Laboratory 13: Cover Sheet

Name: Terence Henriod Date: 10/2/2013

Section: 1001

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Implementation Testing	X	X
Programming Exercise 1	X	X
Programming Exercise 2	X	X
Programming Exercise 3	X	X
Analysis Exercise 1		
Analysis Exercise 2		
	Total	4

Laboratory 13: Implementation Testing

Name: Terence Henriod Date: 10/2/2013

Section: 1001

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

Question 1: What is the resolution of your implementation—that is, what is the shortest time interval it can accurately measure?

See the second table below for an approximate answer to this question.

Test Plan 13-1a (Timer ADT operations)			
Test case	**Actual time period (in seconds)**	**Measured time period (in seconds)**	**Checked**
0 seconds	Assumed to be 0	1.1 E-5	X
1 seconds	Assumed to be 1	1.00099	X
3 seconds	Assumed to be 3	3.01275	X
5 seconds	Assumed to be 5	5.01811	X
7 seconds	Assumed to be 7	7.0172	X

The above test was performed using test17.cpp, where the program was told to stop the timer after a certain number of seconds, and the time recorded by the Timer class was reported.

The following test was performed by writing code that started the stopwatch on one line and stopped it on the very next line. The elapsed time was recorded, and the average of all the samples was taken. This way, we could attempt to find the shortest amount of time (on average) that the Timer class can be used to measure.

Test Plan 13-1b (Timer ADT operations)			
Test case	**Actual time period (in seconds)**	**Measured time period (in seconds)**	**Checked**
Trial 1 (1 million samples)	Nearly 0	1.6641 E-8	X
Trial 2 (1 million samples)	Nearly 0	1.7773 E-8	X
Trial 3 (1 million samples)	Nearly 0	1.735 E-8	X

Laboratory 13: Measurement and Analysis Exercise 1

Name: Terence Henriod Date: 10/2/2013

Section: 1001

In the table below, fill in values of *N*, 2*N*, and 4*N*: try 1000, 2000, 4000. If you do not obtain meaningful timing data—especially for `binarySearch`—change the value of *N* and try again.

Timings Table 13-2 (Search routines execution times)

Routine	Number of keys in the list (numKeys)						
	N = 1000	2N = 2000	4N = 4000	8N = 8000	16N = 16000	32N = 32000	64N = 64000
`linearSearch` O(N)	0.556328	1.095280	2.186530	4.352160	8.710100	17.4536	34.83890
`binarySearch` O(Log N)	0.014348	0.015551	0.016954	0.018073	0.019426	0.02487	0.022595
`STLSearch` O(N)	0.544524	1.090550	2.149960	4.298230	8.583120	17.2620	34.34180

Please list times in seconds

Question 1: How well do your measured times conform to the order-of-magnitude estimates given for the `linearSearch` and `binarySearch` routines?

My measured times conform well to the order of magnitude estimates. The linear search time increases by approximately the specified factor, and the binary search increases rather slowly as the multiplicative factors of the number of keys increases, as we might expect, given the properties of logarithms (log(k*N) = log(k) + log(N)). Both conform well.

Question 2: Using the code in the file *search.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the `STLSearch` routine. Briefly explain your reasoning behind this estimate.

I would estimate that the STL Search has an order-of-magnitude that is linear. My reasoning is that first, the times for STL Search mirror those of `linearSearch` (they are only slightly faster), and second, the times differ by approximately the scaling factor used for `numKeys` in each test.

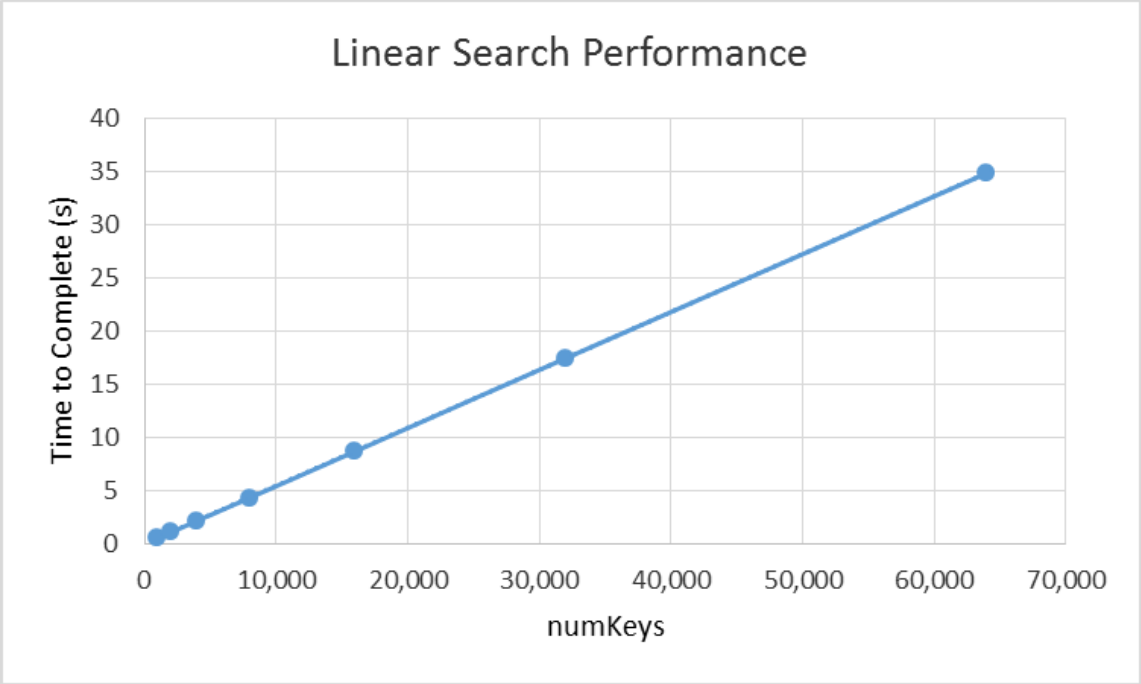

Figure 1: A graph of the `linearSearch`'s performance times. Although it cannot be seen, there has been a linear trend line fit to this graph. It can't be seen because the data line covers it. This supports the claim that the `linearSearch` has a linear operation complexity cost.

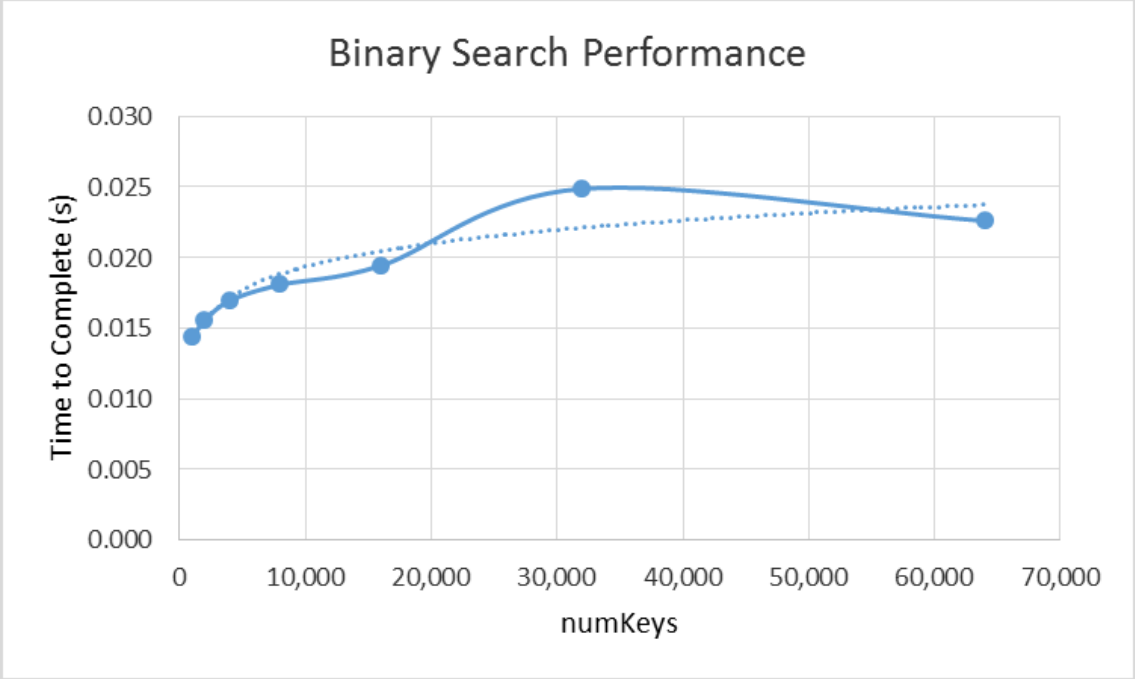

Figure 2: A graph of the `binarySearch`'s performance. A logarithmic trend line has been superimposed on this graph. Although graphically the binary search did not fit the trend line well, it is difficult to imagine what other trends the sort might fit. The strange fluctuations are likely due to random chance and performance issues with the computer's execution of the `binarySearch`. When considering this graph, we should take care to remember that order-of-magnitude/complexity estimates are just that, estimates. Actual results may vary.

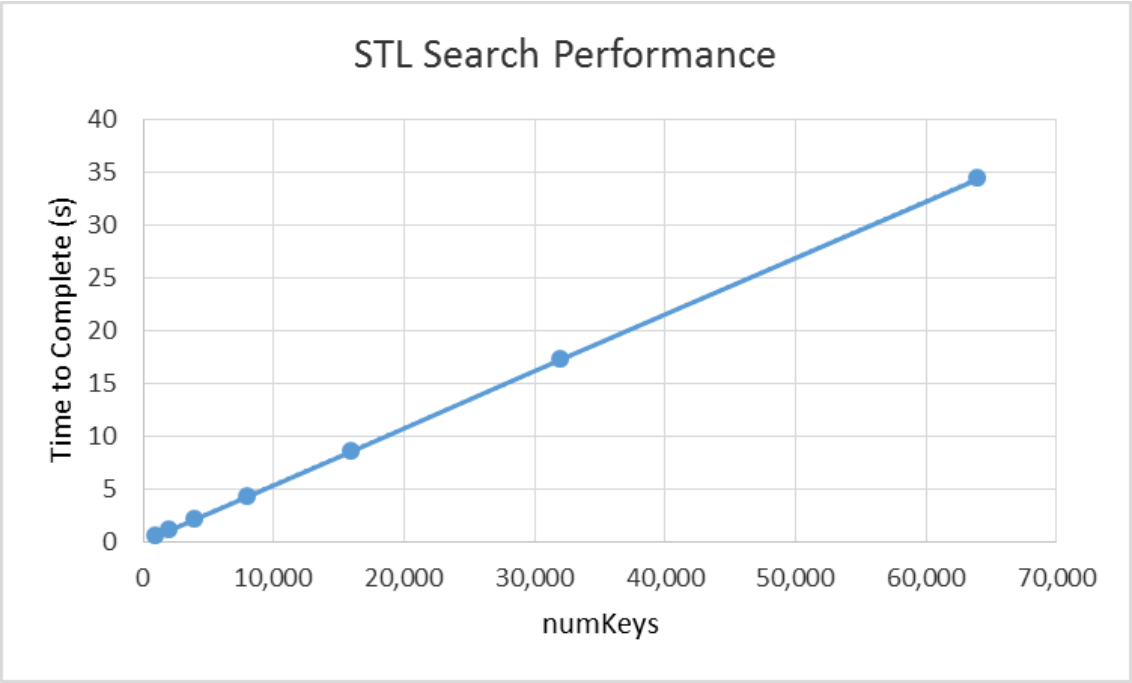

Figure 3: A graph of the STL `search`'s performance. Again, a linear trend line was included in the graph, and again, the trend line was covered up, lending support to the claim that the STL `search` has a linear complexity cost.

Laboratory 13: Measurement and Analysis Exercise 2

Name: Terence Henriod Date: 10/2/2013

Section: 1001

In the table below, fill in values of *N*, 2*N*, and 4*N*: try 1000, 2000, 4000. If you do not obtain meaningful timing data—especially for `quickSort`—change the value of *N* and try again.

Timings Table 13-3 (Execution times of a set of sorting routines)

Routine	Number of keys in the list (numKeys)						
	N = 1000	2N = 2000	4N = 4000	8N = 8000	16N = 16000	32N = 32000	64N = 64000
`selectionSort` O(N^2)	0.507282	1.97771	7.87431	31.24490	125.359	499.936	2019.37
`quickSort` O(N Log N)	0.020712	0.046322	0.10232	0.215066	0.469197	1.03088	2.13300
STL `sort` O(N Log N)	0.01522	0.03363	0.072205	0.155237	0.348939	0.744592	1.60827

Please list times in seconds

Question 1: How well do your measured times conform with the order-of-magnitude estimates given for the `selectionSort` and `quickSort` routines?

Again, the measured times conform well to the order-of-magnitude estimates. The `selectionSort` increases by approximately the square of the scaling factor used for `numKeys`. It can be seen that the increase in time for quicksort does seem to increase approximately by the scaling factor for `numKeys` as well (but the increase is a little faster due to the "+ log(scaling factor)" part of the decomposition of a logarithm).

Question 2: Using the code in the file *sort.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the STL `sort` routine. Briefly explain your reasoning behind this estimate.

I would assert that the STL `sort` has an N log(N) order-of-magnitude estimate. At first, the times appear to increase linearly, but as the scaling factor increases, the times increase faster than would be indicative of a linear trend.

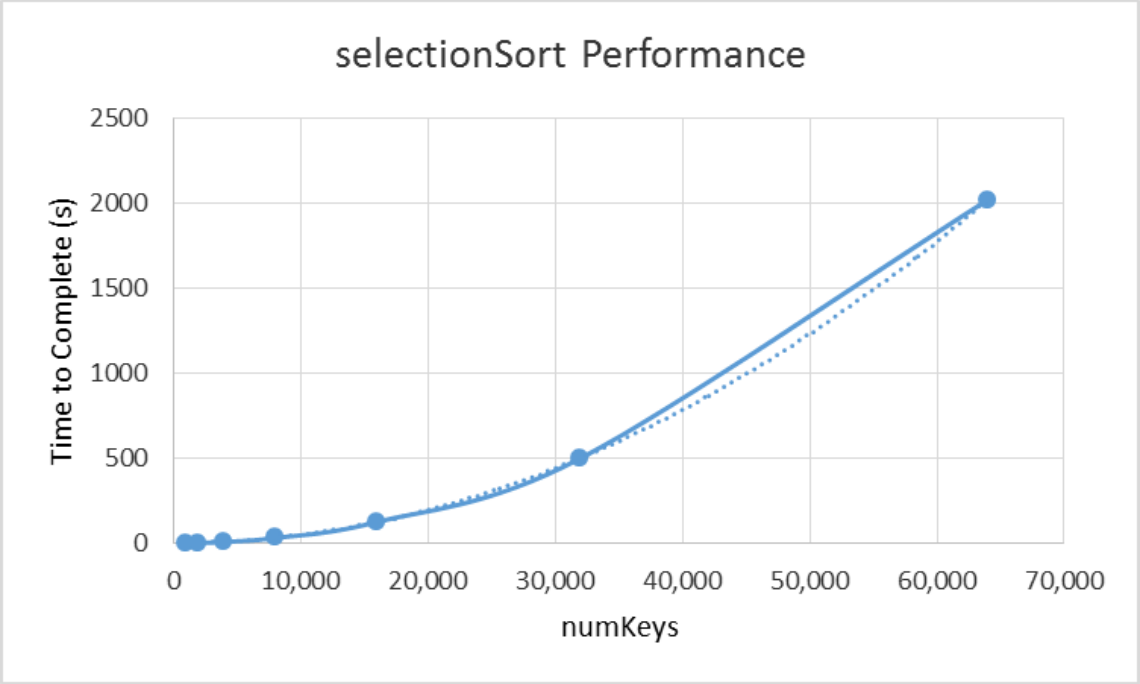

Figure 4: A graph of Selection Sort's performance. A trend line for N^2 is superimposed onto the graph. Although the fit isn't perfect, it can be seen that `selectionSort`'s complexity cost is likely at or near a quadratic model.

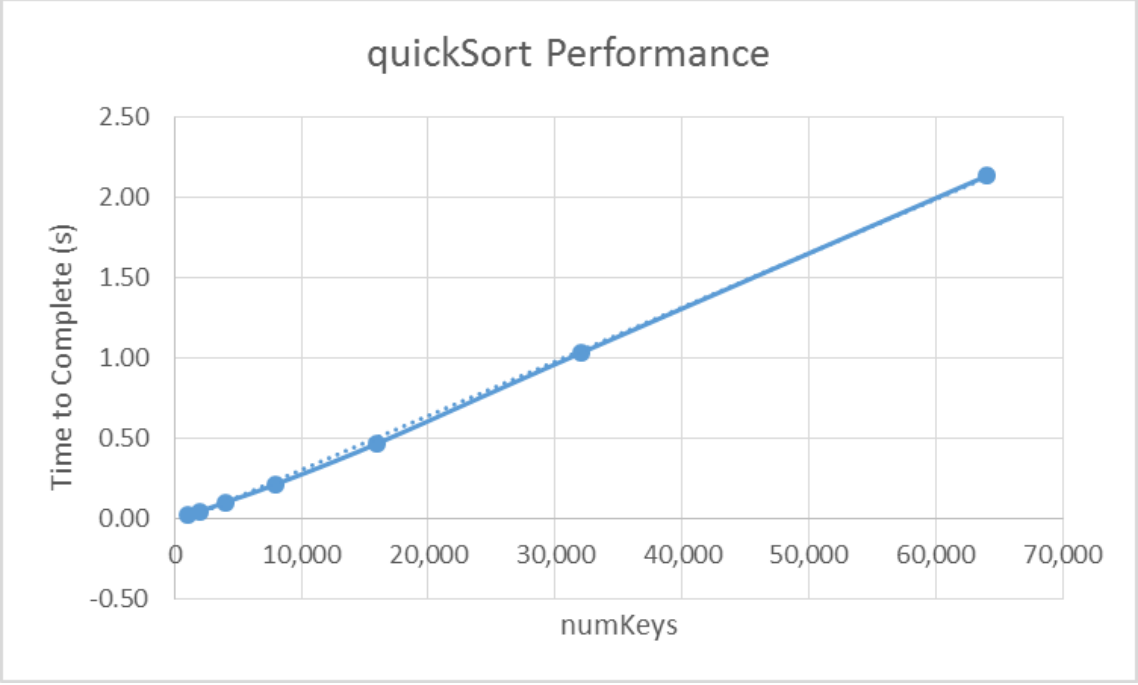

Figure 5: A graph of `quickSort`'s performance. Because Excel does not include an option to plot a N log(N) trend line, I substituted a linear one. When graphing N log(N) against a linear curve, it can be seen that given proper scaling factors and an appropriate frame of reference, it becomes apparent that N log(N) starts out being lower than a linear curve, but later on the N log(N) curve begins to outpace the linear curve's rate of growth. In this graph, the `quickSort` performance starts out lower than the linear trend, but towards the highest data point, it begins to go above the linear trend line, weakly supporting the claim that `quickSort` has an order-of-magnitude estimate of N log (N).

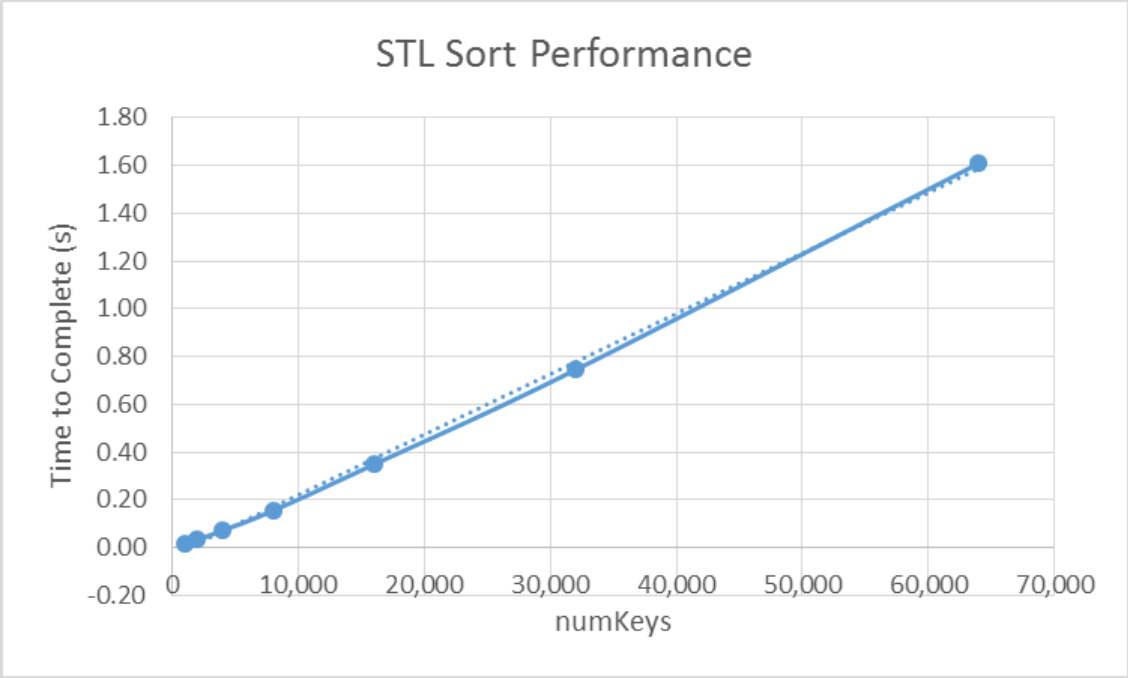

Figure 6: A graph of the STL Sort performance. Again, I used a linear trend line (due to Excel's inadequacy for this kind of graphing) to illustrate the traits that we'd expect to observe for an N log(N) pattern. We observe the same kind of under-then-over behavior we saw when referencing the linear trend line as we did with quickSort, but this time the line is more curved, so the claim that the STL Sort is a little more strongly supported this time. More so than it was with the quickSort (Figure 5).

Laboratory 13: Measurement and Analysis Exercise 3

Name: Terence Henriod Date: 10/2/2013

Section: 1001

In the table below, fill in values for the various constructor tests. Try an initial value of *N*=1000. If you do not obtain meaningful timing data, change the value of *N* and try again.

Timings Table 13-4 (Timing constructor/initialization just before vs. inside loop)		
	Constructor/initialization location	
Your value of N: 64000	Outside loop	Inside loop
`int`	0.006183	0.005473
`double`	0.002777	0.003129
`vector`	10.53010	101.8190
`TestVector`	10.35350	101.4630

Please list times in seconds

Question 1: For each data type, how do your measured times for the constructor just before the loop compare to the times for the constructor inside the loop? What might explain any observed differences?

In general, constructing inside the loop was slower. The only exception was with integers. This does not make intuitive sense, so I would assert that this is due to some kind of "compiler magic," that "saw" what was happening with the integers inside that loop and applied some sort of optimization. Granted, this does not make any intuitive sense. There should be a difference between constructing an object once and doing N operations, and constructing an object N times and doing that operation N times as well.

In the table below, fill in values for the various increment tests. Try an initial value of *N*=1000. If you do not obtain meaningful timing data, change the value of *N* and try again.

Timings Table 13-5 (Timing pre-/post-increment operators)		
Your value of *N*: 8000	pre-increment	post-increment
`int`	0.004544	0.002730
`double`	0.005664	0.002641
`TestVector`	136.829	138.553

Please list times in seconds

Question 1: For each data type, how do your measured times for the pre-increment operator compare to the times for the post-increment operator? What might explain any observed differences?

For the primitive data types, post incrementing was the faster of the two. This is not in accordance with what is said in many online forums, as it is said that the compiler should optimize away any difference. The forums also say that because the operations do the same thing, there should be little difference. When looking at the definitions of the operators (in general), it would seem that the postfix operator should be slower because the postfix increment operator needs to make a copy of the object being incremented (unlike with the pre increment operator).

However for the TestVector, the pre-increment was faster. This makes sense because when looking at the code for TestVector, the overloading of the post increment operator actually calls the pre increment operator, as well as performs additional actions.