データ構造とアルゴリズム レポート課題

—ソーティングアルゴリズムに関する検証と考察—

34719034 水野翔真

2023 年 6 月 20 日

目次

第 1 章

はじめに

1.1 レポート概要

本レポートは、2023 年度 前期 データ構造とアルゴリズムにて出された課題について纏めたレポートである。本レポートは、はじめに, 実験, 追加実験, まとめの 4 部構成である。

1.2 実行環境

円滑なデータの習得とレポート作成のために、本課題は CSE 上ではなく著者の所有するローカルマシンで遂行された。本課題は特に明記のない限り以下の環境で実行されることを予め断っておく。
MacBook Pro 14(2021 モデル)
SoC：M1pro(CPU 8 コア 8 スレッド (P core：3.2Ghz, E core：2.06Ghz), RAM：LPDDR5-6400 16GB)
java11.0.17

第 2 章

実験

2.1 実験概要

以下では課題の内容について説明する。

2.1.1 実装対象

著者は学籍番号が $34719034 \equiv 2\,(mod 4)$ であるので、バブルソート, クイックソート, 基数ソートについてプログラムの実装した。

2.1.2 実験対象

CSE 上の、~ebn02865/lecture/data-algo/ にある data_80MB.zip を展開した中にある以下の 50 個のデータについて実験対象のデータとする

- 10000_0
- 10000_1
- 10000_2
- 10000_3
- 10000_4
- 10000_0_almost
- 10000_1_almost
- 10000_2_almost
- 10000_3_almost
- 10000_4_almost
- 50000_0
- 50000_1
- 50000_2
- 50000_3
- 50000_4
- 50000_0_almost
- 50000_1_almost
- 50000_2_almost
- 50000_3_almost
- 50000_4_almost
- 100000_0
- 100000_1
- 100000_2
- 100000_3
- 100000_4
- 100000_0_almost
- 100000_1_almost
- 100000_2_almost
- 100000_3_almost
- 100000_4_almost
- 500000_0
- 500000_1
- 500000_2
- 500000_3
- 500000_4
- 500000_0_almost
- 500000_1_almost
- 500000_2_almost
- 500000_3_almost
- 500000_4_almost
- 1000000_0
- 1000000_1
- 1000000_2
- 1000000_3
- 1000000_4
- 1000000_0_almost
- 1000000_1_almost
- 1000000_2_almost
- 1000000_3_almost
- 1000000_4_almost

2.1.3 実験方法

以下では簡単な実験方法について示す。

1. data_80MB.zip を CSE からコピーし、展開後 data-algo ディレクトリを任意の階層に配置する
2. ソートアルゴリズムの実装とソートを実行するためのテスターを実装する
3. コンパイルしたソートプログラムとソート対象のデータを分かりやすいように、data-algo ディレクトリ内に program というディレクトリを作成しそこに配置する
4. ターミナルにてテスターを実行し、ソートにかかる時間を計測する
5. テスターが終了したら、得られた計測データをもとに考察をする

2.1.4 実装プログラム

以下に今回の実装したプログラムについて示す。
プログラムが長いため一部移して示す。

テスターのソースコード

ソースコード 1　テスター

```
1  import java.io.*;
2
3  class data_algo_tester extends Thread {
4          public static void main(String arg[]) {
5                  int sortTarget;
6                  String SortTarget = "";
7                  int csvCount = 0;
8                  for (sortTarget = 0; sortTarget < 3; sortTarget++) {
9                          int dataAmount = 0;
10                         for (int dataAmountNum = 0; dataAmountNum < 5; dataAmountNum++) {
11                                 // 読み取るデータの数を変えます
12                                 switch (dataAmount) {
13                                         case 0:
14                                                 dataAmount = 10000;
15                                                 break;
16                                         case 10000:
17                                                 dataAmount *= 5;
18                                                 break;
19                                         case 50000:
20                                                 dataAmount *= 2;
21                                                 break;
22                                         case 100000:
23                                                 dataAmount *= 5;
24                                                 break;
25                                         case 500000:
```

```
                                        dataAmount *= 2;
                                        break;
                                case 1000000:
                                        break;
                        }

                        for (int otherData = 0; otherData < 5; otherData++) {
                                for (int otherDataType = 0; otherDataType < 2;
                                otherDataType++) {
                                        String dataType = "";
                                        // 読み取るデータがnormalかalmostかを変更
                                        switch (otherDataType) {
                                                case 0:
                                                        dataType = "";
                                                        break;
                                                case 1:
                                                        dataType = "_almost";
                                                        break;
                                        }
                                        //ソースコード2に別記//
                                }
                        }
                }
        }
        }
}
```

ソースコード 2　テスター

```
// 各データを10回回します
    long data[] = new long[10];
    double mean = 0;
    double variance = 0;for(
    int loop = 0;loop<10;loop++)
    {
        int a[] = new int[dataAmount];
        int i;
        i = 0;
        try {
            FileReader fr = new FileReader(
                    dataAmount + "/" + dataAmount + "_"
                            + otherData + dataType);
            StreamTokenizer st = new StreamTokenizer(fr);
            while (st.nextToken() != StreamTokenizer.TT_EOF)
            {
                // System.out.print(st.nval + " "); //
                // 読み取ったデータを表示する。
```

```
19                  a[i] = (int) st.nval; // 読み取ったデータを配列に代入
20                  i++;
21              }
22              System.out.println("");
23              fr.close();
24          } catch (Exception e) {
25              System.out.println(e); // エラーが起きたらエラー内容を表示
26          }
27
28          System.out.println("");
29
30          // ソート開始 タイマーオン
31          long startTime = System.currentTimeMillis();
32
33          // ソートアルゴリズムの変更(一番大外のfor文で変更)
34          switch (sortTarget) {
35              case 0:
36                  QuickSort.quickSort(a, 0, dataAmount - 1);
37                  SortTarget = "QuickSort";
38                  break;
39              case 1:
40                  LSDRadixSort.RadixSort(a);
41                  SortTarget = "RadixSort";
42                  break;
43              case 2:
44                  BubbleSort.bubbleSort(a);
45                  SortTarget = "BubbleSort";
46                  break;
47              case 3:
48                  ParallelQuickSort.parallelQuickSort(a);
49                  break;
50          }
51
52          long endTime = System.currentTimeMillis();
53          // ソート終了 タイマーオフ
54
55          // ソート時間の表示
56          data[loop] = endTime - startTime;
57          System.out.println("Start:␣" + startTime + "[ms]");
58          System.out.println("End:␣" + endTime + "[ms]");
59          System.out.println(
60                  "Duration:␣" + (endTime - startTime) + "[ms]");
61          mean += endTime - startTime;
62          variance += Math.pow(data[loop], 2);
63
64          // ソート時間の記録
65          // テキストファイルに出力
66          try {
67              FileWriter fw = new FileWriter(
```

```
68                  "sort_result/result_"
69                          + SortTarget + ".txt",
70                  true);
71          PrintWriter pw = new PrintWriter(
72                  new BufferedWriter(fw));
73          if (loop == 0) {
74              pw.println(dataAmount + "_" + otherData
75                      + dataType);
76          }
77
78          pw.println(loop);
79          pw.println("Duration:␣" + (endTime - startTime)
80                  + "[ms]");
81          if (loop == 9) {
82              mean = mean / 10;
83              variance = variance / 10;
84              variance = variance - Math.pow(mean, 2);
85              pw.println("mean:" + mean);
86              pw.println("variance:" + variance);
87          }
88          pw.close();
89      } catch (IOException e) {
90          e.printStackTrace();
91      }
92      // csvへの出力
93      try {
94          FileWriter fw = new FileWriter(
95                  "sort_result/result_"
96                          + SortTarget + ".csv",
97                  true);
98          PrintWriter pw = new PrintWriter(
99                  new BufferedWriter(fw));
100         if (loop == 0) {
101             pw.print(dataAmount + "_" + otherData
102                     + dataType);
103         } else {
104             pw.print("");
105         }
106         if (csvCount == 0) {
107             pw.print(",");
108             pw.print("loop");
109             pw.print(",");
110             pw.print("Duration[ms]");
111             pw.print(",");
112             pw.print("mean[ms]");
113             pw.print(",");
114             pw.print("variance[ms^2]");
115             pw.println();
116             pw.print("");
```

```
117                 csvCount += 1;
118             }
119             pw.print(",");
120             pw.print(loop);
121             pw.print(",");
122             pw.print(endTime - startTime);
123             if (loop != 9) {
124                 pw.print(",");
125                 pw.print("");
126                 pw.print(",");
127                 pw.print("");
128             } else {
129                 pw.print(",");
130                 pw.print(mean);
131                 pw.print(",");
132                 pw.print(variance);
133             }
134             pw.println();
135             pw.close();
136         } catch (IOException e) {
137             e.printStackTrace();
138         }
139     }
```

テスターでは、対象データを読み込み、クイックソート, 基数ソート, バブルソートの順で各ソートを全対象データに対して 10 回ずつ計測し、その結果を.txt ファイルと.csv ファイルに出力する。

テスターの全自動化にあたって、FileReader オブジェクトが読み込むソート対象データの参照先を変数でコントロールすることで for 文によって容易に変更が可能になる。ソースコード 1 ではそのための準備を行なっている部分が書かれている。

このプログラムを実行すると当環境では約 22 時間ほどかかった。

それに対してソースコード 2 では、ソート対象データの読み込み, 実際にソートを走査する部分, ソート時間の出力について書かれている。

このようにテスターを作成することで、毎回ビルドをして実行する必要がなくなる。また、データについて考察をするときに.csv に出力されていると結果についてまとめやすい。.xlsx への変更も容易でグラフ化することも可能である。おまけ程度の機能として、出力の段階で 10 回ごとの平均と分散を求めるようにした。

バブルソートのソースコード

ソースコード 3　BubbleSort

```
1  public class BubbleSort {
2          static void bubbleSort(int[] data) {
3                  for (int k = 0; k < data.length - 1; k++) {
4                          for (int l = k + 1; l < data.length; l++) {
5                                  if (data[k] > data[l]) {
```

```
6                                       int tmp = data[k];
7                                       data[k] = data[l];
8                                       data[l] = tmp;
9                                   }
10                              }
11                          }
12                  }
13  }
```

バブルソートでは単純交換法を、二重 for 文によって実現した。0 番目のデータのインデックスが、0+x 番目のデータのインデックスに比べて大きければ、そこを交換する。これをデータの数-1 だけ行うと必ず最小値が 0 番目の配列に格納される。これを次は 1 番目のデータと 1+x 番目のデータについて比較していくことで順番に昇順ソートされていく。

最悪計算量は $O(n^2)$ である。

クイックソートのソースコード

ソースコード 4　QuickSort

```
1   public class QuickSort {
2       static void quickSort(int[] data, int left, int right) {
3           if (left >= right) {
4               return;
5           }
6           int x = data[(left + right) / 2];
7           int l = left;
8           int r = right;
9           int tmp;
10          while (l <= r) {
11              while (data[l] < x) {
12                  l++;
13              }
14              while (data[r] > x) {
15                  r--;
16              }
17              if (l <= r) {
18                  tmp = data[l];
19                  data[l] = data[r];
20                  data[r] = tmp;
21
22                  l++; r--;
23              }
24          }
25          quickSort(data, left, r);
26          quickSort(data, l, right);
27      }
```

```
28 }
```

クイックソートでは分割統治法により再起的に問題を解くことでソートを素早く実現している

まず基準となるピボットを設定し、その値より小さいグループと大きいグループに分けた。ピボットの値は多くの求め方があるが、時間をかけると勿体無いので一番左と一番右のインデックスの合計の平均とした。

二つのグループに分けられた問題をそれぞれのグループに対して、上記の手順を呼び出し再起的に解くことでバブルソートより効率的なソートを実現している。

最悪計算量は $O(nlogn)$ である。

基数ソートのソースコード (リスト)

ソースコード 5　LSDRadixSort

```
 1 import java.util.ArrayList;
 2 public class LSDRadixSort {
 3     public static int[] RadixSort(int[] array) {
 4         // 1.配列の最大数を取得し、桁数を取得します。
 5         int max = array[0];
 6         for (int i = 1; i < array.length; i++) {
 7             max = Math.max(max, array[i]);
 8         }
 9         int maxDigit = 0;
10         while (max != 0) {
11             max /= 10;
12             maxDigit++;
13         }
14         int mod = 10, div = 1;
15         ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>();
16         for (int i = 0; i < 10; i++)
17             bucketList.add(new ArrayList<Integer>());
18         for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10) {
19             for (int j = 0; j < array.length; j++) {
20                 int num = (array[j] % mod) / div;
21                 bucketList.get(num).add(array[j]);
22             }
23             int index = 0;
24             for (int j = 0; j < bucketList.size(); j++) {
25                 for (int k = 0; k < bucketList.get(j).size(); k++)
26                     array[index++] = bucketList.get(j).get(k);
27                 bucketList.get(j).clear();
28             }
29         }
30         return array;
31     }
32 }
```

基数ソートではデータの値に対して辞書的な並び替えをすることで計算量を $O(kn)$ と、非常に高速にソートすることを可能にする。上記のコードではそのために二重連結リスト構造を用いて実現している。

まず桁数が何桁あるのかを最初に読み込み、その桁数に応じて辞書的な並び替えを各桁ごとに行う。

次に二重連結リストを作成する。インデックスが 0 から 9 に対応するリスト構造を作成する。そこからさらに数値データを格納するためのリスト構造を各インデックスことに作成する。

数値データを順番に格納していったら、今読み取っている桁のインデックスが小さい順にデータを取り出すことで、その桁についてソートを完了させることができる。これを桁数分だけ繰り返していけばソートが完了する。

こちらのコードに関しては、参考文献 [1] を参考にした。

しかし、桁数を最初に調べるならリスト構造より配列の方が早いのではないかと考えた。以下に上記を参考に、著者が改良した二次元配列構造による基数ソートのソースコードを記載する。

基数ソートのソースコード (配列)

ソースコード 6　LSDRadixSort

```
1  public class LSDRadixSort2 {
2      public static int[] RadixSort(int[] array) {
3          // 配列がnullまたは長さ1ならそのまま返す
4          if (array == null || array.length < 2)
5              return array;
6
7          // 配列の最大数を取得し、桁数を取得
8          int max = array[0];
9          for (int i = 1; i < array.length; i++) {
10             max = Math.max(max, array[i]);
11         }
12         int maxDigit = 0;
13         while (max != 0) {
14             max /= 10;
15             maxDigit++;
16         }
17
18         int mod = 10, div = 1;
19         int[][] bucketList = new int[10][array.length];
20         int[] bucketSize = new int[10];
21
22         for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10) {
23             // バケットの初期化
24             for (int j = 0; j < 10; j++) {
25                 bucketSize[j] = 0;
26             }
27
28             // 各要素をバケットに振り分け
29             for (int j = 0; j < array.length; j++) {
```

```
30                  int num = (array[j] % mod) / div;
31                  bucketList[num][bucketSize[num]] = array[j];
32                  bucketSize[num]++;
33              }
34
35              // バケットから元の配列に戻す
36              int index = 0;
37              for (int j = 0; j < 10; j++) {
38                  for (int k = 0; k < bucketSize[j]; k++) {
39                      array[index] = bucketList[j][k];
40                      index++;
41                  }
42              }
43          }
44
45          return array;
46      }
47  }
```

基本的には連結リストで実現していたことを配列に置き直しただけなので特に仕様の変更はない。

どちらの構造を採用するかは場面にもよるが、今回は要素数が分かっているので配列の方が有利に働くと考えた。しかし、リストに比べて莫大なメモリ量がいることや、要素の追加 (例えば途中から 16 進数に対応する) などに弱いので状況を見て最良な法を選ぶと良い。

2.2 結果

各ソートごとに、各ソート対象データを 10 回ずつ実行した平均時間の表と、データの数ごとに計算した分散の表を以下に示す。

表 2.1 BubbleSort(normal) の結果の平均

	10000	50000	100000	500000	1000000
_0	44.4	2417.9	9707.8	254525.2	1006171.3
_1	56.8	2374.9	9703.6	253259.4	1005388.8
_2	55.9	2391.3	9649.8	253877	1004964.6
_3	55.2	2374.7	9662.4	254371.7	1005486.2
_4	55.4	2401.8	9673.5	253853.8	1006058.6
平均	53.54	2392.12	9679.42	253977.42	1005613.9

表 2.2　BubbleSort(almost) の結果の平均

	10000	50000	100000	500000	1000000
_0_almost	15.3	434.5	1788.4	48697.7	199411.8
_1_almost	15.4	431	1791	48953.8	199898.7
_2_almost	15.2	434.3	1840.6	49075.5	200124.6
_3_almost	15.7	436.9	1802	48538	199102.6
_4_almost	15.2	428.5	1817.1	48828.1	199189
平均	15.36	433.04	1807.82	48818.62	199545.34

表 2.3　BubbleSort の分散

データの個数 (個)	10000	50000	100000	500000	1000000
分散 ($(ms)^2$)	375.6075	959998.6836	15491392.5	10522874552	1.62439E+11

表 2.4　QuickSort(normal) の結果の平均

	10000	50000	100000	500000	1000000
_0	0.8	3.4	7.3	39.8	81.6
_1	0.7	3.4	7.3	39.6	81.2
_2	0.4	3	7	38.5	81
_3	0.5	3.7	7	39.9	80.8
_4	0.6	3.2	7.2	39.1	81.9
平均	0.6	3.34	7.16	39.38	81.3

表 2.5　QuickSort(almost) の結果の平均

	10000	50000	100000	500000	1000000
_0_almost	0.2	1.9	4	21.8	44.8
_1_almost	0.2	1.8	3.9	21.9	44.9
_2_almost	0.3	2	4	21.9	46.2
_3_almost	0.3	2	4	21.6	47.6
_4_almost	0.5	1.8	4	21.5	45.4
平均	0.3	1.9	3.98	21.74	45.78

表 2.6　QuickSort の分散

データの個数 (個)	10000	50000	100000	500000	1000000
分散 ($(ms)^2$)	0.2675	0.6756	2.6651	78.2064	321.5084

表 2.7　リスト RadixSort(normal) の結果の平均

最大桁数	6	6	7	7	8
	10000	50000	100000	500000	1000000
_0	3.4	2.7	4.3	23.5	59.8
_1	0.5	1.8	4.6	23.3	57.1
_2	0.6	1.7	4.2	22.6	59.7
_3	0.5	2	4.8	22.5	59.5
_4	0.6	2.2	4	23.6	56.7
平均	1.12	2.08	4.38	23.1	58.56

表 2.8　リスト RadixSort(almost) の結果の平均

最大桁数	6	6	7	7	8
	10000	50000	100000	500000	1000000
_0_almost	0.6	2.2	4.7	22.4	57.9
_1_almost	0.5	2	4.2	22.5	58.7
_2_almost	0.5	2	4.8	23.2	57.1
_3_almost	0.6	1.8	4.4	24.9	57.2
_4_almost	0.3	1.9	4.3	22.8	57.3
平均	0.5	1.98	4.48	23.16	57.64

表 2.9　リスト RadixSort の分散

最大桁数	6	6	7	7	8
データの個数 (個)	10000	50000	100000	500000	1000000
分散 ($(ms)^2$)	4.1139	0.3291	0.3851	7.5131	21.69

表 2.10　配列 RadixSort(normal) の結果の平均

最大桁数	6	6	7	7	8
	10000	50000	100000	500000	1000000
_0	0.6	0.6	1.4	7.8	16.5
_1	0.2	0.8	1.4	7.1	16.5
_2	0.2	0.8	1.5	7.5	16.5
_3	0.2	0.7	1.3	7.3	16.8
_4	0	0.2	1.3	7.4	16.5
平均	0.24	0.62	1.38	7.42	16.56

表 2.11　配列 RadixSort(almost) の結果の平均

最大桁数	6	6	7	7	8
	10000	50000	100000	500000	1000000
_0_almost	0	0.7	1.6	7.3	16.7
_1_almost	0	0.5	1.3	7.3	16.5
_2_almost	0.1	0.9	1.4	7.5	16.6
_3_almost	0	1	1.4	7.3	16.6
_4_almost	0	0.6	1.4	7.2	16.9
平均	0.02	0.74	1.42	7.32	16.66

表 2.12　配列 RadixSort の分散

最大桁数	6	6	7	7	8
データの個数 (個)	10000	50000	100000	500000	1000000
分散 ($(ms)^2$)	0.1731	0.2176	0.24	0.2931	0.3979

2.3　考察

2.3.1　実験全体を通して

実験は全部で 22 時間ほどかかった。これは主にバブルソートの遅さが原因である。

結果の表には現れていないが、それぞれの実行結果を見るとループのうち最初の方は実行に時間がかかっていた。これは、プロセスがリソースを割り当てるのに時間がかかるためと思われる。これについは次の章で追加実験をする。また、今回はプログラムの実行初期では速度が出ないと言うばらつき考慮して、敢えて外れ値として扱わずに処理している。

それぞれのソートの結果について分かりやすくグラフ化した。特に言及がない場合、データの個数ごとに取った平均の値をグラフに起こした。

2.3.2　バブルソート

バブルソートは他のソートに比べて圧倒的に時間がかかることが分かる。バブルソートは $O(n^2)$ であるので、結果は n^2 に比例するはずである。以下にバブルソートの平均実行時間を n^2 近似したグラフと、そのグラフを両対数で取ったものを示す。

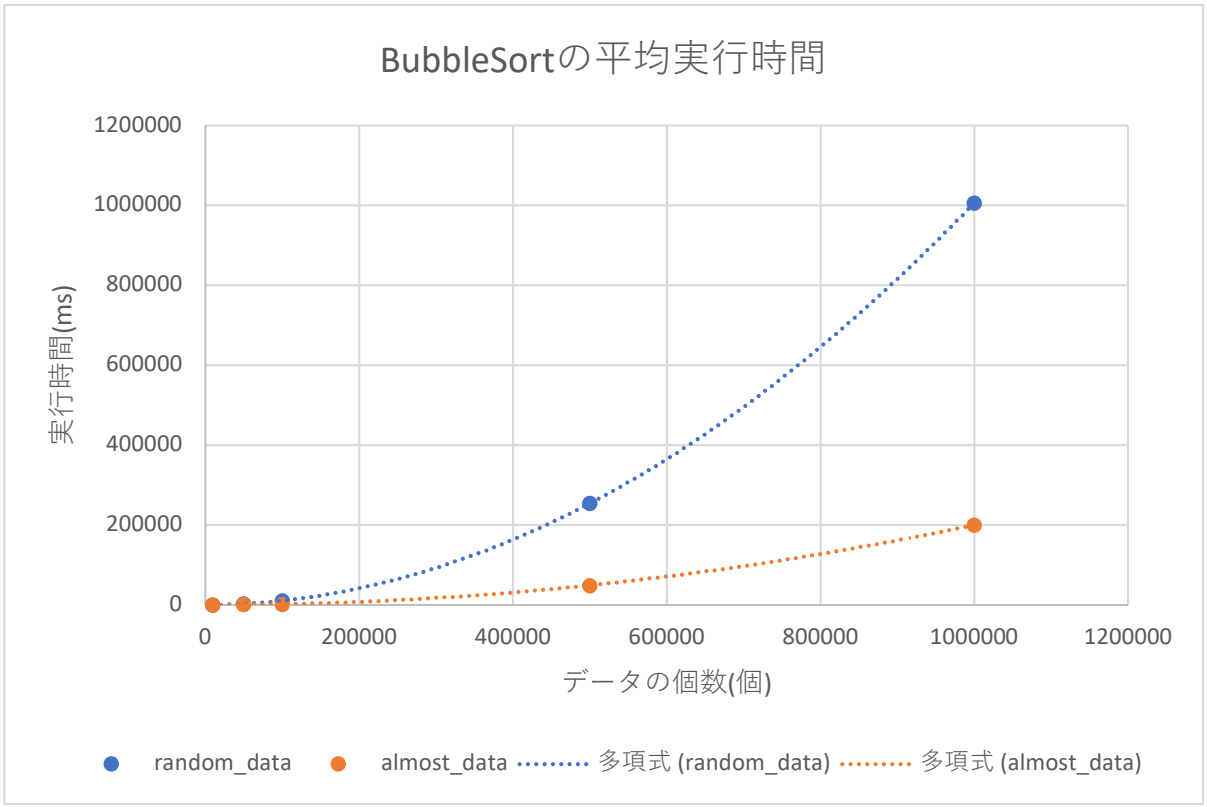

図 2.1　BubbleSort の平均実行時間

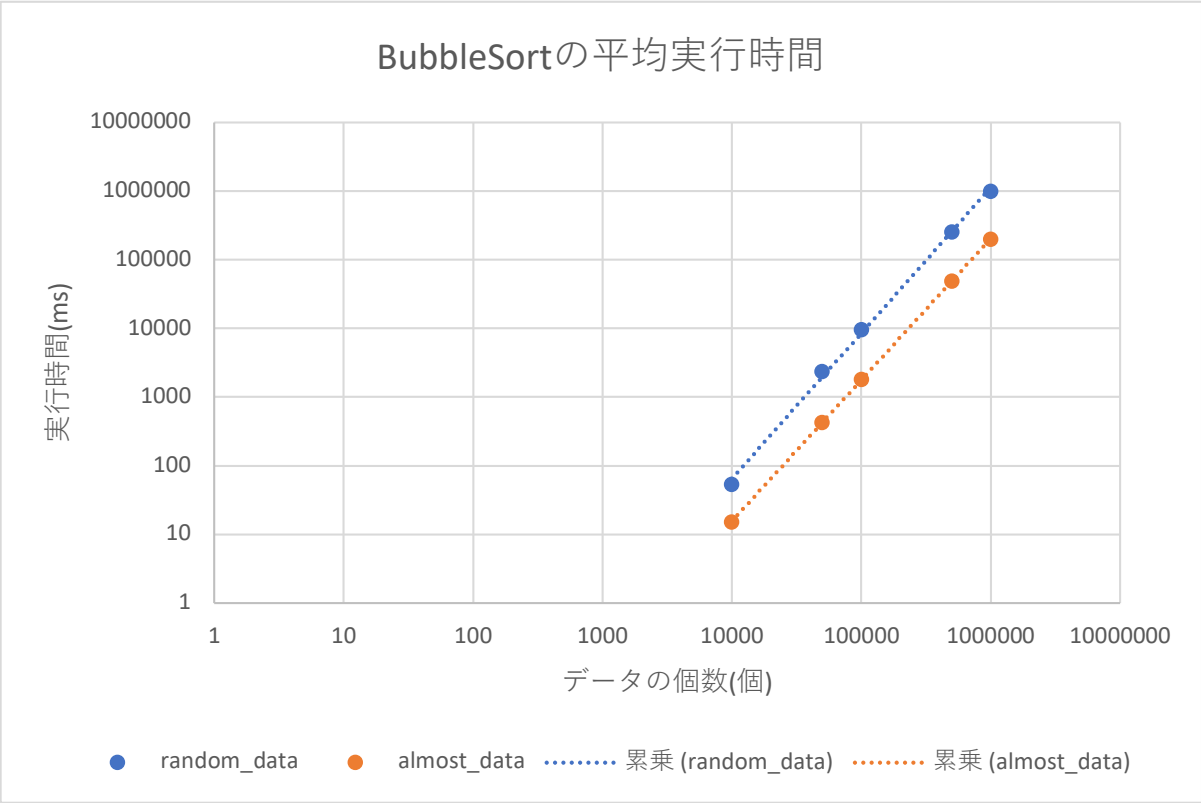

図 2.2　BubbleSort の平均実行時間 (両対数)

図 2.2 によると直線関係が見られ、データの個数が 10 倍になると実行時間は 10^2 倍になる。よって二乗に比例していると言える。これはバブルソートが $O(n^2)$ に比例することと一致し、データが random な場合、almost な場合どちらでも $O(n^2)$ であることが分かった。

また、データがほとんど整列済みであればかかる時間も大幅に削減できることが分かる。

しかしながら、データ数が増えると $O(n^2)$ はとても効率の悪いアルゴリズムである。

また、当初データの格納する配列を double 型で置いていたが、int 型にキャストすることで最大 1.5 倍高速に動作した。これは、double 型にすることで bit 演算をする回数が増えたためと考えられる。

2.3.3　クイックソート

クイックソートは比較的高速に動作するアルゴリズムである。最悪実行時間 $O(nlogn)$ で動作し、バブルソートに比べて非常に高速に動作する。

以下にクイックソートの平均実行時間のグラフについて示す。

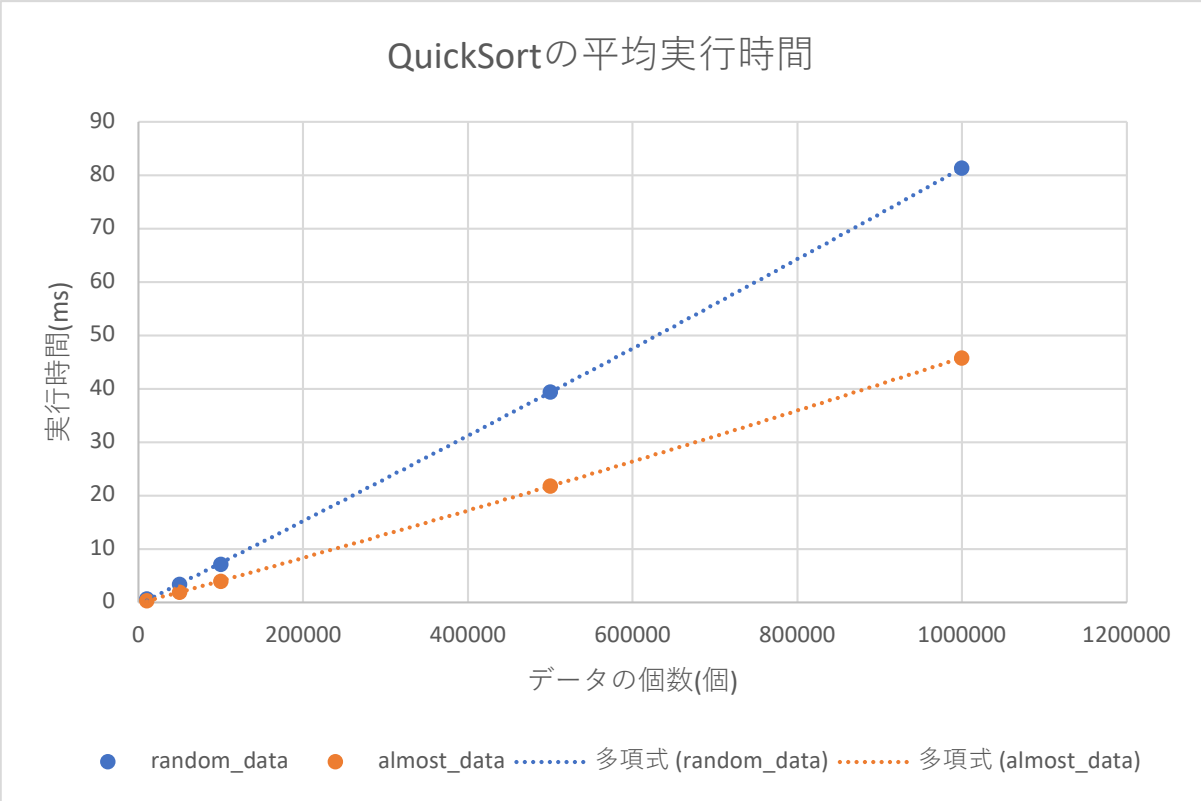

図 2.3　QuickSort の平均実行時間

図 2.3 によると、ソートにかかる時間がほとんど定数倍に比例していることがわかる。実際は多項式近次と定数近似の間で $O(nlogn)$ となるが、データ数が増えても 1,000,000 個程度ならほぼデータの数に比例して増えていると見れる。

2.3.4 基数ソート

基数ソートは桁数を予め知っていれば、辞書的な並び替えで $O(kn)$ でソートすることができ、基本的にはクイックソートの方が早い。以下に、リストによる基数ソートと、それを参考にして作成した配列による基数ソートの実行時間のグラフを示す。

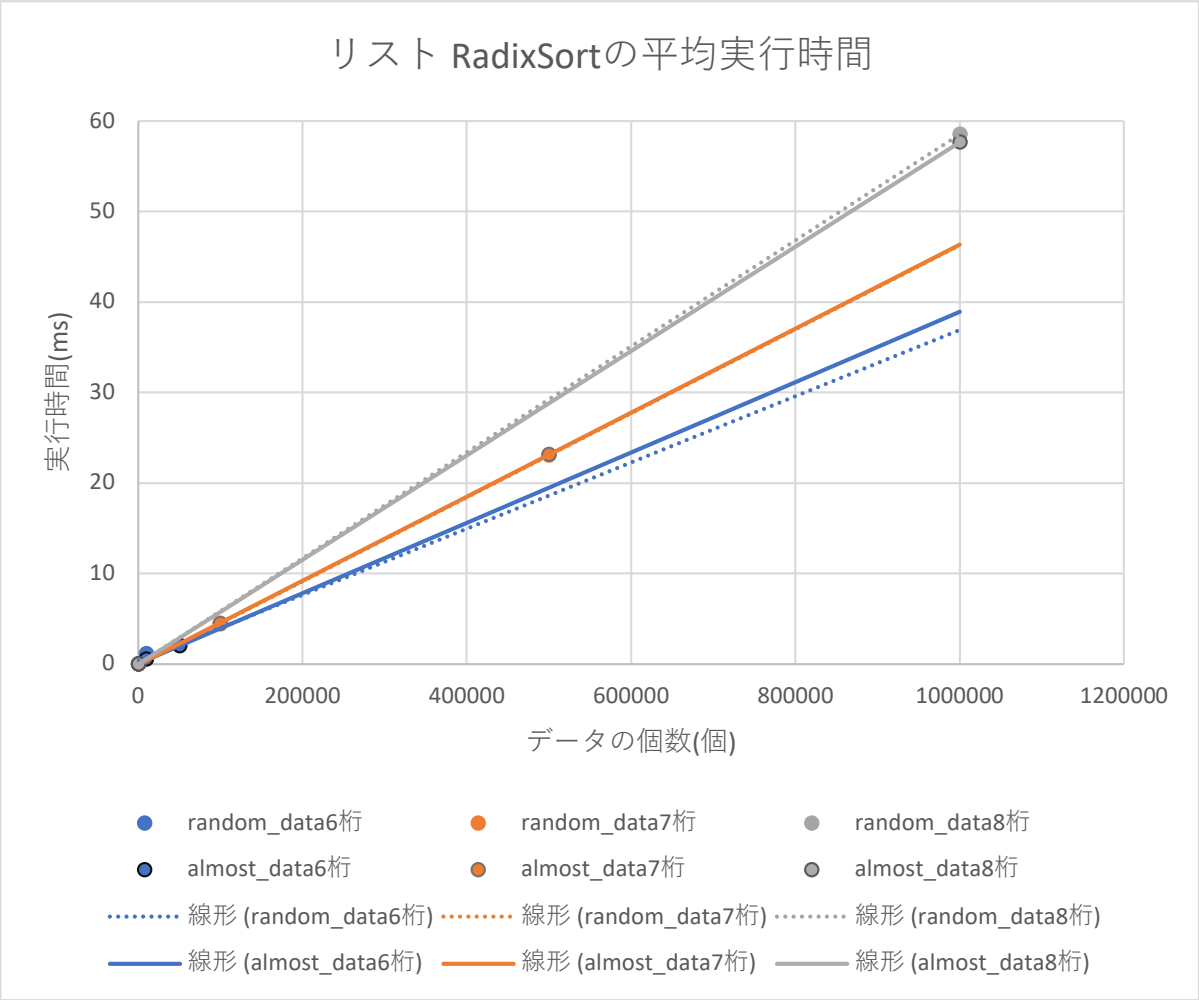

図 2.4　リスト RadixSort の平均実行時間

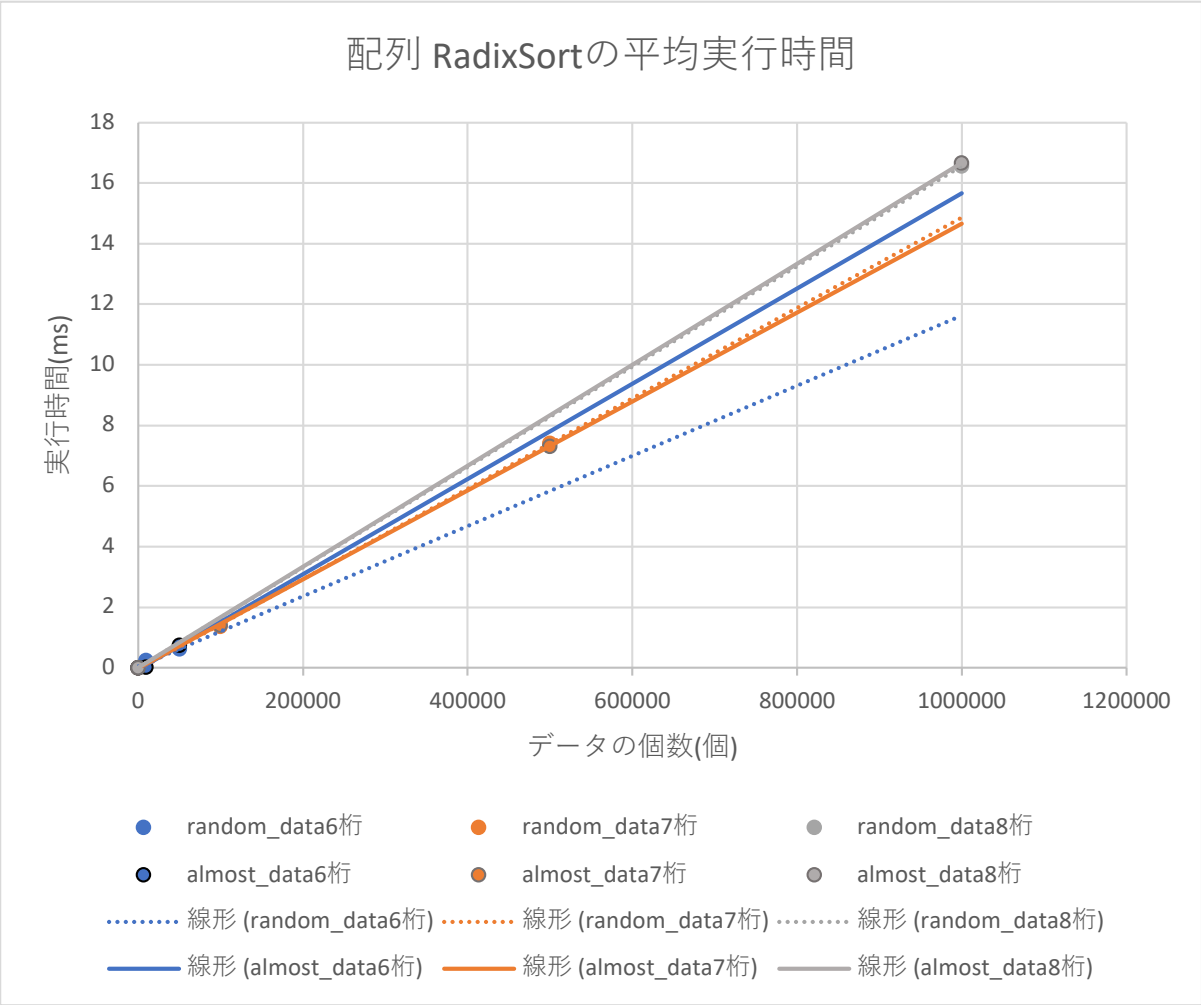

図 2.5　配列 RadixSort の平均実行時間

まず、基数ソートのプログラムでは桁数を事前に把握するためグラフは桁数ごとに色分けした。

図 3.2, 図 2.5 の両方で線形近似したグラフの傾きが、桁数ごとにほぼ 6:7:8 の比になっている。よって、$O(kn)$ であることが確かめられた。

一部、配列の random データ 6 桁で傾向予測にあっていない直線が見られるが、これは実行時間が極めて短いことと、データの個数の間隔が他と比べて狭いことで、少しの揺らぎが大きなずれに繋がったことが要因と考えられる。

次に、リストと配列を比べると、配列の方がソートで 3 倍程早いことが分かる。これは、データのアクセスにかかる時間が違うからと考えられる。しかし、リスト構造は使用するメモリ量が少ないので、比較的高速かつメモリの使用量を抑えるならリストの方が良いと考えられる。逆に、メモリが潤沢にある環境では配列を使って実現する方が良いと考えられる。

また、基数ソートはその性質上クイックソートと違い、ほとんど整列されていても関係なく常に一定のパフォーマンスで動作する。

第 3 章

追加実験

3.1　実験と考察

考察で上がった疑問を解消するために、ここではいくつかの実験を行う。

3.1.1　初期の実行時間が遅い問題

for 文で配列型の基数ソートを使って 1000000_0 を連続で 50 回ソートするプログラムを作成し実行したものと、50 回ソートを単体で実行させた二つの場合を比較する。以下にその結果のグラフを示す。

結果のグラフを確認すると、単発の時は毎回時間がかかっているのに対し、連続して実行すると 3 回目から高速に動作することが分かる。

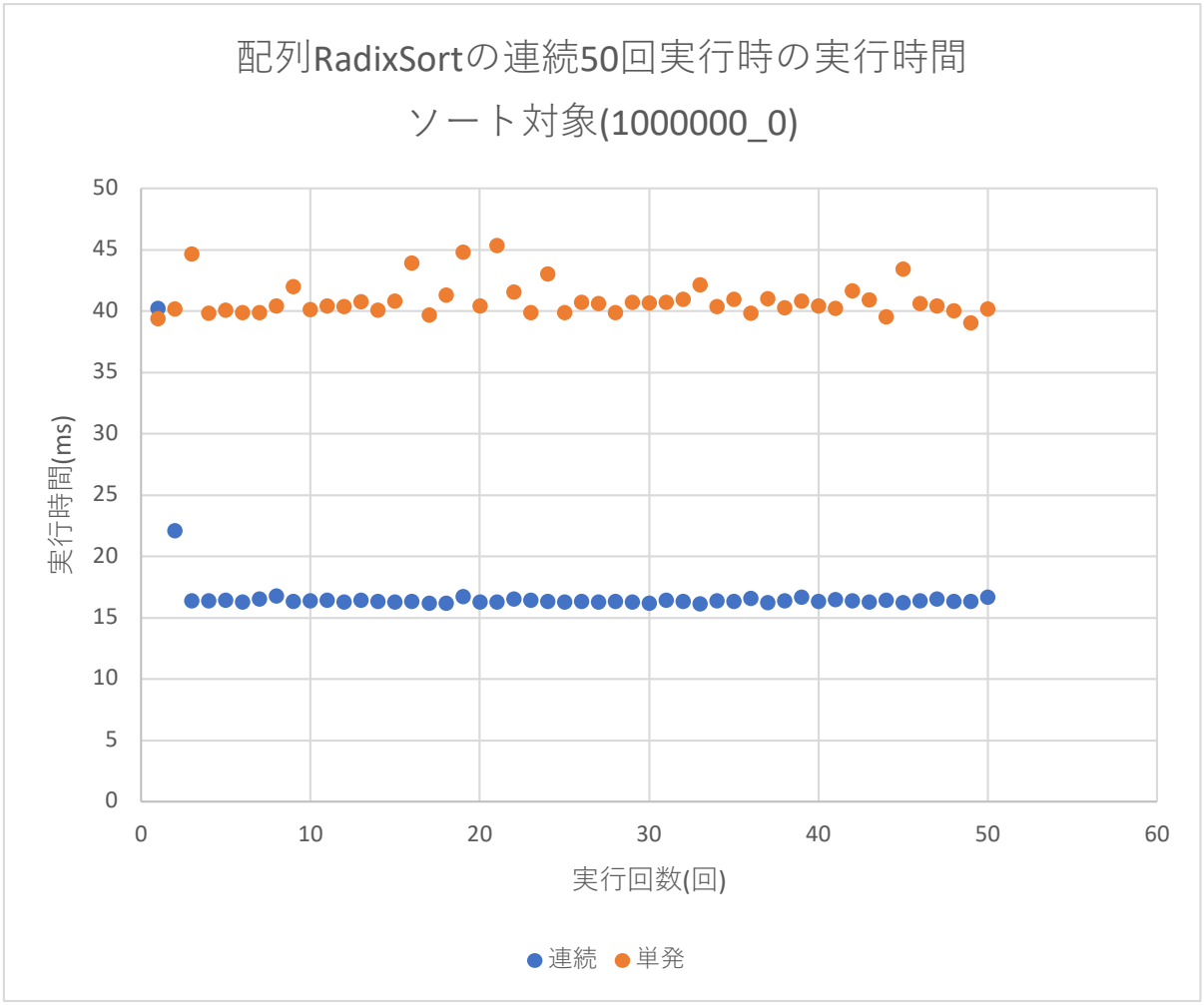

図 3.1　単発と連続したプロセスでの実行時間の比較

これが起きる理由としてすでに考察でもあげたが、java のプログラムに CPU のリソースを割り当てることに時間がかかっているためだと考えられる。その裏付けとして、プログラム実行中の CPU 使用率の変化についてアクティビティーモニターで確認する。すると当然だが CPU 使用率は時間とともに増加し、立ち上がり初期には負荷がそこまで高くないと言うことが分かった。つまり、100% の能力を出し切るのに時間がかかるのではないかと考えた。参考文献 [2] [3]

また、今回の事例にはあまり当てはまらないが、インターネットライブラリから初回のロードをする際に時間がかかると言う記事があり、ローカルでも多少なりとも同じような初回ロードに時間を取られている可能性もあるのではないかと考えた。参考文献 [4]

3.1.2 ソート済みデータに対しての動作

クイックソートはソート済みのデータに対し最良実行時間 $O(n)$ であるので、基数ソートの $O(kn)$ より高速に動作すると予測できる。よって、最大 8 桁の数値データを 1,000,000 個用意しそれを昇順に並べたデータを用意し、クイックソート、基数ソートのどちらが早いかを計測した。上記の考察をもとに for 文で 50 回連続で実行した。その結果のグラフを以下に示す。

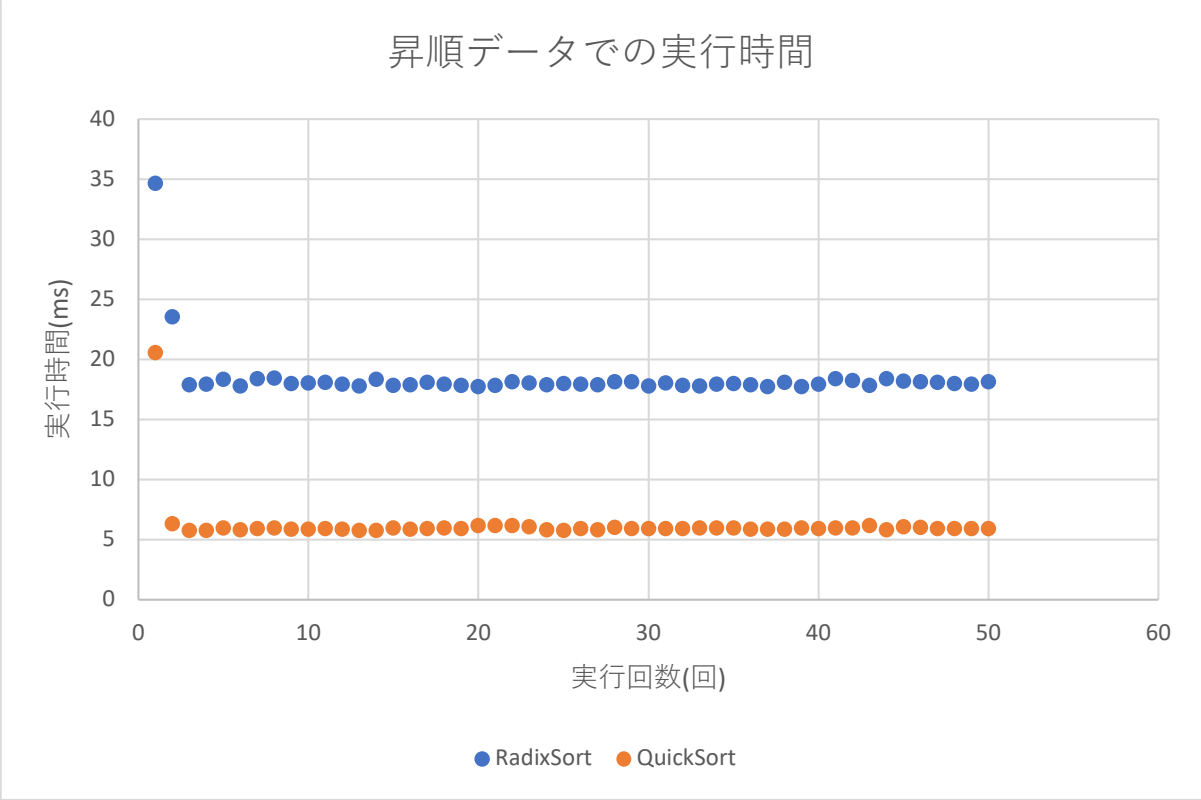

図 3.2 単発と連続したプロセスでの実行時間の比較

結果によると、やはりクイックソートの方が高速であることがわかる。つまりある程度整列されていて、適切なピボットが選択されていればクイックソートの方が早くなる可能性があると言うことが分かった。具体的にどれほどまでソートされていなかったら基数ソートより遅くなるのかは今回の実験ではわからなかったが、

安定の基数ソートに比べ、伸び代が残るクイックソートも条件次第では使えると考えた。

第 4 章

まとめ

4.1　今回のまとめ

今回の実験で、数値データのソートなら配列を使った基数ソートが最も安定して早いと分かった。
バブルソートなどは、その場ソートができ、簡単にコード書ける点では優秀だが、実際は実行時間が実用的でなく使いづらい。
クイックソートは基数ソートに比べると遅いが、データの並び方や、ピボットの取り方によっては基数ソートより早くなるので状況に合わせて使うと良い。

また、データ型は double 型より int 型の方が高速に比較でき、複数回実行するときは単発で実行するよりも連続で実行した方が高速に動作する。

4.2　感想

今回の実験では BubbleSort の実行に最も時間がかかり、これを 3 回データを全て取ったのでほぼ 3 日分はプログラムを走らせていた。これを CSE で実行するとなると恐ろしくて止まない。しかしながらこの長い作業の中で、新たな発見や疑問が生まれ自分自身に良い経験になった。

参考文献

[1] 基数ソート（Radix Sort）https://qiita.com/yp2211/items/683fa423b3f40a871064 2023/6/18

[2] 第 3 回 プロセスとスレッド：Windows OS 入門 https://atmarkit.itmedia.co.jp/ait/spv/1410/30/news150.html 2023/6/20

[3] Linux のしくみを学ぶ - プロセス管理とスケジューリング https://syuu1228.github.io/process_management_and_process_schedule/process_management_and_process_schedule.html 2023/6/20

[4] AWS Lambda の Java は遅い？ https://qiita.com/moritalous/items/632333088948aad7f8c9 2023/6/20