ソートプログラム

釧路工業高等専門学校情報工学科5年・情報工学実験II

氏名：クリスティアン・ハルジュノ

出席番号：21

学籍番号：234071

提出日：2025年6月17日

1　初めに

データに最適なソートアルゴリズムを選択するのは容易ではありません. Big-O 記法を用いて理論的にアルゴリズムを比較することは可能ですが, 現実のソートアルゴリズムはそれ以上の要素を含んでいます.

理論的に効率的なアルゴリズムの中には, 実装前に徹底的に調査する必要がある弱点を持つものもあります. 一部のアルゴリズムは, データがアルゴリズムを最適に実行できる特定の状態にある場合にのみ効率的です.

現代のコンピュータは, 大容量メモリ, 高速 CPU, 効率的なストレージなど, 十分なハードウェアスペックを備えていますが, 一部のアルゴリズムは高速実行のためにリソースを犠牲にしています. データサイズが小さい場合は, これは問題にならないかもしれません. しかし, ビッグデータ企業のようにデータを大規模化すると, リソースは適切に対処する必要がある非常に重要な問題になります. そのため, アルゴリズムを理論的に比較するだけでなく, 対象データセットに対してこれらのアルゴリズムをテストし, 結果を分析することが重要です.

本レポートでは, 実装が最もシンプルなものからより複雑なものまで, 最も普及している 6 つのソートアルゴリズムを検証し, 7 つの固定データセットと比較します. 各データセットには, 1 から 50000 までの整数が 10 万行含まれています. 各データセットは, 完全にランダム, 反転, ソート済みなど, さまざまな段階にあります. これらの比較により, 選択した 6 つのソートアルゴリズムの長所, 短所, および動作を検証できます.

2　背景

このセクションでは, この実験で使用した 6 つのソートアルゴリズムを紹介し, それぞれがどのように機能するかを説明します.

2.1　ソートアルゴリズムを紹介

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.

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.

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.

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り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.

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.

カウントソート バケットソートと似たように, このソート方法でもバケットのような配列を作成する. まず, データ配列内の最大値を探索し, その値をもとにカウント配列（頻度を格納するための配列）を作成する. 次に, 元のデータ配列を 1 つずつ走査し, それぞれの値に対応するインデックスのカウント配列の値をインクリメントすることで, 各値の出現回数を記録する. その後, カウント配列を累積和に変換し, それをもとに元のデータを安定的に新しい配

列へと並べ替えていく. このソートは整数値に限定されるが, 非高速でソートできます.

2.2 理論的な比較

表1, セクション2.1で前述した各ソート手法の基本的な特性または動作を示しています. 各アルゴリズムの特性は, 最良の計算複雑度, 平均的な計算量, 最悪の計算量に分類されます. 最良の計算量は, 特定の入力状態においてデータをソートするために必要な最小の演算量を表します. 平均的な計算量は, 考えられるすべての入力を考慮し, データをソートするために必要な予想される演算量を表します. 最悪の計算量は, 特定の入力状態においてデータ処理に必要な最大の時間を表します.

表 1: 各ソートアルゴリズムの計算量と特徴の比較

アルゴリズム	最良計算量	平均計算量	最悪計算量	空間計算量	インプレース
バブルソート	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	はい
シェーカーソート	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	はい
挿入ソート	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	はい
バケットソート	$O(n+k)$	$O(n+k)$	$O(n^2)^{\dagger}$	$O(n+k)$	いいえ
クイックソート	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)^{*}$	はい
カウントソート	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	いいえ

計算量を指定するために, Big O 表記法を使用します. Big O 表記法は, 処理する必要があるデータの量に基づいてソート アルゴリズムのパフォーマンスを評価する最も基本的な方法の 1 つです. 表 1 に示されているように, バブルソート, シェイカーソート, 挿入ソートの平均計算量は $O(n^2)$ です. Big O 表記法では, n はデータ自体の長さに対応します. データのソートに必要な操作数に対して n をプロットすると, 計算量が $O(n^2)$ の図 1 に示すように, ソートするデータの数がわずかに変化するだけでも操作数が大幅に増加し, 大量のデータのソート効率が低下する可能性があります. しかし, 計算量が $O(n\log n)$ のクイックソートを 1 と比較すると, データ量を大幅に増やした場合でも, 特定のデータをソートするために必要な操作の量は, 複雑度が $O(n^2)$ のアルゴリズムに比べてはるかに少なくなります.

上記のアルゴリズムの最良計算量について、バブルソート、シェイカーソート、挿入ソートはいずれも、与えられた入力データが既にソート済みであることを前提としています。この場合、アルゴリズムはデータを 1 回だけ処理すればよいため、計算量は $O(n^2)$ から $O(n)$ に減少します。バケットソート最良計算量では、すべての要素が各バケットに均等に分散されていると想定されます。クイックソートの最良計算量では、ピボットの選択によって値が左右に均等に分散されると想定されます。最後に、カウントソートの最良計算量では、入力データの整数の範

囲が狭いと想定されます。

最後に, 外部配列を用いてデータの内容をソートするバケットソートとカウントソートの場合, 平均的な計算量は $O(n+k)$ で測定されます. ここで, k は外部配列の処理に必要な演算量を表します. k を調整することで, 特定のデータのソートに必要な演算量を大幅に削減可能性がある.

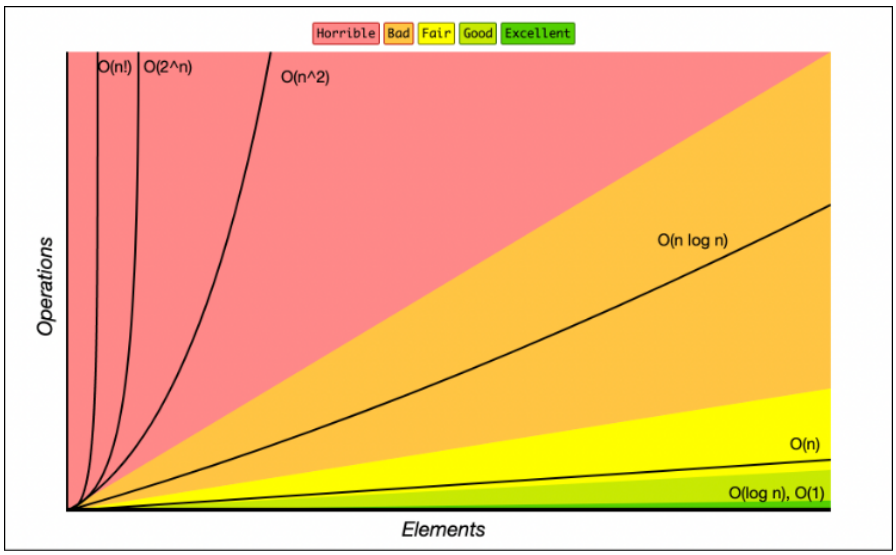

図 1: 様々な計算量を比較

表 1 に基づくと, 空間計算量とインプレースは, ソートが元のデータ配列自体の外で行われることを指します. バブルソート, シェイカーソート, 挿入ソートの空間計算量が $O(1)$ ので, 与えられたデータをソートするために必要な空間の量は最初から変わりません. つまり, これらのアルゴリズムは非常に空間効率が高く, 元のデータとは別に新しいメモリを割り当てる必要がありません. バケットソートとカウントソートのアルゴリズムの空間計算量は $O(n+k)$ と $O(k)$ であり, この場合, k はデータをソートするために作成される新しい空間の量を指します. これらのアルゴリズムは新しい空間を割り当てる必要があり, アルゴリズムの設定方法によっては大量のメモリを消費する可能性があります. クイックソートは特に注目すべき点です. クイックソートの空間計算量は $O(\log n)$ ですが, すべての操作はデータ配列内で行われ, 新しいメモリを割り当てる必要はありません. そのため, クイックソートはインプレースソートアルゴリズムと呼ばれます. クイックソートはソートが進むにつれて, 処理が必要なデータの長さが縮小し続けるためです.

3　実験セットアップ

3.1　ハードウェアと環境

ハードウェアとしてはAppleシリコンのプロセッサに基づいて実験を行います。

機種名 Apple Macbook Pro (M3, 2023年モデル)

プロセッサ Apple M3チップ（8コア：高性能コア4＋高効率コア4）

GPU 10コア統合型GPU

メモリ 16GBユニファイドメモリ

ストレージ 1TB SSD

OS macOS Sequoia 15.5

アーキテクチャ ARM64（Appleシリコン）

また、作成した実験プログラムはC言語で書き、GCCコンパイラでコンパイルされます。

```
Apple clang version 17.0.0 (clang-1700.0.13.5)
Target: arm64-apple-darwin24.5.0
Thread model: posix
```

コンパイルを効率化するため、GCCコンパイラをMakeと同時に利用します。

```
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.
This program built for i386-apple-darwin11.3.0
```

3.2　実験に使用するデータセット

各ソート アルゴリズムの制限と動作を適切にテストするために、1から50000までの範囲の100000個の整数を含む7つのデータセットを用意しました。各データセットの構造は、適用されたソート アルゴリズムの動作を引き出すためにさまざまな方法で配置されています。表2を見ると、データセット1から3にはすべて10万個の整数が含まれており、それらはランダムに並べられていることがわかります。データセット4には、適用されたアルゴリズムの最良のシナリオをシミュ

表 2: 各データファイルの内容

ファイル名	データ数	データの種類
data1.dat	100,000	ランダムデータ
data2.dat	100,000	ランダムデータ
data3.dat	100,000	ランダムデータ
data4.dat	100,000	昇順データ
data5.dat	100,000	降順データ
data6.dat	100,000	バイトニックデータ
data7.dat	100,000	ジグザグデータ

レートするために既にソートされたデータが含まれています。データセット 5 は、ソートアルゴリズムの最悪のシナリオをシミュレートするために、逆順にソートされています。

データセット 6 はバイトニックシーケンスで配置されています。数式 1 により、バイトニックシーケンスとは、データセットの先頭から緩やかに上昇し、中央でピークに達し、その後、データセットの末尾に近づくにつれて緩やかに下降するデータと説明できます。

$$a_1 < a_2 < \cdots < a_k > a_{k+1} > \cdots > a_n \quad (1 \leq k < n) \tag{1}$$

データせっと 6 の先頭付近と中間付近と末尾付近からサンプル要素をランダム的に取って、表 3 のようになります。

表 3: バイトニックデータ（data6.dat）のサンプル値

位置	サンプル値（例）
先頭付近	17131, 41279, 23264, 44242, 2505, 6637, 5374,
中間付近（最大付近）	43248, 43247, 43247, 43246, 43244
末尾付近	12, 12, 10, 10, 9, 9, 9, 9, 8, 8, 7, 5, 2

データセット 7 はジグザグ配列になっています。数式 4 ジグザグ配列とは、データ全体を通して小さな数値と大きな数値が交互に現れるデータのことです。この場合、データセットはランダム化され、「小さい」部分と「大きい」部分が交互に現れます。データセットは、これらの大きな部分と小さな部分が、長さを変えながら交互に現れます。

$$a_1 < a_2 > a_3 < a_4 > a_5 < \cdots \tag{2}$$

データせっと 7 の先頭付近と中間付近と末尾付近からサンプル要素をランダム的に取って、表 3 のようになります。

表 4: ザクザクデータ（data7.dat）のサンプル値

位置	サンプル値（例）
先頭付近	2, 2, 2, 4, 7, 7, 7, 8, 13, 15
中間付近	20778, 42992, 29869, 31747, 22945, 40054, 38945, 47166
末尾付近	11990, 7746, 38998, 35057, 41719, 22808, 23284, 2525

3.3 測定について

ソートアルゴリズムの実行時間を正確に測定するために、C 言語の *time.c* ライブラリを利用します。*time.c* ライブラリには、プログラム実行後に経過した CPU ティック数を測定する *clock*() 関数が用意されています。図 2 により、*clock*() 関数を使用することで、ソート関数の実行前から完了後までのティック数を測定します。得られたティック数を、CPU が 1 ティックを実行するのにかかる時間で割ります。この値を計算すると、ソート関数の合計実行時間が算出されます。

図 2: プログラムの測定分の例

```
start_time = clock();
shakerSort(fileLength, clonedArray);
end_time = clock();
elapsed_time = (double)(end_time - start_time)
               /CLOCKS_PER_SEC;
```

3.4　各アルゴリズムの説明

3.4.1　バブルソート

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.

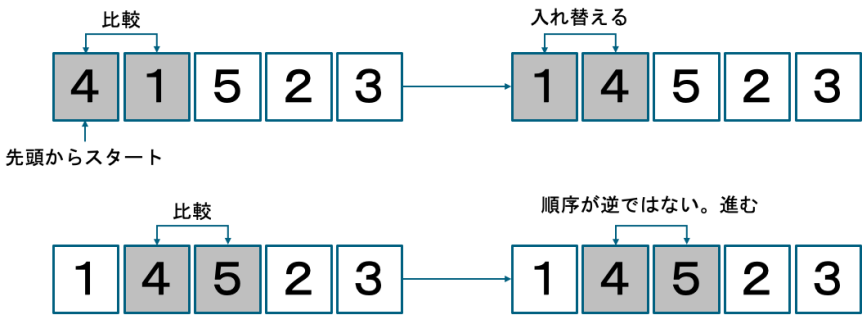

図 3: 先頭から末尾まで数値を比較して, 交換する.

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.

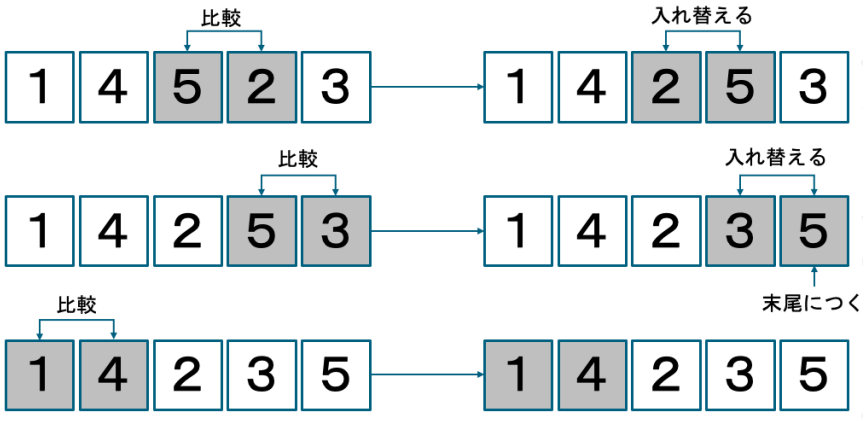

図 4: 通過後最初の位置に戻って, 再び通過

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.

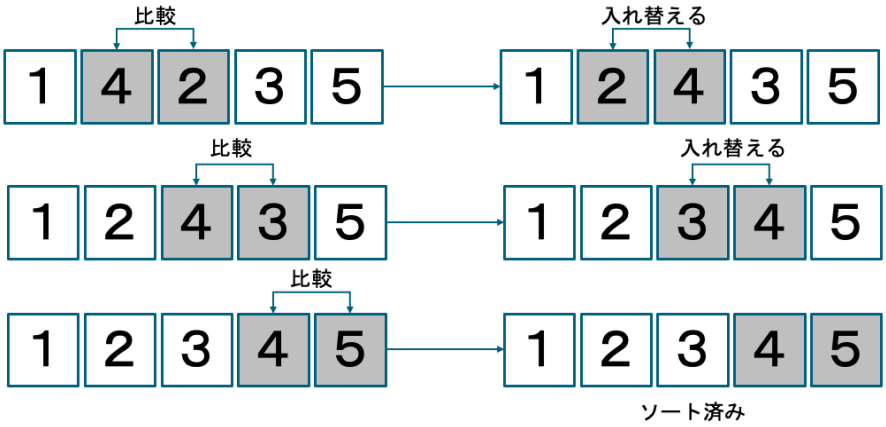

図 5: 通過後に入れ替えされた要素がないと終了

3.4.2 シェーカーソート

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.
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.
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.

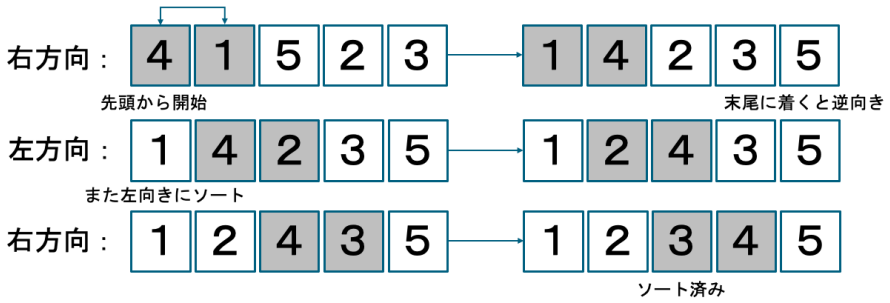

図 6: シェーカーソートの進み方

3.4.3　挿入ソート

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.
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.
4. 適切位置がないとそのままで次の位置に進む

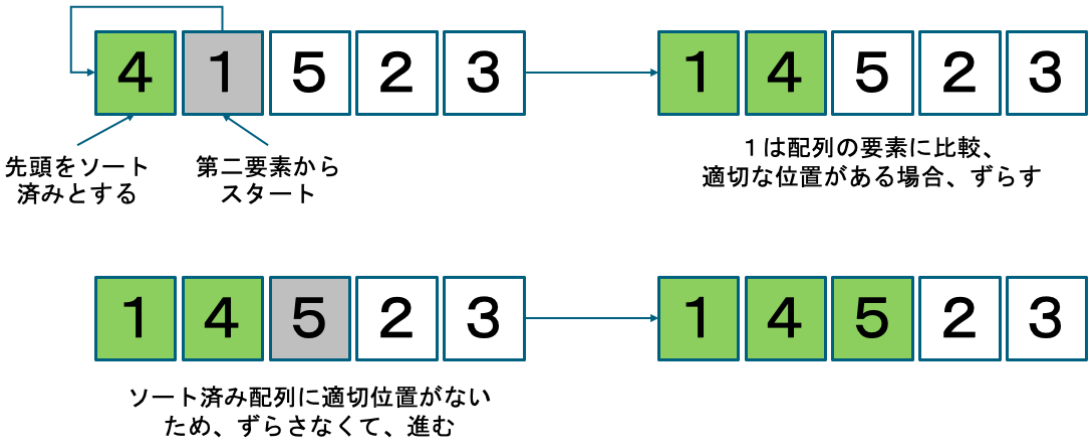

図 7: 第二位置からスタート. ソート済み要素より大きい数値を左にずらす.

5. リストの末尾までステップ 2〜4 を繰り返す.
6. ソートされていない要素がないと終了

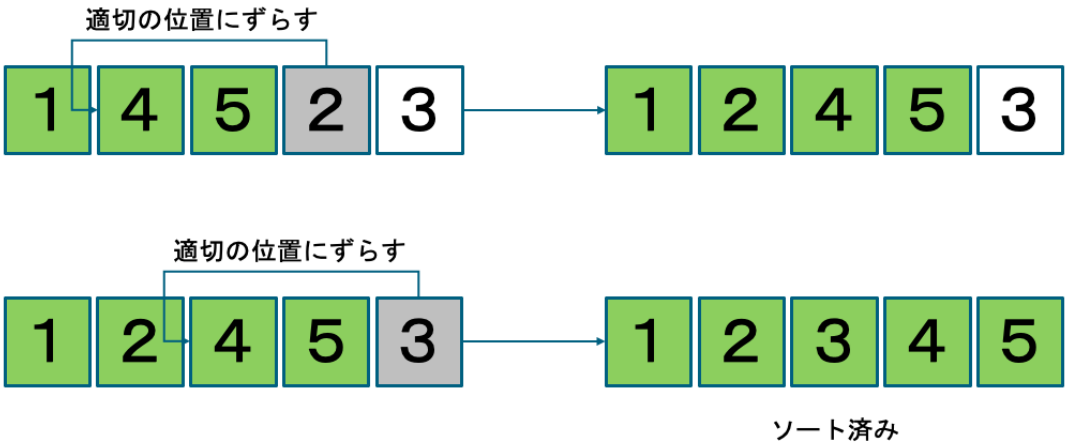

図 8: ソート対象値がないとソート終了

3.4.4 バケットソート

作成するバケット数の選択は、再現性を確保し、パフォーマンス評価の一貫性を維持するために重要であるため、入力特性に基づいて動的に決定するのではなく、すべてのテスト実行でバケット数を固定しました。本レポートの結果の大部分では、バケット数を 100 としています。また、バケット数がメモリ使用量とソート速度に与える影響についても考察します。[1]

図 9: ソート対象データ

1. 最初に、ソート対象の整数の範囲（例：0〜2.5）を確認し、その範囲に応じてバケットの数を決める（例：3 の範囲ごとにバケットを作成するなど）。

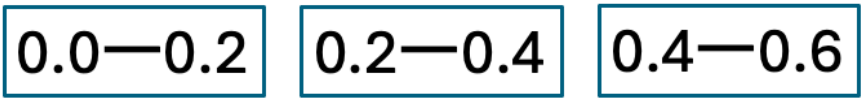

図 10: 作成したバケット

2. 各要素をその値に基づいて対応するバケットに振り分ける。

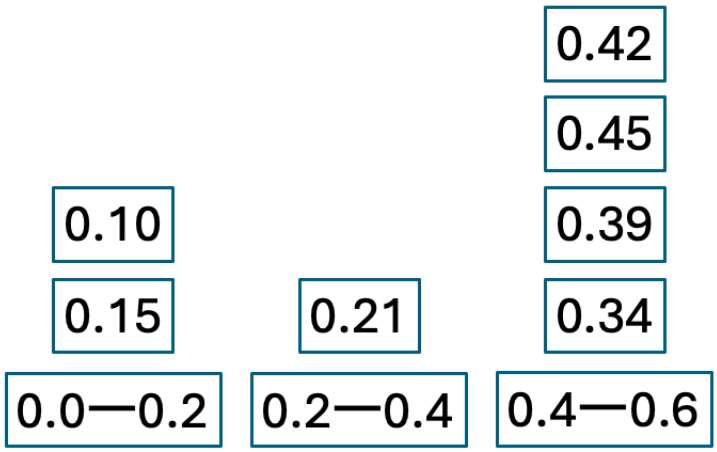

図 11: 適当に作成されたバケットに要素を入れる。

3. 各バケット内の要素をソートする（挿入ソートやバブルソートなど、単純なソートで OK）。

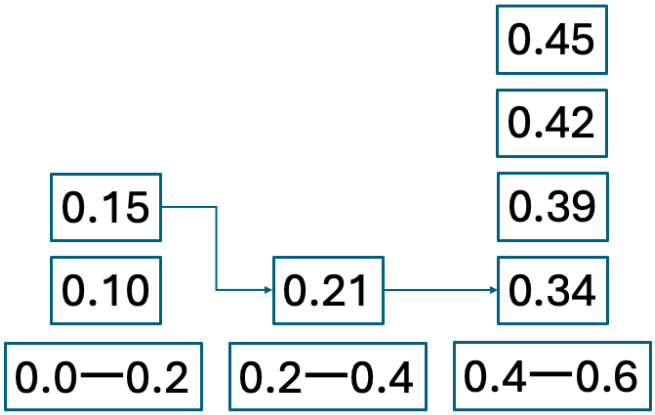

図 12: 各バケットの内容をソートし、連続

4. すべてのバケットを順番に結合して、最終的なソート済み配列を作る。

図 13: ソートずみの配列

3.4.5 クイックソート

クイックソートを利用する場合、ピボットポイントの選択はアルゴリズムのパフォーマンスに重要です。最適なピボットポイントは、小さい（左）配列と大きい（右）配列を均等に分割できるものです。常に最大のピボット値または最小のピボット値を選択すると、すべてのデータ値をピボットポイントの左側または右側に移動する必要があるため、ソートアルゴリズムの効率が大幅に低下します。その結果、計算量は $O(nlogn)$ から $O(n^2)$ に変化します [2]。

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。

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。

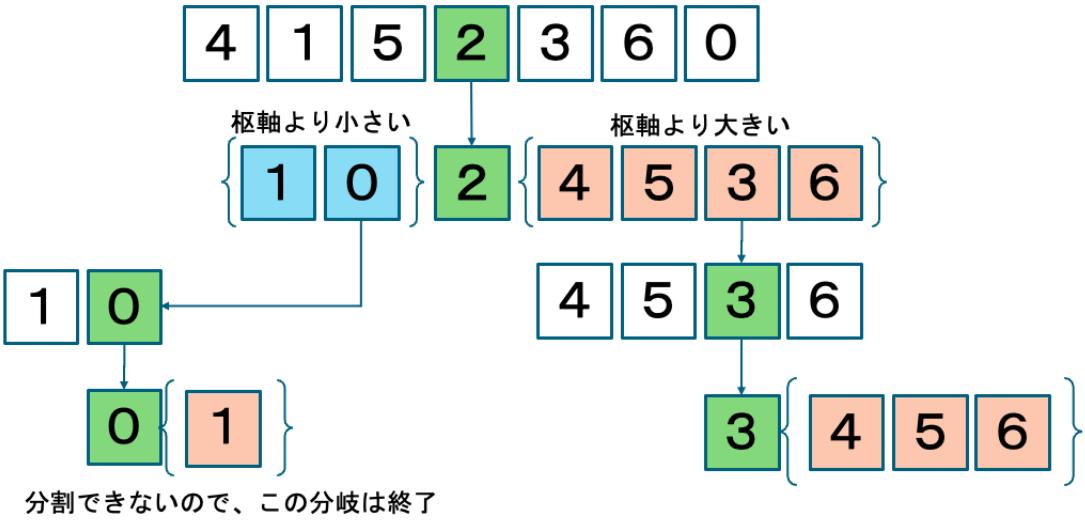

図 14: 枢軸を選び、左側分と右側分に分割します。左と右配列を再帰的に処分

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。

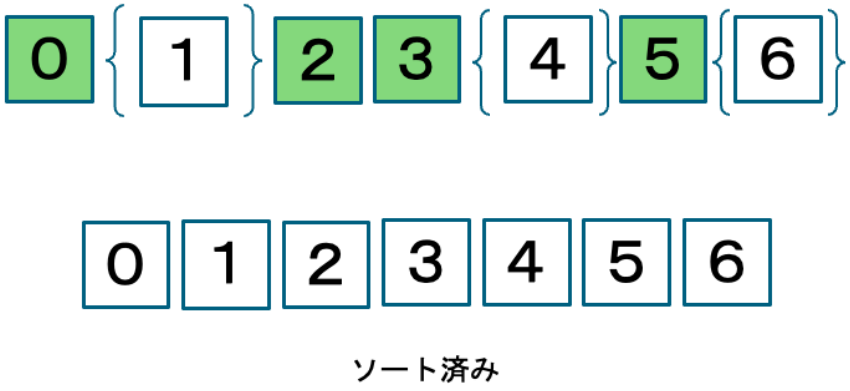

図 15: 部分配列を整列され、結果的に全体がソートされます

3.4.6 カウントソート

4　実験結果

4.1　データセットごとに実行

各ソートアルゴリズムのパフォーマンスを各テストケース（データセット）に対して比較するために、すべてのアルゴリズムを単一のデータセットに対して実行するソートプログラムを実行します。
ハードウェアメーカーによって実装された低レベルの最適化とハードウェア効果により、CPU はプログラムの次の実行部分を予測または推測し、繰り返し実行されるシステムコールを高速化するために最適化することができます [4]。CPU はまた、最初の実行時にこのデータセットを RAM にロードします。アルゴリズムが繰り返し実行されると、ロードされたデータセットは CPU ダイ自体のより高速なキャッシュにゆっくりと移動され、メモリアクセスのレイテンシが大幅に低減されます [3]。表 5 により、これらの最適化のせいで、単一のソートアルゴリズムを繰り返し実行すると、時間の経過とともに結果が徐々に高速化され、結果の不正確さが生じます。この問題を軽減するために、各アルゴリズムを 10 回実行し、結果を平均します。

表 5: ソートアルゴリズムを１０回繰り返して実行する場合

File Name	Sort Method	Data Length	Time(sec)	Memory (KB)
data1.dat	Counting Sort	100000	0.000803	400.00
data1.dat	Counting Sort	100000	0.000804	400.00
data1.dat	Counting Sort	100000	0.000805	400.00
data1.dat	Counting Sort	100000	0.000808	400.00
data1.dat	Counting Sort	100000	0.000790	400.00
data1.dat	Counting Sort	100000	0.000762	400.00
data1.dat	Counting Sort	100000	0.000741	400.00
data1.dat	Counting Sort	100000	0.000736	400.00
data1.dat	Counting Sort	100000	0.000736	400.00
data1.dat	Counting Sort	100000	0.000763	400.00

4.1.1 データセット１に対する実行結果

表 6: 各ソートアルゴリズムの `data1.dat` に対する実行結果

File Name	Sort Method	Data Length	Time Avg (sec)	Memory (KB)
data1.dat	Counting Sort	100000	0.000815	600.00
data1.dat	Bucket Sort	100000	0.003011	1084.88
data1.dat	Quick Sort	100000	0.008440	400.00
data1.dat	Insertion Sort	100000	3.066381	400.00
data1.dat	Shaker Sort	100000	12.019590	400.00
data1.dat	Bubble Sort	100000	16.944878	400.00

5 分析と議論

6 まとめ

7 発表について

参考文献

[1] GeeksforGeeks. Bucket sort. GeeksforGeeks Data Structures & Algorithms Tutorial, 2024. Last updated 23 Jul 2024.

[2] GeeksforGeeks. Quick sort. GeeksforGeeks Data Structures & Algorithms Tutorial, 2025. Last updated 17 Apr 2025.

[3] Apple Inc. Apple silicon cpu optimization guide. Technical report, Apple Inc., 2023. Version 3.0.

[4] Sparsh Mittal. A survey of techniques for dynamic branch prediction. *arXiv preprint arXiv:1805.04389*, 2018.