

CS 14: Data Structures and Algorithms

Lab 4

Peter H. Fröhlich
phf@cs.ucr.edu

Wagner Truppel
wagner@cs.ucr.edu

Out on: January 27, 2003

Due by: February 2, 2003

Remember to follow the instructions and policies for assignments given on the course website. This week's assignment consists of **three** separate parts:

- an **in lab** programming exercise (1)
- an **at home** programming exercise (2)
- two **at home** written exercises (3 & 4)

The “in lab” part is due at the **end** of your **lab section**. The “at home” parts are due **before 8:00 pm** on the date given above. You have to use the department's electronic turnin service for **everything**, including the **written** part. Put your solution for the latter into the **text file** `solution-4.txt` and turn it in together with the other files.

1 In-Lab: Bubble

You can download an archive for this lab from the course website again. Among other things, you will find a file `testbubble.cpp` which contains test cases for the following two functions:

```
bubble( cs14::Array<T> &a )
bubble_smart( cs14::Array<T> &a )
```

The functions are supposed to sort `Array` instances into **ascending** order using the **bubble sort** algorithm. While `bubble` uses the “dumb” algorithm which always runs all passes, `bubble_smart` is supposed to be “smart” and stop once it discovers that the array is already sorted.

Your task is to develop the implementations of these functions inside a file `sorting`, which you will later extend to include more algorithms. The functions need to

be in namespace `cs14`, and you can use a separate `swap` function if you want.

Hints: Start by first putting “empty” functions into the file `sorting` to make `testbubble.cpp` compile. Then implement `bubble` itself and make it work correctly. Then copy your implementation into `bubble_smart` and extend it with the “smart” parts. The array is (obviously?) sorted if you ever go through it without swapping any elements...

2 At-Home: Selection and Insertion

The archive for this lab also contains the two files `testselection.cpp` and `testinsertion.cpp` with test cases for the following two functions:

```
selection( cs14::Array<T> &a )
insertion( cs14::Array<T> &a )
```

The functions are supposed to sort `Array` instances into **ascending** order using the **selection sort** and **insertion sort** algorithms respectively. As before, your task is to develop the implementations of these functions inside the file `sorting`.

Hints: See above, making things compile first and correct later is useful for this part as well...

3 At-Home: Performance

Now you will investigate the **actual** (not asymptotical) performance of the algorithms you implemented. This is what the three test programs `timebubble.cpp`, `timeselection.cpp`, and `timeinsertion.cpp`

are good for. To obtain accurate results, you should **not** modify these test programs in **any** way!

Each program executes a number of sorting operations on arrays filled with various input data, and you will use the Unix command `time` to measure how long it takes to complete these tests. The `time` command produces slightly different output under various versions of Unix, so please read the manual page (using `man time`) for the details.

You will compile each test program a number of times with different optimization levels. First, you use the compiler option `-O0` (“oh zero”) to **disable** all optimizations. Run the three test programs (using `time`) and record the results.¹ Now you repeat these measurements for all programs compiled with `-O1` (“oh one”), then with `-O2`, and finally with `-O3`. After you are done with this, make a **backup copy** of your current `sorting` file, i.e. the one you just measured.

If you used a `swap` function in your implementation of the sorting algorithms, change your code to perform the swapping **without** a function call. If you swapped inline already (and I don’t refer to the keyword `inline` here!), change your implementation to use a `swap` function instead. Before you do any measurements on this modified implementation, please make sure that you did not break anything by running the test cases from Sect. 1 and Sect. 2 again! Once you are confident that your new implementation works, proceed to repeat the measurements, again recording the performance for all three test programs and for all four optimization levels. After you are done, make a **backup copy** of your current `sorting` file, as you’re about to modify it once again.

You now have data to compare the performance of these algorithms, and for the next part you should start with whatever turned out to be most efficient on your machine. Your final task is try to optimize your implementations **by hand**, i.e. by tweaking the code in ways that you think will make things faster. Note that this is an **open-ended** exercise and if you can’t come up with any improvements, that’s an okay result as well. Just try your hand at it, and maybe you can beat all the other sorting code that will get turned in for this lab. Of course, you should collect the performance data for your “fine-tuned” versions as well, without doing this you won’t be able to detect any improvement.

For your written answer, you should include the performance data you collected (each in the form of a table), compare the actual performance of these algorithms

to their asymptotic complexity, and explain your “theories” (if you have any) on why certain implementations are faster or slower than others.

Hint: You can save a **lot** of time by writing a makefile or shell script that compiles, runs, and records things for you...

4 At-Home: Fancy Selection

The basic selection sort algorithm proceeds by finding the maximum (or minimum) element in the unsorted part of an array and swapping this element to the end (or the beginning). Consider a “fancier” selection sort algorithm that proceeds by doing **both** things at the same time. That is, it finds the minimum **and** maximum element in the unsorted part and swaps them to the beginning **and** end respectively. Answer the following questions:

1. What is the **time complexity** of this algorithm in the best, worst, and average case? Give reasonable arguments (not proofs though) to support your claims.
2. Is the algorithm **stable**? That is, do equal elements keep their relative positions during sorting? Give a reason for your answer.
3. In terms of actual performance (not complexity, but **actual** time), where do you expect the algorithm to fall between bubble sort, selection sort, and insertion sort?
4. What is the major consideration for **correctness** when implementing this algorithm? You should probably at least sketch the algorithm as a modification of the standard selection sort to answer this one...

Hint: A good way to answer many of these questions accurately is to actually **implement** the algorithm...

1. If you want to be “really accurate” you should probably measure three executions of each program and then take down the **average** time needed.