1　バブルソート

1.1　原理

バブルソートは昇順に並べる場合、配列の一番後ろの要素から順番に一つ前の要素と比較していき、もし一つ前の要素よりも値が小さければ入れ替えるという動作を繰り返すソートアルゴリズムである。

一回目の動作では一番小さな値が先頭にくる。二回目の動作では二番目に小さな値が先頭から二つ目のところにくる。このように、動作を一回行うごと小さな値が決まっていき、配列の要素数-1 回目の動作で一番後ろとその一つ前の要素の比較され、配列全体のソートが完了する。

1.2　例題

例として int 型の配列 `data[]={61, 4, 36, 11, 9, 77}`の昇順ソートを考える。整列し終わるまでの過程を図 1.1 に示す。赤文字は位置が確定したことを表す。

一回目の動作では、まず一番後ろの 77 とその一つ前の 9 が比較され、9 の方が値が小さいため入れ替えは起こらない。次に 9 と 11 が比較され、9 の方が小さいため 9 と 11 の位置が入れ替わる。同様に次の 36 と 9 の位置も交換される。しかしその次の 4 と比較すると 9 の方が大きいため、4 と 9 の位置は入れ替わらない。最後に 4 と先頭の 61 が比較され、4 の方が小さいため位置が入れ替わる。これによって一番小さな値である 4 が先頭に来た。この動作を繰り返し行なう。

配列の要素数ー 1 である 5 回目の動作時に 61 と 77 を比較しソートが終了する。これは一番最後の 77 は今までの動作で間接的に全ての要素と比較されたことになり、6 回目の動作を行う必要がないためである。

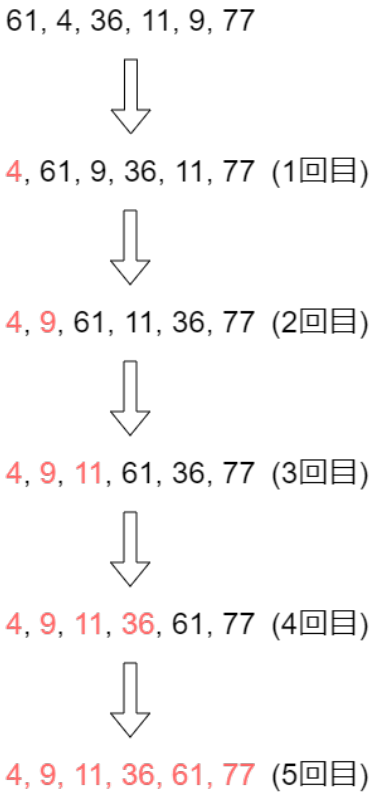

図 1.1: バブルソートの過程

1.3　ソースコードと実行結果

バブルソートを実装した C 言語のソースコードを図 1.2 に示す。

```c
#include <stdio.h>

void bubble_sort(int a[], int n);

int main(){
    int a[10] = {20, 6, 55, 74, 3, 45, 13, 87, 46, 30};
    int n = sizeof(a)/sizeof(int);
    int i;

    bubble_sort(a, n);

    for(i = 0; i < n; i++){
        printf("%d ", a[i]);
    }
}

void bubble_sort(int a[], int n){
    int i,j,t;

    for(i = 0; i < n-1; i++){
        for(j = n - 1; j > i; j--){
            if (a[j-1] > a[j]){
                t = a[j]; a[j] = a[j-1]; a[j-1] = t;
            }
        }
    }
}
```

図 1.2: バブルソートのソースコード

また以下にこのソースコードの実行結果を示す。

```
(base) PS C:\Users\shu\Algo> ./a.exe
an array data={61, 4, 36, 11, 9, 77} will be sorted  in ascending
order by Bubble Sort.

4 9 11 36 61 77
```

2　選択ソート

2.1　原理

選択ソートは、まず配列の中から一番小さな値を探し出して、配列の先頭要素と入れ替える。次に二番目に小さい値を探し出して、先頭から二番目の要素と入れ替えるという手順を繰り返すソートアルゴリズムである。バブルソートと同様、動作は配列の要素数ー 1 回分行われる。

配列の中から最小の値を探し出すのに、結局全ての要素と比較する必要があるため比較回数自体はバブルソートと同じである。しかし交換の回数は減るため、交換にコストがかかる場合はこちらのほうが有効である。

2.2　例題

バブルソートと同様、int 型の配列 `data[]={61, 4, 36, 11, 9, 77}`の昇順ソートを考える。整列し終わるまでの過程を図 2.1 に示す。赤文字は位置が確定したことを表す。

一回目では配列の中で最小である 4 が先頭の 61 と交換される。二回目は 9 と 61、三回目は 11 と 36 が交換される。四回目では 36 が本来いるべき場所にすでにいるため、交換は行われない。最後の五回目も同様の理由から交換が行われずにソートが終了する。

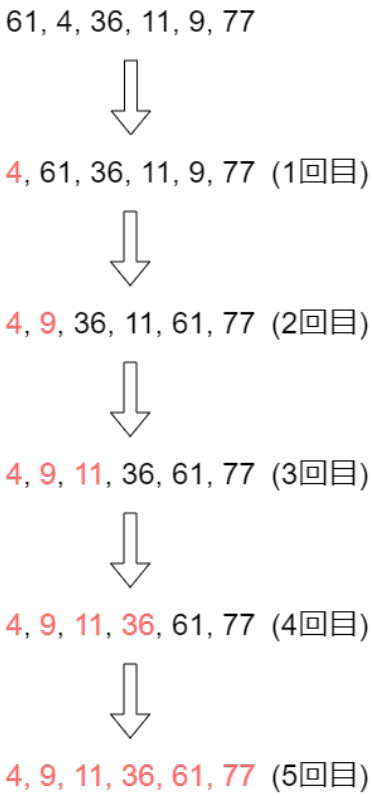

図 2.1: 選択ソートの過程

2.3　ソースコードと実行結果

選択ソートを実装した C 言語のソースコードを図 2.2 に示す。

```c
#include <stdio.h>

void selection_sort(int a[], int n);

int main(void){
    int i;
    int a[] = {3, 11, 1000, 98, 1, 8};
    int n = sizeof(a)/sizeof(int);

    selection_sort(a, n);

    for (i = 0; i < n; i++){
        printf("%d, ", a[i]);
    }
}

void selection_sort(int a[], int n){
    int i, j, t, lowest, lowkey;

    for (i = 0; i < n - 1; i++){
        lowest = i;
        lowkey = a[i];
        for (j = i + 1; j < n; j++){
            if (a[j] < lowkey){
                lowest = j;
                lowkey = a[j];
            }
        }
        t = a[i];
        a[i] = a[lowest];
        a[lowest] = t;
    }
}
```

図 2.2: 選択ソートのソースコード

また以下にこのソースコードの実行結果を示す。

```
(base) PS C:\Users\shu\Algo> ./a.exe
an array data={61, 4, 36, 11, 9, 77} will be sorted  in ascending
order by Section Sort.

4 9 11 36 61 77
```

3　挿入ソート

挿入ソートは昇順に並べる場合、先頭から一つずつ、その数がいるべき適切な場所へ挿入していく手順を繰り返すソートアルゴリズムである。

具体的には先頭の値はそのまま固定で、その次の数から固定されている値と比較し、自分がいる適切な場所に挿入していく。

バブルソートや選択ソートと異なり全ての要素と比較しなくて済む。またある程度既に整列してある配列に対して特に有効であるという特徴がある。

3.1　例題

バブルソートや選択ソートと同様、int 型の配列 `data[]={61, 4, 36, 11, 9, 77}`の昇順ソートを考える。整列し終わるまでの過程を図 3.1 に示す。赤文字は位置が確定したことを表す。

先頭の 61 を基準にし、後ろの 4, 36, 11, 9, 77 が順番に適切な場所に挿入され、ソートが完了している。

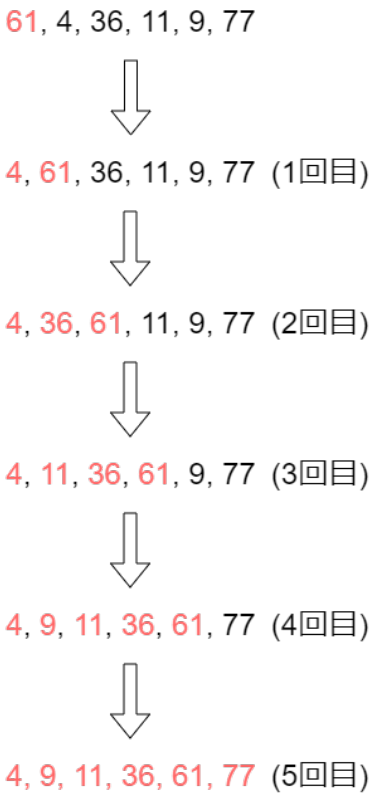

図 3.1: 挿入ソートの過程

3.2　ソースコードと実行結果

挿入ソートを実装した C 言語のソースコードを図 3.2 に示す。

```c
#include <stdio.h>

void insertion_sort(int a[], int n);

int main(){
    int a[] = {20, 6, 55, 74, 3, 45, 13, 87, 46, 30};
    int n = sizeof(a)/sizeof(int);
    int i;

    insertion_sort(a, n);

    for(i = 0; i < n; i++){
        printf("%-3d", a[i]);
    }

    return 0;
}

void insertion_sort(int a[], int n){
    int i, j, t;

    for(i = 1; i < n; i++){
        j = i;
        while(j >= 1 && a[j-1] > a[j]){
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;

            j--;
        }
    }
}
```

図 3.2: 挿入ソートのソースコード

また以下にこのソースコードの実行結果を示す。

```
(base) PS C:\Users\shu\Algo> ./a.exe
an array data={61, 4, 36, 11, 9, 77} will be sorted  in ascending
order by Insertion Sort.

4 9 11 36 61 77
```

4　シェルソート

4.1　原理

シェルソートは挿入ソートの「ある程度整列されている配列に対して有効的である」という特徴を活かして改良されたソートアルゴリズムである。挿入ソートは最初から全ての要素を一つづつ適切な場所に挿入していくが、シェルソートではまず配列を飛ばし飛ばしの要素の組に分ける。例えば要素数 9 の配列に対して、3 つ飛ばしと決めたら、1 つ目と 4 つ目と 7 つ目の組、2 つ目と 5 つ目と 8 つ目の組、3 つ目と 6 つ目と 9 つ目の組と分ける。次にこの分けた組内で挿入ソートを行ったのち、これらを一旦一つの配列に戻す。すると最初の状態からある程度は整列がされている。この動作を組の間隔を段々と狭めながら行っていき、最後には間隔が 1、つまり単純な挿入ソートを行ない整列が完了する。

4.2　例題

int 型の配列 `data[]={9, 24, 1, 99, 87, 56, 11, 77, 32}`について考える。これを間隔 h=3 で分けてソートを行う手順を図 4.1 に示す。分けた組み合わせは (9, 99, 11), (24, 87, 77), (1, 56, 32) になる。それぞれの組に挿入ソートを行うと、(9, 11, 99), (24, 77, 87), (1, 32, 56) になり、元の配列に戻すと `data[] = {9, 24, 1, 11, 77, 32, 99, 87, 56}`になる。最初の配列の状態と比較すると、大雑把には整列が行われていることが分かる。

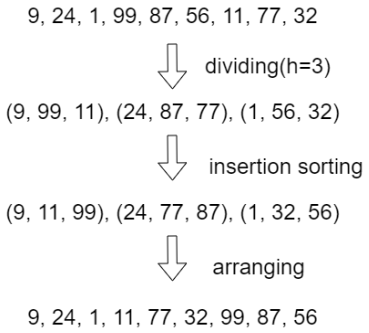

図 4.1: h=3 でのシェルソートの過程

次にこの配列を h=2 で分割しソートする手順を図 4.2 に示す。基本的な手順は h=3 の時と同じである。違いとしては間隔を短くした h=2 の方がより細かくソートされるということである。

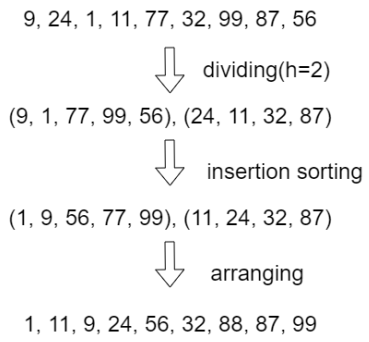

図 4.2: h=2 でのシェルソートの過程

最後にこの配列を h=1、つまり単純な挿入ソートを行ない全体のソートが完了する。具体的な挿入ソートの手順は§3 で詳しく述べたため、この章では割愛する。

h=3 および h=2 で大雑把にソートされた配列 data[]={1, 11, 9, 24, 56, 32, 88, 87, 99} に挿入ソートを行うことと、最初の状態の配列 data[]={9, 24, 1, 99, 87, 56, 11, 77, 32} にいきなり挿入ソートを行うことを比較すると、前者の方が圧倒的に比較回数が少ない。この差は h=3 および h=2 で大雑把にソートを行うことを考慮しても明確であることが多い。これがシェルソートが挿入ソートよりも優れているとされている理由である。

4.3 ソートコードと実行結果

シェルソートを実装した C 言語のソースコードを図 4.3 に示す。

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

void shell_sort(int data[], int lengthy);
int main(void){
    int i, lengthy;
    int data[1000];

    for (i=0; i < 1000; i++){
        data[i] = rand()%900 + 1;
    }

    lengthy = sizeof(data)/sizeof(int);

    shell_sort(data, lengthy);

    for (i=0; i<lengthy; i++){
        printf("%-5d", data[i]);
    }
}

void shell_sort(int data[], int lengthy){
    int h, i, j, t;

    for (h = 1; h < lengthy/9; h = h*3 + 1);

    for(h /= 3; h > 0; h /= 3){
        for (i = h; i < lengthy; i++){
            j = i;
            while ( j >= h && data[j-h] > data[j]){
                t = data[j]; data[j] = data[j-h]; data[j-h] = t;
                j -= h;
            }
        }
    }
}
```

図 4.3: シェルソートのソースコード

また以下にこのソースコードの実行結果を示す。

```
(base) PS C:\Users\shu\Algo> ./a.exe
an array data={9, 24, 1, 99, 87, 56, 11, 77, 32} will be sorted  in
ascending order by Shell Sort.

1 9 11 24 32 56 87 88 99
```

5　クイックソート

クイックソートはまず配列の中から「ピボット」と呼ばれる基準となる値を決め、ピボットより小さい値を左へ、大きい値は右へ移動させ二つのグループに分割する。この動作を分割する要素が 1 つになるまで行うと全体のソートが完了するというソートアルゴリズムである。分割する際の細かな手順については後述の例題で示す。

このような大きな問題を複数の小さな問題に分割し、その小さな問題を一つずつ解決していく手法を「分割統治法」という。

5.1　例題

int 型の配列 `data={4, 94, 34, 28, 9, 55, 42}`について考える。右端の 42 をピボットとして分割する手順を図 5.1 に示す。

ピボットを基準とし、左端からはピボットよりも大きな値、右端からはピボットよりも小さな値を探し出す。一回目は 94 と 9 が見つかり、これらを交換し、矢印を一つずつ進める。この手順を繰り返すといつか左矢印が右矢印を追い越す。このとき左矢印とピボットを入れ替えることで、ピボットより左側にはピボットよりも小さい値が、右側には大きな値が集まる。これが分割の手順である。

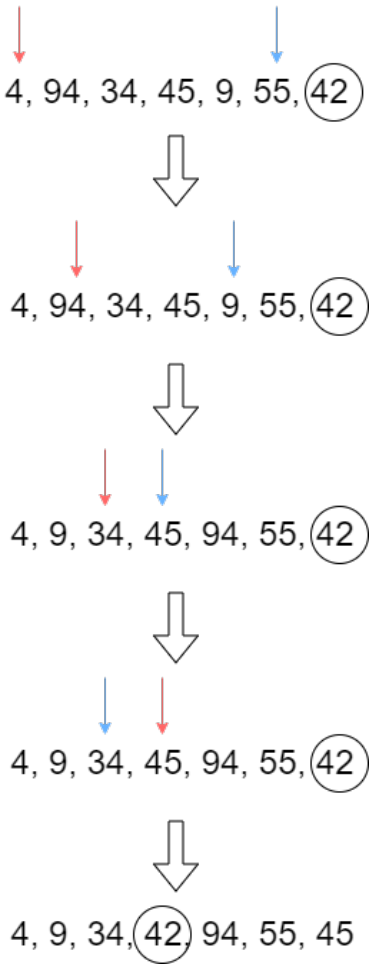

図 5.1: クイックソートの 1 回目の分割の過程

次に二つに分割した後の、さらに細かい分割の過程を図 5.2 に示す。一回目同様、左矢印が右矢印追い越す度にピボットと入れ替える動作を繰り返している。最後に値が一つになれば分割は終

了である。

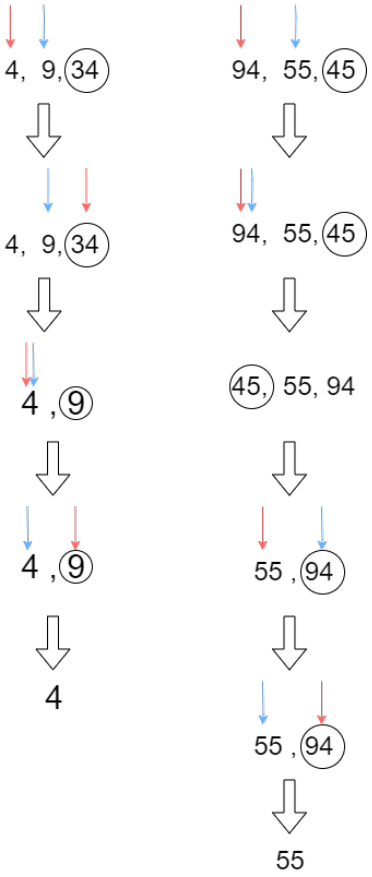

図 5.2: クイックソートの 2 回目以降の分割の過程

5.2 ソースコードと実行結果

クイックソートを実装した C 言語のソースコードを図 5.3 に示す。

```
17  int partition(int data[], int first, int last){
18      int i, j, pivot, tmp;
19
20      i = first-1;
21      j = last;
22
23      pivot = data[last];
24
25      for(;;){
26          while (data[++i] < pivot);
27          while (i < --j && pivot < data[j]);
28
29          if (i >= j)
30              break;
31
32          tmp = data[i]; data[i]=data[j]; data[j]=tmp;
33      }
34
35      tmp = data[i]; data[i] = data[last]; data[last]=tmp;
36      return i;
37  }
38
39  void quick_sort(int data[], int first, int last){
40      int pivot;
41
42      if(first >= last)
43      return;
44
45      pivot = partition(data, first, last);
46
47      quick_sort(data, first, pivot-1);
48      quick_sort(data, pivot+1, last);
49  }
```

図 5.3: クイックソートのソースコード

また以下にこのソースコードの実行結果を示す。

```
(base) PS C:\Users\shu\Algo> ./a.exe
an array data={4, 94, 34, 28, 9, 55, 42} will be sorted  in ascending
order by Quick Sort.

4 9 28 34 42 55 94
```