C S 509 HW1

Author: **Indronil Bhattacharjee**
Submitted on: **September 5, 2023**

Task 1.1: Learning Quarto

Source Quarto, compiled HTML and PDF files are submitted as a part of this homework.

Task 2.1: Function f1() to Subtract Numeric Vectors using For-Loop

The f1() function calculates the element-wise difference between two numeric vectors, x and y, and returns a new vector z where each element of z is the result of subtracting the corresponding elements of y from x using a for-loop.

```
f1 <- function(x, y) {
  n <- length(x)
  z <- numeric(n)
  for (i in 1:n) {
    z[i] <- x[i] - y[i]
  }
  return(z)
}
```

Task 2.2: Function f2() to Subtract Numeric Vectors with Vectorized Operation

The f2() function vectorized version of the element-wise subtraction operation between two numeric vectors, x and y. It leverages R's built-in vectorized features.

```
f2 <- function(x, y) {
  z <- x - y
  return(z)
}
```

Task 2.3: Comparing Runtimes and Testing Equality

We are assessing the performance of two different approaches (using a for-loop in f1 and a vectorized operation in f2) for subtracting two vectors x and y. By measuring the runtime, we can compare the efficiency of these approaches, particularly for large vector sizes, here 1,000,000. Verifying equality with expect_equal() ensures that both approaches produce the same result, demonstrating that the vectorized approach is functionally equivalent to the for-loop approach.

```
# Create vectors x and y of length 1,000,000
x <- rnorm(1000000)
y <- rnorm(1000000)

# Measure runtime and test equality
```

```
t1 <- system.time(f1(x, y))[1]
t2 <- system.time(f2(x, y))[1]

z_loop <- f1(x, y)
z_vectorized <- f2(x, y)

# Print Outputs
expect_equal(z_loop, z_vectorized)

# Print runtimes
print(paste("Runtime f1:", t1))
print(paste("Runtime f2:", t2))
```

```
[1] "Runtime f1: 0.127"
[1] "Runtime f2: 0"
```

Here is the description of plotting the runtimes.

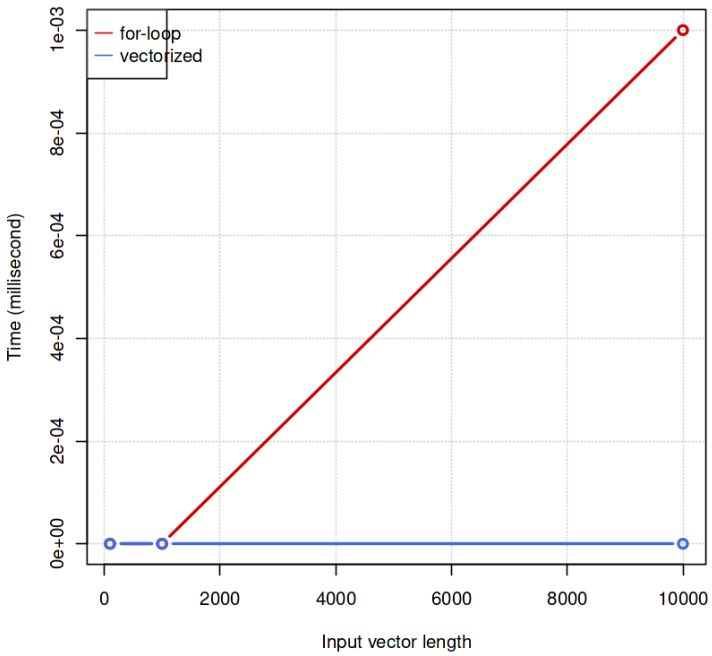
```
#plotting runtime description
plot.runtime <- function(ns, t.loop, t.vectorized)
{
  plot(ns, t.loop, ylim=c(0, max(t.loop)), col="red3",
       type="b", panel.first=grid(), lwd=2,
       xlab="Input vector length",
       ylab="Time (millisecond)")
  lines(ns, t.vectorized, col="royalblue", type="b", lwd=2)
  legend("topleft", c("for-loop", "vectorized"),
         col=c("red3", "royalblue"), lwd=1)
}
```

Task 2.4: Plotting runtime difference

This task involves creating a plot to visualize the runtime difference between two functions, f1 and f2, when applied to input pairs of varying lengths. The goal is to see how the runtime of these functions changes as the input vector length varies.

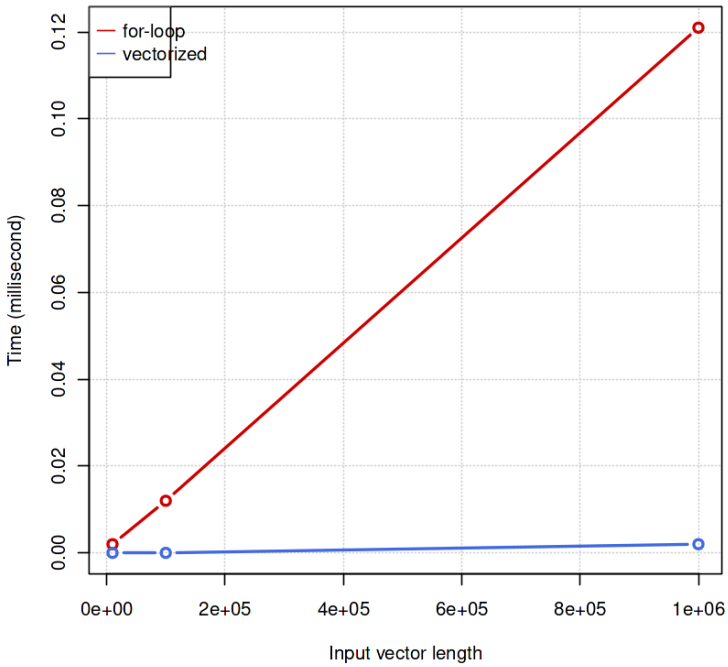
Runtimes for 100, 1000 and 10000 elements:

```
library(forecast)
ns <- c(100, 1000, 10000)  # Varying input lengths
t.loop <- numeric(length(ns))
t.vectorized <- numeric(length(ns))

for (i in 1:length(ns)) {
  x <- rnorm(ns[i])
  y <- rnorm(ns[i])

  t.loop[i] <- system.time(f1(x, y))[1]
  t.vectorized[i] <- system.time(f2(x, y))[1]
}

plot.runtime(ns, t.loop, t.vectorized)
```



Runtimes for 10000, 100000 and 1000000 elements:

```
library(forecast)
ns <- c(10000, 100000, 1000000)  # Varying input lengths
t.loop <- numeric(length(ns))
t.vectorized <- numeric(length(ns))

for (i in 1:length(ns)) {
  x <- rnorm(ns[i])
  y <- rnorm(ns[i])

  t.loop[i] <- system.time(f1(x, y))[1]
  t.vectorized[i] <- system.time(f2(x, y))[1]
}

plot.runtime(ns, t.loop, t.vectorized)
```



Here we can clearly notice that the vectorized operation runs more efficiently than that of a for-loop operation. For a small number of operations, it may not vary so much. But as we are increasing the vector's element size, it shows a daylight difference between the operations.

Task 2.5: Projecting runtime for 10 billion elements

This task measures and compares the runtimes of two functions, f1 and f2, for different input sizes, and estimates the time saved by using the vectorized operation for a very large input size of 10 billion elements. The ratio of runtimes for different input sizes is used to extrapolate the runtime for the larger input size.

The nearest 10's powers of 10 billion (10 million, 100 million, 1 billion) has been selected here as the number of elements to get the time required to subtract two vectors.

```
library(forecast)
ns <- c(10000000, 100000000, 1000000000)  # Varying input lengths
t.loop <- numeric(length(ns))
t.vectorized <- numeric(length(ns))

for (i in 1:length(ns)) {
  x <- rnorm(ns[i])
  y <- rnorm(ns[i])

  t.loop[i] <- system.time(f1(x, y))[1]
  t.vectorized[i] <- system.time(f2(x, y))[1]
}
```

Runtimes for 10000000, 100000000, 1000000000 elements are recorded for both for-loop and vectorized operation.

```
# Runtimes for 10000000, 100000000, 1000000000 using a for-loop
t1 <- t.loop[1]
t2 <- t.loop[2]
t3 <- t.loop[3]
print(paste(t1,t2,t3))

# Runtimes for 10000000, 100000000, 1000000000 using vectorized operation
tv1 <- t.vectorized[1]
tv2 <- t.vectorized[2]
tv3 <- t.vectorized[3]
print(paste(tv1,tv2,tv3))
```

```
[1] "1.322 13.195 134.3"
[1] "0.0139999999999993 0.126000000000001 1.596"
```

Since 10 billion is a very large number, projecting the time it would take to process vectors of 10 billion elements on a typical computer can be challenging due to memory and computational constraints. However, you can perform a rough estimation using the time data you have from smaller vectors and extrapolating based on known relationships.

```
# Calculate the average ratio of runtime increase for for-loop operation
ratio_loop <- ((t2 / t1) + (t3 / t2)) / 2

# Estimate the runtime for 10 billion elements using the ratio
estimated_10_billion <- t3 * ratio_loop
```

```
print(paste("Estimated runtime for 10 billion elements:",
estimated_10_billion))
```

```
[1] "Estimated runtime for 10 billion elements: 1353.68940859469"
```

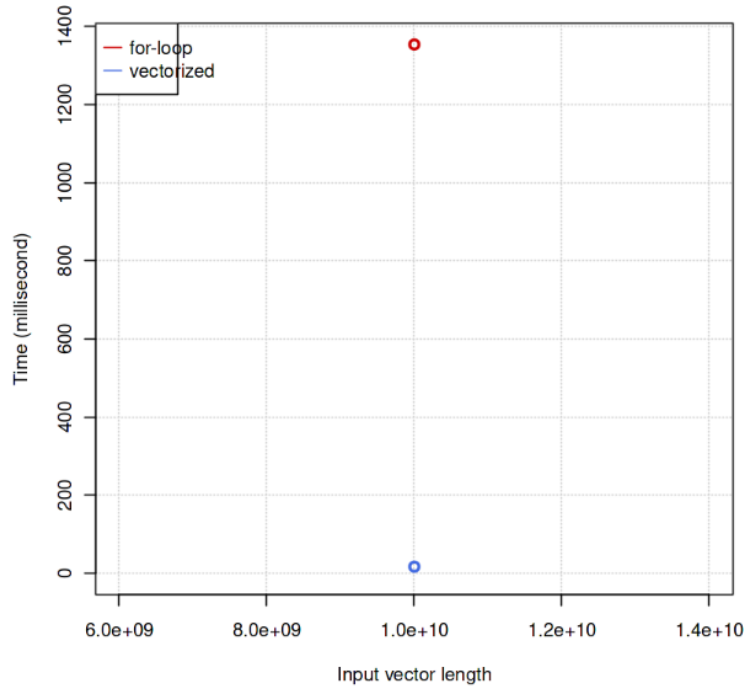
```
# Calculate the average ratio of runtime increase for vectorized operation
ratio_vec <- ((tv2 / tv1) + (tv3 / tv2)) / 2

# Estimate the runtime for 10 billion elements using the ratio
estimated_v_10_billion <- tv3 * ratio_vec

print(paste("Estimated runtime for 10 billion elements:",
estimated_v_10_billion))
```

```
[1] "Estimated runtime for 10 billion elements: 17.2900000000004"
```

```
# Time saved with vectorized operation
time_saved_with_vectorized = estimated_10_billion - estimated_v_10_billion
print(time_saved_with_vectorized)
```

```
[1] 1336.399
```

```
# Plotting estimated runtimes for 10 billion element vector operation
plot.runtime(10000000000, estimated_10_billion, estimated_v_10_billion)
```



The runtime for vectors of varying lengths (e.g., 10 million, 100 million, 1 billion) has been measured, and observed how the runtime scales with increasing vector size, specifically, the relationship between the vector length and the runtime. Next, we extrapolated to estimate the runtime for 10 billion elements.

Here we can observe 1336.399 seconds of difference between the for-loop method and the vectorized method.

Task 2.6: Conclusion

For element-wise operations on huge datasets, vectorized operations often perform better than for-loops. Other element-wise operations like multiplication are predicted to benefit from this efficiency increase as well.

```
# Vector multiplication by for-loop
m1 <- function(x1, y1) {
  n <- length(x1)
  z1 <- numeric(n)
  for (i in 1:n) {
    z1[i] <- x1[i] * y1[i]
  }
  return(z1)
}

# Vector multiplication by vectorized operation
m2 <- function(x1, y1) {
  z1 <- x1 * y1
  return(z1)
}

# Comparing runtime for 1,000,000 elements
x1 <- rnorm(1000000)
y1 <- rnorm(1000000)

# Measure runtime and test equality
mt1 <- system.time(m1(x1, y1))[1]
mt2 <- system.time(m2(x1, y1))[1]

m_loop <- m1(x1, y1)
m_vectorized <- m2(x1, y1)

# Expect equal
expect_equal(m_loop, m_vectorized)

# Print runtimes
print(paste("Runtime m1:", mt1))
print(paste("Runtime m2:", mt2))
```

```
[1] "Runtime m1: 0.143000000000001"
[1] "Runtime m2: 0.00100000000000477"
```

Here m1 and m2 functions are created to do vector multiplication using a for-loop and vectorized operation in the same way. And we can observe the runtimes for 1,000,000 elements are 0.143 and 0.001 seconds in the cases of for-loop and vectorized operations respectively.

So it can be concluded that element-wise operations like multiplication also have better time efficiency in the case of R's built-in vectorized operations.

Task 3.1: GRanges Class

GRanges class is a fundamental data structure offered by the Bioconductor package GenomicRanges in R. It is defined to effectively manage and work with genomic intervals. Genomic intervals are sections of a chromosome whose start and end coordinates are known. Due to its practical representation and handling of interval data, the GRanges class is often used in the study of genomic data.

Each interval in GRanges is determined by the following characteristics:

- **Chromosome:** The chromosomal or sequence identification that the interval is found on.
- **Start Position:** The location on the chromosome where the interval begins.
- **End Position:** The location on the chromosome where the interval ends.
- **Strand:** Orientation of the interval relative to the strand, either forward or backward.
- **Additional metadata:** Any other information connected to the interval, such as annotations or scores, is referred to as supplementary metadata.

The class is an effective tool for examining genomic data since it offers a variety of operations and methods for modifying and querying intervals.

Task 3.2: Biological Data Involving Intervals

Exons and genes: The basic building blocks of genetic information are genes. They include both non-coding (exons) and coding regions (introns). Gene structures are frequently represented using the GRanges class, where each exon is modeled as an interval along the chromosome. Researchers can use this approach to examine gene annotations, find alternative splice variants, and investigate gene expression.

Genomic data involves intervals because it is essential to pinpoint the locations of genes, regulatory elements, and other functional elements on the genome accurately. Genomic intervals are used to define the boundaries of genes, transcription factor binding sites, and other genomic features, allowing researchers to study gene expression, regulation, and genome structure.

Peaks in ChIP-seq: ChIP-seq (Chromatin Immunoprecipitation Sequencing) is a technique for locating DNA areas that are bound by certain proteins, like transcription factors or histones. In the sequencing data, the binding sites are frequently shown as peaks. Data from ChIP-seq peaks, where each peak is represented as an interval, can be stored using the GRanges class. This makes it possible to identify regulatory components and evaluate the distribution of protein-DNA interactions.

ChIP-seq data involves intervals because it precisely identifies the regions where specific proteins interact with DNA. These intervals help researchers understand gene regulation, identify potential regulatory elements, and study how proteins influence chromatin structure. Analyzing ChIP-seq peaks involves working with these intervals to gain insights into gene expression and regulation.

References

[1] http://bioconductor.org/packages/release/bioc/html/GenomicRanges.html

[2] https://en.wikipedia.org/wiki/ChIP-sequencing