様々なソートアルゴリズム

クリスティアン　ハルジュノ *

2025 年 5 月 30 日

事前に この実験に使用するプログラム, データ, スクリプト等は GitHub の上にホストされています. プログラムを試したり検査したい場合は, 次の URL を参照してください.

`https://tianharjuno.com/sort`

1　背景

情報工学における重要な概念の一つにソートアルゴリズムがあります. ソートアルゴリズムの主な目的は, データの配列を最小から最大, または最大から最小の順に並べることです. 一般に, ソートはそれほど複雑な問題ではないと考えられます. しかし, データ量が大幅に増加すると（ビッグデータ企業等）, ソートは極めて困難になる場合があります. ソートにかかる時間や使用メモリ量は重要な課題であり, データの種類や量に応じて適切なアルゴリズムを選択する必要があります. また, あるアルゴリズムは効率的ですが, すべてのユースケースに実装する価値がない場合があります. さらに, 多くの会社に応募するとき（Amazon, Google, Netflix, 等）, これらの大企業のほとんどは, 実技試験でさまざまな複雑なソートアルゴリズムを実装することを要求しています.

2　一般的のソートアルゴリズム

この章では, いくつかの有名なソートアルゴリズムについて説明します. これらのアルゴリズムの実装は比較的簡単であり, 効率も高いため, データの種類や量に応じて適切に選択することができます.

2.1　バブルソート

バブルソートは最もよく知られているソートアルゴリズムの一つです. 実装が非常に簡単であり, 少量のデータに対してはメモリ使用量が少なく, 処理も比較的高速です. この方法にはデータを最初から最後まで調べながら, 現在の数値と次の数値を比較することで機能します. 現在の数値が次の数値よりも大きい場合は, 位置を入れ替えます. このプロセスは, データが完全にソートされるまで何度も繰り返されます.

*釧路工業高等専門学校情報工学科 5 年情報 21 番

バブルソートのやり方

1. リストの最初の位置から始める.
2. 現在の数値と次の数値を比較する.
3. 現在の数値が次の数値より大きければ, 入れ替える.
4. 次の位置に進み, リストの末尾までステップ 2〜3 を繰り返す.

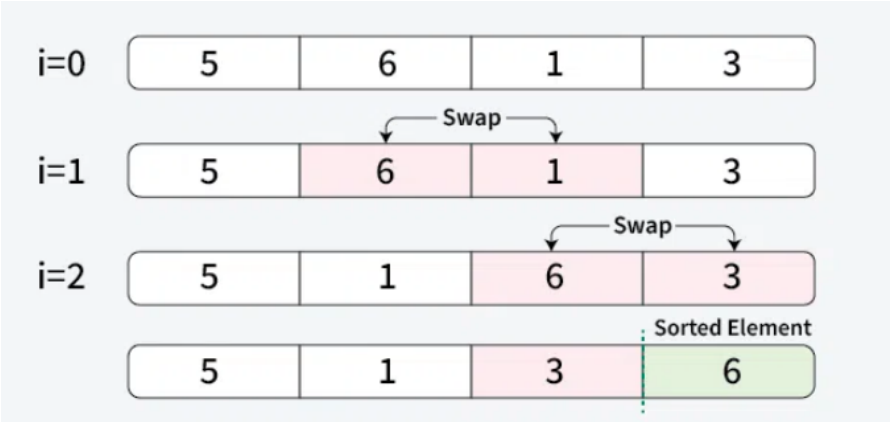

図 1: 先頭から末尾まで数値を比較して, 交換する.

5. 1 回の通過が終わったら, 最初の位置に戻って再び通過を行う.

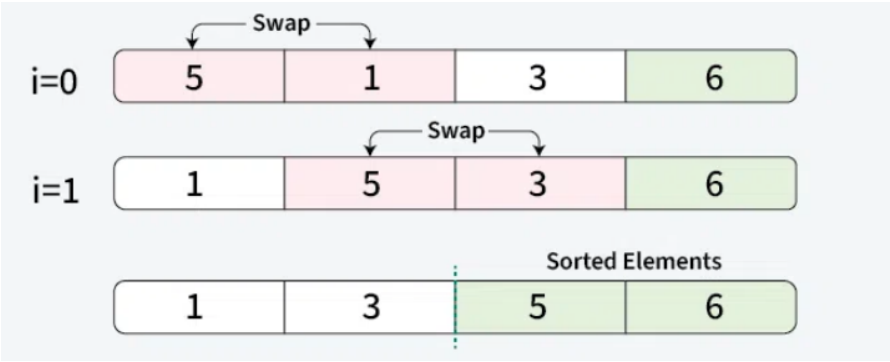

図 2: 通過後最初の位置に戻って, 再び通過

6. 通過中に 1 回も入れ替えがなければ, ソートは完了.
7. 入れ替えがあった場合, ステップ 1〜6 を繰り返す.

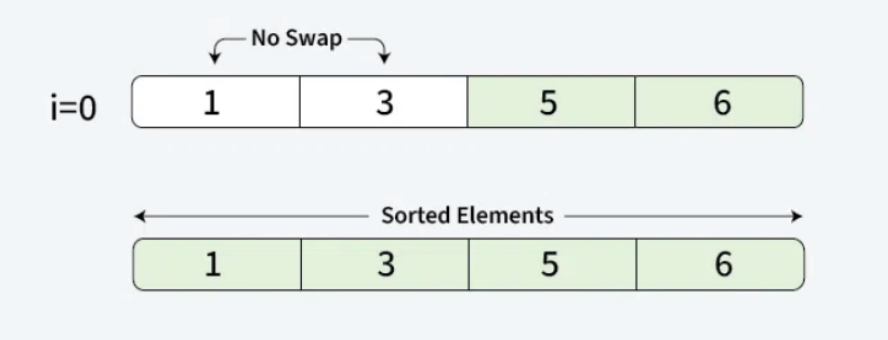

図 3: 通過後に入れ替えされた要素がないと終了

バブルソートの計算量はデータの状況によって定義します.

最良 最適というのはそのデータを事前にソートされている状態です.

$$O(n) \tag{1}$$

平均 入力が均一に分布していると仮定して, すべての可能な入力順列に対して実行される操作の予想数です.

$$O(n^2) \tag{2}$$

最悪 最悪の状態はソート対応するデータが逆順としてソートする状態です.

$$O(n^2) \tag{3}$$

2.2 挿入ソート

挿入ソートは, 要素を適切な位置に直接挿入することで並べ替えを行うアルゴリズムです. 現在処理している要素を, それより前の要素と右から左に 1 つずつ比較し, 比較対象が現在の要素より大きければ, 右にずらします. 適切な位置が見つかったところで, 現在の要素を挿入します.

挿入ソートのやり方

1. 最初の要素をソート済み部分とみなす.
2. 次の要素を取り出して, ソート済み部分の各要素と比較する.

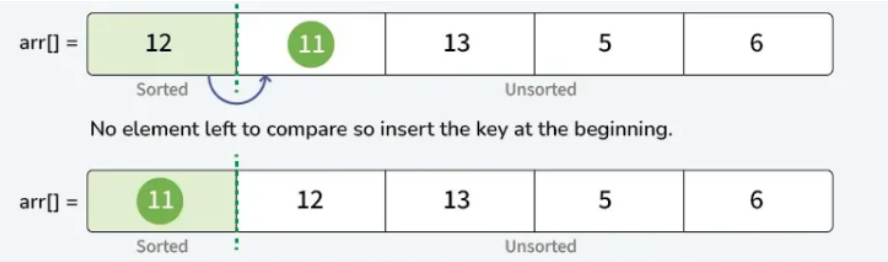

図 4: 第二位置からスタート. ソート済み要素より大きい数値を左にずらす.

3. ソート済み部分の中で, 自分より大きい要素を右にずらす. 適切な位置に挿入する.

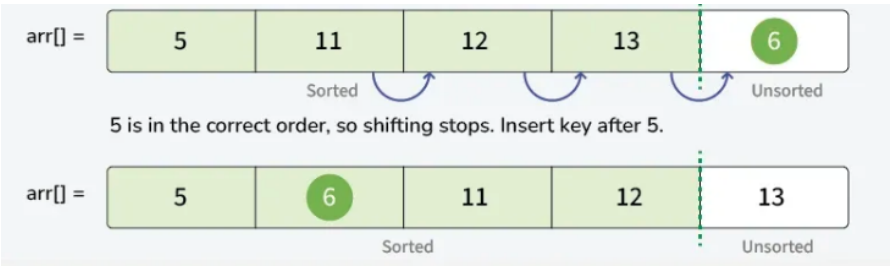

図 5: 繰り返して, ソート済み要素より大きい数値を左にずらす

4. 適切位置がないとそのままで次の位置に進む

5. リストの末尾までステップ 2〜4 を繰り返す.

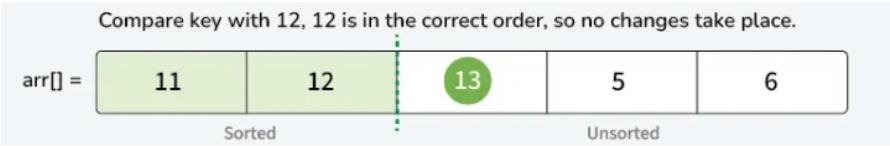

図 6: 適切位置がないとそのままで進む

6. ソートされていない要素がないと終了

挿入ソートの計算量はデータの状況によって定義します.

最良 最適というのはそのデータを事前にソートされている状態です.

$$O(n) \tag{4}$$

平均 入力が均一に分布していると仮定して, すべての可能な入力順列に対して実行される操作の予想数です.

$$O(n^2) \tag{5}$$

最悪 最悪の状態はソート対応するデータが逆順としてソートする状態です.

$$O(n^2) \tag{6}$$

2.3 シェーカーソート

シェーカーソートまたはカクテルソートはバブルソートと似てるですが, シェーカーソートでは左から右に移動だけでなく逆向きにもう通過する.

シェーカーソートのやり方

1. 配列の先頭から走査を開始する.

2. 現在の要素と次の要素を比較し, 現在の要素の方が大きければ交換する.

3. 配列の末尾まで進み, 走査中に交換が一度も行われなければソートを終了する.

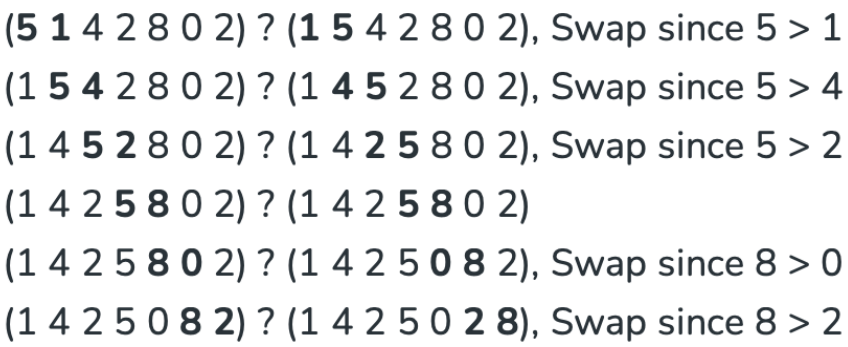

図 7: 先頭からスタートし, 現在の要素と次の要素より大きければ, 交換. 末尾まで進む

4. 次に逆方向（末尾から先頭）に走査する.

5. 現在の要素と前の要素を比較し, 現在の要素の方が小さければ交換する.

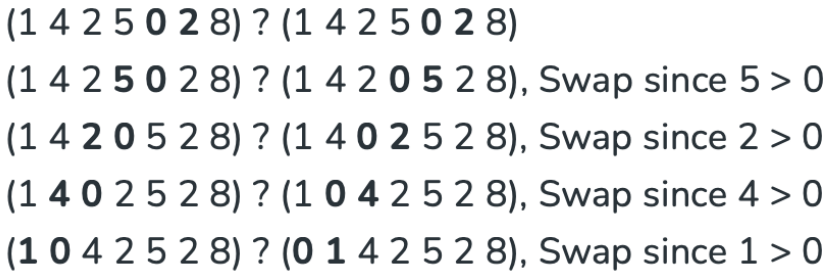

図 8: 末尾からスタート先頭方向に走査. 現在の要素は前の要素より小さければ, 交換. 先頭まで進む

6. 先頭まで進み, 走査中に交換が一度も行われなければソートを終了する.

7. 上記の操作を, 交換が行われる限り繰り返す. 各往復ごとに走査範囲を狭めることができる.

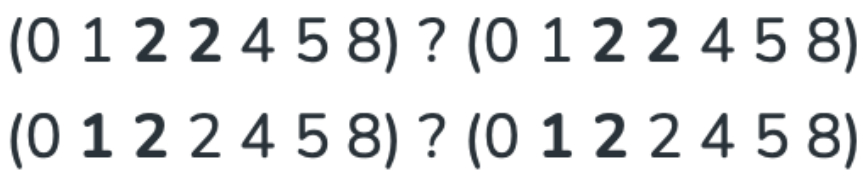

図 9: 末尾方向や先頭方向の通過に交換がなければ, 終了.

シェーカーソートの計算量はデータの状況によって定義します.

最良 最適というのはそのデータを事前にソートされている状態です.

$$O(n) \tag{7}$$

平均 入力が均一に分布していると仮定して, すべての可能な入力順列に対して実行される操作の予想数です.

$$O(n^2) \tag{8}$$

最悪 最悪の状態はソート対応するデータが逆順としてソートする状態です.

$$O(n^2) \tag{9}$$

2.4 クイックソート

クイックソートは分割統治ソートと言われています。このソート方法ではある配列から枢軸を選び、全要素をその枢軸に比較する。枢軸より小さいとその要素を枢軸の左がわにづらす。枢軸より大きいと、その要素を枢軸の右側にづらす。枢軸を分岐点とし、配列を左配列と右配列に分割する。この処理を各文割された配列に対応し、繰り返して行う。最小配列に着くと、各配列をまた分岐点枢軸の左と右側に繋ぎます。

クイックソートのやり方

1. 初めに、配列から枢軸（ピボット）を 1 つ選びます。選び方には、先頭、中央、末尾などさまざまな方法があります。
2. 配列の各要素を枢軸と比較し、小さい要素は枢軸の左側、大きい要素は右側に移動させます（パーティショニング）。
3. 枢軸を基準にして、配列を左側部分（枢軸より小さい要素）と右側部分（枢軸より大きい要素）に分割します。

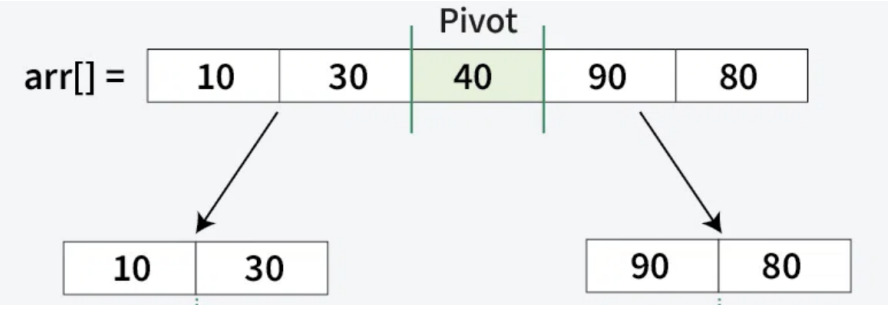

図 10: 左側分と右側分に分割日ます

4. 左右それぞれの部分配列に対して、同様にステップ 1〜3 を再帰的に繰り返します。

5. 各部分配列の要素数が 1 以下になると、すべての部分配列が整列され、結果的に全体がソートされます。

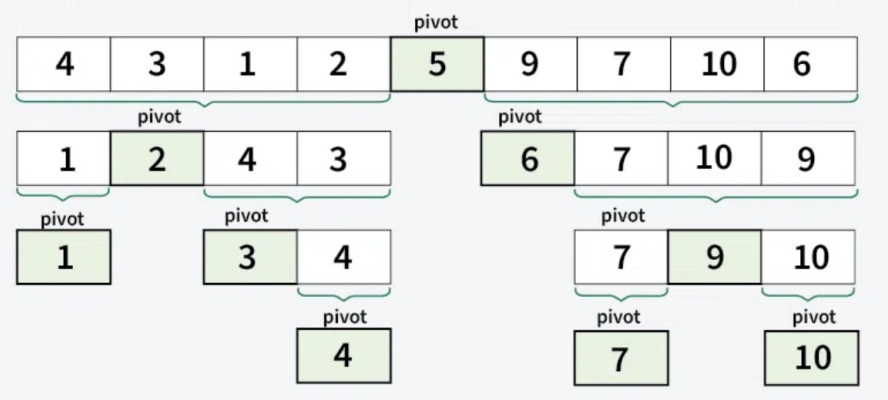

図 11: クイックソートの概念を提示します。

クイックソートの計算量はデータの状況によって定義します.

最良 最適というのはそのデータを事前にソートされている状態です. ソートを進むとともに、通過しないといけない配列の長さは短くなる。

$$O(n \log n) \tag{10}$$

平均 入力が均一に分布していると仮定して, すべての可能な入力順列に対して実行される操作の予想数です. 最良状態か最悪状態でも配列を分割しなければならないので、計算量は等しい。

$$O(n \log n) \tag{11}$$

最悪 この状態は、最大の値を枢軸としてソートを行った際に発生する可能性があります。

$$O(n^2) \tag{12}$$

2.5 バケットソート

バケットソートは他方法より速度が最高と言えます。しかし、ソートの高速にはメモリ使用量の増加が伴う。バケットソートでは、データを複数のバケットに分割します。各バケットは、バブルソートや挿入ソートなどの手法を用いてソートされます。すべての要素を指定されたバケットに挿入した後、すべてのバケットを連結してソート済みの配列を作成します。この手法は、大規模なデータに対して単純なソートアルゴリズムを実行する必要がないため、高速です。前述のように、バブルソートなどの一般的なソート手法は、大規模なデータベースではパフォーマンスが低下します。しかし、少量のデータであれば問題ありません。データを独自のバケットに分割することで、1 回の操作で処理しなければならないデータ量が大幅に削減され、アルゴリズムの負荷が軽減され

ます。
バケットへの分割方法は、データの種類によって異なります。必要なバケットの数を減らすことで、必要なメモリ量を削減できます。50,000 個の異なる値に対して 50,000 個のバケットを作成する代わりに、値がバケットに挿入される範囲を作成できます。例えば、1 から 50 までの値は、50 個の異なる値ではなく、1 つのバケットに混合されます。

3　実験結果