情報学群実験第 2
アセンブリ言語を用いたソートプログラムの実現可能性および高級言語との比較，時間計算量についての検証

学籍番号： 1240293
氏名：植田　蓮
E-mail: 240293p@ugs.kochi-tech.ac.jp

2021 年 11 月 4 日

1　実験の目的

本実験課題では，アセンブリ言語を用いる．アセンブリ言語は原則として，機械語の命令と 1 対 1 対応となっているが，本課題ではアセンブリ言語を用いて，整列アルゴリズムの実現が可能なことを確認することが目的の 1 つである．さらに，アセンブリ言語で，実装された，整列アルゴリズムが，そのアルゴリズムの理論的計算量に従うこと示す．また，高級言語や，疑似コードで記述されたものとの，コードの量を比較し，コード量や複雑さにどれだけの変化があるのかを明らかにする．

2　方法

2.1　実行環境

実験を行った環境を表 1 に示す

表 1: 実行環境

OS	Ubuntu 18.04.5 LTS 64bit
メモリ	11.7GiB
CPU	Intel Xeon(R) Gold 5218R CPU
クロック周波数	2.10GHz
コア数	2
ディスク	158.0GB
仮想化	VMware
GNOME	3.28.2
shell	bash 4.4.20(1)-release (x86_64-pc-linux-gnu)

2.2 設計・実装

本課題では，ソートアルゴリズムとして，最も単純な，バブルソートを実装した．ソースコードは付録 A に示す．

2.2.1 外部仕様

プログラムはサブルーチンとして定義されている．指定された主記憶領域中のダブルワード (4 byte) の列を昇順に整列し，整列対象の先頭番地は EBX レジスタに保存されている．また，ダブルワード列に含まれるダブルワードの個数は ECX レジスタに保存されており，30 万個以下とする．整列対象のダブルワード列に含まれる数の定義域は，0 以上 2^{31} 未満とし，サブルーチンの実行前と実行後で，汎用レジスタの値が変わらないようにする．上記の条件を満たしていないデータ列の入力は想定しておらず、その場合のエラー処理等は特に行わない。

2.2.2 内部仕様

実行前後で，レジスタの値が変わらないようにするために，サブルーチンの最初で各汎用レジスタの値を push を用いて，スタックに保存し，最後に push とは逆順に pop を行うことで，実行前後でのレジスタの値を保存している．

ESI レジスタには，ダブルワード列の先頭の番地を保存しており，EAX レジスタは，探索を行う末尾の番地を表している．

loop0 では，未ソートのダブルワード列の末尾の番地である EAX レジスタを 4 ずつ減算することで更新し，更新した値を先頭番地と比較することで，繰り返し処理の判定を行っている．

loop1 では，注目要素と，その 4 番地後ろの要素を比較し，注目要素が，後ろの要素よりも大きいなら，その 2 つの要素に関して，swap 処理を行う．その後，注目要素の番地を 4 バイト後ろにずらし，その番地と，未ソートの比較番地を保存した EAX レジスタと比較し，同値なら，loop1 を終了し，loop0 に戻る．同値でないなら，再度，loop1 を実行する．

2.3 テスト

サブルーチンが正常に動作しているかの確認のために，以下のデータを用いてテストを行った．

- 300 個の乱数の列の整列
- 0,1,2,....,300 の整列済みのデータ
- 300,299,298,.....,1,0 の逆順に整列されたデータ
- 境界値である，0 と 2147483647 を含むデータ

2.4 Java との比較

高級言語で記述されたコードと，アセンブリ言語で記述されたコードの比較を行うために，本課題では，Java で記述されたコードとの比較を行う．Java を用いてバブルソートを実装したプログラムを付録 2 に示す．アセンブリ及び Java で記述されたコードはどちらも，バブルソートのアルゴリズムを実装しているが，コードの量や，複雑さについて，比較を行う．

2.5 実行時間計測

今回実装した，バブルソートは一般に入力サイズ N に対して，$O(N^2)$ になることが知られているが，アセンブリ言語で実装されたプログラムでも同じことが成り立つことを確認するために N 個の入力に対して，実行ファイルを make コマンドを用いてテストプログラム実行しその実行時間を計測した．計測には bash の time コマンドを用いて時間を計測した．その際の比較に利用したのは，ユーザー CPU 時間 (user) である．以下に実行例を示す．また，Makefile およびテスト実行プログラムを付録 3 に示す.

実行例 1: time コマンド実行例

```
  240293p@KUT20VLIN-338:~/pl2/git/q1-i386-pl2-2021-groupa/chap9$ time make test
nasm -felf test_sort.s
ld -m elf_i386 sort.o print_eax.o test_sort.o -o test_sort
./test_sort
1
5

real 0m0.096s
user 0m0.009s
sys 0m0.005s
```

3 結果

3.1 テスト結果

2.3 節で示した，テストを行った結果，すべてのテストにおいてソートプログラムはデータ列を正常にソートしていた.

3.2 高級言語とのコードの比較

ソースコード 1 とソースコード 2 を比較すると，Java で記述されたものよりアセンブリで記述されたもののほうが明らかにコードの量が増加している．複雑さについては，コード量の増加要因との関係を含めて，4 節に示す.

3.3 アセンブリで記述されたプログラムの時間計算量

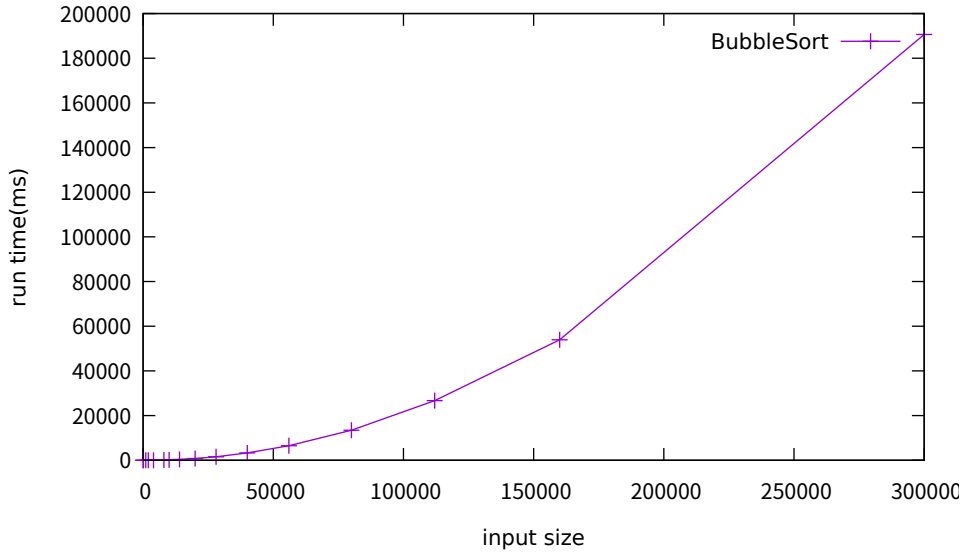
2.5 節に示した，実行時間計測結果を表 2 および図 1 に示す．どちらの図表も，実行時間の単位はミリ秒であることに注意されたい．これを見ると，アセンブリ言語で実装されたバブルソートは入力サイズが 4000 以上になると $O(N^2)$ に従っていることが確認できる.

H

表 2: 入力サイズと実行時間

input size	10	100	1000	2000	4000	8000	10000	14000
run time(ms)	4	6	8	10	22	101	170	365
input size	20000	28000	40000	56000	80000	112000	160000	300000
run time(ms)	773	1573	3281	6486	13452	26674	52921	190635

図 1: 入力サイズと実行時間の変化



4 考察

4.1 アセンブリ言語での整列アルゴリズム実装可能性

3.1 節で示したように，複数のテストデータを用いて，整列プログラムのテストを行った結果，すべてのテストにおいて正確に整列が実行されていることが確認された．このことから，アセンブリ言語によって，整列アルゴリズムを記述することが可能であると考えられる．また，アセンブリ言語は原則として，機械語に対して，1 対 1 の命令セットを持っていることから，機械語による直接の記述が可能であることも同時に言える．

4.2 高級言語とのソースコード比較

実験の結果として，Java による記述と比べると，アセンブリ言語による記述の方がコード量が多くなっていることが確認された．アセンブリ言語では，命令を記述している行が 36 行に及んだのに対して，Java では，class や配列の宣言を含めて，14 行で記述されている．このような結果になる理由として，アセンブリ言語では，代入や比較などを命令として 1 行ずつ記述する必要があるためだと考えられる．例えば，Java で以下のように書くとき

実行例 2: Java での繰り返し文

```
1  for(int i = array.length - 1; i > 0; i--) {
2    .....
3    }
```

アセンブリでは，以下のように記述しなければならない．ここでは，配列の長さが既知とする．arrayLength は配列の長さである．

実行例 3: アセンブリでの繰り返し文

```
1       sub arrayLength, 1
2       mov ebx, arrayLength
3   loopn0:
4       cmp ebx, 0
5       je end
6       ....
7       dec ebx
8       jmp loop0
9
10    end
```

for 文を実現するためには Java では 1 行で記述することができるものをアセンブリでは，約 6 行の記述が必要になる．バブルソートは　2 重ループを用いるアルゴリズムのため，比較は最低でももう一度行うので，コード量が増加することは容易に想像できる．

また，アセンブリ言語のプログラムでは，サブルーチン実行前後で，レジスタの値が変化しないようにする必要があった．そのためのスタックへの push が 6 回ある．サブルーチン終了時には同じ回数 pop するため，レジスタの値を保存するために，合計 12 回の命令が必要である．このことも，アセンブリ言語で記述すると，コード量が増加した原因であると考えられる．

以上のことを踏まえると，繰り返し及び比較を行うと，コード量が増え，サブルーチンでは，レジスタの値保持のためにコードが増えるため，一般にアセンブリ言語で記述すると，高級言語で，記述した場合よりも，コード量が増えると考えられる．

コードの複雑さについては何をもって複雑になっているのかという定義にも依存するが，私は，Java とアセンブリによるバブルソートのソースコードを比較したときにはコードの複雑さは変わっていないと考える．理由は，アセンブリと Java における違いは，文法だけであるからだ．前述したようにたしかに，Java での for 文をアセンブリで実行しようとすると，コードの量は増加すると考えられる．しかし，どちらのプログラムも，実行される命令をトレースしていけばその内容はほとんど一致している．むしろ，高級言語で記述されている文を 1 命令ずつトレースすると，アセンブリ言語に一致すると言ってもいい．つまり，両者の違いは文法であり本質的には，何も変わらないのである．そのことは，Java で実現可能なソートプログラムをアセンブリで実現できたことが証明している．

もちろん高級言語では，グローバルとローカル変数，変数名による違いといったふうに自由に変数を定義できるのに対して，アセンブリ言語では，限られたレジスタと主記憶領域を用いて，記述しなければいけないので，swap 処理の際に代入の回数が増えたり，主記憶領域との値のやり取りが増えたりはするもの，そこについても，使える変数や，メモリへのアクセスが増えるだけで，本質的には swap を行っているだけなので，コードが複雑になったとは考えるべきではない．

しかし、今回の実験では、ソースコードの比較を行っただけであるので、実行時間については、どちらがより実行時間が短いかは不明である。

4.3　アセンブリ言語によるバブルソートの時間計算量

バブルソートの理論計算量は入力サイズ N に対して $O(N^2)$ であるが，実験結果から，アセンブリ言語で実装した際にも $O(N^2)$ に従うと考えられる．今回の実験では，入力サイズが 2000 までは $O(N^2)$　となっていないが．一般に理論計算量は入力サイズが十分に大きい，つまり，$n \to \infty$ のときの漸近的な値を示しているので，今回は，入力サイズが 4000 以上に注目したときに概ね $O(N^2)$ となっているので，理論計算量に従っていると考えられる．

A 付録 1

ソースコード 1: sort.s

```
1  section .text
2  global sort
3  sort:
4  ;; 先頭番地が保存されている ebx
5  ;; に個数が保存されている ecx
6  push eax
7  push ebx
8  push ecx
9  push edx
10 push esi
11 push edi
12
13 mov eax, 4  ;eax=4
14 mov edx, 0
15 mul ecx   ;eax個数=4*
16 mov edx, 0
17 add eax, ebx  配列の末尾の番地;+4
18 sub eax, 4
19 mov esi, ebx  先頭番地を保存しておく;
20
21 loop0:
22 sub eax, nbyte  配列の未ソートの末尾の番地;
23 cmp eax, esi  未ソートの末尾と先頭番地を比較;
24 jl end   小さいなら終了;
25
26 loop1:
27 mov edi, [ebx]
28 cmp edi, [ebx + 4] 前後の数を比べる;
29 jle add4  配列風に書くと;()[n] < [n+1] なら  注目要素を次の要素にする
30
31 swap:    ;[ebxと][ebxを+4]swap
32 mov edi, [ebx]
33 mov [tmp], edi
34 mov edi, [ebx + 4]
35 mov [ebx], edi
36 mov edi, [tmp]
37 mov [ebx + 4], edi
38
39
40 add4:
41 add ebx, 4  ; 比較する番地を 1 要素ずらす
42 cmp ebx, eax  ; 未ソートの末尾番地と比較する番地を比較
43 jle loop1  ; イコールならを脱出 loop1
44 mov ebx, esi  ; 配列の先頭から比較し直す
45 jmp loop0  ; に戻る loop0
46
```

```
47  end:
48  pop edi
49  pop esi
50  pop edx
51  pop ecx
52  pop ebx
53  pop eax
54  ret
55
56  section .data
57  tmp: dd 0
```

B 付録2

ソースコード 2: srot.java

```
1  public class BubbleSort {
2    public static void main(String[] args) {
3      int array[] = {0, 100, 88, 77, 66, 55, 44, 33, 22}; // ソートされる
           データ列
4      for(int i = array.length - 1; i > 0; i--) { // 未ソートの配列の末尾をデク
           リメントしていく
5        for(int j = 0; j < i; j++) { // 先頭から未ソートの末尾までをソートして
             いく
6          if(array[j] > array[j+1]) { // 注目要素よりつ後ろの要素のほうが大きい
               ならスワップ 1
7            int tmp = array[j];
8            array[j] = array[j+1];
9            array[j+1] = tmp;
10         }
11       }
12     }
13   }
14 }
```

C 付録2

ソースコード 3: Makefile

```
1  # マクロ定義
2 AS = nasm -felf
3 Ld = ld
4 LDFLAGS = -m elf_i386
5 OBJS_SORT = sort.o print_eax.o test_sort.o
6
7 # .の生成 o
8 %.o:%.s
```

```
9         $(AS) $<
10
11 # test_sort デフォルトターゲット
12 test_sort: $(OBJS_SORT)
13         $(LD) $(LDFLAGS) $+ -o $@
14
15 # 疑似ターゲット
16 .PHONY:clean test
17 # 不要ファイルの削除
18 clean:
19         rm -f *.o *~ a.out test_sort
20
21 # テストでの実行
22 test: $(OBJS_SORT)
23         $(LD) $(LDFLAGS) $+ -o test_sort
24 ./test_sort
```

ソースコード 4: test_sort.s

```
1
2         section .text
3         global _start
4         extern sort, print_eax
5 _start:
6         mov ebx, data1 ; データの先頭番地
7         mov ecx, ndata1 ; ダブルワードの個数
8         call sort ; 昇順に整列
9
10        mov eax, [data1] ; 先頭最小値=
11        call print_eax
12        mov eax, [data1 + 4 * (ndata1 - 1)] ; 末尾最大値=
13        call print_eax
14
15        mov eax, 1
16        mov ebx, 0
17        int 0x80 ; exit
18
19        section .data
20 data1: dd 5, 4, 3, 2, 1
21 ndata1: equ ($ - data1)/4 ; ダブルワードの個数バイト数 (=/4)
```