Project5

Vincent Barletta

2023-06-13

Background

Consider the function utils::URLdecode(), i.e., in R's own utils package. It converts/decodes percent-encoded strings such as %24abc%5D%2B to $abc]+ where, e.g., the was represented by %24. This is the opposite of what we have to do to escape/encode characters in a URL for an HTTP request, e.g., map each space to %20.

The utils::URLdecode() function is designed to work with URLs which are limited to 2048 characters. These are short and the functions works fine with these. However, it can also be useful when decoding content in the body of a POST request in an HTTP operation. See calcareers.ca.gov, specifically https://www.calcareers.ca.gov/CalHRPublic/Search/AdvancedJobSearch.aspx

The body of that request is URL-encoded, starting with ctl00%24ToolkitScriptManager1=ctl00%24cphMaiThe full string has 591,977 characters. When we pass this in a call to utils::URLdecode(), it takes about 16 minutes on my machine! That's a problem. We need a faster version for these larger inputs.

Step 1: Reading in the UTF-8 Table

Our first step was to read in the UTF-8 percent-encoding table from W3 schools. I initially thought something was wrong with my output, as the console outputted every special character in the format "<U+00A0>". Once I went through attempts to convert between encodings, I realized that they were read in correctly but displayed wrong in the console.

```
u = "https://www.w3schools.com/tags/ref_urlencode.ASP"
tt = suppressWarnings(readLines(u, encoding = "UTF-8"))

doc = htmlParse(tt, encoding = "UTF-8")
tbl = readHTMLTable(doc, encoding = "UTF-8")[1]
tbl_df = as.data.frame(tbl)
tbl_df = tbl_df[-2]
tbl_df[1,1] = " "
colnames(tbl_df) = c("Character", "UTF8")
tbl_df$Character = iconv(tbl_df$Character, from = "UTF-8", to = "UTF-8")
```

Once, I viewed the df, however, I realized that the characters were read in correctly. This photos below are what it originally looked like. Weirdly enough, once I opened up my R session on the second day, it started fully displaying everything correctly in the console.

	Character	From UTF-8
100	ƒ	%C6%92
101	„	%E2%80%9E
102	…	%E2%80%A6
103	†	%E2%80%A0
104	‡	%E2%80%A1
105	ˆ	%CB%86
106	‰	%E2%80%B0
107	Š	%C5%A0
108	‹	%E2%80%B9
109	Œ	%C5%92
110	□	%C5%8D
111	Ž	%C5%BD
112	□	%8F
113	□	%C2%90
114	‘	%E2%80%98
115	’	%E2%80%99
116	“	%E2%80%9C
117	”	%E2%80%9D
118	•	%E2%80%A2
119	–	%E2%80%93

vs.

96	<U+00A0>	%7F
97	<U+20AC>	%E2%82%AC
98	<U+0081>	%81
99	<U+201A>	%E2%80%9A
100	<U+0192>	%C6%92
101	<U+201E>	%E2%80%9E
102	<U+2026>	%E2%80%A6
103	<U+2020>	%E2%80%A0
104	<U+2021>	%E2%80%A1
105	<U+02C6>	%CB%86
106	<U+2030>	%E2%80%B0
107	<U+0160>	%C5%A0
108	<U+2039>	%E2%80%B9
109	<U+0152>	%C5%92
110	<U+008D>	%C5%8D
111	<U+017D>	%C5%BD
112	<U+008F>	%8F
113	<U+0090>	%C2%90
114	<U+2018>	%E2%80%98
115	<U+2019>	%E2%80%99
116	<U+201C>	%E2%80%9C

Step 2: Sample Input Strings

We will now create some sample strings. First, we will just use some gibberish and pass it into URL encode (1 and 2). Next, we will use various UTF-8 character combinations. Lastly, we will combine gibberish and special characters to make a very long string for testing purposes in step 3. Note: We will only display the strings that LaTeX can partially render out

```
test = URLencode("Test string!")
yabba = URLencode("Yabba dabba doooooooooooooooooooooooooooooooooooooooooooooooooooooooooo.!PLOISJEU
                  asduohqwoiejqweasldnalsjdnasd=====098776128371629312 3johasdjasdhkjnxzcknzxckjahsduh
                  asdhqueioiojwqiojYUPYUPYUPYUPYUPYUPYUPYUPYUPYUPYUP")

inputstring = readLines("inputstring.txt")

inputstring_enc = URLencode(inputstring)

sample_strings <- c(test, yabba, "%C2%A9%20%20%F0%9F%94%A5%20%CE%9B%CE%8F%CE%93%CE%9F%CE
                      %A3",
    "%E2%82%AC%20%26%20%D7%AA%D7%97%D7%95%D7%9E%D7%99%D7%9D%20%26%20%E6%96%87%E5%AD%97",
    "%F0%9F%8D%95%20%CE%94%CE%99%CE%91%CE%A4%CE%A1%CE%9F%CE%A6%CE%97%20%F0%9F%8D%8C",
    "%E2%98%80%20%CE%A3%CE%AE%CE%BC%CE%B5%CF%81%CE%B1%20%E2%98%81%20%D8%A7%D9%84%D8%B3%
```

```
    D9%84%D8%A7%D9%85",
    "%E2%9C%A8%20%D0%9C%D0%B0%D0%B3%D0%B8%D1%8F%20%E2%9C%A8%20%F0%9F%8C%9F",
    "%E2%99%9B%20%E0%B9%80%E0%B8%A5%E0%B8%AA%E0%B8%95%E0%B8%A3%E0%B8%B5%E0%B8%A2%E0%B8
    %A1%20%E2%99%9B",
    "%F0%9F%8E%BC%20%E4%BD%A0%E5%A5%BD%20%CE%93%CE%B5%CE%B9%CE%B1%20%F0%9F%8E%BC",
    inputstring_enc
    )
URLdecode(sample_strings[c(3,5)])
```

```
## [1] "©    ΛΩΓΟ\xce\n                    \xa3"
## [2] "  ΔΙΑΤΡΟΦΗ  "
```

Step 3 Estimate run-time performance of utils::URLdecode()

The utils::URLdecode function processes our sample strings very quickly. Our final string is 6320 characters and loads almost instantly. I believe that it is because most of it is normal text and therefore does not need to be decoded.

```
nchar(sample_strings[10])
```

```
## [1] 6296
```

Given that it is able to process this entire 6200 char string in one second, I believe that for our example URL string, it will be able to parse it at a rate of 4000 characters per second.

Step 4 Preallocated Decoding Function

The main reason why the default utils::URLdecode() function is so slow is that it directly checks each character, one at a time, to see if there is a percent. To make this process much faster, we can instead check initially for the indices of each occurrence of a "%" in our URL. With these values in mind, we can then change each bit directly.

Our first change is to initialize "out" as a vector of length(x). Rather than concatenating each character to the end of our output, we insert it into the vector. This is computationally faster than concatenating each element one-by-one, and it also allows us to handle the three special character bits easier.

We now change our conditional statement structure to check if our index i is in the list of percentage indices, pc_indices. If it is, we do the same text processing procedure as in the original method.

Something important to note is that because we have to iterate by three characters to get to the next character, it adds null values "/0" for (i+1) and (i+2). Lastly, in order to remove these values, we select the subset of our output where it does not equal 0.

```
preallocated_URLdecode <- function(URL) {
  vapply(URL, function(URL) {
    x <- charToRaw(URL)
    pc <- charToRaw("%")
    out <- raw(length(x))  # Preallocate the 'out' vector
    i <- 1L
    pc_indices <- which(x == pc)  # Find the indices of '%' in 'x' at the start
    c_calc = c(16L, 1L)
```

```
    while (i <= length(x)) {
      if (i %in% pc_indices) {  # Check if current index is a '%'
        y <- as.integer(x[(i + 1L):(i + 2L)])  # Extract the next two elements
        y[y > 96L] <- y[y > 96L] - 32L
        y[y > 57L] <- y[y > 57L] - 7L
        y <- sum((y - 48L) * c_calc)
        out[i] <- as.raw(as.character(y))  # Insert values into 'out'
        i <- i + 3L
      } else {
        out[i] <- x[i]  # Insert value into 'out'
        i <- i + 1L
      }
    }

    rawToChar(out[which(out != 0)])
  }, character(1), USE.NAMES = FALSE)
}
```

Step 5: Verify that our function gets the correct answers

As our ultimate step is to time how quickly each of the three decoding methods can process the 600k character text file, my first idea for testing was to simply try it on a subset of that file. The first objective is to check the class and nchar of each amount. The next test is to simply check that their outputs are equal.

```
txtsubset = "ctl00%24ToolkitScriptManager1=ctl00%24cphMainContent%24ctl00%7Cctl00%24cphMainContent%24bt
largetext = readLines("PercentEncodedString.txt")
```

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

```
default1 = URLdecode(txtsubset)
class(default1)
```

```
## [1] "character"
```

```
nchar(default1)
```

```
## [1] 892
```

```
pre1 = preallocated_URLdecode(txtsubset)
class(pre1)
```

```
## [1] "character"
```

```
nchar(pre1)
```

```
## [1] 892
```

```
default1 == pre1
```

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

We will try this for some of our sample strings as well.

```
default2 = URLdecode(sample_strings[1])
pre2 = preallocated_URLdecode(sample_strings[1])
print(default2 == pre2)
```

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

```
default3 = URLdecode(sample_strings[2])
pre3 = preallocated_URLdecode(sample_strings[2])
print(default3 == pre3)
```

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

```
default4 = URLdecode(sample_strings[3])
pre4 = preallocated_URLdecode(sample_strings[3])
print(default4 == pre4)
```

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

We find that utils::decode can decode 2271.07 characters per second. Therefore, for a 100 character input, it can decode it in 0.0440 seconds. The graphic of the default processing rate will be displayed with the vectorized and preallocated functions down below.

Step 6: Vectorization

After much trial and error, I eventually found an incredibly efficient and very basic solution for our vectorized approach. It is fundamentally very, very similar to both the base utils::URLdecode and our preallocated approach, but it is much faster than both of them as it does not utilize while/for loops or apply functions.

The process is very similar to our preallocated solution. First, find where the "%" are in x. Rather than looping through and processing each individual special character value, we read in the entire vectors of digit1 and digit2. We apply the same conversion technique as provided in the initial function and apply it to the separate digits. Separating the two out made it conceptually easier for me to understand the process.

Lastly, we simply insert those values into their specified positions in out, and fill the rest of our output with the unchanged data. We also have to remove the two characters after each pc_indice after they have been properly translated. This approach is incredibly fast, and shockingly easy (once I figured it out).

```
vec_decoder <- function(URL) {
  x <- charToRaw(URL)
  pc_indices <- which(x == charToRaw("%"))  # Find the indices of '%' in 'x'

  # Preallocate the 'out' vector with the same length as 'x'
  out <- raw(length(x))
```

```
  # Convert the two-digit hexadecimal ASCII codes to decimal and decode
  y1 <- as.integer(x[pc_indices + 1])
  y2 <- as.integer(x[pc_indices + 2])

  y1[y1 > 96L] <- y1[y1 > 96L] - 32L
  y2[y2 > 96L] <- y2[y2 > 96L] - 32L

  y1[y1 > 57L] <- y1[y1 > 57L] - 7L
  y2[y2 > 57L] <- y2[y2 > 57L] - 7L

  y <- (y1 - 48L) * 16L + (y2 - 48L)

  # Insert the decoded values into the 'out' vector
  out[pc_indices] <- as.raw(y)

  #seq_along provides each index in x; setdiff removes pc_indices from this sequence
  #we copy all of the original untouched data into out
  non_pc_indices <- setdiff(seq_along(x), pc_indices)
  out[non_pc_indices] <- x[non_pc_indices]

  #Remove the trailing two characters from special characters
  out = out[-c((pc_indices+1),(pc_indices+2))]

  rawToChar(out)
}
```

We will now do a rudimentary test to check if it is equivalent to the base function. As our preallocated function matched for all test cases, we will test our vectorized function sample outputs against the preallocated sample outputs.

```
vec1 = vec_decoder(txtsubset)
length(vec1) == length(pre1)
```

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

```
vec1 == pre1
```

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

```
vec2 = vec_decoder(sample_strings[1])
print(vec2 == pre2)
```

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

```
vec3 = vec_decoder(sample_strings[2])
print(vec3 == pre3)
```

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

```
vec4 = vec_decoder(sample_strings[3])
print(vec4 == pre4)
```

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

Step 7: Describe any bugs you had in your preallocation and vectorized functions

Preallocation Decoding Function:

My function utilizes a lot of similar logic from the original URLdecode function, so it was not too messy to make changes initially. The problem was finding a way to avoid concatenation while correctly reading in the string. My main problem was after reading in the two characters following a % character, I needed to skip those two characters to get to the next non-special character. Moving our iterator forward however coerced nulls into the (i + 1) and (i + 2) spots. To combat this, I tried to fill those two spots with characters to replace after the fact. However, this still was ineffective. Instead, I tried multiple ways to get rid of the "/0" null characters before eventually finding that I could simply subset for where it did not equal 0.

Vectorized Decoding Function:

I ran in to many, many issues when trying to create this function. I spent multiple hours trying out many different approaches to try and process all of the different special characters in our dataset without being able to loop through each element. My first idea was to create a percent_count (based on how many percent characters were left in the array) and iterate through my special characters by decreasing the count by 1 after each translation. Then, after rechecking the prompt, I tried to instead use my tbl_df to try and lookup for a match and convert the characters using that logic. I created separate functions for that (posted on Piazza), but a issue I ran into across most of my attempts was that my program would only use the first element in a set and apply its conversion to every other special character, rather than processing each element and applying its individual conversion. The only way I could get it to work was by usign sapply(), which took very long to process the 591k line textset. Finally, I returned back to the most basic function and bewildering made it work.

Step 8: Run-time curves

In order to collect data to build the curves, we will collect for 10 samples in the form of snippets of our 591K file. We will start with 1k lines, 5k lines, 10k lines, 50k lines, 100k lines, 150k lines, 200k lines, 400k, 600k, 1M lines. We will then time each of our functions across the 10 datapoints to see how they compare before finally graphing all of them on one chart.

In this chunk, we simply take substrings of our overall text. For the last point, we paste it together with itself and take a substring to reduce it to 1,000,000 characters.

```
data = length(10)
data[1] = substr(largetext, 1, 1000)
data[2] = substr(largetext, 1, 5000)
data[3] = substr(largetext, 1, 10000)
data[4] = substr(largetext, 1, 50000)
data[5] = substr(largetext, 1, 100000)
data[6] = substr(largetext, 1, 150000)
data[7] = substr(largetext, 1, 200000)
data[8] = substr(largetext, 1, 400000)
data[9] = substr(largetext, 1, 591000)
data10 = paste0(largetext, largetext, collapse = "")
data[10] = substr(data10, 1, 1000000)
```

We run the same for loop here on all three of our functions to collect all the data we need for our graph. Admittedly, this could have been made into a function and simply had the function name passed in as a parameter, but I did it the lazy way instead (its been a long year).

```
results <- data.frame(Function = character(), Runtime = numeric(), stringsAsFactors = FALSE)
  for (i in 1:length(data)) {
      string <- data[i]
      start_time <- Sys.time()
      # Process the string using the specified function
      # Replace the function call below with your actual function call
      processed_string <- vec_decoder(string)
      end_time <- Sys.time()

      runtime <- as.numeric(end_time - start_time)

      # Store the result in the dataframe
      result <- data.frame(Function = "Vectorized", Runtime = runtime)
      results <- rbind(results, result)
  }
```

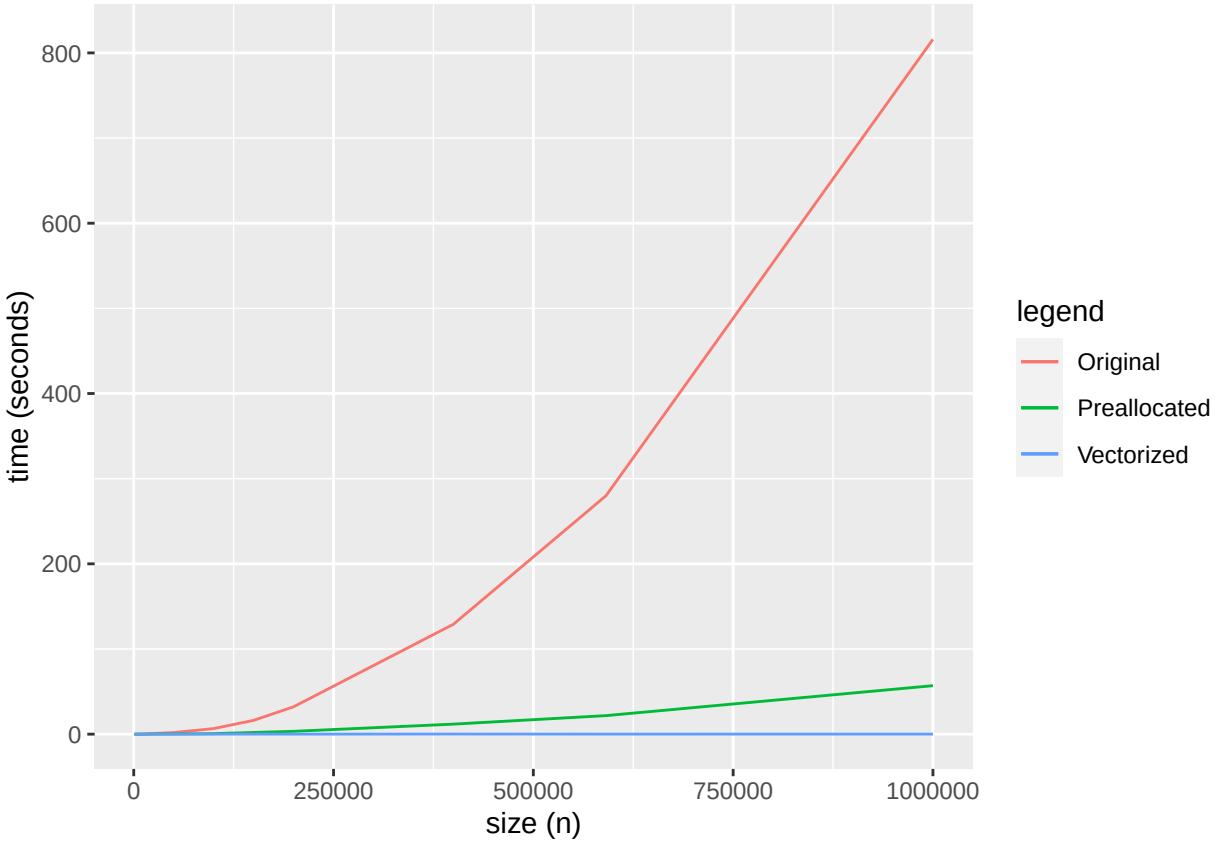
```
## Warning in vec_decoder(string): out-of-range values treated as 0 in coercion to
## raw
```

```
vectorized_df = results
vectorized_df = vectorized_df[-1]
ncharc = c(1000,5000,10000,50000,100000,150000,200000,400000,591000, 1000000)



# results <- data.frame(Function = character(), Runtime = numeric(), stringsAsFactors = FALSE)
#   for (i in 1:length(data)) {
#       string <- data[i]
#       start_time <- Sys.time()
#       # Process the string using the specified function
#       # Replace the function call below with your actual function call
#       processed_string <- preallocated_URLdecode(string)
#       end_time <- Sys.time()
#
#       runtime <- as.numeric(end_time - start_time)
#
#       # Store the result in the dataframe
#       result <- data.frame(Function = "Preallocated", Runtime = runtime)
#       results <- rbind(results, result)
#   }

#preallocated_df = results
#save(preallocated_df, file = "preallocated.RData")
load("~/STA 141B/Project 5/preallocated.RData")
preallocated_df = preallocated_df[-1]

# results <- data.frame(Function = character(), Runtime = numeric(), stringsAsFactors = FALSE)
#   for (i in 1:length(data)) {
#       string <- data[i]
```

```
#       start_time <- Sys.time()
#       # Process the string using the specified function
#       # Replace the function call below with your actual function call
#       processed_string <- URLdecode(string)
#       end_time <- Sys.time()
#
#       runtime <- as.numeric(end_time - start_time)
#
#       # Store the result in the dataframe
#       result <- data.frame(Function = "Original", Runtime = runtime)
#       results <- rbind(results, result)
#   }

#original_df = results
#original_df = original_df[-1]
#original_df[8,1] = original_df[8,1] * 60
#original_df[9,1] = original_df[9,1] * 60
#original_df[10,1] = original_df[10,1] * 60

#save(original_df, file = "originaldf.RData")

load("~/STA 141B/Project 5/originaldf.RData")

tm_df <- cbind(ncharc, original_df, preallocated_df, vectorized_df)
colnames(tm_df) <- c("nchar", "Original", "Preallocated", "Vectorized")
tm_df
```

```
##      nchar     Original Preallocated   Vectorized
## 1     1000 1.391172e-03  0.001667023 0.0001120567
## 2     5000 2.800918e-02  0.021188021 0.0002400875
## 3    10000 9.001613e-02  0.024747849 0.0003559589
## 4    50000 1.861651e+00  0.251849890 0.0095617771
## 5   100000 6.559324e+00  0.613871813 0.0061600208
## 6   150000 1.623506e+01  1.974682808 0.0245840549
## 7   200000 3.212744e+01  3.315594196 0.0207340717
## 8   400000 1.290622e+02 11.781260014 0.1297628880
## 9   591000 2.800473e+02 21.767376900 0.0406200886
## 10 1000000 8.159062e+02 56.936149836 0.0671980381
```

```
ggplot(tm_df, aes(x=`nchar`)) +
            geom_line(aes(y=`Original`, color="Original")) +
            geom_line(aes(y=`Preallocated`, color="Preallocated")) +
            geom_line(aes(y=`Vectorized`, color="Vectorized")) +
          labs(x="size (n)", y="time (seconds)", color="legend")
```



Finally, we can see how the functions escalate over character size increase. The vectorized function does not change very much at all. The preallocated function begins to slow down as nchar approaches 1M. The original function struggles greatly once n = 500k, and takes up to 13 minutes to process a 1M character string.

Step 9: Function timing

We can simply observe the data we received from the last part that we used in order to graph the run-time curves.

The vectorized function reigns supreme by a large margin as it is able to process the entire 591k character string in 0.03 seconds. The preallocated function is able to do it in an impressive 21 seconds. The basic URLdecode function comes in at last and is able to do it in around 280 seconds. For a complex string like the one provided, it can decode at a rate of 2113 characters per second. This is much lower than my initial estimation of 4000 based on the sample strings I provided at the start.

```
nchar(largetext) / original_df[9,1]
```

```
## [1] 2113.846
```

Conclusion

Overall, we learned that even the base functions provided to us in R can be greatly reduced in computation time. I always personally assumed that I would never reach the point in my programming career where I felt

like I would be able to write a package or have any effect on a programming language's development history. Now, it doesn't feel so out of reach!

Another important lesson I learned was what vectorization truly meant. When Professor mentioned it throughout the year, I thought it was the application of apply() functions instead of using for/while loops. During my project, I realized that sometimes those apply() functions can actually be slower than loops, and that vectorization is doing all operations with a vector's bounds. I never realized how incredibly reliable this process was; the fact that it did not budge in computation time from 1k to 1M characters is truly remarkable.

Thanks for a great quarter to everyone on the teaching staff. I learned so much more from these past projects than I did in any other class I have taken here at UC Davis. It certainly was not easy, but I appreciate it!