プログラミングＡレポート（課題１）

担当教員：矢内 直人 教員
提出者：大上 由翔
学籍番号：09B20015

提出年月日：2020 年 7 月 13 日

1　課題内容

挿入ソートとバブルソートを行うプログラムを作成せよ。また、作成したプログラムに様々な配列データを与え、ソートに要する実行時間を測定せよ。

2　アルゴリズム

プログラムの全体像は、図 1 に示す。まず、本プログラムでは独自ヘッダファイルを用いているものの、main 関数のみから構成され、プログラム内で新たに関数を設定していない。冒頭部分で、使用する変数の定義や変数の初期化、独自ヘッダファイル付随の関数（乱数生成）の引数設定を行っている。次に、seed 値 38 のランダムに用意された 10000 個の値を生成している。その後、ソートに要する時間を測定するための関数を用意し、ソート処理部分（図 1 の赤枠）を挟んで定義されている。赤枠の部分がこれ以降に述べる「挿入ソート」および「バブルソート」の処理が行われる部分である。ランダムに与えられた数値を赤枠部分で昇順に並びなおしたのちに、ループ処理でその結果をすべて表示している。さらに、測定していた時間を小数点以下まで表示するようにプログラムした。最後に、生成した 10000 個の乱数をメモリから削除するような関数（独自ヘッダファイルより）を置いて、プログラムは完結する。

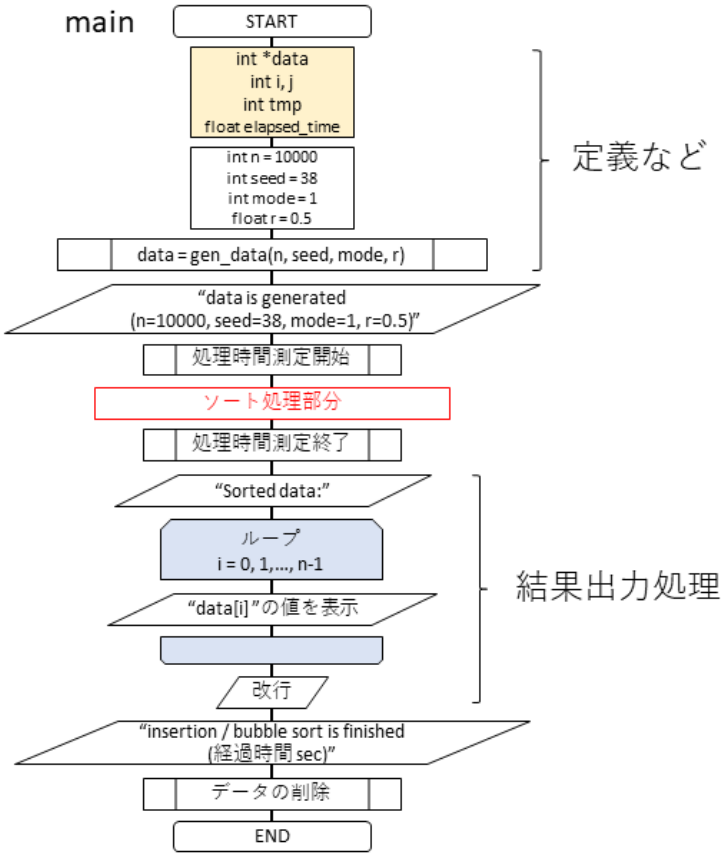

図 1: 本プログラム全体のフローチャート

以下、図 1 の赤枠部分について解説する。

n 個の要素からなる配列 data[0 ⋯ $n-1$] を昇順にソートすることを考える。配列 data の各要素には初期値として乱数を用いてランダムに数字を与えているものとする。

挿入ソートでは、配列のうち、対象となる要素よりも配列番号の小さな要素は昇順に整列しているものと考える。まず、配列の 1 番目は昇順になっていると考えるから（これは明らか)、配列の 2 番目と配列の 1 番目を比較し、配列番号の大きな要素が配列番号の小さな要素よりも小さい場合はこれらを交換する。次に配列の 3 番目と、昇順になっている 1・2 番目とを比較し、同様に配列番号の大きな要素が配列番号の小さな要素よりも小さい場合はこれらを交換する。以下同様にして、対象となる（軸となる）配列番号をだんだん大きくしていき、配列番号 $n-1$ まで同じことを繰り返す。その結果、昇順に並び変える処理を実現する。

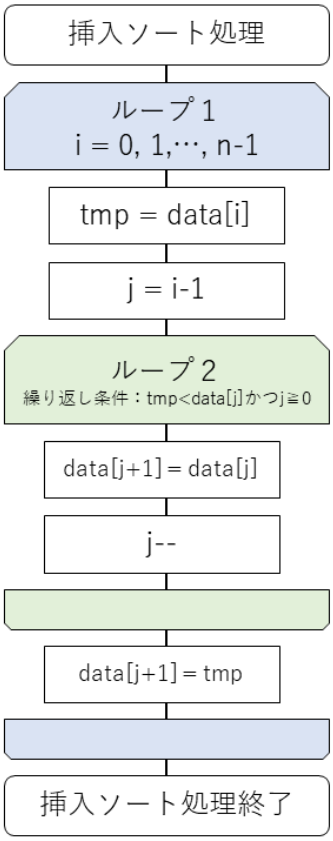

図 2: 挿入ソートのフローチャート

挿入ソートのアルゴリズムをフローチャートで表すと、図 2 のようになる。ループ 1 が、先ほど述べた、軸となる要素を data[0] から data[$n-1$] までずらす処理を実現するループ処理である。ループ 2 が、先ほど述べた、「配列番号の大きな要素が配列番号の小さな要素よりも小さい場合はこれらを交換する」ことを実現するためのループ処理である。このとき、まず変数 tmp に data[i] を格納しておく。そして、交換する配列要素のうち配列番号の大きい配列をその交換する相手方に置き換え、次に、配列番号の小さな配列に一時的に保存していた tmp の値（data[i]）を置き換えるという仕組みである。また、図 2 中、同じ色の図形が、それぞれのループ処理の始めと終わりに対応するようになっている。

バブルソートでは、まず、data[0] と data[1] を比較し、data[1] よりも data[0] のほうが大きい場合は交換し、そうでない場合は交換しない。これを data[0] と data[1]、data[1] と data[2]…data[$n-2$] と data[$n-1$] まで、順にそれぞれ交換したりしなかったりを繰り返すことで、配列の最後にすべての要素のうちの最大値がソートされる（第 1 ステップ）。次に、再び data[0] と data[1]、data[1] と data[2]…data[$n-2$] と data[$n-1$] まで[1]、順にそれぞれ交換したりしなかったりを繰り返すことで、配列の最後から 2 つ目に、先ほどの最大値を除いた時の最大値がソートされる。（第 2 ステップ）

……

以下、同様にして配列の後ろから順に大きいものをソートしていくことを n ステップ繰り返すことにより、昇順に並び変える処理を実現する。

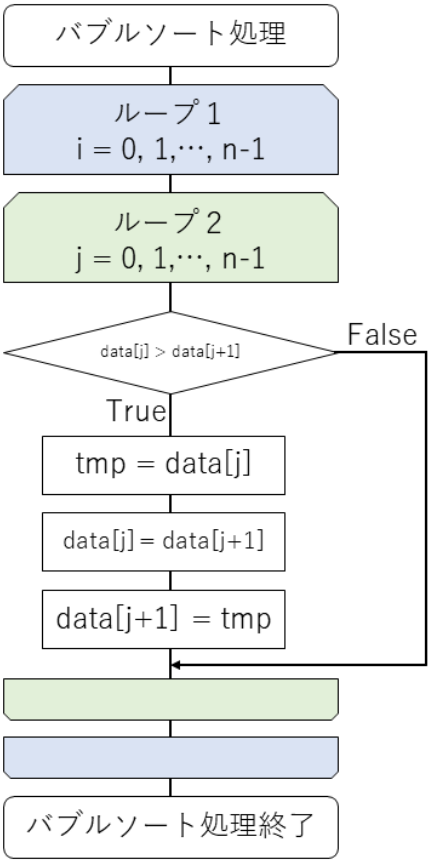

図 3: バブルソートのフローチャート

バブルソートのアルゴリズムをフローチャートで表すと、図 3 のようになる。ループ 2 は、最大値見つけるためのループ処理である。ループ 1 は、ループ 2 で見つけた最大値以外の最大値を n 回見つけるためのループ処理である。どんどん最大値を見つけていくことで、配列の後ろ側から降順に並べ、したがって最終的には要素が昇順に並び替わっているという仕組みである。また、要素の交換に際しては、挿入ソートと同様の仕組みで、C 言語の特性上、tmp を経由して交換を行っている。

[1] プログラム上、配列の最後まで比較するが、例えばこの時点で data[$n-1$] はすでに確定していることに留意されたい

3　プログラムの設計

本節ではプログラムの設計について述べる。

3.1　入力形式の設計

本プログラムでは配列 data の初期データ（乱数による）が入力データとなる。簡単のため、本プログラムでは、初期データはプログラム中で与える。

3.2　出力形式の設計

本プログラムでは、ソート後の配列 data の各要素を標準出力に出力し、ソート処理に要した時間も併せて表示を行う。

3.3　主部の設計

主部は以下のように動作する。まず、配列 data の初期値データを与える。その後、2 章で述べたアルゴリズムに従い、配列のソートを行う。最後に、ソート後の配列の各要素とソート処理に要した時間を出力する。

4　プログラムの説明

C 言語により挿入ソートおよびバブルソートを行うプログラムを作成した。作成した挿入ソートプログラムを付録 A.1 に、バブルソートプログラムを付録 A.2 に載せる。作成に当たってはテキスト [1] を参考にした。

4.1　変数、定数の説明

本プログラムでは、表 1 に示すような変数、定数を用いる。挿入ソート・バブルソートともに変数、定数は共通している。これはプログラムリストの 8 行目から 16 行目に相当する。以下に該当箇所を抜粋したものを示す。

```
 8      int *data;
 9      int i, j; // カウンタ
10      int tmp; // データの一時保管用
11      float elapsed_time; // 経過時間
12
13      int n = 10000; // 配列の長さ
14      int seed = 38; // 乱数のシード
15      int mode = 1; // 生成するデータのモード（1: 全てランダム、2: 部分ソート済み、3:逆順ソート済み）
16      float r = 0.5; // mode=2 の部分ソート済データ生成時のソート済みデータの割合
```

表 1: 挿入ソートプログラムおよびバブルソートプログラムで用いる変数、定数とその用途

変数名	型	用途
data	int 型の 1 次元配列	ソートの対象とする配列
i, j	int 型	ループ処理におけるカウンタ変数
tmp	int 型	変数の交換を行う際に変数を一時的に保管しておく変数
elapsed_time	float 型	ソート処理に要する時間を保管する変数
n	int 型	配列の大きさを表す定数
seed	int 型	乱数を生成する際のシード値を表す定数
mode	int 型	生成するデータのモードを表す定数
r	float 型	mode=2 のソート済みデータの割合を表す定数

4.2 主部の説明

主部では主に、2 章で述べたアルゴリズムに基づくソートを行う。

4.2.1 挿入ソート

これはプログラムリストの 25 行目から 35 行目に相当する。該当箇所を抜粋したものを以下に示す。

```
25 // ここから挿入ソート
26     for(i=1; i<n; i++){
27          tmp=data[i];
28          j=i-1;
29          while((tmp<data[j])&&(j>=0)){
30              data[j+1]=data[j];
31               j--;
32           }
33         data[j+1]=tmp;
34     }
35 // ここまで挿入ソート
```

交換処理の終了後、ソート後の配列の各要素を出力する。また、ソート処理に要した時間も出力する。これはプログラムリストの 40 行目から 47 行目に相当する。該当箇所を抜粋したものを以下に示す。

```
40     // 配列後の表示
41     printf("Sorted data:\n");
42     for (i=0; i<n; i++)
43          printf("%d ", data[i]);
44     printf("\n");
45
46     // 結果を表示
47     printf("insertion sort is finished (%f sec)\n", elapsed_time);
```

4.2.2 バブルソート

これはプログラムリストの 25 行目から 35 行目に相当する。該当箇所を抜粋したものを以下に示す。

```
25 // ここからバブルソート
26     for(i=0;i<n;i++){
27          for(j=0;j<n;j++){
28               if(data[j]>data[j+1]){
29                   tmp=data[j];
30                   data[j]=data[j+1];
31                   data[j+1]=tmp;
32               }
33           }
34      }
35 // ここまでバブルソート
```

交換処理の終了後、ソート後の配列の各要素を出力する。また、ソート処理に要した時間も出力する。これはプログラムリストの 40 行目から 47 行目に相当する。該当箇所を抜粋したものを以下に示す。

```
40 // 配列後の表示
41      printf("Sorted data:\n");
42      for (i=0; i<n; i++)
43         printf("%d ", data[i]);
44      printf("\n");
45
46     // 結果を表示
47     printf("bubble sort is finished (%f sec)\n", elapsed_time);
```

5　工夫した点

プログラム利用者が、標準入力から配列の初期データを（本プログラムであれば 10000 個も）入力するのは煩雑である。そこで、プログラム中で配列の初期データを乱数により与えるように工夫した。このことにより、素早く挿入ソート・バブルソートによるソーティングアルゴリズムを体験できて便利である。また、Excel などで利用者が成績データを管理している場合、カンマ区切りファイルを作り、本プログラムの 8 行目および 13 行目から 16 行目を fopen 関数や fgets 関数を用いたプログラムに書き換えることで、昇順に並び替えるなど実用的な処理を実現することができる[2]。

6　プログラムの制限事項

本プログラムでは、配列の要素として int 型を想定しているため、実数で構成されるようなデータ系列に対しては、ソートを行うことができない。ただし、プログラム中の配列 data および変数 tmp を float 型あるいは double 型に変更することにより、実数で構成されるデータ系列を扱うことが可能となる。また、利用者自身のもつデータを処理したい場合は 5 章で述べた通りであるが、プログラムを一部書き換える必要がある。

7　実行例

プログラムの動作を確認するために、seed 値を 38 に固定し[3]、挿入ソートとバブルソートの各プログラムをコンパイルし、実行した。その結果、いずれの場合においてもソートが実現されていることを確認した。

20 個の初期データを与えてプログラムを実行した場合の実行結果の一例を下記に示す。なお、紙面の都合上、ソート結果をすべて表示することは困難であるため、ソート結果の始め 20 個、終わり 20 個を抜き出すことで、ソート結果がおおむね確認できたものとみなす。

挿入ソートの場合

```
data is genereated (n=10000, seed=38, mode=1, r=0.500000)
Sorted data:
0 0 1 2 7 15 19 21 24 26 29 36 40 50 57 57 61 72 74 75 ~~~
32671 32680 32684 32687 32688 32695 32707 32708 32710 32712
32717 32721 32731 32736 32739 32742 32752 32756 32757 32763
insertion sort is finished (0.111000 sec)
```

バブルソートの場合

[2]Excel 自体にデータのソート機能が付属しているため、本来は Excel でソートすることが想定されるが、あくまで、本プログラムを改良しても同様の処理が実現できることを述べたものである。

[3]38 という値に特に意味はなく、ランダムに選定した。

```
data is genereated (n=10000, seed=38, mode=1, r=0.500000)
Sorted data:
0 0 1 2 7 15 19 21 24 26 29 36 40 50 57 57 61 72 74 75 ~~~
32671 32680 32684 32687 32688 32695 32707 32708 32710 32712
32717 32721 32731 32736 32739 32742 32752 32756 32757 32763
bubble sort is finished (0.386000 sec)
```

最後の行が、ソート処理に要した時間を示している。実行結果に示されるように、10000 個のデータに対して、ソートが実現されていることが確認できる。

8 考察

配列の要素の数 n がソートの性能にどのような影響を与えるかを確認するため、配列の要素の数 n を 1000 から 100000 までさまざまに変え、ソート処理に要する実行時間を計測した。さらに、A.3 節記載のクイックソートも比較対象用として利用しつつ、各ソートの特徴を考察するために、乱数生成のモードを変えるなどしてソート処理に要する実行時間を計測した。また、配列の各要素の初期値としては乱数を用いた。できる限り正確性を期すため時間測定に際しては、各条件・各アルゴリズムにつき 10 回の実行を行い、グラフ上ではその平均値を採用した。計測に用いたコンピュータの諸元を表 2 に示す。なお、評価に用いたプログラムを付録 A.1 節・A.2 節・A.3 節に示す。

表 2: 性能評価に用いた環境

CPU	Intel i7 1.3 GHz
Memory	16 GB
OS	Windows 10
C コンパイラ	gcc

配列の要素数に対する挿入ソート・バブルソート・クイックソートの実行時間をそれぞれ図 4・5・6 に示す。黒い曲線が mode=1, seed=38 のときの実行時間であり、赤い点は mode=3, seed=38. n=50000 のときの実行時間を示している。

まず、2 章で述べた**挿入ソート**のアルゴリズムのうち、比較的負荷のかかる処理は、配列の中身の大小を比較する比較処理と、配列の中身を交換する交換処理であるといえる。このとき、挿入ソートの処理に要する時間は、第 1 ステップでは 1 回の比較または交換が行われ、$\cdots$、第 $n-1$ ステップでは $n-1$ 回の比較または交換が行われることを鑑みれば、比較または交換の回数は、

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \tag{1}$$

と表されるから、比較交換処理に要する時間を t_1 として、次の式のようになる。

$$(\text{挿入ソートの処理に要する全体の時間 } t) = \frac{n(n-1)}{2} t_1 \tag{2}$$

よって、挿入ソートの処理に要する全体の時間 t は最高次が 2 次であるから最悪時間計算量は $O(n^2)$ であり、n が大きいときは、その実行時間がおよそ n^2 に比例した割合で増えていく。そのため、図

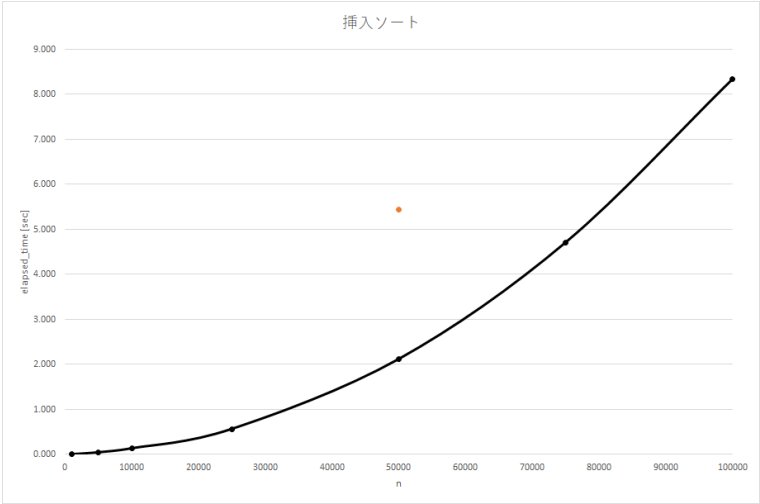

図 4: 挿入ソートの実行時間 (黒：mode=1/赤：mode=3)

4 のような結果が得られると考察できる。挿入ソートは、計算量が多く処理に時間を要するが、安定であり、実装が容易であるなどの点でよく用いられるソーティングアルゴリズムの一つである。

次に、2 章で述べた**バブルソート**のアルゴリズムのうち、先ほどと同様、比較的負荷のかかる処理は、配列の中身の大小を比較する比較処理と、配列の中身を交換する交換処理であるといえる。このとき、バブルソートの処理に要する時間は、第 1 ステップでは $n-1$ 回の比較と交換が行われ、$\cdots$、第 $n-1$ ステップでは 1 回の比較と交換が行われることを鑑みれば、比較または交換の回数は、

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \tag{3}$$

と表されるから、比較処理に要する時間を t_2、交換処理に要する時間を t_3 として、次の式のようになる。

$$(\text{バブルソートの処理に要する全体の時間 } t) = \frac{n(n-1)}{2}t_2 + \frac{n(n-1)}{2}t_3 \tag{4}$$

よって、バブルソートの処理に要する全体の時間 t は最高次が 2 次であるから、先ほどと同様に、最悪時間計算量は $O(n^2)$ であり、n が大きいときは、その実行時間がおよそ n^2 に比例した割合で増えていく。そのため、図 5 のような結果が得られると考察できる。バブルソートも、計算量が多く処理に時間を要するが、実装が容易であるなどの点でよく用いられるソーティングアルゴリズムの一つである。

最後に、挿入ソートとバブルソートとの比較参考用として用意したクイックソートであるが、今回はあくまで参考のため、アルゴリズムは一切関知しないものとした。stdio.h で作ることのできるクイックソートのアルゴリズム（大雑把に述べると、軸となる要素[4]を決めて分割して、その要素よりも小さいグループと大きいグループに整列させて、これを繰り返すことで整列させていく[5]アルゴリズム）は、プログラミング A 講義パートの課題でレポートした。また、今回使用したのは stdlib.h に定義されている quick 関数であるが、（stdio.h で初心者が作るような）単純なクイックソートをさらに改良し、非常に優れた、高速ソートを可能にしたものである。そのため、図 6 と図 4・5 とを比較して見ると一目瞭然であるが、クイックソートは n を増やしても挿入ソートやバ

[4] ピボットという。
[5] 分割統治法という。

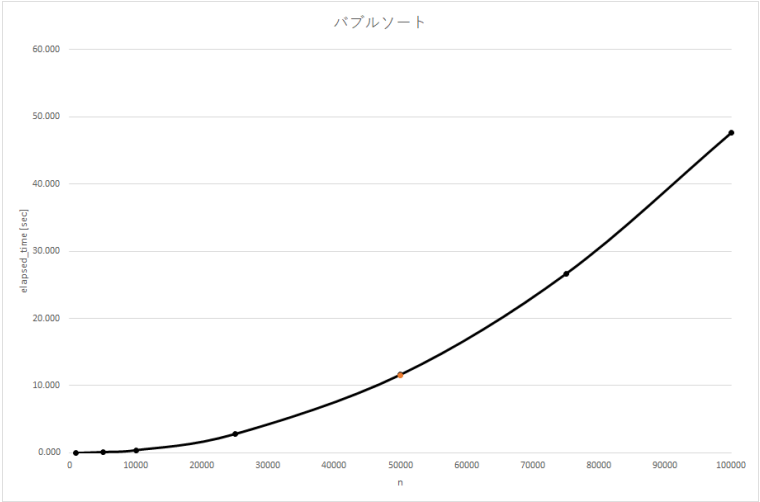

図 5: バブルソートの実行時間 (黒：mode=1/赤：mode=3)

ブルソートと比べてもほとんど一定の速度で高速に処理できていることが分かる。実際に、最悪時間計算量は $O(n^2)$ であるが、平均計算量は $O(n \log n)$ と、実用的には最速といわれている。

そして、mode=3 のときの各プログラムを実行した際、各グラフの赤点のような実行結果となった。バブルソートは mode を 1 から 3 に変えても、ほとんど処理速度に変化はなかった。これは、比較交換回数が、mode=1 のときとほとんど差がなかったためと思われる。一方、挿入ソートでは、mode を 1 から 3 に変えることで、2 倍以上の時間を要するようになった。このことは、挿入ソートが、降順を昇順に並び替えたり、昇順を降順に並び替えたりする作業は適していないことを示唆している。逆に、整列率が高ければ、効率的に処理できるといえる。

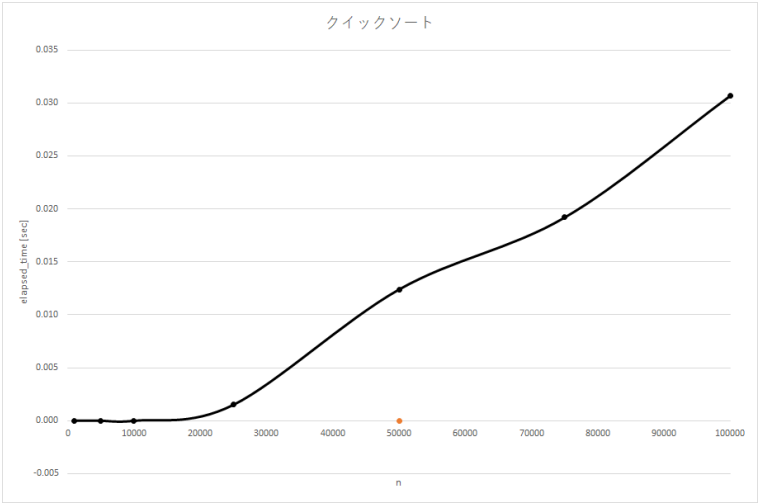

図 6: クイックソートの実行時間 (黒：mode=1/赤：mode=3)

次に、mode=2 のとき、r の値（生成される乱数のソート済み割合）を 0.1 ずつ変化させ、各アルゴリズムごとの挙動を調査した。次の図 7 を参照されたい。バブルソートは比較処理と交換処理をどちらも行わないといけない可能性があるため、最悪時間計算量は多くなるが、ソート済みの割

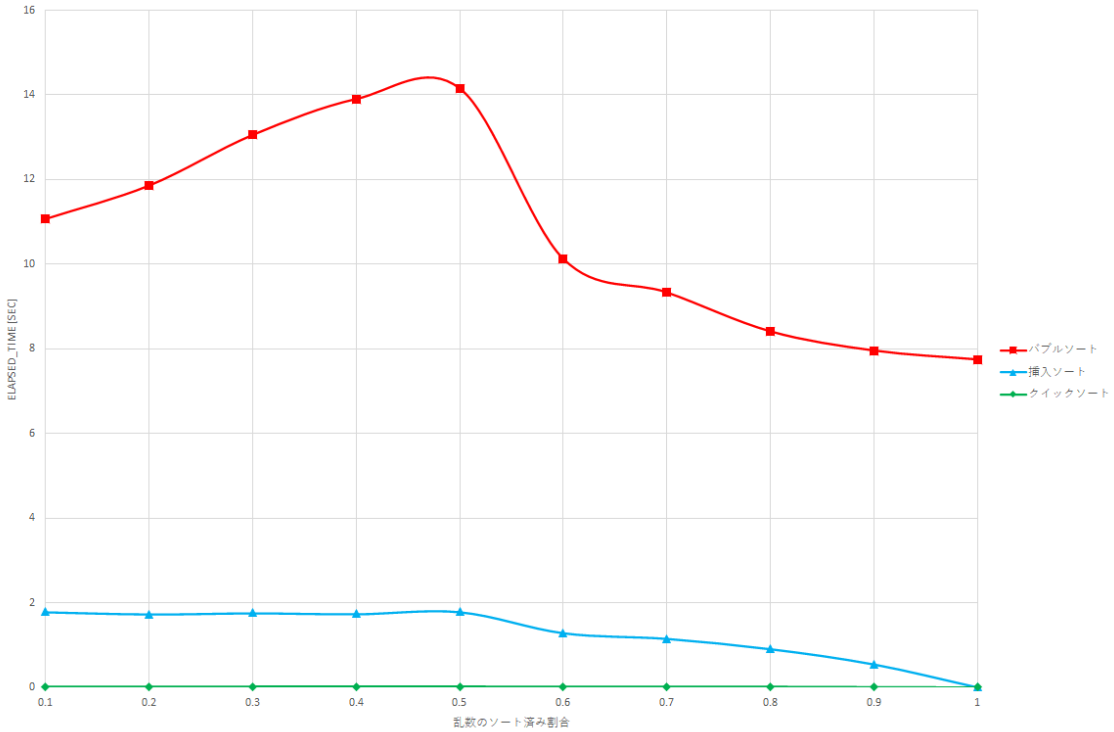

図 7: 挿入ソート・バブルソート・クイックソートの実行時間 (mode=2)

合が増えると、交換処理を行わなくて済むため、ソート処理に要する時間は短くなる。実際に図 7 が示しているように、実行時間が小さくなる傾向がみられる。ただし、使用環境の負荷状況が必ずしも一定に保つことが極めて困難[6]であることから、r=0.1 から r=0.5 にかけて異常な挙動が確認されているものと考えている。

9 感想

はじめのうちは、挿入ソートやバブルソートのプログラムがスラスラかけなかったため、とりあえずプログラムを書いてみて、適当な 3 つほどの数字を手計算でプログラム通りに動かし、きちんとソートできるかどうかを試してみた。何度も試行錯誤を繰り返していくうちに、何を書けばよいのかが分かってきた。コンパイルしてもエラーが何度も出たり、コンパイルが通ってもまったくソートされていなかったりと、かなり苦戦したが、自分なりの答えは出せたと思う。

また、前章の考察においては、挿入ソート・バブルソート・クイックソートを計 200 回実行したため、グラフを作成するのに多大な労力を要した。今後このようなことがあるとすれば、実行する際のコマンド入力を自動化したり、処理時間を自動で Excel に取り込んだりすることができるような技術を身に付けたいと思った。

[6] 意図しないところでバックグラウンド処理がなされている可能性がある。

10　作業工程

課題を提出するまでの作業内容を日付と時間とともに列挙する。

- アルゴリズム設計 (2020 年 6 月 27 日 20:00–20:05)
- プログラム設計 (2020 年 6 月 27 日 20:05–20:10)
- コーディング (2020 年 6 月 27 日 20:10–21:00)
- デバッグ (2020 年 6 月 27 日 21:00–23:30)
- レポート作成 (2020 年 6 月 29 日 17:30–18:00，2020 年 7 月 2 日 13:30–15:00，2020 年 7 月 9 日 13:30–15:00)

参考文献

[1]　浅井 宗海, 栗原 徹,“ プログラミング入門 C 言語 ”, 実教出版, 2005.

A　プログラムリスト

A.1　挿入ソート評価用プログラム

挿入ソートの評価用に作成したプログラムを下記に示す。

```
 1 //
 2 // 挿入ソート
 3 //
 4 #include <stdio.h>
 5 #include "r1_sort.h"
 6 int main()
 7 {
 8       int *data;
 9       int i, j; // カウンタ
10     int tmp; // データの一時保管用
11      float elapsed_time; // 経過時間
12
13      int n = 10000; // 配列の長さ
14      int seed = 38; // 乱数のシード
15      int mode = 1; // 生成するデータのモード (1: 全てランダム、2: 部分ソート済み、3:逆
順ソート済み)
16      float r = 0.5; // mode=2 の部分ソート済データ生成時のソート済みデータの割合
17
18     // データ生成
19      data = gen_data(n, seed, mode, r);
20      printf("data is generated (n=%d, seed=%d, mode=%d, r=%f)\n", n, seed, mode, r);
21
22     // ストップウォッチをセット
23     sw_set();
24
25     // ここから挿入ソート
26     for(i=1; i<n; i++){
27          tmp=data[i];
28          j=i-1;
29         while((tmp<data[j])&&(j>=0)){
30               data[j+1]=data[j];
31                j--;
32           }
33          data[j+1]=tmp;
34       }
35     // ここまで挿入ソート
36
37     // ストップウォッチの経過時間を取得
38      elapsed_time = sw_get();
39
40     // 配列後の表示
41      printf("Sorted data:\n");
42      for (i=0; i<n; i++)
43           printf("%d ", data[i]);
44      printf("\n");
45
46     // 結果を表示
47     printf("insertion sort is finished (%f sec)\n", elapsed_time);
48
49     // データを削除
50     free_data(data);
51 }
```

A.2 バブルソート評価用プログラム

バブルソートの評価用に作成したプログラムを下記に示す。

```
 1 //
 2 // バブルソート
 3 //
 4 #include <stdio.h>
 5 #include "r1_sort.h"
 6 int main()
 7 {
 8     int *data;
 9     int i, j; // カウンタ
10     int tmp; // データの一時保管用
11     float elapsed_time; // 経過時間
12
13     int n = 10000; // 配列の長さ
14     int seed = 38; // 乱数のシード
15     int mode = 1; // 生成するデータのモード (1: 全てランダム、2: 部分ソート済み、3:逆順
ソート済み)
16     float r = 0.5; // mode=2 の部分ソート済データ生成時のソート済みデータの割合
17
18     // データ生成
19     data = gen_data(n, seed, mode, r);
20     printf("data is generated (n=%d, seed=%d, mode=%d, r=%f)\n", n, seed, mode, r);
21
22     // ストップウォッチをセット
23     sw_set();
24
25     // ここからバブルソート
26     for(i=0;i<n;i++){
27          for(j=0;j<n;j++){
28              if(data[j]>data[j+1]){
29                    tmp=data[j];
30                    data[j]=data[j+1];
31                    data[j+1]=tmp;
32                }
33          }
34     }
35     // ここまでバブルソート
36
37     // ストップウォッチの経過時間を取得
38     elapsed_time = sw_get();
39
40     // 配列後の表示
41     printf("Sorted data:\n");
42     for (i=0; i<n; i++)
43          printf("%d ", data[i]);
44     printf("\n");
45
46     // 結果を表示
47     printf("bubble sort is finished (%f sec)\n", elapsed_time);
48
49     // データを削除
50     free_data(data);
51 }
```

A.3　クイックソート評価用プログラム

クイックソートの評価用に作成したプログラムを下記に示す。8 節において、挿入ソート・バブルソートとの比較が主たる利用目的である。

```
 1 //
 2 // クイックソート
 3 //
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include "r1_sort.h"
 7 int compare(const void *a, const void *b){
 8       return *(int*)a-*(int*)b;
 9 }
10
11 int main()
12 {
13     int *data;
14     int i; // カウンタ
15     float elapsed_time; // 経過時間
16
17     int n = 10000; // 配列の長さ
18     int seed = 38; // 乱数のシード
19     int mode = 1; // 生成するデータのモード (1: 全てランダム、2: 部分ソート済み、3:逆順
ソート済み)
20     float r = 0.5; // mode=2 の部分ソート済データ生成時のソート済みデータの割合
21
22     // データ生成
23     data = gen_data(n, seed, mode, r);
24     printf("data is generated (n=%d, seed=%d, mode=%d, r=%f)\n", n, seed, mode, r);
25
26     // ストップウォッチをセット
27     sw_set();
28
29     // ここからクイックソート
30      qsort(data,n,sizeof(int),compare);
31     // ここまでクイックソート
32
33     // ストップウォッチの経過時間を取得
34     elapsed_time = sw_get();
35
36     // 配列後の表示
37      printf("Sorted data:\n");
38      for (i=0; i<n; i++)
39          printf("%d ", data[i]);
40      printf("\n");
41
42     // 結果を表示
43      printf("quick sort is finished (%f sec)\n", elapsed_time);
44
45     // データを削除
46     free_data(data);
47 }
```