

時間と対数時間の差があるため、アルゴリズムによる計算量の差が大きいわけです。マージソートは、挿入ソート ($O(n^2)$) やバブルソート ($O(n^2)$) に比べて高速です。なお、線形時間とは $O(n)$ で、対数時間は $O(\log n)$ のことです。

■ 統治分割法の概念

マージソートを理解するには、まず統治分割法を理解する必要があります。統治分割法は、大きな問題を小さな副問題に分割 (divide) し、それらの副問題を結合 (conquer) していき、本来の問題を解く手法です。たとえば、人間がコンピュータを使用せずに、ランダムに並んだ 100 個の整数データを昇順にソートすることは大変です。しかし整数データの数が 10 度なら、コンピュータを使用しなくても比較的簡単に並べ替えることができます。同様のことをアルゴリズム内で行います。

統治分割法は分割ステップと統治ステップで構成されます。図 4.11 に 4 つの整数值をもつ配列 arr を示します。配列 arr のデータは [5, 1, 3, 4] とします。まずはこの大きさが 4 の配列を分割していきます。[5, 1, 3, 4] という配列を大きさが 2 の配列である [5, 1] と [3, 4] に分割します。さらに部分配列 [5, 1] を [5] と [1] といった大きさが 1 の配列に分割します。同様に部分配列 [3, 4] を [3] と [4] の 2 つの配列に分割します。

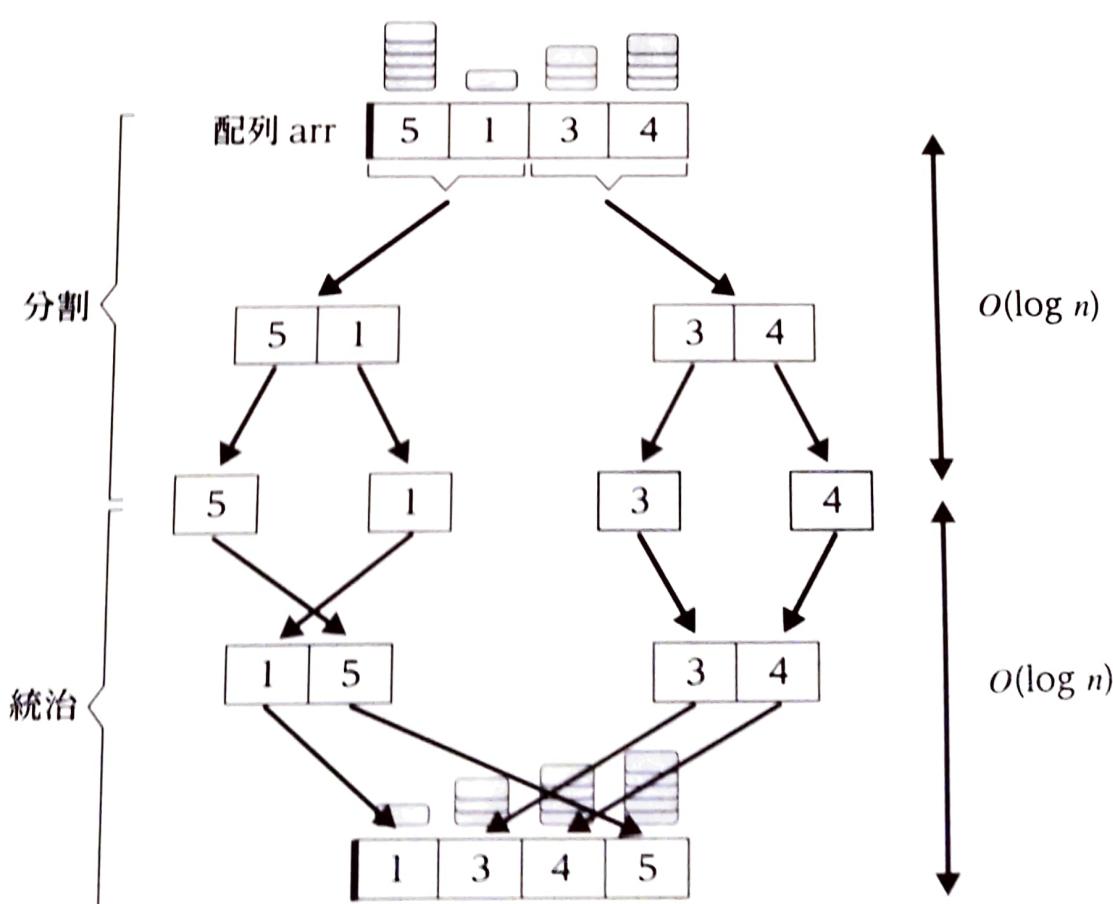


図 4.11 統治分割法の例

これ以上、分割できなくなれば、統治ステップを実行します。具体的には、部分配列をソートしながら 1 つの配列へマージしていきます。まず、2 つの部分配列 [5] と [1] は [1, 5] とソートし

ます。同様に [3] と [4] は [3, 4] となります。最後に大きさが 2 の部分配列 [1, 5] と [3, 4] を [1, 3, 4, 5] にマージします。

図 4.11 に示すように、分割ステップと統治ステップを視覚化すると、階層化することができます。それぞれの階層で 2 つの配列に分割するので、階層の高さは $\log n$ となります。また、統治ステップで 2 つの配列のマージ処理は、それぞれの階層で合計 $O(n)$ の時間がかかります。そのため、計算量が $O(n \log n)$ となります。

それでは分割ステップと統治ステップをもう少し具体的に見ていきます。

■ 分割ステップとは

プログラミングで配列を分割する方法としては、再帰構造を用います。以下に分割ステップの骨格を示します。divide 関数を定義し、その中で 2 つの divide 関数を再帰的に呼び出します。

```
def divide(arr, p, r):
    if p < r:
        q = math.floor((p + r) / 2)
        divide(arr, p, q)
        divide(arr, q + 1, r)
```

divide 関数は 3 つの引数を受け取ります。配列 arr と先頭と最後尾のインデックスを表す変数 p と変数 r とします。2 行目の if $p < r$: では、配列 arr が分割可能かどうかを判定します。配列 arr の大きさが 2 以上であれば、分割を行います。3 行目で真ん中のインデックスを計算し、変数 q とします。そして 4 行目と 5 行目で、再帰的に divide 関数を呼び出します。このようにして配列を分割していきます。配列 arr を [5, 1, 3, 4] に divide 関数を適応させた例を 図 4.12 に示します。まず、 $p=0$ 、 $r=3$ として divide(arr, 0, 3) を実行します。真ん中のインデックスの値は $q=1$ となるため、divide(arr, 0, 1) と divide(arr, 2, 3) が実行されます。

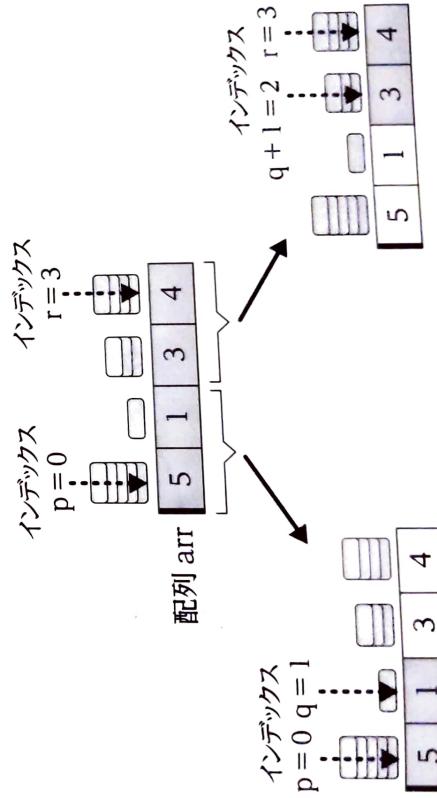


図 4.12 分割ステップの例 1

次に `divide(arr, 0, 1)` を実行したときの状態を図 4.13 に示します。配列 `arr` の部分配列である `arr[0:1]` は大きさが 2 なので、さらに分割が可能です。この時点では、変数 `p` と `r` の値はそれぞれ 0 と 1 となっています。まず、部分配列の真ん中のインデックスである `q` を計算します。`q` は 0 となるため、`divide(arr, 0, 0)` と `divide(arr, 1, 1)` が実行されます。それぞれの部分配列は大きさが 1 なのでこれ以上の分割は行われません。

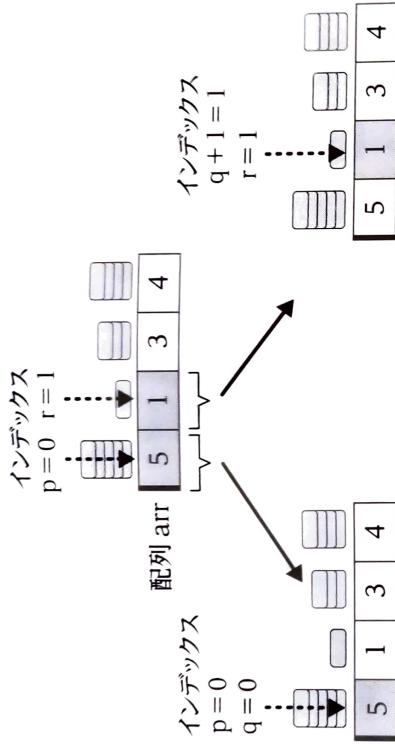


図 4.13 分割ステップの例 2

このように再帰構造を用いて分割を行います。

■ 統治ステップとは

統治ステップでは、2 つの部分配列をソートしながら 1 つの配列にマージします。分割ステップの骨格の最後に `conquer` 関数を加えたものを以下に示します。`conquer` 関数内で 2 つの配列のマージを行いますが、実装が複雑になるので具体的な処理内容はソースコードを見ながら解説します。まずはマージ方法のアイディアを説明します。

```
def divide(arr, p, r):
    if p < r:
        q = math.floor((p + r) / 2)
        divide(arr, p, q)
        divide(arr, q + 1, r)
        conquer(arr, p, q, r) # arr[p:q] と arr[q + 1:r] をマージする
```

`conquer` 関数は配列 `arr` と 3 つのインデックスを表す変数 `p` と `q` と `r` を引数として受け取ります。すなわち、2 つのソート済みの部分配列 `arr[p:q]` と `arr[q+1:r]` をマージし、部分配列 `arr[p:r]` をソートされた状態にします。

図 4.14 に 2 つの部分配列のマージ方法を示します。簡略のため配列変数名を `left` と `right` とし、

それぞれの中身は [1, 5, 8] と [3, 4, 7] とします。各配列のインデックスを表すを変数 i と j とします。それぞれ 0 で初期化します。ソート後の配列を arr とします。まず、 $left[i]$ と $right[j]$ ($left[0]$ と $right[0]$) を比較し、小さい値をもつ要素を $arr[0]$ に格納します。 $left[0] < right[0]$ のなので、 $left[0]$ を $arr[0]$ に格納します。 $left$ と $right$ はすでにソートされているので、 $arr[0]$ が一番小さな値をもちます。また、変数 i の値を 1 増やします。

1つ目の要素が確定したときの状態を図 4.15 に示します。変数 i と j の値がそれぞれ 1 と 0 になっています。再度 $left[i]$ と $right[j]$ ($left[1]$ と $right[0]$) を比較し、小さい方を $arr[1]$ に格納します。今回は $right[0]$ のほうが $left[1]$ より小さいので、 $arr[1]$ に $right[0]$ にある要素を格納します。今度は変数 j の値を 1 増やします。

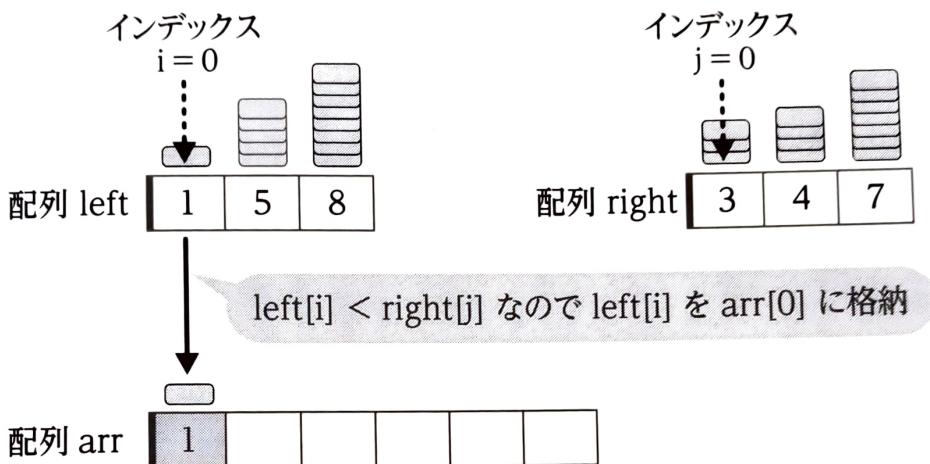


図 4.14 統治ステップの例 1

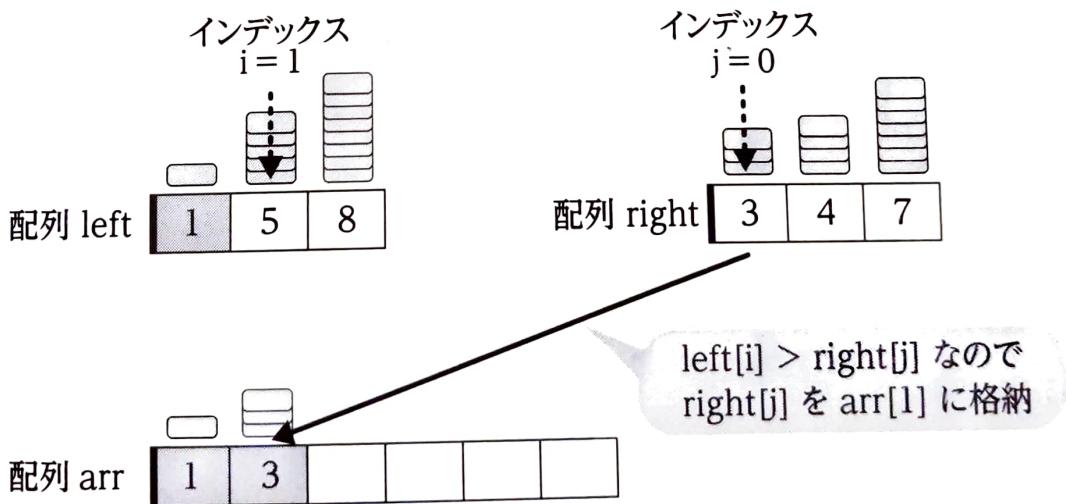


図 4.15 統治ステップの例 2

2つ目の要素が確定したときの状態を図 4.16 に示します。同様の処理を繰り返します。 $left[i]$ と $right[j]$ ($left[1]$ と $right[1]$) を比較し、小さい方を $arr[2]$ に格納します。 $left[1] > right[1]$ のなので、 $right[1]$ の整数值 4 を $arr[2]$ に格納し、変数 j の値を 1 増やします。

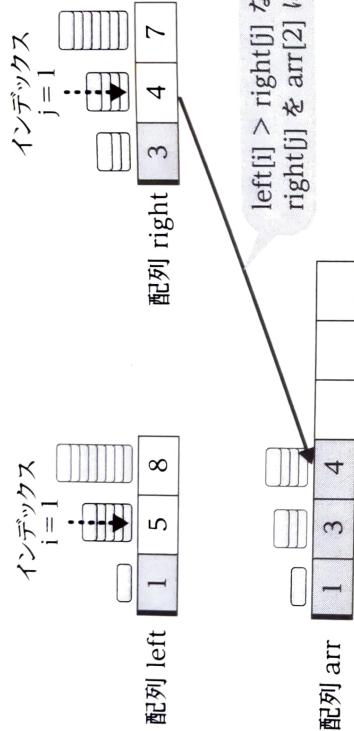


図 4.16 統治ステップの例 3

同様の処理を繰り返し、配列 left と right を最後まで走査すると、最終的に図 4.17 に示す状態になります。配列 arr の中身は $[1, 3, 4, 5, 7, 8]$ となり、ソートされた状態になります。

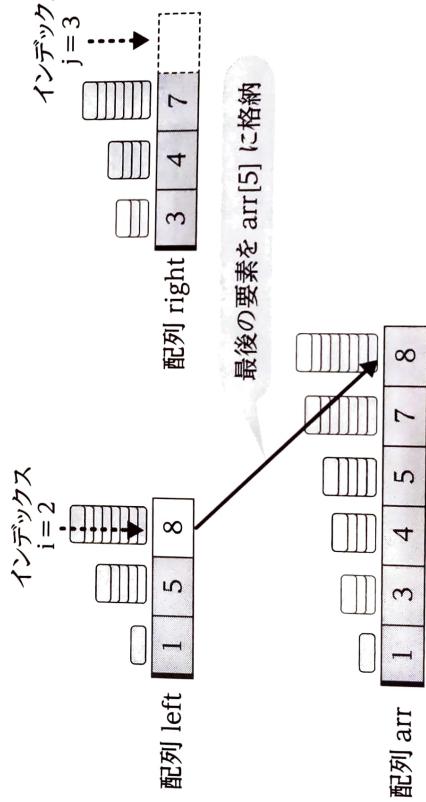


図 4.17 統治ステップの例 4

統治ステップでは、それぞれの部分配列を 1 回だけ走査します。図 4.11において、ある階層内のすべての部分配列の大きさの合計が n になるので、各階層でのマージ処理の計算量は $O(n)$ となります。

4.4.2 マージソートの実装

ソースコード 4.5 (merge.py) にマージソートの実装例を示します。実装方法は多々ありますが、本書で解説するのは典型的な一例です。

1行目では、小数点の切り捨て処理を行う `floor` 関数を使うため、`math モジュール`をインポートします。また、4行目で宣言した変数 `INFTY` は非常に大きな値を代入します。整数型で扱える最大値を設定したいところですが、Python 3.x の言語仕様では整数値の最大値が定義されていないため、本書では符号あり 32 ビットで扱える最大値である $2^{31}-1$ を無限大 (∞) として、読み替えて利用します。なお、連続した 2 つのアスタリスク (*) はべき乗を意味します。

38 行目～45 行目で `main` 関数を定義しています。処理内容は挿入ソートとバブルソートで解説した `main` 関数とほぼ同じなので、説明は省きます。

■ `merge_sort` 関数（統治分割法の骨格）の説明

29 行目～36 行目で `merge_sort` 関数を定義しています。関数名は異なりますが、前項で解説した統治分割法の骨格そのものです。`merge_sort` 関数は配列 `arr` とインデックスを表す変数 `p` と `r` を受け取ります。27 行目の `if` 文で、配列 `arr` が分割可能かどうかを判定し、`True` であれば `if` ブロックの中に入り分割と統治を行います。`False` の場合は何も行われないため、呼び出し元に処理が戻ります。

32 行目で真ん中のインデックスの値を計算し、33 行目と 34 行目で部分配列 `arr[p:q]` と `arr[q+1:r]` に対して、`merge_sort` 関数を適応させます。分割できなくなったところで、36 行目の `merge` 関数で 2 つの部分配列をマージします。

■ `merge` 関数（部分配列のマージ）の説明

7 行目～26 行目で、2 つの部分配列をマージするための `merge` 関数を定義しています。引数として配列 `arr` と 3 つのインデックスを表す変数 `p` と `q` と `r` を受け取ります。2 つの部分配列は `arr[p:q]` と `arr[q+1:r]` となります。ソート後の部分配列は `arr[p:r]` に格納されます。

9 行目と 10 行目で 2 つの部分配列の大きさを計算し、それぞれ変数 `n` と `m` に格納します。11 行目と 12 行目で、一時的に部分配列を保存する配列変数として `left` と `right` を定義します。大きさは、`n+1` と `m+1` とし、無限大を表す変数 `INFTY` を大きな値 ($2^{31}-1$) で初期化します。そして 13 行目～16 行目で、`arr[p:q]` の中身を `left` にコピーし、`arr[q+1:r]` の中身を `right` にコピーします。`left` と

`right` の大きさは、コピー元の配列より 1 つ大きいので、最後の要素には `INFTY` が格納されている状態です。

要素を1つ分大きくする理由ですが、まず前項で説明した図4.17内の変数jが指す配列rightのインデックスを見てください。配列rightの大きさは3なのでright[0:2]に要素が格納されていますが、変数jの値は3を指しています。プログラム実装上の問題として、片方の部分配列をすべて走査し終わった後に、もう片方の部分配列の残りの要素を配列arrに格納するために個別の処理を記述するとソースコードが複雑になります。そこでright[3]を∞として、leftの要素を配列arrにコピーするまで同様の処理を繰り返せば、ソースコードの記述が簡潔になります。すなわち、図4.17を図4.18の状態になるようにソースコードを記述します。そのため、配列leftとrightの大きさを要素1つ分大きくして、最後尾の要素に変数INFTYを格納しておきます。

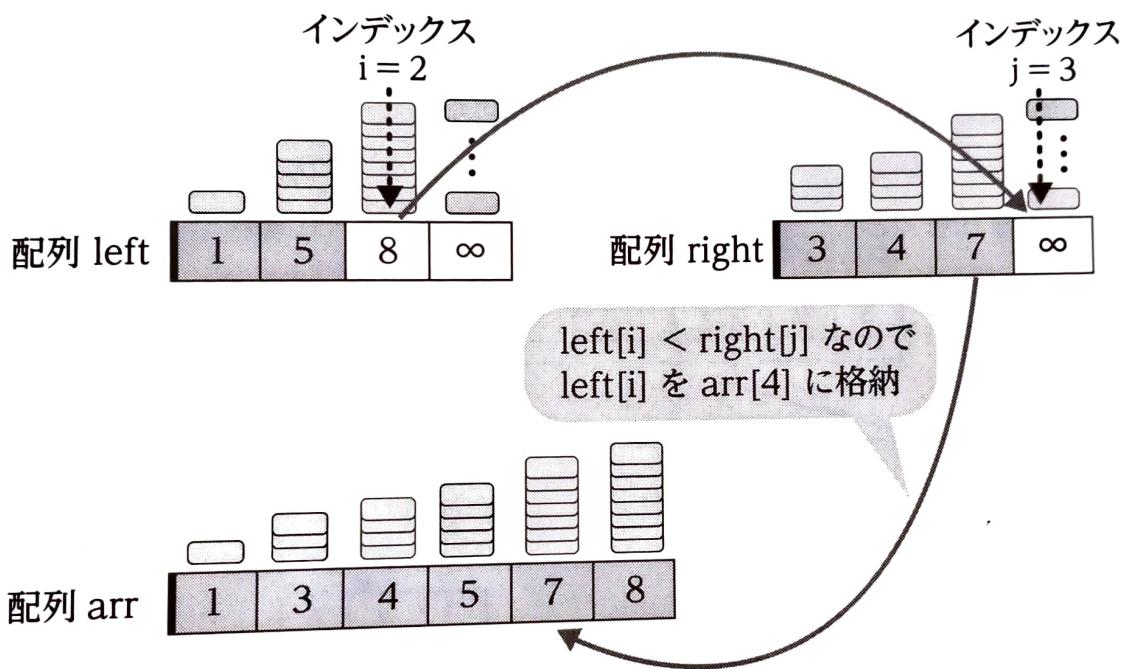


図 4.18 部分配列のマージ（図 4.17 の配列 left と配列 right をそれぞれ 1 つ大きくし、 ∞ を格納したときの状態）

を $\text{arr}[k]$ にコピーし、変数 i または j の値を 1 増加させます。
21 行目で、 $\text{left}[i]$ と $\text{right}[j]$ のどちらの要素が小さいかを判定します。大きくないほうの要素を $\text{arr}[k]$ に格納するため、True であれば 22 行目と 23 行目の命令を実行します。処理内容は $\text{left}[i]$ を $\text{arr}[k]$ に格納して、 i の値を 1 増やすだけです。False であれば 25 行目と 26 行目の命令を実行し、 $\text{right}[j]$ を $\text{arr}[k]$ に格納して、 j の値を 1 増やします。

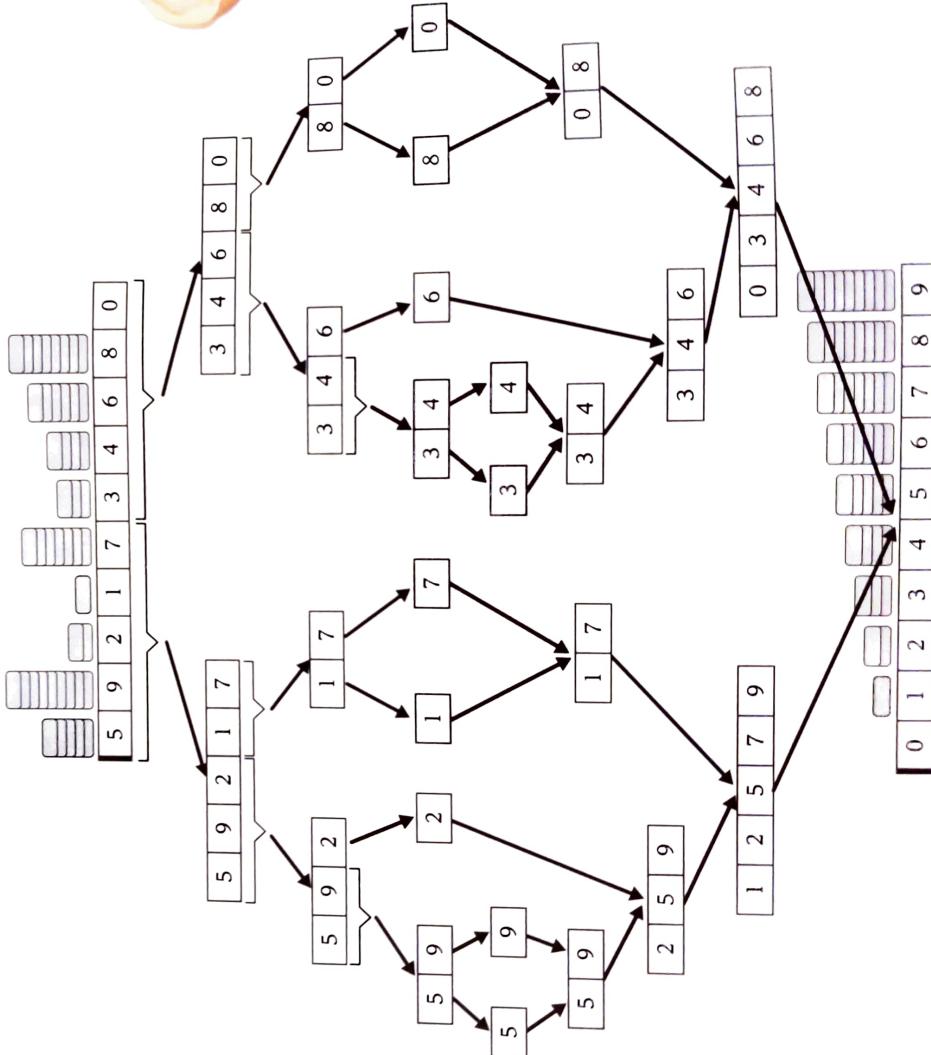
■ マージソートプログラムの実行

ソースコード **4.5** (merge.py) を実行した結果を図**4.6** に示します。マージソートアルゴリズムによって、正しくデータがソートされていることが確認できます。本書の例では、各要素は異なる整数値をもちますが、同じ整数値をもつ要素があつたとしても動作します。

ログ **4.6** merge.py プログラムの実行

```
01 $ python3 merge.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]
```

main 関数内で宣言した配列 arr の大きさが 10 なので、配列を分割していくと、大きさが奇数の部分配列がでてきます。もちろんその場合でも正しく動作します。配列 [5, 9, 2, 1, 7, 3, 4, 6, 8, 0] をソートした場合の処理を視覚化すると図**4.19** のようになります。



図**4.19** 配列 [5, 9, 2, 1, 7, 3, 4, 6, 8, 0] のマージソートの詳細

4.4.3 マージソートの特徴

マージソートの特徴は、計算量が $O(n \log n)$ であるため高速であることです。ただし多くの場合、次節で解説するクイックソートのほうが高速です。

また、マージソートは安定(stable)したソートアルゴリズムです。ここでの「安定」とは、並べ替えの基準であるキーの値が同じであるとき、ソート前の順序がソート後も保たれる、といった性質です。たとえば、学籍番号順に並べられた学生のリストがあつたとします。

学籍番号	名前	点数
1001	アリス	85
1002	ボブ	92
1003	クリス	85
1004	デイビッド	70

上記のリストを点数をキーとして降順にソートしたとします。ここでアリスとクリスの点数が同じ85点なので、ソート前の順序が保たれるならば、ボブ、アリス、クリス、デイビッドという順番にソートされます。すなわち、点数が同じでなければ学籍番号順にソートされます。もし、ソートアルゴリズムが安定していないければ、アリスとクリスの順序を入れ替わる可能性があります。このように安定した性質をもつソートアルゴリズムを**安定ソート**と呼びます。マージソートは安定ソートの一種であるため、安定して高速なソートを求めるときはマージソートが適しています。また、マージソートは並列化と相性が良いです。分割した部分配列をマージすることによってソートを行いますが、merge関数で行う個々のマージ処理は独立しているため、並列化することができます。

4.5

クイックソート

クイックソート(quick sort)は、その名前のとおり高速(quick)なソートアルゴリズムです。統治分割法に基づいたアルゴリズムであるため、データの集合がランダムに並んでいる場合は、平均 $O(n \log n)$ の計算時間でソートできます。ただし、すでにソートされている場合は最悪計算量である $O(n^2)$ も時間がかかります。

4.5.1 クイックソートの特徴

クイックソートを簡単に説明すると、ピボット(pivot)と呼ばれる適当な要素を配列内から選び、ピボットよりも小さい要素を前に移動させ、大きい要素を後ろに移動させます。この処理を再帰的に繰り返します。

クイックソートの骨格は以下のとおりです。conquer 関数の引数は、配列 arr とソートする部分配列の先頭と最後尾のインデックスを表す変数 p と r です。3行目の divide 関数内では、ピボットとなる要素を配列 arr 内から適当に選び、ピボットの値を基に要素を移動させます。移動後のピボットのインデックスを q とします。部分配列 arr[p:q-1] 内のすべての要素は arr[q] より小さい値をもち、部分配列 arr[q+1:r] 内のすべての要素は arr[q] より大きな値をもちます。4行目と5行目で、2つの部分配列に対して再度 conquer 関数を適応させます。

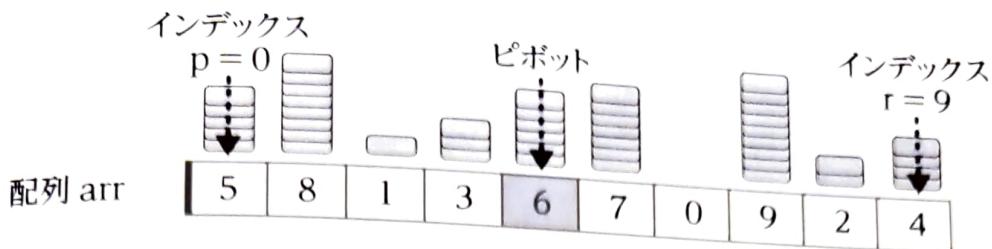
```
def conquer(arr, p, r):
    if p < r:
        q = divide(arr, p, r)
        conquer(arr, p, q - 1)
        conquer(arr, q + 1, r)
```

ピボットの選択方法と要素の移動方法は実装方法によって異なります。ここでは概念だけを説明し、具体的な実装例はソースコードを見ながら解説します。

■ 分割ステップ

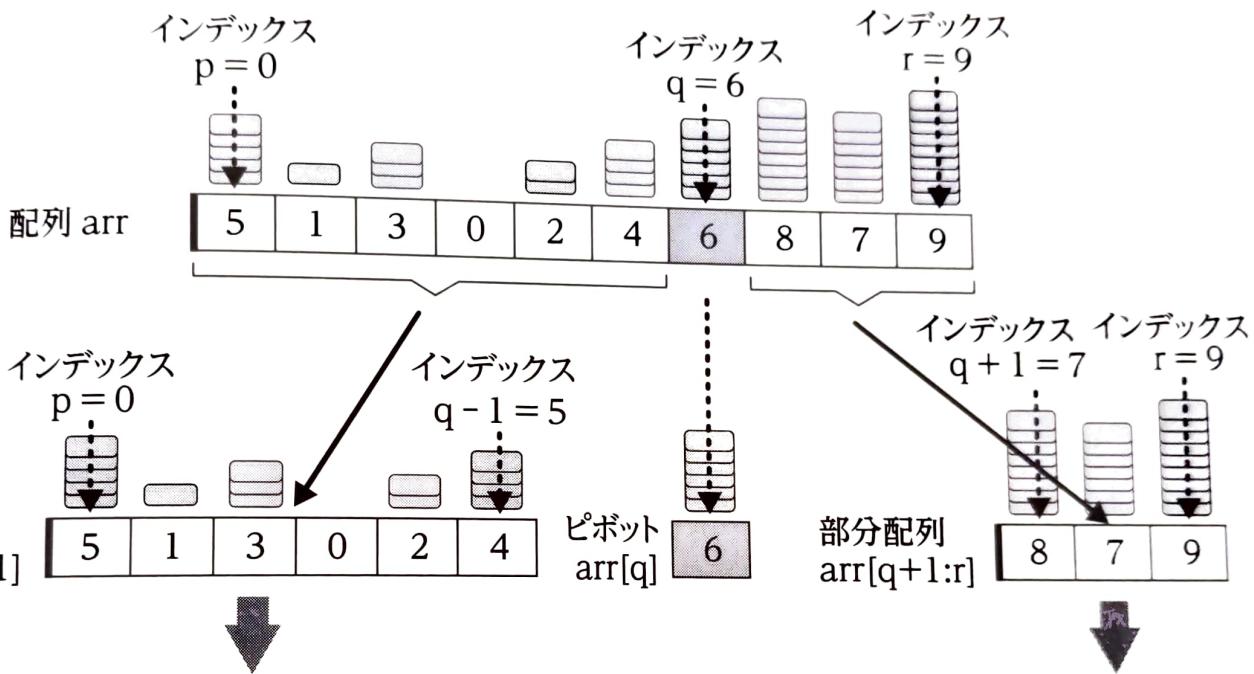
図 4.20 に配列 [5, 8, 1, 3, 6, 7, 0, 9, 2, 4] を用いた例を示します。配列の大きさが 10 なので、先頭と最後尾にインデックス p と r は、それぞれ 0 と 9 です。まず、1 conquer(arr, 0, 9) を実行すると、2 上記のクイックソートの骨格ソースコードの3行目の divide(arr, 0, 9) が実行されます。たとえば、ピボットとして整数値 6 が選択されたとします。ピボットの選び方として、さまざまな方法がありますが、ここでは気にしなくて構いません。後ほど実装例で簡単なピボットの選択方法を説明します。6 より小さな値を前に、大きな値を後ろに移動させると、arr[0:5] が [5, 1, 3, 0, 2, 4]、arr[6] が 6、arr[7:9] が [8, 7, 9] となります。どうやって要素を移動させるかは、ここでは無視して構いません。

① conquer(arr, 0, 9) の実行



② divide(arr, 0, 9) の実行

ピボットより小さな要素を前、
大きな要素を後ろへ移動する



③ conquer(arr, 0, 5) の実行

④ conquer(arr, 7, 9) の実行

図 4.20 クイックソートの分割ステップ

divide 関数の実行後、ピボットのインデックス q の値が 6 となり、配列が 2 つに分割されます。

③ 部分配列 $\text{arr}[0:5]$ に対して $\text{conquer}(\text{arr}, 0, 5)$ 、**④** 部分配列 $\text{arr}[7:9]$ に対して $\text{conquer}(\text{arr}, 7, 9)$ を実行します。部分配列が分割できなくなるまで、同様の処理をくり返します。

■ 統治ステップ (2 つの部分配列の結合)

統治ステップでは 2 つの部分配列を結合します。部分配列はすでにソートされているため、単純に結合するだけです。図 4.20 の最後に実行した $\text{conquer}(\text{arr}, 0, 5)$ と $\text{conquer}(\text{arr}, 7, 9)$ の処理が終了したとします。そのときの状態を図 4.21 に示します。部分配列 $\text{arr}[0:5]$ は $[0, 1, 2, 3, 4, 5]$ となり、ソートされた状態になります。もう一方の部分配列 $\text{arr}[7:9]$ も $[7, 8, 9]$ という状態なのでソート済みです。 $\text{arr}[0:5]$ のすべての要素は $\text{arr}[6]$ より小さく、 $\text{arr}[7:9]$ のすべての要素は

$\text{arr}[6]$ より大きいことが確認できます。あとはこれらの部分配列を結合するだけで、 $\text{arr}[0, 9]$ をソートすることができます。

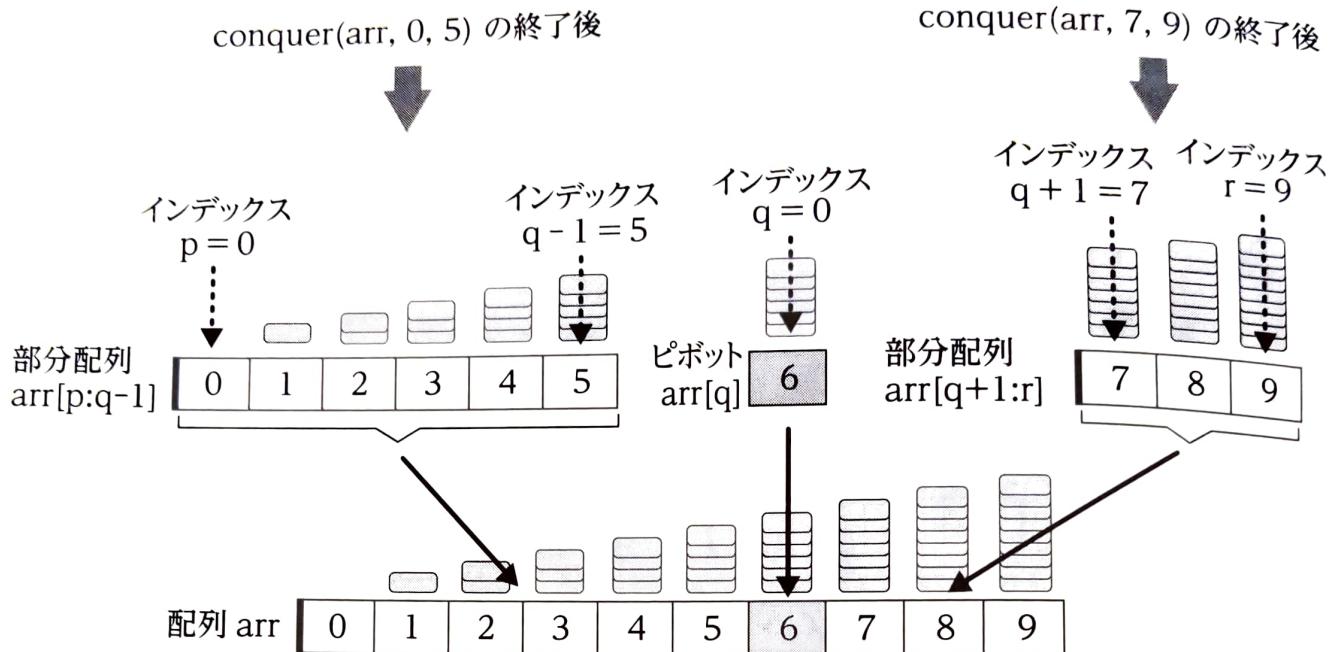


図 4.21 クイックソートの統治ステップ

クイックソートの平均計算量は $O(n \log n)$ です。分割ステップで分割する回数が平均で $\log n$ 回になるからです。また、それぞれの階層での要素の移動にかかる計算量は合計で $O(n)$ となります。そのため、平均計算量が $O(n \log n)$ となります。もし、すべての分割ステップで、片方の部分配列が $r-p$ 個の要素を含み、もう片方が空の部分配列に分割される状況が続くと、分割回数が n 回になります。この場合は $O(n^2)$ の計算量がかかります。

■ クイックソートの実装

ソースコード 4.7 (`quick.py`) にクイックソートの実装例を示します。さまざまな実装方法がありますが、本書での例は一番基本的な手法です。配列を分割するための関数名を `partition`、統治を行う関数名を `quick_sort` と名付けています。

ソースコード 4.7 クイックソートの実装プログラム

`~/ohm/ch4/quick.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 <code>swap</code> 関数の定義
8 行目～17 行目	分割ステップを行う <code>partition</code> 関数の定義
20 行目～24 行目	クイックソートを行う <code>quick_sort</code> 関数の定義
26 行目～33 行目	<code>main</code> 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 # 分割
08 def partition(arr, p, r):
09     pivot = arr[r]
10     i = p
11     for j in range(p, r):
12         if arr[j] <= pivot:
13             swap(arr, i, j)
14             i += 1
15     swap(arr, i, r)
16
17     return i
18
19 # クイックソート
20 def quick_sort(arr, p, r):
21     if p < r:
22         q = partition(arr, p, r)
23         quick_sort(arr, p, q - 1)
24         quick_sort(arr, q + 1, r)
25
26 if __name__ == "__main__":
27     # データの宣言と初期化
28     arr = [5, 8, 1, 3, 6, 7, 0, 9, 2, 4]
29     print("ソート前: ", arr)
30
31     # 要素をソート
32     quick_sort(arr, 0, len(arr) - 1)
33     print("ソート後: ", arr)

```

2行目～5行目の swap 関数と 26 行目～33 行目の main 関数は、前節までに登場した処理とはほぼ同じなので説明を省きます。具体例の説明のしやすさのため、配列の中身だけを [5, 8, 1, 3, 6, 7, 0, 9, 2, 4] に変更しました。

■ quick_sort 関数（クイックソートの実行）の説明

20行目～24行目でクイックソートを実行する quick_sort 関数を定義しています。引数は配列 arr と先頭と最後尾のインデックスを表す変数 p と r です。main 関数内で最初に quick_sort 関数を呼び出すときは、配列 arr 全体を指すために p と r の値はそれぞれ 0 と n-1 とします。なお、n は配列 arr の大きさです。関数名は異なりますが、クイックソートの概要で解説したソースコードの骨格と同じです。

■ partition 関数（配列の分割）の説明

8行目～17行目で、配列の分割を行う partition 関数を定義しています。引数は配列 arr と先頭と最後尾のインデックスを表す変数 p と r です。本書での実装では、最後尾の要素である arr[r] をピボットとして選択します。そのため、9行目で、変数 pivot に arr[r] の値を保存します。10行目で変数 i を宣言し、部分配列 arr の先頭インデックスである p で初期化します。

11行目～14行目の for ループで、ループカウンタ j の値を p～r-1 に 1 ずつ増加させて、ループ内で要素の移動を行います。このときに変数 i と j が指すインデックスの要素を必要に応じて交換します。12行目の if 文で arr[j] の要素が pivot 以下の値であれば、13行目で swap 関数を呼び出して arr[i] と arr[j] の要素を互いに交換します。そして 14行目で変数 i の値を 1 増加させます。言い換えると、arr[i] のより前にある要素は pivot 以下の値をもちます。

また、for ループを抜けた時点では arr[i] は pivot より大きいはずです。そのため、15行目では、arr[i] と arr[r] の要素を交換します。したがって、変数 i がピボットを指すインデックスになります。最後に 17行目で i の値を返します。

partition 関数の実行後、arr[p:i-1] には pivot 以下の値をもつ要素、arr[i] がピボットそのもの、arr[i+1:r] が pivot より大きな値をもつ要素が含まれています。ただし、部分配列 arr[p:i-1] と arr[i+1:r] の中身は未ソートです。

■ 具体例

文章だけではわかりにくいので、具体例を示します。[5, 8, 1, 3, 6, 7, 0, 9, 2, 4] をデータとしてもつ配列 arr に対して、partition(arr, 0, 9) を実行したときの初期状態を図 4.22 に示します。先頭と最後尾のインデックスを表す p と r はそれぞれ 0 と 9 です。arr[0:9] を操作する変数 i とループカウンタ j はともに 0 で初期化されています。まず、9 行目でピボットである arr[r] を pivot に保存します。

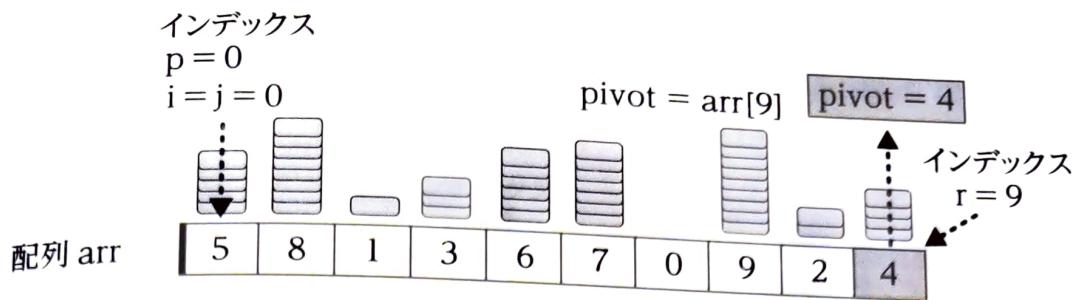


図 4.22 partition 関数の結果例 1 (partition(arr, 0, 9) を実行したときの初期状態)

11 行目の for ループを繰り返すと、j が 2 のときに 12 行目の if 文にある $\text{arr}[j] \leq \text{pivot}$ の条件判定が True になります。視覚化すると図 4.23 に示すように、i と j の値がそれぞれ 0 と 2 になります。

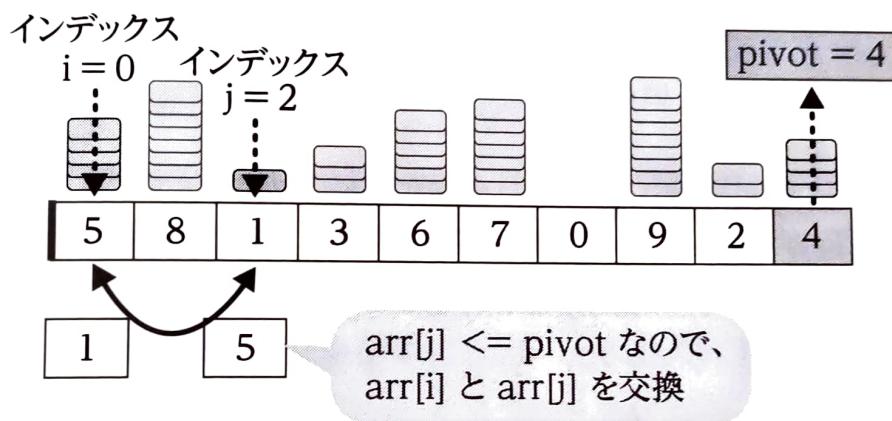
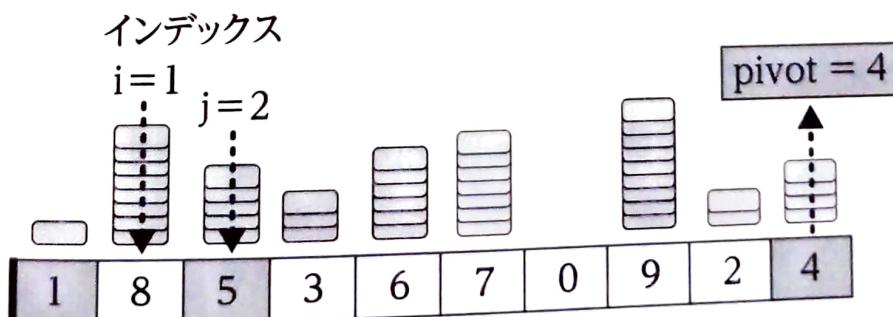


図 4.23 partition 関数の結果例 2

if ブロックに入り、 $\text{arr}[i]$ と $\text{arr}[j]$ の交換処理を行い、 $\text{arr}[i]$ に整数値 1、 $\text{arr}[j]$ に整数値 5 が格納されます。そして i の値を 1 増加させます。その後の状態は図 4.24 に示すとおりになります。 $\text{arr}[0]$ と $\text{arr}[2]$ の要素が入れ替わり、 i の値が 1 に更新されていることに注目してください。



```
swap(arr, i, j)
i += 1
```

図 4.24 partition 関数の結果例 3

引き続きループカウンタ j の値を増やし、for ループを繰り返します。次は j の値が 3 のとき、 $\text{arr}[3]$ の値が 0 なので、12 行目の if 文の条件判定で True になることがわかります。そのときの状態を図 4.25 に示します。今回は i の値が 1 なので、 $\text{arr}[1]$ と $\text{arr}[3]$ を交換すれば良いことがわかります。

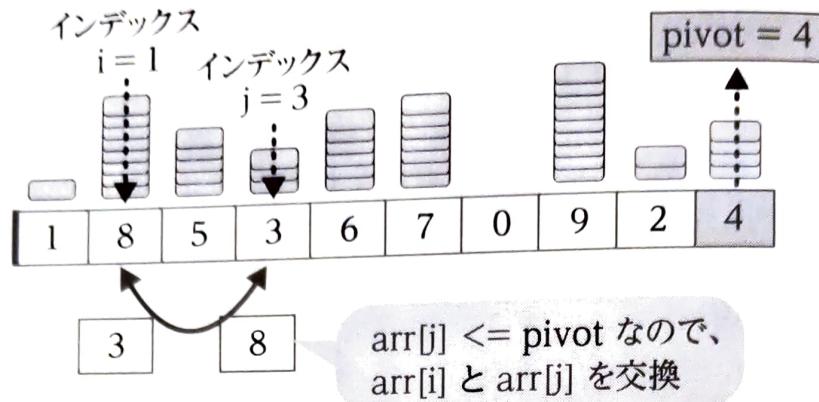


図 4.25 partition 関数の結果例 4

if ブロックの中に入り、swap 関数と実行し、 i の値を増加させたあとの状態を図 4.26 に示します。配列の $\text{arr}[0]$ と $\text{arr}[1]$ がピボットである 4 より小さい値になっています。

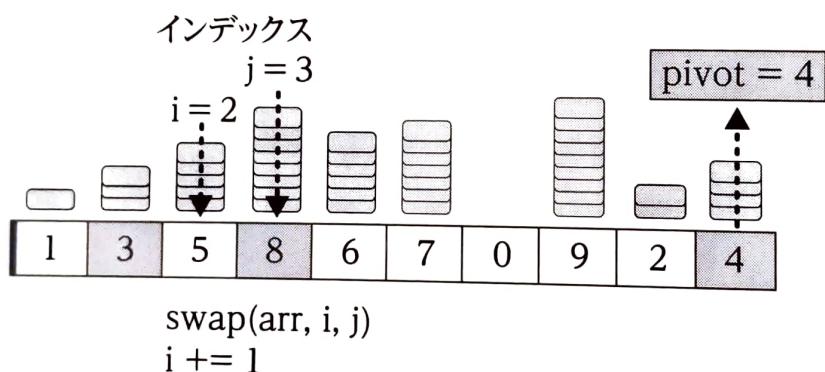


図 4.26 partition 関数の結果例 5

次は j の値が 6 のとき、 $\text{arr}[6]$ が整数値 0 なので、交換処理が発生します。交換処理を行ったあとの状態を図 4.27 に示します。この時点で、変数 i の値は 3 となります。

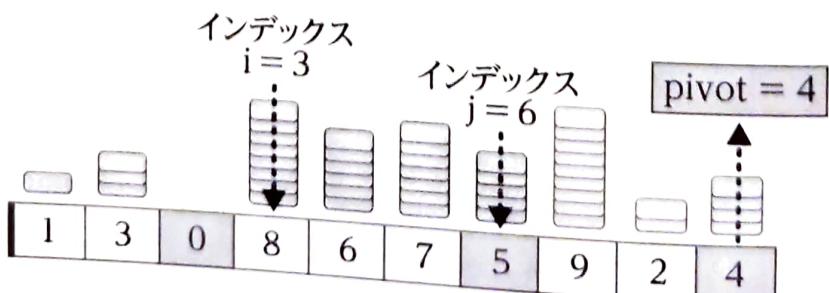


図 4.27 partition 関数の結果例 6

最後に j の値が 8 のとき、 $\text{arr}[8]$ に整数値の 2 が格納されているので、 $\text{arr}[3]$ と $\text{arr}[8]$ の交換処理を行い、変数 i の値を増加させます。その様子を図 4.27 に示します。変数 i の値は 4 になります。



図 4.28 partition 関数の結果例 7

ループカウンタ j の値が $r-1$ まで増加したので、for ループを抜けます。配列の中身ですが、 $\text{arr}[p:i-1]$ には $\text{arr}[r]$ より小さい値、 $\text{arr}[i:r-1]$ には $\text{arr}[r]$ より大きい値が格納されています。なお、 $\text{arr}[i]$ は $\text{arr}[r]$ より大きい値をもちます。そのため、ループを抜けた後に 15 行目の swap 関数の実行で $\text{arr}[i]$ と $\text{arr}[r]$ の要素を交換します。partition 関数の終了後、配列の状態は図 4.29 に示すとおりになります。

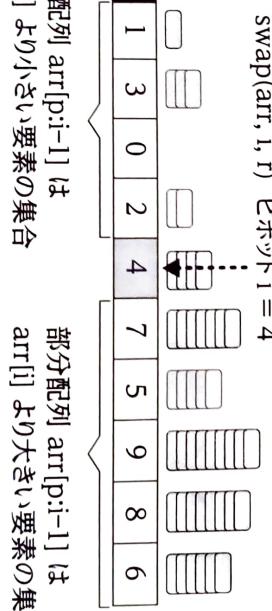


図 4.29 partition 関数の結果例 8

ピボットが $\text{arr}[i]$ に格納されており、ピボットの値を基準に 2 つの部分配列に分割できました。2 つの部分配列に対して partition を適応させ、同様の処理を繰り返します。部分配列の大きさが 2 であれば、片方の要素がピボットとなり、もう片方の要素と大小が比較され、自動的にソートされます。部分配列の大きさが 1 の場合、 p と r の値が同じになるので、21 行目の if 文の条件判定が False になり、統治ステップへ移行します。また、部分配列が空の場合も $p > r$ となるので、条件判定が False になり同様に統治ステップへ移行します。

■ クイックソートプログラム quick.py の実行

ソースコード 4.7 (quick.py) を実行した結果をログ 4.8 に示します。クイックソートによって配列が昇順に並べ替えられていることが確認できます。

ログ 4.8 quick.py プログラムの実行

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

参考に partition 関数終了後の配列の中身を図 4.30 に示します。なお、ピボットの要素はグレーベースで示しています。

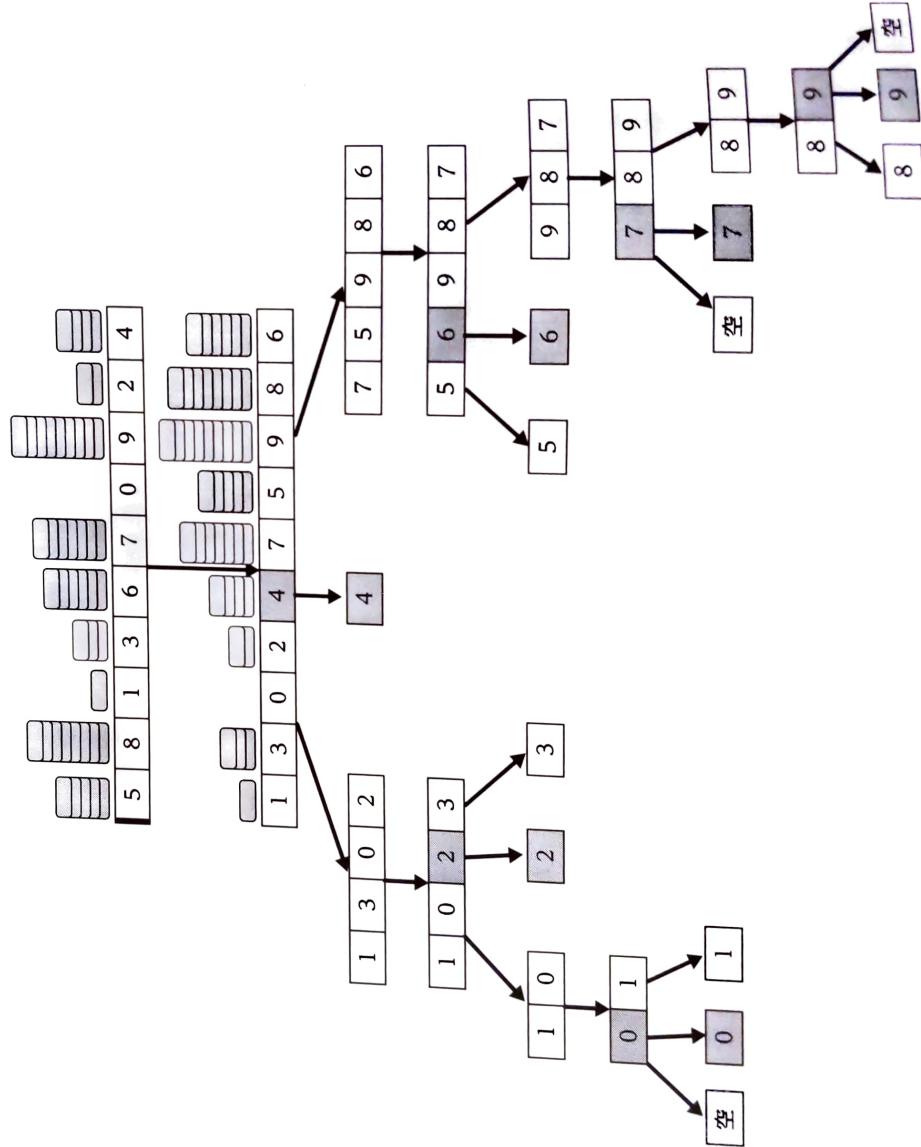


図 4.30 配列 [5, 8, 1, 3, 6, 7, 0, 9, 2, 4] をクイックソートで分割したときの詳細

■ 4.5.2 クイックソートの性質

クイックソートは実行速度の平均計算量が $O(n \log n)$ であるため、先の 3 例に比べ、最悪の場合は挿入ソートやバブルソート並に遅くなります。運用実績的に一番高速なパフォーマンスをもつといえます。ただし前述のとおり、すでに配列がソートされている場合は処理速度が最悪計算量である $O(n^2)$ になります。また、クイックソートは各部分配列の並べ替えを並列化することが可能です。

しかし、クイックソートは安定ソートの性質をもたないため、同じ値をもつ要素があった場合は、それらの要素間でソート前の順序が保証されません。

本書で解説したクイックソートは 1 つの例で、実際にはもう少し工夫をします。たとえば、配列がすでにソートされている場合は最悪のケースとなるので、そのような状況を避けるために配列の要素をランダムにシャッフルをします。具体的にはランダムにピボットを選んで、そのピボットを $\text{arr}[r]$ の要素と交換して、partition 関数を適応させるなどです。このように、クイックソートにはいくつかのバリエーションがあります。

4.6

ソートアルゴリズムの比較

本書で解説したソートアルゴリズムの最良計算時間と平均計算時間、最悪計算時間をまとめました。

表 4.1 データ構造に対する各操作の計算量

	最良計算時間	平均計算時間	最悪計算時間
挿入ソート	$O(n)$	$O(n^2)$	$O(n^2)$
バブルソート	$O(n)$	$O(n^2)$	$O(n^2)$
マージソート	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
クイックソート	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

■ 4.6.1 大きな配列をソートしたときの実測値の比較

それでは、ランダムな 10,000 個の数値を各ソートアルゴリズムで並べ替えたときの、処理時間

を比較します。処理時間として実測値を用いますが、使用するコンピュータの性能に大きく影響します。

比較を行うためのプログラムをソースコード 4.9 (compare.py) に示します。大きさが 10,000 の配列をランダムな数値で初期化し、各アルゴリズムでソートして処理に要した時間を表示するプログラムで、これまでのプログラムを import して使用します。

■ 各モジュールの説明

1行目～7行目で必要なモジュールをインポートします。1行目～4行目は、本章で作成した挿入ソートとバブルソート、マージソートとクイックソートの実装ソースコードのインポートです。5行目の random モジュールは、ランダムな数値を生成するために使用します。6行目の time モジュールは、時間を計測するためのものです。7行目の copy モジュールは配列を複製 (copy) するために必要です。

■ main 関数の説明

11行目～14行目でランダムな数値を要素としてもつ配列の生成します。配列の大きさは10,000とします。あまり大きくしすぎると処理しきれなくなるので、気をつけてください。13行目と14行目の for ループで、ランダムな数値を生成します。random モジュールの uniform 関数を用いると、引数で指定した値の範囲内から一様分布で数値を生成することができます。本書の例では、0から10,000,000を範囲としています。ランダムに数値を生成しているので、同じ数値が2回以上生成される可能性がありますが、本書で解説したソースコードは、この場合も適切に処理できます。

17行目～20行目では、挿入ソートを実行して、ソートに要した時間を計算しています。時間を計算するために、time モジュールの time 関数を用います。time 関数は、ある時点を基準とした経過時間を秒単位で返します。このシステム依存の時間の起点をエポック (epoch)、起点からの経過時間をエポック秒 (seconds from the epoch) と呼びます。Unix 系では、この起点となるエポックは1970年1月1日0時0分0秒(グリニッジ標準時)です。ソートにかかる時間を計測するためには、ソート前にエポック秒を確認し、ソート後に再度エポック秒を確認し、その差を計算します。17行目で変数 start を time.time() で初期化します。19行目では end 変数を宣言し、time.time() で初期化します。このようにすると end - start の計算結果が18行目のソート処理に要した実時間になります。

18行目では、挿入ソートを実行する insertion_sort 関数を実行します。中身は第4.2節で解説したとおりです。引数として、未ソートの配列と配列の大きさを与えます。すでにランダムな要素を含む配列 arr を生成していますが、ここでは配列 arr をコピーした新たな配列を引数として渡します。コピーを渡す理由は、中身が同じ配列を他のソートアルゴリズムでも再使用するからです。

20行目で、ソート処理に要した時間である `end - start` の計算結果を表示します。23行目以降も同様の処理も、`mergeSort`、`partition`、`quickSort`、`mergeSort`に対しても行います。

■ 各ソートアルゴリズムの比較プログラムの実行

ソースコード **4.9** (`compare.py`) を実行した結果をログ **4.10** に示します。処理が高速な順番に並べると、クイックソート、マージソート、挿入ソート、バブルソートとなります。特に統治分割法を用いるクイックソート ($O(n \log n)$) とマージソート ($O(n \log n)$) が挿入ソート ($O(n^2)$) とバブルソート ($O(n^2)$) に比べて、劇的に処理が速いことが確認できます。処理時間は、実行する機器、マシンによって結果が異なります。

ログ **4.10** compare.py プログラムの実行

```
01 $ python3 compare.py
02 挿入ソートの処理時間 = 3.413731098175049
03 バブルソートの処理時間 = 7.7877209186553955
04 マージソートの処理時間 = 0.03681206703186035
05 クイックソートの処理時間 = 0.02162790298461914
```

線形時間 ($O(n)$) と対数時間 ($O(\log n)$) の違いが極めて重要であることがわからると思します。