情報工学実験II
ソーティングプログラム

イマム カイリ ルビス

2023年7月7日

目 次

1 概要

1.1 ソートアルゴリズムとは

ソートアルゴリズムとは，データの要素をある順序に並べるアルゴリズムである．最も頻繁に使用される順序は，数値順と辞書順で，昇順または降順のどちらかである．効率的なソートは，入力データがソートされたデータであることを必要とする他のアルゴリズム（検索やマージアルゴリズムなど）の効率を最適化するために重要である．また，人間が読みやすい出力を作成したりする際にもよく使われる [1].

1.2 計算量

アルゴリズムを実行するのにかかるコンピュタの時間を**時間計算量** (*time complexity*) と呼ばれる．時間計算量は，一般的に *big O notation*(オーダー) という書き方で記す．例えば，$O(n)$, $O(n \log n)$，$O(2^n)$，など．

1.3 効率的なアルゴリズム

効率的なアルゴリズムとは，与えられた課題を最も早く解決するアルゴリズムであると考えることができる．そこで，この実験では，データを数値順に並べるいくつかのソートアルゴリズムを比較し，ある種の数値データに対して，どのアルゴリズムが最も効率的かを調べる．本実験では各プログラムの実行時間は，ソート関数が呼び出される前後のシステム時間差を clock() 関数でカウントされる．

1.4 実行環境

本実験で使用される実行環境：

- プロセッサ：AMD Ryzen 5 5600X
- メモリー：16.0 GB
- OS：Windows 11 Pro
- コンパイラ：gcc

1.5 対象データ

この実験では，すべてのアルゴリズムが C 言語で記述される．ショーティング対象データは，表 (1) に示すように，異なる特性を持っている．全ては 131072 個の要素から作られた．

表 1: ソーティング対象データ

ダーた	特徴
ダーた 1-3	乱数
ダーた 4	昇順
ダーた 5	降順
ダーた 6	バイトニック
ダーた 7	ジグザグ
でーた 8	負を含む乱数

2　バケットソート

バケットソートは，ソートされていない配列要素をバケットと呼ばれるいくつかのグループに分割するソートアルゴリズムである．各バケットは，適切なソートアルゴリズムを使用するか，同じバケットアルゴリズムを再帰的に適用することによってソートされる．

今回は，入力されたデータの値を配列の添字として，他の配列に格納することにする．そのため，データの個数を数える必要はなく，データの最小値と最大値を知ることが必要．

2.1　プログラム

以下はバケットソートのプログラムである．

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void setFileName(char *dst, int index) {
    sprintf(dst, "data%d.dat", index);
}

void setOutFileName(char *dst, int index){
    sprintf(dst, "bucket%d.dat", index);
}

void processTime(clock_t t) {
    double time = ((double)t)/CLOCKS_PER_SEC;
    printf("%.3lf ms\n", time*1000);
}

void bucketSort(FILE *in, int *p, int *sort, int offset, int n) {
    int x;

    rewind(in);

    while (fscanf(in, "%d", &x) != EOF) {
        p[x-offset]++;
    }

    int s = 0;

    for (int i = 0; i <= n; i++) {
        if (p[i] > 0)
            for (int j = 0; j < p[i]; j++) {
```

```
                sort[s] = i + offset;
                s++;
            }
    }
}

int  countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);

    return count;
}

void startSorting(int n) {
    FILE *in;
    FILE *out;


    for (int i = 1; i <= n; i++) {
        int x;
        char *filename = malloc(10);
        char *outname = malloc(10);

        int max = 0;
        int min = 0;

        printf("data %d,", i);

        setFileName(filename, i);

        if((in = fopen(filename, "r"))== NULL)
            printf("input file name wrong\n");

        while(fscanf(in, "%d", &x) != EOF) {
            max = (max < x) ? x : max;
            min = (min > x) ? x : min;
        }

        rewind(in);
        int lines = countLines(in);

        int *p = calloc((max-min+1), sizeof(int));
        int *s = malloc((lines) * sizeof(int));

        clock_t t;

        t = clock();
        bucketSort(in, p, s, min, (max-min));
        t = clock() - t;
        processTime(t);

        setOutFileName(outname, i);


        out = fopen(outname, "w");

        for (int i = 0; i < lines; i++) {
            fprintf(out, "%d\n", s[i]);
        }

```

```
100         free(p);
101         free(s);
102         free(filename);
103         free(outname);
104     }
105
106     fclose(in);
107     fclose(out);
108 }
109
110 int main() {
111     startSorting(8);
112     return 0;
113 }
```

2.2 動作

このアルゴリズムを動けるためには，$(max - min + 1)$ の大きさの配列を用意する必要がある，この理由は後述する．

バケットソートのプログラムの主な流れは，以下の通りである：

1. バケット配列に $(max - min + 1)$ の大きさを設定する．
2. 出力配列に入力データの大きさを設定する．
3. 各入力データ x はバケット配列の添字 $(x - min)$ 値を $+1$ 増やす．
4. バケット配列から出力配列に元の値変更し，格納する．
5. ソート完了．

2.3 時間計算量

バケットソートの時間的複雑さは $O(n + k)$ であり，ここで n は要素数，k はバケット配列の大きさである．

各データに対してバケットソートの実行時間は表（2）のように表す．ここで表す結果は 10 回の実行結果の平均値である．

表 2: 計算時間 (バケットソート)

データ	data1	data2	data3	data4
計算時間 (ms)	8.898	9.032	9.042	7.090

データ	data5	data6	data7	data8
計算時間 (ms)	7.236	6.796	9.122	10.517

2.4　改良案

入力データをバケット配列の添字に変換するだけでは，マイナスの値に対応できなくなる．マイナスの配列添字はそもそもないからだ．そこで，バケット配列の大きさが $(max - min + 1)$ に設定しいないといけない．入力データがすべて同じ数である場合，$x - x = 0$ となり，大きさ 0 の配列はデータを格納できないため，このような場合に $+1$ が不可欠．

表 3: 改良結果

問題	元のアルゴリズム	改良したアルゴリズム
マイナスの対応	×	○
同じ値データ	○	○
バケット配列の大きさ	大きい	小さい

2.5　考察

実行結果から，バケットソートはどのような種類のデータでもうまく動作すると結論付けられる．その差は $\pm 2ms$ 程度であり，あまりに速すぎて認識できない．

残りの問題は，小数が持っている数字をソートしようとした場合，小数は配列の添字になり得ないということだ．そこで，このアルゴリズムの実装方法について，別の考え方を考えなければならない．

3　挿入ソート

挿入ソートは，トランプを並べ替えるのと同じような仕組みである．配列は事実上，ソートされた部分とソートされていない部分に分けられる．ソートされていない部分から値が選ばれ，ソートされた部分の正しい位置に配置される．

今回は，全部の入力データをソートするでなく，入力データを 1 個ずつ取ってからソートする．

3.1　プログラム

以下は挿入ソートのプログラムである．

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int  countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);
```

```
15
16      return count;
17  }
18
19  void store_array(FILE *in, int *p) {
20      int x;
21      int i = 0;
22
23      while (fscanf(in, "%d", &x) != EOF) {
24          p[i] = x;
25          i++;
26      }
27  }
28
29  void processTime(clock_t t) {
30      double time = ((double)t)/CLOCKS_PER_SEC;
31      printf("%.3lf ms\n", time*1000);
32  }
33
34  void insertionSort(FILE *in, int lines, int *p) {
35      int key;
36      int i = 0;
37
38      while (fscanf(in, "%d", &key) != EOF) {
39          p[i] = key;
40          int j = i - 1;
41          while(j >= 0 && p[j] > key) {
42              p[j+1] = p[j];
43              j--;
44          }
45          p[j+1] = key;
46          i ++;
47      }
48  }
49
50  void setFileName(char *dst, int index) {
51      sprintf(dst, "data%d.dat", index);
52  }
53
54  void setOutFileName(char *dst, int index){
55      sprintf(dst, "insert%d.dat", index);
56  }
57
58  void startSorting(int n) {
59      FILE *in;
60      FILE *out;
61
62      for (int i = 1; i <= n; i++) {
63          char *filename = malloc(10);
64          char *outname = malloc(10);
65
66          printf("%d, ", i);
67
68          setFileName(filename, i);
69
70          if((in = fopen(filename, "r")) == NULL)
71              printf("input file name error\n");
72
73          int lines = countLines(in);
74          int *p = malloc(lines * sizeof(int));
75
76          clock_t t;
77          t = clock();
78          insertionSort(in, lines, p);
79          t = clock() - t;
80          processTime(t);
81
```

```
82          setOutFileName(outname, i);
83
84          out = fopen(outname, "w");
85
86          for (int i = 0; i < lines; i++) {
87              fprintf(out, "%d\n", p[i]);
88          }
89
90          free(p);
91          free(filename);
92          free(outname);
93      }
94
95      fclose(in);
96      fclose(out);
97  }
98
99  int main(void) {
100     startSorting(8);
101     return 0;
102 }
```

3.2 動作

挿入ソートは比較によりソーティングを行うアルゴリズムである．配列に格納されるデータは，１個前の要素から最初の要素までに比較し，適切の位置に置くこと．しかし，適切の位置に置くという関数がないため，以下の通りである：

1. 入力データ x が n 番目の要素に入力する．
2. $(n-1)$ 番目から０番目の要素までに比較する．
3. 左隣の要素が着目している要素より大きければ，着目している要素の位置に代入する．
4. 全部ソートされるまで繰り返す．
5. ソート完了．

3.3 時間計算量

挿入ソートの計算量は場合によって分れる．

- 最悪の計算量：$O(N^2)$
- 平均的な計算量：$O(N^2)$
- 最良の計算量：$O(N)$

各データに対して挿入ソートの実行時間は表（4）のように表す．ここで表す結果は 10 回の実行結果の平均値である．

表 4: 計算時間 (挿入ソート)

データ	data1	data2	data3	data4
計算時間 (ms)	6016.842	6107.600	6096.823	7.116

データ	data5	data6	data7	data8
計算時間 (ms)	12272.560	6048.500	6151.236	6063.681

3.4 考察

実行結果を見ると data4 のソートは最速で，7.116ms で全てのデータをソートすることができた．挿入ソートは，データがソートされているかどうかを検出することができると結論づけることができる．そのため，すでにソートされたデータに対しては，アルゴリズムが代入処理を行うことはない．

最も時間がかかるのは，データ 5(降順) である．これは，データを昇順にソートするために，data5 が最も処理時間がかかるからだ．代入の回数を n とデータ個数を m t すると $(n = m + (m-1) + (m-2) + \ldots + 1)$，ソートを実行すると，各反復で代入回が増加する．

4 バブルソート

バブルソートは隣接している要素を比較し，意図する順番になるまで入れ替える．水の気泡が水面に上がっていくように，配列の各要素は反復するごとに末尾に移動していきる．そのため，バブルソートと呼ばれている．

しかし，今回は要素は末尾に移動するでなく，先頭の方向に移動することにする．そうすると，ソートされた配列は配列の先頭から形成される．

4.1 プログラム

以下はバブルソートのプログラムである．

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int  countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);

    return count;
}

void swap(int *a, int *b) {
```

```
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void store_array(FILE *in, int *p) {
    int x;
    int i = 0;

    while (fscanf(in, "%d", &x) != EOF) {
        p[i] = x;
        i++;
    }
}

void processTime(clock_t t) {
    double time = ((double)t)/CLOCKS_PER_SEC;
    printf("%.3lf ms\n", time*1000);
}

void bubbleOpt(int lines, int *p) {
    int k = 0;
    clock_t t;

    t = clock();
    while(k < lines - 1) {
        int last = lines - 1;
        for(int j = lines - 1; j > k; j--) {
            if(p[j-1] > p[j]) {
                swap(&p[j-1], &p[j]);
                last = j;
            }
        }
        k = last;
    }

    t = clock() - t;
    processTime(t);
}

void bubbleSort(int lines, int *p) {
    clock_t t;

    t = clock();

    for (int i = 0; i < lines; i++) {
        for(int j = lines - 1; j > i; j--) {
            if(p[j-1] > p[j])
                swap(&p[j-1], &p[j]);
        }
    }

    t = clock() - t;
    processTime(t);
}

void setFileName(char *dst, int index) {
    sprintf(dst, "data%d.dat", index);
}

void setOutFileName(char *dst, int index){
    sprintf(dst, "bubble%d.dat", index);
}

void startSorting(int n) {
    FILE *in;
    FILE *out;
```

```
87
88      for (int i = 1; i <= n; i++) {
89          char *filename = malloc(10);
90          char *outname = malloc(10);
91
92          printf("%d  | ", i);
93
94          setFileName(filename, i);
95
96          if((in = fopen(filename, "r")) == NULL)
97              printf("input file name error\n");
98
99          int lines = countLines(in);
100
101         int *p = malloc(lines * sizeof(int));
102         store_array(in, p);
103
104         bubbleOpt(lines, p);
105         // bubbleSort(lines, p);
106
107         setOutFileName(outname, i);
108
109         out = fopen(outname, "w");
110
111         for (int i = 0; i < lines; i++) {
112             fprintf(out, "%d\n", p[i]);
113         }
114
115         free(p);
116         free(filename);
117         free(outname);
118     }
119
120     fclose(in);
121     fclose(out);
122 }
123
124 int main(void) {
125     startSorting(8);
126     return 0;
127 }
```

4.2 動作

バブルソートは隣接している要素を比較するに基づく．バブルソートのプログラムの主な流れは，以下の通りである：

1. 末尾の要素に着目する．
2. ソート済の末尾まで比較する．
3. 全部ソートされるまで繰り返す．
4. ソート完了．

4.3 時間計算量

バブルソートの計算量は $O(N^2)$ である．

この実験では，バブルソートの改良版も実装している．これは後で詳しく説明する．各データに対してバケットソートの実行時間は表（5）のように表す．ここで表す結果は 10 回の実行結果の平均値である．

表 5: 計算時間 (バブルソート)

データ	data1	data2	data3	data4
（元）計算時間 (ms)	31294.280	31165.150	31429.729	9071.419
（改良）計算時間 (ms)	31787.804	31725.492	31409.636	0.135

データ	data5	data6	data7	data8
（元）計算時間 (ms)	26548.694	18016.326	27964.376	31265.542
（改良）計算時間 (ms)	28099.591	13931.248	29337.370	31812.626

4.4 改良案

バブルソートは入力データの状態を考えず，隣接している要素を比較する．そうすると，バブルソートはすでにソートされている状態でも，比較を繰り返すことになる．**一連の比較を行うにおいて，ある時点に交換がなければ，それより先頭側はソート済みである** [2]．これをもとにして，すでにソートされたデータをチェックして，比較するのはソートされていない部分だけで行うことにする．

4.5 考察

その結果，表（5）に示すように，data4（昇順）と data6（バイトニック）では処理時間を減り，残りのデータでは遅くなっタ．これは，データがすでにソートされているかどうかを確認するために，いくつの条件があって，単純比較を行うより処理量は少し増やす．なので，大量のデータに対してこのトレードオフを現れる．

5 シェーカーソート

シェーカーソートは，2 つのバブルソートを 2 つの異なる方向に行うものである．しかし，普通のバブルソートと違って，無駄な繰り返しをしないので，大きな配列でも効率よく処理でる [3]．これはシェーカーの動きと似ているため，シェーカーソートと呼ばれている．

5.1 プログラム

以下はシェーカーソートのプログラムである．

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

int  countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);

    return count;
}

void store_array(FILE *in, int *p) {
    int x;
    int i = 0;

    while (fscanf(in, "%d", &x) != EOF) {
        p[i] = x;
        i++;
    }
}

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void processTime(clock_t t) {
    double time = ((double)t)/CLOCKS_PER_SEC;
    printf("%.3lf ms\n", time*1000);
}

void shacker(const int lines, int *p) {
    int right = lines - 1;
    int left = 0;
    bool swapped = true;

    while(left != right && swapped) {
        int last;

        swapped = false;
        for(int i = left; i < right; i++) {
            if(p[i] > p[i+1]) {
                swap(&p[i], &p[i+1]);
                last = i;
                swapped = true;
            }
        }
        right = last;

        if(!swapped) break;

        swapped = false;
        for(int j = right; j > left; j--) {
            if(p[j-1] > p[j]) {
                swap(&p[j-1], &p[j]);
                last = j;
                swapped = true;
            }
```

```
        }
        left = last;
    }
}

void setFileName(char *dst, int index) {
    sprintf(dst, "data%d.dat", index);
}

void setOutFileName(char *dst, int index){
    sprintf(dst, "shacker%d.dat", index);
}

void startSorting(int n) {
    FILE *in;
    FILE *out;

    for (int i = 1; i <= n; i++) {
        char *filename = malloc(10);
        char *outname = malloc(10);

        printf("%d  | ", i);

        setFileName(filename, i);

        if((in = fopen(filename, "r")) == NULL)
            printf("input file name error\n");

        int lines = countLines(in);
        int *p = malloc(lines * sizeof(int));
        store_array(in, p);

        clock_t t;
        t = clock();
        shacker(lines, p);
        t = clock() - t;
        processTime(t);

        setOutFileName(outname, i);

        out = fopen(outname, "w");

        for (int i = 0; i < lines; i++) {
            fprintf(out, "%d\n", p[i]);
        }

        free(p);
        free(filename);
        free(outname);
    }

    fclose(in);
    fclose(out);
}

int main(int argc, char **argv) {
    startSorting(8);
    return 0;
}
```

5.2 動作

シェーカーソートのプログラムのプログラムは，以下の通りである：

1. 末尾の要素に着目する.
2. ソート済の末尾まで比較する.
3. 全部ソートされるまで繰り返す.
4. ソート完了.

5.3 時間計算量

シェーカソートの計算量は場合によって分れる.

- 最悪の計算量：$O(N^2)$
- 平均的な計算量：$O(N^2)$
- 最良の計算量：$O(N)$

各データに対してシェーカーソートの実行時間は表（6）のように表す．ここで表す結果は 10 回の実行結果の平均値である.

表 6: 計算時間 (シェーカーソート)

データ	data1	data2	data3	data4
計算時間 (ms)	23953.497	23687.869	23934.827	0.122

データ	data5	data6	data7	data8
計算時間 (ms)	28164.822	15548.963	22338.184	23866.175

5.4 考察

表（6）に表す結果より，全体的にシェーカーソートはバブルソートより速かった．シェーカーソートは無駄な繰り返しをしないのは，data4 の結果から見れる.

6 クイックソート

クイックソートでは，ある要素をピボットとして選び，ピボットを中心に残りの配列を分割している．今回は，ピボットはソート対象データの真中の要素とする.

6.1 プログラム

以下はクイックーソートのプログラムである.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int  countLines(FILE *in) {
    char c;
    int count = 0;

    do {
        c = fgetc(in);
        if(c == '\n') count++;
    } while (c != EOF);

    rewind(in);

    return count;
}

void store_array(FILE *in, int *p) {
    int x;
    int i = 0;

    while (fscanf(in, "%d", &x) != EOF) {
        p[i] = x;
        i++;
    }
}

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void processTime(clock_t t) {
    double time = ((double)t)/CLOCKS_PER_SEC;
    printf("%.3lf ms\n", time*1000);
}

void sortPivot(int *p) {
    int size = 3;
    for (int i = 0; i < size; i++) {
        for(int j = size - 1; j > i; j--) {
            if(p[j-1] > p[j])
                swap(&p[j-1], &p[j]);
        }
    }
}

void quick(int *p, int left, int right) {
    int pl = left;
    int pr = right;
    int x = p[(pl+pr)/2];

    do {
        while (p[pl] < x) pl++;
        while (p[pr] > x) pr--;
        if(pl <= pr) {
            swap(&p[pl], &p[pr]);
            pl++;
            pr--;
        }
    } while (pl <= pr);
```

```
64
65      if (left < pr) quick(p, left, pr);
66      if (right > pl) quick(p, pl, right);
67  }
68
69  void setFileName(char *dst, int index) {
70      sprintf(dst, "data%d.dat", index);
71  }
72
73  void setOutFileName(char *dst, int index){
74      sprintf(dst, "quick%d.dat", index);
75  }
76
77  void startSorting(int n) {
78      FILE *in;
79      FILE *out;
80
81      for (int i = 1; i <= n; i++) {
82          char *filename = malloc(10);
83          char *outname = malloc(10);
84
85          printf("%d  | ", i);
86
87          setFileName(filename, i);
88
89          if((in = fopen(filename, "r")) == NULL)
90              printf("input file name error\n");
91
92          int lines = countLines(in);
93          int *p = malloc(lines * sizeof(int));
94          store_array(in, p);
95
96          clock_t t;
97          t = clock();
98
99          quick(p, 0, lines -1);
100
101         t = clock() - t;
102         processTime(t);
103
104         setOutFileName(outname, i);
105
106         out = fopen(outname, "w");
107
108         for (int i = 0; i < lines; i++) {
109             fprintf(out, "%d\n", p[i]);
110         }
111
112         free(p);
113         free(filename);
114         free(outname);
115     }
116     free(in);
117     free(out);
118 }
119
120 int main(int argc, char **argv) {
121     startSorting(8);
122
123     return 0;
124 }
```

6.2 動作

クイックソートのプログラムの流れは，以下の通りである：

1. ピボットを選ぶ.
2. 配列をピボットを中点として，ピボットより大小関係に基づいて分れる.
3. ２つ分れた配列をまたクイックソートを行う.
4. 全部ソートされるまで繰り返す.
5. ソート完了.

6.3 時間計算量

クイックソートの計算量は場合によって分れる.

- 最悪の計算量：$O(N^2)$
- 平均的な計算量：$O(NlogN)$
- 最良の計算量：$O(NlogN)$

各データに対してクイックソートの実行時間は表（7）のように表す．ここで表す結果は 10 回の実行結果の平均値である.

表 7: 計算時間 (クイックソート)

データ	data1	data2	data3	data4
計算時間 (ms)	10.394	10.143	10.166	1.561

データ	data5	data6	data7	data8
計算時間 (ms)	1.707	2410.326	10.287	10.109

6.4 考察

表（7）からみると，クイックソートは他のソートアルゴリズムに比べると，全体的に早いであることがわかる．しかし，ピボットが中心の要素にすることで，バイトニックデータが他より時間がかかる.

7 バイトニックソート

ビットニックソートは，比較ベースのソートアルゴリズムで，並列実行が可能である．乱数列を単調に増加し，減少するビットニック配列に変換することに重点を置いている．バイトニックソートは並列で動作させることができるため，GPU を使用した方が効率が良いと言われてる.

ビットニックソートは，入力データが 2^n 個のデータである場合にのみ動ける（n は実数）.

7.1 プログラム

以下はバイトニックソートのプログラムである.

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  void swap(int *a, int *b) {
7      int tmp = *a;
8      *a = *b;
9      *b = tmp;
10 }
11
12 void compSwap(int a[], int i, int j, int dir)
13 {
14   if (dir==(a[i]>a[j]))
15     swap(&a[i],&a[j]);
16 }
17
18 void bitonicMerge(int a[], int low, int count, int dir)
19 {
20   if (count>1)
21   {
22     int k = count/2;
23     for (int i=low; i<low+k; i++) compSwap(a, i, i+k, dir);
24     bitonicMerge(a, low, k, dir);
25     bitonicMerge(a, low+k, k, dir);
26   }
27 }
28
29
30 void bitonicSort(int a[],int low, int count, int dir)
31 {
32   if (count>1)
33   {
34     int k = count/2;
35
36     bitonicSort(a, low, k, 1);
37     bitonicSort(a, low+k, k, 0);
38
39     bitonicMerge(a,low, count, dir);
40   }
41 }
42
43 void sort(int a[], int N, int up)
44 {
45   bitonicSort(a,0, N, up);
46 }
47
48 int  countLines(FILE *in) {
49     char c;
50     int count = 0;
51
52     do {
53         c = fgetc(in);
54         if(c == '\n') count++;
55     } while (c != EOF);
56
57     rewind(in);
58
59     return count;
60 }
61
62 void store_array(FILE *in, int *p) {
63     int x;
```

```
64      int i = 0;
65
66      while (fscanf(in, "%d", &x) != EOF) {
67          p[i] = x;
68          i++;
69      }
70  }
71
72  void processTime(clock_t t) {
73      double time = ((double)t)/CLOCKS_PER_SEC;
74      printf("%.3lf ms\n", time*1000);
75  }
76
77  void setFileName(char *dst, int index) {
78      sprintf(dst, "data%d.dat", index);
79  }
80
81  void setOutFileName(char *dst, int index){
82      sprintf(dst, "bitonic%d.dat", index);
83  }
84
85  int main() {
86      FILE *in;
87      FILE *out;
88
89      for (int i = 1; i <= 8; i++) {
90          char *filename = malloc(10);
91          char *outname = malloc(10);
92
93          printf("%d  | ", i);
94
95          setFileName(filename, i);
96
97          if((in = fopen(filename, "r")) == NULL)
98              printf("input file name error\n");
99
100         int lines = countLines(in);
101
102         int *p = malloc(lines * sizeof(int));
103         store_array(in, p);
104
105         int a[]= {3, 7, 4, 8, 6, 2, 1, 5};
106         int N = sizeof(a)/sizeof(a[0]);
107
108         clock_t t;
109
110         t = clock();
111         sort(p, lines, 1);
112         t = clock() - t;
113         processTime(t);
114
115         setOutFileName(outname, i);
116
117         out = fopen(outname, "w");
118
119         for (int i = 0; i < lines; i++) {
120             fprintf(out, "%d\n", p[i]);
121         }
122
123         free(p);
124         free(filename);
125         free(outname);
126     }
127
128   fclose(in);
129     fclose(out);
130 }
```

7.2 動作

バイトニックソートのプログラムのプログラムは，以下の通りである：

1. バイトニック配列が作成すろ．
2. バイトニック配列の対応する要素同士を比較する．
3. 配列の 2 番目の要素をスワップする．
4. 隣接する要素を入れ替える．

7.3 時間計算量

バイトニックソートの計算量は $O(log_2 N)$ である．

各データに対してシェーカーソートの実行時間は表（8）のように表す．ここで表す結果は 10 回の実行結果の平均値である．

表 8: 計算時間 (バイトニックソート)

データ	data1	data2	data3	data4
計算時間 (ms)	57.945	60.039	60.039	47.086

データ	data5	data6	data7	data8
計算時間 (ms)	48.611	49.806	58.180	58.979

7.4 考察

表（8）から，バブルソートのような他のアルゴリズムよりも，結果はまだ速い．バイトニックソートが最速のアルゴリズムではないが，結果はどのタイプのデータに対してもほぼ同様である．つまり，このアルゴリズムはどのようなタイプのデータに対しても非常に有効であるということである．

8 結論

本研究はいくつのデータ類に対して，どのアルゴリズムが最も効率的かを調べる．以下はこれまでやったソーティングアルゴリズムの処理時間を表（9）でまとめて表す．以下の数字は全ての単位は ms である．

表 9: 実験結果

データ類	バケット	挿入	バブル	シェーカー	クイック	バイトニック
乱数 1	8.898	6016.842	31787.804	23953.497	10.394	57.945
乱数 2	9.032	6107.600	31725.4921	23687.869	10.143	60.039
乱数 3	9.042	6096.823	31409.636	23934.827	10.166	59.707
昇順	7.090	7.116	0.135	0.122	1.561	47.086
降順	7.236	12272.560	28099.591	28164.822	1.707	48.611
バイトニック	6.796	6048.500	13931.248	15548.963	2410.326	49.806
ジグザグ	9.122	6151.236	29337.370	22338.184	10.287	58.180
負を含む乱数	10.517	6063.681	31812.626	23866.175	10.109	58.979

各データに最適なアルゴリズムは，以下のように結論づけられる：

乱数　　：バケットソート
負を含む乱数　：クイックソート
昇順　　：シェーカーソート
降順　　：クイックソート
バイトニック　：バケットソート
ジグザグ　　：バケットソート

バケットソートが最も高速に乱数をソートできるアルゴリズムであることは間違いないようである．しかし，乱数に負の値が含まれる場合は，クイックソートの方が早い．これは，用意するバケット配列のサイズが大きくなるためである．

しかし，これはこの実験が少数を含む値のない実数しか扱っていないために起こることである．小数の表現も考慮すると，バケットソートよりもクイックソートの方が高速になると思う．

昇順の場合は，シャッカーソートが最速となる．これは，1 回のループでどれだけの交換が行われるかを考慮しているためである．何も交換がなければ，そのデータはすでにソートされていることになる．そのため，シャッカーソートは昇順のデータを一度だけチェックすることだけで，ソート完了になる．

バブルソートも同じ考えで実装しているので，同じことが起こるはず．そのため昇順のデータにおいて，バブルソートとシェーカーソートの差は非常に小さくなった．

降順の場合は，前から推測できるように，クイック ソートが最も高速なアルゴリズムである．

バイトニックソートは並列で動作させることができるため，GPU を使用した方が効率が良いと言わるが，結果から見るとこのアルゴリズムは CPU で実行されても，遅いとも言わない．

9　発表感想

全体的に良い発表だった．しかし，特定のデータ類に対してどのアルゴリズムが最も適しているかという結論を出そうとする人はいなかった．次回の発表では，発表者が結論も含めてプレゼンしてくれることを期待する．

質問者については，良い質問をしようとしているのはわかるが，残りの時間についても考えてほしい．発表者は時間がオーバーすることはなかったが，質問だけは時間がかかってしまう．だから，次回はもっと簡潔に質問をするようにしてほしい．

参考文献

[1] https://www.wikipedia.org/wiki/Sorting_algorithm/（参照 2023-06-06）

[2] 柴田望洋，C 言語で学ぶアルゴリズムとでーた構造，SB Creative，2017.

[3] https://www.geeksforgeeks.org/cocktail-sort/（参照 2023-06-08）