CS 2024 FINAL PROJECT USER MANUAL

KIRILL CHERNYSHOV

1. Overview

This is a project that performs time-benchmarking of several sorting algorithms. The algorithms implemented are:

Algorithm	Average case	Best case	Worst case
Bubblesort	$O(n^2)$	$O(n)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

The project provides an interface to call the different algorithms on generated data and measure the execution time. The data can be generated with a specified generation rule, or it can be custom data provided by the user.

2. Interface

To use this functionality, first include SortBenchmark.h. The algorithms are exposed as static methods, for example `SortBenchmark::bubblesort`. Each of these methods has the following signature:

```
void SortBenchmark::bubblesort(int* array, int length);
```

The two arguments are the array of `int`s to sort and its length. The array is sorted "in-place": even if the algorithm is not in-place, the array itself is modified, and nothing is returned.

To benchmark individual runs, use `SortBenchmark::runAlgoBenchmark`, which has the following signature:

```
static long long int runAlgoBenchmark(
    void (* algo)(int*, int),
    int* array,
    int length
);
```

Once again, an array and its length must be provided. The first parameter is for the sorting algorithm, as a pointer. The method returns the time taken in *microseconds* ($1\mu s = 10^{-6}s$). For example, this benchmarks bubblesort:

```
long time = runAlgoBenchmark(
    SortBenchmark::bubblesort,
    int* array,
    int length
);
```

2.1. **Mass benchmarking.** The core of the class is mass-benchmarking of multiple runs of an algorithm. The class allows to run a specified amount of iterations of a certain algorithm, on arrays with a specified length. This functionality is provided by `SortBenchmark::runAlgoBenchmark`. The array data is generated randomly, but this can be changed.

2.1.1. *Iterations and array length.* Aside from the algorithm, two more things must be specified: the number of iterations and the array length. The former is how many times to run the algorithm; the total execution time will be benchmarked, which is the sum of times from all the runs. The latter is how big the arrays given to the argument should be. A recommended value that seems to work well on most systems is 100 iterations of 10000-long arrays.

2.1.2. *Data generation rules.* There are four array generation rules. The first, `SortBenchmark::random_values` is fairly self-explanatory: the data is generated completely randomly with (hopefully) lack of any kind of order. `SortBenchmark::almost_sorted` generates data that is not completely in order, but almost. This is done by generating every element randomly from a small range around the value the element would normally be if the array were sorted. For example, if in a sorted array a value would be 1000, then in an amost-sorted array it would be a random value from 990 to 1010. `SortBenchmark::reverse` simply generates the values in reverse order. This catches out quicksort and bubblesort, which are exceptionally bad at dealing with reverse data. Finally, `SortBenchmark::few_unique` generates an array randomly, but from a small range of values, for example an array of 10000 with numbers ranging from 0 to 99. Note that specifying the generation rule is optional, and will default to `SortBenchmark::random_values`.

2.1.3. *Method signature.* In general, the method signature is as follows:

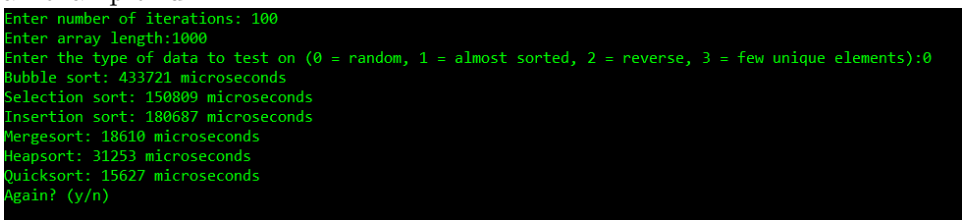
```
static long runAlgoBenchmark(
    void (* algo)(int*, int),
    int iterations,
    int arrayLength,
    DataGenRule rule = random_values
);
```

Once again, the return value is the execution time in *microseconds* ($1\mu s = 10^{-6}s$).

2.1.4. *Example.* To run bubblesort on 100 iterations of 10000-long arrays, filled with reverse-sorted data (warning: this will be slow!), use

```
long time = SortBenchmark::runAlgoBenchmark(
    SortBenchmark::bubblesort,
    100,
    10000,
    SortBenchmark::reverse
);
```

2.2. **Demo.** There is a small, interactive demo provided in main.cpp. Here is an example run:



The demo is also a good use example of some of the functions.