CSCI 152, Performance and Data Structures, Exercise 3

Rules for Exercises:

- You are expected to read the complete exercise, and to make every part of it. 'I did not see this part', will never be a valid excuse for not having made some part of the exercise.
- Submitted code will be checked for correctness, layout and style. We apply high standards, that make you fit for the real world.
- You are allowed to cooperate with other students during lab sessions and to ask for advice for lab assistants. It is, however, not allowed to share code, or to hand in identical solutions. You must write your code by yourself. See the syllabus for more rules. You may be invited for life grading, for discussion of the quality of your code, or its authorship.

Goal of this exercise is to make you familiar with empirical performance evaluation. You will compare different sorting algorithms, using inputs of different sizes.

There are a couple of practical things that are important to know: It is difficult to measure short run times reliably. ($< 10^{-4}$) For short runtimes, it is better to run the same function 1000 times, in order to get enough precision. Even for bigger run times, the measured time can vary quite a lot. (in the range of 10 percent) It is useful to run the same test a couple of times.

Measurement is done by subtracting the (wallclock) start time from the begin time. This is unreliable if there are too many other processes on your computer running, because then your test does not get full CPU time. In particular, make sure that no browsers are open, don't watch videos, don't play music, at the same time.

Download the files **timer.h**, **timetable.h**, **sorttests.cpp**. If you use Linux[1], also download **Makefile**. If you are not using Linux, you must turn on compiler optimization. Eclipse allows to set flags to the compiler. The flags must be `-O3 -flto`.

Class **timer** implements a kind of stopwatch. A timer cannot be stopped, which means that it is always counting. It has the following methods:

- `timer( )`. Creates a timer.

[1] It is not too late to start yet

- `clear( )`. Resets the timer.
- `double time( ) const`. Returns the current counted time in seconds. It is the time since the last `clear( )` or construction.

Here is an example of its use:

```
timer t;
(run algorithm that you want to measure).
double d = t. time( );
   // Time that was taken by algorithm.
t. clear( );
(run another algorithm you want to measure).
d = t. time( );
   // Time that was taken by 'another algorithm'.
```

As was (or will be) explained in class, usually only the *asymptotic behavior* of an algorithm matters. In order to understand the asymptotic behavior, one must run it inputs of different size, and observe how the runtime changes when the input changes. In order to make this easier, we give you a class **timetable** that stores a set of input sizes with their associated run times. It has the following methods:

- `timetable( const std::string& algo )`, `timetable( const char* algo )`. Construct a `timetable` for algorithm with given name.
- `void insert( size_t inputsize, double runtime )`. Store an `inputsize` with a `runtime`.
- `clear( )`. Forget all stored data.

In addition to the given methods, a `timetable` can be printed using `operator <<` . It is **obligatory** to output all your measurements by printing a `timetable`. The reason for this is that we automatically scan your output, so we need to know that all students print their measurements in the same format. Here is an example of the use of class `timetable`.

```
timetable alg1( "slowsort" );
timetable alg2( "bettersort" );

// Doubling s is better than adding a fixed number to s:

for( size_t s = 1000; s < 100000; s = 2 * s )
{
   std::vector<int> test = randomvector(s);
      // A random vector of size s.

   auto test2 = test;
      // Be very careful not to sort the same vector twice.
```

```
      // It is useless.

   timer tt;
   sort1( test );
   alg1. insert( s, tt. time( ));

   tt. clear( );
   sort2( test2 );
   alg2. insert( s, tt. time( ));
}

std::cout << alg1 << "\n";
std::cout << alg2 << "\n";
```

That was a lot of reading, now we are ready to give you the task:

1. Write code that systematically measures the performance of the sorting algorithms `bubble_sort`, `heap_sort`, `quick_sort`, and `insertion_sort`.

   'Systematically measure the performance' means trying inputs of sufficiently many sizes (at least 5).

2. Is there a difference between compilation with optimization on and compilation with optimization off? Compile with and without the flags `-O3 -flto`. In Linux, you can do this by editing the `Makefile`. In Eclipse, you will need to find some other way of editing the flags of the compilers.

   Compare the run times. How big is the difference on average?

3. Establish which sorting algorithms have $O(n^2)$ performance, and which ones have $O(n.\log(n))$ performance.

4. Among those with $O(n^2)$, which one is faster?

5. Among those with $O(n.\log(n))$, which one is faster?

6. If you have time left, you can study the effect of different versions of `ensure_capacity` on the performance. Write:

```
timetable stacktimes( "stack with doubling" );
for( size_t s = 1000; s < 200000;  s = s + s )
{
  timer tt;
  stack st;
  for( size_t i = 0; i < s; ++ i )
     st. push(i);
  stacktimes. insert( s, tt. time( ));
}
std::cout << stacktimes << "\n";
```

Now, in `ensure_capacity`, remove the line:

```
if( c < 2 * current_capacity )
   c = 2 * current_capacity;
```

Make the same measurement again.