1　マージソート

マージソートは、二つの整列済みの配列をマージ (結合) することにより、一つの配列をソートするアルゴリズムである。以下に具体的な手順を示す。

1. 配列 data を真ん中で二つの配列 a と b に分割する。
2. この a と b を更に 2 分割し、要素が 1 になるまで繰り返す
3. 分割した要素をソートしながらマージする
4. 最終的に整列された 1 つの配列ができる

この手順の具体的な例を図 1.1 に示す。

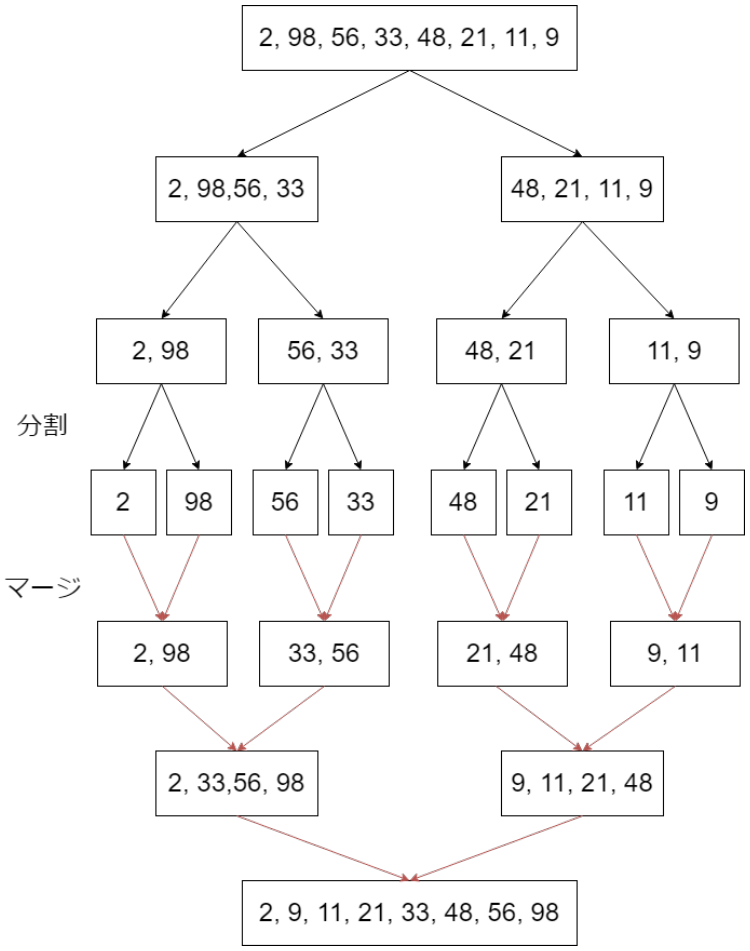

図 1.1: マージソートの原理

以下に C 言語で実装したマージソートのソースコードを示す。

まず merge_sort 関数にてコピーする用の配列を malloc で動的に確保している。その後、merg_sort_main 関数を再帰的に呼び出すことにより、配列を分割し、ソートを行っている。

```
1 void merge_sort(int *data, int low, int high){
2     int *workspace;
3     workspace = (int *)malloc(sizeof(int)*MAX);
4     merge_sort_main(data, low, high, workspace);
5     free(workspace);
6 }
```

```
7
8  void merge_sort_main(int *data, int low, int high, int *workspace){
9      int mid, i, j, k;
10
11     if(low >= high)
12         return;
13
14     mid = (low + high)/2;
15
16     merge_sort_main(data, low, mid, workspace);
17     merge_sort_main(data, mid+1, high, workspace);
18
19     for(i = low; i <= mid; i++)
20         workspace[i] = data[i];
21
22     for(i = mid+1, j= high; i <= high; i++, j--)
23         workspace[i] = data[j];
24
25     i = low; j = high;
26     for(k = low; k <= high; k++)
27         if(workspace[i] <= workspace[j])
28             data[k] = workspace[i++];
29         else
30             data[k] = workspace[j--];
31 }
32
33 /*
34 渡した配列:int data[] = {55, 13, 3, 45, 74, 87, 46, 30}
35 実行結果: 3, 13, 30, 45, 46, 55, 74, 87
36 */
```

2　ヒープソート

ヒープソートはヒープ (半順序木) というデータ構造を利用したソートアルゴリズムである。ヒープとは図のように、上のノード (親) に下のノード (子) が 2 つもしくは 1 つだけくっついている構造 (二分木) のうち、どこでも親の方が子よりも値が小さい (もしくは大きい) ものをさす。ヒープでは根本は全ノードの中で最小 (もしくは最大) になるため、根本を取り出し、またヒープを構築するという手順を繰り返すことによりソートを行うというのがヒープソートのアルゴリズムである。以下に具体的な手順を示す。

1. 整列したい配列でヒープを構築する
2. 根本の値を取出す
3. 残った配列でまたヒープを構築する
4. 以後、配列の要素数が 1 になるまで 2〜3 の手順を繰り返す

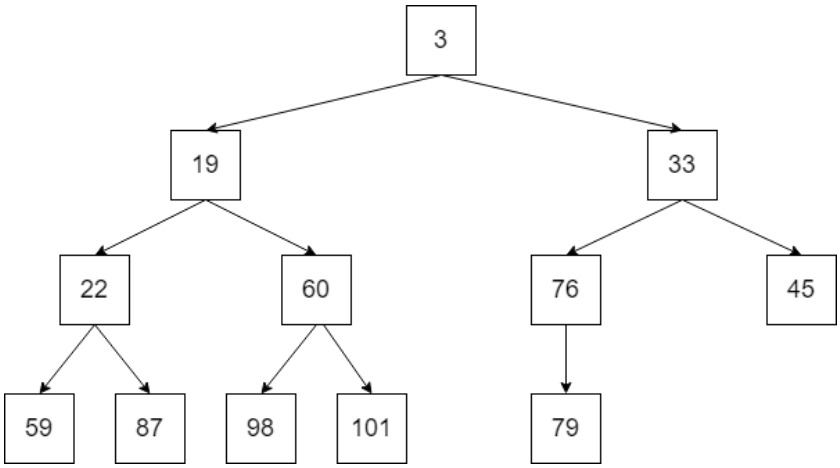

図 2.1: ヒープ (半順序木) の例

以下に C 言語でヒープソートを実装したソースコードを示す。downheap 関数が配列からヒープを構築する関数であり、heap_sort 関数が実際にヒープソートを行う関数である。

まず heap_sort 関数が main 関数で呼び出されると、heap_sort 関数が downheap 関数を呼び出す。次に downheap 関数がヒープを構築し、heap_sort 関数に戻ってくる。戻ってきた heap_sort 関数は配列の最初の要素、つまり最小の値を配列の一番後ろと交換する。また次の downheap 関数は最小の値が入った配列の一番後ろの要素は無視して、配列の要素数-1 の部分でヒープを構築するようになる。この処理を要素数が 1 になるまで行っている。

```
1  void downheap(int from, int to, int heap_data[]){
2      int i, j;
3      int val;
4
5      val = heap_data[from];
6
7      i = from;
8      while(i <= to/2){
9          j = i* 2;
10         if(j+1 <= to && heap_data[j] > heap_data[j+1])
11             j++;
12
```

```
13          if(val <= heap_data[j])
14              break;
15
16          heap_data[i] = heap_data[j];
17          i = j;
18      }
19      heap_data[i] = val;
20  }
21
22  void heap_sort(int heap_data[], int n){
23      int i, tmp;
24
25      for(i = n/2; i >= 1; i--)
26          downheap(i, n, heap_data);
27
28      for(i = n; i >= 2; i--){
29          tmp = heap_data[1];
30          heap_data[1] = heap_data[i];
31          heap_data[i] = tmp;
32          downheap(1, i-1, heap_data);
33      }
34  }
```

3　ソートの安定性

安定なソートアルゴリズムとは、配列内に同値の要素が複数あった場合、ソートの前後でそれらの位置関係が変わらないものを指す。逆に安定でないソートアルゴリズムとはソートの前後で位置関係の保証がされていないもののことである。以下の図 3.1 に安定と安定でないものの例を示す。

便宜上、1 つ目の 4 を 4(a)、2 つ目の 4 を 4(b) とする。安定なソートアルゴリズムではソート後であっても 1 つ目の 4 が 4(a)、2 つ目の 4 が 4(b) となっているが、安定でないソートアルゴリズムは 1 つ目の 4 が 4(b)、2 つ目の 4 が 4(a) と入れ替わってしまっている。

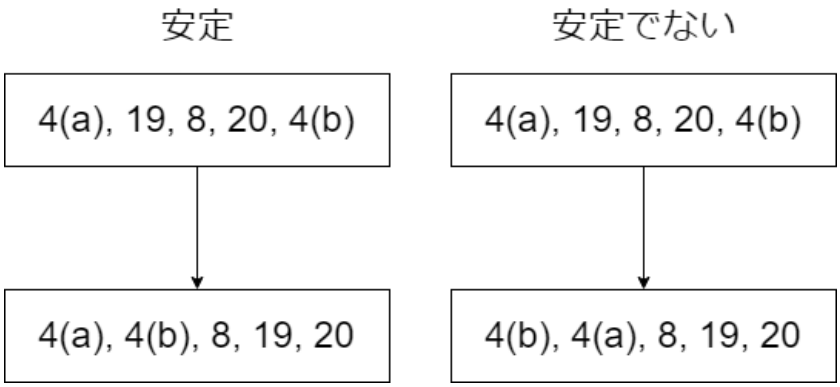

図 3.1: 安定なソートアルゴリズムと安定でないソートアルゴリズムの例

また以下にこれまで学習した 7 つのソートアルゴリズムが安定か安定でないかをまとめた表を示す。

表 3.1: ソートアルゴリズムの安定性

ソートアルゴリズム	安定であるか
バブル	○
選択	×
挿入	○
シェル	×
クイック	×
マージ	○
ヒープ	×

4　計算量

計算量とはアルゴリズムの性能を表現するものであり、入力の大きさ n に応じて、そのアルゴリズムの実行時間がどのように増えていくを示す。計算量はランダウの記号 O を用いたオーダー表記で表される。ランダウの記号とは、例えば $f(x) = 3x^2 + 2x + 9$ という関数があった場合、関数の値に一番大きく影響する x^2 のみを抜き出し、$f(x) = O(x^2)$ とするものである。

単純なソートアルゴリズムの 3 つにおいて、一番時間を要する動作は for 文の二重ループによって数の比較を行うときである。for 文 1 つは入力の大きさ n に比例するため、for 文の 2 重ループは n^2 に比例する。したがってこれらの計算量は $O(n^2)$ となる。

以下にこれまで学習した 7 つのソートアルゴリズムの計算量をまとめた表を示す。

ソートアルゴリズム	計算量
バブル	$O(n^2)$
選択	$O(n^2)$
挿入	$O(n^2)$
シェル	$O(n^{1.25\sim1.5})$
クイック	$O(n \log n)$
マージ	$O(n \log n)$
ヒープ	$O(n \log n)$

5　スタック、キュー、デック

スタックはデータ構造の一種であり、データの挿入と取り出しが共にリストの先頭で行われる。スタックは LIFO(Last In First Out) とも呼ばれ、要素が追加された後、要素を取り出すときは、一番最後に追加された要素から取り出される。この要素を追加する動作を push、要素を取出す動作は pop と呼ばれる。

キューもデータ構造の一種であり、データの挿入が一方の端で行われ、取り出しがその反対側で行われる。キューは FIFO(First In First Out) とも呼ばれ、追加された要素を取り出すときは、スタックとは反対に一番最初のものが取り出される。要素を追加する動作はエンキュー、要素を取り出す動作をデキューという。

デックもまたデータ構造の一種である。デックはリストの両端で要素の挿入および取り出しが可能なものであり、スタックとキューの両方の機能を持つものといえる。

以下にスタックを C 言語で実装したソースコードを示す。

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

#define STACK_SIZE 100

typedef long ELEM;
ELEM stack[STACK_SIZE];
int n;

void error(char *s){
    fprintf(stderr, s);
    exit(1);
}

void init(){
    n = 0;
}

void push(ELEM x){
    if(n >= STACK_SIZE)
        error("stack overflow\n");
    stack[n++] = x;
}

ELEM pop(){
    if(n <= 0)
        error("stack underflow\n");
    return stack[--n];
}

int empty(){
    return n == 0;
}

/*
push(30);
push(25);
x = pop;
```

```
printf("%d\n", x);

実行結果
25
*/
```

次にキューを C 言語で実装したソースコードを示す。

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

#define QUEUE_SIZE 100
#define next(a) (((a) + 1)%QUEUE_SIZE)

typedef long ELEM;
ELEM queue[QUEUE_SIZE];
int front, rear;

void error(char *s){
    fprintf(stderr, s);
    exit(1);
}

void init(){
    front = rear = 0;
}

void enqueue(ELEM x){
    if (next(rear) == front)
        error("Queue is full.\n");
    queue[rear] = x;
    rear = next(rear);
}

ELEM dequeue(){
    ELEM x;

    if(front == rear)
        error("Queue is empty.\n");

    x = queue[front];
    front = next(front);
    return x;
}

int empty(){
    return front == rear;
}

/*
enqueue(30);
enqueue(25);
x = dequeue;
printf("%d\n", x);
```

```
48
49 実行結果
50 30
51 */
```

6　7つのソートアルゴリズムの比較

以下にこれまで学習した7つのソートアルゴリズムを実行し、配列の要素数に応じてかかった時間をプロットしたグラフを図6.1に示す。

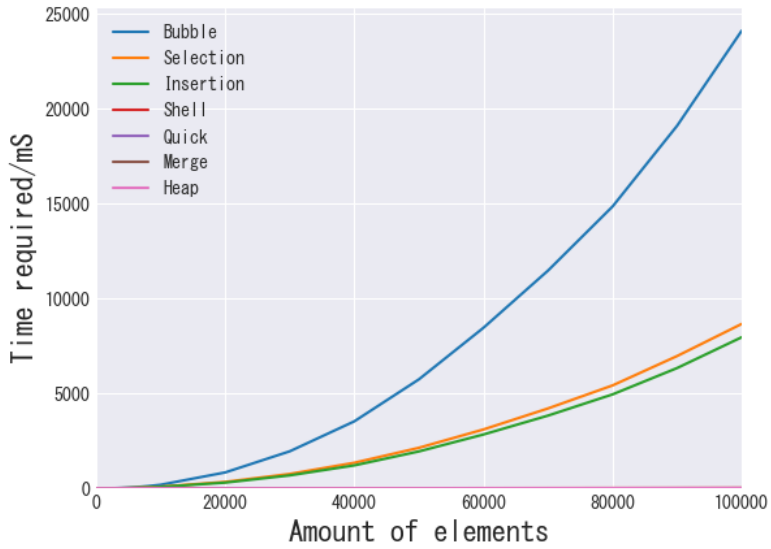

図 6.1: 7つのソートアルゴリズムの計算時間

また単純なアルゴリズムを除いた4つのソートアルゴリズムのみをプロットしたグラフを図6.2に示す。

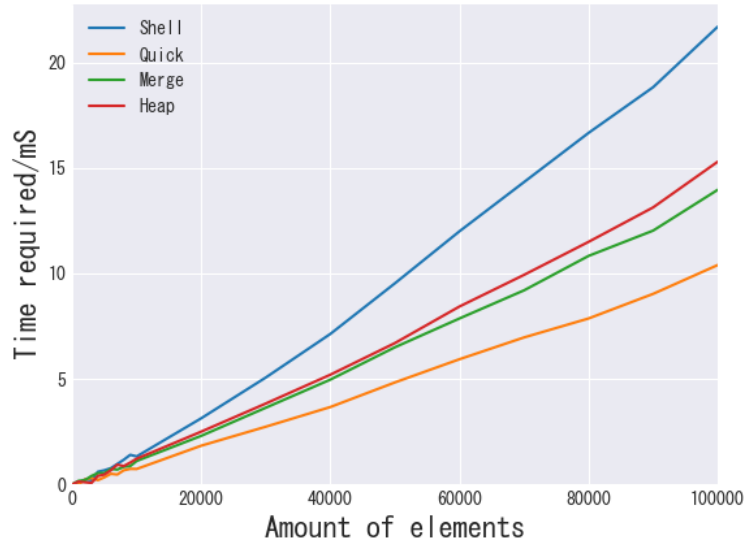

図 6.2: 4つのソートアルゴリズムの計算時間

以下にこれらのグラフを出力するPythonのソースコードを示す。

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
```

```
4
5  exec_time = [[0 for i in range(7)]]
6
7  for i in range(1, 10):
8      exec_num = i*1000
9      file_name = (f'result_{exec_num}.csv')
10     df = pd.read_csv(file_name, header=None, usecols=[0, 1, 2, 3, 4, 5, 6])
11     new_exec_time = df.mean().values
12     exec_time =  np.vstack((exec_time, new_exec_time))
13
14 for i in range(1, 11):
15     exec_num = i*10000
16     file_name = (f'result_{exec_num}.csv')
17     df = pd.read_csv(file_name, header=None, usecols=[0, 1, 2, 3, 4, 5, 6])
18     new_exec_time = df.mean().values
19     exec_time =  np.vstack((exec_time, new_exec_time))
20
21 exec_time = np.array(exec_time).T
22 list = [0, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000,
       20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000]
23
24 plt.style.use('seaborn-dark')
25 plt.rcParams["font.family"] = "MS Gothic"
26 plt.plot(list, exec_time[0], label='Bubble')
27 plt.plot(list, exec_time[1], label='Selection')
28 plt.plot(list, exec_time[2], label='Insertion')
29 plt.plot(list, exec_time[3], label='Shell')
30 plt.plot(list, exec_time[4], label='Quick')
31 plt.plot(list, exec_time[5], label='Merge')
32 plt.plot(list, exec_time[6], label='Heap')
33
34 plt.xlabel('Amount of elements', fontsize=16)
35 plt.ylabel('Time required/mS', fontsize=16)
36 plt.legend(loc='upper left')
37 plt.xlim(0, list[19])
38 plt.ylim(0, )
39 plt.grid()
40 plt.show()
```

また以下に7つのソートアルゴリズムの計算時間を記録し、CSV ファイルに書き込む C 言語のソースコードを示す。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <math.h>
5  #include "sort.h"
6
7  int main(){
8      int i,j,k;
9      char file_name[50];
10
11     //実行回数
12     int num = 30;
13
```

```
14      //clock関数の戻り値格納用
15      clock_t start_clock, end_clock;
16
17      //データのメモリを動的確保
18      int *data, *heap_data;
19      data = (int *)malloc(sizeof(int)*MAX);
20      heap_data = (int *)malloc(sizeof(int)*MAX);
21
22      srand((unsigned int)time(NULL));
23
24      //バブル,選択,挿入,シェルソートの関数ポインタの配列
25      void(*simple_sort[SIMPLE_FUNC_NUM])(int *data) = {
26          bubble_sort,
27          selection_sort,
28          insertion_sort,
29          shell_sort,
30      };
31
32      //クイック,マージソートの関数ポインタの配列
33      void(*comp_sort[COMP_FUNC_NUM])(int *data, int first, int last) = {
34          quick_sort,
35          merge_sort,
36      };
37
38      //ファイルのオープン
39      FILE *fp;
40      sprintf(file_name, "result_%d.csv", MAX);
41      fp = fopen(file_name, "a+");
42      if(fp == NULL){
43          printf("Fail to open this file.\n");
44          return 1;
45      }
46      else{
47          printf("Success to open this file. \n");
48      }
49
50      for(i=0; i<num; i++){
51          //バブル,選択,挿入,シェルソート
52          for(j=0; j<SIMPLE_FUNC_NUM; j++){
53              //整列用データの作成
54              for(k=0; k<MAX; k++){
55                  data[k] = 0 + rand()%1000;
56              }
57
58              start_clock = clock();
59              simple_sort[j](data);
60              end_clock = clock();
61              fprintf(fp, "%f,", (double)(end_clock - start_clock)/
                    CLOCKS_PER_SEC*pow(10, 3));
62          }
63
64          //クイック,マージソート
65          int high;
66          for(j=0; j<COMP_FUNC_NUM; j++){
```

```
67              if(j == 1)
68                  high = MAX - 1;
69              else high = MAX;
70
71              //整列用データの作成
72              for(k=0; k<MAX; k++){
73                  data[k] = 0 + rand()%1000;
74              }
75
76              start_clock = clock();
77              comp_sort[j](data, 0, high);
78              end_clock = clock();
79              fprintf(fp, "%f,", (double)(end_clock - start_clock)/
                    CLOCKS_PER_SEC*pow(10, 3));
80          }
81
82          //ヒープソート
83          //整列用データの作成
84          for(j=0; j<MAX+1; j++)
85              heap_data[j] = 0 + rand()%1000;
86
87          start_clock = clock();
88          heap_sort(heap_data, MAX);
89          end_clock = clock();
90          fprintf(fp, "%f,", (double)(end_clock - start_clock)/CLOCKS_PER_SEC*
                pow(10, 3));
91
92          fprintf(fp, "\n");
93      }
94      fclose(fp);
95
96      free(data);
97      free(heap_data);
98
99      return 0;
100 }
```