Abstract:
Our goal for this project was to learn more about the Haskell programming language by using it to implement various sorting algorithms. We implemented bubble sort, insertion sort, selection sort, and quicksort, and in the process integrated recursion and functional programming techniques into our solution. We then used regular expressions for I/O error handling and to delete unwanted characters from strings. These regular expressions check for the proper format of an email address, including the “@” and “.” symbols, correct domain name, and placement of letters and numbers within the local name. They also check for the correct format of the user’s name, including leading capital letters and lack of numbers. The purpose of the program is to take an email as input, remove punctuation and numbers, and sort the remaining letters to give the user a new name. While it is not particularly beneficial as an application, this program fulfilled its purpose in giving us a greater understanding of Haskell.

Key Words: sorting, Haskell, regular expressions, email

Introduction:
Haskell is a language that focuses the programmer’s efforts on describing computation rather than defining computation. An example of this would be creating the Fibonacci sequence in Haskell. Rather than defining a procedure to calculate a given number of Fibonacci numbers, you can create the sequence by describing the properties that a Fibonacci sequence has through use of list comprehension. This idea of description is further enforced by the use of functions as expressions rather than a series of steps. As a consequence, most of the programmer’s time is spent thinking out a problem rather than programming its solution. Inherent to Haskell is a focus on functional abstraction. Common procedures such as recursion through a list can be done with standard library functions like foldl. Functional abstraction allows the programmer to create programs that are concise and easy to maintain. Despite these advantages, Haskell has a bit of a learning curve as it forces the programmer to think in ways that are different than its imperative
counterparts. The purpose of the project was to overcome this learning curve and understand fundamental Haskell concepts by implementing well-understood sorting algorithms as well as regular expressions.

**Usage Examples:**

When `sortregex` is run, the user is first prompted to enter her name. If the input consists of only upper- or lowercase letters and spaces, it is accepted. The name is sorted, normalized and displayed. If the input contains any other characters, the program will display an error and exit.

The user is then prompted for her email address. This input is also checked for accuracy; if it contains letters, numbers and the @ and . symbols in the right order and with a valid email extension (com/net/org/edu), it is accepted. Otherwise, the program will display an error and exit.

The user must then choose a sorting method, numbered from 1-4. If the user enters a number other than 1, 2, 3, or 4, the program displays an error and exits. If the number is entered correctly, the program removes any numbers and the @ and . symbols from the email address and sorts only the letters to form a new name.

Example of the program running given correct input:
Examples of the program given incorrect input:

```
[rcrosby@student01 finalproj]$ ./sortregex
What's your name?
413x Kunk31
sortregex: That's not a name, jerk!
```

```
[rcrosby@student01 finalproj]$ ./sortregex
What's your name?
Alex Kunkel
Hey Akeekllnux, you suck!
We're going to give you a new name. Enter your email: santaclaws@northpolecats.govz
sortregex: That is not an email address, playa!
```

```
[rcrosby@student01 finalproj]$ ./sortregex
What's your name?
Alex Kunkel
Hey Akeekllnux, you suck!
We're going to give you a new name. Enter your email: santaclaws@northpole.gov
Now choose a sort method:
  (1) Bubble sort
  (2) Insertion sort
  (3) Selection sort
  (4) Quick sort*
  *(disclaimer: QUICKSORT is actually one word.)
99
sortregex: Pick one of the given sorting methods, please!
```

**Approach and Methods:**

**Bubble Sort:**

The bubble sort algorithm consists of doing swaps for every two consecutive elements in the list. This swapping occurs for multiple iterations until the algorithm meets the stopping criteria, which is no longer requiring any additional swaps. Bubble sort was implemented by abstracting out the swapping and stopping criteria into a helper function called swap. In order to encapsulate the stopping condition within the swap function, the swap function returns a tuple that contains both the modified list and the number of swaps that has occurred. Because the function is implemented recursively, the swap function must also accept the previously mentioned tuple as
input. The swap function relies on structural induction by generalizing the list into two elements and the rest of the list using pattern matching. Providing a base case and defining the conditions under which the first two elements are swapped enables the helper function to be applied upon the entire list and represent a single iteration. The main function decides whether to continue to iterate or return the current modified list by evaluating the tuple returned by swap.

**Insertion Sort:**
The insertion sort algorithm consists of building a list from a group of sortable data by choosing elements arbitrarily and placing them into the correct order in a new list. The insertion sort algorithm we implemented relies on a helper function that inserts an element in sorted order into an existing list. This function is recursively called on each element of the input until the input list is empty, sorting each value into the new list as it is removed from the old one. In our case, the “new list” is simply appended to the input list as elements are removed. When the base case of the empty list is called, the list, now sorted, is returned.

**Selection Sort:**
The selection sort algorithm consists of searching through a list of sortable elements, finding the minimum value among this list, and placing it at the beginning of the list. This process is repeated for the remainder of the list until there are no elements remaining. The Haskell language provides a method called “minimum” which takes a list as an argument and returns the smallest value in this list. The selection sort algorithm we implemented in our program uses this function to find the minimum value of the list, and then appends to this the resulting list obtained from recursively calling the selection sort function on the remainder of the list. The base case results when the selection function is called with an empty list given as the argument.

**Quicksort:**
The quicksort algorithm translates to yet another recursive function in Haskell. This algorithm involves picking a semi-random pivot value within a list to be sorted, finding all values smaller and greater than this value, and placing them on their respective sides of the pivot point. This method is recursively repeated on the “list” of values on either side of the pivot point until all values have been correctly sorted. In Haskell, this function is implemented by selecting the first
element of the given list as the pivot point, using the “filter” function to determine values that are
less than or greater than the pivot value, calling quick on these filtered lists, and appending all of
the values together in the proper order. Once again, the base case occurs when the “quick”
function is called with an empty list.

Regular Expressions:
Our program uses the Text.Regex package, a regular expressions library for Haskell, in order to
validate input and to remove characters from strings. From this package, we mainly used the
mkRegex function, which makes and compiles a regular expression with multi-line and case-
sensitive options enabled.

We defined functions called isLetter, isNumber and isEmail that implement mkRegex to check
if an input string satisfies our conditions. To check whether the name entered by the user is
legitimate, we used the regular expression ^[A-Za-z ]*$ to make sure that only letters and spaces
were allowed. Similarly, to check whether the number of the sorting algorithm picked by the user
is legitimate, we used the regular expression [0-9]+. To check for legitimate email addresses, we
originally tried to use ^[a-z]+[0-9]*@[a-z]+\.\.(com|org|net|edu|gov)\$. However, after a number of false positive results, we realized that parentheses did not need to be
escaped in Haskell. We also added the ability to have any combination of letters and numbers in
the email address. Therefore, we altered this initial RE to produce our final email verifier:
^([a-z]+([0-9]*[a-z]*)*@[a-z]+\.\.(com|org|net|edu|gov))$

To remove punctuation from the user’s email address, the program utilizes a function of the
Text.Regex package called subRegex that finds a regular expression and replaces it with another
regular expression in a given string. We replaced all occurrences of [0−9. @ ] (that is, any
numbers along with the “.”, “@” and space characters) with the empty string, effectively deleting
them from the email address.
Error handling and I/O:
In order to perform input and output within our program, we used the `getLine` and `putStrLn` functions, respectively. `putStrLn` is a function which takes a String as input and returns an IO object as output, and `getLine` is a function which does the opposite, retrieving input from the user in the process. As stated in the previous section, we use regular expressions on several occasions throughout the program to verify user input. Helper functions such as `isLetter`, `isNumber`, and `isEmail` are used to make sure the input is formatted correctly. We also use the built-in map function to map `toLowerCase` to the string so that format can be checked more efficiently, without having to check for uppercase letters. Output is also formatted using the “normalize” helper function. This function calls `toUpperCase` on the first letter in the sorted name, and maps `toLowerCase` to the remaining letters, so that the resulting name begins with an uppercase letter and is followed by only lowercase letters. If any of the specifications for input are not met, an error is printed to the screen using the “error” function, which causes the program to quit. This occurs in both the main function if input is entered incorrectly, and also within the “sort” method. This method sorts the characters in the String provided by the user, choosing a sort based on a number which is also provided by the user. If this number is not in the range requested by the program, an error will occur within this function.

Discussion and Future Work:

Given the time constraints for the project, we achieved the results that we desired. However, there are a number of improvements that could be made in terms of conciseness and efficiency. The first area of improvement would be in the space complexity of the algorithms. Due to the immutable default behavior of the variables, each iteration of the algorithms involves passing copies of lists. Using referential monads would allow lists to be modified in-place, greatly decreasing the space used in each algorithm. The next improvement that could have been made is to parallelize our quicksort function. Given the ubiquity of parallel programming, having experience with parallelizing code would both increase the efficiency of the algorithm as well as provide necessary experience that will be useful in future programming endeavors. Implementing additional algorithms such as heap sort and radix sort would provide greater experience with Haskell as well as reinforcing our understanding of the various sorting algorithms from the Data
Structures course. The final thing that could be done to improve our program is greater use of composition and higher level functions like \texttt{foldl} and \texttt{filter} in order to make our code more concise and simplify logic.

Authors:

Antwane Mason is a junior computer science and Japanese dual degree student. His hobbies include playing around with different Linux distributions, reading technical books, and playing video games. His favorite pastry is the chocolate chip cookie. During any given birthday, what he desires most is delicious, succulent crab legs. After graduation, he wants to go to graduate school and perhaps concentrate on something in the field of networking or operating systems.

Tricia Landers is a junior computer science and Japanese double major who enjoys playing video games and reading in her spare time. Unfortunately, having a dual major ensures that her time is anything but spare. Upon graduating she hopes to travel to Japan and find a job at a reputable gaming company such as Square Enix. Cats are her favorite animal, and she would go to drastic measures for a Papa John’s pizza.

Ryan Crosby is a junior computer science major. When not studying, he enjoys activities such as running, biking, skateboarding and playing pool. Sometimes he’ll even write small applications in Python for fun. He loves animals, and feeding squirrels is a favorite pastime of his. When he graduates, he hopes to find a job in web development.
References:

Hackage: Text.RegEx package; regular expression matching.

HaskellWiki
http://www.haskell.org/haskellwiki/Introduction_to_Haskell_IO/Actions