CSCI 152, Performance and Data Structures, Lab Exercise 6

Deadline: Monday, 07.04.2025, 23:59

Purpose of this exercise is to show that asymptotic complexity is something real, how it can be measured, and why it is important.

When making measurements, it is important to choose the input size in such a way that the run times are not too short. It is impossible to measure short run times reliably ($< 10^{-3}$) because of unpredictable start up times. If the function that you want to measure has short run times, you should write a loop that repeats the function many times, in order to get sufficient accuracy. Even for bigger run times, the measured time can vary quite a lot between different runs (in the range of 10 percent.) It is useful to run the same test a couple of times and take the average.

In this task, run time is measured by subtracting the start time from the end time, using the system clock. Such measurements are unreliable if there are other processes on your computer, because then your test does not get full CPU time. In particular, make sure that no browsers are open, no updates are being downloaded. Don't watch videos and don't play music at the same time when you are doing serious measurements.

While doing measurements, make sure that compiler optimization is `-O3 -flto`. If you are using Linux, you can simply use the given **Makefile**. Don't use valgrind during measurements. If you are not using Linux, you will still need to turn on compiler optimization. Eclipse allows to set flags to the compiler. The flags must be still `-O3 -flto`.

The starter code includes a few classes that are helpful when performing time measurements. A `timer` object (defined in **timer.h**) measures time by comparing the current system time with the time on which it was created. It has the following methods:

- `timer( )`. Creates a timer.
- `reset( )`. Resets the timer.
- `double time( ) const`. Gives the current counted time in seconds. It is the time since the timer was constructed or reset for the last time.

Class `timer` should be used as follows:

```
timer t;
(algorithm that you want to measure).
double d = t. time( );
   // Time that was taken by algorithm in seconds.
}
```

As explained in the lectures, nearly always only the asymptotic behavior is important. In order to measure the asymptotic behavior of an algorithm, one must run it with inputs of different size, and observe how the run time changes when the input size changes. If an algorithm has linear complexity, doubling the input size will double its run time. If an algorithm has quadratic complexity, then doubling the input size will quadruple its run time.

In order to make systematic measurement easier, you can use the class `timetable` that stores a set of input sizes with their associated run times. It has the following methods:

- `timetable( const std::string& algname )`,
  `timetable( const char* algname )`. Construct a `timetable` for algorithm with given name.
- `void insert( size_t inputsize, double runtime )`. Store an `inputsize` with a `runtime`.
- `clear( )`. Erase 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` and including it in **main.cpp** in a comment. Here is an example of the use of class `timetable`.

```
timetable slow_times( "slowsort" );
timetable fast_times( "fastsort" );

// One should always double the input size:

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

   {
      auto copy = test;  timer t;
      slowsort( copy ); slow_times. insert( s, t. time( ));
   }

   {
      auto copy = test; timer t;
      fastsort( copy );  fast_times. insert( s, t. time( ));
   }
}

std::cout << slow_times << "\n";
std::cout << fast_times << "\n";
```

1 Sorting

1. Write the function `measure::quick_sort( std::vector< int > & vect )` in file **sorting.cpp**. All you need to do, is call `std::sort( )`. This library function uses quicksort, which is the best known sorting algorithm.
2. Run the first loop in `main( )`, which measures the times for the four sorting functions `heap_sort`, `quick_sort`, `bubble_sort`, and `insertion_sort`, and compare the times. Set the size in such a way that the total run time is at least 20 seconds. Write or copy the output times in a comment in your **main.cpp**.
3. Answer the following question (also in a comment in **main.cpp**): Which sorting algorithms have $O(n^2)$, and which have $O(n.\log(n))$?

2 Allocating

In Assignment 1, we implemented a **vector** class that reallocates when it runs out of space. Reallocation was done by the method `reallocate( size_t c )`, which was implemented in such a way that it always doubles capacity, when more space is needed.

We want to try out different allocation strategies and measure their impact on performance. In order to do this, we will implement a simple string class, which cannot be copied (only moved). Since we also want to practice C^{++} a bit more, we will put this string class in a namespace called `csci152`. We will also study the use of the null-pointer in situations where a memory segment of size 0 is needed. This simplifies the implementation of the default constructor and of moving operations.

2.1 Defining Classes in Namespaces

In order to define class `string` in namespace `csci152`, declare it as follows in file **string.h**:

```
#ifndef CSCI152_STRING_
#define CSCI152_STRING_

namespace csci152
{
   class string
   {
      // Everything as usual.
   };
}

#endif
```

Since the include guard must prevent double inclusion only for `csci152::string`, the namespace must be part of the include guard. Otherwise, the include guard would block the first time inclusion of another `string` class in another namespace.

In the **.cpp** file, every reference to string, with the exception of method parameters, must be preceeded by `csci152::` This applies to return values (in case a method returns as `csci152::string`, and the method name itself. Parameters do not need to be preceeded by `csci152::` because, once the compiler knows the function is in namespace `csci152` , it will by default look in namespace `csci152` for the parameter types. The skeletons are given in file **string.cpp**.

2.2 Using the Null Pointer instead of a Pointer to a Zero-Length Segment

Until now, we have always allocated a small memory segment in the default constructor. For `string`, one could write

```
   string( )
      : cap(4), sz(0), data( new char[4] )
   { }
```

It is also possible to write

```
   string( )
      : cap(0), sz(0), data( new char[0] )
   { }
```

This will work fine, but one has to adapt `ensure_capacity`, so that it will not double the capacity, when capacity is 0.
What happens if one writes the following?

```
   string( )
      : cap(0), sz(0), data( nullptr )
   { }
```

This turns out possible, because

- Both the null pointer and the pointer to a memory segment of size 0, cannot be dereferenced with `*` or `[]`.
- `delete[] p` works correctly when `p` is the null pointer.

This means that, as long as one doesn't explicitly check it, or try to add an index to it, the null pointer is not different from a pointer to a memory segment of length 0. So, we will use the last implementation of the default constructor. We will also use it in the moving constructor, and moving assignment. This is more than a little optimization. It is a fundamental difference, because when they do not allocate anything, they can not fail. Hence, it becomes possible to mark the default constructor, moving constructor, and moving assignment with `noexcept`. Containers like `std::vector` use moving copy constructor during reallocation, but only if the moving operators of the class contained in the vector are `noexcept`.

5. Write the default constructor of class `csci152::string`.

6. Write the destructor of `csci152::string`. The `data` pointer can be `nullptr`, but you don't need to check for it, because `delete[]` already checks it.

7. Write the moving constructor of class `csci152::string`. Make sure to leave `s` empty when it is moved-out, and use the null pointer. The simplest implementation is with `std::exchange`.

8. Write moving assignment for class `csci152::string`. Deallocate `data` and move the data. You don't need to worry about self-assignment because moving self-assignment is UB (Undefined Behaviour).

9. Finish `void string::ensure_capacity( size_t c )`. If `cap == 0`, allocate 4 characters. Otherwise, use `new_cap` to allocate a new segment of memory, use `std::copy` to copy the data into the new segment, and deallocate the old segment.

10. Write `void push_back( char c )`. As usual, it should call `ensure_capacity`.

11. Write `void pop_back( )`. There is no need to reallocate.

12. Write `void push_back( const char* s )`. Repeatedly call `push_back( char c )`.

13. Implement `void clear( )`. You don't need to deallocate.

14. Implement `size_t size( ) const`.

15. Implement `bool empty( ) const`. As usual, you should avoid the useless `if`.

16. Now do measurements, using the following three options of `new_cap`:

    ```
    // size_t new_cap = c + 1000;
    size_t new_cap = c * 2;
    // size_t new_cap = c * 3;
    ```

The measurements for the three versions cannot be made at once, because you have to recompile the program in-between. Include your measurements in a comment in file **main.cpp**. For the three cases, try to estimate the order k in the time complexity $\Theta(n^k)$ of the main loop.

3 Submission

You have to submit two files **string.cpp** and **main.cpp**. In file **main.cpp**, you need to include the measurements for the four sorting algorithms, answer the questions about their orders, include measurements for `csci152::string`, using the three versions of `new_cap`, and answer the question about their orders.