HW 4

AUTHOR
Zheng Cui

1. Linear interval search

(40%) Write an R function count.features.linear(chr, x, y, GFF) to find the number of overlapping features between chromosome coordinates x and y on a given chromosome chr and annotation data frame GFF. You must use for-loop to do the counting.

```
# imoprt GFF Data
library(stringr)
file <- "E:/Language/R/gencode.v44.primary_assembly.annotation.gff3"
GFF <- read.table(file, header = FALSE, sep = "\t")
colnames(GFF) <- c("seqid", "source", "type", "start", "end",
                   "score", "strand", "phase", "attributes")
rownames(GFF) <- 1:nrow(GFF)
```

```
count.features.linear <- function(chr, x, y, GFF) {
  # Initialize count to 0
  count <- 0

  # Loop through each row in the GFF data frame
  for (i in 1:nrow(GFF)) {
    # Check if the current feature is on the given chromosome
    if (GFF$seqid[i] == chr) {
      # Check if the current  qps with the interval [x, y]
      if (GFF$start[i] <= y && GFF$end[i] >= x) {
        count <- count + 1
      }
    }
  }

  return(count)
}

# E.g
chromosome <- "chr1"
interval_start <- 1
interval_end <- 20000
result <- count.features.linear(chromosome, interval_start, interval_end, GFF)
cat("Number overlapping features between chromosome coordinates x and y:", result, "\n")
```

```
Number overlapping features between chromosome coordinates x and y: 27
```

2. Vectorized interval search

(30%) Write an R function count.features.vectorized(chr, x, y, GFF) to find the number of features between position x and position y on a given chromosome chr and annotation data frame GFF. You must avoid using a for-loop and use vectorized operations instead.

```
count.features.vectorized <- function(chr, x, y, GFF) {
  # filter given chromosome chr
  chr_rows <- GFF[GFF$seqid == chr,]

  # count the number of features between position x and position y
  overlaps <- sum(chr_rows$start <= y & chr_rows$end >= x)

  return(overlaps)
}

# E.g
chromosome <- "chr1"
interval_start <- 1
interval_end <- 20000
result <- count.features.vectorized(chromosome, interval_start, interval_end, GFF)
cat("Number overlapping features between chromosome coordinates x and y:", result, "\n")
```

```
Number overlapping features between chromosome coordinates x and y: 27
```

3. Binary interval search

(20%) Write an R function count.features.binary(chr, x, y, sorted.coordinates) to find the number of features between position x and position y on a given chromosome chr and annotation data frame GFF. You must use the binary search algorithms we discussed in class. You need to sort the coordinates of all features on the same chromosome before you call the function. An option is the R function sort().

```
# sort GFF Data seqid == "chr1"
sorted_coordinates_chr1 <- list(start = sort(GFF$start[GFF$seqid == "chr1"]),
                                end = sort(GFF$end[GFF$seqid == "chr1"]))
```

```
# FIND ALL TARGET ENDS AFTER QUERY START
find_query_start <- function(ends, query_start) {
  L <- 1
  R <- length(ends)

  while (L <= R) {
    M <- (L + R) %/% 2
    if (ends[M] < query_start) {
      L <- M + 1
    } else {
      R <- M - 1
    }
```

```
  }

  return(R + 1)
}

# FIND ALL TARGET STARTS BEFORE QUERY END
find_query_end <- function(starts, query_end) {
  L <- 1
  R <- length(starts)

  while (L <= R) {
    M <- (L + R) %/% 2
    if (starts[M] > query_end) {
      R <- M - 1
    } else {
      L <- M + 1
    }
  }

  return(L - 1)
}

count.features.binary <- function(chr, x, y, sorted_coordinates_chr1) {
  # Retrieve the sorted start and end coordinates
  sorted_starts <- sorted_coordinates_chr1$start
  sorted_ends <- sorted_coordinates_chr1$end

  start_idx <- find_query_start(sorted_ends, x)
  end_idx <- find_query_end(sorted_starts, y)

  # Count of features is the difference between the indices (+1 since both are inclusive),
  # but we should also handle the case where no overlapping features are found.
  count <- max(0, end_idx - start_idx + 1)

  return(count)
}

# Assuming you have sorted_coordinates properly set up:
chromosome <- "chr3"
interval_start <- 1
interval_end <- 20000
result <- count.features.binary(chromosome, interval_start, interval_end, sorted_coordinates_c
cat("Number of overlapping features:", result, "\n")
```

```
Number of overlapping features: 27
```

4. Reporting the runtime

(10%) Test the three functions on the same input and human genome annotation file. Report the runtime as for each method on at least two

examples using the system.time() function. Do not include time for loading the genome annotation file or sorting the coordinates.

```
chromosome1 <- "chr1"
interval_start1 <- 1
interval_end1 <- 20000

# Linear search
linear_time1 <- system.time({
  result1_linear <- count.features.linear(chromosome1, interval_start1, interval_end1, GFF)
})
cat("Linear search runtime for example 1:", linear_time1["elapsed"], "seconds\n")
```

```
Linear search runtime for example 1: 5.53 seconds
```

```
# Vectorized search
vectorized_time1 <- system.time({
  result1_vectorized <- count.features.vectorized(chromosome1, interval_start1, interval_end1,
})
cat("Vectorized search runtime for example 1:", vectorized_time1["elapsed"], "seconds\n")
```

```
Vectorized search runtime for example 1: 0.09 seconds
```

```
# Binary search
binary_time1 <- system.time({
  result1_binary <- count.features.binary(chromosome1, interval_start1, interval_end1, sorted_
})
cat("Binary search runtime for example 1:", binary_time1["elapsed"], "seconds\n")
```

```
Binary search runtime for example 1: 0 seconds
```

E1 (100%). Query interval spanning only known coordinates

If the query interval [x, y] has a special property such that x and y must take values from coordinates of some intervals in the annotation, can you develop code to run even faster than the best general solutions you have developed above?

Note

x and y do not have to be from the save interval in the annotation. For example, if the annotation includes intervals {(1,3),(8,15),(2,29)}, then(3,8) is a valid query interval,as well as (8,15)

```
# sort GFF
sorted_gff <- GFF[order(GFF$start, GFF$end),]
```

```r
sorted_coords <- c(sorted_gff$start, sorted_gff$end)
sorted_coords <- sort(unique(sorted_coords))  # sort and unique
#  fibonacci_search function
fibonacci_search <- function(arr, x) {
  fibMm2 <- 0  # (m-2)nd Fibonacci number
  fibMm1 <- 1  # (m-1)st Fibonacci number
  fibM <- fibMm2 + fibMm1

  # Gets the smallest value in the Fibonacci sequence that is larger than the array length
  while (fibM < length(arr)) {
    fibMm2 <- fibMm1
    fibMm1 <- fibM
    fibM <- fibMm2 + fibMm1
  }

  offset <- -1

  while (fibM > 1) {
    i <- min(offset + fibMm2, length(arr)-1)

    if (arr[i] < x) {
      fibM <- fibMm1
      fibMm1 <- fibMm2
      fibMm2 <- fibM - fibMm1
      offset <- i
    } else if (arr[i] > x) {
      fibM <- fibMm2
      fibMm1 <- fibMm1 - fibMm2
      fibMm2 <- fibM - fibMm1
    } else {
      return(i)
    }
  }

  if (fibMm1 && arr[length(arr)] == x) {
    return(length(arr))
  }

  return(-1)
}

# query in fibonacci
query_with_fibonacci <- function(x, y, sorted_coords) {
  # use fibonacci to find x and y
  start_index <- fibonacci_search(sorted_coords, x)
  end_index <- fibonacci_search(sorted_coords, y)

  if (start_index == -1 || end_index == -1) {
    stop("Invalid x or y provided!")
  }

  # located
  overlapped_intervals <- sorted_coords[(start_index+1):end_index]
```

```
  return(overlapped_intervals)
}



x <- 577
y <- 20007

result <- query_with_fibonacci(x, y, sorted_coords)
print(result)
```

```
 [1]   647   648  1601  1602  1670  1671  2585  2675  2676  2678  2692  3229
[13]  3230  3304  3307  4262  4263  4329  4331  4400  4402  4469  4470  4612
[25]  5511  5512  5579  5587  5655  5657  5729  5761  5826  5891  5904  5906
[37]  6094  6101  6126  6127  6129  6216  6370  6991  7102  7322  7404  7445
[49]  7446  7514  7518  7585  7586  7588  7817  7977  7989  8050  8267  8269
[61]  8295  8364  8366  8368  8527  8529  8570  8572  8604  8655  9205  9207
[73]  9209  9365  9537  9990  9991 10058 10059 10140 10191 10404 10405 10469
[85] 10470 10472 10598 10760 10762 10764 10766 10911 11083 11103 11125 11287
[97] 11288 11451 11555 11595 11613 11615 11616 11707 11745 11753 11758 11768
[109] 11769 11799 11802 11861 11869 11908 12010 12057 12134 12137 12138 12179
[121] 12190 12206 12207 12227 12265 12266 12291 12294 12310 12336 12337 12340
[133] 12358 12378 12402 12572 12574 12575 12613 12659 12663 12697 12704 12721
[145] 12726 12733 12740 12801 12822 12824 12834 12902 12906 12964 12975 13001
[157] 13052 13076 13088 13102 13152 13156 13157 13201 13221 13338 13351 13354
[169] 13370 13374 13453 13487 13501 13566 13670 13783 14061 14090 14138 14146
[181] 14148 14149 14195 14299 14381 14404 14409 14475 14490 14497 14501 14511
[193] 14513 14522 14588 14652 14653 14659 14671 14673 14674 14720 14742 14747
[205] 14749 14771 14806 14807 14809 14831 14851 14940 14944 14958 15005 15038
[217] 15081 15085 15149 15153 15481 15617 15633 15796 15822 15887 15888 15909
[229] 15913 15947 15953 15956 16023 16061 16065 16290 16352 16448 16502 16537
[241] 16544 16549 16561 16607 16718 16722 16738 16765 16856 16858 16871 16876
[253] 16880 16898 16916 16922 16965 16969 17051 17052 17055 17119 17166 17170
[265] 17233 17234 17263 17264 17266 17291 17344 17348 17368 17369 17427 17436
[277] 17479 17483 17501 17514 17604 17606 17719 17723 17742 17750 17855 17859
[289] 17915 17957 18028 18037 18058 18061 18068 18174 18183 18227 18268 18296
[301] 18364 18366 18367 18373 18381 18432 18471 18475 18476 18478 18479 18492
[313] 18897 18969 19018 19172 19619 19833 19895 20007
```

E2 (100%). Overlapping intervals for every interval

If you are given a collection of n intervals [x1, y1], . . ., [xn, yn] , can you design an algorithm with implementation in any language your like to find overlapping intervals for each and every interval in less than quadratic time little-oh o(n^2)?

```
# Define a function to find overlap in a series of time intervals
find_overlapping_intervals <- function(intervals) {

# Create an events list
  events <- list()
```

```
# For each interval, add a start event and an end event
# event is a vector containing a point in time, an event type (start or end), and an interval
  for (i in 1:length(intervals)) {
    events <- c(events, list(c(intervals[[i]][1], TRUE, i)), list(c(intervals[[i]][2], FALSE,
  }

# Sort events by their coordinates, and then by their type (start before end)
  sorted_events <- events[order(sapply(events, '[', 1), -sapply(events, '[', 2))]


# Initializes a vector to track the current activity interval
  active_intervals <- integer(0)

# Initializes a list of the same length as the interval list to store overlapping interval ind
  overlap <- vector("list", length(intervals))

# use for to iterate over sorted events
  for (event in sorted_events) {
    idx <- event[3]

    if (event[2]) { # If it's a start event
      for (active_idx in active_intervals) {
# Adds each other's indexes to the overlapping list of currently active intervals and current
        overlap[[idx]] <- c(overlap[[idx]], active_idx)
        overlap[[active_idx]] <- c(overlap[[active_idx]], idx)
      }
# Adds the current interval to the active interval list
      active_intervals <- c(active_intervals, idx)
    } else { # If it's an end event
      active_intervals <- setdiff(active_intervals, idx)# Removes the current interval from th
    }
  }
# Returns a list of overlapping intervals
  return(overlap)
}

# Define a list of intervals to test
intervals <- list(c(1, 5), c(2, 6), c(1, 8), c(7, 9), c(10, 15))

# Using the function and print the result
overlaps <- find_overlapping_intervals(intervals)
print(overlaps)
```

```
[[1]]
[1] 3 2

[[2]]
[1] 1 3

[[3]]
[1] 1 2 4
```

```
[[4]]
[1] 3

[[5]]
NULL
```

1. Function definition:

The find_overlapping_intervals function receives a list of intervals, where each element is a two-digit operator that represents the start and end point of an interval.

2. Create event list:

The function first creates an empty list, events. For each interval in the interval, the function adds two events to events: A “start” event (marked TRUE) that contains the start time of the interval and the index of the interval. An “end” event (marked FALSE) that contains the end time of the interval and the index of the interval. This way, each interval is represented by two events in Events.

3. Sort events:

Use the order function to sort the events, first by point in time (time coordinates) and then by event type (start event before end event). The sorting step is important to check for overlap later.

4. Detection overlap:

Initialize active_intervals (an empty integer initialization) to track the time interval for the current activity (that is, it has started but not yet ended). Initializes overlap (a list) to store indexes of other intervals that each interval overlaps.

5. Handle each event:

Iterate through the sorted list of events: If a start event is encountered, its index is added to the mid-active_interval and the overlap is updated to reflect the overlap between the current active interval and the newly started interval. If it is an end event, the active_interval removes the corresponding interval index because the interval has ended.

6. Return result:

The function finally overlaps a list where each element is an integer operator, containing the indexes of other intervals that overlap the corresponding interval returns.

7. Test function:

In the last part of the code, I use list(c(1, 5), c(2, 6), c(1, 8), c(7, 9), c(10, 15)), call the function to test, and then print the result for overlaps.

E3 (25%). Biological application

Demonstrate on a biology dataset where you can apply any program you developed above. You must provide biological motivation and result

interpretation for this part of work

```
# Convert sorted_coordinates_chr1 to intervals format
 intervals_chr1 <- lapply(1:length(sorted_coordinates_chr1$start), function(i) {
  c(sorted_coordinates_chr1$start[i], sorted_coordinates_chr1$end[i])
})

# Apply the find_overlapping_intervals function
overlaps_chr1 <- find_overlapping_intervals(intervals_chr1)
head(overlaps_chr1, 5)
```

```
[[1]]
[1] 2 3 4 5 6

[[2]]
[1] 1 3 4 5 6 7

[[3]]
[1] 1 2 4 5 6 7

[[4]]
[1] 1 2 3 5 6 7 8 9

[[5]]
[1] 1 2 3 4 6 7 8 9
```

```
head(overlaps_chr1, 5)
```

```
[[1]]
[1] 2 3 4 5 6

[[2]]
[1] 1 3 4 5 6 7

[[3]]
[1] 1 2 4 5 6 7

[[4]]
[1] 1 2 3 5 6 7 8 9

[[5]]
[1] 1 2 3 4 6 7 8 9
```