

第 10 回 整列アルゴリズム (クイックソート)

バブルソートよりも高速なクイックソートについて学ぶ。

【クイックソート】

クイックソートは、データの比較と交換回数が少なく、非常に効率良くソートできる整列アルゴリズムである。1960 年にアントニー・ホアが開発した。クイックソートは、とても高速な並べ替えアルゴリズムで、多くのプログラムで利用されている。非常に有名なアルゴリズムなので「クイックソート」で検索すると多くのページがヒットする。先週学習したバブルソートの平均計算量は、データ数 n に対しておよそ n^2 に比例する（このことを $O(n^2)$ と書く）のに対して、クイックソートの平均計算量は $O(n \log n)$ であり、クイックソートの方が圧倒的に速く、データ数が増えるにつれてバブルソートとクイックソートの計算速度の差が開く。今回はクイックソートを学習し、バブルソートとクイックソート以外のソートアルゴリズムについては「発展」で簡単に触れる。

クイックソートの基本的な処理手順は、次の通りである。

- (1) 配列の中から基準値（枢軸、ピボットと呼びます）を 1 つ決める。
- (2) 枢軸より小さい要素を枢軸より前に、枢軸より大きい要素の枢軸より後ろに移動する。
- (3) 枢軸より前方と後方のそれぞれについて、(1) の処理を再帰的に繰り返す。

この処理を Python で実現する。実現方法はいくつかあるが、ここでは、次の手順に従う。

- (1) 整列対象配列の中央にある要素を枢軸とする。
- (2) 配列を左から順に調べ、枢軸以上の要素を見つけ、その位置を変数 i に格納する。
- (3) 配列を右から順に調べ、枢軸以下の要素を見つけ、その位置を変数 j に格納する。
- (4) $i \leq j$ なら、 i 番目の要素と j 番目の要素を入れ替える。 i を 1 つ右へ進め、 j も 1 つ左を進めた上で (2) に戻る。
- (5) $i > j$ となったら、0 番目から $i-1$ 番目と、 i 番目から $n-1$ 番目の二つの領域に分け、それぞれに対して再帰的に (1) からの処理を行う。要素数が 1 以下の領域ができたら終了。

【★課題】

クイックソートを実現するプログラムを、

<https://paiza.io/projects/pGSKgJEOnM6FH-42805E0g>

にアクセスして、`????` の箇所を編集することで完成させてください。`????` 以外の場所是不変なこと。完成したプログラムを ToyoNet-ACE に提出してください。

```
def quick(left, right):
    pivot = (left + right) // 2 # 枢軸
    print(" 領域 ({0},{1}):{2} を ク イ ッ ク ソ ー ト 、 枢 軸 は {3}".format(left, right, list[left:right+1], list[pivot]))
    i = left # 配列を左から走査する変数
    j = right # 配列を右から走査する変数
    while i <= j:
        while list[i] < list[pivot]: # 枢軸以上の値が見つかるまで i を右へ進める
            i += 1
        while ??? > ??? : # 枢軸以下の値が見つかるまで j を左へ進める
            j -= 1
```

```

    if i <= j:
        # 要素の交換
        print("{0} の {1} と {2} を交換".format(list, list[i], list[j]))
        list[i], list[j] = list[j], list[i]
        i += 1
        j -= 1
# left から i-1 番目のグループを再帰的にクイックソートする
if left < i-1:
    quick(???, ???)
# i から right 番目のグループを再帰的にクイックソートする
if i < right:
    quick(???, ???)

list = list(map(int, input().split()))
print("{0} のクイックソート".format(list))
quick(0, len(list)-1)
print("ソート完了: {0}".format(list))

```

入力: 8 4 3 7 6 5 2 1

出力: [8, 4, 3, 7, 6, 5, 2, 1] のクイックソート

領域(0,7): [8, 4, 3, 7, 6, 5, 2, 1] をクイックソート、枢軸は 7

[8, 4, 3, 7, 6, 5, 2, 1] の 8 と 1 を交換

[1, 4, 3, 7, 6, 5, 2, 8] の 7 と 2 を交換

領域(0,3): [1, 4, 3, 2] をクイックソート、枢軸は 4

[1, 4, 3, 2, 6, 5, 7, 8] の 4 と 2 を交換

領域(0,1): [1, 2] をクイックソート、枢軸は 1

[1, 2, 3, 4, 6, 5, 7, 8] の 1 と 1 を交換

領域(2,3): [3, 4] をクイックソート、枢軸は 3

[1, 2, 3, 4, 6, 5, 7, 8] の 3 と 3 を交換

領域(4,7): [6, 5, 7, 8] をクイックソート、枢軸は 5

[1, 2, 3, 4, 6, 5, 7, 8] の 6 と 5 を交換

領域(5,7): [6, 7, 8] をクイックソート、枢軸は 7

[1, 2, 3, 4, 5, 6, 7, 8] の 7 と 7 を交換

領域(5,6): [6, 7] をクイックソート、枢軸は 6

[1, 2, 3, 4, 5, 6, 7, 8] の 6 と 6 を交換

ソート完了: [1, 2, 3, 4, 5, 6, 7, 8]

入力: 21 6 15 2 29 31 89 65 43 70 54 12 28 13

出力: [21, 6, 15, 2, 29, 31, 89, 65, 43, 70, 54, 12, 28, 13] のクイックソート
(中略)

ソート完了: [2, 6, 15, 12, 21, 29, 13, 31, 28, 43, 54, 65, 70, 89]

最初に、0 番目から 7 番目の全体に対してクイックソートを実施する。枢軸は、(0 番目 + 7 番目) // 2 で 3 番目、つまり 7 を選択する。変数 i により配列を左から調べ、枢軸より大きい 8 が 0 番目にあるのを見つける。また、変数 j により配列を右から調べ、枢軸より小さい 1 が 7 番目にあるのを見つける。これらを交換する。

1 4 3 [7] 6 5 2 8

続いて、変数 i は 1 番目から右へ、変数 j は 6 番目から左へと調べる。変数 i により 3 番目の 7 を発見、変数 j により 6 番目の 2 を発見、これらを交換する。

1 4 3 2 6 5 [7] 8

続いて、変数 i は 4 番目から右へ、変数 j は 5 番目から左へと調べる。変数 i は 6 番目に 7 を発見、変数 j は 5 番目に 5 を発見する。しかし、この時点で変数 i と変数 j は左右が入れ替わってしまったので、交換はしない。変数 i の左隣のところで領域を分割する。

1 4 3 2 6 5 | 7 8

分割された領域について、まずは左側である 0 番目から 5 番目に対して再帰的にクイックソートする。枢軸は 3 を選択する。

1 4 [3] 2 6 5 | 7 8

変数 i は 0 番目から右に調べ、枢軸より大きい 4 を 1 番目に見つけする。変数 j は、5 番目から左に調べ、枢軸より小さい 2 を 3 番目に見つける。これらを交換する。

1 2 [3] 4 6 5 | 7 8

変数 i は右隣の 2 番目へ、変数 j も左隣の 2 番目に来る。ここで、プログラムの上では、2 番目と 2 番目を交換するという処理が発生している。

1 2 [3] 4 6 5 | 7 8

さらに i は 3 番目へ、 j は 1 番目へと進む。この時点で i と j の左右が逆転したので、さらに領域を分割する。

1 2 3 | 4 6 5 | 7 8

以降同様に、再帰的な処理を繰り返す。

【無限ループ】

今回のプログラムには while のループ（繰り返し）があるため、プログラムが正しく書けていないと「無限ループ」が発生する可能性がある。正常なプログラムは、並べ替えが完了した時点でプログラムが終了するようにループができていますが、どこかで間違えているために、ループが終了しなくなってしまうことを「無限ループ」と言う。たとえば、

```
while 3 > 2:
    処理
```

というようなループがあるとする。この最初の while 文の条件式「 $3 > 2$ 」は常に True なので、中の「処理」を何度繰り返してもこのループを抜けることはなく、「処理」の中にプログラムを終了する条件が書かれていない限りは、このプログラムはいつまでも終了しない。これが「無限ループ」である。

Paiza では、一定時間内にプログラムが終了しないと強制的に終了し、「Timeout」と表示される。「Timeout」が出た場合には、無限ループである可能性が高い。

【発展：クイックソートよりも優れたソートアルゴリズム】

クイックソートは平均的には高速にソートできるが（平均計算量 $O(n \log n)$ ）、運が悪ければ（ピボットがうまく選ばれなければ）、あまり高速ではなくなる（最悪計算量 $O(n^2)$ ）。平均計算量、最悪計算量ともに $O(n \log n)$ と性能の良いソートアルゴリズムには、マージソート、ヒープソート、イントロソートなどがある。このように、近年はソートの性能を評価する指標として平均計算量だけでなく、最悪計算量、メモリ使用量、安定性など、様々な指標を総合的に評価して選ぶようになり、以前は「速いソートアルゴリズムといえばクイックソート」というほどクイックソートは有名なアルゴリズムであったが、クイックソート以外のソートアルゴリズムが採択されることが多くなっている。

Python では、リスト型のメソッド `sort()` と組み込み関数 `sorted()` が用意されている。そのアルゴリズムには、マージソートを元に Tim Peters が 2002 年に Python のために開発し

たティムソート(Timsort)が採用されている。Python のソースコードリポジトリの中にそのドキュメントがある¹。Timsort は、Java や Swift にも採用されている。

【発展：Python の実装】

Python はプログラミング言語の名称であり、Python で書かれたプログラムを動かすためのソフトウェア（実装）は一通りではない。その中のいくつかを紹介する。

(1) CPython²

Python の標準的な実装（リファレンス実装）であり、C 言語で実装されている。Python と言えばこの実装を意味し、他の実装を区別するときには CPython と表記される。バイトコードインタプリタであり、バイトコードが仮想マシンで実行されるが、バイトコードへの変換は暗黙に行われるために、ユーザーは通常コンパイラを意識しない。

(2) PyPy³

JIT（Just-in-Time）コンパイラ（実行時コンパイラ）によって、実行時にコードのまとまりを機械語に逐一コンパイルすることで高速に動作する。PyPy のサイトでは、グイド・ヴァン・ロッサムによる“If you want your code to run faster, you should probably just use PyPy.”という言葉が引用されている。

(3) Jython⁴

Java で実装され、Java クラスライブラリを利用できる。Python のプログラムを Java バイトコードにコンパイルできる。

(4) IronPython⁵

C#で実装され、.NET Framework および .NET Framework 互換の Mono 上で動作する。.NET Framework のクラスライブラリを利用できる。

(5) Cython⁶

Python を拡張したプログラミング言語であり、Python のソースファイルを C のコードに変換し、コンパイルすることで Python の拡張モジュールとして出力する。Python のコードの一部に静的な型を宣言することで、コンパイル後のコードの実行速度を高速化できることがある。

¹ <https://github.com/python/cpython/blob/master/Objects/listsort.txt>

² <https://www.python.org/>

³ <https://www.pypy.org/>

⁴ <https://www.jython.org/>

⁵ <https://ironpython.net/>

⁶ <https://cython.org/>