

# CSCI 255: Lab #10 Sorting Performance

Darwin Jacob Groskleg

Thursday, November 21st, 2019

## Contents

<b>Contents</b>	<b>i</b>
<b>Program Output &amp; Questions</b>	<b>1</b>
Q1. Are the set of timings for n=5000 as you expected? . . . . .	1
Q2. Which algorithm is better for data already almost ordered? . . . . .	2
Q3. Is the timings with n=50 as expected? . . . . .	2
<b>main.cpp</b>	<b>3</b>
<b>sort.hpp</b>	<b>6</b>
<b>sort.cpp</b>	<b>7</b>
<b>benchmark.hpp</b>	<b>11</b>
<b>sort_test.cpp</b>	<b>13</b>

## Program Output & Questions

```

~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab10-Sorting  gmake benchmark
Makefile:40: warning: overriding recipe for target 'test'
/Users/darwingroskleg/Dropbox/cxx_templates/MakeMake.v1.mk:125: warning: ignoring old recipe for target 'test'
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O3 -o main.o -c main.cpp
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O3 -o sort.o -c sort.cpp
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O3 -o main.out main.o sort.o
time ./main.out
A) Sorting an unordered array of 5000 random integers:
    InsertionSort took 4460.0 µs, 6321728 moves, 6321728 comparisons
    SelectionSort took 8734.0 µs, 10000 moves, 12497500 comparisons
    * QuickSort took 274.0 µs, 27131 moves, 77314 comparisons
    MergeSort took 942.0 µs, 55241 moves, 55208 comparisons
B) Sorting a pre-sorted array of 5000 random integers:
    * InsertionSort took 10.0 µs, 4999 moves, 4999 comparisons
    SelectionSort took 7964.0 µs, 10000 moves, 12497500 comparisons
    QuickSort took 24181.0 µs, 4999 moves, 12512497 comparisons
    MergeSort took 694.0 µs, 2237 moves, 32427 comparisons
C) Sorting a pre-sorted array of 5000 in reverse order:
    InsertionSort took 9608.0 µs, 12501294 moves, 12501294 comparisons
    SelectionSort took 9494.0 µs, 10000 moves, 12497500 comparisons
    QuickSort took 17839.0 µs, 7499 moves, 12507497 comparisons
    * MergeSort took 737.0 µs, 61808 moves, 29804 comparisons
D) Sorting an unordered array of 50 random integers:
    InsertionSort took 3.0 µs, 585 moves, 585 comparisons
    SelectionSort took 3.0 µs, 100 moves, 1225 comparisons
    * QuickSort took 2.0 µs, 129 moves, 288 comparisons
    MergeSort took 10.0 µs, 210 moves, 228 comparisons
E) Sorting a pre-sorted array of 50 random integers:
    * InsertionSort took 1.0 µs, 49 moves, 49 comparisons
    SelectionSort took 2.0 µs, 100 moves, 1225 comparisons
    QuickSort took 5.0 µs, 49 moves, 1372 comparisons
    MergeSort took 7.0 µs, 0 moves, 153 comparisons
F) Sorting a pre-sorted array of 50 in reverse order:
    * InsertionSort took 3.0 µs, 1274 moves, 1274 comparisons
    SelectionSort took 3.0 µs, 100 moves, 1225 comparisons
    QuickSort took 3.0 µs, 74 moves, 1322 comparisons
    MergeSort took 8.0 µs, 286 moves, 133 comparisons
0.10 real    0.08 user    0.00 sys
~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab10-Sorting

```

Figure 1: Console Sample: fastest is starred \*.

### Q1. Are the set of timings for $n=5000$ as you expected?

I expected either QuickSort or MergeSort to be fastest given their average running time of  $O(n \log n)$ , leaning towards QuickSort since there's no need to make copies. This was true for case A.

Case B (pre-sorted) with InsertionSort being fastest was a surprise given its time complexity of  $O(n^2)$  but makes sense when you notice that it had the fewest number of comparisons by far. This from its adaptive property.

MergeSort winning case C is reasonable given its time complexity, while Quicksort had significantly fewer moves MergeSort made 3 magnitudes of fewer comparisons. This makes sense since the MergeSort implementation used does a few more moves than is necessary while still correct.

**Q2. Which algorithm is better for data already almost ordered?**

The generally poor InsertionSort is the best algorithm to use for data that is almost ordered. Performing significantly better than the others.

**Q3. Is the timings with  $n=50$  as expected?**

The timings with  $n=50$  were not as expected if making predictions based on time and space complexity of the algorithms. Overall InsertionSort faired the best, with it being tied with SelectionSort for reverse-ordered data. This is expected because InsertionSort has a smaller constant factor than QuickSort/MergeSort (i.e. overhead) so when  $n$  is small those factors are more prominent in the performance. The overhead does however seem algorithmic, using optimizations does not seem to improve performance.

**main.cpp**

```

1  /* main.cpp
2  * -----
3  * Authors: Darwin Jacob Groskleg
4  * Date:    Thursday, November 21, 2019
5  *
6  * QUESTIONS:
7  * 1. Are the set of timings for n=5000 as you expected?
8  *     I expected either QuickSort or MergeSort to be fastest given their
9  *     average running time of  $O(n \log n)$ , leaning towards QuickSort since
10  *     there's no need to make copies. This was true for case A.
11  *
12  *     Case B (pre-sorted) with InsertionSort being fastest was a surprise
13  *     given its time complexity of  $O(n^2)$  but makes sense when you notice that
14  *     it had the fewest number of comparisons by far. This from its adaptive
15  *     property.
16  *
17  *     MergeSort winning case C is reasonable given its time complexity, while
18  *     Quicksort had significantly fewer moves MergeSort made 3 magnitudes of
19  *     fewer comparisons. This makes sense since the MergeSort implementation
20  *     used does a few more moves than is necessary while still correct.
21  *
22  * 2. Which algorithm is better for data already almost ordered?
23  *     The generally poor InsertionSort is the best algorithm to use for data
24  *     that is almost ordered. Performing significantly better than the others.
25  *
26  * 3. Is the timings with n=50 as expected?
27  *     The timings with n=50 were not as expected if making predictions based
28  *     on time and space complexity of the algorithms. Overall InsertionSort
29  *     faired the best, with it being tied with SelectionSort for
30  *     reverse-ordered data. This is expected because InsertionSort has a
31  *     smaller constant factor than QuickSort/MergeSort (i.e. overhead) so when
32  *     n is small those factors are more prominent in the performance.
33  *     The overhead does however seem algorithmic, using optimizations does not
34  *     seem to improve performance.
35  *
36  * CONSOLE SAMPLE
37  =====
38  $> gmake
39  clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main.o -c m
40  clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o sort.o -c s
41  clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main.out ma
42
43  $> time ./main.out
44  time ./main.out
45  A) Sorting an unordered array of 5000 random integers:
46      InsertionSort took 19479.0  $\mu$ s, 6222810 moves, 6222810 comparisons
47      SelectionSort took 26618.0  $\mu$ s, 10000 moves, 12497500 comparisons
48      * QuickSort took 487.0  $\mu$ s, 26681 moves, 79745 comparisons
49      MergeSort took 1602.0  $\mu$ s, 55243 moves, 55256 comparisons
50  B) Sorting a pre-sorted array of 5000 random integers:
51      * InsertionSort took 23.0  $\mu$ s, 4999 moves, 4999 comparisons
52      SelectionSort took 24959.0  $\mu$ s, 10000 moves, 12497500 comparisons
53      QuickSort took 31422.0  $\mu$ s, 4999 moves, 12512497 comparisons
54      MergeSort took 1222.0  $\mu$ s, 2339 moves, 32430 comparisons

```

```

55 C) Sorting a pre-sorted array of 5000 in reverse order:
56     InsertionSort took 39624.0 µs, 12501232 moves, 12501232 comparisons
57     SelectionSort took 28290.0 µs, 10000 moves, 12497500 comparisons
58     QuickSort took 31107.0 µs, 7497 moves, 12507501 comparisons
59     * MergeSort took 1289.0 µs, 61808 moves, 29804 comparisons
60 D) Sorting an unordered array of 50 random integers:
61     * InsertionSort took 4.0 µs, 553 moves, 553 comparisons
62     SelectionSort took 7.0 µs, 100 moves, 1225 comparisons
63     QuickSort took 4.0 µs, 121 moves, 322 comparisons
64     MergeSort took 13.0 µs, 216 moves, 220 comparisons
65 E) Sorting a pre-sorted array of 50 random integers:
66     * InsertionSort took 1.0 µs, 49 moves, 49 comparisons
67     SelectionSort took 3.0 µs, 100 moves, 1225 comparisons
68     QuickSort took 6.0 µs, 49 moves, 1372 comparisons
69     MergeSort took 10.0 µs, 0 moves, 153 comparisons
70 F) Sorting a pre-sorted array of 50 in reverse order:
71     InsertionSort took 5.0 µs, 1274 moves, 1274 comparisons
72     * SelectionSort took 4.0 µs, 100 moves, 1225 comparisons
73     QuickSort took 6.0 µs, 74 moves, 1322 comparisons
74     MergeSort took 11.0 µs, 286 moves, 133 comparisons
75 =====
76 */
77 #include <iostream>
78 #include <iomanip>
79 #include <vector>
80 #include <algorithm>
81
82 #include <time.h>
83 #include <stdlib.h>
84
85 #include "sort.hpp"
86 #include "benchmark.hpp"
87
88 using namespace std;
89 using DetailedCSorter = std::function<SortingDetails (int[], int, int)>;
90 using DetailedBenchmarkEntry = BenchmarkEntry<SortingDetails>;
91
92 void display_results(vector<DetailedBenchmarkEntry>);
93
94 int main() {
95     CSortBenchmark< DetailedCSorter > benchmark({
96         {"InsertionSort", InsertionSort},
97         {"SelectionSort", SelectionSort},
98         {"QuickSort", QuickSort},
99         {"MergeSort", MergeSort}
100     });
101
102     srand((unsigned)time(NULL));
103
104     // BUILD 5000 random element array
105     vector<int> rand5000(5000);
106     // generate random numbers and store them in the array
107     for(size_t index=0; index<rand5000.size(); index++)
108         rand5000.at(index) = (rand()%10000)+1;
109     auto args_5_000 = make_tuple(rand5000.data(), 0, rand5000.size()-1);
110

```

```

111 // A) an unordered array of 5000 random integers
112 clog << "A) Sorting an unordered array of 5000 random integers:\n";
113 display_results(benchmark.run_with(args_5_000));
114
115 // B) a pre-sorted array of 5000
116 clog << "B) Sorting a pre-sorted array of 5000 random integers:\n";
117 std::sort(rand5000.begin(), rand5000.end());
118 display_results(benchmark.run_with(args_5_000));
119
120 // C) a pre-sorted array of 5000 in reverse order
121 clog << "C) Sorting a pre-sorted array of 5000 in reverse order:\n";
122 std::sort(rand5000.begin(), rand5000.end(), std::greater<>());
123 display_results(benchmark.run_with(args_5_000));
124
125
126 // BUILD an array of 50 random elements
127 vector<int> rand50(50);
128 // generate random numbers and store them in the array
129 for(size_t index=0; index<rand50.size(); index++)
130     rand50.at(index) = (rand()%10000)+1;
131 auto args_50 = make_tuple(rand50.data(), 0, rand50.size()-1);
132
133 // D) an unordered array of 50 random integers
134 clog << "D) Sorting an unordered array of 50 random integers:\n";
135 display_results(benchmark.run_with(args_50));
136
137 // E) a pre-sorted array of 50
138 clog << "E) Sorting a pre-sorted array of 50 random integers:\n";
139 std::sort(rand50.begin(), rand50.end());
140 display_results(benchmark.run_with(args_50));
141
142 // F) a pre-sorted array of 50 in reverse order
143 clog << "F) Sorting a pre-sorted array of 50 in reverse order:\n";
144 std::sort(rand50.begin(), rand50.end(), std::greater<>());
145 display_results(benchmark.run_with(args_50));
146
147 return 0;
148 }
149
150 void display_results(vector<DetailedBenchmarkEntry> results) {
151     auto cmp = [] (DetailedBenchmarkEntry& l, DetailedBenchmarkEntry& r) {
152         return l.time_in_seconds < r.time_in_seconds;
153     };
154     auto fastest = std::min_element(results.begin(), results.end(), cmp);
155     char fastc;
156     for (auto& entry : results) {
157         fastc = (entry.label == fastest->label) ? '*' : ' ';
158         clog << '\t' << fastc
159             << setw(14) << entry.label << " took " << fixed << setprecision(1)
160             << setw(8) << entry.time_in_seconds * 1000000 << " μs, "
161             << setw(8) << entry.return_value.moves << " moves, "
162             << setw(8) << entry.return_value.comparisons << " comparisons\n";
163     }
164 }

```

## sort.hpp

```
1  /* sort.hpp
2   * -----
3   * Authors: Darwin Jacob Groskleg
4   * Date:    Thursday, November 21, 2019
5   */
6  #ifndef SORT_HPP_INCLUDED
7  #define SORT_HPP_INCLUDED
8
9  // Sort functions must be able to return sorting algorithm details.
10 // - number of value comparisons
11 // - number of data movement (values reorder, ie swap)
12 struct SortingDetails {
13     int comparisons = 0;
14     int moves       = 0;
15     SortingDetails& operator+(const SortingDetails& rhs);
16     SortingDetails& operator+=(const SortingDetails& rhs);
17 };
18
19 //void InsertionSort(int [], int, int, SortingDetails = SortingDetails{});
20 SortingDetails InsertionSort(int A[], int left, int right);
21 SortingDetails SelectionSort(int A[], int left, int right);
22 SortingDetails QuickSort(int A[], int start, int end);
23 SortingDetails MergeSort(int A[], int left, int right);
24
25 // Helpers
26 void displayArray(int [], int size);
27 void displayArray(int array[], int start, int end);
28
29 void reverseSort(int [], int start, int end);
30
31 #endif // SORT_HPP_INCLUDED
```

## sort.cpp

```

1  /* sort.cpp
2   * -----
3   * Authors: Darwin Jacob Groskleg
4   * Date:    Thursday, November 21, 2019
5   */
6  #include "sort.hpp"
7  #include <iostream>
8
9  SortingDetails merge(int A[], int left, int middle, int right);
10
11 // MergeSort recursive function
12 // Move: consists of a single value relocation within the array A,
13 //       not copies into some dynamic array used to compute the correct move.
14 // Comparison: consists of a comparison between 2 values within the array A for
15 //             the purpose of deciding whether a move is required,
16 //             not comparison of array indices used for bounds checking.
17 //
18 // Stable sorting
19 //  $T(n) = O(n \log(n))$ 
20 //  $S(n) = O(n)$ 
21 SortingDetails MergeSort(int A[], int left, int right)
22 {
23     auto d = SortingDetails{};
24     // Want to immediately return if range is not 2 elements or more.
25     if (left >= right)
26         return d;
27
28     int middle = (left + right)/2;
29
30     auto dl = MergeSort(A, left, middle);
31     auto dr = MergeSort(A, middle+1, right);
32     auto dm = merge(A, left, middle, right);
33     d += dl + dr + dm;
34     return d;
35 }
36
37 // Merges two sorted subarrays into a larger sorted array
38 //  $T(n) = O(n)$ 
39 //  $S(n) = O(n)$ 
40 SortingDetails merge(int A[], int left, int middle, int right)
41 {
42     SortingDetails d{};
43     int i1 = left;
44     int i2 = middle + 1;
45
46     int k = 0; // sorted frontier index
47     int *sorted = new int [right - left + 1];
48
49     // Tracks moved if not into same relative position in the sorted array
50     // then store it in the sorted array.
51     auto compute_move = [&d, &sorted, &A, left] (int& dst, int& src) {
52         if (dst != src-left)
53             d.moves++;
54         sorted[dst++] = A[src++];
55     };

```



```

55     };
56
57     // Initializing "sorted" array
58     // out of values to copy when i1 reaches the middle or i2 reaches end.
59     while ((i1 <= middle) && (i2 <= right)) { // not comparison
60         d.comparisons++;
61         if(A[i1] < A[i2])
62             compute_move(k, i1);
63         else
64             compute_move(k, i2);
65     }
66
67     // load the rest of the remaining elements,
68     // if the range has uneven number of elements
69     while (i1 <= middle)
70         compute_move(k, i1);
71     while(i2 <= right)
72         compute_move(k, i2);
73
74     // Copy them back to the original array, overwriting it
75     for (int i = left; i <= right; i++)
76         A[i] = sorted[i-left];
77
78     delete [] sorted;
79     return d;
80 }
81
82 // Not Stable
83 //  $T(n) = O(n^2)$ 
84 // But on average  $T(n) = O(n \lg(n))$ 
85 //  $S(n) = O(1)$ 
86 SortingDetails QuickSort(int A[], int start, int end) {
87     SortingDetails d{};
88     // Early Exit: degenerate cases
89     if (start >= end)
90         return d;
91
92     int left = start, right = end;
93     int pivot = A[start];
94
95     // Partition
96     while (left < right) {
97         while (A[right] >= pivot && left < right)
98             d.comparisons++ && right--;
99         d.comparisons++;
100        if (left < right) {
101            A[left] = A[right];
102            d.moves++;
103            left++;
104        }
105
106        while (A[left] <= pivot && left < right)
107            d.comparisons++ && left++;
108        d.comparisons++;
109        if (left < right) {
110            A[right] = A[left];

```

```

111         d.moves++;
112         right--;
113     }
114 }
115 A[left] = pivot;
116 d.moves++;
117
118 // Recursion
119 if (start < left)
120     d += QuickSort(A, start, left-1);
121 if (left < end)
122     d += QuickSort(A, left+1, end);
123
124 return d;
125 }
126
127 // T(n) = O(n^2)
128 //     two nested loops
129 // S(n) = O(1)
130 //     no new allocations need to be made in relation to n
131 //     in place
132 // Is stable
133 // Is adaptive: efficient for mostly ordered sets -> O(n)
134 //
135 // The "playing-card" sorting algorithm
136 SortingDetails InsertionSort(int A[], int start, int end) {
137     SortingDetails d{};
138     int key, j;
139
140     // Use i,j to search through the unsorted portion of the array,
141     // elements to the right.
142     for (int i=start+1; i<=end; i++) {
143         key = A[i];
144         j = i - 1;
145         while (j >= start && A[j] > key) {
146             d.comparisons++;
147             A[j+1] = A[j];
148             d.moves++;
149             j--;
150         }
151         d.comparisons++;
152
153         A[j+1] = key;
154         d.moves++;
155     }
156     return d;
157 }
158
159 // T(n) = O(n^2)
160 //     two nested loops over n elements
161 // S(n) = O(1)
162 //     no new allocations need to be made in relation to n
163 SortingDetails SelectionSort(int A[], int start, int end) {
164     SortingDetails d{};
165     // Early Exit: degenerate cases
166     if (start >= end)

```

```

167         return d;
168
169     int min_index, j;
170     for (int i=start; i<=end; i++) {
171         min_index = i;
172         j = i + 1;
173         while (j <= end) {
174             if (A[j] < A[min_index]) {
175                 min_index = j;
176             }
177             d.comparisons++;
178             j++;
179         }
180         std::swap(A[min_index], A[i]);
181         // swaps count as 2 moves, does 1 more swap than is necessary
182         d.moves += 2;
183     }
184     return d;
185 }
186
187 // Helpers! =====
188
189 void displayArray(int A[], int size) {
190     displayArray(A, 0, size-1);
191 }
192
193 void displayArray(int A[], int start, int end) {
194     std::clog << "{";
195     for(int i=start; i<=end; i++)
196         std::clog << A[i] << " ";
197     std::clog << "}\n";
198 }
199
200 // T(n) = O(n log(n))
201 void reverseSort(int A[], int start, int end) {
202     MergeSort(A, start, end);
203     while (start < end) {
204         std::swap(A[start], A[end]);
205         start++;
206         end--;
207     }
208 }
209
210 SortingDetails& SortingDetails::operator+(const SortingDetails& rhs) {
211     this->comparisons += rhs.comparisons;
212     this->moves += rhs.moves;
213     return *this;
214 }
215
216 SortingDetails& SortingDetails::operator+=(const SortingDetails& rhs) {
217     this->comparisons += rhs.comparisons;
218     this->moves += rhs.moves;
219     return *this;
220 }

```

## benchmark.hpp

```

1  /* benchmark.hpp
2  * -----
3  * Authors: Darwin Jacob Groskleg
4  * Date:    Thursday, November 21, 2019
5  *
6  * Purpose: compare performance of multiple algorithms over multiple data sets.
7  */
8  #ifndef BENCHMARK_HPP_INCLUDED
9  #define BENCHMARK_HPP_INCLUDED
10
11 #include <functional>
12 #include <unordered_map>
13 #include <string>
14 #include <vector>
15 #include <tuple>
16
17 #include <ctime>
18
19 using CSorterParams = std::tuple<int*, int, int>;
20 using CSorter       = std::function<void (int[], int, int)>;
21 template<class Sig>
22 using return_t      = typename Sig::result_type;
23
24 // Details relating to an algorithm executed as part of a benchmark run.
25 template<typename R>
26 struct BenchmarkEntry {
27     std::string label;
28     double      time_in_seconds;
29     // R could be void, I don't know what would happen
30     R           return_value;
31 };
32
33
34 // CSortBenchmark
35 //
36 // Responsible for timing the execution of multiple sorting algorithms of the
37 // same type where there is an arbitrary number of possible arrays to serve as
38 // benchmarks.
39 // The benchmarking object specifically deals with c-style sorting functions
40 // that pass operate on an array given to them via pointer.
41 //
42 // What does benchmark need to know about?
43 // - the type signature
44 //   - return type of sort?    no use template
45 //   - argument list?         yes, not variable
46 //
47 template<class F>
48 class CSortBenchmark {
49     const std::unordered_map<std::string, F> function_list;
50
51 public:
52     // Pass a list of labels and functions
53     CSortBenchmark(std::unordered_map<std::string, F> flist) :
54         function_list{flist}

```

```

55     {}
56
57     // Returns labelled list of executions times in seconds.
58     auto run_with(CSorterParams args)
59         -> std::vector< BenchmarkEntry<return_t<F> > >;
60 };
61
62 // run_with
63 //
64 // CONSTRAINTS
65 // - template must be a std::function
66 // - Must be able to handle return values:
67 //     - void
68 //     - Sorting Details
69 // - ArgList must be copy-able!
70 // - if arglist contains an array.
71 template<class F>
72 auto CSortBenchmark<F>::
73 run_with(CSorterParams args)
74     -> std::vector< BenchmarkEntry<return_t<F>> >
75 {
76     int *A, head, tail;
77     std::tie(A, head, tail) = args;
78     std::vector<int> copy_of_A(tail+1);
79     std::vector< BenchmarkEntry<return_t<F>> > elapsed;
80     return_t<F> result;
81
82     double start;
83     double finish;
84     for (auto &f : function_list) {
85         // copy the array befor timing
86         // no need to erase, will always be the same size so just overwrite.
87         copy_of_A.assign(A, A+tail+1);
88
89         start = static_cast<double>(clock()) / CLOCKS_PER_SEC;
90         result = f.second(copy_of_A.data(), head, tail);
91         finish = static_cast<double>(clock()) / CLOCKS_PER_SEC;
92
93         elapsed.push_back({f.first, finish - start, result});
94     }
95     return elapsed;
96 }
97
98 #endif // BENCHMARK_HPP_INCLUDED

```

## sort\_test.cpp

```

1  /* sort_test.cpp
2   * -----
3   * Executable.
4   */
5  #include <algorithm>
6  #include <vector>
7
8  #define CATCH_CONFIG_MAIN
9  #include <catch2/catch.hpp>
10
11 #include "sort.hpp"
12
13 //using namespace std;
14 //using namespace Catch;
15 using Catch::Equals;
16
17 // MergeSort()=====
18 TEST_CASE("MergeSort with 1 element", "[merge]") {
19     std::vector<int> a{1};
20
21     SECTION("same value 1, same place") {
22         MergeSort(a.data(), 0, 0);
23         REQUIRE(1 == a[0]);
24     }
25
26     // implementation testing, very bad, acceptable on the degenerate case.
27     SECTION("0 value comparisons and 0 moves") {
28         auto d = MergeSort(a.data(), 0, 0);
29         REQUIRE(0 == d.moves);
30         REQUIRE(0 == d.comparisons);
31     }
32 }
33
34 TEST_CASE("MergeSort with 0 elements", "[merge]") {
35     std::vector<int> a{};
36
37     SECTION("no exceptions with start=0, end=0") {
38         REQUIRE_NOTHROW(MergeSort(a.data(), 0, 0));
39     }
40
41     // implementation testing, very bad, acceptable on the degenerate case.
42     SECTION("0 value comparisons and 0 moves") {
43         auto d = MergeSort(a.data(), 0, 0);
44         REQUIRE(0 == d.moves);
45         REQUIRE(0 == d.comparisons);
46     }
47 }
48
49 // smallest case without no operation
50 TEST_CASE("MergeSort with 2 elements", "[merge]") {
51     std::vector<int> a{2, 1};
52
53     SECTION("result is ordered") {
54         MergeSort(a.data(), 0, 1);

```

```

55     REQUIRE_THAT(a, Equals(std::vector<int>{1,2}));
56 }
57
58 // implementation testing, very bad, acceptable on the small cases.
59 SECTION("1 value comparison and 2 moves(1 swap)") {
60     auto d = MergeSort(a.data(), 0, 1);
61     REQUIRE(1 == d.comparisons);
62     REQUIRE(2 == d.moves);
63 }
64 }
65
66 TEST_CASE("MergeSort with 3 elements", "[merge]") {
67     std::vector<int> a{3, 2, 1};
68
69     SECTION("result is ordered") {
70         MergeSort(a.data(), 0, 2);
71         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
72     }
73
74     // implementation testing, very bad, acceptable on the small cases.
75     SECTION("3 value comparisons and 2 moves when sorted in reverse") {
76         auto d = MergeSort(a.data(), 0, 2);
77         REQUIRE(5 == d.moves);
78         REQUIRE(2 == d.comparisons);
79     }
80
81     // implementation testing, very bad, acceptable on the small cases.
82     SECTION("details on half sorted") {
83         a = {1, 3, 2};
84         auto d = MergeSort(a.data(), 0, 2);
85         REQUIRE(2 == d.moves);
86         REQUIRE(3 == d.comparisons);
87     }
88 }
89
90 TEST_CASE("MergeSort with 10 elements(even) is ordered", "[merge]") {
91     std::vector<int> a{10, 9, 8, 7, 6,
92                       5, 4, 3, 3, 1};
93
94     SECTION("sorting the left half") {
95         MergeSort(a.data(), 0, 4);
96         std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
97         REQUIRE_THAT(a, Equals(expected));
98     }
99
100    SECTION("sorting the right half") {
101        MergeSort(a.data(), 5, 9);
102        std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
103        REQUIRE_THAT(a, Equals(expected));
104    }
105
106    SECTION("sorting on all but the last element") {
107        MergeSort(a.data(), 0, 8);
108        std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
109        REQUIRE_THAT(a, Equals(expected));
110    }

```

```

111
112     SECTION("sorting on the whole array") {
113         MergeSort(a.data(), 0, 9);
114         std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
115         REQUIRE_THAT(a, Equals(expected));
116     }
117 }
118
119
120 // QuickSort()=====
121 TEST_CASE("QuickSort with 1 element", "[quick]") {
122     std::vector<int> a{1};
123
124     SECTION("same value 1, same place") {
125         QuickSort(a.data(), 0, 0);
126         REQUIRE(1 == a[0]);
127     }
128
129     // implementation testing, very bad, acceptable on the degenerate case.
130     SECTION("0 value comparisons and 0 moves") {
131         auto d = QuickSort(a.data(), 0, 0);
132         REQUIRE(0 == d.moves);
133         REQUIRE(0 == d.comparisons);
134     }
135 }
136
137 TEST_CASE("QuickSort with 0 elements", "[quick]") {
138     std::vector<int> a{};
139
140     SECTION("no exceptions with start=0, end=0") {
141         REQUIRE_NOTHROW(QuickSort(a.data(), 0, 0));
142     }
143
144     // implementation testing, very bad, acceptable on the degenerate case.
145     SECTION("0 value comparisons and 0 moves") {
146         auto d = QuickSort(a.data(), 0, 0);
147         REQUIRE(0 == d.moves);
148         REQUIRE(0 == d.comparisons);
149     }
150 }
151
152 // smallest case without no operation
153 TEST_CASE("QuickSort with 2 elements", "[quick]") {
154     std::vector<int> a{2, 1};
155
156     SECTION("result is ordered") {
157         QuickSort(a.data(), 0, 1);
158         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2}));
159     }
160
161     // implementation testing, very bad, acceptable on the small cases.
162     SECTION("value comparison and moves when pre-sorted in reverse") {
163         auto d = QuickSort(a.data(), 0, 1);
164         REQUIRE(2 == d.comparisons);
165         REQUIRE(2 == d.moves);
166     }

```



```

167 }
168
169 TEST_CASE("QuickSort with 3 elements", "[quick]") {
170     std::vector<int> a{3, 2, 1};
171
172     SECTION("result is ordered") {
173         QuickSort(a.data(), 0, 2);
174         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
175     }
176
177     // implementation testing, very bad, acceptable on the small cases.
178     SECTION("value comparisons and moves when sorted in reverse") {
179         auto d = QuickSort(a.data(), 0, 2);
180         REQUIRE(3 == d.moves);
181         REQUIRE(7 == d.comparisons);
182     }
183
184     // implementation testing, very bad, acceptable on the small cases.
185     SECTION("details on half sorted") {
186         a = {1, 3, 2};
187         auto d = QuickSort(a.data(), 0, 2);
188         REQUIRE(3 == d.moves);
189         REQUIRE(7 == d.comparisons);
190     }
191 }
192
193 TEST_CASE("QuickSort with 10 elements(even) is ordered", "[quick]") {
194     std::vector<int> a{10, 9, 8, 7, 6,
195                       5, 4, 3, 3, 1};
196
197     SECTION("sorting the left half") {
198         QuickSort(a.data(), 0, 4);
199         std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
200         REQUIRE_THAT(a, Equals(expected));
201     }
202
203     SECTION("sorting the right half") {
204         QuickSort(a.data(), 5, 9);
205         std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
206         REQUIRE_THAT(a, Equals(expected));
207     }
208
209     SECTION("sorting on all but the last element") {
210         QuickSort(a.data(), 0, 8);
211         std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
212         REQUIRE_THAT(a, Equals(expected));
213     }
214
215     SECTION("sorting on the whole array") {
216         QuickSort(a.data(), 0, 9);
217         std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
218         REQUIRE_THAT(a, Equals(expected));
219     }
220 }
221
222

```

```

223 // InsertionSort()=====
224 TEST_CASE("InsertionSort with 1 element", "[insertion]") {
225     std::vector<int> a{1};
226
227     SECTION("same value 1, same place") {
228         InsertionSort(a.data(), 0, 0);
229         REQUIRE(1 == a[0]);
230     }
231
232     // implementation testing, very bad, acceptable on the degenerate case.
233     SECTION("0 value comparisons and 0 moves") {
234         auto d = InsertionSort(a.data(), 0, 0);
235         REQUIRE(0 == d.moves);
236         REQUIRE(0 == d.comparisons);
237     }
238 }
239
240 TEST_CASE("InsertionSort with 0 elements", "[insertion]") {
241     std::vector<int> a{};
242
243     SECTION("no exceptions with start=0, end=0") {
244         REQUIRE_NOTHROW(InsertionSort(a.data(), 0, 0));
245     }
246
247     // implementation testing, very bad, acceptable on the degenerate case.
248     SECTION("0 value comparisons and 0 moves") {
249         auto d = InsertionSort(a.data(), 0, 0);
250         REQUIRE(0 == d.moves);
251         REQUIRE(0 == d.comparisons);
252     }
253 }
254
255 // smallest case without no operation
256 TEST_CASE("InsertionSort with 2 elements", "[insertion]") {
257     std::vector<int> a{2, 1};
258
259     SECTION("result is ordered") {
260         InsertionSort(a.data(), 0, 1);
261         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2}));
262     }
263
264     // implementation testing, very bad, acceptable on the small cases.
265     SECTION("value comparison and moves when pre-sorted in reverse") {
266         auto d = InsertionSort(a.data(), 0, 1);
267         REQUIRE(2 == d.comparisons);
268         REQUIRE(2 == d.moves);
269     }
270 }
271
272 TEST_CASE("InsertionSort with 3 elements", "[insertion]") {
273     std::vector<int> a{3, 2, 1};
274
275     SECTION("result is ordered") {
276         InsertionSort(a.data(), 0, 2);
277         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
278     }

```

```

279 // implementation testing, very bad, acceptable on the small cases.
280 SECTION("value comparisons and moves when sorted in reverse") {
281     auto d = InsertionSort(a.data(), 0, 2);
282     REQUIRE(5 == d.moves);
283     REQUIRE(5 == d.comparisons);
284 }
285
286 // implementation testing, very bad, acceptable on the small cases.
287 SECTION("details on half sorted") {
288     a = {1, 3, 2};
289     auto d = InsertionSort(a.data(), 0, 2);
290     REQUIRE(3 == d.moves);
291     REQUIRE(3 == d.comparisons);
292 }
293 }
294
295 TEST_CASE("InsertionSort with 10 elements(even) is ordered", "[insertion]") {
296     std::vector<int> a{10, 9, 8, 7, 6,
297                      5, 4, 3, 3, 1};
298
299     SECTION("sorting the left half") {
300         InsertionSort(a.data(), 0, 4);
301         std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
302         REQUIRE_THAT(a, Equals(expected));
303     }
304
305     SECTION("sorting the right half") {
306         InsertionSort(a.data(), 5, 9);
307         std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
308         REQUIRE_THAT(a, Equals(expected));
309     }
310
311     SECTION("sorting on all but the last element") {
312         InsertionSort(a.data(), 0, 8);
313         std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
314         REQUIRE_THAT(a, Equals(expected));
315     }
316
317     SECTION("sorting on the whole array") {
318         InsertionSort(a.data(), 0, 9);
319         std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
320         REQUIRE_THAT(a, Equals(expected));
321     }
322 }
323
324
325 // SelectionSort()=====
326 TEST_CASE("SelectionSort with 1 element", "[selection]") {
327     std::vector<int> a{1};
328
329     SECTION("same value 1, same place") {
330         SelectionSort(a.data(), 0, 0);
331         REQUIRE(1 == a[0]);
332     }
333 }
334

```

```

335 // implementation testing, very bad, acceptable on the degenerate case.
336 SECTION("0 value comparisons and 0 moves") {
337     auto d = SelectionSort(a.data(), 0, 0);
338     REQUIRE(0 == d.moves);
339     REQUIRE(0 == d.comparisons);
340 }
341 }
342
343 TEST_CASE("SelectionSort with 0 elements", "[selection]") {
344     std::vector<int> a{};
345
346     SECTION("no exceptions with start=0, end=0") {
347         REQUIRE_NOTHROW(SelectionSort(a.data(), 0, 0));
348     }
349
350 // implementation testing, very bad, acceptable on the degenerate case.
351 SECTION("0 value comparisons and 0 moves") {
352     auto d = SelectionSort(a.data(), 0, 0);
353     REQUIRE(0 == d.moves);
354     REQUIRE(0 == d.comparisons);
355 }
356 }
357
358 // smallest case without no operation
359 TEST_CASE("SelectionSort with 2 elements", "[selection]") {
360     std::vector<int> a{2, 1};
361
362     SECTION("result is ordered") {
363         SelectionSort(a.data(), 0, 1);
364         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2}));
365     }
366
367 // implementation testing, very bad, acceptable on the small cases.
368 SECTION("value comparison and moves when pre-sorted in reverse") {
369     auto d = SelectionSort(a.data(), 0, 1);
370     REQUIRE(1 == d.comparisons);
371     REQUIRE(4 == d.moves);
372 }
373 }
374
375 TEST_CASE("SelectionSort with 3 elements", "[selection]") {
376     std::vector<int> a{3, 2, 1};
377
378     SECTION("result is ordered") {
379         SelectionSort(a.data(), 0, 2);
380         REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
381     }
382
383 // implementation testing, very bad, acceptable on the small cases.
384 SECTION("value comparisons and moves when sorted in reverse") {
385     auto d = SelectionSort(a.data(), 0, 2);
386     REQUIRE(6 == d.moves);
387     REQUIRE(2 == d.comparisons);
388 }
389
390 // implementation testing, very bad, acceptable on the small cases.

```

```
391     SECTION("details on half sorted") {
392         a = {1, 3, 2};
393         auto d = SelectionSort(a.data(), 0, 2);
394         REQUIRE(6 == d.moves);
395         REQUIRE(1 == d.comparisons);
396     }
397 }
398
399 TEST_CASE("SelectionSort with 10 elements(even) is ordered", "[selection]") {
400     std::vector<int> a{10, 9, 8, 7, 6,
401                      5, 4, 3, 3, 1};
402
403     SECTION("sorting the left half") {
404         SelectionSort(a.data(), 0, 4);
405         std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
406         REQUIRE_THAT(a, Equals(expected));
407     }
408
409     SECTION("sorting the right half") {
410         SelectionSort(a.data(), 5, 9);
411         std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
412         REQUIRE_THAT(a, Equals(expected));
413     }
414
415     SECTION("sorting on all but the last element") {
416         SelectionSort(a.data(), 0, 8);
417         std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
418         REQUIRE_THAT(a, Equals(expected));
419     }
420
421     SECTION("sorting on the whole array") {
422         SelectionSort(a.data(), 0, 9);
423         std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
424         REQUIRE_THAT(a, Equals(expected));
425     }
426 }
```