Objective

Three different sorting algorithms (quicksort, heapsort, and insertion-sort) will be implemented in the Haskell programming language. The results and performance will be documented, analyzed, and compared to other implementations of the same algorithms.

Pseudo-Code

Below are examples of pseudo-code, provided by Cormen, Leiserson, Rivest, and Strein, for each algorithm and the equivalent Haskell implementation.

QuickSort

Cormen, Leiserson, Rivest, and Strein (CLRS) psuedocode[1]:

```
quicksort(array A, int p, int r)
  if p < r
    q = partition(A, p, r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)

partition(array A, int p, int r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if A[j] <= x
      i = i + 1
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Haskell implementation using list comprehensions[2]:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort []     = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted  = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

[1] *Introduction to Algorithms, 3rd edition,* pp 171
[2] Quicksort code provided by http://learnyouahaskell.com/recursion#quick-sort

Notice how the Haskell implementation is smaller and more concise than the provided pseudocode.

InsertionSort

Cormen, Leiserson, Rivest, and Strein (CLRS) psuedocode[3]:

```
insertionSort(array A)
  for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

Haskell implementation[4]:

```
insertionSort :: (a -> a -> Bool) -> [a] -> [a]
insertionSort _ [] = []
insertionSort p (x:xs) = insert p x (insertionSort p xs)

insert :: (a -> a -> Bool) -> a -> [a] -> [a]
insert p x []     = [x]
insert p x (y:ys)
  | p x y  = (x:y:ys)
  | otherwise = y:(insert p x ys)
```

The Haskell implementation for insertion sort is generally the same as the provided pseudocode. However, the use of guards (the pipe "`|`" operator) allows for more concise `if/else` statements which reduce the number of function calls. For example, the insertion sort code itself is contained within three lines. Additional functionality is provided by the insert function, whose logic is contained within a mere 6 lines.

It is useful to note the use of a predicate function in the call to `insertionSort`. The use of the predicate allows the array to be sorted in different order, depending on the function. For example, passing a > symbol will yield a descending order sort whereas passing a < symbol yields an ascending order sort.

[3] *Introduction to Algorithms, 3rd edition,* pp 18
[4] InsertionSort code provided by
http://en.literateprograms.org/Special:DownloadCode/Insertion_sort_(Haskell)

Heapsort

CLRS pseudocode[5]:

```
heapsort(array A)
  build-max-heap(A)
  for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    max-heapify(A,1)
```

Haskell implementation[6]:

```
import Data.Graph.Inductive.Internal.Heap(
  Heap(..),insert,findMin,deteMin)

build :: (Ord a) => [(a,b)] -> Heap a b
build = foldr insert Empty

toList :: (Ord a) => Heap a b -> [(a,b)]
toList Empty = []
toList h     = x:toList r
  where (x,r) = (findMin h, deleteMin h)

heapsort :: (Ord a) => [a] -> [a]
heapsort = (map fst) . toList . build . map(\x->(x,x))
```

The Haskell implementation makes use of an additional structure provided by the fgl[7] package from HackageDB, a repository of additional packages for use with Haskell programs.

Implementation

The project is managed using git version control and hosted remotely on Github.com. The repository can be forked at the following link:

https://github.com/Amadeus98/cs430-project

The Haskell implementation of each sorting algorithm is written in the `sorting-implementation.hs` file.

[5] *Introduction to Algorithms, 3rd edition,* pp 160
[6] Heapsort code provided by http://rosettacode.org/wiki/Sorting_algorithms/Heapsort#Haskell
[7] fgl package located at http://hackage.haskell.org/package/fgl

Compiling

Before attempting to load the file, verify the Haskell compiler is installed by using the following command:

```
$> ghc --version
```

If an error appears, use the following link to install Haskell on your local system:

http://www.haskell.org/platform/

Once installed, the implementation can be loaded by changing into the directory holding the file and then using the following commands:

```
$> ghci
Prelude> :l sorting-implementation.hs
```

Each algorithm can then be run with the following commands. For simplicity's sake, each algorithm is run with a simple unordered list of integers, established in the first line of code

```
*Main> let unsorted = [3,6,9,1,3,-1,0,8]
*Main> quicksort unsorted
[-1,0,1,3,3,6,8,9]
*Main> insertionSort (<) unsorted
[-1,0,1,3,3,6,8,9]
*Main> heapsort unsorted
[-1,0,1,3,3,6,8,9]
```

Other data may appear on the screen when the file is first implemented. Unless an error appears, they can be ignored.

Testing

Testing the implementation of each sorting algorithm is fairly simple. A random list of integers can be generated by using the command:

```
*Main> take LENGTH (randomRs (LOWER_BOUND,UPPER_BOUND) g)
```

Where `LENGTH` is the length of the the random list, `LOWER_BOUND` is the lower bound of the numbers generated, `UPPER_BOUND` is the upper bound of numbers generated, and `g` is the generator, provided by the implementation, seeded with an integer of value of 5675309. The seed can be changed within the implementation if need be.

For example, in order to generate a list of 5 random integers between -10 and 10, one would use the command:

```
*Main> take 5 (randomRs (-10,10) g)
[10,3,-4,9,-7]
```

Of course, the output will change with each call. In order to suppress the output of a larger array, such as one of 1,000 values, the output of the previous command can be assigned to a variable as follows:

```
*Main> let test1 = take 1000 (randomRs (-20,20) g)
```

`test1` can then be used with one of the sorting implementations like so:

```
*Main> quicksort test1
```

Timing

Haskell provides a way of measuring the computation time for executed functions by using the following command in the GHCI interpreter:

```
*Main> :set +s
```

Running any command after the timing feature shows the computation time and the memory used. For example, the following comes from running on a Macbook Pro with a 2.3GHz Intel Core i5 processor and 8GB of 1333 MHz DDR3 RAM:

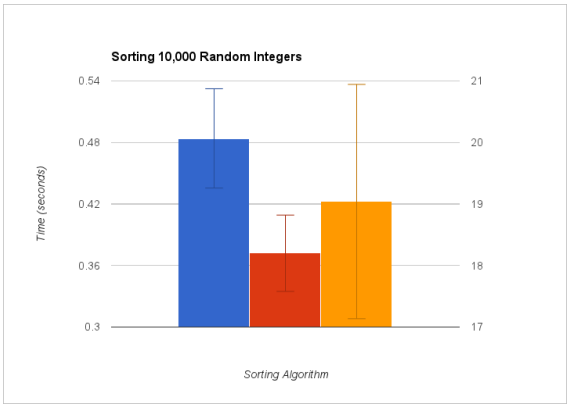
```
*Main> let test = take 5 (randomRs (-20,20) g)
(0.00 secs, 1108624 bytes)
*Main> quicksort test
[0,5,6,11,13]
(0.00 secs, 1145720 bytes)
```

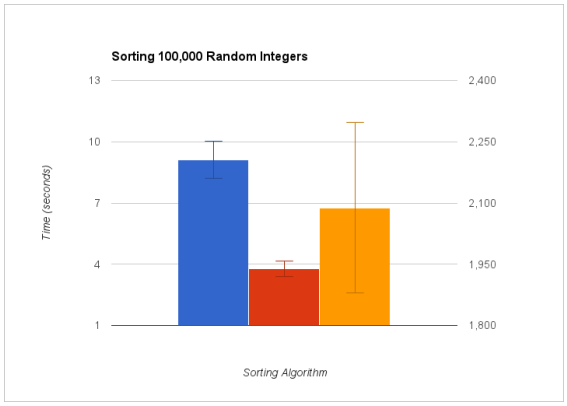
Although slightly inaccurate, the timing of each implementation will be recorded using the above method.

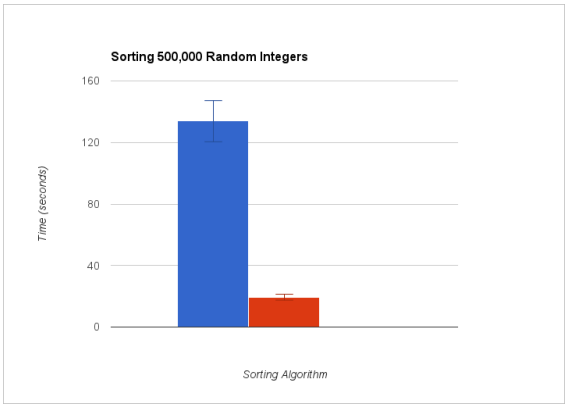
Results

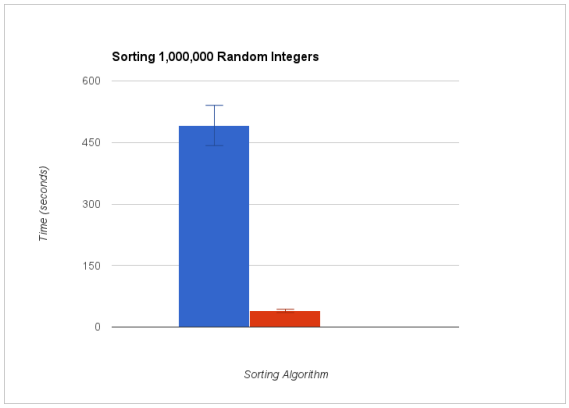
Each implementation was executed 5 times with 5 different list lengths to analyze the average time to completion. The full results can be viewed at the following link to a public Google Spreadsheet. Below are the summarized results.

http://goo.gl/voegf









Blue bars represent the average time for the quicksort algorithm, red for heapsort, and orange for insertion sort. Due to the large amount of time required by insertion sort for lists greater than 10,000 integers, no testing was performed. For the purposes of this analysis, it can be assumed insertion sort would have taken more than 2000 seconds.

Each sort was performed on the same machine as mentioned above - Macbook Pro with a 2.3GHz Intel Core i5 processor and 8GB of 1333 MHz DDR3 RAM. Although only essential processes were left running on the machine at the time of testing, recorded values may be erroneous due to other interfering processes hogging resources. Recorded values may also be erroneous due to the methods used by Haskell to record execution time. However, the recorded values are sufficient to analyze the relative performance of the implemented algorithms.

Analysis

The recorded data show a general trend of heapsort being the most efficient algorithm regardless of the list size. Most impressive is heapsort taking less than 40 seconds to sort 1,000,000 random integers.

According to the Wikipedia comparison of sorting algorithms[8], the results obtained in this analysis are not out of the ordinary. Heap sort is the most efficient algorithm due to it's $n \log n$ average time complexity. Quick sort is the second most efficient algorithm, operating with an average time complexity of $n \log n$ as well, except when presented with a worst-case scenario. Unfortunately, insertion sort is the least efficient algorithm due it's average n^2 time complexity.

Although an argument can be made for using more efficient code or testing environment, the general trend of the algorithms cannot be ignored. Regardless of the environment, or the method of implementation, these results can be replicated with a certain degree of accuracy.

Future experimentation should include the implementation of the mergesort and introsort algorithms for comparison. According to the Wikipedia comparison, the algorithms should perform with a similar efficiency as heapsort, but it would be interesting to analyze the data obtained in the results.

Additional work should also include comparisons of the algorithms across different implementations, such as with an object oriented language. Functional languages, such as Haskell used in this analysis, process data differently. It would be interesting to analyze the execution times of the algorithms in a language such as Ruby or Python.

[8] Wikipedia sorting algorithm comparison:
http://en.wikipedia.org/wiki/Sorting_algorithms#Comparison_of_algorithms