情報学群実験第 2 レポート

アセンブリ言語による整列アルゴリズムに関する実験

1250373 溝口洸熙[*1]

2022 年 11 月 9 日

[*1] 高知工科大学 情報学群 2 年 清水研究室

目次

図目次

表目次

ソースコード

第 1 章

アセンブリ言語による整列アルゴリズム記述可否の検証

1　実験の目的

高級プログラミング言語，Java，C，Python などは，『コンパイラ』と呼ばれる装置を使って機械語に書き換えられ，コンピュータで実行されている．

それに対して，アセンブリ言語は各機械語命令につけられた「意味する名前」（ニーモニック；mnemonic）を使ってプログラムを表記する表記法である．[**?**, 第 1 章] アセンブリ言語表記を機械語のビット列に変換する作業をアセンブルと言い，それを行うソフトウェアをアセンブラと言う．

コンパイラとアセンブラは別物であり，アセンブラは機械語の表記を変えたものである故にコンピュータへの命令を 1 対 1 で書き換えるものである点がコンパイラと大きく違う点である．

本実験課題の目的は，このようなアセンブリ言語・機械語に対して，コンパイラを使わずに整列アルゴリズムを直接技術することが可能であるか確認することである．

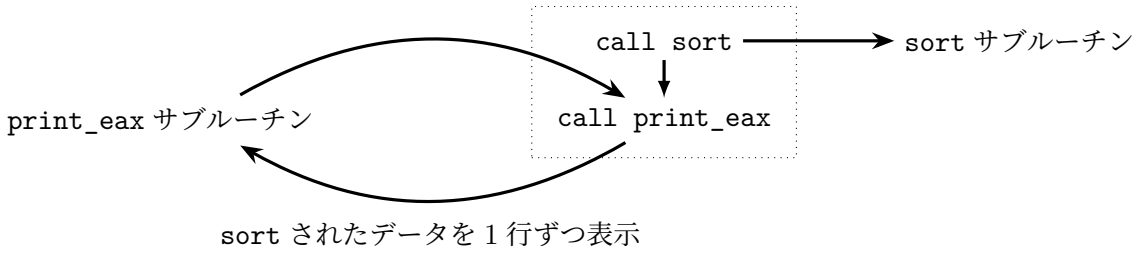
2　プログラムの仕様

0 以上 2^{31} 未満の自然数列に対して昇順に整列するアルゴリズムをアセンブリ言語で記述する．その際，`test_sort.s` ファイルが `sort` サブルーチンを呼び出して整列を行う．検証したい自然数列は，`test_sort.s` 内の `data1` にダブルワードで，データの個数は `ndata1` で定義しており，`test_sort.s` 実行時 `data1` に格納してある自然数列が `print_eax` サブルーチンによって出力される．

`data1` の先頭番地は `EBX`，`ndata1` は `ECX` に格納する．`sort` の呼び出し前後で他の汎用レジスタの値は変化しないように設計されている．内部処理の概要を Fig 1.1 に示す．整列アルゴリズムは，**選択ソートアルゴリズム**[*1] を元に作成している．

[*1] 入力データの最大値を見つけ，それを整列アルゴリズムから除外することを繰り返し行うアルゴリズム [**?**, p.49]

Fig 1.1　処理概要



3　実験

3.1　実験方法

src 1.1 に示すコンピュータとソフトウェアを利用して，アセンブリ言語で書いた整列プログラムを実行可能ファイルに変換し実行した．実行結果を確認し，整列されているか確認するとともに，Java で記述したプログラムとアセンブリプログラムを比較してどの部分をどのようにアセンブリ言語化したかを確認する．

src 1.1　使用したコンピュータとソフトウェア

```
$ uname -a
Linux KUT20VLIN-462 5.4.0-70-generic #78~18.04.1-Ubuntu SMP Sat Mar 20 14:10:07 UTC
2021 x86_64 x86_64 x86_64 GNU/Linux
$ nasm --version
NASM version 2.13.02
$ ld --version
GNU ld (GNU Binutils for Ubuntu) 2.30
$ java --version
openjdk 11.0.10 2021-01-19
```

src 1.2　アセンブル実行コマンド

```
$ nasm -felf print_eax.s ; nasm -felf test_sort.s ; nasm -felf sort.s
$ ld -m elf_i386 print_eax.o test_sort.o sort.o
$ ./a.out
```

3.2　実験結果

実験結果と期待される結果を Tbl 1.1 に示す．自然数列を昇順に整列されていることが確認できる．

Tbl 1.1　実験結果と比較

入力	1 3 5 7 9 2 4 6 8 0 1 2
出力	0 1 1 2 2 3 4 5 6 7 8 9
期待出力	0 1 1 2 2 3 4 5 6 7 8 9

Java で記述した選択ソート（src 1.3）と，アセンブリ言語で記述した選択ソート（src 1.4）を比較して，言語化の箇所を確認する．前者は入力データ列を data とし，後者は EBX に入力データ列の先頭番地，ECX 入力データ列の個数を格納している．ソースコードの各処理の内容と，2 つの言語による対応を Tbl 1.2 に記す．

Tbl 1.2　ソースコードの行対応

処理内容	src 1.3	src 1.4
ループ処理 1 の条件比較・変数処理	3	2 - 3, 25, 26
最大値の更新	4	5
最大値インデックスの更新	5	6
ループ処理 2 の条件比較・変数処理	6	7 - 9, 18, 19
最大値の比較・更新	7 - 11	10 - 17
data[max_index] と data[i] の入れ替え	12 - 14	21 - 24

src 1.3　Java で記述した選択ソート

```
1  int max_index = 0;
2  int max = 0;
3  for (int i = data.length - 1; i > 0; i--) {
4    max = data[0];
5    max_index = 0;
6    for (int j = 1; j <= i; j++) {
7        if (data[j] >= max) {
8                max = data[j];
9                max_index = j;
10           }
11       }
12       int m = data[max_index];
13       data[max_index] = data[i];
14       data[i] = m;
15 }
```

注)
src 1.3，src 1.4 いずれも，整列アルゴリズムの部分のみ掲載している．

src 1.4　アセンブリ言語で記述した選択ソート

```
1  loop0:
2    cmp   ecx,  0
3    jle   endp
4    mov   edx,  [ebx]
5    mov   eax,  0
6    mov   edi,  1
7    loop1:
8      cmp   edi,  ecx
9      jg    loop0l
10     mov   esi,  [ebx + edi*4]
11     cmp   esi,  edx
12     jge   then
13     jmp   endif
14     then:
15       mov edx,  [ebx + edi*4]
16       mov eax,  edi
17     endif:
18       inc edi
19       jmp loop1
20   loop0l:
21     mov   esi,  [ebx + eax*4]
22     mov   edi,  [ebx + ecx*4]
23     mov   [ebx + eax*4],  edi
24     mov   [ebx + ecx*4],  esi
25     dec   ecx
26     jmp   loop0
```

4　考察

実験結果より自然数列を整列アルゴリズムをアセンブリ言語で記述できることがわかった．

ただ，これはあくまで 0 以上 2^{31} 未満の自然数に限った整列アルゴリズムであるため，負の整数やその他の有理数などを対象にした整列アルゴリズムが記述可能であるかは，この実験では検証できていない．

第 2 章

アセンブリ言語・高級言語の実装の違い

1　実験の目的

先に述べたように，アセンブリ言語は機械語を 1 対 1 に記した記法である．それに対して高級言語（高水準言語）は，人間の言語・概念に近づけて設計されてたプログラム言語である．[?]

本実験の目的は，アセンブリ言語や機械語などの低級言語は Java などの高級言語と比べて，実装に関して大きく異なる点があるかどうかを明らかにすること，すなわち，コードの複雑さやコード量がどうなるかを明らかにすることである．

2　実験の方法

実装に関して大きく異なる点を明らかにするため，低級言語と高級言語の一般的な実装手順を書き出し比較をする．低級言語はアセンブリ言語，高級言語は Java を利用する．

さらに，コードの複雑さやコード量がどうなるかを明らかにするため，実際に 2 つの言語で書いた同一のアルゴリズムに対して，行数や条件分岐，ループの回数を比べる．

Java におけるループ回数は `for` 文の個数，比較回数は `if` 文の個数とし，アセンブリ言語における比較回数は `cmp` の個数，ループ回数は一定条件下で上の行のラベルにジャンプする回数とする．評価値は

$$\text{評価値} = \text{行数} + \text{比較回数} + \text{ループ回数}$$

と定義し，評価値とコードの複雑さは比例するものとする．

3　実験の結果

src A.1，src A.2 は，アセンブリ言語で記述したプログラム，src A.3 は Java で記述したプログラムである．いずれも，入力データを受け取り選択ソートアルゴリズムで整列してその整列結果を 1 行ずつ出力するプログラムであり，入力と出力は一致している．それぞれの行数と比較回数を Tbl 2.1 に示す．

Tbl 2.1　行数とループ・比較回数

記述言語	ファイル名	行数	比較回数	ループ回数	評価値
アセンブリ言語	`sort.s`	44	3	2	77
	`test_sort.s`	26	1	1	
Java	`sort.java`	19	1	2	22

4　考察

実験の結果より，両言語の評価値を比べるとアセンブリ言語の評価値の方が Java に比べて 3.5 倍であることが確認できる．1 番目立った違いは行数であろう．アセンブリ言語で記述したものに比べて Java で記述したアルゴリズムは約 1/3 とより簡潔に記述できることが分かる．

その原因として，Java のループに使われる for 文は，比較とジャンプ，ループ変数の定義と処理を 1 行で行うことが可能であることに対して，アセンブリ言語では比較・ジャンプ・ループ変数の処理の命令を 1 つずつ記述する必要のあることが挙げられる．

今回の実験で，低級言語であるアセンブリ言語の方が，高級言語である Java よりも複雑でゴードの量も多くなることが分かった．ただし，この実験では独自の指標でコードの複雑さを測っているため，一般的な複雑の指標である，サイクロマティック複雑度（循環的複雑度）で計測できていない．

第 3 章

低級言語での理論的計算量に関する実験

1　実験の目的

今回の実験の目的は，アセンブリ言語・機械語で直接記述した場合も実行時間は論理的計算量（アルゴリズムによって $O(n^2)$ や $O(n \log n)$ など）に従うことを示すことである．選択ソートのアルゴリズムの計算量は $O(n^2)$ である [?, p.50,51] ので，アセンブリ言語で記述した選択ソートアルゴリズムもそれに従うか検証するれば，目的を満たすことになる．

2　実験方法

より正確に選択ソートの実行時間を計測するため，`test_sort.s` ファイルを `print_eax` を利用する箇所を削除し，src 3.2 に書き換える．

実験環境は，src 1.1 に示した通り．アセンブリ言語での実行時間の検証は，Linux 標準の `time` コマンドを用いて計測する．src 1.2 のコマンドを実行し，実行ファイル `a.out` を生成した後，src 3.1 を実行し実行時間を計測する．各実行時間の中でも `real` が引数コマンドを実行するのにかかった時間である．実験回数は 1 つのテストにつき 3 回行い実行時間を平均する．さらに，選択ソートは最良時間計算量と最悪時間計算量が等しく [?, p.50]，整列対象のデータ列は計測時間に依存しない故，今回はデータ列を全て 0 に定める．実験結果を Excel を使って多項式近似（2 次）を求め，その関数曲線と実験結果を比較する．テストデータ数は (3.1) の集合 T である．

src 3.1　時間計測実行コマンド

```
$ time ./a.out
-- 実行結果（略） --
real    0m0.002s
user    0m0.001s
sys     0m0.000s
```

src 3.2　`test_sort.s` 書き換え後

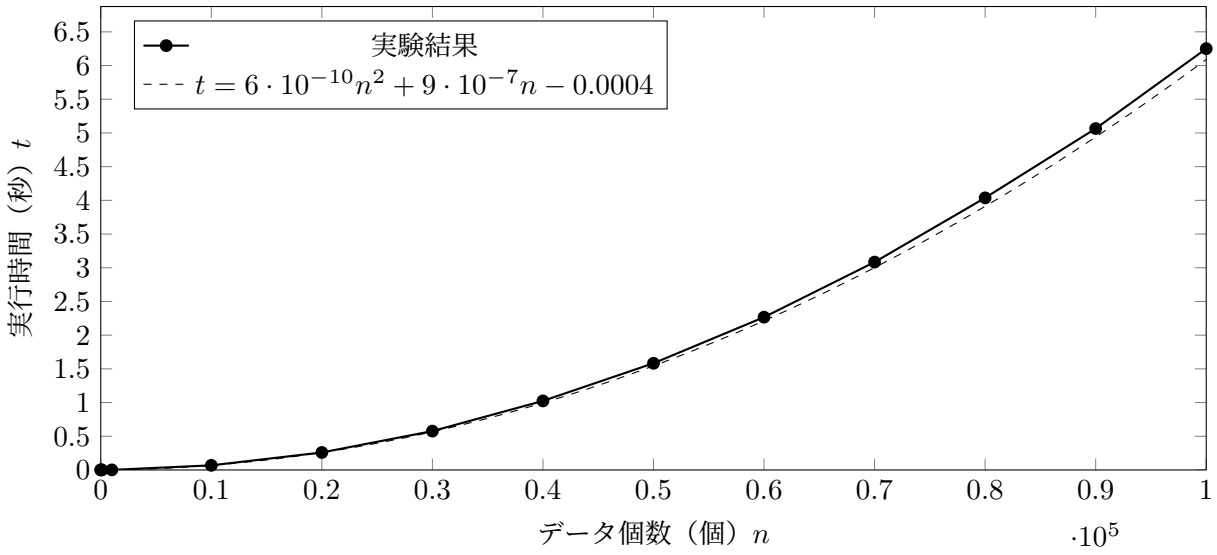
```
略
    extern  sort
_start:
mov ebx,    data
mov ecx,    ndata
call sort
mov eax,    1
mov ebx,    0
int 0x80
    section .data
data: times データ個数 dd 0
ndata: equ ($ - data) / 4
```

$$
\begin{aligned}
T_1 &= \{10^n \mid n \in \mathbb{N}, n \leq 5\} \\
T_2 &= \{n \times 10^4 \mid 2 \leq n \leq 9\} \\
T &= T_1 \cup T_2
\end{aligned} \tag{3.1}
$$

3　実験結果

実験結果 Fig 3.1 に示す．実行時間実験結果詳細は，Tbl B.1 に掲載している．

Fig 3.1　アセンブリ言語のデータの個数と実行時間の曲線グラフ



実行時間を t，データの個数を n とすると，近似多項式は (3.2).

$$t = 6 \cdot 10^{-10}n^2 + 9 \cdot 10^{-7}n - 0.0004 \tag{3.2}$$

4　考察

オーダ記法は，アルゴリズムの時間計算量の入力サイズ n を用いた関数 $f(n)$ に対して，その関数の主要項の係数を削除した $O(f(n))$ である．[?, p.7]

従って，(3.2) を元にオーダ記法の定義より，このアルゴリズムの計算量は $O(n^2)$ であることが分かる．

第 4 章

高級言語と低級言語の計算時間に関する実験

1　実験の目的

アセンブリ言語と高級言語での実装を比べて計算時間に差があるかどうかを明らかにする．

2　実験の方法

データ数は，src A.3 の 3 行目を

```
int[] data = new int[データ数];
```

とすることでデータ数に応じた実験ができる．

実験環境は，src 1.1 に示した通りである．Java のコンパイル・実行は src 4.1 に示す．

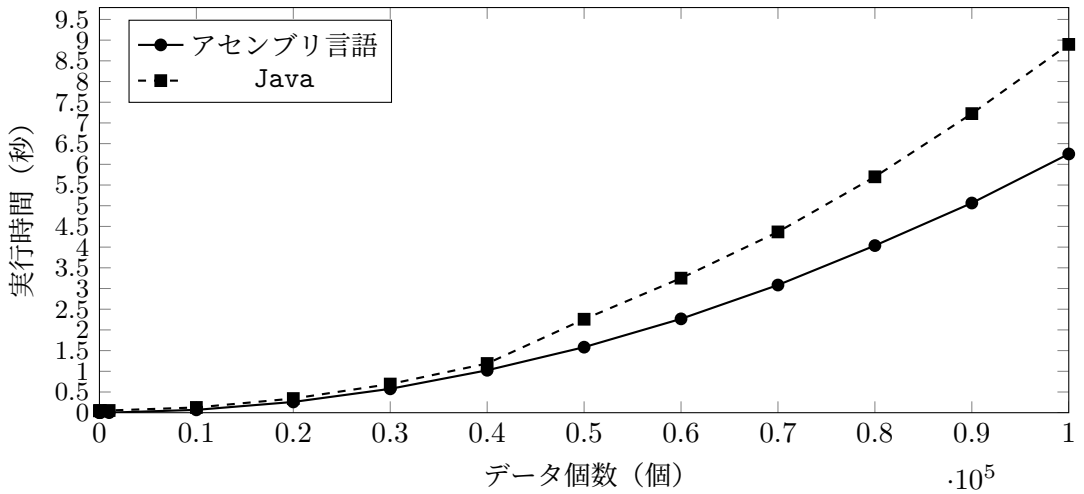
src 4.1　Java コンパイル・実行時間の計測

```
$ javac sort.java
$ time java sort
```

3　実験結果

実験結果を Fig 4.1 に示す．Java の実行時間実験結果詳細は，Tbl B.2 に掲載している．

Fig 4.1　各言語の実行時間比較



4　考察

実験結果より，アセンブリ言語で記述した選択ソートの方が Java で記述した選択ソートよりも実行時間は短いことがわかった．ただし，いずれも src A.3，src A.1 のアルゴリズムでの実行時間であるので，一般的にアセンブリ言語の方が Java よりも実行時間が短いとの結論にはならない．

Java はクラスファイルからの実行はインタプリタ方式をとっているため，機械語に翻訳された実行ファイルよりも実行に時間がかかると考えらえる．

第 5 章

アセンブリ言語による記述の利点

1　目的

1 章から 4 章で得られた事実や考察に基づいて，アセンブリ言語で整列アルゴリズムを直接記述することに利点があるかどうかを明らかにする．

2　考察

4 章の実験結果より，高級言語である Java よりもアセンブリ言語で記述した選択アルゴリズムの方が，実行時間は短いことが分かっている．ただ，先にも述べたように Java がコンパイラ・インタプリタ方式採用していることが大きな原因と考えられ，仮にコンパイラ方式を採用している言語で実験したとしても，一般に高級言語よりも低級言語の方が実行時間は短いとは断定できない．今回の実験に限ってはアセンブリ言語での記述によって整列アルゴリズムがより高速に実行できたので，この点についてはアセンブリ言語で記述する利点であろう．

アセンブリ言語は CPU 内のレジスタの値の操作，主記憶領域の値の操作を 1 つずつ記入する．2 の実験を踏まえると，コード量は一般の高級言語に比べるととても多くなり複雑になる．これは，プログラマーの人的ミスを引き起こしやすい要因でもあり，コードが複雑化するアセンブリ言語の欠点とも言える．

謝辞

本実験課題は本学情報学群 1250372 三上柊氏と共同で実施した．また本学情報学群の高田 喜朗准教授には，整列アルゴリズムに関する様々な助言をいただいた．これらの方々に深く感謝いたします．

溝口洸熙

付録

A　ソースコード

src A.1　sort.s

```
1     section .text
2     global  sort
3   sort:
4     push  esi
5     push  edi
6     push  edx
7     push  ecx
8     push  ebx
9     push  eax
10    dec   ecx
11  loop0:
12    cmp   ecx,  0 ; ecx = i
13    jle   endp
14    mov   edx,  [ebx]   ; max = data[0]
15    mov   eax,  0       ; max_indent = 0
16    mov   edi,  1 ; edi = j
17    loop1:
18      cmp   edi,  ecx   ; j > i?
19      jg    loop0l
20      mov   esi,  [ebx + edi*4] ; data[j]
21      cmp   esi,  edx       ; data[j] >= max
22      jge   then
23      jmp   endif
24      then:
25        mov edx,  [ebx + edi*4] ; max = data[j]
26        mov eax,  edi           ; max_index = j
27      endif:
28        inc edi
29        jmp loop1
30    loop0l:
31      mov   esi,  [ebx + eax*4]   ; m = data[max_index]
32      mov   edi,  [ebx + ecx*4]   ; edi = data[i]
33      mov   [ebx + eax*4],  edi   ; data[max_index],  data[i]
34      mov   [ebx + ecx*4],  esi   ; data[i] = m
35      dec   ecx
36      jmp   loop0
37  endp:
```

```
38      pop eax
39      pop ebx
40      pop ecx
41      pop edx
42      pop edi
43      pop esi
44      ret
```

src A.2 test_sort.java

```
1   section .text
2   global  _start
3   extern  sort, print_eax
4
5  _start:
6   mov ebx,  data
7   mov ecx,  ndata
8   call  sort      ; ソート
9   mov edi,  0
10 loop:   ; 結果の出力
11  cmp edi,  ndata
12  je  endp
13  mov eax,  [data + edi * 4]
14  call print_eax
15  inc edi
16  jmp loop
17
18
19 endp:
20  mov eax,  1
21  mov ebx,  0
22  int 0x80
23
24  section .data
25 data:  dd 1, 3, 5, 7, 9, 2, 4, 6, 8, 0, 1, 2
26 ndata  equ ($ - data) / 4
```

src A.3 sort.java

```
1  class sort{
2      public static void main(String[] args){
3          int[] data = {1, 3, 5, 7, 9, 2, 4, 6, 8, 0, 1, 2};
4          buble(data);
5          for(int d: data){
6              System.out.println(d);// 出力
7          }
8      }
9      private static void buble(int[] data) {
10         int max_index = 0; int max = 0;
11         for (int i = data.length - 1; i > 0; i--) {
12             max = data[0]; max_index = 0;
```

```
13              for (int j = 1; j <= i; j++) {
14                  if (data[j] >= max) { max = data[j]; max_index = j; }
15              }
16              int m = data[max_index]; data[max_index] = data[i]; data[i] = m;
17          }
18      }
19  }
```

B 実験結果の詳細

Tbl B.1 実行時間計測実験結果（アセンブリ言語）

データ個数	実行時間（秒）			平均
	1 回目	2 回目	3 回目	
1	0.001	0.001	0.001	0.001
10	0.002	0.001	0.001	0.00133333333333333
100	0.001	0.001	0.002	0.00133333333333333
1000	0.001	0.001	0.002	0.00133333333333333
10000	0.072	0.066	0.068	0.0686666666666667
20000	0.264	0.254	0.261	0.259666666666667
30000	0.588	0.577	0.565	0.576666666666667
40000	1.033	1.016	1.025	1.02466666666667
50000	1.584	1.588	1.577	1.583
60000	2.254	2.271	2.276	2.267
70000	3.090	3.084	3.078	3.084
80000	4.024	4.053	4.035	4.03733333333333
90000	5.068	5.096	5.032	5.06533333333333
100000	6.254	6.254	6.244	6.25066666666667

Tbl B.2 実行時間計測実験結果（Java）

データ個数	実行時間（秒）			平均
	1 回目	2 回目	3 回目	
1	0.045	0.047	0.049	0.047
10	0.046	0.050	0.047	0.0476666666666667
100	0.044	0.048	0.056	0.0493333333333333
1000	0.050	0.057	0.054	0.0536666666666667
10000	0.122	0.131	0.127	0.126666666666667
20000	0.342	0.345	0.339	0.342
30000	0.689	0.697	0.691	0.692333333333333
40000	1.194	1.191	1.181	1.18866666666667
50000	2.226	2.272	2.275	2.25766666666667
60000	3.239	3.272	3.240	3.25033333333333
70000	4.379	4.339	4.384	4.36733333333333
80000	5.703	5.687	5.706	5.69866666666667
90000	7.249	7.188	7.238	7.225
100000	8.906	8.899	8.887	8.89733333333333