情報学群実験第 3 回レポート

細川 夏風

2025 年 12 月 9 日

1　目的

本実験の目的は以下である．

1). コンパイラを使わずにアセンブリ言語・機械語によってソートアルゴリズムを直接記述することが可能であることを示す．
2). 擬似コードや高級言語での実装と比べて大きく異なる点があるかについて記述する．コードの複雑さやコード量がどの程度変化するかも示す．
3). アセンブリ言語に置いて実行時間がアルゴリズムの時間計算量に従うことを示す．
4). 高級言語での実装と比べて計算時間に差があることを示す．
5). アセンブリ言語でのソートアルゴリズムを直接記述することにの利点があることを明らかにする．

2　前提

以下のアルゴリズムを示すうえで，いくつかの前提知識が存在し，それらを以下に示す [1].

a). アセンブリ言語とは，機械語に近い低水準言語であり，CPU が解釈可能な命令といくつかのニーモニックが一対一に対応している言語である．
b). 再帰とは，それ自身が自分を含んでいたり，定義の中に自分自身を用いているような関数やその事象のことをいう．
c). ソートとは，データの集合を主要のある項目の値の大小関係によって並べ替える作業のことである．
d). 計算量とは，アルゴリズムの性能を客観的に評価する指標である．実行に要する時間を評価するための時間計算量と，実行に要する記憶領域を評価するための領域計算量がある．

3 方法

3.1 目的 1 を示す方法

まず，コンパイラを用いずにアセンブリ言語によってソートアルゴリズムが直接記述可能であることを今課題のために作成したプログラムコード (付録 [A.2]) から示す．ここではクイックソートを実装した．前提として，これには再帰プログラムを用いており，クイックソートの漸化式 (1) よりソートを行っている．

$$
\begin{aligned}
C_1 &= 0 \\
C_n &= n + 1 + \frac{2}{n} \sum_{k=0}^{n-1} C_k \quad (n \geq 2)
\end{aligned}
\tag{1}
$$

3.2 目的 2 を示す方法

次に，擬似コードや高級言語での実装と比べて大きく異なる点について以下の C 言語のプログラムとの違いから示す．上記のアセンブリ言語と比べて，C 言語でのクイックソートの実装は付録 [A.1] のようになる ([1] 一部改良)．高級言語をアセンブリ言語に変換するという作業を行っていないため，参考文献 [1] のコードを一部改良したものを付録 [A.1] に示す．

3.3 目的 3 を示す方法

次に，アセンブリ言語において実行時間がアルゴリズムの時間計算量に従うことを示ために付録 [B.1] のプログラムを用いる．クイックソートの平均計算時間は $O(n \log n)$ であり，最悪計算時間は $O(n^2)$ である．

3.4 目的 4 を示す方法

次に，高級言語での実装と比べて計算時間に差があることを示すために．大量のデータによって発生する実行時間の差を計測するプログラムを付録 [B.2] に示す．

3.5 目的 5 を示す方法

これらのアルゴリズム，プログラムを作成するにあたりそこに発生した苦労や時間，そこから得たアセンブリ言語の安全性などから考える．

4 結果

4.1 目的 1 に対する結果

付録 [A.2] のプログラムコードから，コンパイラを用いずにアセンブリ言語によってソートアルゴリズムが直接記述可能であることを示す.

アルゴリズムは以下の通りである.

(1). 配列の先頭要素を基準値 (pivot) とする．(行 20)
(2). 配列の先頭 (基準値の次の要素) から順に要素を見ていき，基準値より大きい要素を見つけたらその位置を記憶する．(行 25 .loop1)
(3). その後，配列の末尾から順に要素を見ていき，基準値より小さい要素を見つけたらその位置を記憶する．(行 41 .loop2)
(4). 基準値より大きい要素と基準値より小さい要素を交換する．これを，基準値より大きい要素と基準値より小さい要素に分かれるまで続ける (探索がすれ違うまで続ける)．(行 61)
(5). 基準値を大きい要素群と小さい要素群との境界に移動させる．(行 73)
(6). 再帰に入る前に配列の端と基準値をスタックに保存 (行 81)
(7). 基準値の左側と右側の配列に対して，再帰的に適用する．(行 89，行 100)
(8). 再帰から戻った後，スタックから配列の端と基準値を復元する．(行 92)

4.2 目的 2 に対する結果

付録 [A.1] と付録 [A.2] の 2 つのプログラムコードにおいて大きな違いとなる点がある．それは，アセンブリ言語では再帰の際のスタックへの PUSH と POP を明示しているが．C 言語ではそれがないという点だ．C 言語では関数呼び出し時に自動的にスタック操作が行われるため，プログラマが明示的に記述する必要がない．一方で，アセンブリ言語ではプログラマがスタック操作を明示的に記述する必要があるため，コード量が増加し，コードの複雑さも増す．また，アセンブリ言語では，値の比較はレジスタという CPU 内の記憶領域を用いて行われる．これにより，変数等をメモリ内に避難させなければ “eax”，“ebx”，“ecx”，“edx”，“esi”, “edi” の計 6 つ，もしくはレジスタ内のアドレスの指す値による計算しか行えない．それに比べて C 言語ではメモリが許す限りの変数を扱うことができる.

4.3 目的 3 に対する結果

以下の表 1 及び図 2 は各ソートに対して，データ量を変化させながら実行時間を計測した結果である.

表 1: データ数 N における各ソートアルゴリズムの実行時間計測結果（単位：秒）

データ数 N	Quick Sort	Bubble Sort
5,000	0.000	0.007
10,000	0.001	0.065
15,000	0.001	0.213
20,000	0.002	0.477
25,000	0.003	0.828
30,000	0.003	1.243
35,000	0.003	1.778
40,000	0.003	2.350
45,000	0.004	3.049
50,000	0.004	3.805
100,000	0.008	—
500,000	0.038	—
1,000,000	0.095	—
2,000,000	0.193	—
5,000,000	0.493	—

※ $N \geq 100,000$ の Bubble Sort は実行時間が著しく長くなるため計測を省略した（—で表記）。

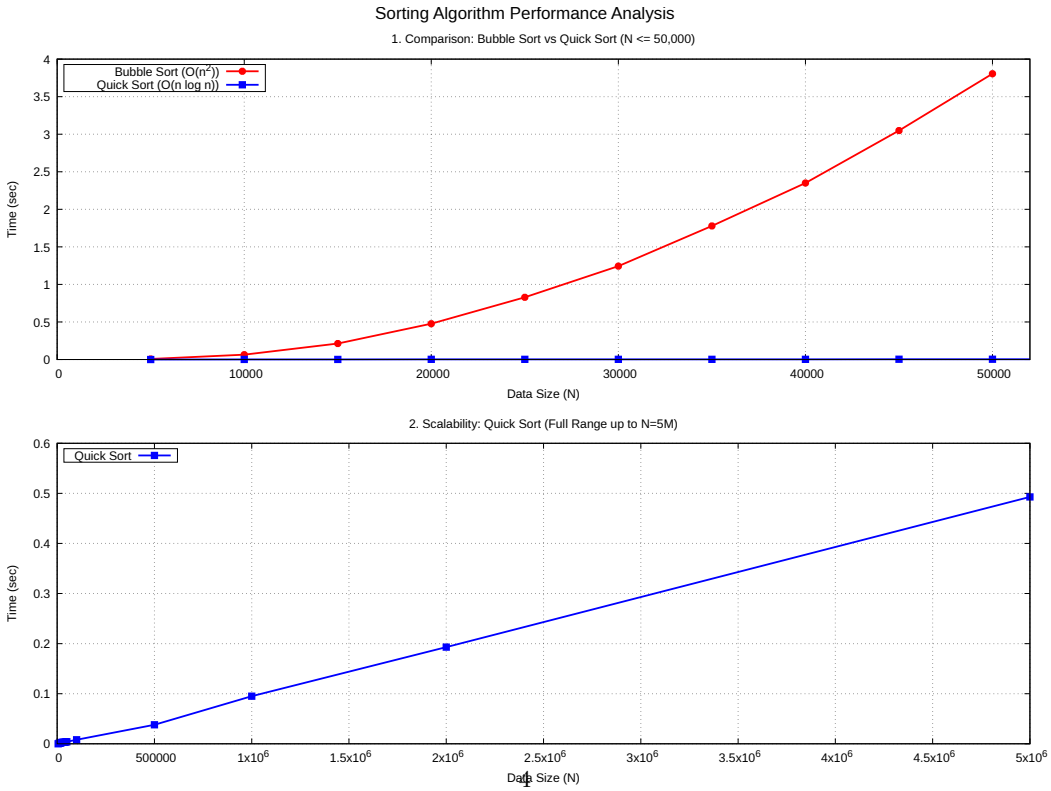

表 2: データ数 N に対する実行時間の推移（上：Bubble Sort との比較，下：Quick Sort のスケーラビリティ）

4.4 目的 4 に対する結果

以下に表 3 に C 言語とアセンブリ言語のクイックソートの実行時間の 10 回分の平均の比較を示す．また，表 4 は最適化 (-O2) オプションを用いている [3]．また，最適化 (-O3) オプションは最適化 (-O2) オプションより高速であるが，バグの問題があるため，(-O2) を用いる．データ数は $1,000,000$ とした．

表 3: クイックソートにおける C 言語とアセンブリ言語の実行時間比較（データ数 $N = 1,000,000$, 最適化なし）

実装言語	**User (秒)**	System (秒)	Real (秒)
C 言語 (Quick Sort)	0.219	0.010	0.263
アセンブリ言語 (Quick Sort)	**0.149**	0.010	0.148

※ User はユーザープロセスが消費した CPU 時間，System はカーネルプロセスが消費した CPU 時間，Real は実行に要した実時間を示す．

表 4: 最適化オプション (-O2) 適用後の C 言語とアセンブリ言語の実行時間比較（データ数 $N = 1,000,000$）

実装言語	**User (秒)**	System (秒)	Real (秒)
C 言語 (Quick Sort, -O2)	0.151	0.000	0.192
アセンブリ言語 (Quick Sort)	**0.143**	0.019	0.168

※ **User** 時間の比較において、C 言語の実装がアセンブリ言語（最適化あり）に近い性能を示した。

4.5 目的 5 に対する結果

これらの結果を用いて，アセンブリ言語でのソートアルゴリズムの有用性について考える．4.4 節の結果から，アセンブリ言語で実装したクイックソートと C 言語で実装したクイックソートには明確に実行時間の差が生じている．よって，アセンブリ言語でソートアルゴリズムを記述することの利点は，実行時間の短縮にあるといえる．また，ビッグデータが注目されている昨今において，データ量が膨大になることが予想されるため，アセンブリ言語での実装は有用であると考えられる．表 4 から最適化オプションを用いた C 言語とアセンブリ言語には大きな差がないことがわかる．よって，最適化オプションを用いることで C 言語でもアセンブリ言語に近い実行時間を再現できることがわかった．

5 考察

本実験では，アセンブリ言語を用いてソートアルゴリズムを実装した．その結果，アセンブリ言語特有のスタック操作やレジスタなどの制約が存在することが明らかになった．他にも，アセンブリ言語でもクイックソートなどの高度なソートアルゴリズムや再帰を実装することが可能であることがわかった．また，実行時間の計測結果から，アセンブリ言語での実装は C 言語での実装よりも高速であることが示された．特に最適化オプションを用いた場合でも，アセンブリ言語が僅かに優れた性能を示した．これらの結果から，アセンブリ言語でソートアルゴリズムを記述することには，実行時間の短縮という利点があると言える．アルゴリズムに対してアセンブリ言語でもデータ量が増加するとそれに伴い実行時間が増加することが確認できた．また，その増加はアルゴリズムの時間計算量に従うことも確認できた．アセンブリ言語はソートアルゴリズムに対して有用であることは確認できたが，その実装は困難極まりなく，プログラマに対する負担がとても大きいと感じる．よって，アセンブリ言語での実装は，極端に実行時間の短縮が重要な場合や使用領域に大きな制限がある場合のみに限られるべきであると考える．また，実験結果から最適化を行えば C 言語でもアセンブリ言語に近い実行時間を再現できた．これにより，最適化オプションによるバグの影響が少なく，実行時間と保守性を考える場合は C 言語での実装が望ましいと考える．

参考文献

[1] 柴田望洋，“新版 C 言語によるアルゴリズムとデータ構造”，ソフトバンククリエイティブ，株式会社，2006 年 10 月 10 日．

[2] 日向 俊二，“独習アセンブラ”，株式会社 翔泳社，2018 年 3 月 18 日．

[3] 青木峰郎，“ふつうの Linux システムプログラミング”，ソフトバンク クリエイティブ株式会社，2006 年 10 月 30 日．

A　実験に使用したプログラムソースコード

A.1　クイックソート（C 言語）：quicksort.c

以下，プログラムの一部は参考文献:[1] を参照している．

Listing 1: quicksort.c

```c
#include <stdio.h>
#include "data.h"

// 値を交換するマクロ
#define swap(type, x, y) //省略

// クイックソート関数
void quick_sort(int a[], int left, int right) {
    int pl = left;
    int pr = right;
    int x = a[(pl + pr) / 2];

    do {
        while (a[pl] < x) pl++;
        while (a[pr] > x) pr--;
        if (pl <= pr) {
            swap(int, a[pl], a[pr]);
            pl++;
            pr--;
        }
    } while (pl <= pr);

    if (left < pr) quick_sort(a, left, pr);
    if (pl < right) quick_sort(a, pl, right);
}

int main(void) {

    // クイックソート呼び出し
```

```
30      quick_sort(data, 0, ndata - 1);
31
32      return 0;
33  }
```

A.2 クイックソート（アセンブリ言語）：sort.s

以下，プログラムの一部は参考文献:[2] を参照している．

Listing 2: sort.s

```
1   section .text
2   global _start
3   _start:
4   mov eax, data                       ;配列の先頭のアドレス
5   mov ebx, data + (ndata - 1) * 4     ;配列の最後のアドレス
6
7   call quick_sort
8
9   mov eax, 1
10  mov ebx, 0
11  int 0x80
12
13  quick_sort: ;quick_sort関数のようなもの
14
15  ;ベースケースを作る(配列の数が1のとき)
16  cmp eax, ebx
17  jge .done
18
19  ;分割と基準値決め
20  mov edi, eax ;基準値のアドレス
21  mov esi, edi ;左端
22  add esi, 4 ;左端
23  mov edx, ebx ;右端
24
25  .loop1: ;左から基準値より大きい数の探索
26
27    mov ecx, [edi]
```

```
28
29      ;端まで行くとループ終了
30      cmp esi, ebx
31      jge .end_loop1
32
33      cmp ecx, [esi] ;基準値の中身と左端の中身の比較
34      jl .end_loop1
35
36      add esi, 4
37      jmp .loop1
38
39  .end_loop1:
40
41  .loop2: ;右から基準値より小さい数の探索
42
43      cmp edx, eax
44      jle .end_loop2
45
46      mov ecx, [edi]
47
48      cmp ecx, [edx]
49      jg .end_loop2
50
51      sub edx, 4
52      jmp .loop2
53
54  .end_loop2:
55
56  ;端同士が通り過ぎたら分割終了
57  cmp esi, edx
58  jge .partition_end
59
60  ;交換処理
61  push dword [esi]
62  push dword [edx]
63
```

```
64    pop dword [esi]
65    pop dword [edx]
66
67    ;再度探索を続ける
68    add esi, 4
69    sub edx, 4
70    jmp .loop1
71
72  .partition_end:
73    ;基準値を境にする
74    push dword [edi]
75    push dword [edx]
76
77    pop dword [edi]
78    pop dword [edx]
79
80    ;再帰のための配列の端と基準値の保存
81    push eax
82    push ebx
83    push edx
84
85    ;左への再帰
86    mov ebx, edx
87    sub ebx, 4
88
89    call quick_sort
90
91    ;再帰から戻ってきたあとの保存の復活
92    pop edx
93    pop ebx
94    pop eax
95
96    ;右への再帰
97    mov eax, edx
98    add eax, 4
99
```

```
100   call quick_sort
101
102   .done: ;呼び出し元に戻る
103     ret
104
105 section .data
106   %include "data.inc"
```

B 実験・計測環境構築スクリプト

B.1 自動計測ベンチマークスクリプト: backmark.py

Listing 3: backmark.py

```
1  import subprocess
2  import random
3  import re
4  import os
5
6  # 出力ファイル名
7  RESULT_FILE = "result_detailed.csv"
8
9  def generate_data_inc(n):
10     # アセンブリ用のデータファイルを生成
11     print(f"Generating data for N={n}...")
12     with open("data.inc", "w") as f:
13         # ランダムデータ生成
14         data = [str(random.randint(0, 2147483647)) for _ in range(n)]
15
16         f.write(f"ndata equ {n}\n")
17         f.write("data:\n")  # ラベル
18
19         # 1000個ずつ区切って dd 命令を書く
20         chunk_size = 1000
21         for i in range(0, len(data), chunk_size):
22             chunk = data[i:i + chunk_size]
```

```
23              f.write("  dd " + ", ".join(chunk) + "\n")
24
25  def compile_asm(filename):
26      # 実行ファイル
27      obj_file = filename.replace(".s", ".o")
28      exe_file = filename.replace(".s", "")
29
30      ret = subprocess.call(["nasm", "-f", "elf32", filename, "-o",
            obj_file])
31      if ret != 0: return False
32
33      ret = subprocess.call(["ld", "-m", "elf_i386", obj_file, "-o",
            exe_file])
34      return ret == 0
35
36  def measure_time(exe_file):
37      # timeコマンドから実行時間を取得
38      try:
39          cmd = f"time ./{exe_file}"
40          result = subprocess.run(cmd, shell=True, stderr=subprocess.PIPE,
                stdout=subprocess.DEVNULL, executable="/bin/bash")
41          output = result.stderr.decode("utf-8")
42
43          match = re.search(r"user\s+(\d+)m(\d+\.\d+)s", output)
44          if match:
45              minutes = float(match.group(1))
46              seconds = float(match.group(2))
47              return minutes * 60 + seconds
48          return 0.0
49      except Exception as e:
50          print(f"Error: {e}")
51          return 0.0
52
53  def main():
54      # CSVファイルの作成
55      with open(RESULT_FILE, "w") as f:
```

```
56          f.write("N,QuickSort_Time(s),BubbleSort_Time(s)\n")
57
58      # 1. 細かい比較 (5,000刻みで 50,000 まで)
59      small_range = range(5000, 50001, 5000)
60
61      # 2. クイックソートのみの比較 (10万, 50万, 100万, 200万, 500万)
62      large_range = [100000, 500000, 1000000, 2000000, 5000000]
63
64      # 計測
65
66      # 1: 両方計測 (5,000 ~ 50,000)
67      for n in small_range:
68          generate_data_inc(n)
69
70          # Quick
71          if compile_asm("quick.s"): t_quick = measure_time("quick")
72          else: t_quick = 0.0
73
74          # Bubble
75          if compile_asm("bubble.s"): t_bubble = measure_time("bubble")
76          else: t_bubble = 0.0
77
78          print(f"N={n:7}: Quick={t_quick:.3f}s, Bubble={t_bubble:.3f}s")
79
80          with open(RESULT_FILE, "a") as f:
81              f.write(f"{n},{t_quick},{t_bubble}\n")
82
83      # 2: Quickのみ計測 (100,000 ~ )
84      print("\n--- Skipping BubbleSort for large N (Too slow) ---")
85      for n in large_range:
86          generate_data_inc(n)
87
88          # Quick
89          if compile_asm("quick.s"): t_quick = measure_time("quick")
90          else: t_quick = 0.0
91
```

```
92          # Bubbleは計測しないので空欄にしておく
93          t_bubble = ""
94
95          print(f"N={n:7}: Quick={t_quick:.3f}s, Bubble= (Skipped)")
96
97          with open(RESULT_FILE, "a") as f:
98              f.write(f"{n},{t_quick},\n")
99
100     print(f"\nDone! Results saved to {RESULT_FILE}")
101
102 if __name__ == "__main__":
103     main()
```

B.2 テストデータ生成スクリプト: gen_data.py

Listing 4: gen_data.py

```
1  import random
2
3  # データ数を1,000,000
4  N = 1000000
5
6  # 乱数の最大値
7  MAX_VAL = 2147483647
8
9  data = [random.randint(0, MAX_VAL) for _ in range(N)]
10
11 # 1. C言語用のヘッダファイル (data.h)
12 with open("data.h", "w") as f:
13     f.write(f"int ndata = {N};\n")
14     f.write("int data[] = {\n")
15     f.write(", ".join(map(str, data)))
16     f.write("\n};\n")
17
18 # 2. アセンブリ言語用のインクルードファイル (data.inc)
19 with open("data.inc", "w") as f:
20     f.write("data: dd " + ", ".join(map(str, data)) + "\n")
```

```
21      f.write(f"ndata equ {N}\n")
22
23  print(f"Generate {N} numbers to data.h and data.inc")
```