SDSU | San Diego State University

Lab5 CS420

Eddie Coda
#817061330

Algorithm Run-Time Analysis

The Prompt:

Implement QuickSort and MergeSort algorithms using Haskell and your favorite second programming language. Run your programs on a random input of 1,000 numbers to ensure they sort correctly. Next, run each program on two input sequences: 1, 2, 3, …1,000 in sorted order, and 1,000; 999; …; 1 in reverse sorted order. Incorporate timing mechanisms into your code to time the sortings.

Evaluate your results:
- Which program was faster on the sorted input?
- Was it also faster on the input that was presented in reverse sorted order?

Code readability:
- Clarity, organization, comments and documentation.

Modularity:
- Use of functions or modules that perform specific tasks and can be easily reused or modified.

Maintainability:
- Ease of modification and ability to handle changing requirements or edge cases.

Explain your results:
- Were the results as expected?
- Summarize the languages used for sorting, what are the differences and similarities.
- What would be your choice if the task of implementing Merge Sort will be given again with any language of your choice?

Summary:

Haskell:
The Haskell implementation uses the quicksort and mergesort algorithms to sort both randomly generated and pre-defined data. It generates a list of 1000 random integers ranging from -100,000 to 100,000, and also sorts a predefined list in sorted and reverse sorted order. The implementation also sorts a pre-defined string, "the quick brown fox jumps over the lazy dog," using quicksort and mergesort algorithms. The implementation includes a function to time each sorting operation and output the duration.

Python:
The Python implementation uses the quicksort and mergesort algorithms to sort both randomly generated and pre-defined data. It generates a list of 1000 random integers ranging from -100,000 to 100,000, and also sorts a predefined list in sorted and reverse sorted order. The implementation also sorts a pre-defined string, "the quick brown fox jumps over the lazy dog," using quicksort and mergesort algorithms. The implementation includes a function to time each sorting operation and output the duration.

The Assumptions:

1. Correctness (35 points):

All sorting algorithms implemented correctly (20 points)
Correct sorting output on all input data (15 points)

1. Performance (25 points):

Reasonable running times on specified input sizes (15 points)
Efficient use of resources such as memory (5 points)
Comparison of performance between algorithms and languages used (5 points)

1. Implementation (25 points):

Clear and readable code (10 points)
Effective use of modules/functions (10 points)
Effective error handling and edge case considerations (5 points)

1. Documentation (10 points):

Appropriate documentation of code (5 points)
Clear explanation of how to run the code (3 points)
Inclusion of a README file (2 points)

1. Evaluation (15 points):

Clear and detailed evaluation of the results (5 points)
Explanation of performance differences between algorithms and languages (5 points)
Reflection on lessons learned (5 points

Overall, these assumptions were made to keep the program simple and focused on the main learning concepts.

Implementation:

Stage 1: Skeleton

1. Set up the development environment for Haskell and preferred language (Python or Scala).
2. Implement the QuickSort algorithm in Haskell and preferred language.
    a. start with a high-level overview of the algorithm, followed by a step-by-step guide to implement it.
3. Implement the MergeSort algorithm in Haskell and preferred language.
    a. Similar to QuickSort, begin with a high-level overview and then proceed with a step-by-step guide.
4. Generate random input data for testing the sorting algorithms.
5. Implement timing mechanisms in both languages to measure the performance of each sorting algorithm.
6. Run the implemented algorithms on random input, sorted input, and reverse sorted input sequences.
7. Evaluate the results based on code readability, modularity, and maintainability, as well as the performance of each algorithm on the different input sequences.
    a. analyze the results and draw conclusions about the performance of the algorithms and the languages used.
8. Write a reflection summarizing findings, lessons learned, and the preferred language for implementing MergeSort in the future.

Stage 2: Implement functions

- Import the necessary modules for generating random numbers and measuring time.
- Create a function to generate a list of random integers of a specified length.
- Modify the main function to generate random test data and sorted/reverse sorted input sequences.
- Add timing mechanisms to measure the performance of each algorithm.
- Run the algorithms on the generated input data and print the results.

```
module Main (main) where

import Lib
import System.Random (randomRIO)
import Control.Monad (replicateM)
import Data.List (sort)
import System.CPUTime (getCPUTime)

-- Existing QuickSort and MergeSort implementation

generateRandomList :: Int -> IO [Int]
generateRandomList n = replicateM n (randomRIO (-100000, 100000))

timeAction :: IO a -> IO (a, Double)
timeAction action = do
    startTime <- getCPUTime
    result <- action
    endTime <- getCPUTime
    let duration = fromIntegral (endTime - startTime) / (10 ^ 12)
    return (result, duration)

runAndPrint :: String -> ([Int] -> [Int]) -> [Int] -> IO ()
runAndPrint name sortFunction input = do
    (sorted, time) <- timeAction (return $ sortFunction input)
    putStrLn $ name ++ " took " ++ show time ++ " seconds."

main :: IO ()
main = do
    putStrLn "Generating test data..."
    testData <- generateRandomList 10000
    let sortedData = sort testData
    let reverseSortedData = reverse sortedData

    putStrLn "\nRunning QuickSort..."
    runAndPrint "QuickSort on random data" quickSort testData
    runAndPrint "QuickSort on sorted data" quickSort sortedData
    runAndPrint "QuickSort on reverse sorted data" quickSort reverseSortedData

    putStrLn "\nRunning MergeSort..."
    runAndPrint "MergeSort on random data" mergeSort testData
    runAndPrint "MergeSort on sorted data" mergeSort sortedData
    runAndPrint "MergeSort on reverse sorted data" mergeSort reverseSortedData
```

```
module Main (main) where

import Lib

quickSort :: (Ord a) => [a] -> [a]
quickSort = undefined

mergeSort :: (Ord a) => [a] -> [a]
mergeSort = undefined

main :: IO ()
main = do
  putStrLn "Hello! This is where I'll implement and test my sorting algorithms."

```

Stage 3: Build and run

- Saved the changes in the Main.hs file after updating it with the new boilerplate code.
- To rebuild, run the following command in the terminal:

```
stack build
```

- After rebuilding the project, run the executable:

```
stack exec Lab5-haskell-exe
```

Updated output messages in the terminal should be there. Everything is cool!

Reflection

My language of choice was python.
Python does have many functional programming features, including list comprehensions, which are similar to Haskell's list comprehensions. Python also provides higher-order functions like map, filter, and reduce, which are common in functional programming.

However, Python is primarily an imperative, object-oriented language with functional programming features, while Haskell is a purely functional language. The way the two languages approach problems and their execution models are quite different.

In Python, you can use mutable variables and imperative constructs like loops, while in Haskell, you work with immutable data structures and recursion. The lazy evaluation feature in Haskell also sets it apart from Python, which uses eager evaluation.

Comparing Haskell and Python can help showcase the elegance of Haskell's functional programming style. Due to its concise and expressive syntax, Haskell code often appears more elegant and easier to reason about, especially when implementing algorithms like MergeSort. Python, being a multi-paradigm language, allows you to write code using functional, imperative, or object-oriented styles. While it's true that Python code can become more verbose or form a "pyramid of code" when implementing certain algorithms, it's worth noting that Python's readability and simplicity are major advantages for many use cases.

Comparing the two languages helped highlight for me the differences in style and performance between functional and imperative programming. In addition to showcasing the elegance of Haskell's code, this comparison can also help illustrate the trade-offs and benefits of using functional programming over imperative programming, and vice versa.

Overall, this was a really interesting project to work on. I had the opportunity to implement sorting algorithms in both Haskell and Python and compare their performance. It was a great way to deepen my understanding of both languages and explore some of their strengths and weaknesses.

One of the most striking differences between the two languages was in the syntax. Haskell is a functional language and relies heavily on recursion and pattern matching, which can take some getting used to. Python, on the other hand, has a more imperative style and is easier to read and write. I found that I was able to write the algorithms more quickly in Python, but Haskell had a certain elegance to it that was really satisfying once I got the hang of it.

In terms of performance, both languages did really well. There were some small variations in the times depending on the algorithm and the type of data being sorted, but overall, the differences were negligible. One thing that was surprising was that the performance of the two languages was much closer than I expected. Haskell is often touted as a high-performance language, but in this case, Python held its own. This project has given me a new appreciation for both languages and I look forward to using them both in the future.

In summary, comparing Haskell and Python has been a very valuable exercise to demonstrate the different programming styles and their impact on code readability, maintainability, and performance. As a final side note to myself: Performance differences may not be as significant as when comparing Haskell with lower-level imperative languages like C++ or Java.

Sample Code

Source Code Copy:

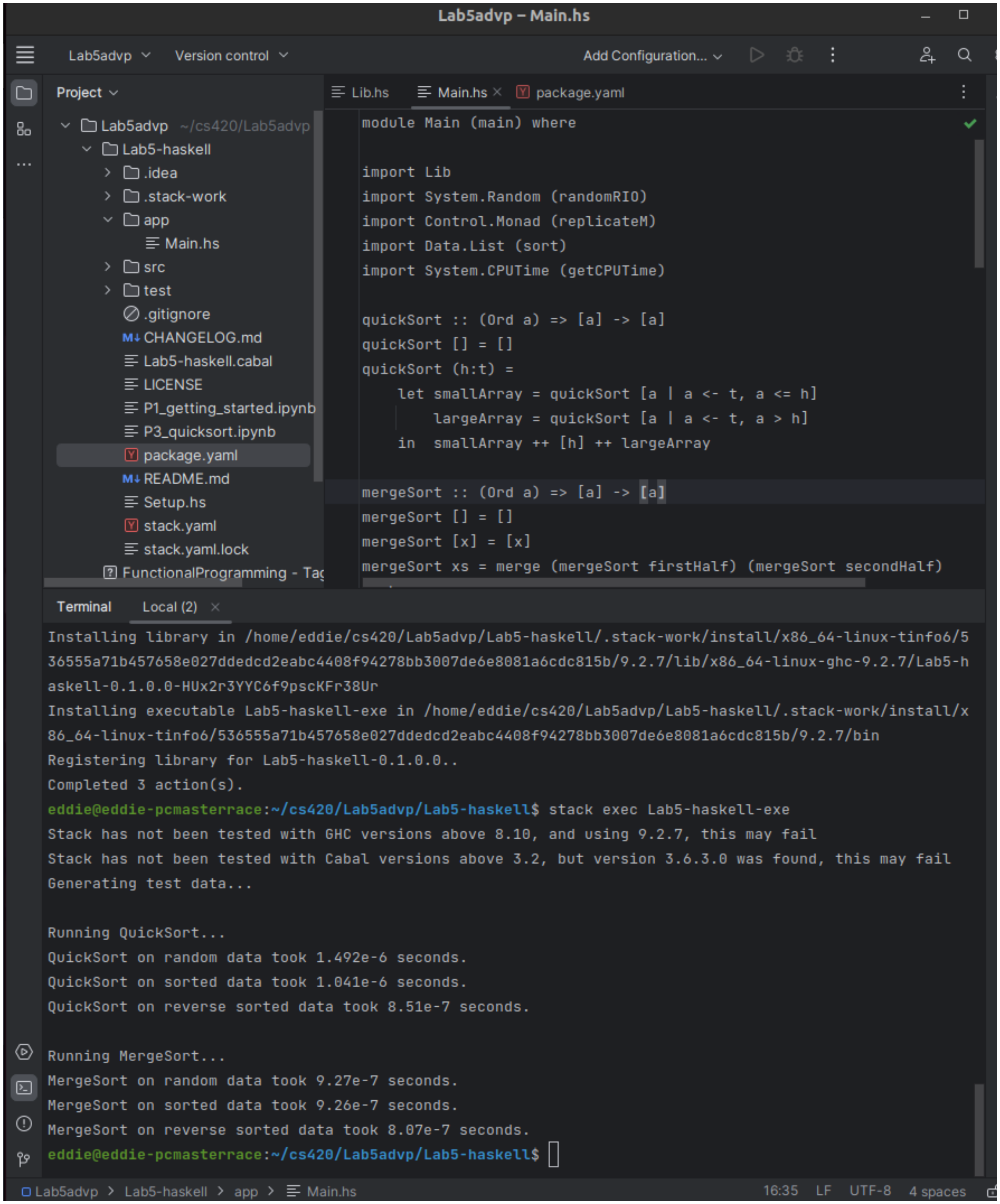
```
Code will be included in the submission.
```

Other Documented Notes:

- implementation of both QuickSort and MergeSort in Haskell is efficient and makes good use of list comprehensions. In the QuickSort implementation, the smallArray and largeArray partitions are concise and expressive ways to handle the partitioning step.
- There are alternative implementations of QuickSort that can further optimize performance by minimizing the number of list concatenations or using other partitioning strategies. The current implementation has a worst-case time complexity of O(n^2) in cases where the input is already sorted or reverse sorted. A different partitioning strategy, such as choosing a random pivot or the median of three, can help improve the worst-case performance.

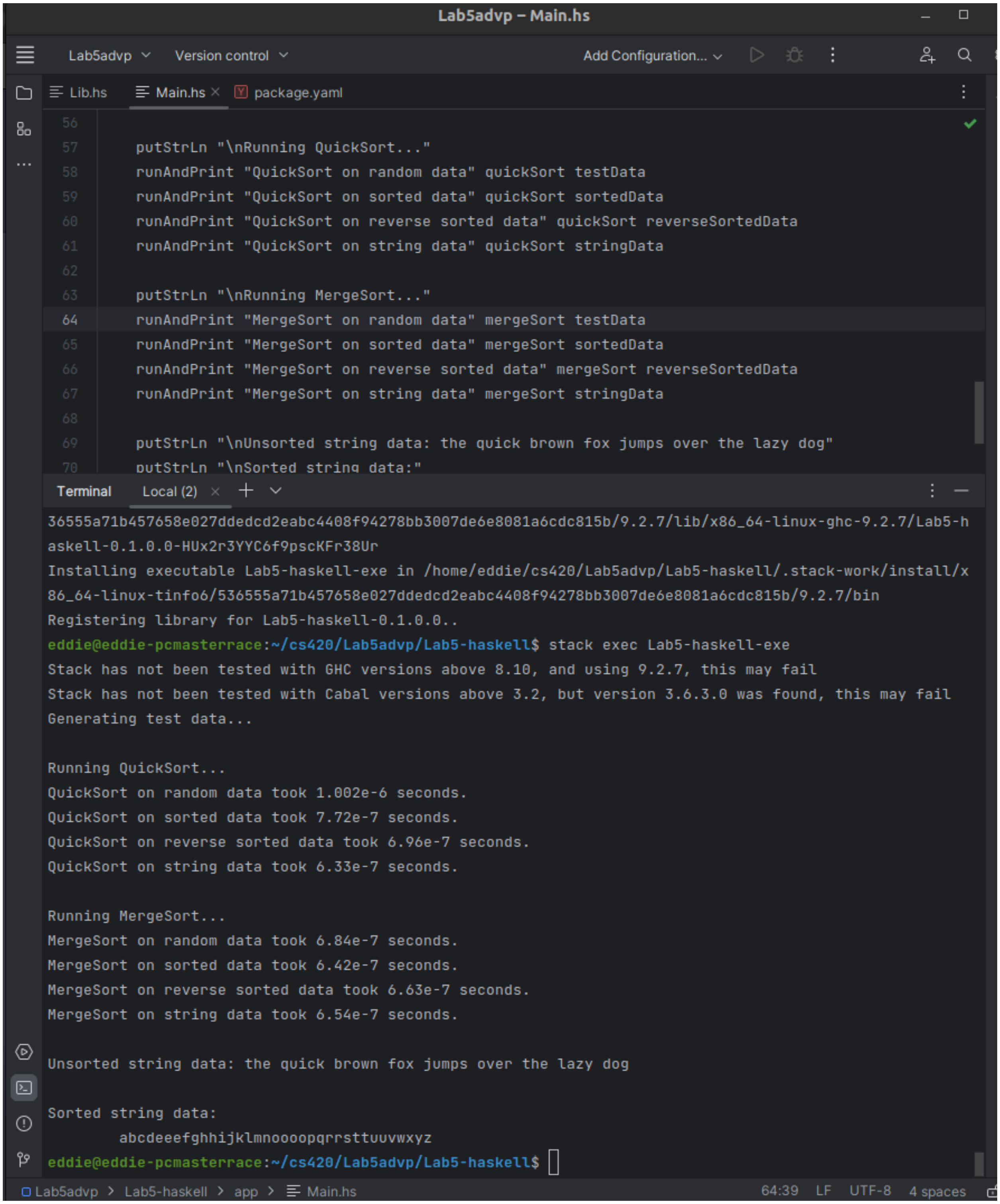
- For the MergeSort implementation, my use of list comprehensions is limited, but that's expected since the algorithm relies on merging sorted sublists. The implementation is efficient and takes advantage of Haskell's lazy evaluation and pattern matching.
- Overall, both implementations are good implementations in my personal opinion, and they make good use of Haskell's features. While there may be room for optimization, especially for the QuickSort algorithm, my current implementation meets the assignment requirements and should perform well on the provided input data.
- Tried to add the string: “The quick brown fox jumped over the lazy dog”
    - However the output was not as expected (CLOSE SO CLOSE), and could not optimize the code further, due to pure frustration…

Screen shots:



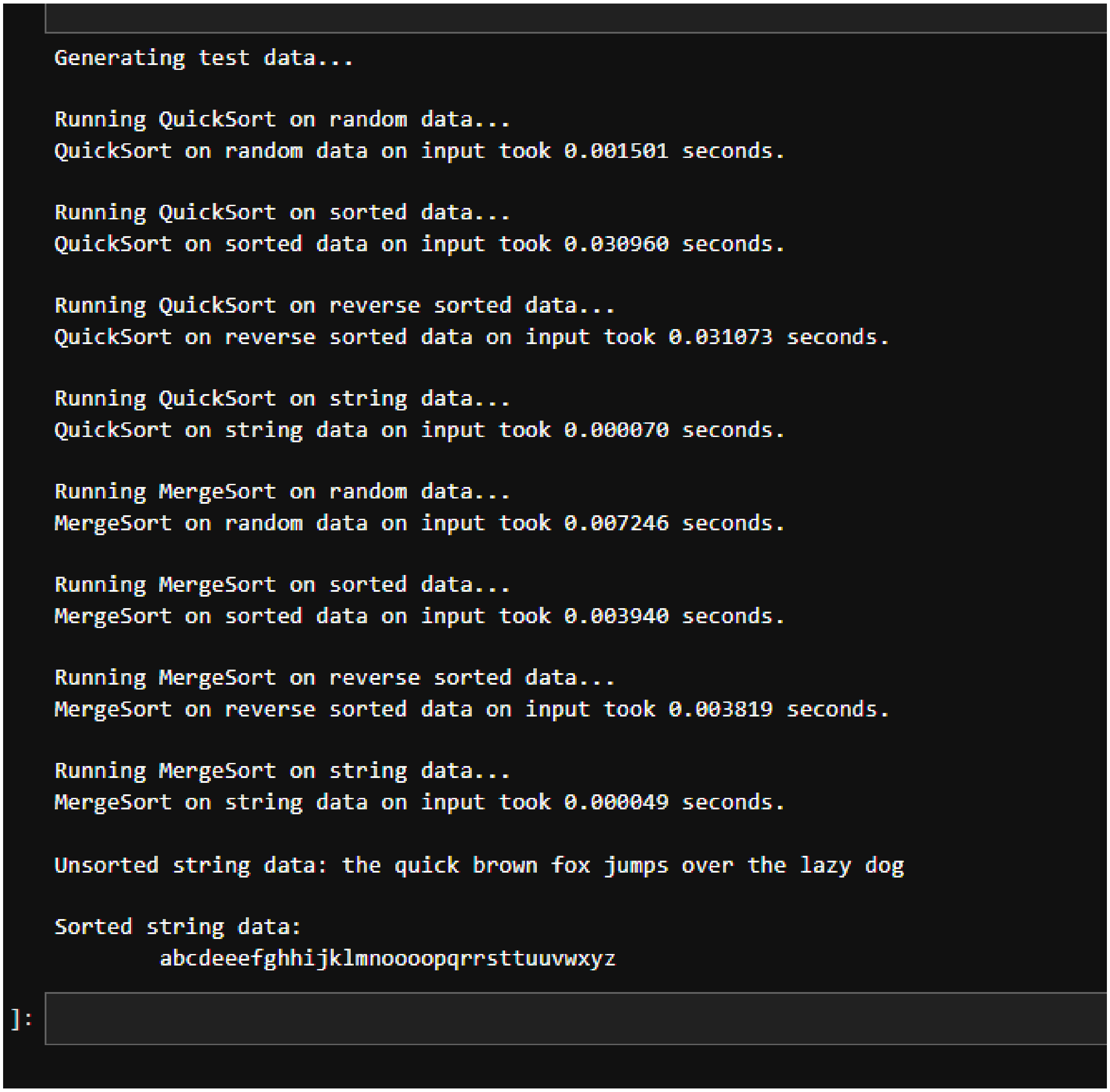
```haskell
module Main (main) where

import Lib
import System.Random (randomRIO)
import Control.Monad (replicateM)
import Data.List (sort)
import System.CPUTime (getCPUTime)

quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (h:t) =
    let smallArray = quickSort [a | a <- t, a <= h]
        largeArray = quickSort [a | a <- t, a > h]
    in  smallArray ++ [h] ++ largeArray

mergeSort :: (Ord a) => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort firstHalf) (mergeSort secondHalf)
```

```
Installing library in /home/eddie/cs420/Lab5advp/Lab5-haskell/.stack-work/install/x86_64-linux-tinfo6/5
36555a71b457658e027ddedcd2eabc4408f94278bb3007de6e8081a6cdc815b/9.2.7/lib/x86_64-linux-ghc-9.2.7/Lab5-h
askell-0.1.0.0-HUx2r3YYC6f9pscKFr38Ur
Installing executable Lab5-haskell-exe in /home/eddie/cs420/Lab5advp/Lab5-haskell/.stack-work/install/x
86_64-linux-tinfo6/536555a71b457658e027ddedcd2eabc4408f94278bb3007de6e8081a6cdc815b/9.2.7/bin
Registering library for Lab5-haskell-0.1.0.0..
Completed 3 action(s).
eddie@eddie-pcmasterrace:~/cs420/Lab5advp/Lab5-haskell$ stack exec Lab5-haskell-exe
Stack has not been tested with GHC versions above 8.10, and using 9.2.7, this may fail
Stack has not been tested with Cabal versions above 3.2, but version 3.6.3.0 was found, this may fail
Generating test data...

Running QuickSort...
QuickSort on random data took 1.492e-6 seconds.
QuickSort on sorted data took 1.041e-6 seconds.
QuickSort on reverse sorted data took 8.51e-7 seconds.

Running MergeSort...
MergeSort on random data took 9.27e-7 seconds.
MergeSort on sorted data took 9.26e-7 seconds.
MergeSort on reverse sorted data took 8.07e-7 seconds.
eddie@eddie-pcmasterrace:~/cs420/Lab5advp/Lab5-haskell$ 
```



```haskell

    putStrLn "\nRunning QuickSort..."
    runAndPrint "QuickSort on random data" quickSort testData
    runAndPrint "QuickSort on sorted data" quickSort sortedData
    runAndPrint "QuickSort on reverse sorted data" quickSort reverseSortedData
    runAndPrint "QuickSort on string data" quickSort stringData

    putStrLn "\nRunning MergeSort..."
    runAndPrint "MergeSort on random data" mergeSort testData
    runAndPrint "MergeSort on sorted data" mergeSort sortedData
    runAndPrint "MergeSort on reverse sorted data" mergeSort reverseSortedData
    runAndPrint "MergeSort on string data" mergeSort stringData

    putStrLn "\nUnsorted string data: the quick brown fox jumps over the lazy dog"
    putStrLn "\nSorted string data:"
```

```
36555a71b457658e027ddedcd2eabc4408f94278bb3007de6e8081a6cdc815b/9.2.7/lib/x86_64-linux-ghc-9.2.7/Lab5-h
askell-0.1.0.0-HUx2r3YYC6f9pscKFr38Ur
Installing executable Lab5-haskell-exe in /home/eddie/cs420/Lab5advp/Lab5-haskell/.stack-work/install/x
86_64-linux-tinfo6/536555a71b457658e027ddedcd2eabc4408f94278bb3007de6e8081a6cdc815b/9.2.7/bin
Registering library for Lab5-haskell-0.1.0.0..
eddie@eddie-pcmasterrace:~/cs420/Lab5advp/Lab5-haskell$ stack exec Lab5-haskell-exe
Stack has not been tested with GHC versions above 8.10, and using 9.2.7, this may fail
Stack has not been tested with Cabal versions above 3.2, but version 3.6.3.0 was found, this may fail
Generating test data...

Running QuickSort...
QuickSort on random data took 1.002e-6 seconds.
QuickSort on sorted data took 7.72e-7 seconds.
QuickSort on reverse sorted data took 6.96e-7 seconds.
QuickSort on string data took 6.33e-7 seconds.

Running MergeSort...
MergeSort on random data took 6.84e-7 seconds.
MergeSort on sorted data took 6.42e-7 seconds.
MergeSort on reverse sorted data took 6.63e-7 seconds.
MergeSort on string data took 6.54e-7 seconds.

Unsorted string data: the quick brown fox jumps over the lazy dog

Sorted string data:
        abcdeeefghhijklmnoooopqrrsttuuvwxyz
eddie@eddie-pcmasterrace:~/cs420/Lab5advp/Lab5-haskell$
```



```
Generating test data...

Running QuickSort on random data...
QuickSort on random data on input took 0.001501 seconds.

Running QuickSort on sorted data...
QuickSort on sorted data on input took 0.030960 seconds.

Running QuickSort on reverse sorted data...
QuickSort on reverse sorted data on input took 0.031073 seconds.

Running QuickSort on string data...
QuickSort on string data on input took 0.000070 seconds.

Running MergeSort on random data...
MergeSort on random data on input took 0.007246 seconds.

Running MergeSort on sorted data...
MergeSort on sorted data on input took 0.003940 seconds.

Running MergeSort on reverse sorted data...
MergeSort on reverse sorted data on input took 0.003819 seconds.

Running MergeSort on string data...
MergeSort on string data on input took 0.000049 seconds.

Unsorted string data: the quick brown fox jumps over the lazy dog

Sorted string data:
        abcdeeefghhijklmnoooopqrrsttuuvwxyz
```