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

目 次

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ない. 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あるが, 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る.

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る. 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ある.

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るが, 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る. 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ない. 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る. 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ある.

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する. 各データセットには, 1 から 50000 までの整数が 10 万行含まれている. 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る. 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る.

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する.

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ある. 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ある. 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する. 現在の数値が次の数値よりも大きい場合は, 位置を入れ替える. 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る.

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あるが, 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ある. 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, 右にずらす. 適切な位置が見つかったところで, 現在の要素を挿入する.

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る. 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する. 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る. 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する. 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, 高速である. 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する. しかし, 少量のデータであれば問題ない. 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る.

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る. 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る.

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る.

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る. 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る.

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する.

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 表記法を使用する. 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 つである. 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 $O(n^2)$ である. Big O 表記法では, n はデータ自体の長さに対応する. 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る. 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る.

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る. 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 $O(n)$ に減少する. 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る. 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る. 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る.

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 $O(n+k)$ で測定される. ここで, k は外部配列の処理に必要な演算量を表す. 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.

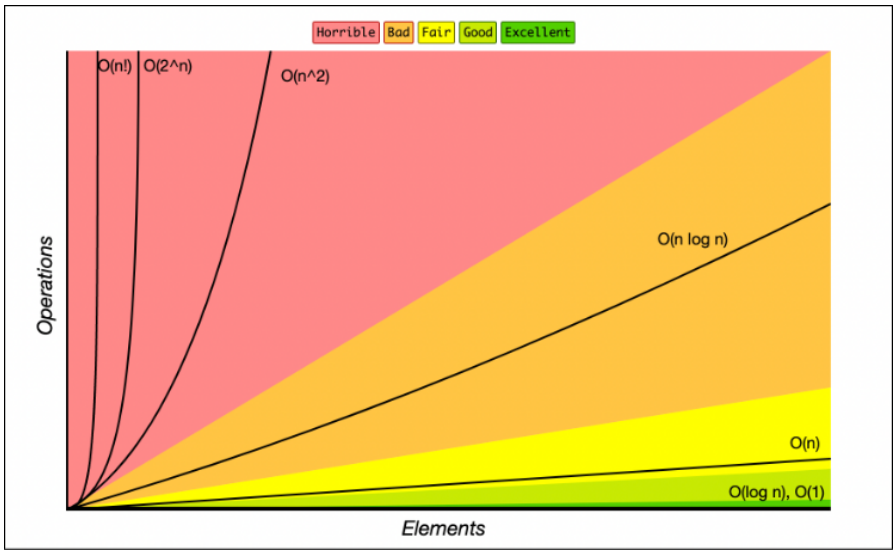


図 1: 様々な計算量を比較

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らない. 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ない. 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る. クイックソートは特に注目すべき点である. クイックソートの空間計算量は $O(\log n)$ であるが, 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ない. 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る. 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ある.

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う.

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る.

```
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 と同時に利用する.

```
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る. 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

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

らはランダムに並べられていることがわかる. 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る. 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る.

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る. 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る.

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

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 のようになる.

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る. 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ある. 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」部分が交互に現れる. 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る.

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

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 のようになる.

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* ライブラリを利用する. *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る. 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する. 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る. 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る.

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.

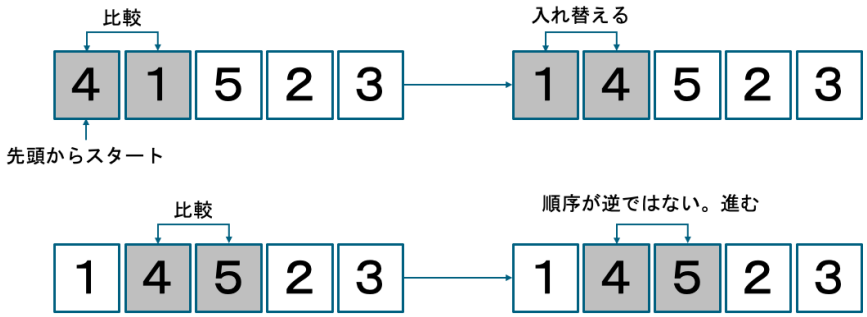


図 3: 先頭から末尾まで数値を比較して, 交換する.

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.

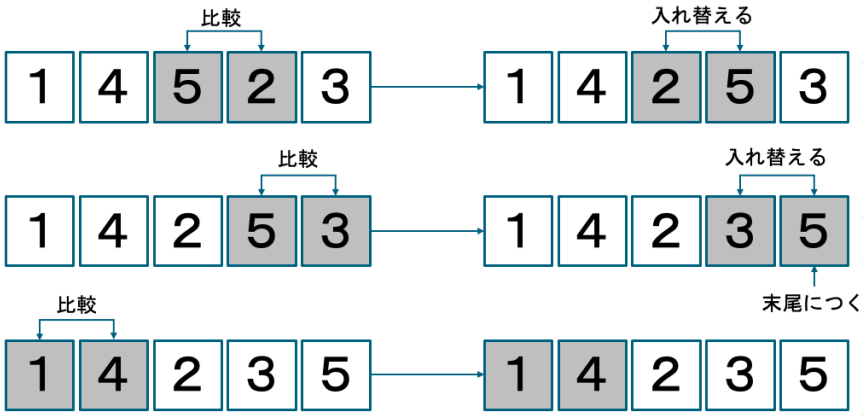


図 4: 通過後最初の位置に戻って, 再び通過

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.

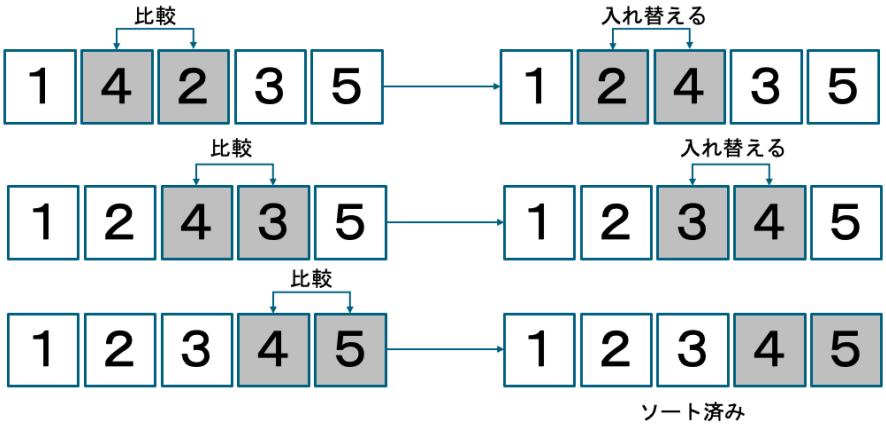


図 5: 通過後に入れ替えされた要素がないと終了

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.

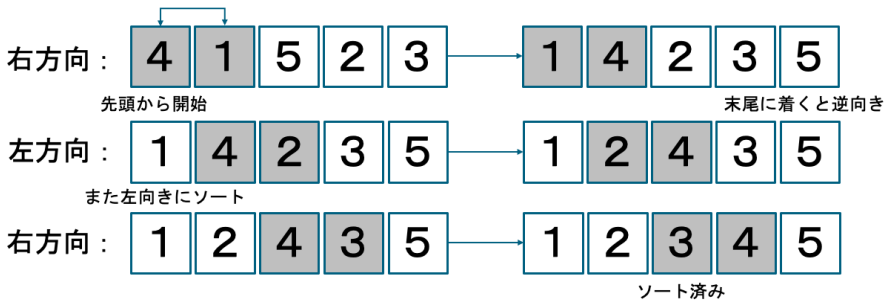


図 6: シェーカーソートの進み方

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

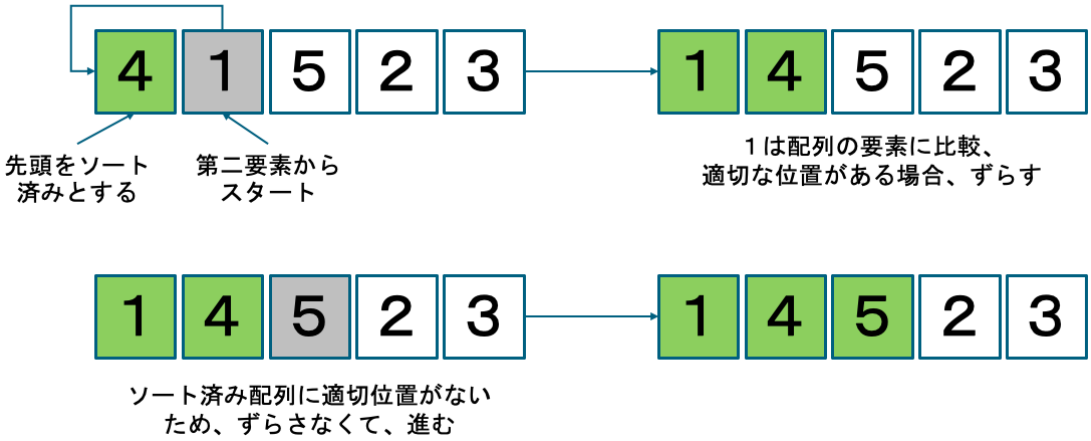


図 7: 第二位置からスタート. ソート済み要素より大きい数値を左にずらす.

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

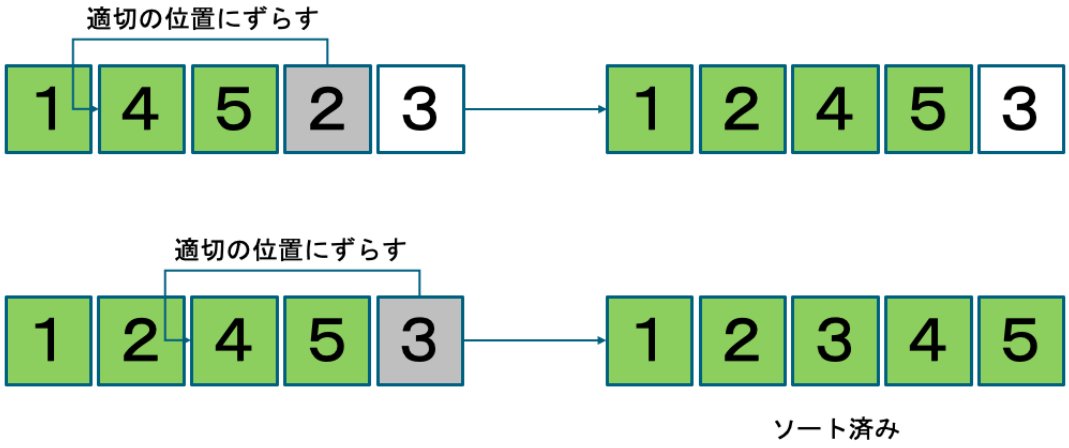


図 8: ソート対象値がないとソート終了

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 としている. 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する.
更に、各バケットをソートする必要があるため、挿入ソートを利用する.[2]

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）.

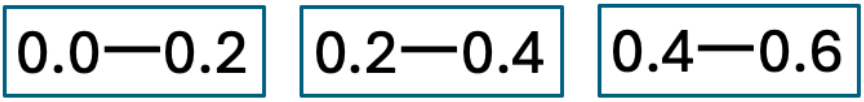


図 10: 作成したバケット

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.

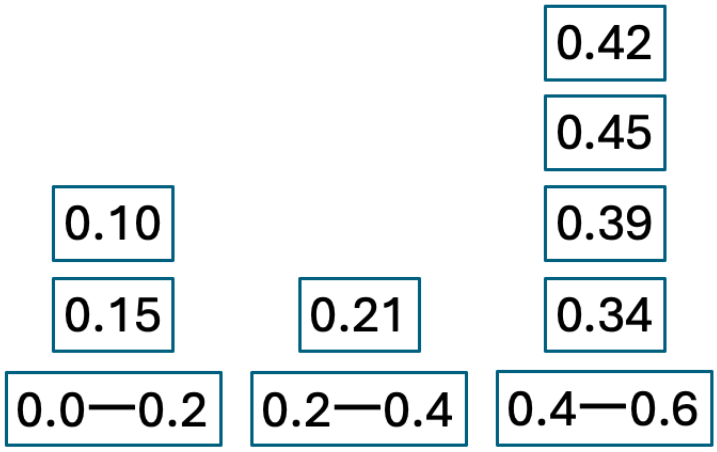


図 11: 適当に作成されたバケットに要素を入れる.

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）.

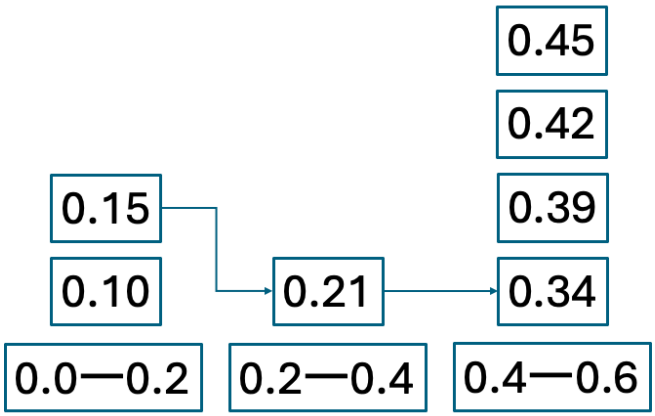


図 12: 各バケットの内容をソートし, 連続

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ある. 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ある. 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する. その結果, 計算量は $O(nlogn)$ から $O(n^2)$ に変化してしまう [4]. この実験では, データ配列の最後の要素をピボット ポイントとして利用して, 提供されたすべてのデータセットに対してクイック ソートを実行する.

1. 初めに, 配列から枢軸（ピボット）を 1 つ選ぶ. 選び方には, 先頭, 中央, 末尾などさまざまな方法がある.
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る（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）に分割する.

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 を再帰的に繰り返す.

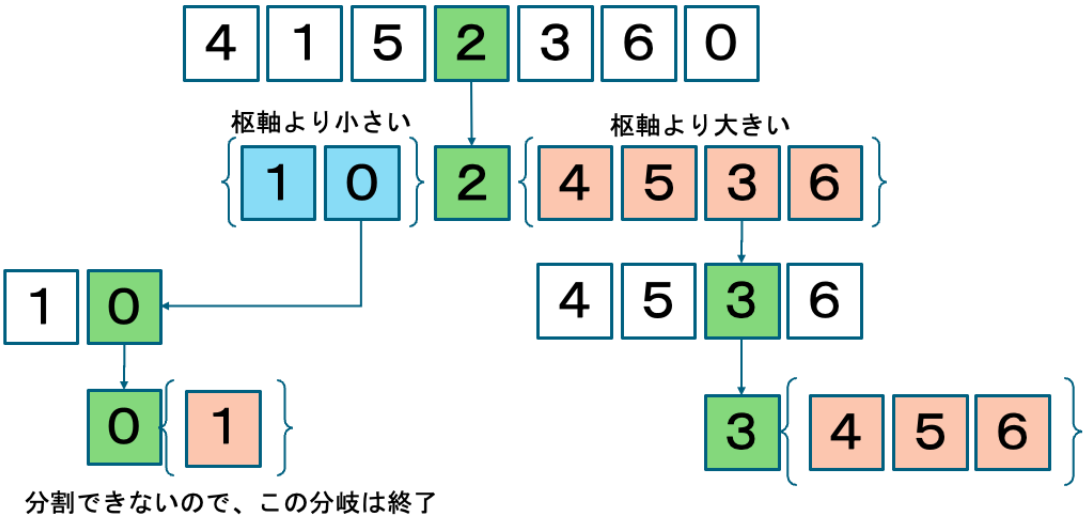


図 14: 枢軸を選び, 左側分と右側分に分割する. 左と右配列を再帰的に処分

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る.

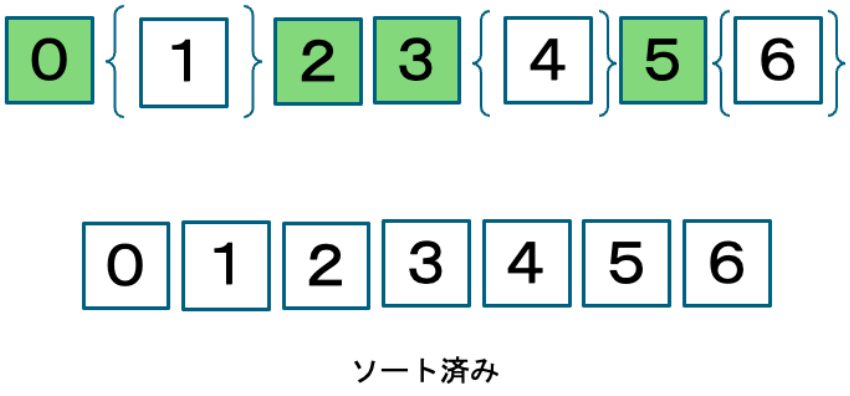


図 15: 部分配列を整列され, 結果的に全体がソートされる

3.4.6 カウントソート

カウントソートでは、データ配列内で最大値を見つけることが重要である。数値が大きいほど、作成されるカウント配列のサイズも大きくなる。そのため、このアルゴリズムのパフォーマンスは値の範囲に影響を受ける可能性がある。この手法では、他のアルゴリズムのように比較やスワップは利用しない [3]。

1. 配列中の最大値と最小値を探索する（負の値が存在する場合は補正が必要となる）.

図 16: ソート対象データから最大値を探す

2. 最大値に基づいてカウント配列（補助配列）を初期化する.

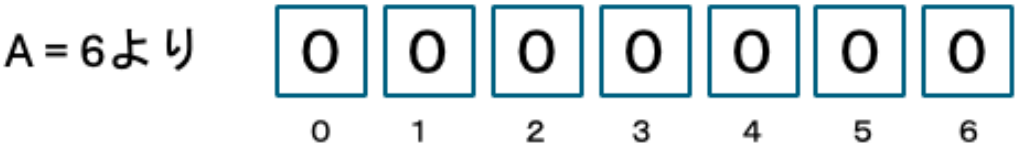


図 17: 最大値を長さとして、カウント配列を作成する

3. 元の配列を走査し，各要素の出現回数をカウント配列に記録する.

4. カウント配列を累積和に変換し，各値の出現位置を計算できるようにする.

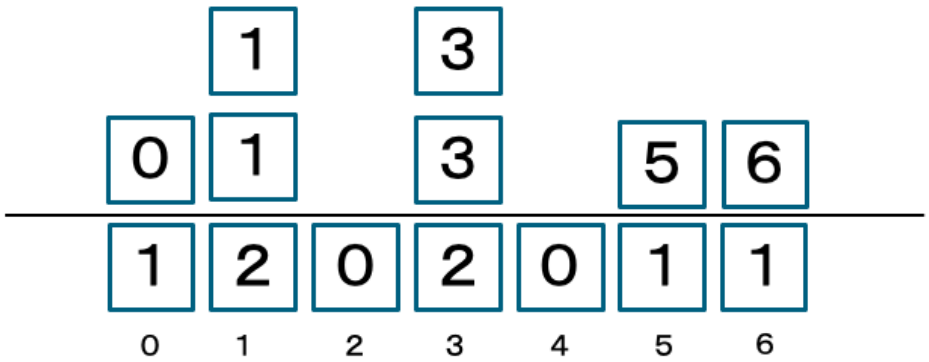


図 18: データ配列の要素をインデックスとし、カウント配列の指定地の値を上げる。

5. 最終的なソート結果を元の配列にコピーする.

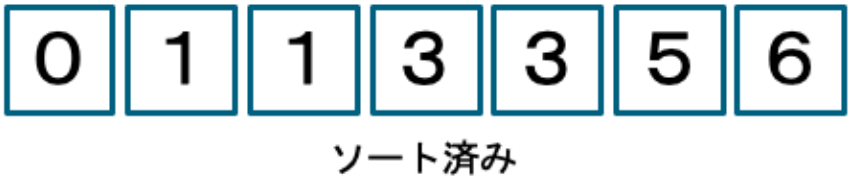


図 19: 最大値を長さとして、カウント配列を作成する

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する.
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る [6]. CPU はまた, 最初の実行時にこのデータセットを RAM にロードする. 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る [5]. データセット１にカウントソートを１０回連続実行すると, 結果は表 5 のようになる.

表 5: カウントソートをデータセット１に対する１０回繰り返した場合

ステップ	メソッド	データ長	時間 (秒)	メモリ (KB)
1	Counting Sort	100000	0.000803	400.00
2	Counting Sort	100000	0.000804	400.00
3	Counting Sort	100000	0.000805	400.00
4	Counting Sort	100000	0.000808	400.00
5	Counting Sort	100000	0.000790	400.00
6	Counting Sort	100000	0.000762	400.00
7	Counting Sort	100000	0.000741	400.00
8	Counting Sort	100000	0.000736	400.00
9	Counting Sort	100000	0.000736	400.00
10	Counting Sort	100000	0.000763	400.00

このデータを図 20 にプロットすると, カウントソートの実行時間が繰り返されるにつれてゆっくりと減少していくことが示している.

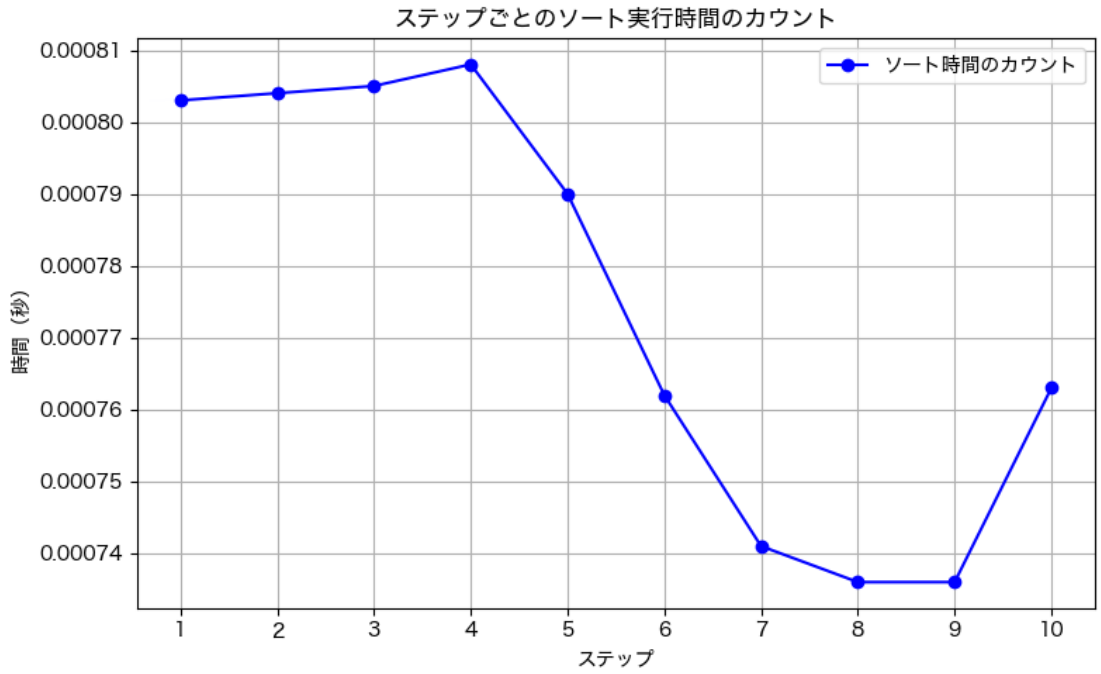


図 20: カウントソートを１０回繰り返し実行する結果グラフ

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る. 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, 結果を平均する.

4.1.1 データセット 1 に対する実行結果

このデータに対する各ソート手法の実行結果は表 6 にまとめられ、図 21 に可視化されている．ランダム構造を持つデータセット 1 に対して利用可能な 6 つのソートアルゴリズムすべてを実行した結果, カウンティングソートとクイックソートの両方でソート速度が大幅に高速であることが見られる。

表 6: 各ソート手法の実行時間とメモリ使用量（データ長 100000, data1.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data1.dat	Counting Sort	100000	0.000764	600.00
data1.dat	Quick Sort	100000	0.008333	400.00
data1.dat	Bucket Sort	100000	0.031620	913.60
data1.dat	Insertion Sort	100000	2.984688	400.00
data1.dat	Shaker Sort	100000	12.459189	400.00
data1.dat	Bubble Sort	100000	14.802736	400.00

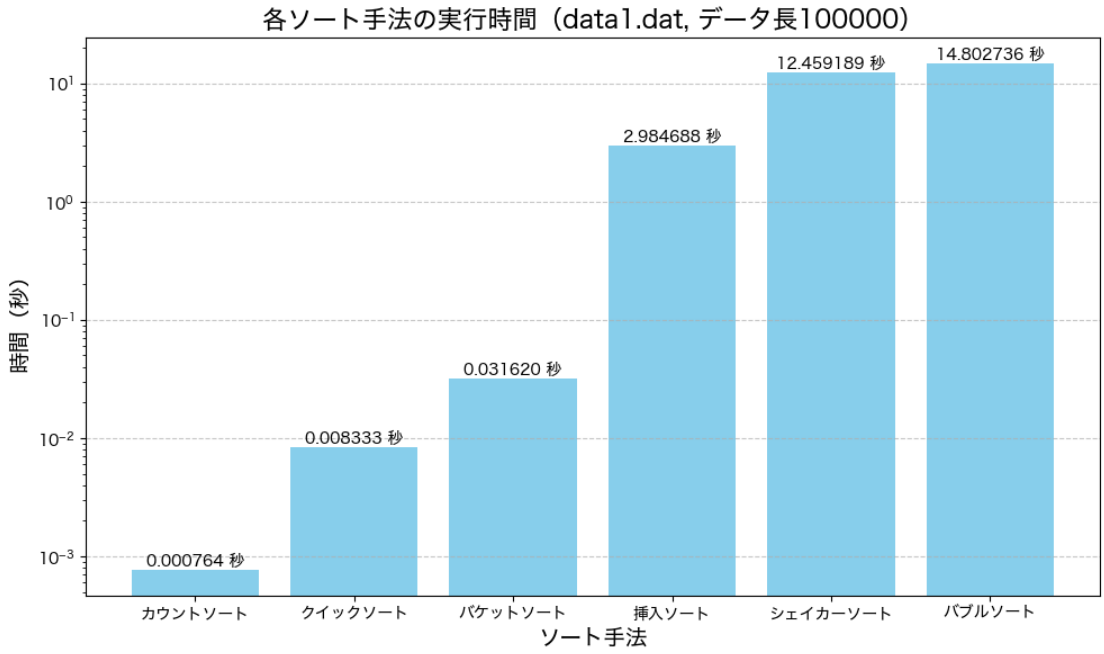


図 21: 各ソート手法の実行時間（データ長 100000）

カウントソートは平均実行時間が 0.000764 秒で最も高速であり、メモリ使用量は 600KB であった。クイックソートは 0.008333 秒の処理時間を要し、メモリ使用量は 400KB と最も少なかった。バケットソートは 0.031620 秒の実行時間で、メモリ使用量は 913.6KB と最大である。挿入ソート、シェイカーソート、バブルソートはそれぞれ 2.984688 秒、12.459189 秒、14.802736 秒の処理時間を示し、メモリ使用量は 400KB で一定であった。これらの結果から、カウントソートは高速であるもののメモリ使用量が多い一方、クイックソートは

メモリ効率に優れていることがわかる。また、バケットソートは比較的遅く、メモリ消費も大きい。挿入ソート以下の単純ソートは処理時間が著しく長い。

4.1.2 データセット 2 に対する実行結果

このデータに対する各ソート手法の実行結果は表 7 にまとめられ、図 22 に可視化されている. データセット 2 のランダム化構造はデータセット 1 と同様であるため, データセット 2 に対してすべてのソート方法を実行した結果, ソート時間とメモリ使用量は同様になる。

表 7: 各ソート手法の実行時間とメモリ使用量（データ長 100000, data2.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data2.dat	Counting Sort	100000	0.000775	600.00
data2.dat	Quick Sort	100000	0.008439	400.00
data2.dat	Bucket Sort	100000	0.031297	913.60
data2.dat	Insertion Sort	100000	2.990967	400.00
data2.dat	Shaker Sort	100000	12.589956	400.00
data2.dat	Bubble Sort	100000	14.921321	400.00

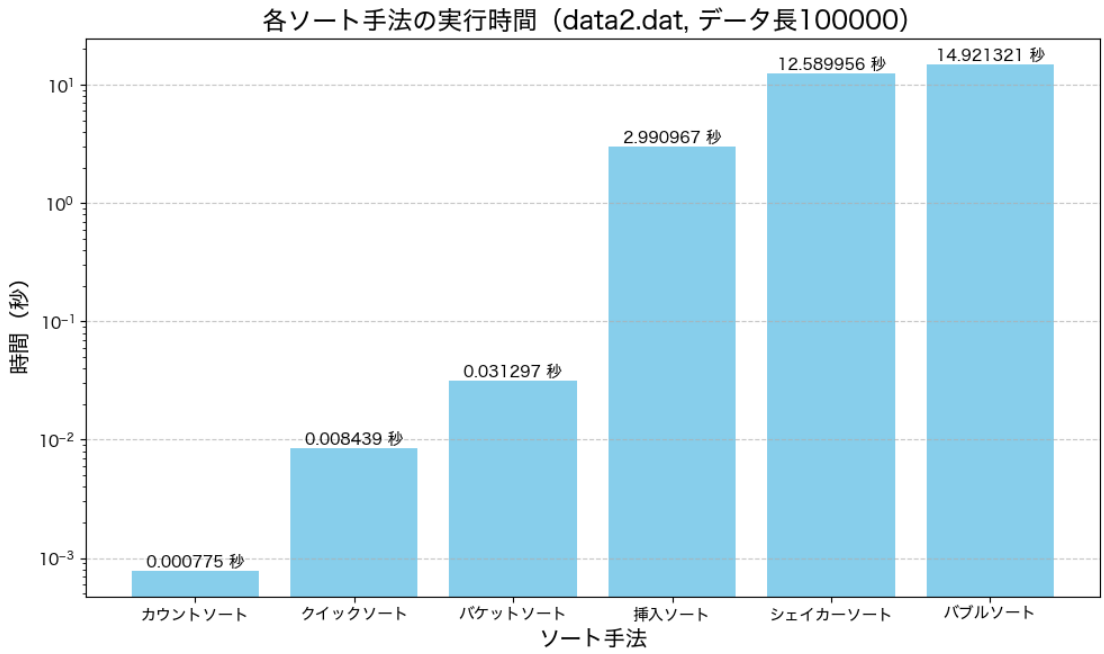


図 22: 各ソート手法の実行時間（データ長 100000）

カウントソートは平均実行時間が 0.000775 秒で最も高速で、メモリ使用量は 600KB であった。クイックソートは 0.008439 秒の処理時間を要し、メモリ使用量は 400KB と最小だった。バケットソートは 0.031297 秒かかり、メモリ使用量は 913.6KB と最も多い。挿入ソート、シェイカーソート、バブルソートはそれぞれ 2.990967 秒、12.589956 秒、14.921321

秒の処理時間を示し、メモリ使用量は 400KB で一定である。この結果から、カウントソートは高速性に優れている一方でメモリ消費がやや大きいこと、クイックソートはメモリ効率が良いがバケットソートに比べて高速であることが確認できる。

4.1.3 データセット 3 に対する実行結果

このデータに対する各ソート手法の実行結果は表 8 にまとめられ、図 23 に可視化されている. データセット 3 のランダム化構造はデータセット 1 と同様であるため, データセット 3 に対してすべてのソート方法を実行した結果, ソート時間とメモリ使用量は同様になる.

表 8: 各ソート手法の実行時間とメモリ使用量（データ長 100000、data3.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data3.dat	Counting Sort	100000	0.000791	600.00
data3.dat	Quick Sort	100000	0.008298	400.00
data3.dat	Bucket Sort	100000	0.032391	913.60
data3.dat	Insertion Sort	100000	2.970459	400.00
data3.dat	Shaker Sort	100000	12.603889	400.00
data3.dat	Bubble Sort	100000	14.806170	400.00

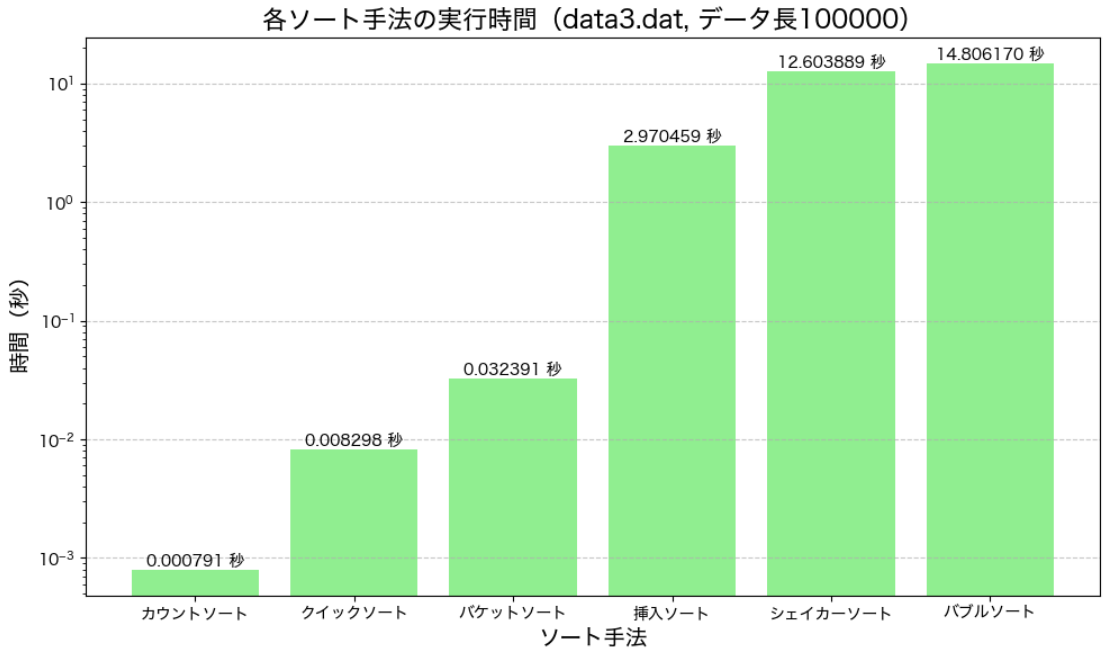


図 23: 各ソート手法の実行時間（データ長 100000）

カウントソートは平均実行時間が 0.000791 秒で最も高速であり、メモリ使用量は 600KB であった。クイックソートは 0.008298 秒とカウントソートより遅いが、メモリ使用量は 400KB と少ない。バケットソートは 0.032391 秒かかり、メモリ使用量は 913.6KB と最も

多い。挿入ソート、シェイカーソート、バブルソートはそれぞれ 2.970459 秒、12.603889 秒、14.806170 秒の実行時間で、メモリ使用量は 400KB で一定である。全体として、カウントソートが最も高速であり、バケットソートは高いメモリ使用量と比較的遅い処理時間を示している。また、単純な比較ではクイックソートがバランスの取れた性能であることが分かる。

4.1.4　データセット 4 に対する実行結果

セクション 3.2 で述べた通り、すでにソート済みのデータセット 4 に対して各ソートアルゴリズムを実行した結果を表 9 にまとめられ、図 24 に可視化されている

表 9: 各ソート手法の実行時間とメモリ使用量（データ長 100000、data4.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data4.dat	Shaker Sort	100000	0.000110	400.00
data4.dat	Bubble Sort	100000	0.000117	400.00
data4.dat	Insertion Sort	100000	0.000196	400.00
data4.dat	Counting Sort	100000	0.001088	600.00
data4.dat	Bucket Sort	100000	0.001245	913.60
data4.dat	Quick Sort	100000	4.036942	400.00

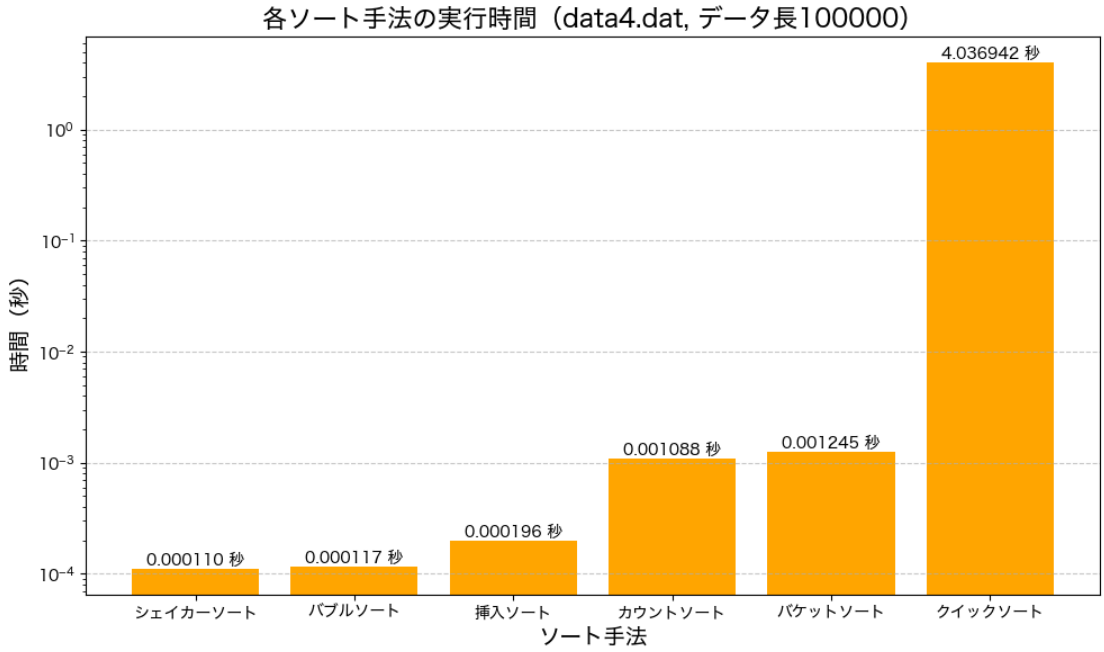


図 24: 各ソート手法の実行時間（データ長 100000）

シェーカーソートは 0.000110 秒、バブルソートは 0.000117 秒、挿入ソートは 0.000196 秒で処理を完了した。カウントソートは 0.001088 秒、バケットソートは 0.001245 秒であった。一方、クイックソートは 4.036942 秒と最も遅い結果となった。メモリ使用量はシェーカーソート、バブルソート、挿入ソート、クイックソートが 400.00 KB、カウントソートが 600.00 KB、バケットソートが 913.60 KB であった。

4.1.5　データセット 5 に対する実行結果

3.2 節の各データセットの説明で述べたように、データセット 5 は逆順の状態でソートされている。このデータに対する各ソート手法の実行結果は表 10 にまとめられ、図 25 に可視化されている。最も高速だったのはカウントソートであり、わずか 0.000780 秒で処理を完了した。一方、バケットソートは 0.059791 秒を要し、メモリ使用量は全手法中最大の 913.60 KB であった。クイックソートは 4.485886 秒を要し、挿入ソート（6.175989 秒）、バブルソート（10.275611 秒）、シェーカーソート（14.925627 秒）と比較して高速であったが、カウントソートおよびバケットソートには大きく劣っていた。

表 10: 各ソート手法の実行時間とメモリ使用量（データ長 100000、data5.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data5.dat	Counting Sort	100000	0.000780	600.00
data5.dat	Bucket Sort	100000	0.059791	913.60
data5.dat	Quick Sort	100000	4.485886	400.00
data5.dat	Insertion Sort	100000	6.175989	400.00
data5.dat	Bubble Sort	100000	10.275611	400.00
data5.dat	Shaker Sort	100000	14.925627	400.00

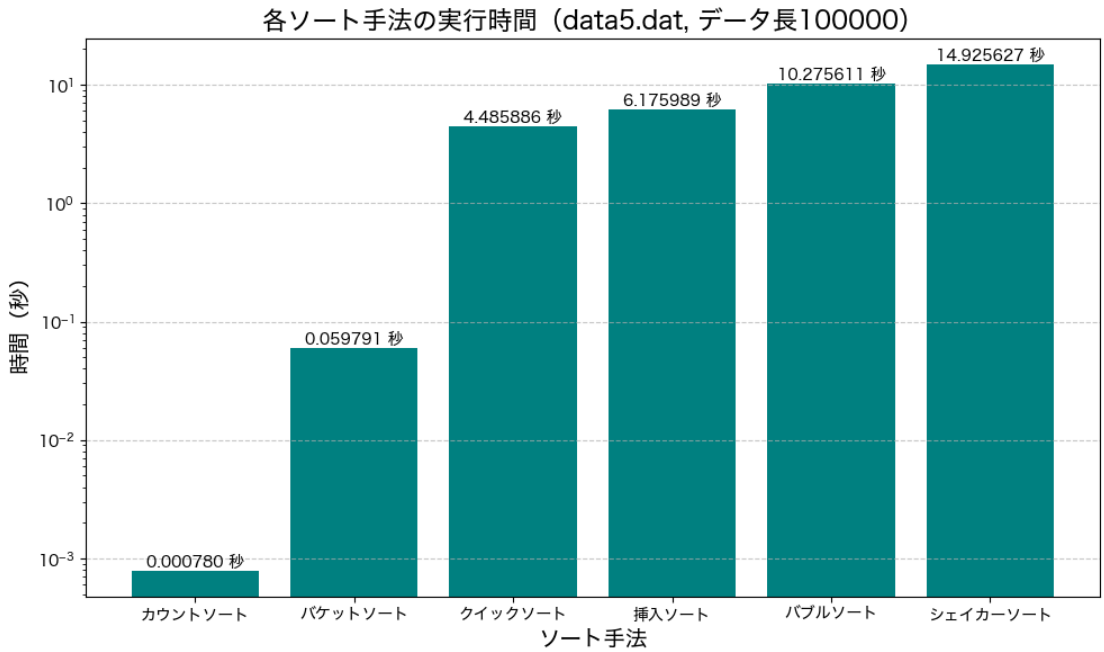


図 25: 各ソート手法の実行時間（データ長 100000）

4.1.6　データセット 6 に対する実行結果

3.2 節の各データセットの説明で述べたように、データセット 6 バイトニックで構成されている。このデータに対する各ソート手法の実行結果は表 11 にまとめられ、図 26 に可

視化されている。最も高速だったのはカウントソートであり、0.000979 秒で処理を完了した。次に高速だったのはバケットソートであり、0.035034 秒を要したが、メモリ使用量は最大の 913.60 KB であった。クイックソートは 0.800326 秒であり、挿入ソート（3.265788 秒）、バブルソート（8.474608 秒）、シェーカーソート（9.416852 秒）よりも高速だったが、カウントソートおよびバケットソートに対しては依然として劣る結果となった。

表 11: 各ソート手法の実行時間とメモリ使用量（データ長 100000、data6.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data6.dat	Counting Sort	100000	0.000979	600.00
data6.dat	Bucket Sort	100000	0.035034	913.60
data6.dat	Quick Sort	100000	0.800326	400.00
data6.dat	Insertion Sort	100000	3.265788	400.00
data6.dat	Bubble Sort	100000	8.474608	400.00
data6.dat	Shaker Sort	100000	9.416852	400.00

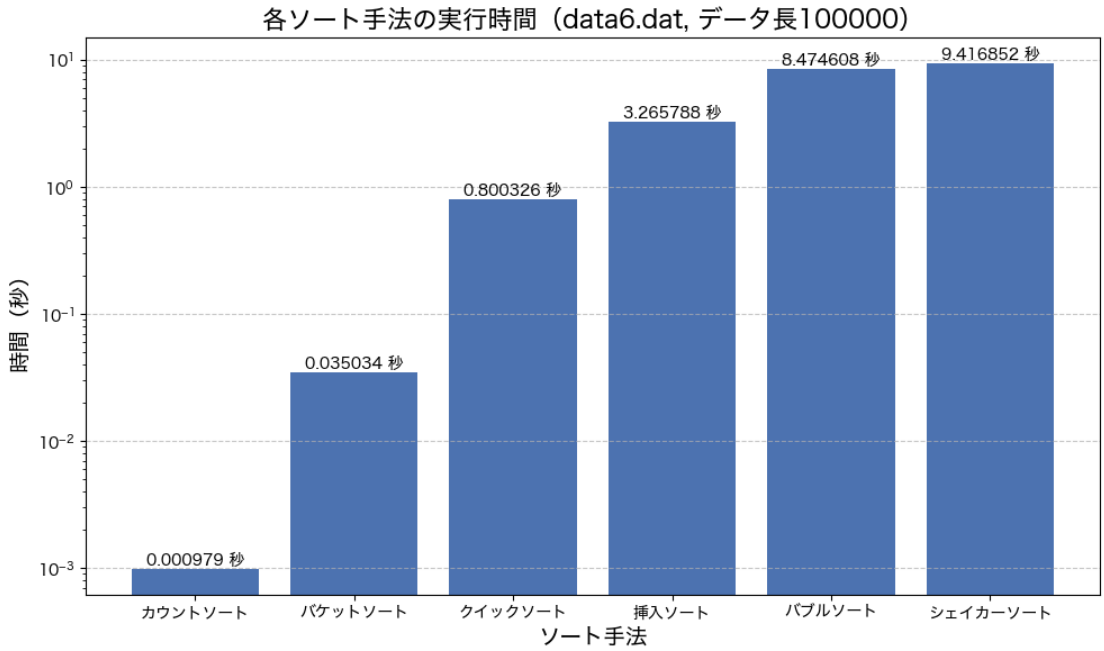


図 26: 各ソート手法の実行時間（データ長 100000）

4.1.7 データセット 7 に対する実行結果

3.2 節で説明したように、データセット 7 はジグザグ状態である。このデータに対する各ソート手法の実行結果は表 12 にまとめられ、図 27 に可視化されている。最も高速だったのはカウントソートで、0.000835 秒で処理を完了した。次いでクイックソートが 0.009582 秒でデータを処理し、メモリ使用量も少ない。バケットソートは 0.033965 秒とやや時間を要したが、他の比較的遅いアルゴリズム群、すなわち挿入ソート（3.168507 秒）、シェー

カーソート（12.092268 秒）、バブルソート（14.169538 秒）と比べれば十分高速であることがわかる。

表 12: 各ソート手法の実行時間とメモリ使用量（データ長 100000、data7.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data7.dat	Counting Sort	100000	0.000835	600.00
data7.dat	Quick Sort	100000	0.009582	400.00
data7.dat	Bucket Sort	100000	0.033965	913.60
data7.dat	Insertion Sort	100000	3.168507	400.00
data7.dat	Shaker Sort	100000	12.092268	400.00
data7.dat	Bubble Sort	100000	14.169538	400.00

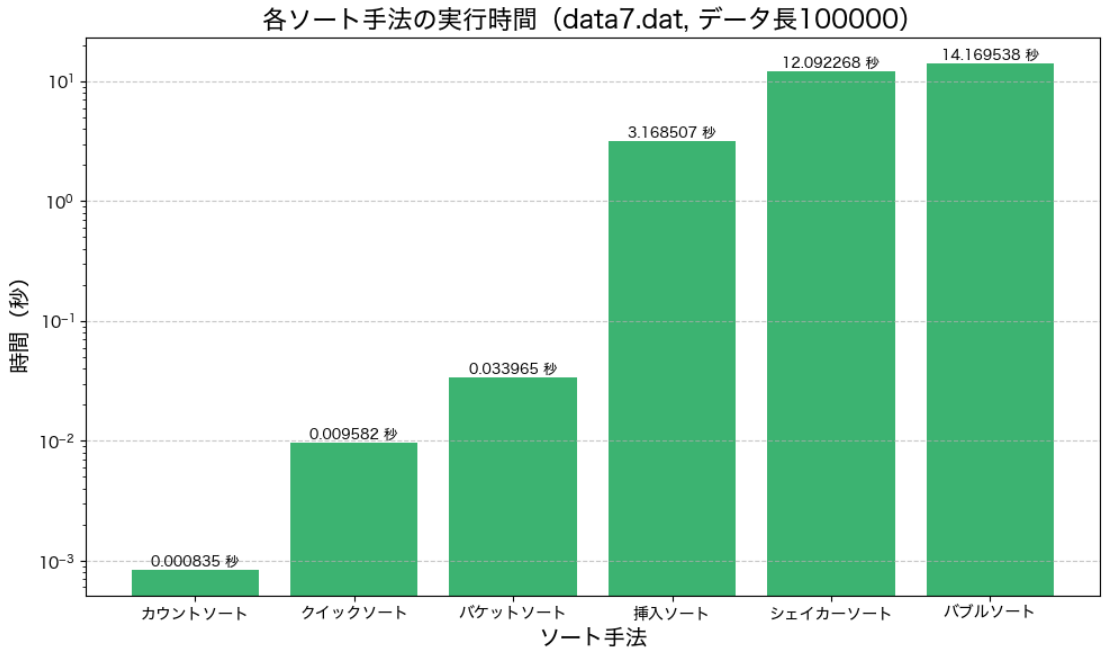


図 27: 各ソート手法の実行時間（データ長 100000）

4.2 データ数のソート時間に対する影響

一般的なデータセットに対するデータ量の影響を視覚化するために、データセット 1 にバブルソートを実行し、さまざまなデータサイズがソート速度にどのように影響するか検査する。バブルソートにおけるデータ長の増加に伴う処理時間の変化を表 13 にまとまれ、図 28 に可視化されている。データ長が 10 の場合、実行時間は 0.000001 秒とほぼ無視できる程度であるが、1,000 件では 0.001039 秒、10,000 件では 0.097349 秒、そして 100,000 件になると 14.997694 秒にまで増加している。この結果は、バブルソートの計算量が $O(n^2)$ であることを実証しており、データサイズの増加に対して実行時間が指数的

に増加する傾向があることが確認できる。また、メモリ使用量はデータ長に比例して増加し、10 件で 0.04 KB、100,000 件で 400.00 KB となった。

表 13: データ長の増加に伴うバブルソートの実行時間とメモリ使用量（data1.dat）

ファイル名	ソート手法	データ長	時間（秒）	メモリ（KB）
data1.dat	バブルソート	10	0.000001	0.04
data1.dat	バブルソート	100	0.000024	0.40
data1.dat	バブルソート	1000	0.001039	4.00
data1.dat	バブルソート	10000	0.097349	40.00
data1.dat	バブルソート	100000	14.997694	400.00

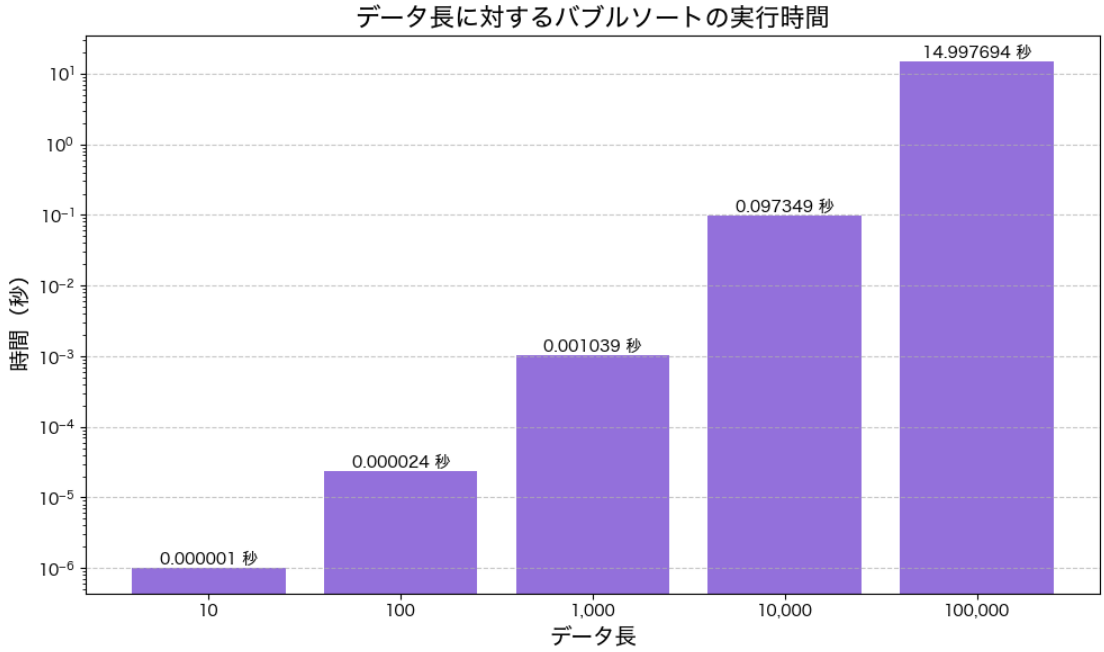


図 28: データ長による実行時間

4.3　バケット数のバケットソート時間に対する影響

データセット 1 に対して、異なるバケット数を設定したバケットソートの性能を測定した結果を表 22 にまとまれ、図 29 と図 30 で示されている。バケット数を 10 とした場合、処理時間は 0.295539 秒と最も遅く、メモリ使用量は 809.76 KB であった。バケット数を 100 に増やすと、処理時間は 0.031729 秒に大幅に短縮された。さらに 1,000 バケットでは 0.004825 秒、10,000 バケットでは 0.002712 秒と、処理速度が向上していることが確認できる。一方、バケット数 100,000 の場合、処理時間は一時的に 0.006176 秒と増加し、メモリ使用量は 6,000.04 KB に達した。

表 14: バケット数ごとのバケットソートの実行時間とメモリ使用量（データ長 100000）

バケット数	データ長	時間（秒）	メモリ（KB）
10	100000	0.295539	809.76
100	100000	0.031729	913.60
1000	100000	0.004825	1039.80
10000	100000	0.002712	1186.28
100000	100000	0.006176	6000.04

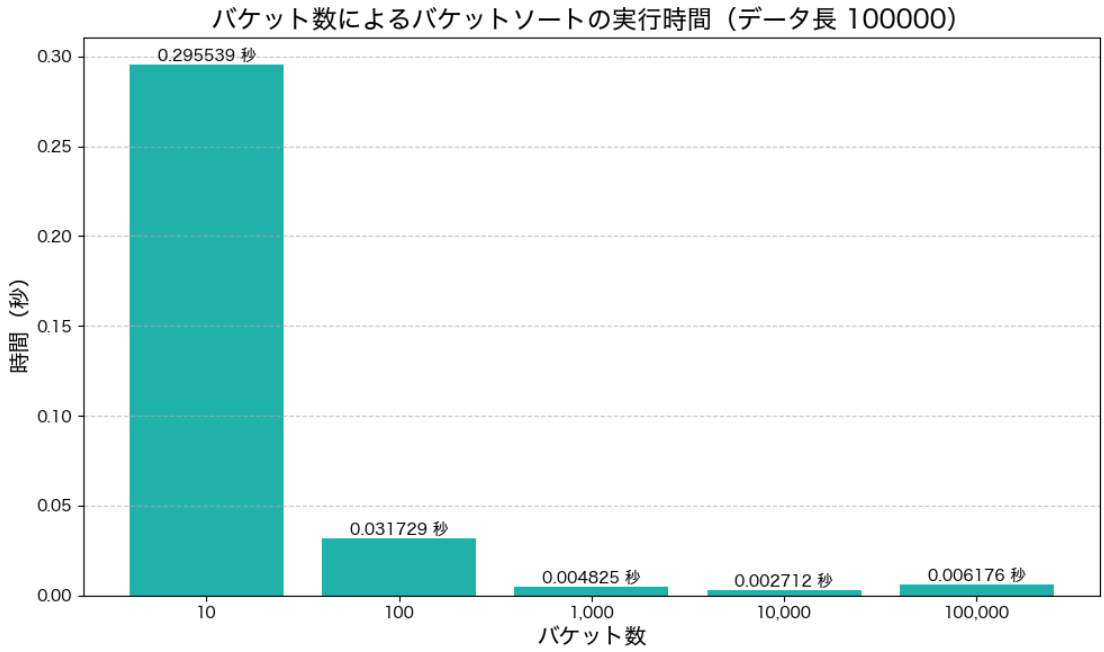


図 29: バケット数による実行時間（データ長 100000）

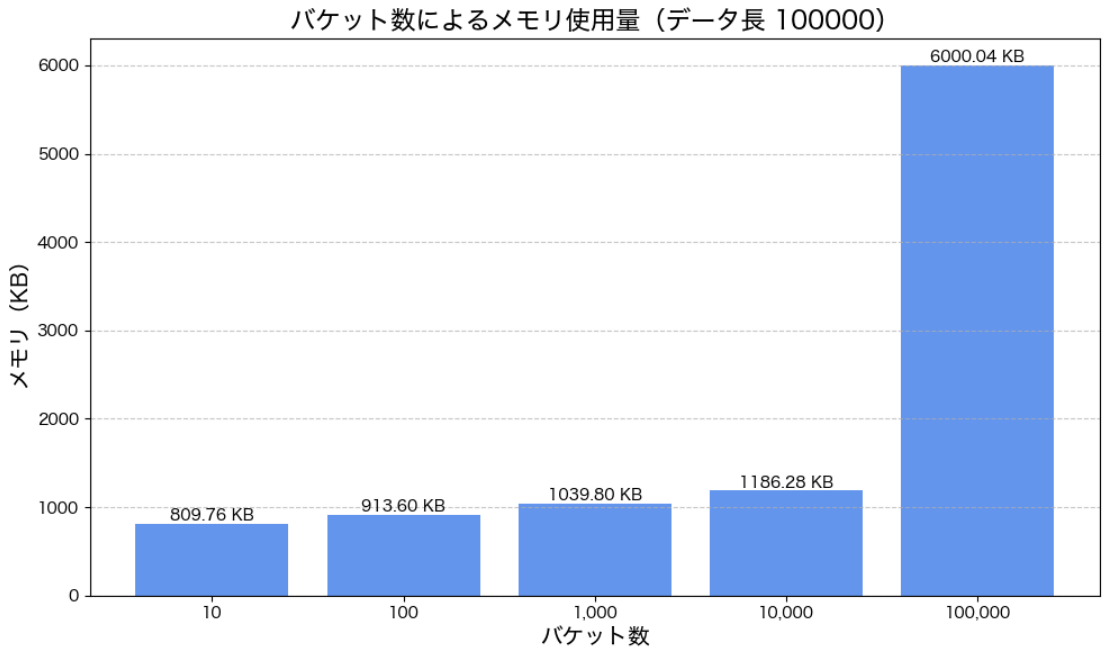


図 30: バケット数によりメモリ使用量（データ長 100000）

5　分析と議論

5.1　データ状態によって、実行結果を分析

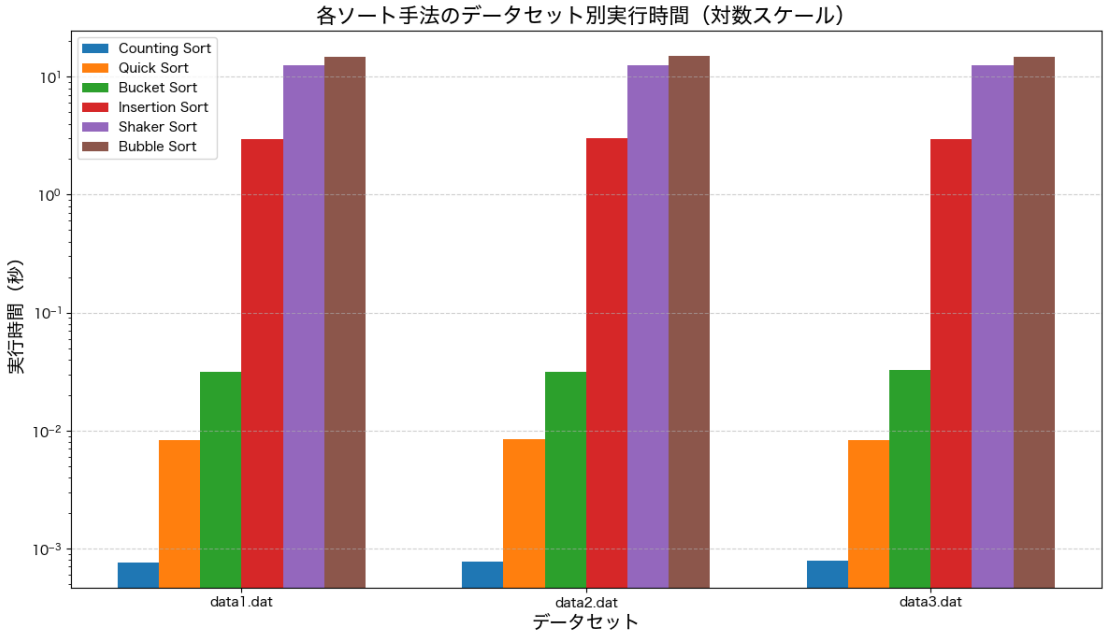
5.1.1　ランダム構造のデータセット

セクション 3.2 の表 2 で説明したように、データセット 1、2、3 のすべてにランダムに構造化された整数が含まれている。表 6, 表 7 と表 8 にまとめられたデータセット 1, 2, 3 の結果を表 15 にまとまれ、図 31 に可視化されている。

表 15: 各データセットにおけるソート手法別の実行時間（データ長：100000）および平均時間

ソート手法	data1.dat 時間（秒）	data2.dat 時間（秒）	data3.dat 時間（秒）	平均時間（秒）
カウントソート	0.000764	0.000775	0.000791	0.000777
クイックソート	0.008333	0.008439	0.008298	0.008357
バケットソート	0.031620	0.031297	0.032391	0.031769
挿入ソート	2.984688	2.990967	2.970459	2.982705
シェイカーソート	12.459189	12.589956	12.603889	12.551678
バブルソート	14.802736	14.921321	14.806170	14.843409

図 31: ランダムデータセットの実行結果を比較



すべて以前にランダム化された構造を持つデータセット 1、2、3 の結果を比較すると、同じソートアルゴリズム内のデータセット間に大きな変化はないことが示されている。 ランダム構造データセット内では、一貫して最も速いソート時間をカウント ソートで達成しており、すべてのデータセットを平均 0.000777 秒でソートすることができた。これに続いてクイック ソートがあり、すべてのデータセットを平均 0.008357 秒でソートすることができ、975.6% 以上の増加となっている。ただし、処理時間のこの大幅な増加は、約 33% のメモリ使用量の減少も伴っている。表 16 に示すように。処理時間のこの大幅な増加は、セクション 2.2 の表 1 で以前に説明したように、それぞれ計算量 $O(n+k)$ と $O(n \log n)$ で表されるカウント ソートとクイック ソートの一般的な動作によるものである。ただし、クイック ソートはデータをインプレース つまり、同じデータセット配列内 で処理できることを考慮すると、ソートされる整数の範囲に大きく依存するカウント ソートと比較して、クイック ソート アルゴリズムはメモリをはるかに効率的に割り当てることができた。

表 16: 各データセットにおけるソート手法別のメモリ使用量（KB）および平均使用量（データ長：100000）

ソート手法	data1.dat メモリ（KB）	data2.dat メモリ（KB）	data3.dat メモリ（KB）	平均メモリ（KB）
カウントソート	600.00	600.00	600.00	600.00
クイックソート	400.00	400.00	400.00	400.00
バケットソート	913.60	913.60	913.60	913.60
挿入ソート	400.00	400.00	400.00	400.00
シェイカーソート	400.00	400.00	400.00	400.00
バブルソート	400.00	400.00	400.00	400.00

クイックソートとバケットソートの結果を比較すると、ランダム構造データセット全体をそれぞれ平均 0.008357 秒と 0.031760 秒でソートが完了した。これら 2 つのアルゴリズムでは、ソート時間が約 280.1% 増加している。しかし、ソート時間の増加に伴い、メモリ

使用量も平均 400KB から 913KB に増加している。メモリ使用量の増加は、バケットソートでは割り当てる必要のあるバケット数が事前に設定されているためである。各バケットはそれ自体が配列であるため、データセット配列から値を追加するために追加のメモリ空間を割り当てる必要がある。ソート時間の増加は、各バケットをソートするために、より単純だが低速なソートアルゴリズムを使用していることに起因している。この実験では、小規模なデータセットでより効率的に動作することが知られている挿入ソートを使用した [1]。実験設定セクション 3.4.4 で前述したように、100 個のバケットも使用した。各データセットには 100000 件のデータが含まれているという事実を考慮すると、100000 のデータ 100 バケット は、すべてのデータがすべてのバケットに均等に分散される最良のシナリオであっても、各バケットには挿入ソートで処理する必要がある約 1000 個のデータが含まれることを意味する。Bingmann 2022 は、小規模なアイテムセットに対する高速ソーターについて論文の中で言及しており、挿入ソートはわずか 20 要素の配列で最も優れたパフォーマンスを発揮する。

最も遅い 3 つのアルゴリズムを見ると、挿入ソートの平均ソート時間は 2.982705 秒で最も良好なパフォーマンスを示した。一方、シェイカー ソートとバブル ソートは、提供されたデータセットをそれぞれ平均 12.551678 秒と 14.843409 秒でソートした。興味深いことに、セクション 2.2 の表 1 で前述したアルゴリズムの計算量を見ると、バブル ソート、シェイカー ソート、挿入ソートのすべてが、最良、平均、最悪のシナリオで同じ計算量になっている。ただし、挿入ソートとシェイカー ソートおよびバブル ソートの結果を比較すると、挿入ソートはそれぞれ 76.2% と 79.9% 速く終了することができた。理論的には、挿入ソート、シェイカー ソート、バブル ソートは同様に実行できるはずである。これは、シェイカー ソートとバブル ソートの両方が、単一の値を正しい場所に移動するためだけにデータセット全体を複数回走査する必要があるという事実によって説明できる。一方、挿入ソートでは、値を正しい位置に移動するために複数回走査する必要はない。挿入ソートでは 1 回のみ走査が必要である。

5.1.2 昇順的なデータ（ソート済み）

すでにソート済みのデータセットにすべてのソートアルゴリズムを適用することは、アルゴリズムが見つけることができる理論的に最良のシナリオをテストすることである。この場合、すべてのソートアルゴリズムは、表 2 で説明した、すでにソート済みのデータセットを含むデータセット 4 に対して実行される。セクション 4.1.4 に示すように、この実行の結果は、セクション 5.1.1 で以前に説明したランダム化データ構造と比較して、ソート速度が大幅に変化したことを示している。この実行では、シェーカー ソート、バブル ソート、挿入ソートによって最速のソート時間が達成された。シェーカー ソートとバブル ソートはどちらも 0.000007 秒の時間差があり、これは誤差の範囲内である。両方のアルゴリズムが同様に機能するという事実を考慮すると、この小さな時間差は無視でき、重要な意味を持たないデータノイズによるものと考えられる。しかし、シェーカーソートとバブルソートの結果を挿入ソートと比較すると、ソート時間の差はそれぞれ 0.000086 秒（78.18%の増加）、0.000079 秒（67.52%の増加）と、より大きな差が見られる。前述の 3 つのアルゴリズムは、最良のケース、最悪のケース、平均的なケースのいずれにおいても計算量が同じであるにもかかわらず、挿入ソートはパフォーマンスが劣っていた。現在、

最良のケースのデータセットにおいて、挿入ソートがバブルソートやシェーカーソートよりもパフォーマンスが低い理由について、明確な研究や調査は行われていない。

しかし、プロファイラで挿入ソートを実行すると、興味深い結果が得られた。デバッグシンボルを付加してプログラムを再コンパイルし、MacOS に付属の *sample* プログラムを使用してプロファイリングした。

表 17: プロフィリングの呼び出しスタック (サンプル数でソート)

Function	**Samples**
main (in algcomp)	3736
insertionSort (in algcomp)	3736
insertionSort +64,68	1806
insertionSort +32	1655
insertionSort +44,36	272
insertionSort +52	3
_platform_memmove (libsystem)	554
main + 648 (in algcomp)	554
main + 884 (in algcomp)	29
clock (libsystem_c)	29
getrusage (libsystem_kernel)	28
main + 904 (in algcomp)	23
clock (libsystem_c)	23
getrusage (libsystem_kernel)	23
DYLD-STUB$$getrusage (libsystem_c)	1
start (in dyld)	4343

表 17 に示すように、システムコールの大部分は *insertionSort* 関数によって実行され、合計 3736 回呼び出された。しかし、*memmove* による重要なシステムコールも合計 554 回実行された。C 言語では、*memmove* はメモリに値を代入する動作である。

```
void insertionSort(int arrayLength, int *array){
  for (int i = 1; i < arrayLength; ++i){
    int key = array[i];
    int j = i - 1;
    while (j >= 0 && array[j] > key){
      array[j + 1] = array[j];
      j = j - 1;
    }
    array[j + 1] = key;  //ここに注目
  }
}
```

挿入ソート関数は、対象値とソート済み配列を比較した後にスワップが検出されない場合でも、ループごとにメモリへの書き込み操作 $array[j+1] = key;$ を実行するように記述

されている。これにより、ソート速度にわずかながらも顕著な遅延が発生し、挿入ソートがバブルソートやシェイカーソートよりもわずかにパフォーマンスが劣る理由を説明できるかもしれない。

5.1.3 降順的なデータ（逆ソート済み）

これらのソートアルゴリズムを逆順にソートされたデータに適用する主な目的は、最悪のシナリオでアルゴリズムの動作をテストすることである。表 10 の実行結果と、表 7 のデータセット 2 の結果を比較すると、表 18 により、以前は効率的だった多くのアルゴリズムで実行時間が大幅に増加している。

表 18: ソート時間の比較と変化率（単位：秒, 変化率は data2 から data5 への%増減）

ソート手法	data2.dat	data5.dat	変化率 (%)
カウントソート	0.000775	0.000780	+0.65%
クイックソート	0.008439	4.485886	+53099.69%
バケットソート	0.031297	0.059791	+91.01%
挿入ソート	2.990967	6.175989	+106.51%
バブルソート	14.921321	10.275611	−31.13%
シェイカーソート	12.589956	14.925627	+18.55%

カウントソートでは比較が不要なため、大きな時間差は見られない。クイックソートでは、パフォーマンスが 1 秒未満から 4 秒以上に大幅に低下した。これは、アルゴリズムが配列の最後の要素（このデータセットでは最小の要素）をピボットポイントとして選択するように実装されていることに起因している。つまり、アルゴリズムはピボットポイント以外のすべての要素を配列の右側に移動する必要があり、平均計算量は $O(n \log n)$ から $O(n^2)$ に変化する。これは、クイックソートの最大の欠点の 1 つであり、配列の最小の要素をピボットポイントとして選択するとアルゴリズムのパフォーマンスが低下する可能性があることを示している。そのため、クイックソートの効率とパフォーマンスを維持するには、最適なピボットポイントを選択することが不可欠である。

バケットソートではパフォーマンスがわずかに低下するのであるが、これはバケット内の要素の構造に起因する。各バケットには、実装された挿入ソートを使用して処理する必要がある反転配列が含まれる。挿入ソートの結果を見ると、パフォーマンスが 2.990967 秒から 6.175989 秒へと大幅に低下していることがわかる。表 1 で説明したように、挿入ソートの最悪のシナリオは $O(n^2)$ である。これは挿入ソートの仕組みによるものである。データが反転されると、アルゴリズムはループごとに各対象要素を配列の先頭までプッシュする必要があり、ソート時間に大きな影響を与える。

最も遅い 3 つのアルゴリズムを見ると、挿入ソートの平均ソート時間が 2.982705 秒で最も良好なパフォーマンスを示した。一方、シェイカー ソートとバブル ソートは、提供されたデータセットをそれぞれ平均 12.551678 秒と 14.843409 秒でソートした。セクション 2.2 の表 1 で前述したアルゴリズムの計算量を見ると、バブル ソート、シェイカー ソート、挿入ソートのすべてが、最良、平均、最悪のシナリオで同じ計算量になっている。ただ

し、挿入ソートとシェイカー ソートおよびバブル ソートの結果を比較すると、挿入ソートはそれぞれ 76.2% と 79.9% 速く終了することができた。理論的には、挿入ソート、シェイカー ソート、バブル ソートは同様に実行できるはずである。これは、各ループを (パスごとに) 2 回実行する必要があることによる追加のオーバーヘッドによって説明できる。さらに、CPU は予測可能な順方向および線形アクセスに最適化されている。シェイカーソートの逆トラバーサルはこのパターンを破壊し、アルゴリズムのパフォーマンスに影響を与えるオーバーヘッドを追加する。

5.1.4 バイトニックなデータ

前回と同様に、カウントソートでは比較が行われていないため、表 19 のデータセット 2 とデータセット 6 のソート結果を比較しても、目立った改善や低下は見られない。0.0002 秒というわずかな差は誤差範囲内であり、データノイズに起因する可能性がある。バイトニックデータとは、データの前半が最小値から最大値へとゆっくりと上昇し、最後に向かって再びゆっくりと下降する、半分ソートされたデータと簡単に説明できる。このため、バブルソートやシェイカーソートなどの一部のソートアルゴリズムでは改善が見られる。カウントソートと同様に、バケットソートでは目立った改善や低下は見られなかった。データセット 2 とデータセット 6 の間のわずかな差は、データノイズに起因する可能性がある。

表 19: ソート時間の比較と変化率（単位：秒, 変化率は data2 から data6 への%増減）

ソート手法	data2.dat	data6.dat	変化率 (%)
カウントソート	0.000775	0.000979	+26.32%
クイックソート	0.008439	0.800326	+9379.41%
バケットソート	0.031297	0.035034	+11.94%
挿入ソート	2.990967	3.265788	+9.20%
バブルソート	14.921321	8.474608	−43.21%
シェイカーソート	12.589956	9.416852	−25.22%

ただし、クイックソートと挿入ソートの両方で目立ったパフォーマンスの低下が見られる。前述のように、このプログラムにおけるクイックソートの実装では、配列の最後の要素をピボットポイントとして選択している。バイトニックシーケンスは末尾に向かって減少することを考慮すると、選択されるピボットポイントはデータセットシーケンス内の最小値になる。これにより、アルゴリズムは配列のすべての要素を配列の片側にシフトする必要があり、平均計算量は $O(n \log n)$ から $O(n^2)$ に変化する。一方、挿入ソートのパフォーマンスは、データセットのシーケンスの中央点に達すると低下する。中央から最後まで、アルゴリズムは逆順に配列をソートする必要があり、これはセクション 5.1.3 で既に説明したように、パフォーマンスに大きな影響を与える。

5.1.5 ジグザクなデータ

表 20 のデータセット 2 とデータセット 7 に対してすべてのアルゴリズムの実行結果を比較すると、カウント ソート、バケット ソート、挿入ソート、シェイカー ショート、バ

ブル ソートの間でソート パフォーマンスに大きな変化がないことがわかる。これらの結果のそれぞれで、デルタ変化は 10% 未満であり、これはまだ誤差範囲内であり、実行時のシステムの状態とデータ ノイズに起因すると考えられる。ただし、クイック ソートの結果比較では、わずかなパフォーマンスの低下が見られる。小さな値の部分と大きな値の部分とがランダムに混在するため、多数の値をピボット ポイントの左側または右側にシフトする必要があり、パフォーマンスがわずかに低下する可能性がある。

表 20: ソート時間の比較と変化率（単位：秒, 変化率は data2 から data7 への%増減）

ソート手法	data2.dat	data7.dat	変化率 (%)
カウントソート	0.000775	0.000835	+7.74%
クイックソート	0.008439	0.009582	+13.56%
バケットソート	0.031297	0.033965	+8.52%
挿入ソート	2.990967	3.168507	+5.94%
シェイカーソート	12.589956	12.092268	−3.95%
バブルソート	14.921321	14.169538	−5.04%

5.2 データ数の影響

表 13 に示すランダム構造を持つデータセット 1 に対して、複数のデータ長制限を適用した実行結果に基づくと、各実行間でソート速度に明確な違いが見られる。データ量が少ないほど、ソートアルゴリズムはデータをより速く処理できる。実装されたバブルソートの計算量は表**??**に基づくと $O(n^2)$m であることを考慮すると、パフォーマンスは超線形に変化し、期待される計算量 n^2 と一致する。

表 21: バブルソートのデータ長による時間・メモリの変化率

データ長	時間（秒）	メモリ（KB）	増加率（時間）
10	0.000001	0.04	—
100	0.000024	0.40	+2300%
1000	0.001039	4.00	+4229%
10000	0.097349	40.00	+9269%
100000	14.997694	400.00	+15200%

5.3 バケットソートのバケット数の影響

表 22 によると、各反復処理でバケット数を 10 倍に増やしていくと、ソート性能が明らかに向上することがわかる。しかし、ある時点では、バケット数がデータ数と同じ 100000 に達すると、実際には性能が向上しず、むしろ大幅に低下する。ソート速度が低下するだけでなく、各反復処理で使用されるメモリの平均量も増加し続け、システムリソースを大量に浪費する。このため、対象データに最適なバケット量をテストすることが重要である。データサイズが動的に変化する場合は、アルゴリズムの性能を最大限に引き出すためにバケットサイズも調整する必要がある。

表 22: バケット数ごとのバケットソートの実行時間とメモリ使用量（データ長 100000）

バケット数	データ長	時間（秒）	メモリ（KB）	前回比（時間）
10	100000	0.295539	809.76	—
100	100000	0.031729	913.60	−89.26%
1000	100000	0.004825	1039.80	−84.79%
10000	100000	0.002712	1186.28	−43.80%
100000	100000	0.006176	6000.04	+127.68%

6　まとめ

ソートするデータに最適なソートアルゴリズムを選択することは、試行錯誤とデータの分析という面倒な作業である。すべてのテスト結果に基づくと、一般的に、カウントソートは他のすべてのソートアルゴリズムよりも一貫して優れている。ただし、カウントソートでソートできるデータの種類は非常に限られているため、実際の使用には実用的ではない。より汎用的なアルゴリズムに切り替えると、クイックソートは、データに既に逆順にソートされている小さな部分または大きな部分が含まれていない限り、他のアルゴリズムよりも優れたパフォーマンスを発揮する傾向がある。逆順にソートされている場合、クイックソートのパフォーマンスは大幅に低下する。有望な結果を示す別のソートアルゴリズムはバケットソートである。ただし、バケットソートは、他のソート方法と比較して、一貫して最も多くのメモリを使用する。それだけでなく、データのソートに使用する最適なバケット量を見つけるには、多少の試行錯誤と、最大限のパフォーマンスを維持するための微調整が必要である。計算量が $O(n^2)$ のアルゴリズムの中で、実世界に近いデータで一貫して優れたパフォーマンスを発揮するのは挿入ソートである。挿入ソートは最高のパフォーマンスを発揮するわけではありませんが、複数の種類のデータセット構造（逆構造、ランダム構造、ジグザグ構造など）にわたって一貫したパフォーマンスを維持しながら、メモリ使用量のバランスが最も優れている。

7　発表について

クリス 残念ながら、プレゼンテーションの主旨を的確に捉えることができず、致命的なミスを犯してしまいました。先生が授業で説明してくれたように、プレゼンテーションの目的は研究や実験の結果を発表することであり、そのため、結果を裏付けるために多くのデータが必要です。しかし、私はソートアルゴリズムそのものを「教える」べきだと誤解していました。これは大きな間違いで、指示を誤解しないように日本語のスキルをさらに磨いていきたいと思います。また、できるだけ多くのデータを短時間のプレゼンテーションに詰め込むのは、私にとって大きな課題です。そのため、プレゼンテーションの構成スキルを大幅に向上させ、主旨に関係のない不要な情報を排除する必要があります。一方で、わかりやすい資料を作成するスキルは向上したと感じています。スライド番号が小さく読みにくいことを除けば、図や文字は十分な大きさになっています。

参考文献

[1] Timo Bingmann, Jasper Marianczuk, and Peter Sanders. Engineering faster sorters for small sets of items. *Software: Practice and Experience*, 2020. Preprint available at arXiv.

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

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

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

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

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