CS3210 Lab 1 Report

Wang Xiyu

August 27, 2025

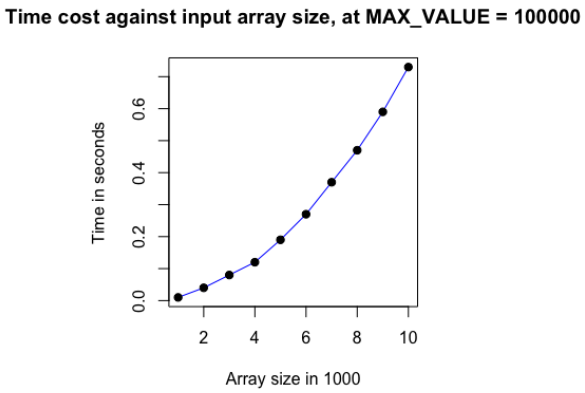
1 Code Overview

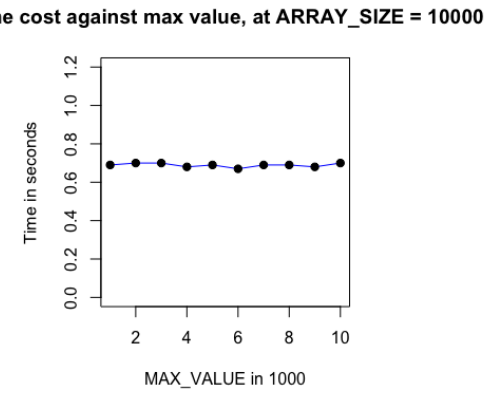
The provided code describes the bubble sort algorithm which is known to be sub-optimal ($\Theta(n^2)$). This function sorts an array by comparing every pair of values in the array, and swap when out of order. A random large array generation utility function and the main function to test the program are also included. These are out of the scope of discussion.

2 Performance

At `ARRAY_SIZE = 100000`, `MAX_VALUE = 100000`, the time statistic from running `/usr/bin/time ./asdf` is as such:

```
Command being timed: "./asdf"
User time (seconds): 71.48
System time (seconds): 0.00
Percent of CPU this job got: 99%
Elapsed (wall clock) time (h:mm:ss or m:ss): 1:11.49
```



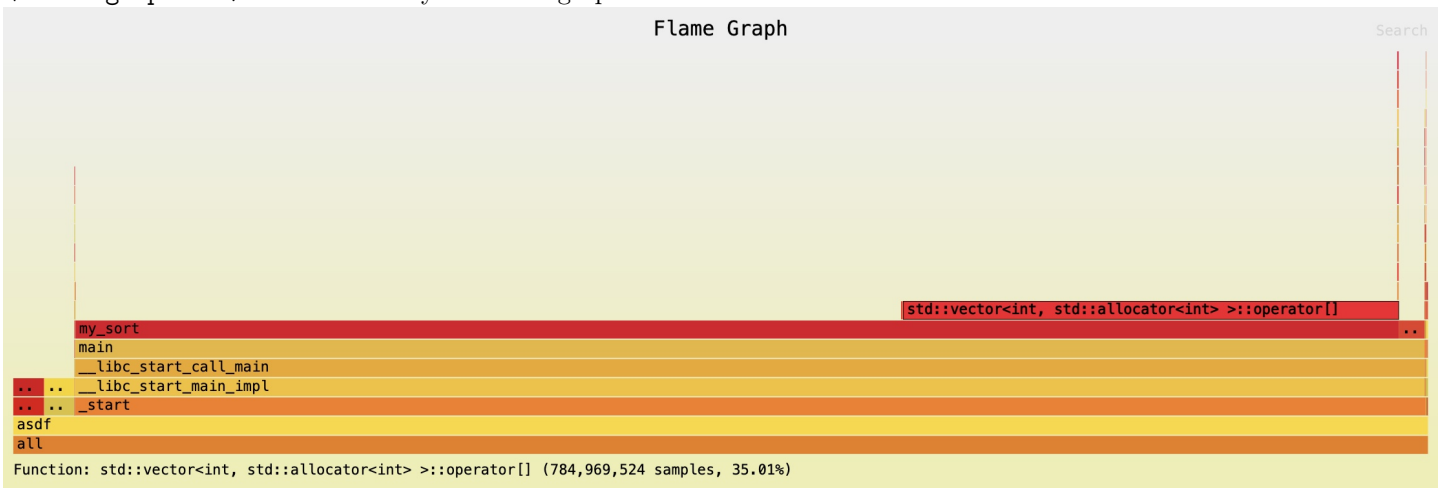


It's observed that the time growth in a quadratic manner as the array size increases, align with the common knowledge that bubble sort has a time complexity of $\Theta(n^2)$.
It's also observed that `MAX_VALUE` has no substantial impact on the performance.

3 Optimization

Performance bottleneck

`$ flamegraph - ./asdf "10000"` yields flamegraph as shown below:



Since we do not focus on improve the algorithm, we foucs on the overhead named `std::vector<int, std::allocator<int> >::operator[]`, which is the cost associated with access elements in `vector` container, within `my_sort()` function call.

```
$ perf record -- ./asdf 10000
$ perf report
Overhead Command Shared Object       Symbol
 61.23%  asdf    asdf                [.] my_sort(std::vector<int, std::allocator<int> >&)
 38.32%  asdf    asdf                [.] std::vector<int, std::allocator<int> >::operator[](unsigned long)
```

Hypothesis and Implementation

We hypothesize that this overhead caused by accessing the internal array elements of a vector can be optimized by accessing via pointers directly. Although doing so does not change the asymptotic order of growth, the saved random access constant overhead would have significant performance improvement for large n in practice.

Original

```
void my_sort(std::vector<int>& array) {
    size_t n = array.size();

    for (size_t i = 0; i < n - 1; ++i) {
        for (size_t j = 0; j < n - i - 1; ++j) {


            if (array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
}}}}
```

Optimized

```
void my_sort_o(std::vector<int>& array) {
    size_t n = array.size();
    int* init_ptr = array.data();
    for (size_t i = 0; i < n - 1; ++i) {
        for (size_t j = 0; j < n - i - 1; ++j) {
            int* cur = init_ptr + j;
            int* nxt = cur + 1;
            if (*cur > *nxt) {
                int t = *cur;
                *cur = *nxt;
                *nxt = t;
}}}}
```

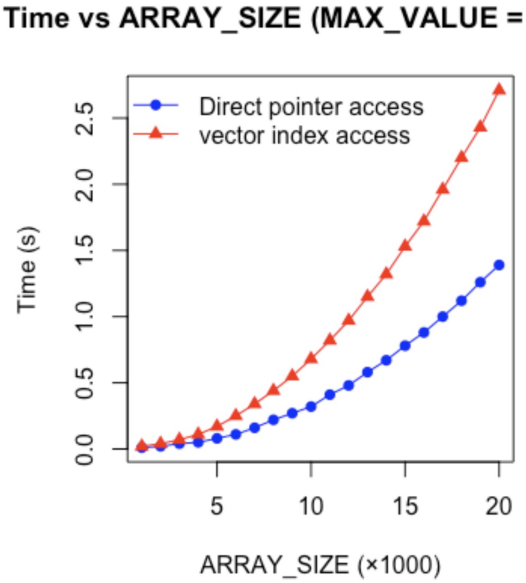
4 Reproduce result

Compile the original code `asdf.cpp` and optimized version `asdfo.cpp`,

```
g++ -g -o asdf asdf.cpp
g++ -g -o asdfo asdfo.cpp
```

Run `./time_test.sh` with custom steps and `ARRAY_SIZE` and check `res.txt` and `res_o.txt` for runtime log.

5 Performance measurement



We can see at large n ($n \geq 10000$), the optimzed version has $\sim$ 50% decrease in runtime, while retaining quadratic increasing trend, which proves our hypothesis that the constant access cost of vector element can be optimized.

6 Appendix

All resources used in preparing the diagrams, e.g. scripts, modified code can be found here: Link