*) のアルゴリズムを考えていきます。まず最小の要素1に着目しましょう。

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しました。

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。

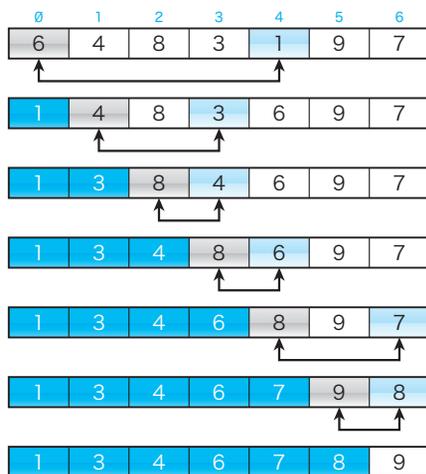
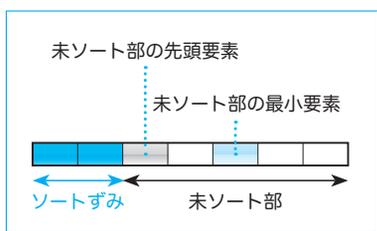


Fig.6-10 単純選択ソートの手順

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。すなわち、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**

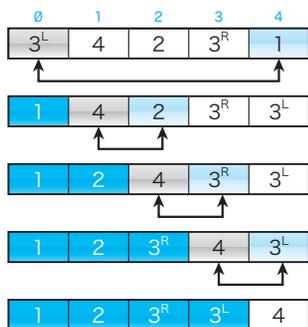
Chap06/SelectionSort.java

```
///--- 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ため、**安定ではありません**。

安定でないソートが行われる具体例が **Fig.6-11** です。値3の要素が二つあります（識別のために、ソート前の先頭側を  $3^L$ 、末尾側を  $3^R$  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。



もともと先頭側の  $3^L$  が末尾側に、  
末尾側の  $3^R$  が先頭側に移動している !!

**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

単純挿入ソートは、着目要素をそれより先頭側の適切な位置に“挿入する”作業を繰り返してソートを行うアルゴリズムです。

### 単純挿入ソート

**単純挿入ソート** (*straight insertion sort*) は、トランプのカードを並べるときに使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しょう。

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します。

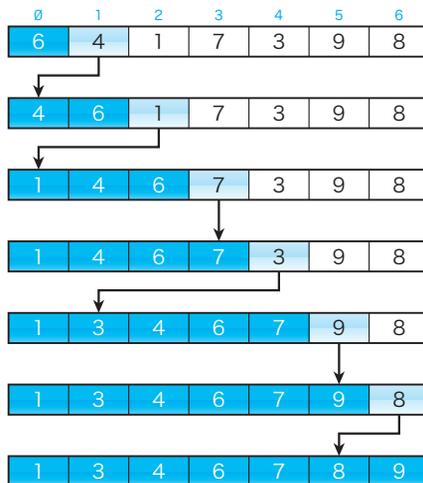
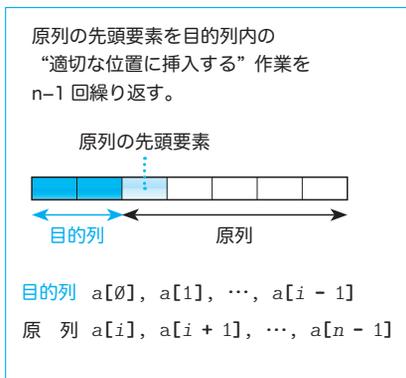


Fig.6-12 単純挿入ソートの手順

このとき、 $i$  を  $1, 2, \dots, n - 1$  とインクリメントしながら、インデックスが  $i$  の要素を取り出して、それを目的列内の適切な位置に挿入します。そのため、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、そこから先は走査の必要がありませんので、そこに挿入する値を代入します。

- ①～③… 3より小さい要素に出会うまで一つ左側の要素を代入する操作を繰り返す。  
④ … ストップした位置に3を代入。

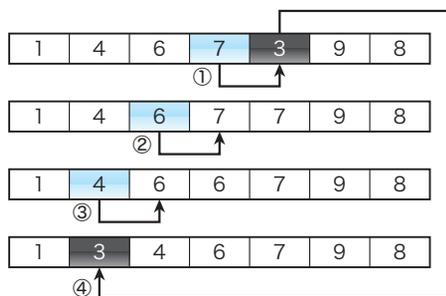


Fig.6-13 単純挿入ソートにおける《挿入》の手続き

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 (Column 1-7 : p.20) 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あいだ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

```
// 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
```

## 6

ソ  
ー  
ト

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。

なお、単純挿入ソートは、**シャトルソート** (shuttle sort) とも呼ばれます。

### ■ 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

要素の交換過程を詳細に表示するように書きかえた、単純選択ソートのプログラムを作成せよ。

右に示すように、未ソート部の先頭要素の上に記号文字 '\*' を表示して、未ソート部の最小要素の上に記号文字 '+' を表示すること。

※本問は、前節の内容に関する演習問題である。

	*				+		
6	4	8	3	1	9	7	
	*			+			
1	4	8	3	6	9	7	
		*		+			
1	3	8	4	6	9	7	
…以下省略…							

### 演習 6-7

要素の挿入過程を詳細に表示するように書きかえた、単純挿入ソートのプログラムを作成せよ。

右に示すように、着目要素の下に記号文字 '+' を表示し、挿入される位置の要素の下に記号文字 '^' を表示し、それらのあいだを記号文字 '-' で埋めること。

なお、挿入が行われない（要素の移動が必要ない）場合は、着目要素の下に '+' のみを表示する。

6	4	8	5	2	9	7	
~-----+							
4	6	8	5	2	9	7	
		+					
4	6	8	5	2	9	7	
	^-----+						
4	5	6	8	2	9	7	
~-----+							
…以下省略…							

### 演習 6-8

単純挿入ソートにおいて、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

単純挿入ソートは、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られる。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。