研究レポート

ソート処理時間から考察する計算量再評価

東京都立日比谷高等学校

佐藤 光　３年

実験期間・データ収集期間　2022年４月～2023年９月

A. 研究目的

コンピュータ開発の進展により，学術研究の分野においても—模擬実験やデータ処理などの多くの場面で—計算機の力を借りることが増えてきた。たとえば創薬の分野では，薬剤の化学物質が病気の原因タンパク質に高い確率で入り込めるかどうかをスーパーコンピュータを用いてシミュレーションしたりしている。現場で行われるこのような計算は非常に膨大になり，スーパーコンピュータ「京」を用いてもシミュレーティングに 5 時間 45 分が必要になることもある。一般的な PC の処理能力を 10 億回/秒とし，これに対して「京」の処理能力を 1 京回/秒とすると，家庭で使うような PC で同様の計算をするにはおよそ 57,500,000 時間 (=7,000 年近く) かかることが予測される。コンピュータを用いた計算において—とくに膨大な量の計算をする場合には—その計算時間が非常に重要な要素となっている。

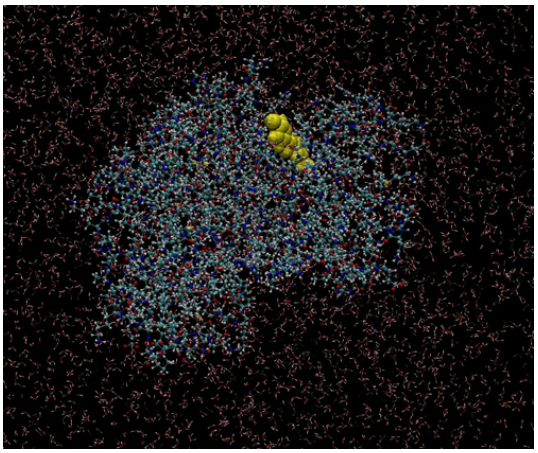


図 1: 「京」で行われたシミュレーションの例

出典：理化学研究所 計算科学研究機構 (AICS).”創薬とスパコン”.2023-05-01.https://aics.riken.jp/jp/post-k/pi/drugdiscovery.html (参照 2023-07-22)

そのため，種々のアルゴリズムを利用する分野ではその処理にかかる時間をあらかじめ評価しておく手法が用いられる。処理時間はコンピュータに与えられるデータの大きさに左右されるため，データの大きさを n として処理時間を n の関数で表すことが多い。ただし，計算時間はコンパイラやマシンの性能などさまざまなソフト的・ハード的要因に影響されるため，正確な処理時間を予測することは不可能である。そのため，計算時間を**漸近的に**評価する big-O，little-O，ω，Ω，Θ などの手法が用いられる。しかし，これらの手法はいづれも処理時間を高解像度で考察するのには不向きな点があることがしばしば指摘されてきた。たとえば本研究で題材とする big-O(以降「O 記法」と略記) の定義は

$$f(n) = O(g(n)) \quad (n \longrightarrow \infty) \quad \Longleftrightarrow \quad \exists k,\, \exists c,\, \Big[\forall n > k,\, \big(\big|f(n)\big| \leqq c\big|g(n)\big| \big) \Big]$$

であり，（つまり定数倍すれば十分に大きな n で必ず実際の計算時間を上回るということ）これを用いると

$$3n^2 + 4n + 9 = O(n^2) \qquad 5log_2 n + 9 = O(logn) \qquad \sin(n) = O(1)$$

のようになる。[*1]そのため，O 記法などを用いるのみではプログラムに含まれるループの重なりや二分探索に見られるような特徴的な分岐が含まれるかどうかがわかるのみで，類似したアルゴリズム同士での計算時間の比較をするには適していないといえる。しかし，計算時間を短縮させるための対策を講じる前後では処理機構は類似している場合が多く，このままの O 記法では処理時間がどれほど改善されたのか判別することはできない。この「O 記法」のもつあいまいさを軽減することで，今後ひろがっていくと見込まれているコンピュータを用いた実験がより効率よく行えるようになるだろう。

B. 研究方法

実験に使用した PC スペックおよび実行環境

- オペレーティング システム：Windows 10 Home 64 ビット OS x64 ベースプロセッサ
- CPU：Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz
- メモリ：24 GB RAM (23.93 GB RAM は使用可能)
- Python3.11.3 IDLE 上

[*1] 対数関数の底の違いは，定数倍の違いにしかならないため，O 記法においては対数の底を省略して書くのが普通。

実験１．様々なプログラムの処理時間を計測する

表 1: 実装したソートアルゴリズム一覧

ソート名	最良計算量	最悪計算量	平均計算量
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick	$O(n)$	$O(n^2)$	$O(n \log n)$

O 記法による計算量評価と実際の計算処理にかかる時間には差異がたしかに存在することを実証するために，ここで**実験１**を行った。（実験時期：2022 年 8 月）

実験 1 では表 1 に示したソートのアルゴリズムを 6 種類（$O(n^2)$ 型 3 種・$O(n \log n)$ 型 3 種）を Python3 上で実装し，与えられるデータ量を変化させながらその計算処理にかかる時間を計測した。ここでソートアルゴリズムを実験対象としてとりあげたのには以下の理由がある。

- 数あるアルゴリズムの中では比較的単純で実装が簡単
- 種類が豊富
- 与えるデータの準備が容易

なお，計算時間の測定には time モジュールの `time.time()` 関数を用いている。この関数は実在世界の時計を反映しており，CPU 上での処理時間に対応する `time.process_time()` ではない。それは，後者の関数の方が戻り値の解像度が低いために n が小さい場面において正確な測定が期待できなかったためである。ただし，`time.time()` 関数と `time.process_time()` 関数の結果の差異は十分に小さい。この実験において使用した Python コードは右の図 2 に示したフローチャートのとおりである。はじめ「被ソート配列定義」ではカンマ区切りで並べられた数字列が記載されている外部ファイルを読み取り，Python 内で配列を定義している。被ソート配列の n は大きければ大きいほど有効な結果を得やすいが，PC のスペック上 1,000,000 程度が限度であった。つづいて，`A = time.time()` によって現在時刻を記録し，ソート関数を実行後 `B` についても同様にして現在時刻を記録しておく。配列が問題なく昇順に並べ替えられていることを確認後，被ソート配列の n およびソート処理時間として `B-A` を外部ファイルに記録し，プログラムは終了する。

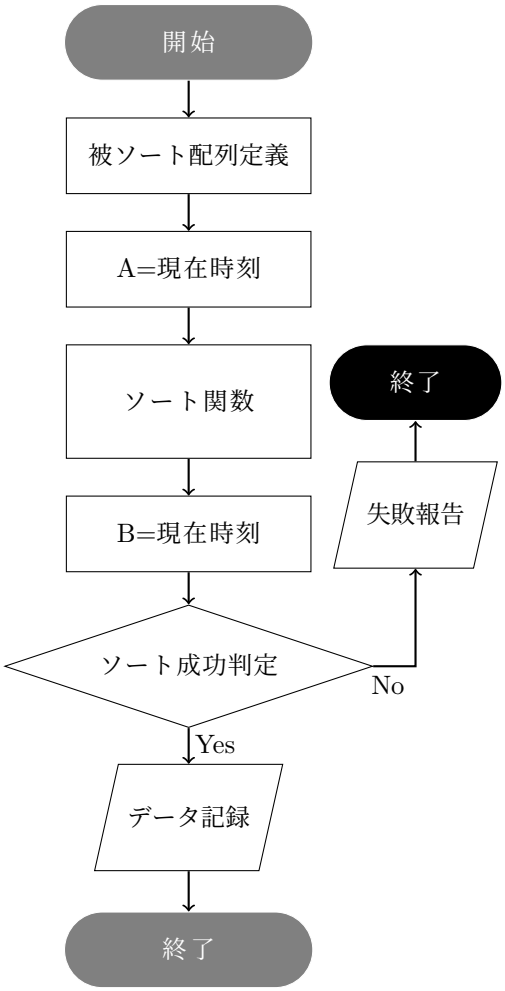


図 2: 実験用プログラムのフローチャート

なお，「被ソート配列定義」での処理には以下の三つの場合がある。●「最良」：はじめから昇順 ●「最悪」：はじめから降順 ●「平均」：完全シャッフル[*2]

ただし，配列は全て 最小値 1 最大値 n の互いに相異なる n 個 の整数からなり，「最良」および「最悪」については $10,000\ 20,000\ 30,000\ \ldots\ldots\ 100,000$ において，「平均」は $100\ 200\ 300\ \ldots\ldots\ 1,000\ 2,000\ 3,000\ \ldots\ldots\ 10,000\ 20,000\ 30,000\ \ldots\ldots\ 100,000\ 200,000\ 300,000\ \ldots\ldots\ 1,000,000$ において定義されたもので検証した。また，$O(n^2)$ 型については「最良」「最悪」「平均」について検証し，$O(n \log n)$ 型については「平均」について検証した。[*3]

[*2] 全ての要素をランダムな他者と入れ替えて定義した

[*3] 実験で使用した n の範囲について $O(n \log n)$ 型の処理時間は少なく，「最良」や「最悪」が有意な結果とならなかったため

また，結果についてはグラフにおこしたほか，n^2 や $n \log n$ に比例する数字で除することによって O 記法との乖離がどれほどあるかを視覚化して考察した。

実験２．プログラムの逆アセンブル

ソートのアルゴリズムが CPU においてどのように処理されているのかを知るために**実験２**を実施した。ここでは，**実験１**で作成した Python コードを dis モジュールを用いて逆アセンブルした。（実験時期：2022 年 12 月）

Listing 1: if 文の逆アセンブル例

```
1  >>> import dis
2
3  >>> def IF(X):
4          if x > 0:
5              return a
6          else:
7              return b
8
9  >>> dis.dis(IF)
10   1 0 RESUME 0
11
12   2 2 LOAD_FAST 0 (x)
13     14 LOAD_CONST 1 (0)
14     16 COMPARE_OP 4 (>)
15     22 POP_JUMP_FORWARD_IF_FALSE 7 (to 38)
16
17   3 24 LOAD_GLOBAL 2 (a)
18     36 RETURN_VALUE
19
20   5 >> 38 LOAD_GLOBAL 4 (b)
21     50 RETURN_VALUE
```

逆アセンブルとは

我々が普段 PC 上で編集するプログラムはそのままマシンに実行されるわけではない。我々が記述できるのは「プログラミング言語」と呼ばれ，人間が読んだり編集したりするために作られたものである。PC 上でこれを実行するためには「プログラミング言語」を機械が読み取ることができる「機械語」に翻訳する必要がある。その作業のことをアセンブルと呼び，アセンブルされた結果である「機械語」が CPU 上で計算されている。逆アセンブルとは CPU 上で処理される「機械語」を人間にも読めるようにする操作のことで，ここで生成されるのは「プログラミング言語」ではなく「アセンブリ」言語である。「アセンブリ」は CPU 上で処理される内容そのままではないが，それを忠実に反映しているため，計算時間の評価において逆アセンブルが有効であるという仮説をたてた。

ここでは，処理される関数の時間的な負荷のことを “重さ” として表現する。関数の “重さ” を評価するために，逆アセンブルによって生成されるアセンブリから読みとれるバイトコードの行数をその “相対的な重” と定義した。たとえば，与えられた変数と 0 との大小を比較する関数を右上のように逆アセンブルした。dis モジュールでの逆アセンブル結果は，Listing1 の 10 から 21 行目に示されている。項目ごとに分けられているうち，いちばん左にある数字が本文（3 から 7 行目）における行番号で，今回注目したい if 文は 2 行目と 4 行目に関わっている。なお，この元となる関数 *IF* では条件分岐後の else: の行は省略しても差し支えがないので逆アセンブル結果からは除かれている。この逆アセンブル結果から分かることは，if 文が主に次のような手順で処理されているということである。なお，括弧の内側は “相対的な重さ” を示している。

- **2 LOAD FAST 0** ローカルな変数[*4]X のアドレス[*5]をスタック[*6]にプッシュ[*7]する（12）
- **14 LOAD CONST 1** 定数 0 をスタックにプッシュする (2)
- **16 COMPARE OP 4** 左右のデータの大小を比較し，その結果をスタックにプッシュする (6)
- **22 POP JUMP FORWARD IF FALSE 7** スタックのトップにある値が ‘false’ である場合にプログラムのバイトコードカウンタ[*8]を 7 だけ増やす (2)

バイトコード上での 21 行目の処理が終わったあと，スタックのトップにある値が ‘true’ の場合 3 行目（バ

[*4] 関数の中で宣言された変数のこと。関数本文の IF(**X**) という部分で定義している。この関数が終了した時点で捨てられる。
[*5] 変数がメモリ上でどこに記述されているかを示す番号のこと。
[*6] 計算機がデータを保持するために用いられる，最後に格納したデータから順に取り出せる後入先出しのデータ構造のこと。
[*7] スタックにデータを”積む”作業のこと (*push*)。なお、スタックからデータを取り出す操作のことは**ポップ** (*pop*) という。
[*8] 擬似的なマシンで実行されるバイトコードと呼ばれる中間言語で，どの部分を処理しているのかを示すカウンタ．バイトコードの行番号は $7 \times 2 = 14$ だけ増える

イトコード 24 行目）が処理され、‘false’ の場合は 5 行目（バイトコード 38 行目）が処理される。以上より，if 文にはバイトコードと上での 4 個の関数，もしくは 22 行の処理が関わっていることが分かる。

同様のことを繰り返して，実験で用いた 6 種類のソート関数を逆アセンブルし，その “相対的な重さ” について考察した。

C. 得られた結果

実験１

表 1 に示したソートアルゴリズムのうち $O(n^2)$ の 3 種について，被ソート配列の要素数を横軸 ($\times 10^3$) に，処理にかかる時間を縦軸してグラフに起こすと右の図 3a に示すようになった。なお，実線部は「平均」であり，塗りつぶしの下端が「最良」，上端が「最悪」である。また，縦横軸ともに線形目盛であり，３つのグラフについてスケールは一致している。

同じく，$O(\log n)$ 型ソートアルゴリズム 3 種についても同様にし，図 3b のような結果が得られた。なお，先述したとおりここでは「最良」と「最悪」は計測していない。

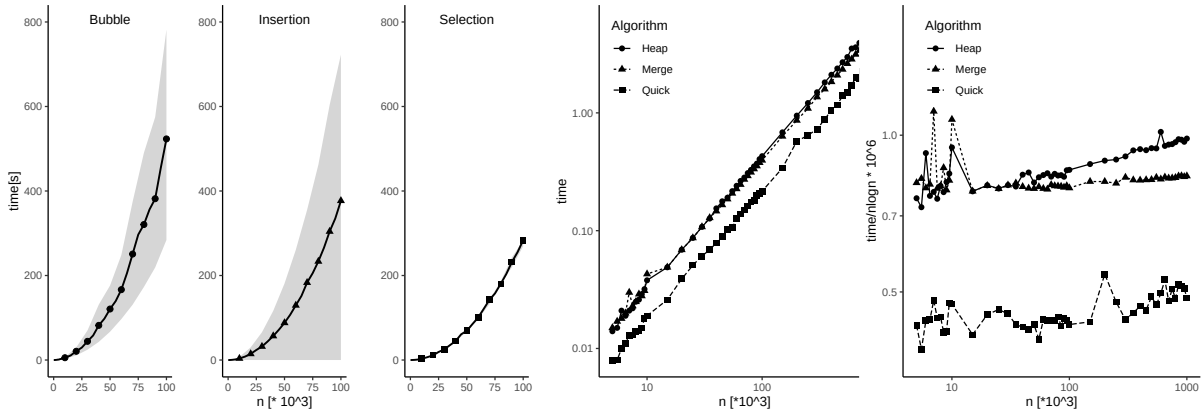
実験２

Bubble ソートの逆アセンブル結果を 5 頁の Listing2 に，Insertion ソートを 6 頁の Listing3 に，Selection ソートを 7 頁の Listing4 示した。なお，LaTeX による組版の影響で一部余白を短縮した部分があるほか，途中での中略部分については~~~としている。詳細は **D 考察-実験２** にて記述する。

D. 考察

実験１

C 得られた結果-実験１からもわかるように，たしかに計算時間は O 記法による評価に従っているといえる。また，実験誤差は十分に小さいものとして捉えてよい。ここで，グラフ 3c に示したのは $O(n \log n)$ で表されるアルゴリズム 3 種について，計算時間を $n \log n \times 10^{-6}$ で除算した結果を縦軸に取ったグラフである。ここで，グラフから Heap ソートと Merge ソートが Quick ソートと比較すると定数倍の差異があるこがわかる。また，Heap ソートと Merge ソートを比較すると，これら 2 つのグラフには傾きの差異があるといえる。以上より，O 記法は●定数倍の誤差 ●傾きの差異のふたつの誤差をはらんでいるといえる。



(a) $O(n^2)$ 型　(b) $O(n \log n)$ 型　(c) 修正 $O(n \log n)$ 型

図 3: ソートアルゴリズムの処理時間

実験２

Bubble ソート再評価の検討

Bubble ソートの処理時間再評価を行う。Bubble ソートは隣り合う要素同士の大小を比較しあい，これらを入れ替えるという操作を繰り返し行うことでソートするアルゴリズムである。ソースコードは右に示した Listing2 の冒頭で定義されている。Bubble ソートは 2 重のループから構成されている。`length` とは配列 `M` の長さであり，n と一致する。2 行目に見られるループは $n-1$ 回の繰り返しを含み、3 行目に見られるループは $n-x-1$ 回のループが含まれる。最奥の for ループに対応しているのは，逆アセンブル結果の 32 行目から 79 行目であり，「最良」の場合において if 文でスキップされる部分は 52 行目から 77 行目である。そのほかのループ構造を考慮すると，Bubble ソートの「最良」・「最悪」の場合それぞれの処理時間再評価結果は以下のようになる。

$$
\begin{aligned}
&13+\sum_{x=0}^{n-2}\{18+\sum_{y=x+1}^{n}(20+2)+2\}+4\\
&=17+\sum_{x=1}^{n-1}\{20+22(n-x-1)\}=11n^2-13n+19\\
&13+\sum_{x=0}^{n-2}\{18+\sum_{y=x+1}^{n}48+2\}+4\\
&=17+\sum_{x=1}^{n-1}\{20+48(n-x-1)\}=24n^2-52n+45
\end{aligned}
$$

ここで，最高次の項に注目したい。ループの最奥にある “相対的な重さ” の和の 2 分の 1 が結果の最高次項の係数になっている。

一般に，処理時間が $O(n^2)$ で表されるプログラムについて，プログラム自体が再起などを含まず for などの単純な 2 重ループから構成されており，1 重目のループの変数が 2 重目のループに影響している場合，逆アセンブルによって再評価された計算時間を $g(n)$，最奥のループ中の “相対的な重さ” の和を W とすると，等式

$$\lim_{n\to\infty}\frac{g(n)}{n^2}=\frac{W}{2}$$

が成り立ち，類似したプログラム同士の計算時間比較をする場合 W を比較するだけでその大小を比較できると言える。よって，ここでは**実験１**にて用意した $O(n^2)$ 型のアルゴリズム 3 種について，それぞれ「最良」

Listing 2: Bubble ソートの逆アセンブル

```
>>> import dis

>>> def BubbleSort():
      for x in range(0, length-1):
          for y in range(x+1, length):
            if M[x] > M[y]:
              M[x], M[y] = M[y], M[x]


>>> dis.dis(BubbleSort)
       2 0 LOAD_GLOBAL 0 (range)
         2 LOAD_CONST 1 (0)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     >> 14 FOR_ITER 33 (to 82)
        16 STORE_FAST 0 (x)

       3 18 LOAD_GLOBAL 0 (range)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        30 GET_ITER
     >> 32 FOR_ITER 23 (to 80)
        34 STORE_FAST 1 (y)

       4 36 LOAD_GLOBAL 2 (M)
         38 LOAD_FAST 0 (x)
         40 BINARY_SUBSCR
         42 LOAD_GLOBAL 2 (M)
         44 LOAD_FAST 1 (y)
         46 BINARY_SUBSCR
         48 COMPARE_OP 4 (>)
         50 POP_JUMP_IF_FALSE 39 (to 78)

       5 52 LOAD_GLOBAL 2 (M)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        76 STORE_SUBSCR
     >> 78 JUMP_ABSOLUTE 16 (to 32)

   3 >> 80 JUMP_ABSOLUTE 7 (to 14)

   2 >> 82 LOAD_CONST 0 (None)
        84 RETURN_VALUE
```

「最悪」について W を計算した。[*9]

Bubble ソートについて「最良」は $W = 22$ であり，「最悪」は $W = 48$ となる。

Insertion ソート再評価の検討

Insertion ソートの処理時間を再評価する。Insertion ソートは挿入ソートとも呼ばれ，すでにソートされた部分的な配列に新たな要素を適切な位置に挿入することで全体をソートするアルゴリズムである。Insertion ソートの逆アセンブル結果を考察する点では while 文の内部構造に注意する必要がある。ここでの while 文は `position > 0` という条件と `M[position - 1] > current` という条件を AND により結合している。「最悪」の場合，この条件を true で返して while 文の中身が実行されるためには二つの条件がどちらとも満たされる必要があり，バイトコードでの 26 行目から 49 行目がすべて実行される。「最良」の場合，この条件が false で返されて while 文の中身が実行されないことになるが，これが満たされるのは `position < 0` または `M[position - 1] < current` のときである。ここで，while 文に入る前に `position = i` が定義されているため，前半の条件分によりはじかれることはない。被ソート配列が昇順になっている「最良」では後半の条件文によってはじかれる。[*10] よってこの場合も同様に逆アセンブル結果の 26 行目から 49 行目がすべて実行される。

「最悪」の場合について，ループは for 文と while 文で二重になり，最奥のループは逆アセンブル結果の 50 行目から 97 行目である。

「最良」の場合について，while はループしないため，このときの最奥のループは for 文となる。このため，Insertion ソートは $O(n)$ 型になり，ほかの $O(n^2)$ 型のアルゴリズムと計算時間を比較するために W を計算する必要はない。

Selection ソート再評価の検討

ここでは Selection ソートの処理時間再評価を行う。Selection ソートは最奥のループが「最良」「最悪」の両方において 2 層目の for 文である。このループ部分は逆アセンブル結果の 32 行目から 57 行目に対応し，「最悪」の場合について if 文のうちスキップされる部分はバイトコードの 52 行目から 55 行目である。よって「最悪」は $W' = 57 - 32 + 1 = 26$ であり，「最良」は $W = 26 - (55 - 52 + 1) = 26 - 4 = 22$

ここで注意したいのが，「最悪」の場合において，必

Listing 3: Insertion ソートの逆アセンブル

```
>>> import dis

>>> def InsertionSort():
      for i in range(1, length):
        position = i
        current = M[i]
        while position > 0 and M[position -
            1] > current:
          M[position] = M[position - 1]
          position -= 1
        M[position] = current

>>> dis.dis(InsertionSort)
      2 0 LOAD_GLOBAL 0 (range)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    >> 10 FOR_ITER 48 (to 108)
       12 STORE_FAST 0 (i)

     3 14 LOAD_FAST 0 (i)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     5 26 LOAD_FAST 1 (position)
       28 LOAD_CONST 2 (0)
       30 COMPARE_OP 4 (>)
       32 POP_JUMP_IF_FALSE 49 (to 98)
       34 LOAD_GLOBAL 2 (M)
       36 LOAD_FAST 1 (position)
       38 LOAD_CONST 1 (1)
       40 BINARY_SUBTRACT
       42 BINARY_SUBSCR
       44 LOAD_FAST 2 (current)
       46 COMPARE_OP 4 (>)
       48 POP_JUMP_IF_FALSE 49 (to 98)

  6 >> 50 LOAD_GLOBAL 2 (M)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       72 STORE_FAST 1 (position)

     5 74 LOAD_FAST 1 (position)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       94 COMPARE_OP 4 (>)
       96 POP_JUMP_IF_TRUE 25 (to 50)

  8 >> 98 LOAD_FAST 2 (current)
      100 LOAD_GLOBAL 2 (M)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      106 JUMP_ABSOLUTE 5 (to 10)

 2 >> 108 LOAD_CONST 0 (None)
      110 RETURN_VALUE
```

[*9] 「平均」はループ最奥にある分岐が単純ではなく，W を決定できない

[*10] AND で繋がれた二つの条件分を入れ替えることによって Insertion ソートの処理時間を削減することができる。

ずしも if 文の内容がスキップされずに処理されるとは限らないということである。

Selection ソートでは要素の中で最小のものを見つけ，配列の先頭と交換するソートの方法であり，降順で定義された「最悪」について配列の前半分を処理すると配列のうしろ半分はおのずと昇順になっている。このことを考慮した上で W を計算したい。うしろ半分は昇順になった長さ $\frac{n}{2}$ の配列のソートであり，これの処理時間をかりに 1 とする。そのとき，前半分は長さ n の配列のソートの一部であり Selection ソートは $O(n^2)$ であるためこの処理時間は $2^2 - 1 = 3$ となる。よって「最悪」について

$$W = \frac{26 \times 3 + 22 \times 1}{3 + 1} = \frac{100}{4} = 25$$

であるといえる。

$O(n \log n)$ 型再評価の検討

Listing 5: MergeSort のソースコード

```
def mergeSort(List):
    n = len(List)
    if n < 2:
        return
    mid = n // 2
    S1 = List[0:mid]
    S2 = List[mid:n]
    mergeSort(S1)
    mergeSort(S2)
    merge(S1, S2, List)

def merge(S1, S2, List):
    i = j = 0
    while i + j < len(List):
        if j == len(S2) or (i < len(S1)
            and S1[i] < S2[j]):
            List[i + j] = S1[i]
            i += 1
        else:
            List[i + j] = S2[j]
            j += 1
```

$O(n \log n)$ 型のアルゴリズムは条件分岐が $O(n^2)$ 型のそれよりも込み入っているため，「最良」や「最悪」などという範疇で考察するのではなく，完全シャッフルの「平均」の場合にどのような挙動をするかを考察する。

Listing 4: Selection ソートの逆アセンブル

```
>>> import dis

>>> def SelectionSort():
      for i in range(0, length):
        min_pos = i
        for j in range(i + 1, length):
          if M[j] < M[min_pos]:
            min_pos = j
         M[min_pos], M[i] = M[i], M[min_pos]


>>> dis.dis(SelectionSort)
      2 0 LOAD_GLOBAL 0 (range)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        8 GET_ITER
    >> 10 FOR_ITER 37 (to 86)
       12 STORE_FAST 0 (i)

     3 14 LOAD_FAST 0 (i)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       30 GET_ITER
    >> 32 FOR_ITER 12 (to 58)
       34 STORE_FAST 2 (j)

     5 36 LOAD_GLOBAL 2 (M)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       48 COMPARE_OP 0 (<)
       50 POP_JUMP_IF_FALSE 28 (to 56)

     6 52 LOAD_FAST 2 (j)
       54 STORE_FAST 1 (min_pos)
    >> 56 JUMP_ABSOLUTE 16 (to 32)

  7 >> 58 LOAD_GLOBAL 2 (M)
       60 LOAD_FAST 0 (i)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       82 STORE_SUBSCR
       84 JUMP_ABSOLUTE 5 (to 10)

  2 >> 86 LOAD_CONST 0 (None)
       88 RETURN_VALUE
```

ここでは，Merge ソートについて処理時間再評価を検討する。Merge ソートは被ソート関数を分割していく操作を繰り返し，その要素数が 1 になるまで細分化したのち，その要素同士を整列しながら結合（merge）することでソートするアルゴリズムである。そのソースコードは Listing5 に示した。

長さ n の配列が与えられたときの `mergeSort()` 関数の処理時間を a_n と，`merge()` 関数の処理時間を b_n とおく。`mergeSort()` 関数に長さ n の配列 `List` が与えられた時，この配列中で長さ $\left[\frac{n}{2}\right]$ の配列 `S1` と長さ $n - \left[\frac{n}{2}\right]$ の配列 `S2` が生成される。なお，$[x]$ は x を超えない最大の整数を示す。ただしここで，n が十分に大

きいとき，$\left[\frac{n}{2}\right] \fallingdotseq \frac{n}{2}$，$n - \left[\frac{n}{2}\right] \fallingdotseq \frac{n}{2}$ に近似できる。逆アセンブルの結果は示す余地がないが，`mergeSort()` 関数は 83 行のバイナリコードによって処理されていることがわかる。よって，以下の漸化式を導ける。

$$a_n \fallingdotseq 83 + 2a_{\frac{n}{2}} + b_n$$

また，b_n すなわち `merge()` 関数の処理時間について考える。この関数内では，与えられたふたつの配列 `S1` および `S2` の要素を昇順に `List` 配列に代入する操作が行われている。ここで，if 文における分岐の性質について分析する。配列 `S1` の要素を `List` に格納するとき，`else:` 以前が実行され，`i` がインクリメントされる。逆に，配列 `S2` の要素を `List` に格納するとき，`else:` 以降が実行され，`j` がインクリメントされる。n が十分大きいときに `S1` および `S2` の大きさは等しいと考えてよく，すなわち `else:` 以前と `else:` 以降とはほぼ同じ回数だけ実行されるということである。よって，ソースコード中の if 文について `else:` 以前と `else:` 以降の平均が if 文全体の “相対的な重さ” となる。if 文の内部の “相対的な重さ” が 50 であり，分岐した時の平均は 2 で割った 25 である。また if 文自体が 40 であり，if 文の外部であり while 文以内にあるのが 16 であるため $b_n \fallingdotseq (25 + 40 + 16)n \fallingdotseq 81n$ である。したがって，a_n の漸化式について，n が十分に大きいとき $83 + n \fallingdotseq n$ と近似してよく

$$a_n \fallingdotseq +83 + 2a_{\frac{n}{2}} + 81n \fallingdotseq 2a_{\frac{n}{2}} + 81n$$

関数は再起するごとに n は半分になり，この操作が 1 になるまで繰り返されるため，これは $\log_2 n$ 層重なる。また，m 層目は 2^{m-1} 個の関数に分割されている。ここで，数列 $\{c_n\}$ を $c_m = 2^{m-1}a_{\frac{n}{2^m}} = c_{m+1} + 2^{m-1} \times 81\frac{n}{2^m} = c_{m+1} + \frac{81}{2}n$ とすると，これは $\log_2 n$ 層目から m 層目までの処理時間の合計値である。数列 $\{d_n\}$ を階差的に $d_m = c_{m+1} - c_m = \frac{81}{2}n$ とすると，これは m 層目だけにおける処理時間の合計値である。これが $\log_2 n$ だけ重なっているため，処理時間を合計すると $a_n = \frac{81}{2}n\log_2 n$ であるといえる。ここでの 81 を W とする。[*11]

同様にしてほかの $O(n\log n)$ ソートについても W を計算した。なお，スペースの関係で Quick ソートについては結果のみを示すこととする。($W = 35$)

Heap ソートの W は算出することができなかった。$O(n\log n)$ 型の他のソートアルゴリズム，すなわち Merge ソートおよび Quick ソートはどちらもその処理時間が $a_n = 2 \times a_{\frac{n}{2}}$ であらわされたが，Heap ソートは同じような構造ではないことが原因と考えられる。右に示すのが Heap ソートのソースコードである。

ここで, `heapSort()` 関数の処理時間を a_n, `downheap()` 関数の処理時間を b_n とすると，これらに成立する関係式は

$$a_n \fallingdotseq \sum_{k=1}^{\frac{n}{2}} b_k + \sum_{k=1}^{n} b_{k-1}$$

であり，関数の内部構造が Merge ソートや Quick ソートのそれとは大きく異っているといえる。また，この差異が **D 考察-実験 1** にて判明した $O(n\log n)$ 型における「傾きの誤差」の原因になっていると考えられる。

Listing 6: HeapSort のソースコード

```
1  def heapSort():
2      for i in range((length - 1) // 2,
           -1, -1):
3          downheap(i, length - 1)
4      for i in range(length - 1, 0, -1):
5          M[0], M[i] = M[i], M[0]
6          downheap(0, i - 1)
7
8  def downheap(left, right):
9      tmp = M[left]
10     child = None
11     parent = left
12     while parent < (right + 1) // 2:
13         cl = parent * 2 + 1
14         cr = cl + 1
15         if cr <= right and M[cr] > M[cl
               ]:
16             child = cr
17         else:
18             child = cl
19         if tmp >= M[child]:
20             break
21         M[parent] = M[child]
22         parent = child
23     M[parent] = tmp
```

総合評価

実験 1 で計測した処理時間をそれぞれの W で除した。なお,「平均」は考慮していない。グラフは次頁冒頭に示す

[*11] 配列を 2 分割していく構造を持つ場合，log の底は 2 になるため無視する．また，分母は同様の考察をしたときに必ず 2 になるため無視してよい

図 4 のようになった。

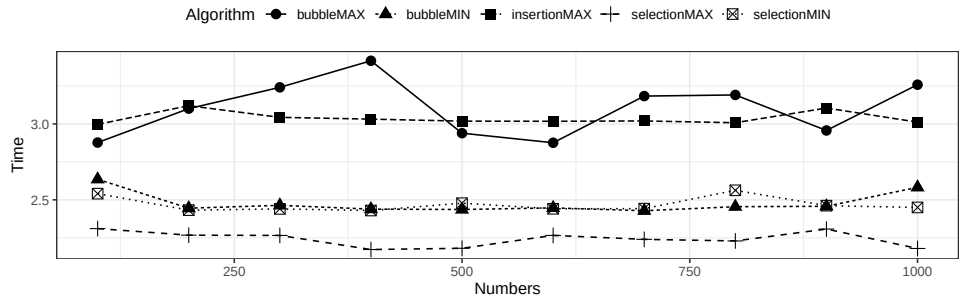


図 4: $O(n^2)$ 型アルゴリズムの修正済評価との誤差評価

これと W で除する前のデータとを比較するため，以下の検証をした。まず，W で除する前の処理時間 ($= A$ とする) と除したあとの処理時間 ($= A/W$) について，ともに $n^2 \times 10^{-9}$ で割った結果を計算する。ついで，それぞれのデータの平均値と標準偏差[*12]を計算後，標準偏差/平均値の値を**ばらつきの度合い**として用いる。その結果，下に示す表 2 のようになった。ただし，B は $A \times 10^9/n^2$ の平均値であり，B/W は $A \times 10^9/(W \times n^2)$ の平均値である。

表 2: ばらつきの度合いの計算

Algorithm		W	B	B/W	Algorithm	W	B	B/W
Bubble	最良	22	27.2686693	1.23948497	Merge	81	250.0034537	3.086462391
	最悪	48	74.4970717	1.55202233	Quick	35	138.9740454	3.970687013
Insertion	最悪	48	72.8952761	1.51865159				
Selection	最良	22	27.1473644	1.23397111				
	最悪	25	29.1457744	1.16582978				
平均値			46.1908252	1.34199195	平均値		194.4887496	3.528574702
標準偏差			25.1277111	0.17925588	標準偏差		78.50964746	0.625241226
標準偏差/平均値			0.54299788	0.13357448	標準偏差/平均値		0.403671923	0.177193705

表からもわかるように，W を用いた修正を行うことによって標準偏差/平均値，すなわち “ばらつきの度合い” が減少する。つまり，はじめ O 記法がもっていた「定数倍の誤差」を軽減できるといえる。

E. 結論

O 記法には主にふたつの誤差があり，それは「定数倍の誤差」「傾きの差異」である。「定数倍の誤差」は，異なるアルゴリズム同士で処理時間の比較を行ったときにそれらに定数倍のちがいがあることであり，これは構造が類似したプログラム同士を比較したときの，それらの W の大小と大きな相関があることが判明した。W とは，逆アセンブルよって知ることができるバイトコードの行数のうち，最奥のループにあたる部分のことである。また，「傾きの差異」は処理時間をグラフにとったときにアルゴリズムごとにその増加度合いが異なることであり，傾きが異なるアルゴリズム同士では内部構造が大きく異なるという特徴があった。

しかし，本研究でデータを収集したのは主要な 6 種のアルゴリズムに留まっており，理論を結論づけるには

[*12] 引数を標本と見なし、標本に基づいて母集団の標準偏差の推定値を返す STDEX.S 関数（EXCEL 搭載）による。

十分ではないように思える。今後も研究が進み，同様の性質がほかのアルゴリズムにおいても確認されることで計算量評価についてあらたな発見が広がることを期待する。

また，W によってばらつきを軽減できるとはいえ，ソフトの面でのアプローチにとどまっており，ハード的な要因などさまざまな要因により実際の処理時間とは誤差がいまだ残っている。今後の更なる研究によりこの誤差がより軽減されていくことが，計算機の活用の発展に貢献していくだろう。

F. 謝辞

本研究の遂行にあたり，ご指導をいただいた打田孝一先生，田中洋先生に深く感謝いたします。先生方の監督のもと 1 年半にわたり，素晴らしい環境で研究・実験をすることができました。探究活動に際して，とくに人に「伝える」ことについてのご指導をいただき，たいへん大きな学びとなりました。また，三浦謙一先生にはお忙しい中時間を設けて本研究に関してご助言をいただきました。深く感謝します。また，岡野達雄先生ならびに加戸百合先生には三浦先生との対談の機会を設けていただいただけでなく，研究に関するご助言をいただきました。お礼申し上げます。最後に，日比谷高校理数探究 I 情報科選択の 4 人のメンバーおよび兄ははじめから最後までとても良い相談相手でした。本研究の遂行には決して欠かせない仲間です。ありがとうございました。

G. 参考文献

- 明治大学「並べ替えのアルゴリズム」https://www.isc.meiji.ac.jp/~mizutani/python/sort_algorithm.html(アクセス日：2022 年 9 月 10 日)[*13]
- MMGames, 『苦しんで覚える C 言語』, 秀和システム. 2011 年
- ノーム・ニッサン, サイモン・ショッケン著, 斎藤 康毅訳,『コンピュータシステムの理論と実装 —モダンなコンピュータの作り方』, オライリー・ジャパン. 2015 年 (原著：Noam Nisan, Shimon Schocken, "*The Elements of Computing System*", The MIT Press, 2005)
- 大滝みや子, 岡嶋 裕史, 『令和 03 年【春季】【秋季】応用情報技術者 合格教本』, 技術評論者. 2020 年
- Paul E. Black, "big-O notation", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 6 September 2019. (参照 2023-04-03) Available from:https://www.nist.gov/dads/HTML/bigOnotation.html
- Python Software Foundation, "dis — Python バイトコードの逆アセンブラー Python3.11.4 ドキュメント", 2023-06-25, https://docs.python.org/ja/3/library/dis.html (参照 2023-08-05)
- 理化学研究所 計算科学研究機構 (AICS)."創薬とスパコン".2023-05-01.https://aics.riken.jp/jp/post-k/pi/drugdiscovery.html(参照 2023-07-22)
- 柴田望洋, 『新・明解 C 言語 実践編』, SB クリエイティブ. 2015 年
- 米田優峻, 『競技プログラミングの鉄則 アルゴリズム力と思考力を高める 77 の技術』, マイナビ出版. 2022 年

[*13] 現在は閲覧できない状態になっている。