Stat 142 Machine Problem 1

First Semester, A.Y. 2025-26

First Name: Anne Christine

Last Name: Amores

Student Number: 2021-04650

Section: WFU

First Name: Aleksa Chloe

Last Name: Cayanan

Student Number: 2021-02926

Section: WFU

First Name: Nicole

Last Name: Clemente

Student Number: 2022-13838

Section: WFU

I. Approximating Logarithms

Recall the $f(x) = log(x),\ \forall x \in R,\ x > 0$ can be approximated using the following series:

$$log(x) = 2\sum_{k=0}^{\infty} \frac{1}{2k+1}\left(\frac{x-1}{x+1}\right)^{2k+1}$$

Your task is to write functions that can implement the approximation above through the following algorithm:

i. Compute for the first term of the series above. This will be your initial guess.

ii. Check whether the absolute difference between x and the exponential value of the initial guess is less than the tolerance value.

iii. If the absolute difference is less than the tolerance (default value = `1e-6`), return guess. Otherwise, update guess by including next term in the series, e.g., second guess will include the first two terms of the series, while third guess will include the first three terms, and so on.

1. Without using recursion, implement functions as arguments and functions as general methods via **improve** in performing the algorithm provided. Name your general method function as `log_approx_gen`, which should have at least three arguments, x, *tolerance* with default value of `1e-6`, and an argument accepting a function. It's up to you how many functions you will create to implement this item (as needed). No need to include documentations here.

```r
log_term <- function(x, k) {
  2 * (1/(2*k+1)) * ((x-1)/(x+1))^(2*k+1)
}

log_approx_gen <- function(x, term_func, tolerance = 1e-6) {
  k <- 0
  guess <- term_func(x, k)

  while (abs(x - exp(guess)) > tolerance) {
    k <- k + 1
    guess <- guess + term_func(x, k)
  }
  return(guess)
}
```

2. Modify your function `log_approx_gen` such that it returns NaN when $x < 0$, `-Inf` when $x = 0$, and NA if x is non-numeric (you may use a built-in R function here).

```r
log_term <- function(x, k) {
  2 * (1/(2*k+1)) * ((x-1)/(x+1))^(2*k+1)
}

log_approx_gen <- function(x, term_func, tolerance = 1e-6) {

  # Description
  # Approximates the natural logarithm of a positive number x using a series expansion.

  # Parameters
  # x -- a positive numeric value for which log(x) is approximated
  # term_func -- a function that computes the k-th term of the log series expansion
  # tolerance -- numeric threshold (default: 1e-6) wherein the approximation stops once
  # the absolute difference between exp(guess) and x is smaller than this value

  # Value
  # Returns the approximated log(x) as a numeric value, for x > 0.
  # If x < 0, returns NaN; if x == 0, returns -Inf; and if x is non-numeric, returns NA

  if (!is.numeric(x) || x <= 0) {
    if (x < 0) {
      return(NaN)
    } else if (x == 0) {
      return(-Inf)
    } else {
     return(NA)
    }
  }
  k <- 0
  guess <- term_func(x, k)

  while (abs(x - exp(guess)) >= tolerance) {
    k <- k + 1
    guess <- guess + term_func(x, k)
  }
  return(guess)
}
```

3. Without using recursion, implement the same algorithm (including the input validation), but only define **one function** (call this `log_approx`) in the **global environment**, such that your function should accept ONLY two arguments, x and *tolerance*, with default value of `1e-6`.

```r
log_approx <- function(x, tolerance = 1e-6) {

  # Description
  # Approximates the natural logarithm of a positive number x using a series expansion.

  # Parameters
  # x -- a positive numeric value for which log(x) is approximated
  # tolerance -- numeric threshold (default: 1e-6) wherein the approximation stops once
  # the absolute difference between exp(guess) and x is smaller than this value

  # Value
  # Returns the approximated log(x) as a numeric value, for x > 0.
  # If x < 0; returns NaN; if x == 0, returns -Inf; and if x is non-numeric, returns NA

  if (!is.numeric(x) || x <= 0) {
    if (x < 0) {
     return(NaN)
    } else if (x == 0) {
     return(-Inf)
    } else {
      return(NA)
    }
  }

  k <- 0
  guess <- 2 * (1/(2*k+1)) * ((x-1)/(x+1))^(2*k+1)

  while (abs(x - exp(guess)) >= tolerance) {
    k <- k + 1
    guess <- guess + 2 * (1/(2*k+1)) * ((x-1)/(x+1))^(2*k+1)
  }
  return(guess)
}
```

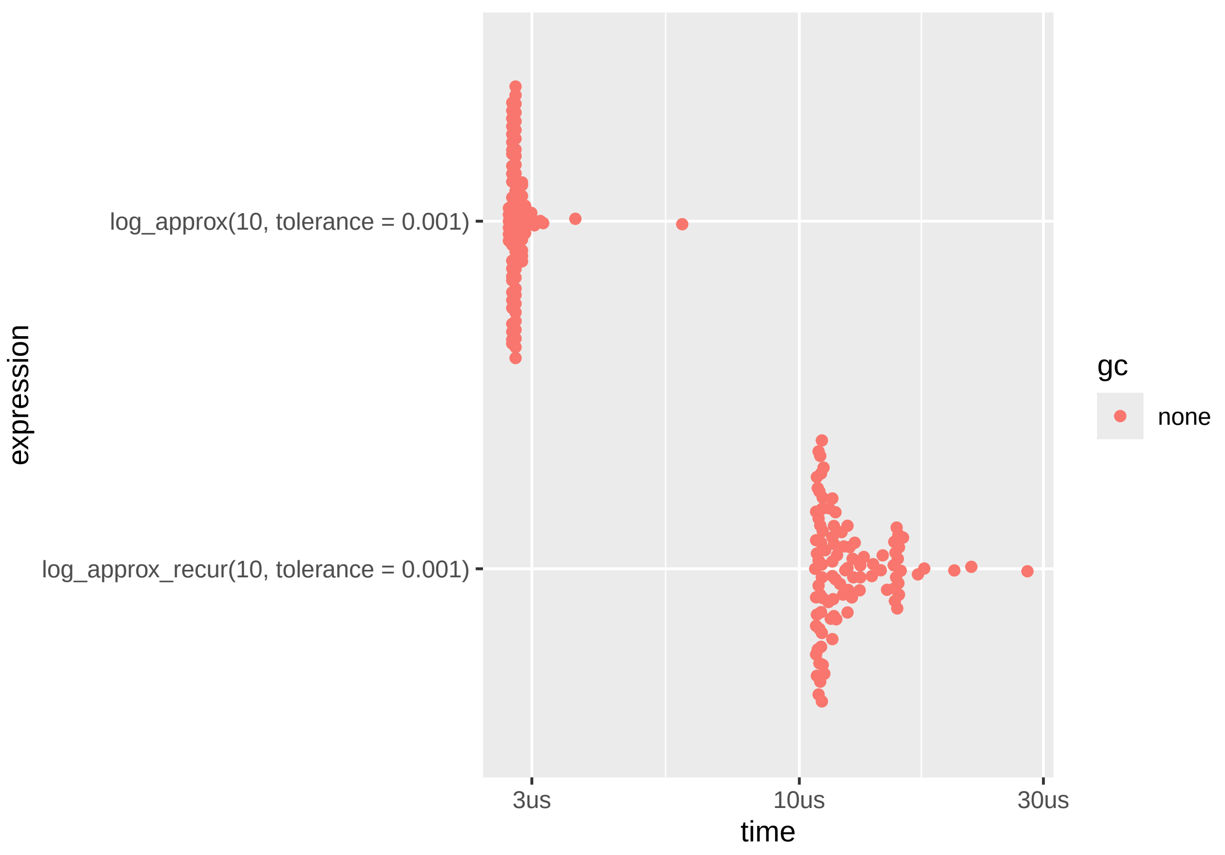
4. Create a recursive version of `log_approx` and call this `log_approx_recur`. It should have at least two arguments, x and *tolerance*. Again, include the input validation.

```r
log_approx_recur <- function(x, tolerance = 1e-6, k = 0, sum = 0) {

  # Description
  # Recursively approximates the natural logarithm of a positive number x using a series
  # expansion

  # Parameters
  # x -- a positive numeric value for which log(x) is approximated
  # tolerance -- numeric threshold (default: 1e-6) wherein the recursion stops once
  # the absolute difference between exp(sum) and x is smaller than this value
  # k -- index of the current term in the series (default:0)
  # sum -- accumulated sum of the series up to the current term (default:0)

  # Value
  # Returns the approximated log(x) as a numeric value, for x > 0.
  # If x < 0, returns NaN; if x == 0, returns -Inf; and if x is non-numeric, returns NA

  if (!is.numeric(x) || x <= 0) {
    if (x < 0) {
      return(NaN)
    } else if (x == 0) {
      return(-Inf)
    } else {
      return(NA)
    }
  }

  guess <- 2 * (1/(2*k+1)) * ((x-1)/(x+1))^(2*k+1)
  sum <- sum + guess
  # Base case
  if (abs(x - exp(sum)) < tolerance) {
    return(sum)
  }
  # Recursive call
  log_approx_recur(x, tolerance, k + 1, sum)
}
```

5. Plot the median run time of `log_approx` and `log_approx_recur` using the following test values (and use `iterations` = 100):

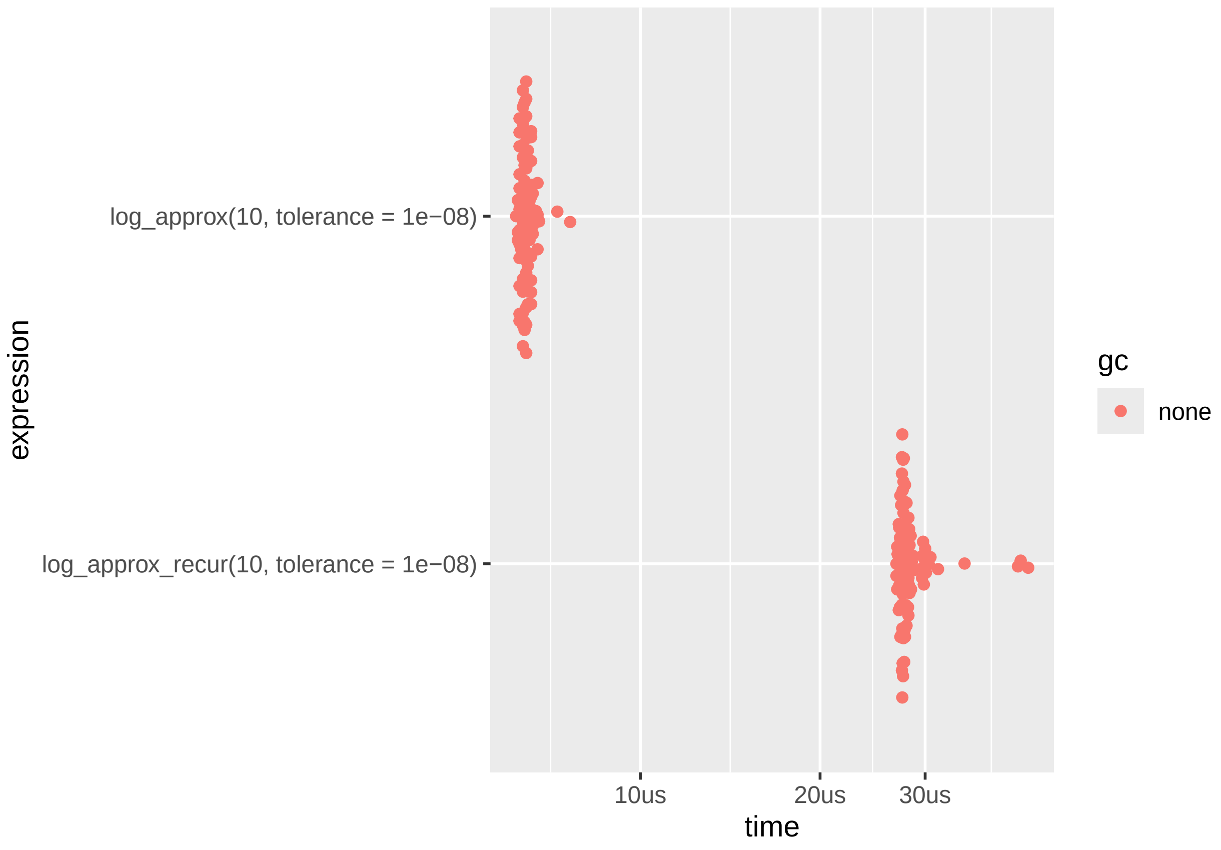
a. `x` = 10, `tolerance` = `1e-3`

```r
benchmark1 <- bench::mark(log_approx(10, tolerance = 1e-3),
                          log_approx_recur(10, tolerance = 1e-3),
                          iterations = 100,
                          time_unit = "ms")
plot(benchmark1)
```



b. `x` $=10$, `tolerance` $=$ `1e-8`

```r
benchmark2 <- bench::mark(log_approx(10, tolerance = 1e-8),
                          log_approx_recur(10, tolerance = 1e-8),
                          iterations = 100,
                          time_unit = "ms")
plot(benchmark2)
```



6. From item 5, which of the two functions is slower and why do you think this is the case?

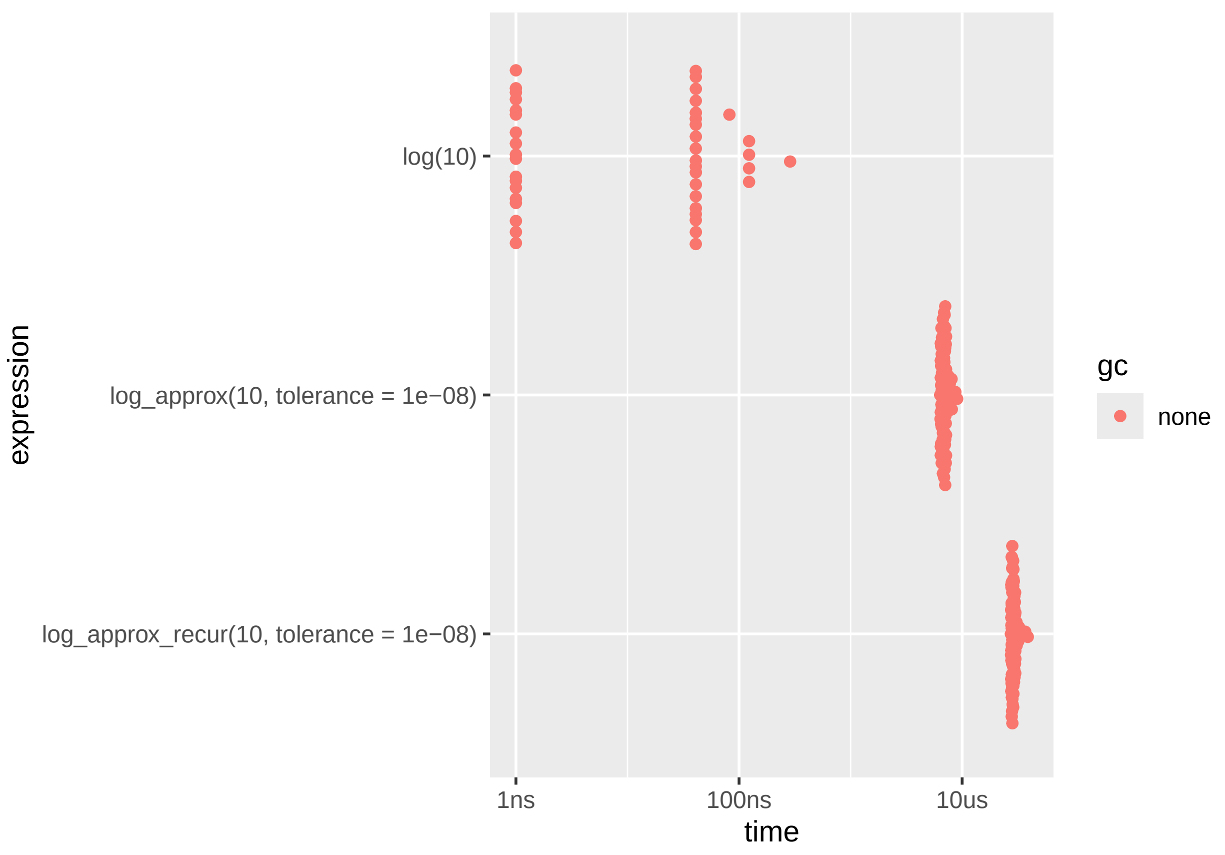
`log_approx_recur()` is slower than `log_approx()` in computing the approximation of `log()`. This is probably because every time `log_approx_recur()` calls itself, R has to create a new environment to store that function call's variables and arguments. Setting up and cleaning up all these separate environments uses more memory and time, making the recursive approach slower than the iterative one. This confirms what we have previously discussed in class, that recursive functions tend to be neater or more mathematically elegant, but less computationally efficient than iterative functions.

7. Plot the median run time again for `x` = 10, `tolerance` = `1e-8` but include the `log()` built-in function in your plot (note that this may have a different value than your approximation functions). What did you notice?

```
benchmark3 <- bench::mark(log_approx(10, tolerance = 1e-8),
                          log_approx_recur(10, tolerance = 1e-8),
                          log(10),
                          iterations = 100,
                          time_unit = "ms")
plot(benchmark3)
```

```
## Warning in scale_x_bench_time(): bch:tm-10 transformation introduced infinite
## values.
```

```
## Warning: Removed 57 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



Looking at the plot, the built-in function `log()` is the fastest, and the recursive function `log_approx_recur()` is the slowest. While the iterative function `log_approx()` is faster than `log_approx_recur()`, it is stil not faster than `log()`.

II. Euler's Method for Integration

Suppose we want to evaluate a definite integral

$$\int_D f(x)dx$$

where D represents some domain, and f is a given function. For most functions, there is no closed-form solution for such definite integrals.

Numerical analysis offers several techniques for approximating definite integrals, one of the simplest being Euler's method for one-dimensional integration:

$$\int_b^a f(x)dx \approx \sum_{i=1}^{\lfloor (b-a)/h \rfloor} hf(a+ih)$$

Where $\lfloor (b-a)/h \rfloor$ denotes the floor of $(b-a)/h$

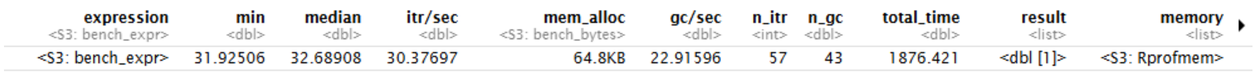
1. Your task is to write a function `integral` that uses Euler's method to approximate a definite integral, which should accept `f` and `h` as parameters, as well as `a` and `b` which represent the lower and upper limits of integration, respectively. Include documentations.

```r
# Assume x = a + i*h (based on the first source in the References section)

integral <- function(func, a, b, h) {

  # Description
  # Applies Euler's method to aproximate a definite integral.

  # Parameters
  # func -- The function to be evaluated inside the definite integral
  # h -- given value
  # a -- upper limit of the integration
  # b -- lower limit of the integration

  # Value
  # Evaluated approximation of definite integral using Euler's method.

  n <- floor((b-a)/h) # Upper bound of the summation
  term <- 0
  x <- a # Because when i = 0, x = a + 0*h = a
  for (i in 1:n){
    x <- a+(i*h)
    term <- term + h*func(x)
    }
  return(term)
}
```

```r
# Should get value of 0.6931397
integral(function(x){1/x}, a=1, b=2, h=1e-5)
```

```
## [1] 0.6931397
```

To show that `integral()` runs no longer than 45 milliseconds (for `iterations = 100`),



expression	min	median	itr/sec	mem_alloc	gc/sec	n_itr	n_gc	total_time	result	memory
<S3: bench_expr>	<dbl>	<dbl>	<dbl>	<S3: bench_bytes>	<dbl>	<int>	<dbl>	<dbl>	<list>	<list>
<S3: bench_expr>	31.92506	32.68908	30.37697	64.8KB	22.91596	57	43	1876.421	<dbl [1]>	<S3: Rprofmem>

2. How does the value of h affect the time complexity and performance of the algorithm? You may try testing on $h = 0.1$, 0.01, 0.001, 0.0001, 0.00001 to describe the phenomenon, no need to plot the complexity.

```r
a <- bench::mark(
  integral(function(x){1/x}, a=1, b=2, h=0.1),
  iterations=100,
  time_unit="ms")

# For h = 0.01
b <- bench::mark(
  integral(function(x){1/x}, a=1, b=2, h=0.01),
  iterations=100,
  time_unit="ms")

# For h = 0.001
```

```
c <- bench::mark(
  integral(function(x){1/x}, a=1, b=2, h=0.001),
  iterations=100,
  time_unit="ms")

# For h = 0.0001
d <- bench::mark(
  integral(function(x){1/x}, a=1, b=2, h=0.0001),
  iterations=100,
  time_unit="ms")

# For h = 0.00001
e <- bench::mark(
  integral(function(x){1/x}, a=1, b=2, h=0.00001),
  iterations=100,
  time_unit="ms")
```

```
a
```

```
## # A tibble: 1 x 6
##   expression                                 min  median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                               <dbl>   <dbl>     <dbl> <bch:byt>    <dbl>
## 1 integral(function(x) { 1/x }, a ~ 0.00238 0.00254   388872.        0B        0
```

```
b
```

```
## # A tibble: 1 x 6
##   expression                                  min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                                <dbl>  <dbl>     <dbl> <bch:byt>    <dbl>
## 1 integral(function(x) { 1/x }, a = ~ 0.0171 0.0180    51966.        0B        0
```

```
c
```

```
## # A tibble: 1 x 6
##   expression                                  min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                                <dbl>  <dbl>     <dbl> <bch:byt>    <dbl>
## 1 integral(function(x) { 1/x }, a = 1~ 0.165  0.172      5641.        0B     115.
```

```
d
```

```
## # A tibble: 1 x 6
##   expression                                  min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                                <dbl>  <dbl>     <dbl> <bch:byt>    <dbl>
## 1 integral(function(x) { 1/x }, a = 1~   1.64   1.70       579.        0B     118.
```

```
e
```

```
## # A tibble: 1 x 6
##   expression                                  min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                                <dbl>  <dbl>     <dbl> <bch:byt>    <dbl>
## 1 integral(function(x) { 1/x }, a = 1~   17.1   17.5       56.7        0B     95.9
```

From the results of the benchmark tests, we can see that as h gets smaller, the processing time increases, i.e., `integral()` becomes slower in performing the computation.

III. Estimation via Resampling

Resampling techniques are essential tools in data science and statistics because they allow us to estimate the variability of statistics, approximate distributions, and evaluate model performance without needing strong assumptions about the underlying population.

One simple but powerful resampling technique is subsampling, where we repeatedly draw smaller samples without replacement from the original dataset and compute a statistics (e.g., mean, variance) per subtest.

1. In this item, you will implement subsampling using recursion:

   Write a function `subsample_recur` that performs subsampling recursively and has **at least** the following function arguments/parameters:

   - samp - the original dataset (a numeric vector)
   - k - the size of each subsample (must be less than the length of samp)
   - B - the number of times subsampling should be performed
   - func - function that would compute a specific statistic

   The function should:

   - Draw a subsample **without replacement** and compute a statistic using that subsample (e.g., mean).
   - Do the first step multiple times, and stop when B subsamples have been drawn
   - Return a vector of results (e.g., vector of means)

NOTE:

- Your function should ensure that $k < length(sample)$ when performing subsampling. Otherwise, it should inform the user that their k is invalid.
- Use `sample()` with `replace = FALSE` to draw subsamples. Make sure to set a seed number before every call of the `sample()` function such that you will set the seed using the current resample count (e.g., if you are at the 10th subsample, out of B = 100, then set your seed number to 10).

For example:

```
sample_function <- function(arg1, arg2, ...) {
  # <instructions>
  set.seed(resample_count)
  subsamp <- sample(arg1, arg2, replace = FALSE)
  # <instructions>
  return(obj)
}
```

- Do not forget to include **documentations**.

```
#1: Sub sampling using recursion
subsample_recur <- function(samp, k, B, func, resample_count=1) {

  # Description
  # Recursively performs subsampling on a dataset, wherein for each subsample drawn
  # without replacement, a statistic is computed.
  # Parameters
  # samp -- original dataset, a numeric vector
  # k -- size of each subsample
  # B -- number of times subsampling is drawn
  # func -- function to compute the statistic (e.g. mean, variance) to each subsample
  # resample_count -- indicates the current subsample during recursion

  # Value
  # A numeric vector of length B, where each entry is the statistic computed from a
  # subsample

  if (resample_count==1 && k>=length(samp)) {
    cat("k is invalid")
    return(NULL)
  }

  # base case
  if(resample_count>B) {
    return(c())
  }

  # recursive call
  else {
    set.seed(resample_count)
    subsamp <- sample(samp, k, replace = FALSE)
    statistic <- func(subsamp)
    return(c(statistic, subsample_recur(samp, k, B, func,  resample_count+1)))
  }
}
```

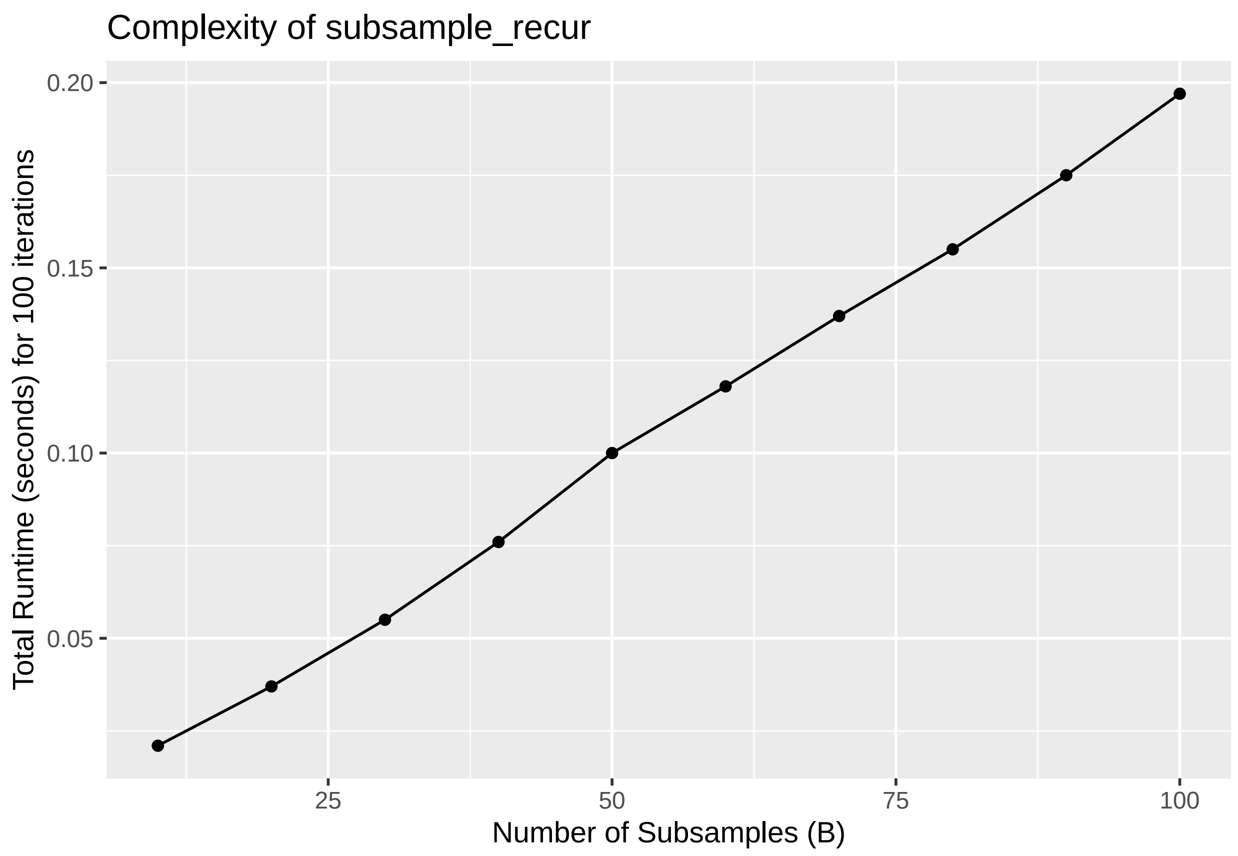
2. Use the following dataset:

```
set.seed(142)
x <- rnorm(1000, mean = 50, sd = 10)
```

Plot the complexity of `subsample_recur` for different sizes of B (i.e., B = 10, 20, 30, . . . , 100) and `iterations = 100`. Describe the complexity that you will be able to form from the plot.

```
#2: Complexity of subsample_recur for different sizes

set.seed(142)
x <- rnorm(1000, mean=50, sd=10)

B_sizes <- seq(from=10,to=100,by=10)
average <- c()

for (B in B_sizes) {
  time <- as.numeric(system.time({
    for (i in 1:100) {
      subsample_recur(x,k=500,B=B,func=mean)
    }
  })
```

```r
  ["elapsed"])

  average<- c(average, time)
}

complexity <- data.frame(B = B_sizes, total_time = average)

# Plot of the complexity of subsample_recur
ggplot(complexity, aes(x = B, y = total_time)) +
  geom_line() +
  geom_point() +
  labs(title = "Complexity of subsample_recur",
       x = "Number of Subsamples (B)",
       y = "Total Runtime (seconds) for 100 iterations"
  )
```



Based on the plot, the total runtime increases as the number of subsamples B increases. The overall trend appears to be linear, which is expected expected since each subsample requires one recursive call, making the time complexity proportional to B. In other words, O(B).

IV. Querying databases in SQL

The database `chinook.db` contains several tables with information on music purchases and customers.

To prepare the connection, run the following (make sure to save **`chinook.db`** in your working directory):

```r
library(DBI)
library(RSQLite)

# Create a connection
con <- dbConnect(RSQLite::SQLite(), dbname = "chinook.db")
```

The database contains invoice information spread across multiple tables. Use the following SQL code to create a new table called `invoice_summary`, which contains **unique** invoice-level information including customer details and transaction totals. You would be using this on top of the other tables available in this database.

```sql
CREATE TABLE invoice_summary AS
SELECT DISTINCT
    i.InvoiceId,
    i.CustomerId,
    c.Country,
    i.InvoiceDate,
    i.Total
FROM invoices i
JOIN customers c ON i.CustomerId = c.CustomerId
```

Part 1. Basic SQL queries

1. From the `invoice_summary` table, compute/show the following **by country** (make sure to always show the countries involved

a. Total number of transactions

```sql
SELECT Country, COUNT() AS n_transactions
  FROM invoice_summary
  GROUP BY Country
```

Table 1: Displaying records 1 - 10

Country	n_transactions
Argentina	7
Australia	7
Austria	7
Belgium	7
Brazil	35
Canada	56
Chile	7
Czech Republic	14
Denmark	7
Finland	7

b. Average revenue (or the average of all money received from transactions; Only show the first 5 observations)

```
SELECT Country, AVG(Total) AS avg_revenue
  FROM invoice_summary
  GROUP BY Country
  LIMIT 5
```

Table 2: 5 records

Country	avg_revenue
Argentina	5.374286
Australia	5.374286
Austria	6.088571
Belgium	5.374286
Brazil	5.431429

c. Top 10 countries with the highest average revenue

```
SELECT Country, AVG(Total) AS avg_revenue
  FROM invoice_summary
  GROUP BY Country
  ORDER BY avg_revenue DESC
  LIMIT 10
```

Table 3: Displaying records 1 - 10

Country	avg_revenue
Chile	6.660000
Ireland	6.517143
Hungary	6.517143
Czech Republic	6.445714
Austria	6.088571
Finland	5.945714
Netherlands	5.802857
India	5.789231
USA	5.747912
Norway	5.660000

d. Maximum and minimum invoice totals

```
SELECT Country, MIN(Total), MAX(Total)
  FROM invoice_summary
  GROUP BY Country
```

Table 4: Displaying records 1 - 10

Country	MIN(Total)	MAX(Total)
Argentina	0.99	13.86
Australia	0.99	13.86
Austria	0.99	18.86
Belgium	0.99	13.86
Brazil	0.99	13.86
Canada	0.99	13.86
Chile	0.99	17.91
Czech Republic	0.99	25.86
Denmark	0.99	13.86
Finland	0.99	13.86

2. Combine the `artists` table with the `albums` table and only show the observations for ArtistID = 27.

```
SELECT *
  FROM artists
    JOIN albums
      ON artists.ArtistId = albums.ArtistId
WHERE artists.ArtistId = 27
```

Table 5: 3 records

ArtistId	Name	AlbumId	Title	ArtistId
27	Gilberto Gil	85	As Canções de Eu Tu Eles	27
27	Gilberto Gil	86	Quanta Gente Veio Ver (Live)	27
27	Gilberto Gil	87	Quanta Gente Veio ver–Bônus De Carnaval	27

Part 2. Subsampling Analysis

1. From the `invoices` table, query only the invoices with a `Total` greater than `10.0`.
   - Store the resulting `Total` column as a numeric vector named `high_value_totals`. Ensure that the `Total` column is a numeric vector and not a data frame.

```
invoices <- dbGetQuery(con, "
    SELECT *
    FROM invoices
    WHERE Total > 10.0
")
high_value_totals <- as.numeric(invoices[ ,9])
head(high_value_totals)
```

```
## [1] 13.86 13.86 13.86 13.86 13.86 13.86
```

1. Apply your subsampling function (`subsample_recur()`) to `high_value_totals` with the following parameters:
   - Subsample size `k = 50`
   - Number of subsamples `B = 1000`

- Store the resulting vector of mean values as `sample_means`

```r
sample_means <- subsample_recur(high_value_totals, k = 50, B = 1000, func = mean)
head(sample_means)
```

```
## [1] 14.4436 14.8246 14.8056 14.8446 14.8036 14.5456
```

2. Get the variance of `sample_means` and print its result.

```r
var(sample_means)
```

```
## [1] 0.0322794
```

References:

1. https://www.cookbook-r.com/Scripts_and_functions/Measuring_elapsed_time/
2. https://math.libretexts.org/Courses/Monroe_Community_College/MTH_225_Differential_Equations/03%3A_Numerical_Methods/3.01%3A_Euler%27s_Method