

# 第4章 ソートアルゴリズム

本章からアルゴリズムを学習します。まずはアルゴリズムの具体例を学ぶのに適した基本的なソートアルゴリズム (sort algorithm) について解説します。ソート (sort) とは、データ構造内の要素をある基準に従って整列させることです。たとえば、配列に格納された整数値を昇順に並べ替える処理がソートにあたります。

本書では、本題である要素の位置指定などを単純に実現するために、Python 標準ライブラリのリストを配列として利用します。本書で用いる例では、配列の要素を整数値としますが、比較演算が可能な型であれば動作します。また、簡略化のため、配列 arr の部分配列である  $arr[x] \sim arr[y]$  を表すときは、 $arr[x:y]$  と記述します。

## 4.1

# ソートアルゴリズムの概要

まず、ソートアルゴリズムがどのような場面で利用されるか考えてみましょう。一番最初に思いつくのが、数値の整列です。たとえば、入学試験の受験者と点数が手元にあるとします。一般的に点数の高い受験生から順番に合格させます。そのため、受験者の点数を基にして降順に並べ替える、といった作業が必要です。

このような処理を一般的に、特定のキーによるソート、と呼びます。エクセルなどのスプレッドシートを使用しているときによく使う機能です。エクセルに社員や学生の情報が格納されているとします。これらを社員番号・学籍番号や名前、年齢など特定のキーによって並べ替える処理にソートアルゴリズムを適応します。

他の例では、ビッグデータ解析においては膨大なデータを木構造(第6章)やグラフ構造(第7章)を用いて格納します。これらのビッグデータから何かを探索したりする場合の前処理として、データの集合をソートしたりします。この使用例は大学院の研究レベルの話になります。

本書では、挿入ソート(insertion)とバブルソート(bubble)、マージソート(merge)、クイックソート(quick)と呼ばれる4つのアルゴリズムを解説します。それぞれ計算量が異なります。データ数を  $n$ とした場合、挿入ソートとバブルソートの計算量は  $O(n^2)$ 、マージソートは  $O(n \log n)$  です。一番高速と言われるクイックソートの計算量ですが、平均計算量が  $O(n \log n)$  で、最悪の場合は  $O(n^2)$  かかります。

挿入ソートとバブルソートは、ちから技であり、ちょっとと考えれば考へ付くアルゴリズムです。マージソートとクイックソートは、ちょっと見には複雑ですが、挿入ソートやバブルソートよりも、ずいぶん計算量は少なく、高速に実行されます。アルゴリズム(考え方)の重要さがわかる例です。実装時には、それぞれの特徴を抑え、選択すると良いでしょう。

また、各アルゴリズムがどれほどのメモリを使用するか、といった領域的な計算量もアルゴリズムを評価する指標の1つです。プログラマにとって一番重要なのは、同一ハードにおける処理時間を見下す計算量になります。そのため、本書では時間的な計算量を主に説明します。

マージソートとクイックソートは、挿入ソートとバブルソートに比べて計算速度が速いということがわかります。これは統治分割法(divide-and-conquer method)と呼ばれるアルゴリズム分野において極めて重要なテクニックに基づいて設計されているからです。統治分割法については第4.4節で解説します。

多くの場合、クイックソートが一番高速だと言われています。しかし、データがすでに規則に従って整列している場合など特殊なケースでは、クイックソートの実行速度は遅くなります。アルゴリズム毎に特徴があるので、それぞれ解説していきます。

# 4.2

## 挿入ソート

要素を、先の 2 つと比較し、並べ替えたとします。そして、その次へといった具合に、**挿入ソート (insertion sort)** は、ソート済みの部分配列に適切な場所に要素を挿入するこ

とにによって、データの集合をソートします。

### 4.2.1 挿入ソートの概要

挿入ソートの計算量は  $O(n^2)$  です。アルゴリズム全般に言えることですが、 $n^2$  ということとはデータが格納された配列のすべての要素を  $n \times n$  回チェックすることを意味します。ループ構成の中にループ構造を入れて配列の各要素にアクセスします。これをネストループ (nested loop) と呼びます。

以下のネストループを考えてください。

```
for i in range(1, n):
    for j in range(i - 1, -1, -1):
```

#### 処理 A

外側の for ループでループカウンタ  $i$  の値が 1 から  $n-1$  になるまで、ループ内の処理が繰り返されます。ループカウンタ  $i$  の値は 1 ずつ増加するので、イテレーション (ループ 1 回分の処理) の回数は合計  $n-1$  回です。内側の for ループではループカウンタ  $j$  の値が  $i-1$  から 0 になるまで繰り返されます。ループカウンタ  $j$  の値は 1 ずつ減っていく、イテレーションの回数は合計  $i$  回です。処理 A は合計  $(n-1) \times i$  回繰り返されます。内側ループで  $i$  の値が最大で  $n-1$  になるので、この場合も計算量は  $O(n^2)$  です。

挿入ソートの基本は、ソート済みの部分配列に追加要素を適切な場所に挿入することです。データを保持する配列の変数名を  $arr$  とします。インデックス 0 の要素だけに着目してください。部分配列を  $arr[0]$  とすると、1 つしか要素がないので  $arr[0]$  はすでにソートされています。次に  $arr[0:1]$  の部分配列ですが、要素が 2 つだけです。この 2 つの要素の大小を比べるだけです。次に  $arr[0:1]$  をソートすることができます。視覚化すると図 4.1 に示す配列の状態になります。 $arr[0:1]$  がソート済みで、 $arr[2:n-1]$  が未ソートの状態です。

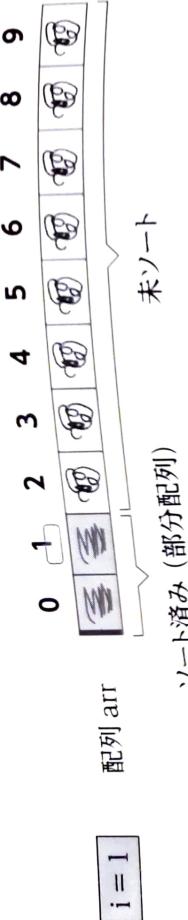


図 4.1 挿入ソートの外側の for ループ (1/3)

その次は `arr[0:2]` の部分配列に注目します。`arr[0:1]` がすでにソート済みなので、`arr[2]` の要素を `arr[0:2]` の適切な場所に挿入することによって、部分配列 `arr[0:2]` をソートすることができます。図 4.2 に示す配列のようになります。



図 4.2 挿入ソートの外側の for ループ (2/3)

同様に `arr[3]` の要素をソート済みの `arr[0:3]` の適切な場所に挿入し、部分配列 `arr[0:3]` をソートします。図 4.3 に示す配列のようになります。このように、前述のネストループの外側の for ループで、部分配列である `arr[0:i]` をソートしていきます。

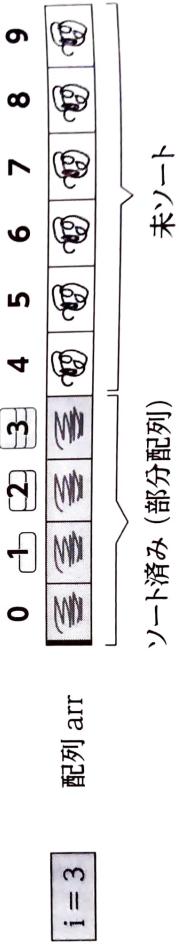


図 4.3 挿入ソートの外側の for ループ (3/3)

では、ネストループの内側の for ループを見ていきましょう。ループカウンタ `j` の値を、すでにソート済みの部分配列のインデックスである `i-1 ~ 0` に変化させて、`arr[i]` の要素を適切な場所に挿入します。たとえば、ループカウンタ `i` の値が 3 のときの配列の初期状態が図 4.4 の状態だつたとします。`arr[0:2]` がソート済みで、ここに `arr[3]` を挿入して部分配列 `arr[0:3]` をソートします。部分配列の中身が [2, 5, 8] となっていますので、整数値 3 は整数値 2 と 5 の間 (インデックス 1) に挿入すれば良いのです。この処理を行うために、ループカウンタ `j` の値を  $i-1$  から 0 に変化させて `arr[3]` と `arr[j]` の大小を確認していきます。

整数値 3 を整数値 2 と 5 の間に挿入したい。

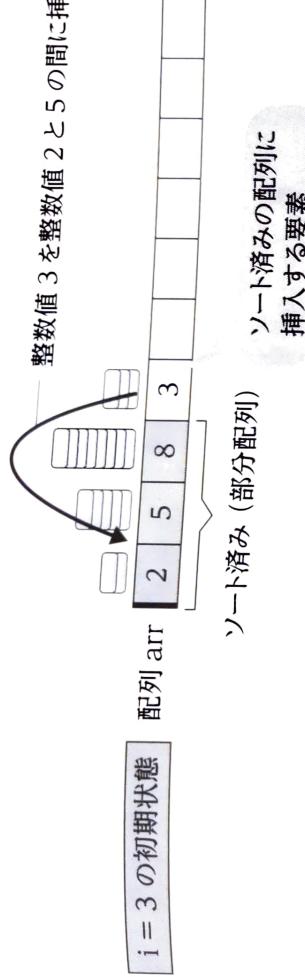


図 4.4 挿入ソートの内側 for ループ (1/2)

部分配列に対する挿入処理なので、 $O(n)$  の計算量がかかります。挿入後の配列は、図 4.5 に示すように arr[0:3] がソートされた状態になります。



図 4.5 挿入ソートの内側 for ループ (2/2)

本書での実装は、内側のループは for ループ構造の代わりに while ループ構造を利用します。理由は追加要素を挿入するインデックスが明らかになった時点で、内側のループを中断するためです。

## 4.2.2 挿入ソートの実装

挿入ソートの実装例をソースコード 4.1 (insertion.py) に示します。

ソースコード 4.1 挿入ソートの実装プログラム

[~/ohm/ch4/insertion.py](#)

2 行目～11 行目 插入ソートを実行する insertion\_sort 関数  
13 行目～20 行目 main 関数

01 # 插入ソート  
02 def insertion\_sort(arr, n):

```

04     val = arr[i]
05
06     # arr[i]をソート済みのarr[0]~arr[i - 1]のいずれかへ挿入
07     j = i - 1
08     while j >= 0 and arr[j] > val: # 内側のループ
09         arr[j + 1] = arr[j]
10         j -= 1
11     arr[j + 1] = val
12
13 if __name__ == "__main__":
14     # データの宣言と初期化
15     arr = [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
16     print("ソート前: ", arr)
17
18     # 要素をソート
19     insertion_sort(arr, len(arr))
20     print("ソート後: ", arr)

```

## ■ main 関数の説明

13行目～20行目で main 関数を定義しています。15行目で整数の集合をデータとしてもつ配列を宣言し（リストを活用）、変数 arr に格納します。整数はランダムな値を設定します。本書では [5, 9, 2, 1, 7, 3, 4, 6, 8, 0] としました。各要素は異なる値をもつことを前提としてソースコードを記述しますが、同じ値をもつ要素が複数あったとしても動作します。16行目で、ソート前の配列 arr の中身を表示します。

19行目で insertion\_sort 関数を呼び出し、配列 arr をソートします。20行目でソート後の配列を表示します。昇順にソートするため [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] と表示されます。

## ■ insertion\_sort 関数（挿入ソートアルゴリズムの定義）の説明

挿入ソートアルゴリズムは 2 行目～11 行目の insertion\_sort 関数で定義しています。引数はデータが格納された配列 arr と配列の大きさを表す変数 n です。3 行目で for ループを定義し、ループカウンタ i を 1 ~ n-1 まで 1 ずつ増加させ、4 行目～11 行目の処理を繰り返します。部分配列の大きさが 1 のときは、ソートする必要がないため、i の値は 1 からになっています。この外側の for ループで arr[0:i] をソートします。

for ループの内側で、arr[i] の要素をソート済みである部分配列 arr[0:i-1] の適切な場所に挿入します。この処理は第 3.1 で解説した配列の挿入とほぼ同じです。まず、変数 val を宣言し、arr[i] の値を代入します。内側のループでは while 文を用いるので、7 行目でループカウンタ j を定義して初期値を i-1 に設定します。8 行目～10 行目が while ループになります。ループカウンタ j が 0 以

上の値、かつ  $\text{arr}[i]$  が  $\text{val}$  より大きい、といった条件式を設定します。9行目と10行目で部分配列の要素を1つずつ後ろにずらし、 $j$  の値を減算します。while ループの条件式が False になったとき、 $\text{val}$  の値は  $\text{arr}[j]$  以下、かつ  $\text{arr}[j+1]$  より小さくなっているはずです。したがって 11 行目で  $\text{val}$  の値を  $\text{arr}[j+1]$  に代入します。

外側の for ループのループカウンタ  $i$  の値が 4 のときの処理内容を用いて、具体例を示します。 $i$  が 4 のときの配列の初期状態を図 4.6 に示します。変数  $\text{val}$  の値は  $\text{arr}[4]$  に格納されている整数値の 7 が保存され、内側ループカウンタ  $j$  の値は 3 で初期化されます。

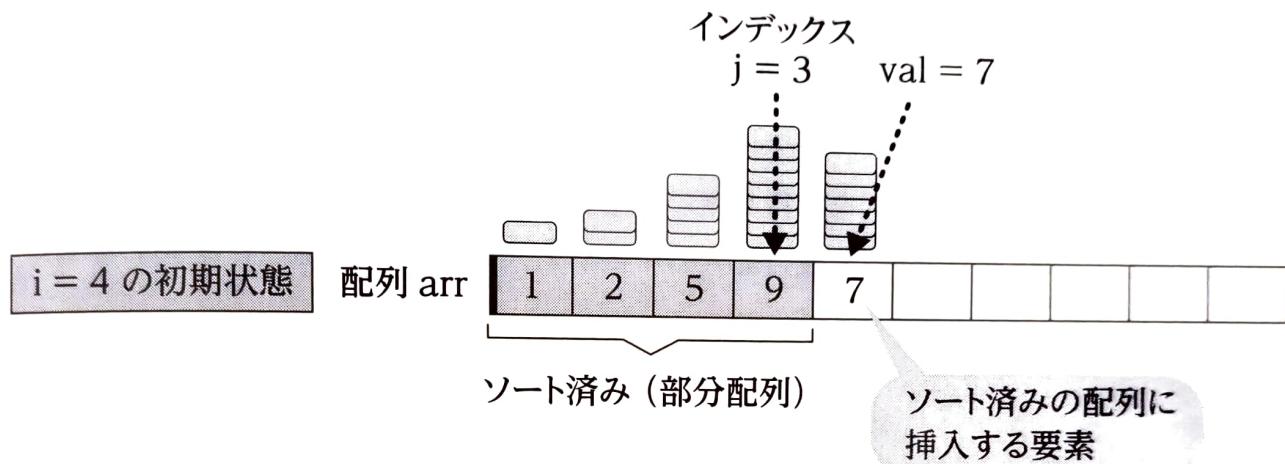


図 4.6  $i = 4$  の場合の内側ループの処理 (1/3)

内側の while ループに入り、 $j >= 0$  かつ  $\text{arr}[j] > \text{val}$  が True である限り、ループを繰り返します。この例では  $j$  が 2 のときに条件式が False となり、while ループを抜けます。そのときの状態を図 4.7 に示します。 $\text{arr}[j:i-1]$  の要素が1つずつ後ろにずれます。この例では  $\text{arr}[2]$  だけなので、 $\text{arr}[3]$  と  $\text{arr}[4]$  の要素が 9 という整数値になっています。

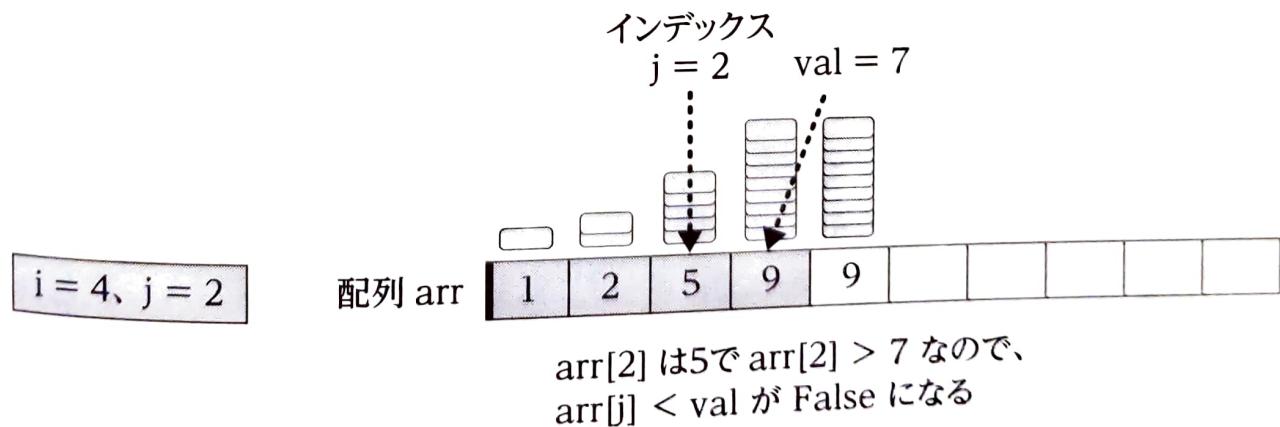


図 4.7  $i = 4$  の場合の内側ループの処理 (2/3)

while ループを抜け、11 行目の  $\text{arr}[j+1] = \text{val}$  が実行され、 $\text{arr}[2]$  に整数値 7 が格納されます。そのときの状態を図 4.8 に示します。 $i$  が 4 のときのイテレーションが終了した時点で、 $\text{arr}[0:4]$  がソート済みになります。

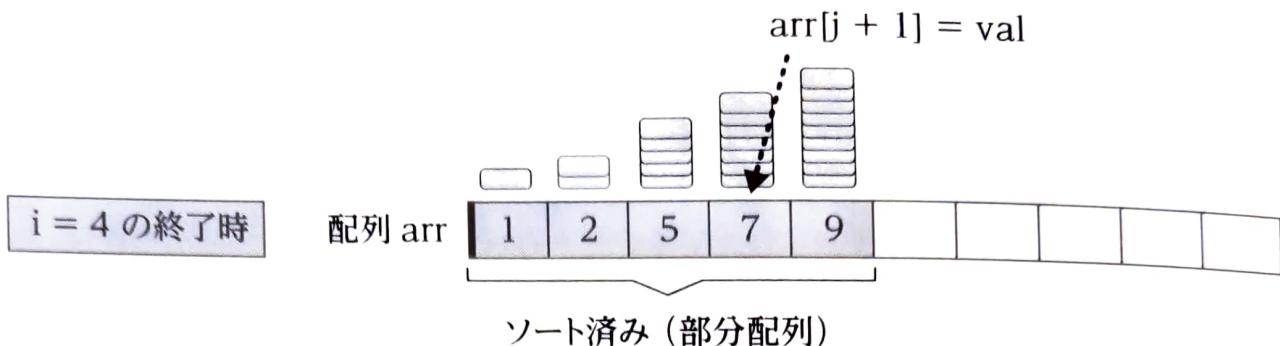


図 4.8 i = 4 の場合の内側ループの処理 (3/3)

## ■ 挿入ソートプログラムの実行

ソースコード 4.1 (insertion.py) を実行した結果をログ 4.2 に示します。2 行目にソート前の配列の中身が表示され、3 行目にソート後の配列の中身が表示されます。確かに配列の要素が昇順に並べ替えられています。

### ログ 4.2 insertion.py プログラムの実行

```
01 $ python3 insertion.py
02 ソート前: [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
03 ソート後: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

外側の for ループの各イテレーションが終了した時点での配列は以下のようになります。イテレーション毎に  $arr[0:i]$  の部分配列がソートされていることが確認できます。ご自分でも数字を入れ替えて、確認してみてください。内側の while ループを抜けた 12 行目に、`print("i = ", i, ":", arr)` という命令を追加すると良いでしょう。

- i = 1 の終了時: [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
- i = 2 の終了時: [2, 5, 9, 1, 7, 3, 4, 6, 8, 0]
- i = 3 の終了時: [1, 2, 5, 9, 7, 3, 4, 6, 8, 0]
- i = 4 の終了時: [1, 2, 5, 7, 9, 3, 4, 6, 8, 0]
- i = 5 の終了時: [1, 2, 3, 5, 7, 9, 4, 6, 8, 0]
- i = 6 の終了時: [1, 2, 3, 4, 5, 7, 9, 6, 8, 0]
- i = 7 の終了時: [1, 2, 3, 4, 5, 6, 7, 9, 8, 0]
- i = 8 の終了時: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
- i = 9 の終了時: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## 4.2.3 挿入ソートの特徴

**挿入ソートの特徴**としては、すでにデータが整列されているときは高速になることです。たとえば、配列の中身が  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$  だとします。ソースコード 4.1 の 7 行目で、内側の while ループのループカウンタ  $j$  が  $i - 1$  で初期化されます。次の 8 行目で while 文の条件判定式で  $arr[j] > val$  かどうかを確認しますが、配列  $arr$  がすでにソート済みである場合は、必ず False になります。すなはって計算量は、外側 for ループの繰り返し回数に依存するため、 $O(n)$  となります。

整列済みのデータはそもそもソートする必要がないと感じるかもしれません、データを受け取った時点では整列されているかどうかはわかりません。また受け取ったデータは整列済みでなくても、ほぼ整列されている場合もあります。

```
j = i - 1
while j >= 0 and arr[j] > val:
    whileループ内の処理
```

このようにデータがすでにソート済みであれば、 $O(n)$  の処理時間しかかかりません。そのため、ほとんどソート済みの配列をソートする場合に適しています。

## 4.3 バブルソート

バブルソート (bubble sort) は、配列の隣り合う要素の大小を比較しながら、並べ替えを行うアルゴリズムです。計算量は  $O(n^2)$  ですが、実際の実行時間は挿入ソートに比べて遅いです。しかし、ソートとはどういうことかの理解にはちょうどよいアルゴリズムです。しかし、実装が簡単といつても、挿入ソートよりは行数があるので、ちょっと見には、難しいと思います。

## 4.3.1 バブルソートの概要

計算量が  $O(n^2)$  であるため、ループ構造をネストさせて配列を走査します。なお、走査とは、配列の各要素を 1 つずつ確認することです。バブルソートの場合は以下のようない斯特ループになります。

```
for i in range(0, n - 1):
    for j in range(n - 1, i, -1):
        処理A
```

外側の for ループのループカウンタ  $i$  は 0 から  $n - 2$  の値を取り、内側の for ループのループカウ

ンタ j は  $n-1$  から  $i+1$  の値を取ります。処理 A は合計で  $O(n^2)$  回繰り返されます。外側の for ループの各イテレーションで配列の先頭から部分的にソートします。具体的には、 $i=0$  のときのイテレーション終了時に部分配列 arr[0] がソートされ、 $i=1$  のときのイテレーション終了時に部分配列 arr[0:i] がソートされます。以下同様です。一般化すると各イテレーションで、部分配列 arr[0:i] がソートされます。ソート済みの部分配列の大きさを増やしていく、といった点では挿入ソートと同じです。

同じです。

内側の for ループでは、配列の後ろから隣接する要素を交換しながら、未ソートの部分から一番小さな値を前に移動させる、といった処理を行います。そのため、ループカウンタ  $j$  の値は配列 arr の最後尾のインデックスである  $n-1$  から始まります。簡単な例を **図 4.9** に示します。図のステップ 1 のおり、大きさが 4 の配列 arr の状態が  $[5, 1, 4, 3]$  だとします。最後尾の要素から隣接する要素の大小を比べて、必要に応じて要素を交換していきます。まず、 $arr[3]$  が 3,  $arr[2]$  が 4 です。 $arr[3] < arr[2]$  ので、この 2 つの要素を交換します。交換後の配列の内容を **図 4.9** のステップ 2 に示します。配列の中身が  $[5, 1, 3, 4]$  となります。次に  $arr[2]$  と  $arr[1]$  を比較しますが、 $arr[1]$  の値のほうが  $arr[2]$  より小さいので、交換は行いません。最後に **図 4.9** のステップ 3 に進みます。 $arr[0]$  と  $arr[1]$  を比較すると、 $arr[1] < arr[0]$  なので、この 2 つの要素を交換します。

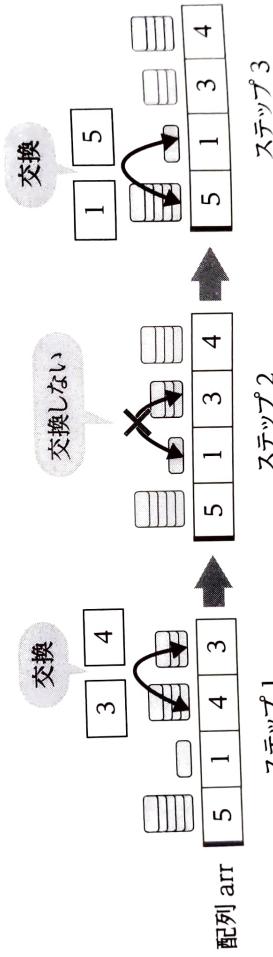
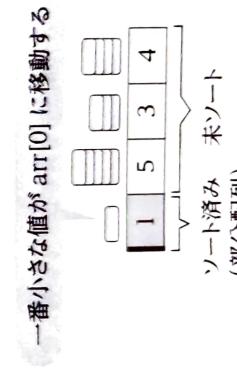


図 1.2 バイオニアードに掛ける交換原理 (1/3)

最終的に配列の状態は、図 4.10 に示すとおりになります。配列 arr の中で一番小さな値である整数値 1 が arr[0] に移動します。すなわち、未ソートの部分配列内で一番小さな値をもつ要素が、一番左側（インデックスの値が小さい側）に移動します。



卷之三

これが内側のforループのイテレーションの動作です。同様の処理を外側のforループで繰り返し、配列全体をソートします。

## 4.3.2 バブルソートの実装

ソースコード4.3 (bubble.py) にバブルソートの実装例を示します。本書の例では、配列のインデックスの値が小さい方から昇順に要素をソートします。同じ昇順に並べ替えるバブルソートでも、配列のインデックスが高い方から降順にソートする実装する例もあります。

ソースコード4.3 バブルソートの実装プログラム

~/ohm/ch4/bubble.py

### ソースコードの概要

2行目～5行目	配列の要素を入れ替える swap 関数の定義
8行目～12行目	バブルソートを実行する bubblesort 数の定義
14行目～21行目	main 関数の定義

```

01 # スワップ関数
02 def swap(arr, i, j):
03     tmp = arr[i]
04     arr[i] = arr[j]
05     arr[j] = tmp
06
07 # バブルソート
08 def bubble_sort(arr, n):
09     for i in range(0, n - 1): # 外側のループ
10         for j in range(n - 1, i, -1): # 内側のループ
11             if arr[j] < arr[j - 1]:
12                 swap(arr, j, j - 1)
13
14 if __name__ == "__main__":
15     # データの宣言と初期化
16     arr = [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
17     print("ソート前: ", arr)
18
19     # 要素をソート
20     bubble_sort(arr, len(arr))
21     print("ソート後: ", arr)

```

## ■ main 関数の説明

14行目～21行目でmain関数を定義していますが、処理内容は挿入ソートのソースコードのmain関数とほぼ同じです。データの集合である[5, 9, 2, 1, 7, 3, 4, 6, 8, 0]を昇順に並べ替えて、ソート後の配列の中身を表示する処理になっています。違いはソートをするときに、20行目でbubble\_sort関数を呼び出している箇所だけです。

## ■ swap 関数(配列の要素の交換)の説明

2行目～5行目のswap関数では配列の要素を交換します。引数として配列arrと2つのインデックスを変数iと変数jで受け取ります。処理内容はarr[i]とarr[j]を入れ替えるだけです。バブルソート内で使用するので、このように別途swap関数を定義しました。

## ■ bubble\_sort 関数(バブルソートアルゴリズムの定義)の説明

バブルソートアルゴリズムは8行目～12行目で定義されています。引数としてデータが格納された配列とその大きさを変数arrとnで受け取ります。forループをネストし、交換処理を行います。内側のループでは、まず11行目でarr[j]がarr[j-1]より小さいか否かを確認し、Trueあれば12行目でswap関数を呼び出し、arr[j]とarr[j-1]を入れ替えます。条件判定がFalseであれば、何もしません。

外側のforループのイテレーション毎に、未ソートの部分配列arr[i:n-1]の中から一番小さい値をもつ要素をarr[i]に移動させ、部分配列arr[0:i]をソートされた状態にします。これを繰り返すことによって配列全体をソートします。具体例は、前項で説明したとおりです。

## ■ バブルソートプログラムの実行

ソースコード4.3(bubble.py)を実行した結果をログ4.4に示します。2行目と3行目にソート前とソート後の配列の中身を表示しています。正しくソートされていることが確認できます。

### ログ4.4 bubble.py プログラムの実行

```
01 $ python3 bubble.py
02 ソート前: [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
03 ソート後: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

外側forループの各イテレーション終了後の配列の中身は以下のようになります。イテレーション毎に配列の先頭からソートされます。気になる読者は、bubble\_sort関数の各ループが終了したときに配列の中身を表示する命令を記述し、中身を確認してください。13行目に、print("i = ", i, ":", arr)という命令を追加すると良いでしょう。

- $i = 0$  の終了時:  $[0, 5, 9, 2, 1, 7, 3, 4, 6, 8]$
- $i = 1$  の終了時:  $[0, 1, 5, 9, 2, 3, 7, 4, 6, 8]$
- $i = 2$  の終了時:  $[0, 1, 2, 5, 9, 3, 4, 7, 6, 8]$
- $i = 3$  の終了時:  $[0, 1, 2, 3, 5, 9, 4, 6, 7, 8]$
- $i = 4$  の終了時:  $[0, 1, 2, 3, 4, 5, 9, 6, 7, 8]$
- $i = 5$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 9, 7, 8]$
- $i = 6$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]$
- $i = 7$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
- $i = 8$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

## ■ 4.3.3 バブルソートの特徴

バブルソートの特徴は実装が簡単であることです。ただし実行速度は非常に遅いです。あくまでアルゴリズムの基礎と実装方法を学ぶためのソートアルゴリズムです。

また、場合によっては最良計算時間が  $O(n)$  になります。ソースコード 4.3 の内側の for ループ内で、一度でも交換処理を行ったかどうかを確認する命令を追加したとします。もし、交換処理を一度も行わなかった場合、すでにデータはソートされているので、それ以上外側の for ループをイテレーションする必要はありません。その場合、外側の for ループを抜けても問題ありません。この場合、外側 for ループが 1 回、内側 for ループが  $n-1$  回実行され、計算量は  $O(n)$  になります。ただし最良計算時間が達成できるのは、要素がすでにソートされていた場合のみです。

最良計算時間が  $O(n)$  である点は挿入ソートと同じですが、バブルソートの場合は少しでも要素の交換があると速度が著しく遅くなります。そのため、多くの場合、挿入ソートより遅くなります。

## 4.4 マージソート

マージソート (merge sort) は、データの集合である配列を小さな部分配列に分割し、ソート済みの小さな部分配列を統合 (merge) しながらソートを行うアルゴリズムです。統治分割法 (divide-and-conquer method) と呼ばれる重要なテクニックを用います。

挿入ソートやバブルソートは、感覚的にアタリのつくソート手法です。しかし、マージソートや以降のクイックソートのアルゴリズムは学者が考えた画期的なものです。この考えをプログラムコード化することに手間がかかりますが、効果は絶大です。

## ■ 4.4.1 マージソートの概要

データの数を  $n$  とした場合、マージソートの計算量は  $O(n \log n)$  です。なお、 $n$  と  $\log n$  は線形

## ■ main 関数の説明

14行目～21行目でmain関数を定義していますが、処理内容は挿入ソートのソースコードのmain関数とほぼ同じです。データの集合である[5, 9, 2, 1, 7, 3, 4, 6, 8, 0]を昇順に並べ替えて、ソート後の配列の中身を表示する処理になります。違いはソートをするときに、20行目でbubble\_sort関数を呼び出している箇所だけです。

## ■ swap 関数（配列の要素の交換）の説明

2行目～5行目のswap関数では配列の要素を交換します。引数として配列arrと2つのインデックスを変数iと変数jで受け取ります。処理内容はarr[i]とarr[j]を入れ替えるだけです。バブルソート内で使用するので、このように別途swap関数を定義しました。

## ■ bubble\_sort 関数（バブルソートアルゴリズムの定義）の説明

バブルソートアルゴリズムは8行目～12行目で定義されています。引数としてデータが格納された配列とその大きさを変数arrとnで受け取ります。forループをネストし、交換処理を行います。内側のループでは、まず11行目でarr[i]がarr[j-1]より小さいか否かを確認し、Trueであれば12行目でswap関数を呼び出し、arr[j]とarr[j-1]を入れ替えます。条件判定がFalseであれば、何もしません。

外側のforループのイテレーション毎に、未ソートの部分配列arr[i:n-1]の中から一番小さい値をもつ要素をarr[i]に移動させ、部分配列arr[0:i]をソートされた状態にします。これを繰り返すことによって配列全体をソートします。具体例は、前項で説明したとおりです。

## ■ バブルソートプログラムの実行

ソースコード4.3(bubble.py)を実行した結果をログ4.4に示します。2行目と3行目にソート前とソート後の配列の中身を表示しています。正しくソートされていることが確認できます。

### ログ4.4 bubble.py プログラムの実行

```
01 $ python3 bubble.py
02 ソート前: [5, 9, 2, 1, 7, 3, 4, 6, 8, 0]
03 ソート後: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

外側forループの各イテレーション終了後の配列の中身は以下のようになります。イテレーション毎に配列の先頭からソートされます。気になる読者は、bubble\_sort関数の各ループが終したときには配列の中身を表示する命令を記述し、中身を確認してください。13行目に、print("i = ", i, "arr")という命令を追加すると良いでしょう。

- $i = 0$  の終了時:  $[0, 5, 9, 2, 1, 7, 3, 4, 6, 8]$
- $i = 1$  の終了時:  $[0, 1, 5, 9, 2, 3, 7, 4, 6, 8]$
- $i = 2$  の終了時:  $[0, 1, 2, 5, 9, 3, 4, 7, 6, 8]$
- $i = 3$  の終了時:  $[0, 1, 2, 3, 5, 9, 4, 6, 7, 8]$
- $i = 4$  の終了時:  $[0, 1, 2, 3, 4, 5, 9, 6, 7, 8]$
- $i = 5$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 9, 7, 8]$
- $i = 6$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]$
- $i = 7$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$
- $i = 8$  の終了時:  $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$

### ■ 4.3.3 バブルソートの特徴

バブルソートの特徴は実装が簡単であることです。ただし実行速度は非常に遅いです。あくまでアルゴリズムの基礎と実装方法を学ぶためのソートアルゴリズムです。

また、場合によつては最良計算時間が  $O(n)$  になります。ソースコード 4.3 の内側の for ループ内で、一度でも交換処理を行つたかどうかを確認する命令を追加したとします。もし、交換処理を行わなかつた場合、すでにデータはソートされているので、それ以上外側の for ループをイテレーションする必要はありません。その場合、外側の for ループを抜けても問題ありません。この場合、外側 for ループが 1 回、内側 for ループが  $n-1$  回実行され、計算量は  $O(n)$  になります。ただし最良計算時間が  $O(n)$  である点は挿入ソートと同じですが、バブルソートの場合は少しでも要素の交換があると速度が著しく遅くなります。そのため、多くの場合、挿入ソートより遅くなります。