CSCI 255: Lab #10 Sorting Performance

Darwin Jacob Groskleg

Thursday, November 21st, 2019

Contents

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 spacde 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

```
/* main.cpp
 * --------
 * Authors: Darwin Jacob Groskleg
 * Date:    Thursday, November 21, 2019
 *
 * QUESTIONS:
 * 1. 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.
 *
 * 2. 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.
 *
 * 3. Is the timings with n=50 as expected?
 *      The timings with n=50 were not as expected if making predictions based
 *      on time and spacde 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.
 *
 * CONSOLE SAMPLE
================================================================================
$> gmake
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main.o -c m
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o sort.o -c s
clang++ -std=c++14 -stdlib=libc++ -Wpedantic -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0 -o main.out ma

$> time ./main.out
time ./main.out
A) Sorting an unordered array of 5000 random integers:
      InsertionSort took  19479.0 μs,  6222810 moves,  6222810 comparisons
      SelectionSort took  26618.0 μs,    10000 moves, 12497500 comparisons
    *     QuickSort took    487.0 μs,    26681 moves,    79745 comparisons
          MergeSort took   1602.0 μs,    55243 moves,    55256 comparisons
B) Sorting a pre-sorted array of 5000 random integers:
    * InsertionSort took     23.0 μs,     4999 moves,     4999 comparisons
      SelectionSort took  24959.0 μs,    10000 moves, 12497500 comparisons
          QuickSort took  31422.0 μs,     4999 moves, 12512497 comparisons
          MergeSort took   1222.0 μs,     2339 moves,    32430 comparisons
```

```
C) Sorting a pre-sorted array of 5000 in reverse order:
      InsertionSort took  39624.0 µs, 12501232 moves, 12501232 comparisons
      SelectionSort took  28290.0 µs,    10000 moves, 12497500 comparisons
          QuickSort took  31107.0 µs,     7497 moves, 12507501 comparisons
    *     MergeSort took   1289.0 µs,    61808 moves,    29804 comparisons
D) Sorting an unordered array of 50 random integers:
    * InsertionSort took      4.0 µs,      553 moves,      553 comparisons
      SelectionSort took      7.0 µs,      100 moves,     1225 comparisons
          QuickSort took      4.0 µs,      121 moves,      322 comparisons
          MergeSort took     13.0 µs,      216 moves,      220 comparisons
E) Sorting a pre-sorted array of 50 random integers:
    * InsertionSort took      1.0 µs,       49 moves,       49 comparisons
      SelectionSort took      3.0 µs,      100 moves,     1225 comparisons
          QuickSort took      6.0 µs,       49 moves,     1372 comparisons
          MergeSort took     10.0 µs,        0 moves,      153 comparisons
F) Sorting a pre-sorted array of 50 in reverse order:
      InsertionSort took      5.0 µs,     1274 moves,     1274 comparisons
    * SelectionSort took      4.0 µs,      100 moves,     1225 comparisons
          QuickSort took      6.0 µs,       74 moves,     1322 comparisons
          MergeSort took     11.0 µs,      286 moves,      133 comparisons
===============================================================================
 */
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

#include <time.h>
#include <stdlib.h>

#include "sort.hpp"
#include "benchmark.hpp"

using namespace std;
using DetailedCSorter        = std::function<SortingDetails (int[], int, int)>;
using DetailedBenchmarkEntry = BenchmarkEntry<SortingDetails>;

void display_results(vector<DetailedBenchmarkEntry>);

int main() {
    CSortBenchmark< DetailedCSorter > benchmark({
        {"InsertionSort", InsertionSort},
        {"SelectionSort", SelectionSort},
        {"QuickSort",     QuickSort},
        {"MergeSort",     MergeSort}
    });

    srand((unsigned)time(NULL));

    // BUILD 5000 random element array
    vector<int> rand5000(5'000);
    // generate random numbers and store them in the array
    for(size_t index=0; index<rand5000.size(); index++)
        rand5000.at(index) = (rand()%10'000)+1;
    auto args_5_000 = make_tuple(rand5000.data(), 0, rand5000.size()-1);

```

```cpp
    // A) an unordered array of 5000 random integers
    clog << "A) Sorting an unordered array of 5000 random integers:\n";
    display_results(benchmark.run_with(args_5_000));

    // B) a pre-sorted array of 5000
    clog << "B) Sorting a pre-sorted array of 5000 random integers:\n";
    std::sort(rand5000.begin(), rand5000.end());
    display_results(benchmark.run_with(args_5_000));

    // C) a pre-sorted array of 5000 in reverse order
    clog << "C) Sorting a pre-sorted array of 5000 in reverse order:\n";
    std::sort(rand5000.begin(), rand5000.end(), std::greater<>());
    display_results(benchmark.run_with(args_5_000));


    // BUILD an array of 50 random elements
    vector<int> rand50(50);
    // generate random numbers and store them in the array
    for(size_t index=0; index<rand50.size(); index++)
        rand50.at(index) = (rand()%10'000)+1;
    auto args_50 = make_tuple(rand50.data(), 0, rand50.size()-1);

    // D) an unordered array of 50 random integers
    clog << "D) Sorting an unordered array of 50 random integers:\n";
    display_results(benchmark.run_with(args_50));

    // E) a pre-sorted array of 50
    clog << "E) Sorting a pre-sorted array of 50 random integers:\n";
    std::sort(rand50.begin(), rand50.end());
    display_results(benchmark.run_with(args_50));

    // F) a pre-sorted array of 50 in reverse order
    clog << "F) Sorting a pre-sorted array of 50 in reverse order:\n";
    std::sort(rand50.begin(), rand50.end(), std::greater<>());
    display_results(benchmark.run_with(args_50));

    return 0;
}

void display_results(vector<DetailedBenchmarkEntry> results) {
    auto cmp = [] (DetailedBenchmarkEntry& l, DetailedBenchmarkEntry& r) {
        return l.time_in_seconds < r.time_in_seconds;
    };
    auto fastest = std::min_element(results.begin(), results.end(), cmp);
    char fastc;
    for (auto& entry : results) {
        fastc = (entry.label == fastest->label) ? '*' : ' ';
        clog << '\t' << fastc
            << setw(14) << entry.label << " took " << fixed << setprecision(1)
            << setw(8)  << entry.time_in_seconds * 1'000'000 << " µs, "
            << setw(8)  << entry.return_value.moves << " moves, "
            << setw(8)  << entry.return_value.comparisons << " comparisons\n";
    }
}
```

sort.hpp

```cpp
/* sort.hpp
 * ---------
 * Authors: Darwin Jacob Groskleg
 * Date:    Thursday, November 21, 2019
 */
#ifndef SORT_HPP_INCLUDED
#define SORT_HPP_INCLUDED

// Sort functions must be able to return sorting algorithm details.
//  - number of value comparisons
//  - number of data movement (values reorder, ie swap)
struct SortingDetails {
    int comparisons = 0;
    int moves       = 0;
    SortingDetails& operator+(const SortingDetails& rhs);
    SortingDetails& operator+=(const SortingDetails& rhs);
};

//void InsertionSort(int [], int, int, SortingDetails = SortingDetails{});
SortingDetails InsertionSort(int A[], int left, int right);
SortingDetails SelectionSort(int A[], int left, int right);
SortingDetails QuickSort(int A[], int start, int end);
SortingDetails MergeSort(int A[], int left, int right);

// Helpers
void displayArray(int [], int size);
void displayArray(int array[], int start, int end);

void reverseSort(int [], int start, int end);

#endif // SORT_HPP_INCLUDED
```

sort.cpp

```cpp
/* sort.cpp
 * --------
 * Authors: Darwin Jacob Groskleg
 * Date:    Thursday, November 21, 2019
 */
#include "sort.hpp"
#include <iostream>

SortingDetails merge(int A[], int left, int middle, int right);

// MergeSort recursive function
//  Move: consists of a single value relocation within the array A,
//        not copies into some dynamic array used to compute the correct move.
//  Comparison: consists of a comparison between 2 values within the array A for
//              the purpose of deciding whether a move is required,
//              not comparison of array indices used for bounds checking.
//
// Stable sorting
// T(n) = O( n log(n) )
// S(n) = O(n)
SortingDetails MergeSort(int A[], int left, int right)
{
    auto d = SortingDetails{};
    // Want to immediately return if range is not 2 elements or more.
    if (left >= right)
        return d;

    int middle = (left + right)/2;

    auto dl = MergeSort(A, left, middle);
    auto dr = MergeSort(A, middle+1, right);
    auto dm = merge(A, left, middle, right);
    d += dl + dr + dm;
    return d;
}

// Merges two sorted subarrays into a larger sorted array
// T(n) = O(n)
// S(n) = O(n)
SortingDetails merge(int A[], int left, int middle, int right)
{
    SortingDetails d{};
    int i1 = left;
    int i2 = middle + 1;

    int k  = 0; // sorted frontier index
    int *sorted = new int [right - left + 1];

    // Tracks moved if not into same relative position in the sorted array
    // then store it in the sorted array.
    auto compute_move = [&d, &sorted, &A, left] (int& dst, int& src) {
        if (dst != src-left)
            d.moves++;
        sorted[dst++] = A[src++];
```

```
    };

    // Initializing "sorted" array
    //  out of values to copy when i1 reaches the middle or i2 reaches end.
    while ((i1 <= middle) && (i2 <= right)) { // not comparison
        d.comparisons++;
        if(A[i1] < A[i2])
            compute_move(k, i1);
        else
            compute_move(k, i2);
    }

    // load the rest of the remaining elements,
    //      if the range has uneven number of elements
    while (i1 <= middle)
        compute_move(k, i1);
    while(i2 <= right)
        compute_move(k, i2);

    // Copy them back to the original array, overwriting it
    for (int i = left; i <= right; i++)
        A[i] = sorted[i-left];

    delete [] sorted;
    return d;
}

// Not Stable
// T(n) = O(n^2)
//      But on average T(n) = O(n lg(n))
// S(n) = O(1)
SortingDetails QuickSort(int A[], int start, int end) {
    SortingDetails d{};
    // Early Exit: degenate cases
    if (start >= end)
        return d;

    int left = start, right = end;
    int pivot = A[start];

    // Partition
    while (left < right) {
        while (A[right] >= pivot && left < right)
            d.comparisons++ && right--;
        d.comparisons++;
        if (left < right) {
            A[left] = A[right];
            d.moves++;
            left ++;
        }

        while (A[left] <= pivot && left < right)
            d.comparisons++ && left++;
        d.comparisons++;
        if (left < right) {
            A[right] = A[left];
```

```
            d.moves++;
            right --;
        }
    }
    A[left] = pivot;
    d.moves++;

    // Recursion
    if (start < left)
        d += QuickSort(A, start, left-1);
    if (left < end)
        d += QuickSort(A, left+1, end);

    return d;
}

// T(n) = O(n^2)
//       two nested loops
// S(n) = O(1)
//       no new allocations need to be made in relation to n
//       in place
// Is stable
// Is adaptive: efficient for mostly ordered sets -> O(n)
//
// The "playing-card" sorting algorithm
SortingDetails InsertionSort(int A[], int start, int end) {
    SortingDetails d{};
    int key, j;

    // Use i,j to search throught the unsorted portion of the array,
    //  elements to the right.
    for (int i=start+1; i<=end; i++) {
        key = A[i];
        j = i - 1;
        while (j >= start && A[j] > key) {
            d.comparisons++;
            A[j+1] = A[j];
            d.moves++;
            j--;
        }
        d.comparisons++;

        A[j+1] = key;
        d.moves++;
    }
    return d;
}

// T(n) = O(n^2)
//       two nested loops over n elements
// S(n) = O(1)
//       no new allocations need to be made in relation to n
SortingDetails SelectionSort(int A[], int start, int end) {
    SortingDetails d{};
    // Early Exit: degenate cases
    if (start >= end)
```

```
        return d;

    int min_index, j;
    for (int i=start; i<=end; i++) {
        min_index = i;
        j = i + 1;
        while (j <= end) {
            if (A[j] < A[min_index]) {
                min_index = j;
            }
            d.comparisons++;
            j++;
        }
        std::swap(A[min_index], A[i]);
        // swaps count as 2 moves, does 1 more swap than is necessary
        d.moves += 2;
    }
    return d;
}

// Helpers! ====================================================================

void displayArray(int A[], int size) {
    displayArray(A, 0, size-1);
}

void displayArray(int A[], int start, int end) {
    std::clog << "{";
    for(int i=start; i<=end; i++)
        std::clog << A[i] << " ";
    std::clog << "}\n";
}

// T(n) = O(n log(n))
void reverseSort(int A[], int start, int end) {
    MergeSort(A, start, end);
    while (start < end) {
        std::swap(A[start], A[end]);
        start++;
        end--;
    }
}

SortingDetails& SortingDetails::operator+(const SortingDetails& rhs) {
    this->comparisons += rhs.comparisons;
    this->moves       += rhs.moves;
    return *this;
}

SortingDetails& SortingDetails::operator+=(const SortingDetails& rhs) {
    this->comparisons += rhs.comparisons;
    this->moves       += rhs.moves;
    return *this;
}
```

benchmark.hpp

```cpp
/* benchmark.hpp
 * -------------
 * Authors: Darwin Jacob Groskleg
 * Date:    Thursday, November 21, 2019
 *
 * Purpose: compare performance of multiple algorithms over multiple data sets.
 */
#ifndef BENCHMARK_HPP_INCLUDED
#define BENCHMARK_HPP_INCLUDED

#include <functional>
#include <unordered_map>
#include <string>
#include <vector>
#include <tuple>

#include <ctime>

using CSorterParams = std::tuple<int*, int, int>;
using CSorter       = std::function<void (int[], int, int)>;
template<class Sig>
using return_t      = typename Sig::result_type;

// Details relating to an algorithm executed as part of a benchmark run.
template<typename R>
struct BenchmarkEntry {
    std::string label;
    double      time_in_seconds;
    // R could be void, I don't know what would happen
    R           return_value;
};


// CSortBenchmark
//
// Responsible for timing the execution of multiple sorting algorithms of the
// same type where there is an arbitrary number of possible arrays to serve as
// benchmarks.
// The benchmarking object specifically deals with c-style sorting functions
// that pass operate on an array given to them via pointer.
//
// What does benchmark need to know about?
// -  the type signature
//    - return type of sort?    no use template
//    - argument list?          yes, not variable
//
template<class F>
class CSortBenchmark {
    const std::unordered_map<std::string, F> function_list;

  public:
    // Pass a list of labels and functions
    CSortBenchmark(std::unordered_map<std::string, F> flist) :
        function_list{flist}
```

```
    {}

    // Returns labelled list of executions times in seconds.
    auto run_with(CSorterParams args)
        -> std::vector< BenchmarkEntry<return_t<F> > >;
};

// run_with
//
//  CONSTRAINTS
//  - template must be a std::function
//  - Must be able to handle return values:
//      - void
//      - Sorting Details
//  - ArgList must be copy-able!
//      - if arglist contains an array.
  template<class F>
  auto CSortBenchmark<F>::
run_with(CSorterParams args)
    -> std::vector< BenchmarkEntry<return_t<F>> >
{
    int *A, head, tail;
    std::tie(A, head, tail) = args;
    std::vector<int> copy_of_A(tail+1);
    std::vector< BenchmarkEntry<return_t<F>> > elapsed;
    return_t<F> result;

    double start;
    double finish;
    for (auto &f : function_list) {
        // copy the array befor timing
        //  no need to erase, will always be the same size so just overwrite.
        copy_of_A.assign(A, A+tail+1);

        start = static_cast<double>(clock()) / CLOCKS_PER_SEC;
        result = f.second(copy_of_A.data(), head, tail);
        finish = static_cast<double>(clock()) / CLOCKS_PER_SEC;

        elapsed.push_back({f.first, finish - start, result});
    }
    return elapsed;
}

#endif // BENCHMARK_HPP_INCLUDED
```

sort_test.cpp

```cpp
/* sort_test.cpp
 * -------------
 *  Executable.
 */
#include <algorithm>
#include <vector>

#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>

#include "sort.hpp"

//using namespace std;
//using namespace Catch;
using Catch::Equals;

// MergeSort()=================================================================
TEST_CASE("MergeSort with 1 element", "[merge]") {
    std::vector<int> a{1};

    SECTION("same value 1, same place") {
        MergeSort(a.data(), 0, 0);
        REQUIRE(1 == a[0]);
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = MergeSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

TEST_CASE("MergeSort with 0 elements", "[merge]") {
    std::vector<int> a{};

    SECTION("no exceptions with start=0, end=0") {
        REQUIRE_NOTHROW(MergeSort(a.data(), 0, 0));
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = MergeSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

// smallest case without no operation
TEST_CASE("MergeSort with 2 elements", "[merge]") {
    std::vector<int> a{2, 1};

    SECTION("result is ordered") {
        MergeSort(a.data(), 0, 1);
```

```
        REQUIRE_THAT(a, Equals(std::vector<int>{1,2}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("1 value comparison and 2 moves(1 swap)") {
        auto d = MergeSort(a.data(), 0, 1);
        REQUIRE(1 == d.comparisons);
        REQUIRE(2 == d.moves);
    }
}

TEST_CASE("MergeSort with 3 elements", "[merge]") {
    std::vector<int> a{3, 2, 1};

    SECTION("result is ordered") {
        MergeSort(a.data(), 0, 2);
        REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("3 value comparisons and 2 moves when sorted in reverse") {
        auto d = MergeSort(a.data(), 0, 2);
        REQUIRE(5 == d.moves);
        REQUIRE(2 == d.comparisons);
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("details on half sorted") {
        a = {1, 3, 2};
        auto d = MergeSort(a.data(), 0, 2);
        REQUIRE(2 == d.moves);
        REQUIRE(3 == d.comparisons);
    }
}

TEST_CASE("MergeSort with 10 elements(even) is ordered", "[merge]") {
    std::vector<int> a{10, 9, 8, 7, 6,
                        5, 4, 3, 3, 1};

    SECTION("sorting the left half") {
        MergeSort(a.data(), 0, 4);
        std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting the right half") {
        MergeSort(a.data(), 5, 9);
        std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on all but the last element") {
        MergeSort(a.data(), 0, 8);
        std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
        REQUIRE_THAT(a, Equals(expected));
    }
```

```cpp

    SECTION("sorting on the whole array") {
        MergeSort(a.data(), 0, 9);
        std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
        REQUIRE_THAT(a, Equals(expected));
    }
}


// QuickSort()=================================================================
TEST_CASE("QuickSort with 1 element", "[quick]") {
    std::vector<int> a{1};

    SECTION("same value 1, same place") {
        QuickSort(a.data(), 0, 0);
        REQUIRE(1 == a[0]);
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = QuickSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

TEST_CASE("QuickSort with 0 elements", "[quick]") {
    std::vector<int> a{};

    SECTION("no exceptions with start=0, end=0") {
        REQUIRE_NOTHROW(QuickSort(a.data(), 0, 0));
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = QuickSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

// smallest case without no operation
TEST_CASE("QuickSort with 2 elements", "[quick]") {
    std::vector<int> a{2, 1};

    SECTION("result is ordered") {
        QuickSort(a.data(), 0, 1);
        REQUIRE_THAT(a, Equals(std::vector<int>{1,2}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparison and moves when pre-sorted in reverse") {
        auto d = QuickSort(a.data(), 0, 1);
        REQUIRE(2 == d.comparisons);
        REQUIRE(2 == d.moves);
    }
```

```
}

TEST_CASE("QuickSort with 3 elements", "[quick]") {
    std::vector<int> a{3, 2, 1};

    SECTION("result is ordered") {
        QuickSort(a.data(), 0, 2);
        REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparisons and moves when sorted in reverse") {
        auto d = QuickSort(a.data(), 0, 2);
        REQUIRE(3 == d.moves);
        REQUIRE(7 == d.comparisons);
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("details on half sorted") {
        a = {1, 3, 2};
        auto d = QuickSort(a.data(), 0, 2);
        REQUIRE(3 == d.moves);
        REQUIRE(7 == d.comparisons);
    }
}

TEST_CASE("QuickSort with 10 elements(even) is ordered", "[quick]") {
    std::vector<int> a{10, 9, 8, 7, 6,
                       5, 4, 3, 3, 1};

    SECTION("sorting the left half") {
        QuickSort(a.data(), 0, 4);
        std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting the right half") {
        QuickSort(a.data(), 5, 9);
        std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on all but the last element") {
        QuickSort(a.data(), 0, 8);
        std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on the whole array") {
        QuickSort(a.data(), 0, 9);
        std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
        REQUIRE_THAT(a, Equals(expected));
    }
}
```

```
// InsertionSort()==============================================================
TEST_CASE("InsertionSort with 1 element", "[insertion]") {
    std::vector<int> a{1};

    SECTION("same value 1, same place") {
        InsertionSort(a.data(), 0, 0);
        REQUIRE(1 == a[0]);
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = InsertionSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

TEST_CASE("InsertionSort with 0 elements", "[insertion]") {
    std::vector<int> a{};

    SECTION("no exceptions with start=0, end=0") {
        REQUIRE_NOTHROW(InsertionSort(a.data(), 0, 0));
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = InsertionSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

// smallest case without no operation
TEST_CASE("InsertionSort with 2 elements", "[insertion]") {
    std::vector<int> a{2, 1};

    SECTION("result is ordered") {
        InsertionSort(a.data(), 0, 1);
        REQUIRE_THAT(a, Equals(std::vector<int>{1,2}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparison and moves when pre-sorted in reverse") {
        auto d = InsertionSort(a.data(), 0, 1);
        REQUIRE(2 == d.comparisons);
        REQUIRE(2 == d.moves);
    }
}

TEST_CASE("InsertionSort with 3 elements", "[insertion]") {
    std::vector<int> a{3, 2, 1};

    SECTION("result is ordered") {
        InsertionSort(a.data(), 0, 2);
        REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
    }
```

```

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparisons and moves when sorted in reverse") {
        auto d = InsertionSort(a.data(), 0, 2);
        REQUIRE(5 == d.moves);
        REQUIRE(5 == d.comparisons);
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("details on half sorted") {
        a = {1, 3, 2};
        auto d = InsertionSort(a.data(), 0, 2);
        REQUIRE(3 == d.moves);
        REQUIRE(3 == d.comparisons);
    }
}

TEST_CASE("InsertionSort with 10 elements(even) is ordered", "[insertion]") {
    std::vector<int> a{10, 9, 8, 7, 6,
                        5, 4, 3, 3, 1};

    SECTION("sorting the left half") {
        InsertionSort(a.data(), 0, 4);
        std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting the right half") {
        InsertionSort(a.data(), 5, 9);
        std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on all but the last element") {
        InsertionSort(a.data(), 0, 8);
        std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on the whole array") {
        InsertionSort(a.data(), 0, 9);
        std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
        REQUIRE_THAT(a, Equals(expected));
    }
}


// SelectionSort()=============================================================
TEST_CASE("SelectionSort with 1 element", "[selection]") {
    std::vector<int> a{1};

    SECTION("same value 1, same place") {
        SelectionSort(a.data(), 0, 0);
        REQUIRE(1 == a[0]);
    }

```

```
    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = SelectionSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

TEST_CASE("SelectionSort with 0 elements", "[selection]") {
    std::vector<int> a{};

    SECTION("no exceptions with start=0, end=0") {
        REQUIRE_NOTHROW(SelectionSort(a.data(), 0, 0));
    }

    // implementation testing, very bad, acceptable on the degenerate case.
    SECTION("0 value comparisons and 0 moves") {
        auto d = SelectionSort(a.data(), 0, 0);
        REQUIRE(0 == d.moves);
        REQUIRE(0 == d.comparisons);
    }
}

// smallest case without no operation
TEST_CASE("SelectionSort with 2 elements", "[selection]") {
    std::vector<int> a{2, 1};

    SECTION("result is ordered") {
        SelectionSort(a.data(), 0, 1);
        REQUIRE_THAT(a, Equals(std::vector<int>{1,2}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparison and moves when pre-sorted in reverse") {
        auto d = SelectionSort(a.data(), 0, 1);
        REQUIRE(1 == d.comparisons);
        REQUIRE(4 == d.moves);
    }
}

TEST_CASE("SelectionSort with 3 elements", "[selection]") {
    std::vector<int> a{3, 2, 1};

    SECTION("result is ordered") {
        SelectionSort(a.data(), 0, 2);
        REQUIRE_THAT(a, Equals(std::vector<int>{1, 2, 3}));
    }

    // implementation testing, very bad, acceptable on the small cases.
    SECTION("value comparisons and moves when sorted in reverse") {
        auto d = SelectionSort(a.data(), 0, 2);
        REQUIRE(6 == d.moves);
        REQUIRE(2 == d.comparisons);
    }

    // implementation testing, very bad, acceptable on the small cases.
```

```
    SECTION("details on half sorted") {
        a = {1, 3, 2};
        auto d = SelectionSort(a.data(), 0, 2);
        REQUIRE(6 == d.moves);
        REQUIRE(1 == d.comparisons);
    }
}

TEST_CASE("SelectionSort with 10 elements(even) is ordered", "[selection]") {
    std::vector<int> a{10, 9, 8, 7, 6,
                        5, 4, 3, 3, 1};

    SECTION("sorting the left half") {
        SelectionSort(a.data(), 0, 4);
        std::vector<int> expected{6, 7, 8, 9, 10, 5, 4, 3, 3, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting the right half") {
        SelectionSort(a.data(), 5, 9);
        std::vector<int> expected{10, 9, 8, 7, 6, 1, 3, 3, 4, 5};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on all but the last element") {
        SelectionSort(a.data(), 0, 8);
        std::vector<int> expected{3, 3, 4, 5, 6, 7, 8, 9, 10, 1};
        REQUIRE_THAT(a, Equals(expected));
    }

    SECTION("sorting on the whole array") {
        SelectionSort(a.data(), 0, 9);
        std::vector<int> expected{1, 3, 3, 4, 5, 6, 7, 8, 9, 10};
        REQUIRE_THAT(a, Equals(expected));
    }
}
```