Rep. Fac. Sci., Kagoshima Univ.,
No. 53, pp. 1-15 (2020)

整列アルゴリズムの比較評価と二分探索挿入ソートの提案

Comparative Evaluation of Some Sorting Algorithms and Proposal of Binary Search Insertion Sort

新森 修一[1)*]・荒谷 遙香[1)]
Shuichi SHINMORI[1)*], Haruka ARATANI[1)]

[1)] 鹿児島大学大学院理工学研究科数理情報科学専攻
[1)] Graduate School of Science Engineering, Kagoshima University, Kagoshima 890-0065
* 責任著者 e-mail address : shinmori@sci.kagoshima-u.ac.jp

Abstract: We studied on typical sorting algorithms, compared their efficiencies, and proposed and evaluated a new sorting algorithm. In this paper, for some typical sorting algorithms, the principle and properties of each algorithm are described, and a function-type program in C language is given. Numerical experiments were used to compare and evaluate the execution times of the three $O(n^2)$ sorting algorithms (i.e., Bubble sort, Insertion sort and Selection sort) and the three $O(n\log n)$ sorting algorithms (i.e., Quicksort, Heaposrt and Mergesort). Furthermore, we proposed a new algorithm called Binary search insertion sort, compared it with the conventional Insertion sort, and confirmed that it can sort at a high speed of about 68%.

Keywords: Sorting algorithm, Insertion sort, Quicksort, Data structure, Numerical experiments.

1. はじめに

整列(sorting) とは，全順序集合の要素である複数のデータを，全順序「≦」に従って小さいデータから大きいデータへと並べ替えること（昇順という），あるいは逆に，大きいデータから小さいデータに並べ替えること（降順という）をいい，ソート，ソーティングともいう。クイックソート，バブルソート，挿入ソートなど，整列のための数多くのアルゴリズムが研究されているが，これらを総称して整列アルゴリズムという（[14], [15]）。我々の身近なところには，様々な多くのデータが存在する。これらのデータの中には，全順序≦で順序付けられるものが数多くある。例えば，整数や実数の数値データであれば大小順，日本語や英語の単語などの文字データであれば，五十音順やアルファベット順などの辞書式順序で大小関係が与えられる。データを効率良く整列させる方法はデータを探索する方法などと並んで基本的な問題であり，以前から多くの研究がなされている（例えば，[1], [2], [3], [6], [12]）。

本論文では，整列アルゴリズムの中でも代表的な 6 種類のアルゴリズムに注目し，各アルゴリズムの概要の説明とその C 言語による関数形（プログラム）やアルゴリズムの理論的な性質を示し，実際に大規模な数値データに対して，これらのプログラムを実行させる数値実験により各整列アルゴリズムの効率性の評価を行う。後半では，挿入ソートとこれを改良した二分探索挿入ソートを提案し，C 言語でのプログラム化の方法，実際のデータを用いた数値実験による効率性の評価を行う。

第 2 節では，アルゴリズムの評価方法である時間計算量と整列アルゴリズムの安定性について述べる。第 3，4 節では，代表的な 6 種類の整列アルゴリズムに注目し，これらを時間計算量により 2 つのグループに分類する。そして，各整列アルゴリズムの概要の説明，C 言語を用いてのプログラム化，数値実験による効率性の比較評価を行う。第 5，6 節では，二分探索挿入ソートを提案し，数値実験により，従来の挿入ソートとの効率性の比較評価を行う。第 7 節はまとめと今後の課題である。

2．時間計算量とソートの安定性

2．1．　アルゴリズムの時間計算量

アルゴリズムの計算量は個々の問題例によって変化するので，全体的な評価を行うには問題例の規

模に応じて計算量がどのように変化するのか，その関数形を求めることが重要である。一般に，問題例の規模はそれを入力するためのデータの長さ n（入力サイズという）で評価する（[9], [13]）。

時間計算量(time complexity)とは，アルゴリズムの基本操作は，四則演算や比較などの基本ステップに分解できるが，アルゴリズムが終了するまでの基本ステップの実行回数のことである。アルゴリズムの時間計算量は入力サイズに依存する場合が多い。また，ある入力サイズ n において，適当な条件下で最も速くアルゴリズムを実行できる場合の時間計算量を**最良時間計算量**とよび，アルゴリズムの実行に最も時間のかかる場合の時間計算量を**最悪時間計算量**とよぶ。

アルゴリズムの計算量については，時間計算量の他にアルゴリズムのデータ使用量を表す**領域計算量**(space complexity)がある。この領域計算量も値が小さいほど良いアルゴリズムと言える。しかし，実際の計算では，領域計算量に比べ時間計算量の方が重要になる場合が多いので，単に効率の良いアルゴリズムといえば，時間計算量の小さいアルゴリズムを指す場合が多い。

また，時間計算量についても，入力に対する時間計算量の期待値を考える場合がある。この場合の時間計算量を**平均時間計算量**と呼ぶ。多くの場合，平均時間計算量は最悪時間計算量に等しいことが知られている（例えば，[7], [8]）。

さて，アルゴリズムの効率性を評価する際は，入力サイズ n がある程度大きい場合を想定しているが，これらの評価に用いられるアルゴリズムの時間計算量を**漸近的な時間計算量**とよぶ。この漸近的な時間計算量は，以下に定義される**オーダ記法**を用いて表わされる（[8], [13]）。

> [**定義 1**] **（オーダ記法）**　入力サイズ n の関数として表わされる時間計算量 $T(n)$ が，ある関数 $f(n)$ に対して $O\big(f(n)\big)$ であるとは，適当な 2 つの正の定数 n_0 と c が存在し，n_0 以上のすべての n について $T(n) \le c \cdot f(n)$ が成り立つことをいう。
>
> i.e.　$T(n) = O\big(f(n)\big) \iff {}^{\exists}c > 0,\ {}^{\exists}n_0 \in \mathbb{N},\ {}^{\forall}n \ge n_0, T(n) \le c \cdot f(n)$　(1)

また，$T(n) = O\big(f(n)\big)$ である場合，“アルゴリズムの時間計算量は $O\big(f(n)\big)$ である”もしくは“アルゴリズムは $O\big(f(n)\big)$ 時間で実行できる”という。

２．２．　整列アルゴリズムと安定性

整列（あるいは**ソート** (sort)）とは，次に定義されるように “与えられたデータを決められた順番に並べ替える” という操作であり，この操作や手順を厳密に定義したものが整列（ソート）アルゴリズムである。

> [**定義 2**]　**整列**（あるいは**ソート**）とは，入力された n 個のデータ $d_0, d_1, \cdots, d_{n-1}$ が与えられたときに，そのデータを昇順，または降順に並べ替える操作のことである。

一般的に整列アルゴリズムの入力では同じ値を持ったデータが多数存在する場合が考えられる。整列アルゴリズムについては，この同じ値を持ったデータをどういう順序で並べるかという点に注意する必要がある。次の定義で示すように，同じ値のデータがソートの前後で変化しない整列アルゴリズムを安定なアルゴリズムという。

> [**定義 3**]　**安定な整列アルゴリズム**とは，以下の条件を満たすアルゴリズムのことである。
>
> (a) 与えられたデータを昇順，または降順に並べ替える。
>
> (b) 同じ値のデータは，当初の入力の順序どおりに並べ替える。

整列アルゴリズムには効率的であるが安定でないもの，効率性は高くないが安定性の条件を満たしているものなど多くの種類がある。アルゴリズムの効率性（実行速度）だけでなく，場合によっては安定性も考慮して，ソートの目的に応じた整列アルゴリズムを選択する必要がある。

３．代表的な整列アルゴリズムとその関数形

本節では，代表的な整列アルゴリズムの原理，時間計算量による評価，C 言語による関数プログラムの実装例について述べる。本論文で扱うアルゴリズムやプログラムに共通していることは，整列の対象としてのデータは整数型（非負整数値）であり，対象となる全データは配列 D に格納されているものとしている点である。また，データ数は変数 n を用いており，そのため整列アルゴリズムの対象は，D[0], D[1],･･･, D[n-1]である。これらのデータに，ある整列アルゴリズムが適用されると，配列 D のデータは昇順に並べ替えられることになる。

３．１．　$O(n^2)$ の整列アルゴリズム

$O(n^2)$ の整列アルゴリズムとして，バブルソート，挿入ソート，選択ソートの順に述べる。それぞれ整列アルゴリズムの概要と C 言語による関数形式のプログラムを示し，各整列アルゴリムの詳細について説明する。

(1) バブルソート

バブルソート (Bubble sort) とは，後方から前方へ隣同士のデータを比較・交換することで小さいデータを順に前に送っていく方法であり，これを繰り返すことにより，データ全体が昇順（あるいは降順）にソートされる。このアルゴリズムを C 言語によりプログラム化すると以下のようになる。

プログラム 1　バブルソート

```
1  // ---------- バブルソート ---------- //
2  void Bubblesort(int D[], int n){
3        int i, j;
4        for(i = 0 ; i < n-1 ; i++){
5              for(j = n-1 ; j > i ; j--){
6                    if(D[j-1] > D[j]){ swap(D[j-1], D[j]); }
7              }
8        }
9  }
```

上のバブルソートのプログラムについて説明する。for 文の入れ子構造により右端から隣同士の値を比較し左側，つまり，添字の小さい方の値が大きければ swap 関数を実行する。swap(x,y)は，変数 x と y の値を交換する関数である。これを繰り返すことで，ソートされていない範囲の最小値を左端に移動させることが出来る。これを何回も繰り返し，上のプログラムの二重の for 文が終了すると，配列 D のデータは昇順に並べ替えられたことになる。

バブルソートの時間計算量について考える。バブルソート単体の時間計算量を考えるため，プログラム 1 の for 文の動作に注目する。バブルソートの場合，データを比較し，昇順にするために交換する操作は $O(1)$ であるため，最良時間計算量と最悪時間計算量はともに 2 つの for 文に依存する。4 行目から始まる i に関する for 文は $n-1$ 回実行され，5 行目から始まる j に関する for 文は $n-i$ 回実行されるため，時間計算量は

$$T(n) = O(1) \times \sum_{k=1}^{n-1} k = O(1) \times \frac{n(n-1)}{2} = O(n^2) \qquad (2)$$

である。同様の理由で，平均時間計算量も $O(n^2)$ である。

また，プログラム 1 の 6 行目の条件文“D$[j-1] >$ D$[j]$”から分かるように，バブルソートは隣同士の値が同じ場合に入れ替えは発生しない。そのため，定義 3 の条件(b)を満たす。よって，バブルソートは安定な整列アルゴリズムであると言える。

(2) 挿入ソート

挿入ソート (Insertion sort) とは，既にソートされた部分に，新たなデータをデータの値の大きさに

応じた適切な場所を見つけてそこへデータを挿入していく方法であり，これを最後尾まで繰り返すとソートが完了する。プログラムで記述すると，以下のようになる。

プログラム 2　挿入ソート

```
1  // ------------ 挿入ソート ------------ //
2  void Insertsort(int D[], int n){
3        int i, j, x;
4        for(i = 1 ; i < n ; i++){
5              x = D[i];   j = i;
6              while ((D[j-1] > x) && (j > 0)){
7                    D[j] = D[j-1];   j = j-1;
8              }
9              D[j] = x;
10       }
11     return;
12 }
```

プログラム 2 について説明する。4 行目で $i = 1$ と初期化し，$i < n$ が成立しなくなるまで i をインクリメントしながら for 文を実行する。5 行目で新しいデータを $x = \mathrm{D}[i]$，$j = i$ とし，次の while 文に移動する。$\mathrm{D}[j-1] > x$ かつ $j > 0$ の場合，7 行目を実行して $\mathrm{D}[\mathrm{j}-1]$ の値を右にずらし，j をディクリメントさせる。それ以外の場合，つまり，挿入する値 x が比較する値より大きい場合，または $\mathrm{D}[0]$ よりも値が小さい場合に，データの格納される場所が D[j]であるので，9 行目を実行する。

挿入ソートの時間計算量について考える。プログラム 2 は for 文とそれに含まれる while 文で構成されており，入力によって時間計算量は変化する。最もアルゴリズムの動作が高速な場合は，与えられたデータが既にソートされている場合である。この場合，6 行目の $\mathrm{D}[j-1] > x$ という条件は一度も成立しない。したがって，while 文の中の処理は一度も実行されず，for 文の処理は $n-1$ 回しか実行されないことになる。for 文の中の操作は，数回の比較・代入操作のみであり，$O(1)$ となる。よって，最良時間計算量は次のようになる。

$$T(n) = (n-1) \times O(1) = O(n) \tag{3}$$

一方，アルゴリズムが最も遅い場合は，入力データが降順にソートされている場合である。この場合，挿入するデータの格納場所は常に配列の左端となり，$i = k$ の場合，while 文は k 回繰り返される。while 文の中の操作も数回の比較・代入操作のみであり，$O(1)$ となる。上の最良時間計算量で述べたように，while 文を除いた for 文の時間計算量は $O(n)$ であるので，最悪時間計算量は

$$T(n) = O(1) \times \sum_{k=1}^{n} k + O(n) = O(n^2) \tag{4}$$

である。また，証明は省略するが，平均時間計算量も $O(n^2)$ であることが分かっている。

挿入ソートの安定性は，同じ値が含まれている時は，入力の順序どおりに並べ替えられるため成立している。これは，プログラム 2 の 6 行目が関係している。$\mathrm{D}[j-1] > x$ の場合は値を右にずらし，それ以外の場合は $\mathrm{D}[j]$ に値を挿入，つまり，$\mathrm{D}[j-1] = x$ の場合は $\mathrm{D}[j-1]$ の後に x を挿入する。以上から，定義 3 の条件(b)を満たすため，挿入ソートは安定な整列アルゴリズムであると言える。

(3) 選択ソート

選択ソート(Selection sort) とは，入力データから最大値を見つけて最後尾へ，残りのデータに対しても同様の操作を繰り返してソートする方法である。選択ソートのアルゴリズムをプログラムで記述すると以下のようになる。

プログラム 3　選択ソート

```
1  // ------------ 選択ソート ------------ //
2  void Selectionsort (int D[] , int n){
3        int max, max_index, i, j;
```

```
 4       for(i = n-1 ; i > 0 ; i--){
 5            max = D[0];   max_index = 0;
 6            for(j = 1 ; j <= i ; j++){
 7                 if(D[j] >= max){
 8                      max = D[j];   max_index = j;
 9                 }
10            }
11            swap(D[max_index], D[i]);
12       }
13       return;
14  }
```

プログラム 3 に関して説明する。サイズ n の配列 D をソートするため，関数の仮引数は配列 D とデータの個数 n である。4 行目で $i = n - 1$ と初期化し，$i > 0$ が成立しなくなるまで i をディクリメントしながら for 文の中身を実行させる。5 行目で $max = \mathrm{D}[0]$，$max_{index} = 0$ とし，6 行目の for 文に移動する。$j = 1$ と初期化し，$j \leq i$ が成立しなくなるまで j をインクリメントしながら for 文の中身を実行させる。7 行目で $\mathrm{D}[j]$ と max を比較し，$\mathrm{D}[j] \geq max$ の場合は 8 行目で $max = \mathrm{D}[j]$，$max_{index} = j$ を実行する。内側の for 文を実行し，ソートされていない配列の中で最大値を見つけたら，11 行目の swap 関数で，最大値とソートされていない配列の右端の値を交換する。これを $i > 0$ が成立しなくなるまで繰り返すことにより，選択ソートは完了する。

選択ソートの時間計算量について考える。プログラム 3 は，入力によってアルゴリズムの実行時間（比較回数）が変化しないため，最良時間計算量と最悪時間計算量は等しい。というのは，最大値を探索する際に，ソートされていない配列を全て暫定の最大値と必ず比較するためである。

では，時間計算量について考えていく。プログラム 3 は二重の for 文により構成されている。外側の for 文の繰り返し回数は $n - 1$ 回であり，内側の for 文は外側の for 文の変数 i に依存し，$k = i$ の場合，for 文は k 回繰り返される。各 for 文の中の操作は，数回の比較・代入操作のみであり，$O(1)$ となる。よって，時間計算量は次のようになる。

$$T(n) = \sum_{k=1}^{n-1} k \times O(1) = O(1) \times \frac{n(n-1)}{2} = O(n^2) \tag{5}$$

次に，選択ソートの安定性について説明する。選択ソートは最大値と右端の値を交換するため，データの順序はバラバラになる。よって，一般的には選択ソートは安定性の条件を満たしていないので，安定なソートにするためには工夫が必要となる。

３.２.　$O(n\log n)$ の整列アルゴリズム

$O(n\log n)$ の整列アルゴリズムとして，クイックソート（[2]），ヒープソート（[3]），マージソート（[7]）の順に述べる。それぞれ整列アルゴリズムの概要と C 言語による関数形式のプログラムを示し，各整列アルゴリムの詳細について説明する。

(1) クイックソート

クイックソート(Quick sort)とは，あるデータを基準値(ピボット)とし，対象となるデータ集合を基準値より大きいグループと小さいグループに分類し，各グループについて同様の操作を繰り返すことによってソートする方法であり，これを各グループに再帰的に実行することによりソートが完了する。この再帰的な部分を実現するプログラムは以下のようになる。

プログラム 4-1　クイックソートの再帰部分

```
1  // ------------ Quick sort 関数 ------------ //
2  void Quicksort(int D[], int left, int right){
3       int pivot_index;
```

```
4       if(left >= right) return;
5       pivot_index = partition(D, left, right);
6       Quicksort(D, left, pivot_index - 1);
7       Quicksort(D, pivot_index + 1, right);
8   }
```

プログラム 4-1 を説明する。まず，配列 D の左端の添字を left，右端の添字を right と設定している。4 行目で left と right の値を比較し，$\text{left} \geq \text{right}$ つまり，配列 D に含まれる値が 1 つ以下の場合はアルゴリズム終了する。5 行目の partition 関数で基準値が格納されている配列の添字 pivot_index を算出する。partition 関数内では，基準値を決定する操作，配列 D を 2 つに分ける操作を行っているが，詳しくは後述する。partition 関数内で配列 D を基準値未満のデータの集合 D_1 ，基準値以上のデータの集合 D_2 に分けることができるため，これを各々6, 7 行目で再帰的に実行させている。

クイックソートは，次に詳述する partition 関数が重要な役割を持っており，この関数のプログラムは以下のようになる。

プログラム 4-2　partition 関数

```
1   // ------------ partition 関数 ------------ //
2   int partition(int D[], int left, int right){
3       int k, i, j;
4       k = rand() % (right - left + 1) + left;
5       swap(D[k],D[right]);
6       i = left;   j = right - 1;
7       while(i <= j){
8           while(D[i] < D[right]) i++;
9           while((D[j] >= D[right]) && (j>=i)) j--;
10          if(i < j){ swap(D[i],D[j]);}
11      }
12      swap(D[i],D[right]);
13      return i;
14  }
```

プログラム 4-2 について説明する。4 行目で $\mathrm{D}[left]$ から $\mathrm{D}[right]$ の間で基準値をランダムに決める。5 行目で基準値と右端の値を入れ替え，6 行目で i と j の値を初期化する。7 行目から 11 行目の while 文で基準値未満のデータを左に，基準値以上のデータを右に移動させる。$i > j$ となって while 文を抜け出し，12 行目で $\mathrm{D}[i]$，つまり基準値以上のデータの集合の左端と基準値が格納されている $\mathrm{D}[right]$ を交換する。13 行目で基準値が格納されている添字 i を Quicksort 関数に返す。

クイックソートの時間計算量について考える。まず，partition 関数の配列のサイズを $(right - left + 1) = n$ と仮定する。この関数は主に 2 重の while 文で構成されているが，外側の while 文の繰り返し回数に関わらず，内側の 2 つの while 文の繰り返し回数は合わせて最大 $n - 1$ 回である。なぜなら，最初 $n - 1$ だけ離れている i と j は内側の while 文を実行するたびに 1 つずつ近づき，最大 $n - 1$ 回の実行により $i > j$ となり，外側の while 文の終了条件が満たされるからである。while 文以外の操作は定数時間で実行可能なので，partition 関数の最大時間計算量は

$$T(n) = O(1) \cdot (n - 1) = O(n) \tag{6}$$

となる。以下，時間計算量を計算しやすくするため，サイズ n の場合の partition 関数の時間計算量は，定数 c を用いて $O(n) = cn$ とする。

次に，クイックソートの再帰を踏まえて時間計算量を考える。クイックソートは，基準値の選び方によって時間計算量が変化する。アルゴリズムが最も遅い場合は，基準値として常に最小値もしくは最大値が選ばれる場合である。このとき，配列 D は“基準値と基準値以上のデータ”もしくは“基準値と基準値以下のデータ”に分割される。今回は，基準値に毎回最小値が選ばれる場合を考える。詳細は省略するが，この場合の再帰木を構築し，再帰木から時間計算量を求めると再帰木の高さは n であり，節

点の時間計算量は $cn, c(n-1), \cdots, c$ となっている。再帰アルゴリズムの時間計算量は，再帰木のすべての節点が表す時間計算量の和に等しいため，クイックソートの最悪時間計算量は次のようになる。

$$T(n) = \sum_{i=0}^{n-1} c(n-i) = \sum_{i=1}^{n} ci = c\frac{n(n+1)}{2} = O(n^2) \tag{7}$$

一方，最もアルゴリズムの動作が高速な場合は，基準値として常に中央値が選ばれる場合である。この場合，“基準値未満のデータ”と“基準値以上のデータ”のサイズはほぼ等しくなり，均等に分割される。この場合の再帰木を構築すると，葉の数はデータ数 n であり，木の高さは $O(\log n)$ である。また，再帰木の各レベルの節点に含まれる時間計算量の和はすべて cn である。よって，最良時間計算量は

$$T(n) = cn \cdot O(\log n) = O(n \log n) \tag{8}$$

である。また，n 個のデータから基準値を選ぶ方法はランダムなため，基準値が各添字に格納される確率は $1/n$ である。このことから，平均時間計算量も $O(n \log n)$ であることが証明されている。

クイックソートの安定性について考える。クイックソートの場合，データがどの位置に格納されているかを考慮せずに交換の操作を行うため，同じ値のデータが入力の順序どおりに並べ替られるとはいえない。そのため，一般的にクイックソートは安定性の条件を満たしていないので，安定なソートにするためには工夫が必要となる。

(2) ヒープソート

ヒープソート(Heapsort)とは，2 分木の各節点にヒープの条件を満たすようにデータを格納し，根にある最大値のデータを取り出した後，ヒープの再構成を行う。これを繰り返すことによりソートする方法である。ヒープとはデータ構造のひとつで，以下のように定義される。

定義 4 （ヒープ）　次の 2 つの条件を満たす 2 分木をヒープと呼ぶ。

(a) 2 分木の最大のレベルを l_m とすると，$0 \le k \le l_m - 1$ を満たす各レベル k には，2^k 個の節点が存在し，レベル l_m に存在する葉はそのレベルに左詰めされている。

(b) 各節点に保存されているデータは，その子に保存されるデータより大きい。

定義 4 にしたがって 2 分木にデータを保存していくと，必ず根に最大値が格納されることになる。このヒープを利用し，以下の 2 つの関数を用いることでソートが完了する。

(a) push_heap 関数
配列に格納された n 個のデータについて，定義 4 (b) の条件を満たすように並べ替えの操作を行い，ヒープの性質を満たす 2 分木を作成する。

(b) delete_maximum 関数
(a) で作成されたヒープを表す 2 分木に対して，根の値の出力とヒープの再構成を n 回繰り返し，データを取り出した順に並べる。

上のアルゴリズムをプログラムで記述すると以下のようになる。

プログラム 5-1　ヒープソート

```
1  // ------------ Heapsort 関数 ------------ //
2  void Heapsort(int D[], int n){
3        int i, *T;   // ポインタ型の変数を作成 (int 型)
4        T = (int*)malloc(sizeof(int)*n+1);
5        push_heap(D, T, n);
6        for(i = n-1 ; i >= 0 ; i--){
7              D[i] = delete_maximum(T, i+1);
8        }
9        free(T);
10 }
```

プログラム 5-1 に関して説明する。まず，4 行目で malloc 関数を用いてサイズ $n+1$ の配列 T を生成する。5 行目で push_heap 関数を呼び出してヒープを生成し，6 行目に移る。for 文内の条件を満たさなくなるまで 7 行目を実行させる。7 行目で delete_maximum 関数を呼び出し，D[i] に根の値を代入する。また，push_heap 関数は以下のようになる。

プログラム 5-2　push_heap 関数

```
1  // ---------- push_heap 関数 ---------- //
2  void push_heap(int D[], int T[], int n){
3        int k, i, size = 1;
4        for(i = 0 ; i < n ; i++){
5              T[size] = D[i];
6              k = size;
7              while((T[k] > T[k/2])&&(k > 1)){
8                    swap(T[k], T[k/2]);   k = k/2;
9              }
10             size++;
11       }
12 }
```

プログラム 5-2 に関して，まず 4 行目の for 文内の条件を満たさなくなるまで 5 行目から 11 行目までを実行させる。5 行目でデータの追加，6 行目で添字の初期化を行い，7 行目で親と子の値を比較する。子 (T[k]) ＞ 親 (T[$\lfloor k/2 \rfloor$]) かつ $k > 1$ が成立しなくなるまで 8 行目の交換・添字の更新を行う。while 文を抜けたら 10 行目で $size$ をインクリメントし，これを繰り返すことでヒープを作成する。

プログラム 5-1 の 7 行目の delete_maximum 関数は以下のようになる。

プログラム 5-3　delete_maximum 関数

```
1  // ---------- delete_maximum 関数 ---------- //
2  int delete_maximum(int T[], int i){
3        int k, size, big;
4        size = i;
5        T[0] = T[1];
6        T[1] = T[size];
7        k = 1;
8        while(2*k <= size){
9              if(2*k == size){
10                   if(T[k] < T[2*k]){
11                         swap(T[k], T[2*k]);   k = 2*k;
12                   }
13                   else{ break; }
14             }
15             else{
16                   if(T[2*k] < T[2*k+1]) { big = 2 * k + 1; }
17                   else{ big = 2 * k; }
18                   if(T[k] < T[big]){
19                         swap(T[k], T[big]);   k = big;
20                   }
21                   else{ break; }
22              }
23       }
24             return T[0];
25 }
```

プログラム 5-3 に関して説明する。4〜7 行目で最大値の取り出し，値の移動を行い，8 行目からヒープの再構成を行う。8 行目で子の有無を確認し，子がある場合は 9 行目に移動する。9 行目の if 文が成

り立つ場合，つまり，子が 1 つの場合は 10～14 行目を実行し，交換・代入操作もしくは 24 行目に移動する。9 行目の条件が成り立たない場合，つまり，子が 2 つの場合は 16～21 行目を実行し，交換・代入操作もしくは 24 行目に移動する。8～23 行目を繰り返すことによりヒープが再構成されたら，24 行目を実行し T[0] の値を Heapsort 関数の 7 行目に返す。

ヒープソートの最悪時間計算量について考える。まず，Heapsort 関数は主に 1 回の push_heap 関数の呼び出しと n 回の delete_maximum 関数の呼び出しで構成されている。push_heap 関数は，for 文による繰り返し回数が n 回，for 文の変数 i に依存している while 文の繰り返し回数は，$k=i$ とすると最大 $\log_2 k$ 回である。何故なら，while 文は葉に格納された値を最悪の場合は根まで移動させるものなので，木のレベルが while 文の繰り返し回数になるからである。また，delete_maximum 関数は heapsort 関数の for 文の変数 i に依存している。$k=i$ とすると，while 文の繰り返し回数は最大 $\log_2 k$ 回である。これは，push_heap 関数と同様で，while 文は根に移動した値を最悪の場合は葉まで移動させるものなので，木のレベルが while 文の繰り返し回数になるからである。

以上より，ヒープソートの最悪時間計算量は次のようになり，証明は省略するが，最良時間計算量も $O(n\log n)$ であることが分かっている。

$$\begin{aligned}\sum_{k=1}^{n-1}\left(O(1)\times(\log_2 k+\log_2 k)\right) &= O(1)\times 2\sum_{k=1}^{n-1}\log_2 k \\ &\le O(1)\times 2(n-1)\log_2(n-1) \\ &= O(n\log n)\end{aligned} \tag{9}$$

最後に，ヒープソートの安定性について考える。プログラムの動作から分かるように，ヒープは入力の順序を考慮せずに値の交換を行う。そのため，一般的にヒープソートは安定性の条件を満たしていないので，安定なソートにするためには工夫が必要となる。

(3) マージソート

マージソート(Merge sort)とは，入力したデータを要素が 1 つになるまで分割し，その後，マージ（併合）という操作でソートしながら組み合わせることによってソートを完成させる方法である。

まず，マージとは，ソート済みの 2 つのデータ列を 1 つのソートされたデータ列に併合することである。マージソートは，以下の 2 つの関数を用いることでソートが完了する。

(a) Mergesort 関数（分割）
入力したデータを，$D_1=\{d_0,d_1,\cdots,d_{n/2-1}\}$，$D_2=\{d_{n/2},d_{n/2+1},\cdots,d_{n-1}\}$ という 2 つの集合に分割する。要素が 1 つになるまで再帰関数を用いて繰り返す。

(b) merge 関数
隣同士の集合 D_1 と集合 D_2 をマージする。この操作を集合が 1 つになるまで繰り返す。

上の分割する Mergesort 関数をプログラムで記述すると以下のようになる。

プログラム 6-1　Mergesort 関数

```
1  // ------------- Mergesort ------------- //
2  void Mergesort(int D[], int left, int right){
3        int mid;
4        if(left >= right) return;
5        mid = (left + right) / 2;
6        if(left<mid) Mergesort(D, left, mid);
7        if(mid+1<right) Mergesort(D, mid + 1, right);
8        merge(D, left, mid, right);
9  }
```

プログラム 6-1 について説明する。4 行目で配列 D に含まれる要素の数を確認し，要素数が 2 つ以上の場合 5 行目に進む。5 行目で mid の値を計算し，6・7 行目で配列を 2 つに分ける。この操作を繰

り返すことで分割が完了する。

上のプログラムの 8 行目の merge 関数を記述すると以下のようになる。

プログラム 6-2　merge 関数

```
1  // ---------- merge 関数 ---------- //
2  void merge(int D[], int left, int mid, int right){
3        int x, y, i, *A;
4        int m = right - left + 1;
5        A = (int*)malloc(sizeof(int)*m);
6        x = left;   y = mid + 1;
7        for(i = 0 ; i <= m-1 ; i++){
8              if(x == mid + 1){ A[i] = D[y]; y++; }
9              else if(y == right+1){ A[i] = D[x]; x++; }
10             else if(D[x] <= D[y]){ A[i] = D[x]; x++; }
11             else{ A[i] = D[y]; y++; }
12       }
13       for(i = 0 ; i <= m-1 ; i++) D[i+left] = A[i];
14       free(A);
15 }
```

プログラム 6-2 について説明する。5 行目で malloc 関数を用いてサイズ m の配列 A を生成する。6 行目で変数 x と y の値を配列 D_1 と配列 D_2 の左端の値で初期化し，7 行目から 12 行目までの for 文へ移動する。for 文では，配列 D_1 と配列 D_2 に格納されていたデータが昇順に並ぶように配列 A に格納されていく。$i \leq m-1$ が不成立となると，配列 A にデータが全て格納されたことになる。13 行目で配列 A に格納されたデータを配列 D に移し，merge 関数は終了する。

マージソートの時間計算量について考える。マージソートは最悪時間計算量と最良時間計算量が同じである。何故なら，データがどう並んでいるかに拘わらず，分割とマージが行われているためである。マージソートは 2 つの 2 分木で成り立っている。各再帰部分の実行時間はデータ数に依存するため，時間計算量はクイックソート関数の時に記述した再帰木と同様の考え方となる。よって，分割にかかる時間計算量は $O(n\log n)$，マージにかかる時間計算量も $O(n\log n)$である。このことから，マージソートの時間計算量は $O(n\log n)$である。

また，プログラム 6-2 の 10 行目から分かるように，マージソートは同じ値がある場合，初めに配列 A に格納されるのは入力の順序が早いデータである。よって，定義 3 の条件(b)を満たす。したがって，マージソートは安定な整列アルゴリズムであると言える。

４．各整列アルゴリズムの数値実験による比較評価

本節では，上述した整列アルゴリズムに対して，実際に非負整数データを与え，整列が完了するまでの実行時間を計測していく。今回は各ソートの各データ数を 10 回ずつ計測し，その平均を算出した。また，データは整数乱数を使用し，上限は $D_{max} = RAND_{max}$（21 億程度）としている。実行時間の制約から，$O(n^2)$ の挿入・選択・バブルソートでは，データ数 n を 1,000 から 25 万個まで，$O(n\log n)$ のクイック・マージ・ヒープソートでは，5 万から 1 億個まで変化させて計測している。これらの結果を表 1-1, 1-2 にまとめている。なお，実行時間を計測した環境は，「デバイス : V510-15IKB, CPU : Intel(R) Core(TM) i5-7200U, 2.50GHz，OS：Windows 10 Pro，RAM：8.00GB」である。

表 1-1　$O(n^2)$のソートの実行時間（単位：秒）

データ数 n	1,000	5,000	1 万	5 万	10 万	25 万
挿入ソート	0.00	0.02	0.07	1.76	7.05	44.11
選択ソート	0.00	0.03	0.13	3.22	12.90	80.60
バブルソート	0.01	0.08	0.39	10.48	42.57	266.86

表 1-2　$O(n \log n)$のソートの実行時間（単位：秒）

データ数 n	5 万	10 万	50 万	1 千万	5 千万	1 億
クイックソート	0.01	0.02	0.09	2.35	12.78	26.45
マージソート	0.02	0.03	0.17	4.07	22.00	45.37
ヒープソート	0.01	0.03	0.16	5.55	36.24	81.02

表 1-1 の挿入ソート，選択ソート，バブルソートの実行時間をグラフ化すると，以下のようになる。

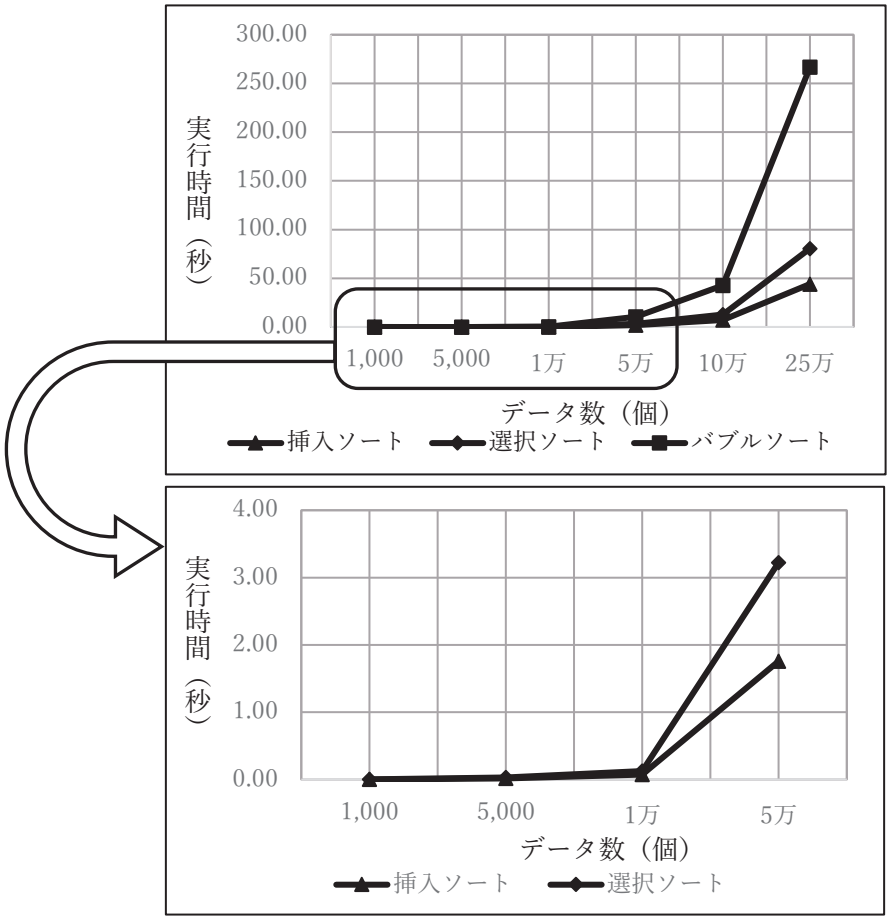

図 1　$O(n^2)$ のソートの実行時間

図 1 は挿入ソート，選択ソート，バブルソートの実行時間をグラフ化したものである。縦軸を実行時間，横軸をデータ数とし，単位はそれぞれ秒，個としている。また，上の図のみではデータ数 1,000 から 5 万個までの相違が挿入ソートと選択ソートが接近していて分かりづらいため，下の図で拡大した。図 1 を見ると，$O(n^2)$ のソートは挿入ソート，選択ソート，バブルソートの順で実行時間が早くなっていることが分かる。また，3 つのソート法とも $O(n^2)$ と同じであるが，全体を見ると実行時間に最大 6 倍程度の差が生じている。

次の図 2 は，表 1-2 のクイックソート，マージソート，ヒープソートの実行時間をグラフ化したものである。なお，上の図のみではデータ数 5 万から 1 千万個までの相違がクイックソートとマージソートが接近していて分かりづらいため，下の図で拡大している。

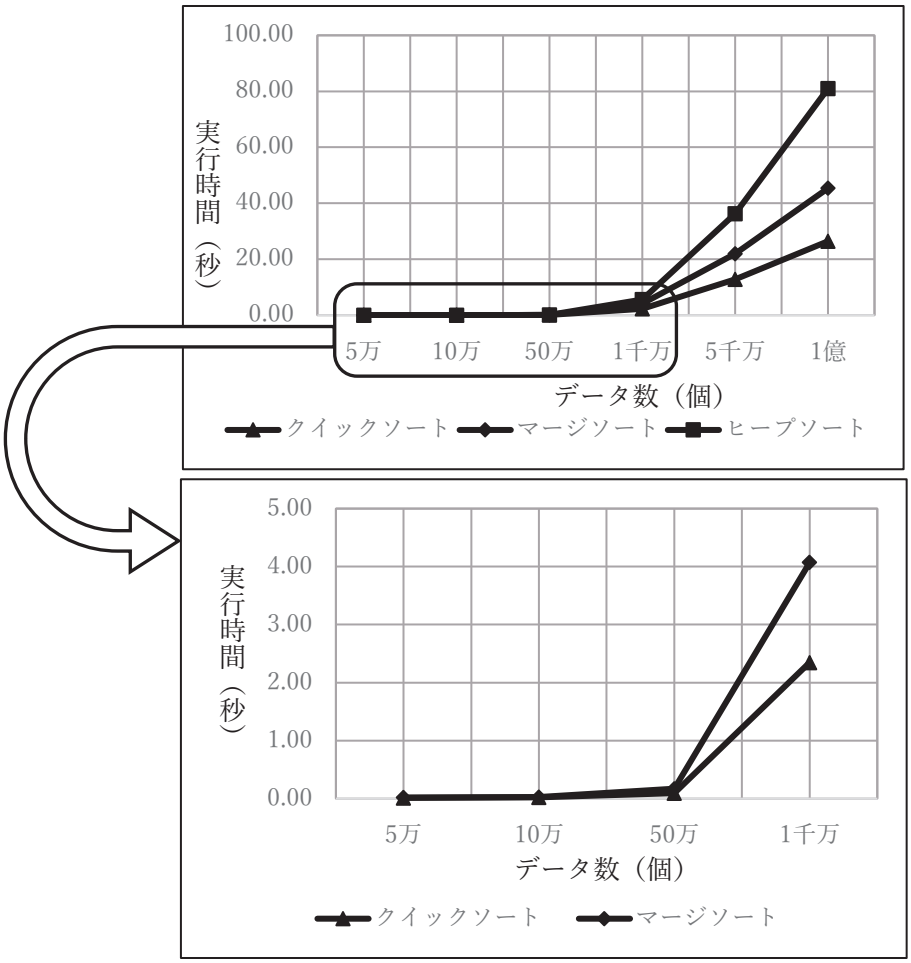

図 2　$O(n\log n)$ のソートの実行時間

図 2 を見ると，$O(n\log n)$ のソートはクイックソート，マージソート，ヒープソートの順で実行時間は早くなっていることが分かる。また，3 つのソート法とも $O(n\log n)$ と同じであるが，全体を見ると実行時間に最大 3 倍程度の差が生じている。

表 2　クイックソートとマージソートの比較（単位：秒）

データ数	5 万	10 万	50 万	1 千万	5 千万	1 億
クイックソート (a)	0.01	0.02	0.09	2.35	12.78	26.45
マージソート (b)	0.02	0.03	0.17	4.07	22.00	45.37
比 a/b	0.60	0.67	0.54	0.58	0.58	0.58

表 1-1, 表 1-2, 図 1 と図 2 から分かるように, 時間計算量と実行時間の結果から, $O(n^2)$ と $O(n\log n)$ の間には大きな差が生じている。例えば，データ数が共通している 10 万個のデータに対して，最速のクイックソートは 0.02 秒，最も時間のかかるバブルソートは 42.57 秒を要している。その比は約 2100 倍程度のかなり大きな値となる。今回の数値実験により，代表的な 6 種類の整列アルゴリムの中で，最も早い整列アルゴリズムはクイックソート，最も遅いソートはバブルソート，$O(n^2)$の中で最も早い整列アルゴリズムは挿入ソートであることが分かった。特に，クイックソートとマージソートを比較すると，表 2 のようになる。表 2 は，表 1-2 のデータに対してクイックソートとマージソートの実行時間の比を表の 4 行目に載せたものである。表 2 を見て分かるように，クイックソートはマージソート

の約半分程度（約 58%）の時間で整列を完了することが分かった。

５．二分探索挿入ソートの提案

第 4 節で述べたように挿入ソートは，$O(n^2)$のアルゴリズムの中で最も早い整列アルゴリズムであり，ソートの安定性の条件を満たしたアルゴリズムである。挿入ソートでは，「すでに整列している配列（整列済みデータ）に新たなデータを適切な場所を見つけて追加する操作を繰り返す方法」であり，新しいデータと整列済みデータの最後尾のデータ（整列済みのデータの最大値）とを比較し，新しいデータが小さければ，最後尾のデータを一つ後ろへずらし，整列済みの最後尾の一つ前のデータと新しいデータを比較する。これを繰り返し，新しいデータよりも小さいか等しいデータが出現したら，その後に新しいデータを挿入する。このようにして，与えられたデータを昇順に整列させるアルゴリムである。このため，もし与えられたデータがほぼ昇順に並んでいるよう場合は，新しいデータは最後尾のデータなどと数回比較してすぐに適切な場所が判明しそこへ新しいデータが挿入される。これを最後のデータまで行うと整列が完了するので，最良時間計算量は$O(n)$であると考えられ，クイックソートなどの$O(n\log n)$のアルゴリズムよりも早く整列できる。一方，データがほぼ降順に並んでいるような場合は，整列済みのデータに対して，新しいデータは常に先頭近くに位置するので，最後尾からそこまでのデータの比較を毎回繰り返すことになる。これは，ちょうどバブルソートの整列方法の操作と同程度の比較回数が必要になるため，$O(n^2)$のアルゴリズムの中でも時間のかかる場合に相当する。

一般のランダムなデータに対しては，整列済みのデータ数が m 個であれば，新しいデータの場所を探すのに平均的には $m/2$ 回の比較が必要となる。そのため，整列済みのデータの個数 m が全データ数n 個に近づいて行けばいくほど，多くの比較を行うことになる。そこで，本節では，データがすでにソートされているという利点を生かし，新しいデータに対して，二分探索法により挿入すべき場所を探し出す。そして，挿入すべき場所以降のデータを一気に後方へずらし，挿入すべき場所に新しいデータを格納する方法を提案する。この整列アルゴリズムを，**BI** ソート (Binary search & Insertion Sort) と呼ぶこととする。BI ソートのプログラムを以下に示す。

プログラム 7　BI ソート

```
1  // ------------ BIsort 関数 ------------ //
2  void BIsort(int D[], int n){
3      int   i, j, right, left, mid, x;
4      for(i = 1 ; i < n ; i++){
5          x = D[i];   left = 0;   right = i;
6          while(left < right){
7              mid = (left + right) / 2;
8              if(D[mid] <= x)   left = mid + 1;
9              else   right = mid;
10         }
11         if(i != left){
12             for(j = i ; j > left ; j--){
13                 D[j] = D[j-1];
14             }
15             D[left] = x;
16         }
17     }
18     return;
19 }
```

プログラム 7 について説明する。4 行目で $i=1$ と初期化し，$i<n$ が成立しなくなるまで i をインクリメントしながら for 文の本体（17 行目まで）を実行させる。5 行目で $x=\mathrm{D}[i]$，$left=0$，$right=i$ とし，6 行目の while 文に移動する。$left<right$ の場合，7 行目で $mid=\lfloor(left+right)/2\rfloor$ を計算する。$\mathrm{D}[mid]$ と x を比較した結果に応じて 8，9 行目を実行する。while 文の条件を満たさなくな

るまで 7 行目から 9 行目を実行すると，データ x を挿入する位置 $left$ が見つかる。11 行目で $i \neq jeft$，つまり，データ x を挿入する場所が $D[i]$ でない場合，12 行目から 13 行目を実行し，挿入する位置より右にあるデータを一つずつずらす。最後に空いた $D[left]$ にデータ x を格納し 4 行目へ戻る。$i = left$ の場合は，データ x が既にソートされているデータの中で最大であるため，if 文の中身を実行せずに 4 行目へ戻る。以上を$i = n$ が成立するまで繰り返すと整列が完了することとなる。

BI ソートの安定性に関して考察していく。プログラム 7 の 6 行目の while 文を見ても分かるように，同じデータが含まれているときは入力の順序どおりに並べ替えられる。具体的には，プログラム 7 の 8 行目が関係しているが，$D[mid] \leq x$，つまり，データが同じ値の場合はソートされているデータの方を小さいとして処理するためである。以上から，定義 3 の条件(b)を満たすため，BI ソートは安定な整列アルゴリズムであると言える。

６．二分探索挿入ソートの数値実験による効率性の評価

第 5 節で提案した BI ソートは二分探索を利用することでデータの比較回数を少なくしている。また，整列済みのデータに新たに挿入しようとするデータがどのような値のデータであっても，二分探索では比較回数は変わらないため，データ数が大きくなればなるほど挿入ソートよりも効率よくソートすることが可能であると予想される。では，実際に SI ソートの整列までの実行時間を計測し，どれほど実行時間が早まるのか見ていく。今回は各ソートを 5 回ずつ実行し，その平均を算出した。また，データは整数乱数を使用し，上限は $D_{max} = RAND_{max}$（21 億程度）としている。それぞれ 5 万から 50 万個までデータ数を 5 万個ずつ変化させた。これらのデータをまとめ，従来の挿入ソートに対する BI ソートの実行時間の短縮率を示したものが，次の表 3 である。

表 3　2 つのソートの実行時間（単位：秒）

データ数	5 万	10 万	15 万	20 万	25 万	30 万	35 万	40 万	45 万	50 万
BI ソート	1.20	4.75	10.73	19.11	30.05	42.93	58.55	76.48	96.85	119.39
挿入ソート	1.78	7.19	16.13	28.79	44.75	64.24	88.17	114.57	145.25	179.52
比率（%）(BI / 挿入)	67.42	66.06	66.52	66.38	67.15	66.83	66.41	66.75	66.68	66.51

表 3 をグラフ化すると，以下のようになる。

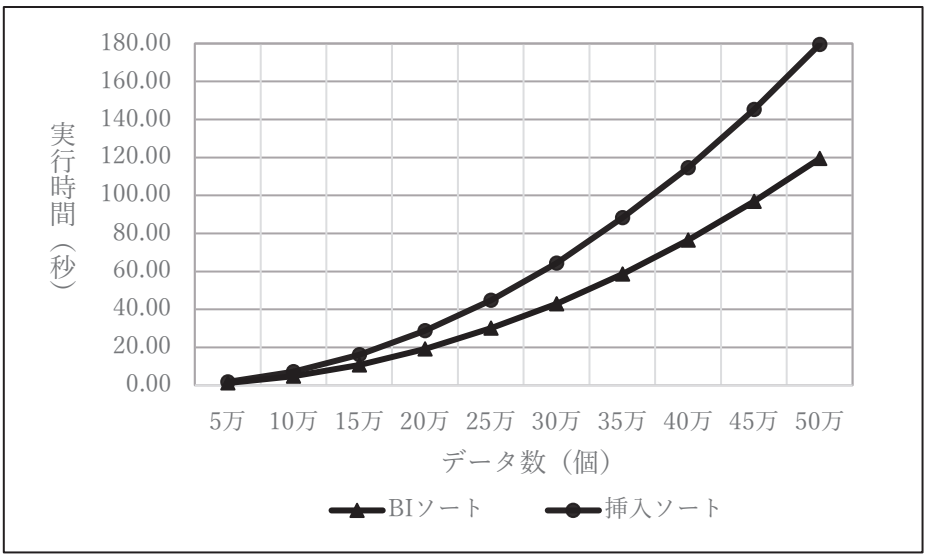

図 3　2 つのソートの実行時間

図 3 は表 3 の BI ソートと挿入ソートの実行時間をグラフ化したものである。縦軸を実行時間，横軸をデータ数とし，単位はそれぞれ秒，個としている。表 3，図 3 を見ると，データ数が 5 万から 50 万個の全ての場合において，BI ソートの方が挿入ソートよりも高速に整列を完了することが分かった。ただ，時間計算量の評価では，挿入ソートと BI ソートは共に同じ$O(n^2)$の整列アルゴリズムである。また，表 3 を見ると，BI ソートは挿入ソートよりも約 66~68% 効率よくソートが完了することも分かる。このことより，BI ソートは挿入ソートと同じく安定なソートで，挿入ソートよりも高速に実行できる整列アルゴリズムであると言える。

7．　まとめと今後の課題

本論文では，6 種類の代表的な整列アルゴリズムについて，それぞれ整列アルゴリズムの概要や理論的な性質と C 言語による関数形式のプログラムを示し，各整列アルゴリムの詳細について説明した。$O(n^2)$ の整列アルゴリズムとして，バブルソート，挿入ソート，選択ソート，$O(n\log n)$ の整列アルゴリズムとして，クイックソート，ヒープソート，マージソートについて考察した。また，数値実験により実際の整数データに対する各整列アルゴリムの実行時間を計測・比較評価し，$O(n^2)$ の整列アルゴリズムでは挿入ソートが，$O(n\log n)$ の整列アルゴリズムではクイックソートが最も効率が良いことが分かった。さらに，二分探索挿入ソートという新しい整列アルゴリズムを提案し，従来の挿入ソートとの比較を行い，約 68%程度で高速にソートできることを確認した。

今後，本論文の成果をもとに，クイックソートと今回提案した二分探索挿入ソートを組み合わせたハイブリッドなソート方法の提案と数値実験による比較評価，二分探索挿入ソートをバケットソートなどのデータの比較を伴わない高速なソート方法へ応用する方法などの研究を展開したい。

参考文献

[1] D. L. Shell, “A High-speed Sorting Procedure.”, Communications of the ACM, Vol.2, No.7, pp.30-32, 1959.
[2] C. A. R. Hoare, “Quicksort.”, Computer Journal, Vol.5, No.1, pp.10-15, 1962.
[3] J. W. J. Williams, “Algorithm 232: Heapsort.”, Communications of the ACM, Vol.7, No.6, pp.347-348, 1964.
[4] B.W.カーニハン，D.M.リッチー著，石田晴久訳，プログラミング言語 C，共立出版，1989.
[5] 奥村晴彦，C 言語による最新アルゴリズム事典，技術評論社，1991.
[6] R. Sedgewich, “Analysis of Shellsort and Related Algorithms.” Lecture Notes in Computer Science, Vol.1136, pp.1-11, 1996.
[7] R. Sedgewick 著，野下他訳，アルゴリズム C 第一巻 基礎・整列，近代科学社，1996.
[8] 近藤嘉雪，C プログラマのためのアルゴリズムとデータ構造，SB クリエイティブ，1998.
[9] 茨木俊秀，C によるアルゴリズムとデータ構造，昭晃堂，1999.
[10] 紀平拓男，春日伸弥，アルゴリズムとデータ構造，ソフトバンクパブリッシング，2003.
[11] R. Sedgewick 著，野下他訳，アルゴリズム C・新版，近代科学社，2004.
[12] M. A. Bender, M. F. Colton and M. A. Mosteiro, “Insertion Sort is $O(n \log n)$.”, Theory of Computing Systems, Vol.39, No.3, pp.391-397, 2006.
[13] Michael Sipser 著，太田他訳，計算理論の基礎 3.複雑さの理論，共立出版，2008.
[14] 原・永田・大川著，西尾監修，アルゴリズムとデータ構造，共立出版，2012.
[15] T. H. Cormen 他著，浅野他訳，アルゴリズムイントロダクション第３版総合版：世界標準 MIT 教科書，近代科学社，2013.
[16] 藤原暁宏，アルゴリズムとデータ構造，森北出版，2016.
[17] 柴田望洋，新・明解 C 言語で学ぶアルゴリズムとデータ構造，SB クリエイティブ，2017.
[18] 河西朝雄，改訂第 4 版 C 言語によるはじめてのアルゴリズム入門，技術評論社，2017.