2023 年度
システムプログラミング実験

第 2 回
テクニカルライティング
実験レポート

実験日：2023 年 10 月 10 日（水）
提出日：2023 年 10 月 10 日（水）
学修番号：22140003
氏名：佐倉仙汰郎

1 初めに

本書ではシステムプログラミング実験第２回の課題を実施した結果を報告する．課題は４つの異なるソートの手段について、計算量の違いについて解析するものである．解析結果をグラフに示し、その結果について考察を行う．

2 実験の説明

整列アルゴリズムの計算時間を計測し，その結果に基づき性能を評価する．評価を行うアルゴリズムは、

- バブルソート: f_{B}
- クイックソート: f_{Q}
- マージソート: f_{M}
- C++ Standard Template Library[*1] の sort 関数によるソート: f_{STL}

の４つである．これらの整列ソートは要素数を N として，$f_{\mathrm{B}}(N) = O(N^2), f_{\mathrm{Q}}(N) = O(N \ln N), f_{\mathrm{M}}(N) = O(N \ln N)$ となる。計算量の違いがどのように計算時間に影響するかを図を用いて解析していく．上記ソートについて計測を行い，横軸を要素数，縦軸を実行時間としたグラフを作成する．そして横軸縦軸は対数軸を使用する．要素数 N については対数軸をとるため、第 k 回の実験における要素数を N_k とし、次の式 (1) に示す．

$$M_k = 1.2M_{k-1}, N_k = R(M_k), k = 1, 2, ..., K - 1. \tag{1}$$

ここでは $M_1 = 32$ を選択した．M_1 の値が小さすぎると処理回数が増えるだけで、計算量の差異がわかりやすくなるわけではない．計算量の違いは N の値が大きくなる時に差が出るので $M_1 = 32$ は妥当な値である．今回は前述したとおり対数軸を使うので、N の値は等比数列で設定した．このようにして生成したデータ点の 2 次元グラフによる可視化と，実際の数値の値から結果を考察する．

3 実験結果

前節で説明した方法を C ＋＋言語により実装した．実装環境は以下のとおりである．

- Central Processing Unit: 11th Gen Intel(R) Core(TM) i7-1167G7 @ 2.80 GHz
- 主記憶: Double Data Rate 4 Synchronous Dynamic Random-Access Memory
- コンパイラ: g++ version 11.2.0
- コンパイルコマンド: g++ -std=c++14 sort.cpp (sort.cpp はソースプログラムのファイル名である．)
- Operating System: Arch Linux[*2]
- 数値型: 倍精度浮動小数点数[*3]

[*1] https://www.boost.org/sgi/stl/doc_introduction.html
[*2] https://archlinux.org/
[*3] https://www.gnu.org/software/gsl/doc/html/ieee754.html

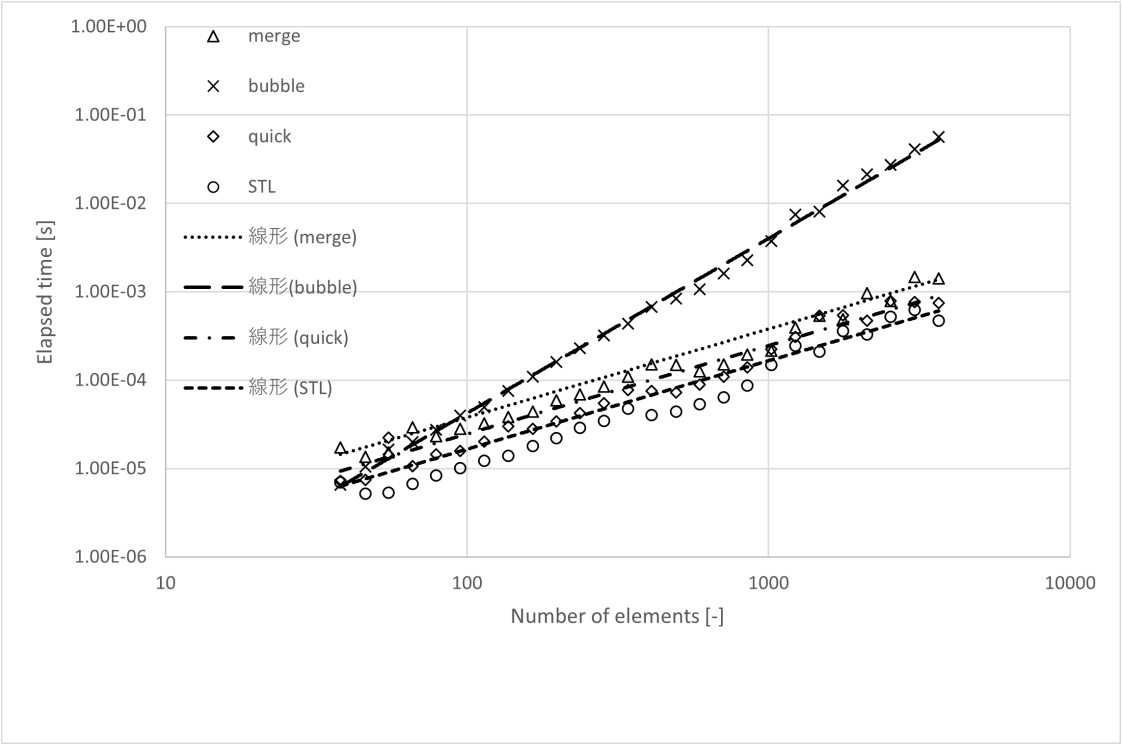

図１　実験結果のグラフ．４つのソートアルゴリズムの計算時間が縦軸、横軸が試行回数 N になっている．

４つの異なるソートアルゴリズムの計算量とそれに用いた要素数を軸とした図を示す．この図からバブルソートの計算量が N の値が大きくなるとほかのアルゴリズムの計算量との差が大きくなることが分かる．ほか３つのアルゴリズム、クイックソート、マージソート、C++ Standard Template Library に関しては回帰直線の傾きがある程度近似していることが分かる．バブルソートの計算量が $O(n^2)$ であることから、図（１）のバブルソートの回帰直線の傾きはおおよそ２になっている。ただし、これらの３つのソートアルゴリズムに有意な差はこの図からは読み取れないため、実際の数値を読み取る必要がある．
表 (1) より、N <= 66 の範囲では、バブルソートはマージソートよりも計算時間がかかっている．STL のソートは N のすべての範囲で計算量が一番早い．クイックソートは STL のソートの次に早いことがこの表 (1) からわかった．

4　考察

今回の実験により計算量の違いが有意な差であることがわかった．$O(n^2)$ であるバブルソートは図（１）からわかるように計算時間が寄りかかった．このことからバブルソートは膨大なデータに向かないこず、ほかのソートアルゴリズムを検討することが妥当である．また今回の実験で、計算量が同じソートアルゴリズムでも、それぞれに特徴があることが分かった．マージソートは計算量が $O(nlogn)$ でクイックソートと等しいが、N の値が小さいときにはクイックソートよりも遅いので、使い場合を選ぶ．以上より、アルゴリズムを選択するときには使用するデータセットの特徴を見極めそれに適したものを選ぶ必要があることがわかった．今回使用したソートアルゴリズムの中で唯一具体的な計算量が分からない STL のソートだが、クイックソートとマージソートと大差ないことから、$O(nlog(n))$ であると考えるのが妥当だろう．

表 1　要素数 N に対する 4 つのアルゴリズムの計算時間である.

N	merge	bubble	quick	STL	N	merge	bubble	quick	STL
38	0.0000173	0.0000065	0.0000073	0.0000070	410	0.0001500	0.0006722	0.0000756	0.0000404
46	0.0000136	0.0000105	0.0000074	0.0000052	493	0.0001495	0.0008352	0.0000727	0.0000443
55	0.0000152	0.0000165	0.0000223	0.0000054	591	0.0001252	0.0010661	0.0000891	0.0000533
66	0.0000291	0.0000198	0.0000107	0.0000067	709	0.0001498	0.0016033	0.0001094	0.0000639
79	0.0000232	0.0000275	0.0000144	0.0000084	851	0.0001954	0.0022816	0.0001388	0.0000863
95	0.0000283	0.0000400	0.0000158	0.0000101	1022	0.0002156	0.0037578	0.0002237	0.0001496
114	0.0000324	0.0000494	0.0000203	0.0000123	1226	0.0003923	0.0074607	0.0003071	0.0002443
137	0.0000384	0.0000750	0.0000299	0.0000139	1472	0.0005330	0.0080440	0.0005325	0.0002096
165	0.0000438	0.0001100	0.0000282	0.0000179	1766	0.0004890	0.0158310	0.0005427	0.0003573
198	0.0000592	0.0001612	0.0000339	0.0000221	2119	0.0009590	0.0212213	0.0004713	0.0003282
237	0.0000692	0.0002304	0.0000426	0.0000290	2543	0.0007882	0.0273119	0.0007873	0.0005225
285	0.0000850	0.0003221	0.0000549	0.0000346	3052	0.0014629	0.0411523	0.0007628	0.0006274
342	0.0001092	0.0004362	0.0000769	0.0000477	3663	0.0014079	0.0568365	0.0007464	0.0004689

5　おわりに

本書ではシステムプログラミング実験の課題として、整列アルゴリズムの計算時間を計測し，その結果に基づき性能を評価した。結果として計算量の違いにより計算時間に大きな差が出ることがわかった。また同じ計算量のアルゴリズムでもそれぞれに特色があり、状況に応じてソートアルゴリズムを選択する必要があることがわかった.

6　付録

図 (1) の作成に用いたソースコードをここに記載する。

```
1   #include <iostream>
2   #include <random>
3   #include <string>
4   #include <chrono>
5   #include <functional> // function library to define a function in a function
6   #include <fstream>
7   #include <algorithm>
8
9   using namespace std;
10
11  void mergeSort(int* vA, int N)
12  {
13      int* vM = vA;
14      int* vMt = new int[N];
15      function<void(int, int, int)> merge = [&](int left, int mid, int right) { //
            lambda function.
16      // このラムダ関数を使えば，再帰関数を定義するのに，わざわざグローバル領域に関数
            を宣言せずにすむ.
17      // ラムダ関数は関数の中に宣言できる特別な関数である.
18      // 宣言された関数内のローカル変数を参照できる.
19      // 構文は，14行目のとおりである.
20      // − function: ラムダ関数を入れる特別なクラス
21      // − void(int, int, int): 関数の型．int の引数がmergeソートの要件から3つほしいた
            め，このように定義している．返り値必要ないため void とした.
22      // = [&](int left, int mid, int right){}: 関数の処理内容の定義．& はローカル変数を
            ラムダ関数の中で改変可能とする宣言．ほかは普通の関数と同様の意味
23          int il = 0;
24          int ir = 0;
25          int el = mid − left;
26          int er = right − mid;
27          while (true) {
28              if (il == el) {
29                  if (ir == er) {
30                      return;
31                  } else {
32                      vMt[left + il + ir] = vM[mid + ir];
33                      ++ir;
34                  }
35              } else {
36                  if (ir == er) {
37                      vMt[left + il + ir] = vM[left + il];
38                      ++il;
39                  } else {
40                      auto tl = vM[left + il];
41                      auto tr = vM[mid + ir];
42                      if (tl <= tr) {
43                          vMt[left + il + ir] = tl;
```