情報工学実験II
ソーティングプログラム

ライモン ウィジャヤ

2021-06-11

目 次

1　概要

少量または大量なデータに対して, 小さい順 (昇順) または大きい順 (降順) に並び替えする作業はソーティングである. データをソーティング時間が早ければ早いほどソーティングの効率がいいと言える. ソーティングの中ではたった一つのソーティング方法でどんな数値列を早くソーティングするのはない. 次はソーティングアルゴリズムをいくつか紹介する. どんなアルゴリズムを持ってるか, どれぐらいデータをソーティングする時間がかかるかなどを説明する.

アルゴリズムに従って計算を実行をそのアルゴリズムの**時間計算量** (*time complexity*) と呼び, 例えば, あるｎデータに対してそのアルゴリズム計算量を表し方はオーダー ($O(\mathrm{n})$) と記す.

全プログラムのコードは C 言語で書いている. 用意乱数データは九つあり, それぞれの特徴は表 1 に載せている.

表 1: データタイプ

データ	タイプ
data 1-3	ランダム
data 4	昇順
data 5	降順
data 6	バイトニック
data 7	ジグザグ
data 8	ランダム-マイナス
data 9	ランダム-マイナス-少数

2　バケットソート

バケットソートとは、別名バケツソートとも呼ばれ, あらかじめ順番通り並べて準備されたバケツに, データを放り込むことで並べ替えを行おうという, いささか乱暴なソートアルゴリズムである.

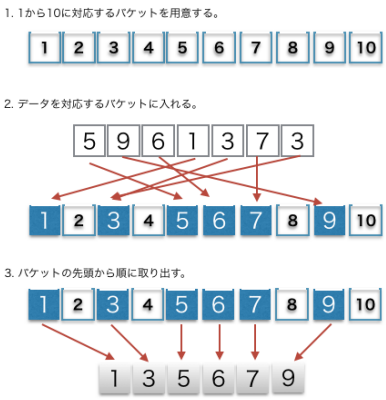

図 1: バケットソートイメージ

この方法は, データの存在する範囲が有限個に限定されていないと使えませんが, 使える場合は非常に高速に並べ替えを実行できる, きわめて有用なアルゴリズムである. イメージは図 1 のようにである.

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<time.h>
#define N 100000

int maxval(int a[])
{
    int max = a[0];
    int i;

    for(i = 1; i < N; i++){
        if(max < a[i]){
            max = a[i];
        }
    }
    return max;
}

void bucket_sort(int number[], int maxvalue)
{
    int bucket[maxvalue + 1];
    int i,j, counter;

    for(i = 0; i <= maxvalue; i++){
        bucket[i] = 0;
    }

    for(i = 0; i < N; i++){
        bucket[number[i]] += 1;
    }

    j = 0;
    for(i = 0; i <= maxvalue; i++){
        if (bucket[i] != 0){
            counter = bucket[i];
            while(counter > 0){
                number[j] = i;
                j++; counter--;
            }
        }
    }
}

int main(int argc, char *argv[]){
    FILE *fp1; FILE *fp2;
    int number[N];
    int i;
    int maxvalue;

/*--------- ファイルからデータの読み込み ----------*/
    if(argc != 2){
        puts("Parameter Error");
        return 0;
    }
    if ((fp1 = fopen(argv[1], "r")) == NULL){
```

```
56          puts("NO FILE WITH THAT NAME");
57          return 0;
58      }
59      for(i = 0; i < N; i++){
60          fscanf(fp1, "%d", &number[i]);
61      }
62      fclose(fp1);
63
64  /*---------- プログラム処理 ----------*/
65      clock_t begin = clock();
66      maxvalue = maxval(number);
67      bucket_sort(number, maxvalue);
68      clock_t end = clock();
69
70      fp2 = fopen("./output000.dat", "w");
71      for(i = 0; i < N; i++){
72          fprintf(fp2, "%d\n", number[i]);
73      }
74      fclose(fp2);
75      puts("Done!");
76      printf("The elapsed time is %lf sec\n", (double)(end-begin)/CLOCKS_PER_SEC);
77      return 0;
78  }
```

2.2 動作

まずは数値データを読み込んで, 数値データの中で maxval() 関数で最大値を探索する. 最大値とデータの配列を bucket_sort() 関数に渡す.

bucket_sort() 関数の中に次の動作が行われている.

1. バケットを最大値 +1 用意して, 値を初期化する.
2. 数値データと配列のインデックスが依存するので, 読み込んだ数値が配列のその数値のバケット (インデックス) を 1 増やす.
3. 全数値を読み込んだ後は, 元配列に一つずつバケットの中身を順番で取り出す.
4. ソート完了.

2.3 時間計算量

まずはバケットを m 個を準備する時間計算量は $O(\mathrm{m})$ である. 次は n 数値データをバケットに入れる時間計算量は $O(\mathrm{n})$ である. バケットソートの計算量は $O(\mathrm{m}+\mathrm{n})$ である. 次は最大値を求める時間計算量は $O(\mathrm{n})$ なので, このプログラム全体時間計算量は $O(\mathrm{m}+2\mathrm{n})$ である.

以下の表 2.3 は上のコードで実行したソート計算時間だ. (CPU は intel i7 7^{th} Gen, RAM は 8GB)

表 2.3 より, 数値データ間の計算時間の差はあまり多くないから, 数値データが乱数や降順や昇順など関係ない.

表 2: 計算時間 (バケットソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	0.001350	0.001547	0.001357	0.001740	0.001389

データ	data6	data7	data8	data9
計算時間 (秒)	0.001422	0.001358	—	—

3　挿入ソート

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. 図 2 を見たらわかりやすい. 具体的に言うと, 整列してある配列に追加要素を適切な場所に挿入するアルゴリズムである.

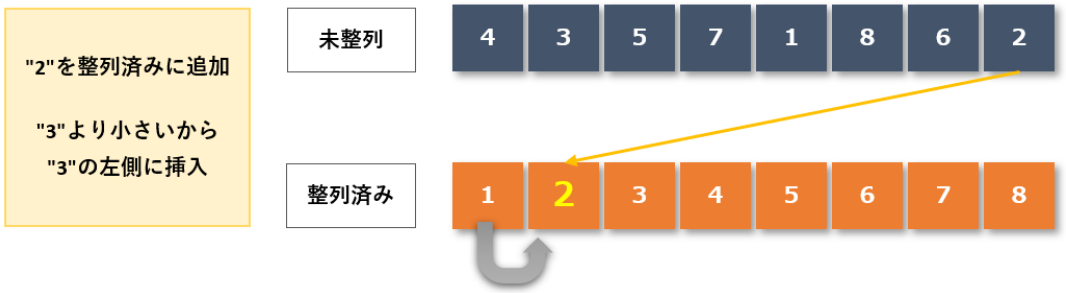

図 2: 挿入ソートイメージ

3.1　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<time.h>
#define N 100000

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

void insertion(int number[])
{
    int i, j, tmp;

    for(i=1; i < N; i++){
        tmp = number[i];
        j = i;
        while((j > 0) && (number[j-1] > tmp)){
            number[j] = number[j-1];
            j--;
        }
        number[j] = tmp;
```

```
25      }
26  }
27
28
29  int main(int argc, char *argv[])
30  {
31      FILE *fp1;
32      FILE *fp2;
33      int number[N];
34      int i = 0;
35
36      if(argc != 2){
37          puts("Parameter Error");
38          return 0;
39      }
40
41      if ((fp1 = fopen(argv[1], "r")) == NULL){
42          puts("NO FILE WITH THAT NAME");
43          return 0;
44      }
45
46      for(i = 0; i < N; i++){
47          fscanf(fp1, "%d", &number[i]);
48      }
49
50      fclose(fp1);
51  /*---------- プログラム処理 ----------*/
52      clock_t begin = clock();
53      insertion(number);
54      clock_t end = clock();
55
56      fp2 = fopen("./output000.dat", "w");
57      for(i = 0; i < N; i++){
58          fprintf(fp2, "%d\n", number[i]);
59      }
60      fclose(fp2);
61
62      puts("Done!");
63      printf("The elapsed time is %lf sec\n", (double)(end-begin)/CLOCKS_PER_SEC);
64
65      return 0;
66  }
```

3.2 動作

まずは i, j, tmp(一時的なものという) 変換を用意する. 次はソート動作に入る.

1. tmp 変数に配列 number[i] を入れ, j 変数は i 変数の値を持たせる. j 変数は前に何個数値データがどれぐらいあるかを表すものだ.

2. tmp 変数の値より大きかったら, それらの値いを入れ替える $[number[j] = number[j-1]]$ ともに j の値が 1 減る.

3. tmp 変換の値が number$[j-1]$ より高かったらループが終わりで, 最後に number[j] に tmp の値を入れる (その値の適切な場所である).

4. これは i が N まで繰り返す

5. ソートは終わり

3.3 時間計算量

格時点で着目している a_i を, その時までにソートされている系列 $(a_1, a_2, \ldots, a_{i-1})$ の間の適切な場所に挿入するというもので, a_i より大きい数値を右シフトするので時間計算量 $O(n^2)$ である.

以下の表 3.3 は上のコードで実行したソート計算時間だ. (CPU は intel i7 7^{th} Gen, RAM は 8GB) data4 は昇順データなのでソート処理は全くないから, 一瞬で終わる. しかし, 降順データ

表 3: 計算時間 (挿入ソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	5.814234	5.791112	5.762070	0.000307	11.566258
データ	data6	data7	data8	data9	
計算時間 (秒)	5.780712	5.808727	5.770340	5.934736	

data5 に対して, これは最悪状況から, 一番遅かった. このソート方はランダムのデータに対して, 少し有効だ.

4 バブルソート

バブルソートは配列において隣り合うふたつの要素の値を比較して条件に応じた交換を行う整列アルゴリズムです. 条件とは値の大小関係です.「値の大きい順 (降順)」か「値の小さい順 (昇順)」に配列を並び替えます. 図 3 がバブルソートのイメージだ.

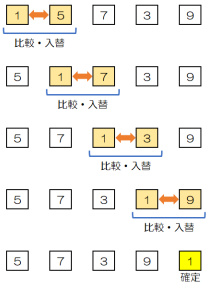

図 3: バブルソートイメージ

4.1 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<time.h>
#define N 100000

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

```
8       *a = *b;
9       *b = tmp;
10  }
11
12  void bubble_sort(int number[]){
13      int i,j, sorted;
14      j = N;
15      do{
16          sorted = 1; j=j-1;
17          for(i=1; i <= j; i++){
18              if(number[i-1] > number[i]){
19                  swap(&number[i-1], &number[i]);
20                  sorted = 0;
21              }
22          }
23      } while(!sorted);
24  }
25
26
27  int main(int argc, char *argv[])
28  {
29      FILE *fp1;
30      FILE *fp2;
31      int number[N];
32      int i;
33
34      if(argc != 2){
35          puts("Parameter Error");
36          return 0;
37      }
38
39      if ((fp1 = fopen(argv[1], "r")) == NULL){
40          puts("NO FILE WITH THAT NAME");
41          return 0;
42      }
43
44      for(i = 0; i < N; i++){
45          fscanf(fp1, "%d", &number[i]);
46      }
47
48      fclose(fp1);
49  /*---------- プログラム処理 ----------*/
50      clock_t begin = clock();
51      bubble_sort(number);
52      clock_t end = clock();
53
54      fp2 = fopen("./output000.dat", "w");
55      for(i = 0; i < N; i++){
56          fprintf(fp2, "%d\n", number[i]);
57      }
58      fclose(fp2);
59
60      puts("Done!");
61      printf("The elapsed time is %lf sec\n", (double)(end-begin)/CLOCKS_PER_SEC);
62      return 0;
63  }
```

4.2 動作

バブルソートのアルゴリズムは以下通り行われている.

1. 先頭の要素'A' と隣り合う次の要素'B' の値を比較する
2. 要素'A' が要素'B' より大きいなら、要素'A' と要素'B' の値を交換する

3. 先頭の要素を'B' に移し、要素'B' と隣り合う要素'C' の値を比較/交換する
4. 先頭の要素を'C','D','E'... と移動しながら比較/交換をリストの終端まで繰り返す
5. 最も大きい値を持つ要素が配列の終端へ浮かびあがる
6. 配列の終端には最も大きな値が入っているので、配列の終端の位置をずらして (j 変数をひとつ減らして) 手順 1〜6 を繰り返す

4.3 時間計算量

配列が n 個データを持つとすると, 一番目の要素が $n-1$ 回スキャンし, 2 番目の要素が -2 回スキャンし, $n-1$ 番目の要素までだから, $n-1+n-2+\cdots+1$ で, 結果は $\frac{n}{2}\times(n)$ ということだ. つまりバブルソートの時間計算量は $O(n^2)$ である.

以下の表 4.3 は上のコードで実行したソート計算時間だ. (CPU は intel i7 7^{th} Gen, RAM は 8GB) 表 4.3 を見ると, 乱数のデータが一番時間がかかるということがわかる. バイトニックやジ

表 4: 計算時間 (バブルソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	31.531213	31.511556	31.479725	0.000236	23.110471
データ	data6	data7	data8	data9	
計算時間 (秒)	16.938872	27.564562	31.556714	31.548570	

グザグデータは昇順データ以外よりやや早いということがわかる. data5 の比較回数や交換回数は data1 や data2 により多かったが, data1 や data2 によりもっと早い. この原因は CPU の**分岐予測**で説明できる. 逆順のデータに対しては, 常に入れ替え動作をする. 分岐予測の種類による, 常に「実行されると CPU が予測している」なら, パイプラインを最大限に活用して高速な処理を実現できる. 一方, ランダムなデータであれば, 分岐結果が一定ではないので予測の失敗が多発. この場合パイプラインを生かし切れずに性能が劣化する [1] .

5 シェーカーソート

バブルソートを、効率がよくなるように改良したものだ. 別名は、双方向バブルソート, 改良交換法である.

バブルソートではスキャンを一方向にしか行わないのに対し, シェーカーソートでは交互に二方向に行う. バブルソートと同じく安定な内部ソートである. 図 4 はシェーカーソートのイメージだ.

5.1 プログラム

以下はバケットソートのプログラムである.

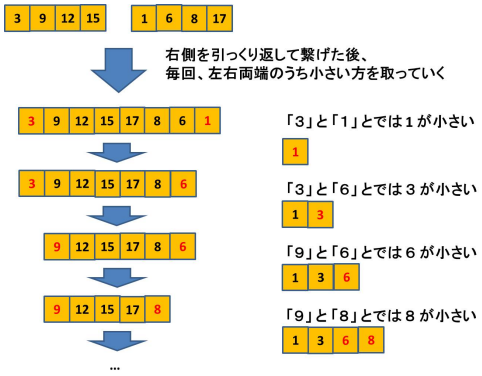

図 4: シェーカーソートイメージ

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define N 100000

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

void shaker_sort(int number[]){
    int i, j;
    int right, left;
    int sorted = 0;

    left = 0;
    right = N - 1;

    while (left != right){
        for(i = left; i < right; i++){
            if(number[i] > number[i + 1]){
                swap(&number[i], &number[i + 1]);
                sorted = i;
            }
        }
        right = sorted;

        for(i=right; i > left; i--){
            if(number[i] < number[i-1]){
                swap(&number[i-1], &number[i]);
                sorted = i;
            }
        }
        left = sorted;
    }
}



int main(int argc, char *argv[])
{
    FILE *fp1;
    FILE *fp2;
    int number[N];
    int i = 0;

```

```
48      if(argc != 2){
49          puts("Parameter Error");
50          return 0;
51      }
52
53      if ((fp1 = fopen(argv[1], "r")) == NULL){
54          puts("NO FILE WITH THAT NAME");
55          return 0;
56      }
57
58      for(i = 0; i < N; i++){
59          fscanf(fp1, "%d", &number[i]);
60      }
61
62      fclose(fp1);
63  /*---------- プログラム処理 ----------*/
64      clock_t begin = clock();
65      shaker_sort(number);
66      clock_t end = clock();
67
68      fp2 = fopen("./output000.dat", "w");
69      for(i = 0; i < N; i++){
70          fprintf(fp2, "%d\n", number[i]);
71      }
72      fclose(fp2);
73
74      puts("Done!");
75      printf("The elapsed time is %lf sec\n", (double)(end-begin)/CLOCKS_PER_SEC);
76      return 0;
77  }
```

5.2 動作

シェーカーソートの動作は以下通りである.

1. right, left 変数を用意し, right に N-1 で left に 0 を入れる
2. 左から始まり, A 要素と隣り合う次の B 要素と比較し, もし B が A より小さかったらスワップし, right のところまでこの動作をする.
3. 終わったら, 右側に何番までスワップするか新しい値が right 変数に入れる.
4. 動作の 2 番と同じ動作が行うが, 逆で右から始まり, 終わったら, 左側に何番までスワップするか新しい値が left 変数に入れる.
5. 動作 2 番目から 4 番目までを繰り返し, right 変数と left 変数の値が同じになったら, ソートが終わり.

5.3 時間計算量

シェーカーソートはバブルソートのと同じく最悪状況で時間計算量は $O(n^2)$ である. ただバブルソートの無駄な部分を少し省いたから, バブルソートより早いということになる.

以下の表 5.3 は上のコードで実行したソート計算時間だ (CPU は intel i7 7^{th} Gen, RAM は 8GB). 表 5.3 を見ると, バブルソートの計算時間より少し早かったということがわかる. 降順データに対するソーティングは一番時間がかかるが, データ 1-3 番目まではデータ 5 とあまり違わない.

表 5: 計算時間 (シェーカーソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	21.773718	21.781498	21.866999	0.000227	24.272765

データ	data6	data7	data8	data9
計算時間 (秒)	13.876875	20.052660	21.909268	22.275070

6　クイックソート

データの比較と交換回数が非常に少ないのが特徴で, 一般的なばらばらデータ（ランダムに散らばっているデータ）に対して, 最も効率良く並べ替えを実行します. クイックソートのイメージは図 5 である. クイックソートは, 実用上もっとも高速であるとされている並べ替えアルゴリズムで, 多くのプログラムで利用されている.

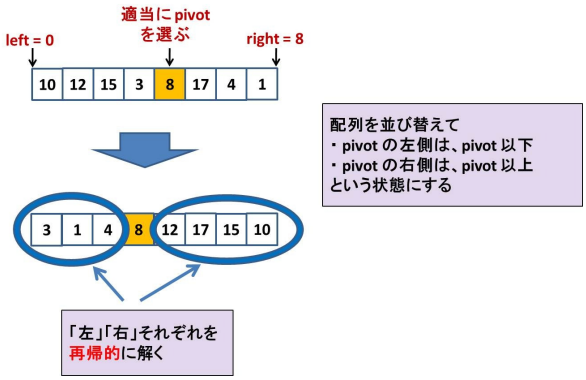

図 5: クイックソートイメージ

6.1　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<time.h>
#define N 100000

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

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
```

```
18      {
19          if (arr[j] < pivot)
20          {
21              i++;
22              swap(&arr[i], &arr[j]);
23          }
24      }
25      swap(&arr[i + 1], &arr[high]);
26      return (i + 1);
27  }
28
29  void quick_sort(int arr[], int low, int high)
30  {
31      if (low < high)
32      {
33          int pi = partition(arr, low, high)
34
35          quick_sort(arr, low, pi - 1);
36          quick_sort(arr, pi + 1, high);
37      }
38  }
39
40
41  int main(int argc, char *argv[])
42  {
43      FILE *fp1;
44      FILE *fp2;
45      int number[N];
46      int i = 0;
47
48      if(argc != 2){
49          puts("Parameter Error");
50          return 0;
51      }
52
53      if ((fp1 = fopen(argv[1], "r")) == NULL){
54          puts("NO FILE WITH THAT NAME");
55          return 0;
56      }
57
58      for(i = 0; i < N; i++){
59          fscanf(fp1, "%d", &number[i]);
60      }
61
62      fclose(fp1);
63  /*---------- プログラム処理 ----------*/
64      clock_t begin = clock();
65      quick_sort(number, 0, N-1);
66      clock_t end = clock();
67
68      fp2 = fopen("./output000.dat", "w");
69      for(i = 0; i < N; i++){
70          fprintf(fp2, "%d\n", number[i]);
71      }
72      fclose(fp2);
73
74      puts("Done!");
75      printf("The elapsed time is %lf sec\n", (double)(end-begin)/CLOCKS_PER_SEC);
76      return 0;
77  }
```

6.2 動作

クイックソートの動作は以下通り行われている.

1. partition 関数が pivot のインデックスを返して, partition 関数の中に, pivot の値が決められ, pivot の値より小さかったら左にインデックス i とスワップし, i を 1 増やす.
2. 1 番目の動作は high 変数の値まで繰り返す. 最後に pivot のインデックスと値がインデックス i のところに入れる.
3. pivot に区切られたデータが動作 1-2 が行って, 繰り返し, ソート完了.

6.3 時間計算量

最悪の場合は分割が 1 と $N-1$ に行われつづけた場合で、この場合、クイックソートの分割の深さは N となり, それぞれのデータの深さは N, N, $N-1$, $N-2$, ..., 3, 2 となる. したがって, この場合のクイックソートの計算時間は,

$$N+N+N-1+N-2+\cdots+3+2=\frac{N^2}{2}+\frac{3N}{2}-1 \tag{1}$$

となり, 時間計算量は $O(N^2)$ となる.

また, 平均的な場合として, すべて異なる一様分布のデータに対しては, $O(\mathrm{N}\log N)$ になることが知られている.

以下の表 6.3 は上のコードで実行したソート計算時間だ (CPU は intel i7 7^{th} Gen, RAM は 8GB). 表 6.3 を見ると, 乱数データに対しては, 計算時間一番早いということがわかる. 昇順データと降順データが最悪の場合でるから, 一番遅いということがわかる.

表 6: 計算時間 (クイックソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	0.014566	0.014927	0.015274	10.184581	10.126059

データ	data6	data7	data8	data9
計算時間 (秒)	2.012471	0.014439	0.014676	0.016183

7　ソートアルゴリズムの比較

表 7 はソートアルゴリズムの計算時間を比べる表である.

表 7: 計算時間の比較

	バケツソート	挿入ソート	バブルソート	シェーカーソート	クイックソート
data1	0.001350	5.814234	31.531213	21.773718	0.014566
data2	0.001547	5.791112	31.511556	21.781498	0.014927
data3	0.001357	5.762070	31.479725	21.866999	0.015274
data4	0.001740	0.000307	0.000236	0.000227	10.184581
data5	0.001389	11.566258	23.110471	24.272765	10.126059
data6	0.001422	5.780712	16.938872	13.876875	2.012471
data7	0.001358	5.808727	27.564562	20.052660	0.014439
data8	—	5.770340	31.556714	21.909268	0.014676
data9	—	5.934736	31.548570	22.275070	0.016183

表 7 を見ると, こんな事をまとめることができる

1. バケットソートはどんなソートアルゴリズムより早いということがわかる. だが, メモリが非常にたくさん使ってしまう.
2. 少ないメモリを使って, 早くソートするはクイックソートである.
3. バブルソートとシェーカーソートは比較回数が多いため, 計算が遅いということがわかる. 同じ計算時間量 $O(n^2)$ のなかで, 挿入ソートが一番早い.
4. マイナスと少数があっても計算時間が大きい変化がない (遅くならない).

8　他のソートアルゴリズム (ツリーソート)

ツリーソートはソーティングをするのに, 二分木が作られ, 次に、要素がソートされた順序で出てくるように, ツリーを (順番に) トラバースするアルゴリズムである. 図 6 ではソートの動作が描いてある.

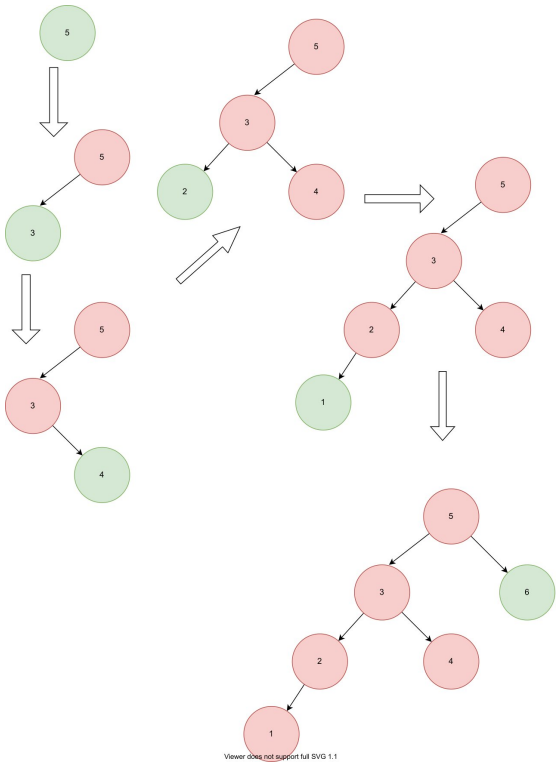

図 6: ツリーソートのイメージ

8.1　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 <time.h>
#define N 100000

struct btreenode
{
    struct btreenode *leftchild;
    double data;
    struct btreenode *rightchild;
};

void insert(struct btreenode **sr, double num)
{
    if (*sr == NULL)
    {
        *sr = malloc (sizeof(struct btreenode));
```

```
18          (*sr) -> leftchild = NULL;
19          (*sr) -> data = num;
20          (*sr) -> rightchild = NULL;
21      }
22      else
23      {
24          if (num < (*sr) -> data)
25              insert(&((*sr) -> leftchild), num);
26          else
27              insert(&((*sr) -> rightchild), num);
28      }
29  }
30
31  void inorder(struct btreenode *sr, FILE *fp2)
32  {
33      if(sr != NULL)
34      {
35          inorder(sr -> leftchild, fp2);
36          fprintf(fp2, "%lf\n", sr -> data);
37          inorder(sr -> rightchild, fp2);
38      }
39  }
40
41  int main(int argc, char *argv[])
42  {
43      FILE *fp1;
44      FILE *fp2;
45      int i;
46      double number[N];
47      struct btreenode *bt;
48      bt = NULL;
49
50      if(argc != 2){
51          puts("Parameter Error");
52          return 0;
53      }
54
55      if ((fp1 = fopen(argv[1], "r")) == NULL){
56          puts("NO FILE WITH THAT NAME");
57          return 0;
58      }
59
60      for(i = 0; i < N; i++){
61          fscanf(fp1, "%lf", &number[i]);
62      }
63
64      fclose(fp1);
65      fp2 = fopen("./output000.dat", "w");
66      clock_t begin = clock();
67      for (i = 0 ; i < N ; i++){
68          insert(&bt, number[i]);
69      }
70      inorder(bt, fp2);
71      clock_t end = clock();
72      fclose(fp2);
73      printf("%lf\n", (double)(end - begin)/CLOCKS_PER_SEC);
74      return 0;
75  }
```

8.2 時間計算量

平均的なケースでは, 二分木に n ノードを挿入する時間の複雑さは $O(\mathrm{n}\log n)$ のオーダーだ. これは, 形成される二分木がバランスのとれた二分木である場合に発生する. したがって, 時間の複

雑さは $O(\mathrm{nlog}\,n)$ である.

最悪の場合は, 配列がソートされ、最大高さ $O(\mathrm{n})$ の不均衡な 2 値探索木が形成される. 高さが $\log n$ の通常の二分木の場合の探索時間は $O(\log n)$ であるのに対し, 探索には $O(\mathrm{n})$ 時間, 挿入には $O(n^2)$ が必要. 最悪の場合の時間的複雑さは $O(n^2)$ である.

ベストケースは形成された二値探索木が均衡している場合である. 最良の時間複雑度は $O(\mathrm{nlog}\,n)$. これは平均ケースの時間複雑度と同じ.

実験結果は表 8.2 以下の通りだ. 一番遅かったはデータ 4 である. ソートされるデータはツリーソートでソートされるとき, 一直線になってしまって, ソートする時間計算量は $O(n^2)$ になるから, 遅くなるわけだ. 少数やマイナスの整数に対しては, 処理時間に遅くなるケースがない. data5 と data6 はバブルソートのケースと同じ**分岐予測**で説明でき, ページ 9 を参照する.

表 8: 計算時間 (ツリーソート)

データ	data1	data2	data3	data4	data5
計算時間 (秒)	0.0316187	0.0316239	0.4148764	30.579234	15.141092

データ	data6	data7	data8	data9
計算時間 (秒)	14.654189	0.0312984	0.0316928	0.0316987

9 新しいデータを挿入する処理時間

新しいデータを挿入する処理時間について, ツリーソートとクイックソートを比較する. この実験で新しいデータを挿入する数は 100 個, 1000 個, 10000 個, 50000 個だ. その値は 0-n 個で昇順データだ. ソートされたデータに対して, 最後の要素後に入れる. 比較結果は表 9 以下の通りだ.

表 9: 処理時間 (秒)

データ数	100 個	1000 個	10000 個	50000 個
クイックソート	0.112925	0.105410	1.350866	17.586362
ツリーソート	0.016001	0.016357	0.018030	0.031383

クイックソートはソート処理時間はとても早いが, 新しいデータを入れて, またソートすると処理時間が遅い. 一方, ツリーソートはとても早いということがわかる.

10　発表感想

今回の発表はミスはたくさんあって, 表のデータに単位を書くやスライドをめくるのは早すぎるなどだ. 次回のプレゼントがあればこういうミスをもっと気をつける. 質問を答え方もあまりはっきりではないと思う. 良かったのは発表が指定された時間で発表できることだ. 質問者はちゃんと発表を聞いて, 気付かないところまで質問されてとても良かったと思う.

参考文献

[1] 分岐予測 https://ja.wikipedia.org/wiki/分岐予測