CS3210 Tutorial 1

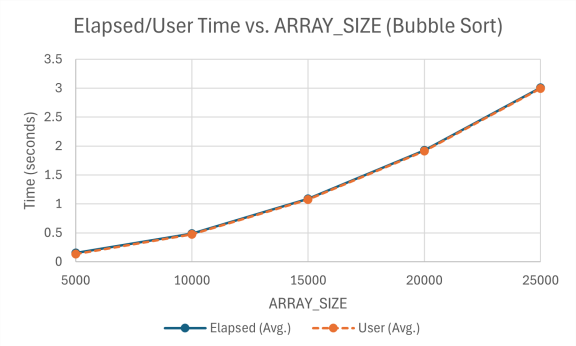
AY 25/26 Sem 2 — github/omgeta

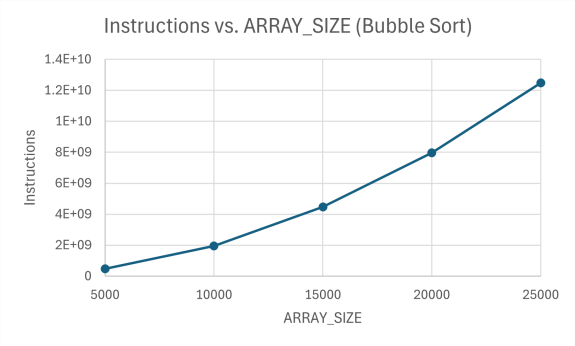
1 Overview

Firstly, the code initializes two integer constants, `ARRAY_SIZE` and `MAX_VALUE`. Secondly, a randomized array of `ARRAY_SIZE` integers, each in the range `[0, MAX_VALUE]`, is created with the helper function `std::vector<int> generateLargeArray`. Thirdly, the generated array is passed to a sorting helper function `void my_sort` which internally implements an in-place Bubble Sort. Finally, the last 5 elements of the sorted array are printed to `stdout`.

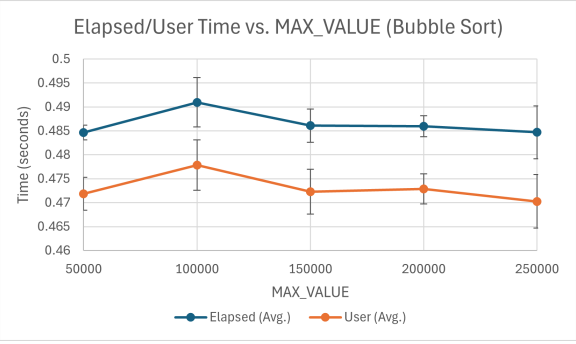
2 Description, Visualization and Data

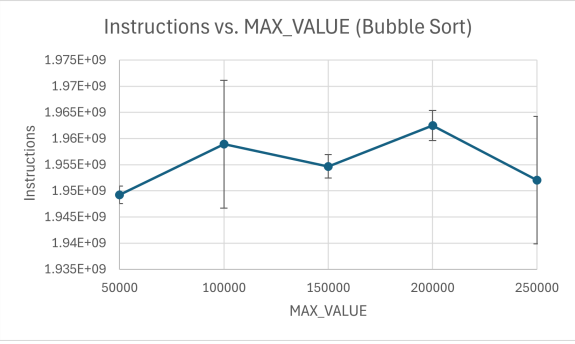
As `ARRAY_SIZE` varies, both elapsed time and instructions executed grow approximately quadratically. This matches the expected $O(n^2)$ running time of bubble sort (double-nested loop performs $\Theta(n^2)$ comparisons/swaps). Dominance of user time in elapsed time indicates that the runtime is largely CPU-bound, with cost coming mainly from the sorting itself. In particular, `my_sort` dominates the runtime for sufficiently large arrays.





In contrast, varying `MAX_VALUE` has little impact on both elapsed time and instructions. This is consistent with bubble sort as regardless of the value distribution, it still performs $\Theta(n^2)$ comparisons in the worst case. Therefore, we can deduce `ARRAY_SIZE` dominates performance cost, while `MAX_VALUE` is comparatively insignificant.



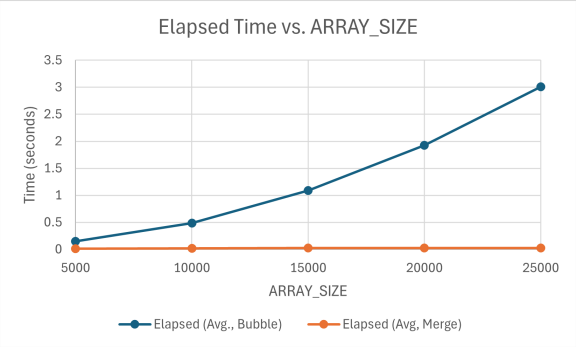


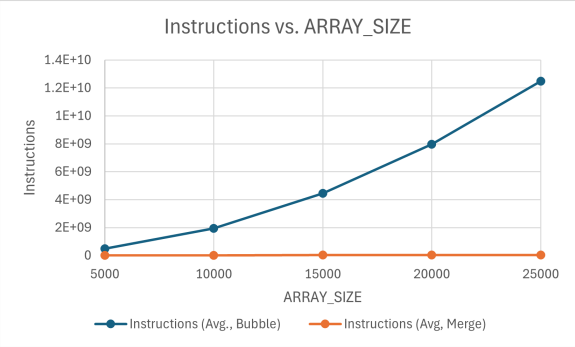
3 Optimization

Since we have noted the primary performance cost comes from the current implementation of `my_sort` with bubble sort, where both runtime and instruction count grow quadratically as a function of `ARRAY_SIZE`, we should consider a more efficient algorithm. Let us choose `ARRAY_SIZE=5000` as our definition of sufficiently large, while fixing `MAX_VALUE=100000` since it does not matter much to the performance.

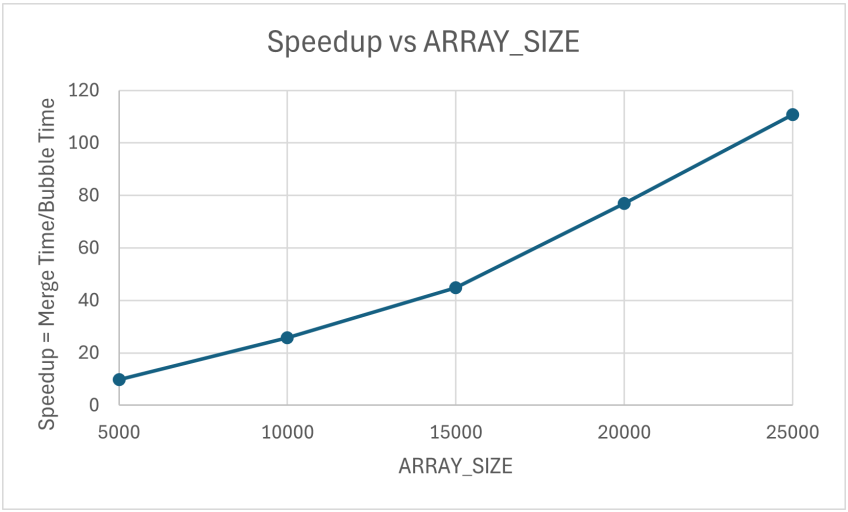
Consider (non-parallel) Merge Sort, an algorithm which runs in $\Theta(n \log n)$ and reduces the number of comparions significantly for sufficiently large n. Let us swap out the `my_sort` algorithm with the merge sort code in the appendix.

Now, as `ARRAY_SIZE` varies, we see not only elapsed time and instruction count much lower than in the old code, but which grows at a rate significantly slower, consistent with the expected improvement from $O(n^2)$ to $O(n \log n)$ time complexity.





Notably, as `ARRAY_SIZE` increases, the measures speedup $S(n) = \frac{T_{\text{bubble}}(n)}{T_{\text{merge}}(n)}$ grows signficantly. At a baseline of `ARRAY_SIZE=5000`, the speedup is $\sim 9\times$ while it grows up to $\sim 110\times$ at `ARRAY_SIZE=25000`. This is a significant speedup for sufficiently large arrays.



For further optimization, we can also consider Parallel Merge Sort.

Appendix

To control parameters `ARRAY_SIZE` and `MAX_VALUE` from the CLI, the `const int` local variables for the parameters are replaced by macros which default to the original values given in the code, allowing us to recompile with different parameters using the `-D` flag:

```
#ifndef ARRAY_SIZE
#define ARRAY_SIZE 10000 //default 10000 if -DARRAY_SIZE=<val> absent
#endif

#ifndef MAX_VALUE
#define MAX_VALUE 100000 //default 100000 if -DMAX_VALUE=<val> absent
#endif
```

Listing 1: Definition of parameter macros

To replace the inefficient bubble sort implementation with merge sort, replace the `my_sort` code segment as such:

```
void merge(std::vector<int>& array, size_t left, size_t mid, size_t
    right) {
  int n1 = mid - left + 1;

  // copy left half into separate std::vector
  // right half stays in array[m+1..r]
  std::vector<int> left_half(n1);
  for (size_t i = 0; i < n1; i++) left_half[i] = array[left + i];

  size_t i = 0;       // index in left
  size_t j = mid + 1; // index in right half of array
  size_t k = left;    // write index in array

  while (i < n1 && j <= right) {
    if (left_half[i] <= array[j]) array[k++] = left_half[i++];
    else array[k++] = array[j++];
  }

  // copy the remaining elements in the left half
  // if there are remaining elements, it means they are all < any
    element in j..r
  while (i < n1) array[k++] = left_half[i++];
}

void merge_sort(std::vector<int>& array, size_t left, size_t right) {
  if (left >= right) return;

  size_t mid = left + (right - left) / 2;
  merge_sort(array, left, mid);
  merge_sort(array, mid+1, right);
  merge(array, left, mid, right);
}

void my_sort(std::vector<int>& array) {
  merge_sort(array, 0, array.size() - 1);
}
```

Listing 2: Merge Sort Implementation

To collect the measurement data, a bash script was used to compile each program with `g++ -D<PARAMETER>=<VAL> -o <BIN> <SRC>` on the local machine, and then `srun perf stat -e cycles,instructions` thrice on the `i7-7700` partition for before and after optimization, compiling measurements in `out.csv`:

```
#!/usr/bin/env bash

ARRAY_SIZES=(5000 10000 15000 20000 25000)
MAX_VALUES=(50000 100000 200000 250000 300000)
ITERATIONS=3
OUT="out.csv"
RECORDED="cycles,instructions"
PARTITION="i7-7700"
PROGRAMS=("asdf" "asdf_merge")

perf_stat() {
  srun --partition "$PARTITION" perf stat -e "$RECORDED" "$1" 2>&1 1>/dev/null # perf
    spits out results in stderr
}

get_cycles() {
  awk '/cycles/ { gsub(/,/,"",$1); print $1; exit }'
}

get_instructions() {
  awk '/instructions/ { gsub(/,/,"",$1); print $1; exit }'
}

get_elapsed() {
  awk '/seconds time elapsed/ { gsub(/,/,"",$1); print $1; exit }'
}

get_user() {
  awk '/seconds user/ { gsub(/,/,"",$1); print $1; exit }'
}

# Measure across array size
echo "program,ARRAY_SIZE,elapsed,user,instructions,cycles" > "$OUT"
for P in "${PROGRAMS[@]}"; do
  for N in "${ARRAY_SIZES[@]}"; do
    g++ -DARRAY_SIZE=$N -o "$P" "$P".cpp

    for I in $(seq 1 "$ITERATIONS"); do
      out="$(perf_stat ./"$P")"
      elapsed="$(printf '%s\n' "$out" | get_elapsed)"
      user="$(printf '%s\n' "$out" | get_user)"
      instructions="$(printf '%s\n' "$out" | get_instructions)"
      cycles="$(printf '%s\n' "$out" | get_cycles)"
      echo "$P,$N,$elapsed,$user,$cycles,$instructions" >> "$OUT"
    done
  done
done

echo "" >> "$OUT"
echo "program,MAX_VALUE,elapsed,user,instructions,cycles" >> "$OUT"
for P in "${PROGRAMS[@]}"; do
  for N in ${MAX_VALUES[@]}; do
    g++ -DMAX_VALUE=$N -o "$P" "$P".cpp

    for I in $(seq 1 "$ITERATIONS"); do
      out="$(perf_stat ./"$P")"
      elapsed="$(printf '%s\n' "$out" | get_elapsed)"
      user="$(printf '%s\n' "$out" | get_user)"
      instructions="$(printf '%s\n' "$out" | get_instructions)"
      cycles="$(printf '%s\n' "$out" | get_cycles)"
      echo "$P,$N,$elapsed,$user,$cycles,$instructions" >> "$OUT"
    done
  done
done
```

Listing 3: measure.sh

Code, scripts, and output data is available for reference in this Dropbox folder.

Performance measurement data is compiled in table form below:

ARRAY_SIZE	Run	Elapsed (s)	User (s)	Instructions	Cycles
5000	1	0.159,930,326	0.143,906,000	476,978,443	1,069,169,100
5000	2	0.160,122,112	0.145,224,000	476,638,251	1,069,157,020
5000	3	0.127,528,080	0.115,556,000	477,736,360	1,068,972,456
10000	1	0.487,853,300	0.473,425,000	1,951,851,612	4,252,216,746
10000	2	0.480,106,495	0.466,597,000	1,952,801,517	4,252,108,216
10000	3	0.487,267,905	0.473,717,000	1,950,290,129	4,252,203,186
15000	1	1.101,197,667	1.088,891,000	4,494,854,285	9,549,779,648
15000	2	1.082,110,396	1.070,141,000	4,449,841,339	9,549,622,665
15000	3	1.078,566,927	1.066,499,000	4,442,731,862	9,549,554,877
20000	1	1.933,352,842	1.918,846,000	7,972,856,454	17,007,303,576
20000	2	1.928,956,772	1.916,875,000	7,952,027,167	17,007,088,498
20000	3	1.918,663,080	1.904,432,000	7,946,093,504	17,007,101,855
25000	1	3.003,776,120	2.991,833,000	12,475,906,951	26,584,012,562
25000	2	3.006,411,040	2.992,772,000	12,467,746,360	26,583,434,011
25000	3	3.019,920,333	3.007,931,000	12,511,704,934	26,583,222,226

Table 1: Bubble Sort (`asdf.cpp`) – Varying ARRAY_SIZE

ARRAY_SIZE	Run	Elapsed (s)	User (s)	Instructions	Cycles
5000	1	0.015,370,741	0.003,408,000	11,074,033	15,896,692
5000	2	0.014,762,198	0.002,162,000	11,131,558	15,931,968
5000	3	0.015,319,478	0.002,267,000	11,040,184	15,934,923
10000	1	0.018,302,150	0.005,069,000	18,943,533	29,143,012
10000	2	0.018,849,097	0.005,158,000	18,862,514	29,155,558
10000	3	0.019,480,664	0.006,225,000	19,404,870	29,148,250
15000	1	0.019,290,581	0.007,281,000	27,108,451	42,579,047
15000	2	0.034,598,408	0.020,217,000	27,235,476	42,695,544
15000	3	0.018,872,232	0.006,166,000	27,132,801	42,557,496
20000	1	0.027,419,321	0.013,342,000	35,182,072	56,686,217
20000	2	0.024,090,875	0.010,276,000	35,284,573	56,680,530
20000	3	0.023,702,617	0.011,122,000	35,227,814	56,658,465
25000	1	0.026,166,625	0.010,742,000	44,017,716	70,798,906
25000	2	0.026,442,287	0.014,704,000	43,448,860	70,805,738
25000	3	0.028,855,132	0.016,029,000	43,742,174	70,852,855

Table 2: Merge Sort (`asdf_merge.cpp`) – Varying ARRAY_SIZE

MAX_VALUE	Run	Elapsed (s)	User (s)	Instructions	Cycles
50000	1	0.482,947,124	0.467,903,000	1,951,128,484	4,240,356,993
50000	2	0.485,911,062	0.473,535,000	1,948,452,045	4,240,358,505
50000	3	0.485,154,010	0.474,100,000	1,948,067,597	4,240,380,203
100000	1	0.485,053,359	0.471,868,000	1,949,944,521	4,252,064,241
100000	2	0.493,735,460	0.479,877,000	1,972,822,808	4,252,167,656
100000	3	0.494,105,320	0.481,807,000	1,953,980,465	4,252,175,191
150000	1	0.482,237,810	0.467,086,000	1,954,400,971	4,246,119,769
150000	2	0.487,014,623	0.473,701,000	1,957,033,045	4,245,707,279
150000	3	0.488,984,955	0.476,069,000	1,952,612,107	4,245,831,052
200000	1	0.484,068,831	0.471,029,000	1,964,176,708	4,269,852,977
200000	2	0.488,394,434	0.476,459,000	1,964,171,608	4,269,856,693
200000	3	0.485,452,746	0.471,042,000	1,959,165,290	4,269,939,050
250000	1	0.480,186,128	0.466,198,000	1,944,537,723	4,243,788,025
250000	2	0.490,875,638	0.476,621,000	1,966,103,339	4,243,931,409
250000	3	0.483,001,732	0.467,986,000	1,945,462,798	4,243,825,404

Table 3: Bubble Sort (`asdf.cpp`) – Varying `MAX_VALUE`

MAX_VALUE	Run	Elapsed (s)	User (s)	Instructions	Cycles
50000	1	0.030,096,830	0.015,575,000	19,084,927	29,238,902
50000	2	0.018,364,161	0.003,936,000	19,612,452	29,156,557
50000	3	0.017,731,149	0.004,228,000	19,048,229	29,131,031
100000	1	0.018,274,987	0.004,664,000	18,966,112	29,129,632
100000	2	0.020,920,694	0.007,625,000	19,001,601	29,173,433
100000	3	0.018,837,875	0.005,225,000	18,954,088	29,134,584
150000	1	0.017,947,129	0.004,776,000	19,031,190	29,170,474
150000	2	0.022,258,463	0.006,691,000	18,878,788	29,170,504
150000	3	0.020,920,319	0.006,872,000	18,899,067	29,183,003
200000	1	0.016,698,237	0.004,959,000	18,951,850	29,149,145
200000	2	0.016,659,872	0.004,599,000	18,979,485	29,139,834
200000	3	0.018,510,100	0.004,455,000	19,053,252	29,202,986
250000	1	0.019,037,782	0.005,582,000	18,888,059	29,124,090
250000	2	0.018,373,205	0.004,995,000	18,975,864	29,149,748
250000	3	0.017,659,391	0.003,479,000	18,973,178	29,120,563

Table 4: Merge Sort (`asdf_merge.cpp`) – Varying `MAX_VALUE`

AI Tool Declaration

I used GPT-5.2 to format my measurement data into the tables above. I am responsible for the content and quality of the submitted work.