Functions, Debugging, Etc.

```
library(RCurl)
library(XML)
library(ggplot2)
library(stringr)
source("HW5_Functions.R")
```

1.) Scraping from W3 for UTF-8

I read in the table by web-scraping. It seems that the req. headers that are necessary for an HTTP request followed the format as Stack Overflow.

```
w3 = "https://www.w3schools.com/tags/ref_urlencode.ASP"
cookies = "_gid=GA1.2.958916006.1686004611; _ga_9YNMTB56NB=GS1.1.1686004610.2.1.1686004621.49.0.0; _ga=
useragent = "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chr


doc = getURLContent(w3, .opts = list(
  followlocation = TRUE,
  verbose = FALSE,
  cookie = cookies,
  httpheader = c(Accept = "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,
))

doc2 = htmlParse(doc)
```

```
# First column: char
char = xmlValue(getNodeSet(doc2, "//table[@class = 'ws-table-all notranslate'][1]/tr/*[self::td][1]"))

# Last Column: UTF-8
utf = xmlValue(getNodeSet(doc2, "//table[@class = 'ws-table-all notranslate'][1]/tr/*[self::td][3]"))

# Table
w3table = data.frame(char, utf)
head(w3table)
```

```
##    char utf
## 1 space %20
## 2     ! %21
## 3     " %22
## 4     # %23
## 5     $ %24
## 6     % %25
```

```
tail(w3table)
```

```
##     char    utf
## 219    ú %C3%BA
## 220    û %C3%BB
## 221    ü %C3%BC
## 222    ý %C3%BD
## 223    þ %C3%BE
## 224    ÿ %C3%BF
```

2.) Create a collection of sample input strings of different sizes containing regular chars and percent encoded content

I sampled all of the %encoded content, upper and lowercase letters, and digits. The different input sizes I had ranged from 100-200000, however the actual size of the string (nchar) was different as sampling one instance of %encoded content = 3 characters.

```
# Sampling content (Chars and %encodings)
sample_chars = c(letters, LETTERS, 0:9, utf)

# Test different input string lengths
input_lengths <- c(100, 1000, 10000, 100000,200000)
```

```
# Generate a random input string of the given length
sample_inputs <- lapply(input_lengths, function(x) paste0(sample(sample_chars, x, replace = TRUE), colla
sample_inputs[[1]]
```

```
## [1] "%C2%A0%C2%AAs%3D%E2%80%9Ey%C3%80%4B%61iR%E2%80%93%64p%6D%2D%32%5Fd%C5%BE%53%7A%55%C2%AD%C3%B6%C3
```

```
nchar(sample_inputs[[1]])
```

```
## [1] 387
```

I showed an example of one an input above.

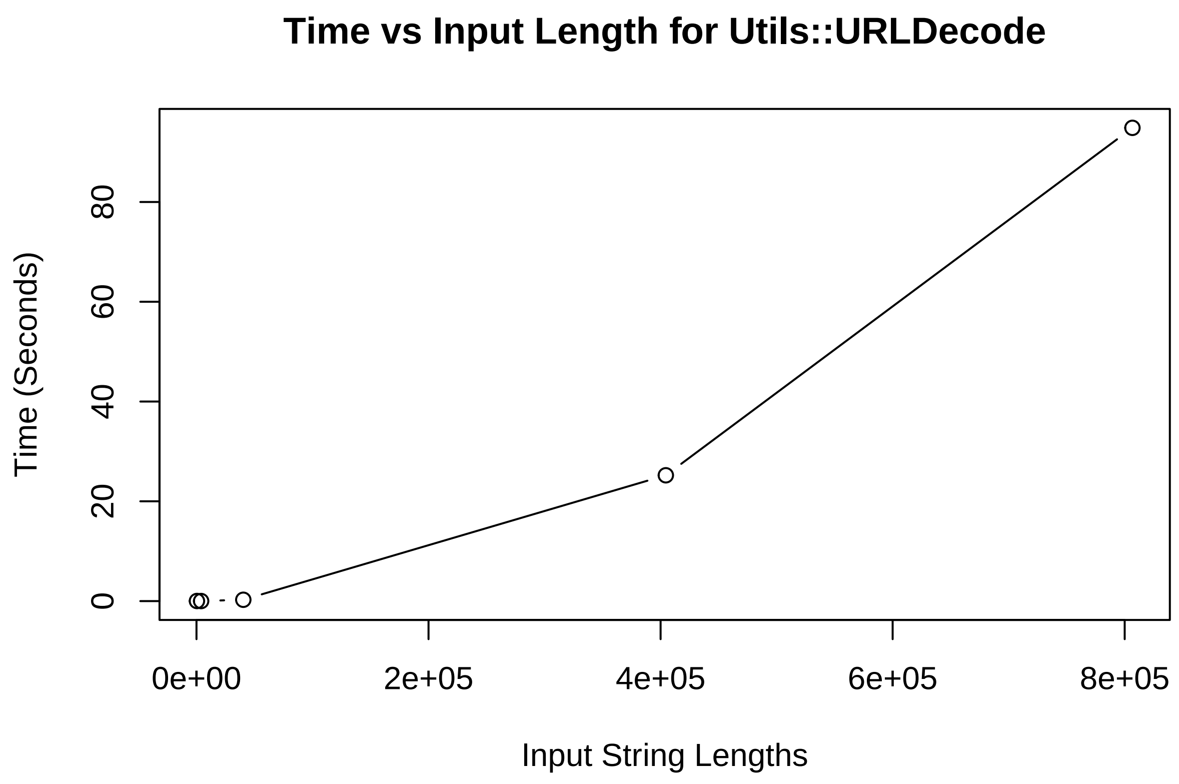
```
# Number of characters + % encoding per input
sample_lengths = lapply(sample_inputs, nchar)
sample_lengths
```

```
## [[1]]
## [1] 387
##
## [[2]]
## [1] 4012
##
## [[3]]
## [1] 40365
##
## [[4]]
## [1] 404520
##
## [[5]]
## [1] 806668
```

3.) Run-time Performance for Utils::URLDecode

If we are focusing on one input string, then just by inspecting the code I would predict that the run-time performance would be O(n). The linear time is exhibited by the while loop, which iterates through every character of the URL one at a time. This means the output time is proportional to the size of the input, which implies that a large input string would take a longer time to output the result.

```r
# Run times for above inputs
estimate = lapply(sample_inputs, function(x) system.time(original_decode(x)))

times = c()
for (i in 1:length(estimate)){
  times = c(times, estimate[[i]]["elapsed"])
}

plot(unlist(sample_lengths), times, type = "b", xlab = "Input String Lengths",
     ylab = "Time (Seconds)", main = "Time vs Input Length for Utils::URLDecode")
```



After plotting it however, I observe that utils::URLdecode has more of a quadratic fit. Based on my plot, if I were to estimate how long it would take the read in a .txt file with 591k characters, it would take around 40 seconds. That does not seem right and is definitely was too fast. This is pretty suspicious and could be looked into more.

4.) Modifying Original

The implementation of the preallocated decode is in my R script. Here is my breakdown of how I changed the function:

I only changed three things. I changed the variable out to have the length of the input URL. This is for preallocation purposes. Next, I changed the concatenation of out ("c(, out))" into that of out[j] = x[i]. Finally, I iterated for j once for each statement. The run-time improvement was pretty noticeable, and to verify that my output of out was the same as utils::URLdecode, I pasted together a sample URL and checked if out had the same length.

```
sample_url = paste0(sample(sample_chars, 1000, replace = TRUE), collapse = "")
original_decode(sample_url) == preallocated_decode(sample_url)
```

```
## [1] TRUE
```

5.) Verify for Multiple Inputs

My strategy for verifying that the preallocated function gets the right answer was to compare both functions on varying input lengths, and using stopifnot() and identical() to verify for similarity. To do this, I applied both functions to multiple inputs using lapply.

```
test1 = lapply(sample_inputs, original_decode)
test2 = lapply(sample_inputs, preallocated_decode)
```

If test1 and test2 were not identical, then the statement below would stop and produce an error. If nothing was displayed, then the preallocated function was correct.

```
# See if values in both lists are TRUE
stopifnot(identical(test1, test2))
```

There's no output, therefore we verified that the function works properly.

6.) Implementation of a New URLDecode w/ Vectorization

Thought Process: Remove the while() loop

I found the indexes in the URL such that a character matched the % character. For those characters that did, I grabbed the indexes and extracted the two characters following after every %, and placed them into a matrix with two rows. This meant that every character pair for each % was in one column. Next, I applied colSums() as a replacement for sum() in the utils::URLdecode due to the syntax of the matrix. This gave me the character I needed back, and I placed them in the indexes corresponding to where I found the % characters.

For the characters that did not equal to %, I did a gsub of every %encoded character (i.e %20, %21) and replaced them with 3 blank spaces. This gave me a vector that had the length of the original URL, however had 00 whenever there was a %encoding. This allowed me to get the exact values of the characters that did not follow the %encoding syntax. After I found this, I put these values into out in their respective positions.

7.) Bugs and Debugging

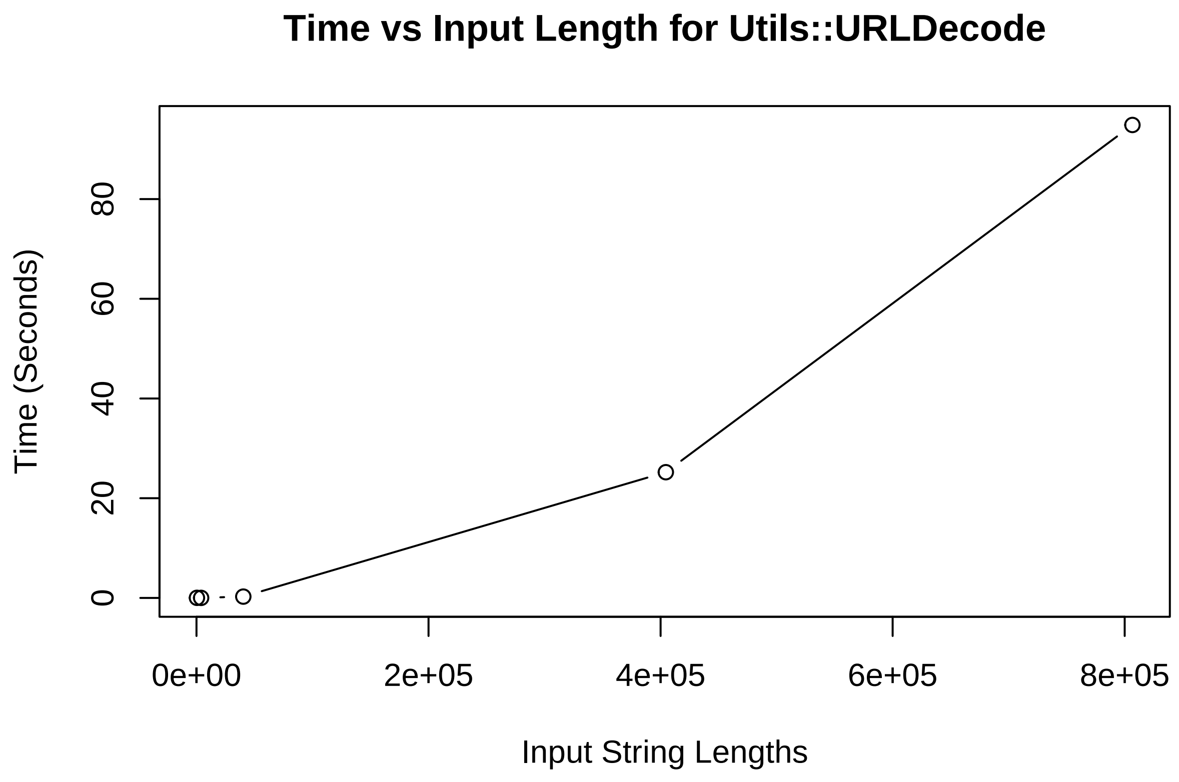
I debugged a lot to examine the behavior of the functions. I did not run into trouble with the preallocation whatsoever. The only concern is that it executes extremely fast, and I'm not sure if it is supposed to be like that. I understand it totally depends on specs, memory, and other things and I double checked with Xiner, and yeah there was nothing obviously incorrect with my code.

I did run into the peak issue with vectorization. There was once instance where I tested the system.time() on both functions with my sample URL I created in Q4.), and the vectorization function produced a longer time at a relative shorter input length. For example, I think it gave me .023 seconds for about 4k characters, however the plot reached this same height later on with greater input lengths as well. The problem did seems to dissolve on its own when I ran the function again, however I'm not sure what the root problem is.

8.) Run-time Curve for 3 functions

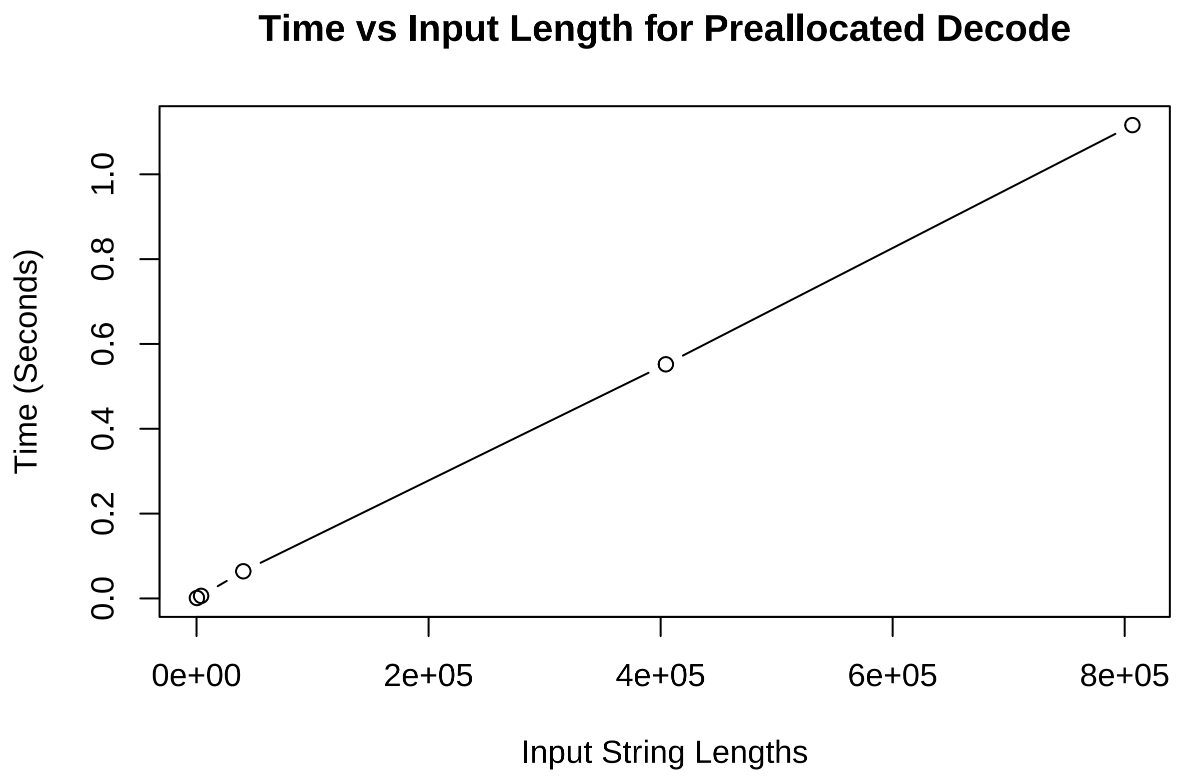
Utils

```r
plot(unlist(sample_lengths), times, type = "b", xlab = "Input String Lengths",
     ylab = "Time (Seconds)", main = "Time vs Input Length for Utils::URLDecode")
```



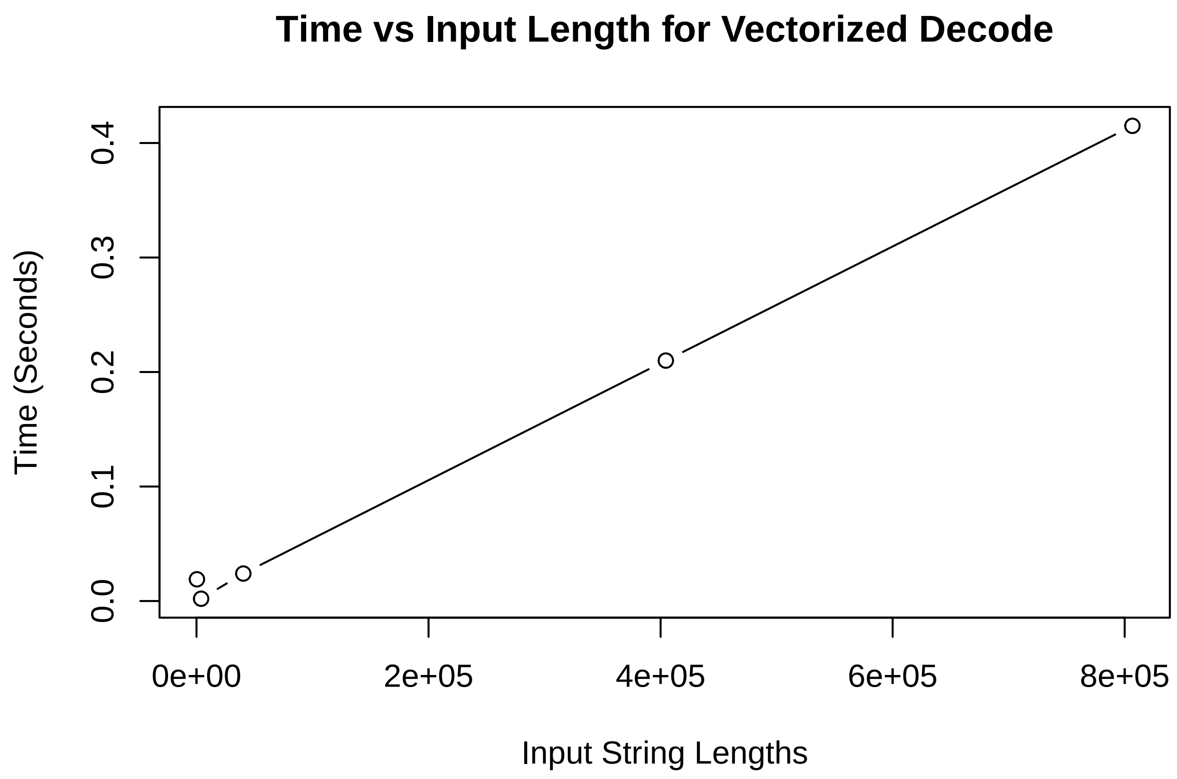
Preallocated

```r
preallocated_estimate = lapply(sample_inputs, function(x) system.time(preallocated_decode(x)))

preallocated_times = c()
for (i in 1:length(preallocated_estimate)){
  preallocated_times = c(preallocated_times, preallocated_estimate[[i]]["elapsed"])
}

plot(unlist(sample_lengths), preallocated_times, type = "b", xlab = "Input String Lengths",
     ylab = "Time (Seconds)", main = "Time vs Input Length for Preallocated Decode")
```



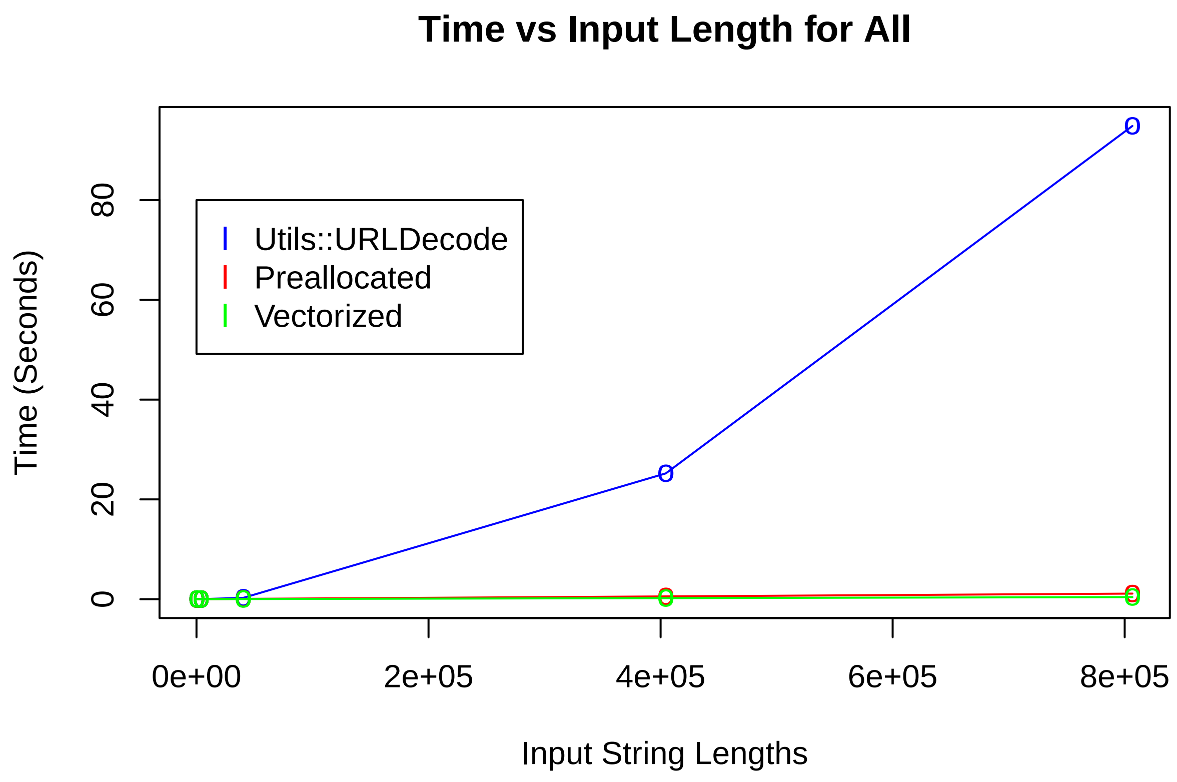
Vectorization

```r
vectorized_estimate = lapply(sample_inputs, function(x) system.time(vectorization_decode(x)))

vectorized_times = c()
for (i in 1:length(vectorized_estimate)){
  vectorized_times = c(vectorized_times, vectorized_estimate[[i]]["elapsed"])
}

plot(unlist(sample_lengths), vectorized_times, type = "b", xlab = "Input String Lengths",
     ylab = "Time (Seconds)", main = "Time vs Input Length for Vectorized Decode")
```



All function Run-Times

```r
# Original
plot(unlist(sample_lengths), times, type = "o", col = "blue", pch = "o", xlab = "Input String Lengths",
     ylab = "Time (Seconds)", main = "Time vs Input Length for All")
# Preallocated
points(unlist(sample_lengths), preallocated_times, col="red", pch="o")
lines(unlist(sample_lengths), preallocated_times, col="red",lty=1)
# Vectorization
points(unlist(sample_lengths), vectorized_times, col="green", pch="o")
lines(unlist(sample_lengths), vectorized_times, col="green",lty=1)
legend(1, 80, legend = c("Utils::URLDecode","Preallocated", "Vectorized"), col = c("blue","red","green"
       pch = c("l","l","l"))
```



It is clear that preallocated and vectorized both outperform the original decode() tremendously in runtime. While vectorization is the fastest, preallocation is a close second.

9.) Testing of 591k file

```
txt = readLines("PercentEncodedString.txt")
```

```
## Warning in readLines("PercentEncodedString.txt"): incomplete final line found on
## 'PercentEncodedString.txt'
```

```
# Verification
nchar(txt)
```

```
## [1] 591977
```

Verifying the txt file was read in correctly. (591k chars)

```
# Testing
system.time(original_decode(txt))
```

```
##    user  system elapsed
## 268.957  43.768 318.449
```

```r
system.time(preallocated_decode(txt))
```

```
##    user  system elapsed 
##   0.346   0.002   0.349
```

```r
system.time(vectorization_decode(txt))
```

```
##    user  system elapsed 
##   0.045   0.006   0.051
```

The estimate of utils::URLdecode was off. Again, it is strange, and I'm not sure what the problem is. It probably has to do something with my character string and the variation of certain characters? Vectorization is the fastest.

```r
# Is the Original = Preallocation?
preallocated_decode(txt) == vectorization_decode(txt)
```

```
## [1] TRUE
```

```r
# Testing for preallocation and vectorization
stopifnot(identical(lapply(sample_inputs, vectorization_decode), lapply(sample_inputs, preallocated_dec
```