Rosalind problems 23-30

Erik Sundberg

6/21/2019

Contents

23. Enumerating k-mers lexicographically

Given: A collection of at most 10 symbols defining an ordered alphabet, and a positive integer n (n 10).

Return: All strings of length n that can be formed from the alphabet, ordered lexicographically (use the standard order of symbols in the English alphabet).

```
abn <- readLines("rosalind_lexf.txt")
ab <- unlist(strsplit(abn[1], " ")) #given ordered alphabet
n <- as.numeric(abn[2]) #given string-length
len <- length(ab) #nr of letters in the alphabet

lexico <- function(ab, n) {

  letz <- c()
  for(i in 1:(n+1)) {
    letz <- paste(letz, rep(rep(ab, each = len^(n-i) ),
                    len^(n)/(len^(n-i+1) ) ), sep ="")
  }
  return(letz)
}

lex <-lexico(ab, n)
writeLines(lex, "output.txt")
```

Alternatively

```
input <- readLines('rosalind_lexf.txt')
alphi <- strsplit(input[1], ' ')[[1]]
combt <- as.numeric(input[2])
```

```
res <- alphi
len <- length(alphi)
for (i in 2:combt) {
    res <- paste(
        rep(alphi, each = len ** (i - 1)),
        rep(res, times = len),
        sep = ''
    )
}
cat(res, sep = '\n')
```

24. Longest increasing and decreasing subsequence

A subsequence of a permutation is a collection of elements of the permutation in the order that they appear. For example, (5, 3, 4) is a subsequence of (5, 1, 3, 4, 2).

A subsequence is increasing if the elements of the subsequence increase, and decreasing if the elements decrease. For example, given the permutation (8, 2, 1, 6, 5, 7, 4, 3, 9), an increasing subsequence is (2, 6, 7, 9), and a decreasing subsequence is (8, 6, 5, 4, 3). You may verify that these two subsequences are as long as possible.

Given: A positive integer n 10000 followed by a permutation of length n.

Return: A longest increasing subsequence of , followed by a longest decreasing subsequence of .

```
#Solved in Python
```

Alternatively

```
#R
#!/usr/bin/Rscript
seq <- as.numeric(strsplit(readLines('rosalind_lgis.txt')[2], '\\ ')[[1]])
len <- length(seq)
## Preallocate linking and count recording arries
inc_lk <- numeric(len)
inc_ct <- numeric(len)
dec_lk <- numeric(len)
dec_ct <- numeric(len)
for (i in 2:len) {
    max_inc_ct <- 0
    max_dec_ct <- 0
    for (j in which(seq[1:{i-1}] < seq[i])) {
        if (inc_ct[j] >= max_inc_ct) {
            max_inc_ct <- inc_ct[j]
            inc_lk[i] <- j
        }
    }
    for (j in which(seq[1:{i-1}] > seq[i])) {
        if (dec_ct[j] >= max_dec_ct) {
            max_dec_ct <- dec_ct[j]
            dec_lk[i] <- j
        }
    }
    inc_ct[i] <- ifelse(inc_lk[i], 1 + inc_ct[inc_lk[i]], 0)
```

```
    dec_ct[i] <- ifelse(dec_lk[i], 1 + dec_ct[dec_lk[i]], 0)
}
print_seq <- function(link, start) {
    recs <- function(ind) {
        if (ind) {
            recs(link[ind])
            cat(seq[ind], '')
        }
    }
    recs(start)
    cat('\n')
}
print_seq(inc_lk, which(inc_ct == max(inc_ct))[1])
print_seq(dec_lk, which(dec_ct == max(dec_ct))[1])
```

25. Genome assembly as shortest superstring

For a collection of strings, a larger string containing every one of the smaller strings as a substring is called a superstring.

By the assumption of parsimony, a shortest possible superstring over a collection of reads serves as a candidate chromosome.

Given: At most 50 DNA strings of approximately equal length, not exceeding 1 kbp, in FASTA format (which represent reads deriving from the same strand of a single linear chromosome).

The dataset is guaranteed to satisfy the following condition: there exists a unique way to reconstruct the entire chromosome from these reads by gluing together pairs of reads that overlap by more than half their length.

Return: A shortest superstring containing all the given strings (thus corresponding to a reconstructed chromosome).

```
library(seqinr)
reads <- read.fasta("rosalind_long.txt")
dnas <- toupper(unname(sapply(reads, paste, collapse="")))

Gass <- function(dnas) {
first <- c() #beginning of the superstring
for(i in 1:length(dnas)) {
    j = ceiling(nchar(dnas[i])/2)+1
    pat <- substr(dnas[i], 1, j)
    f <- grep(pat, dnas)
    if( length(f) == 1){
      first <- dnas[i]
      break
    }
}
dnas <- dnas[-i] #remove first string

supz <- first #superstring in the making
while( length(dnas) > 0) {
  for(j in 1:floor(nchar(first)/2) ) {
    #suffix-substring pattern to look for in a prefix:
    ss <- substr(supz, (nchar(supz)-nchar(first)+j),nchar(supz))
```

```
    ix <- grep(ss, dnas) #index for string w matching prefix

    if(length(ix) != 0) {
      #add suffix of matching string to the superstring
      supz <- paste(supz, substr(dnas[ix], nchar(ss)+1, nchar(dnas[ix])), sep="")
    break
    }
  }
dnas <- dnas[-ix] #remove DNA-string already added to the superstring
}

writeLines(supz, "output.txt")
}
```

26. Perfect matchings and RNA secondary structures

Given: An RNA string s of length at most 80 bp having the same number of occurrences of 'A' as 'U' and the same number of occurrences of 'C' as 'G'.

Return: The total possible number of perfect matchings of basepair edges in the bonding graph of s.

```
library(seqinr)
library(gmp)
rna <- as.factor(unlist(read.fasta("rosalind_pmch.txt")))

factorialZ(length(rna[rna == "a"]))*factorialZ(length(rna[rna == "g"]))
```

27. Partial permutations

A partial permutation is an ordering of only k objects taken from a collection containing n objects (i.e., k n). For example, one partial permutation of three of the first eight positive integers is given by (5,7,2).

The statistic P(n,k) counts the total number of partial permutations of k objects that can be formed from a collection of n objects. Note that P(n,n) is just the number of permutations of n objects, which we found to be equal to n!=n(n−1)(n−2) (3)(2) in "Enumerating Gene Orders".

Given: Positive integers n and k such that 100 n>0 and 10 k>0.

Return: The total number of partial permutations P(n,k), modulo 1,000,000.

```
nk <- scan("rosalind_pper.txt")
n <- nk[1]; k <- nk[2]
(as.bigz(choose(n,k))*factorialZ(k))%%10^6
```

Alternatively

```
n <- 87L
k <- 10L
nums <- n:(n-k+1)

f <- function(x, y) {
  return( (x*y) %% 1E6L )
}

Reduce( f, nums ) #??
```

28. Introduction to random strings

Given: A DNA string s of length at most 100 bp and an array A containing at most 20 numbers between 0 and 1.

Return: An array B having the same length as A in which B[k] represents the common logarithm of the probability that a random string constructed with the GC-content found in A[k] will match s exactly.

```
prob <- readLines("rosalind_prob.txt")
s <- unlist(strsplit(prob[1],"")) #DNA-string bases
A <- as.numeric(unlist(strsplit(prob[2], " ")))#GC-content
B <- c()
for(j in 1:length(A)) {
  tmp <- c()
  for(i in 1:length(s)) {
    if(s[i]=="A" | s[i]=="T") {
      tmp[i] <- log10(((1-A[j])/2))
    } else if(s[i]=="G" | s[i]=="C") {
      tmp[i] <- log10((A[j]/2))
    }
  }
  B[j] <- round(sum(tmp), 3)
}

write(B, "output.txt",ncolumns = length(A))
```

Alternatively

```
prob <- readLines("rosalind_prob.txt")
s <- prob[1]
A <- as.numeric(unlist(strsplit(prob[2], " ")))

scharsGC <- unlist(strsplit(s, "")) %in% c("G", "C")

B <- sapply(A, function(x){ logAT<-log10((1-x)/2); logGC<-log10(x/2); sum(scharsGC)*logGC+(sum(!scharsG
write(round(B, 3),"output.txt", ncolumns = length(B))
```

29. Enumerating oriented gene orderings

A signed permutation of length n is some ordering of the positive integers {1,2,…,n} in which each integer is then provided with either a positive or negative sign (for the sake of simplicity, we omit the positive sign). For example, =(5,−3,−2,1,4) is a signed permutation of length 5.

Given: A positive integer n 6.

Return: The total number of signed permutations of length n , followed by a list of all such permutations (you may list the signed permutations in any order).

```
n = 6

origen <- function(n) {
library(combinat)
perms <- permn(1:n)

np <- matrix(ncol = n, nrow=2^n)
```

```
for(j in 1:n) {
  np[,j] <- rep(rep(c(1,-1), each=2^(n-j)), by=2^(j-1))
}

sperm <- c()
for(i in 1:length(perms)) {
  sperm <- rbind(sperm,t(perms[[i]]*t(np)))
}
write(nrow(sperm), "output.txt")
write.table(sperm, "output.txt", col.names = FALSE, row.names = FALSE, append = TRUE) }

system.time(origen(n),gcFirst = FALSE)
```

Alternatively

```
#slow....
n = 3
signed_perm2.0 = function(n){
    el =c(seq(n),-seq(n))

    res = matrix(nrow=1e6,ncol=n)
    for(i in 1:1e6 ) {
        res[i,]=sample(x=el,n,replace=F)
    }
    res = unique(res)
    tmp = abs(res)
    tmp = apply(tmp,1,function(x){ any(duplicated(x)) })
    res = res[-which(tmp),]
    return(res)
}
system.time(signed_perm2.0(3), gcFirst = FALSE)
```

30. Finding a spliced motif

A subsequence of a string is a collection of symbols contained in order (though not necessarily contiguously) in the string (e.g., ACG is a subsequence of TATGCTAAGATC). The indices of a subsequence are the positions in the string at which the symbols of the subsequence appear; thus, the indices of ACG in TATGCTAAGATC can be represented by (2, 5, 9).

As a substring can have multiple locations, a subsequence can have multiple collections of indices, and the same index can be reused in more than one appearance of the subsequence; for example, ACG is a subsequence of AACCGGTT in 8 different ways.

Given: Two DNA strings s and t (each of length at most 1 kbp) in FASTA format.

Return: One collection of indices of s in which the symbols of t appear as a subsequence of s. If multiple solutions exist, you may return any one.

```
library(seqinr)
sseq <- read.fasta("rosalind_sseq.txt")
dna <- sseq[[1]]
subs <- sseq[[2]]
ind <- c()
for(i in 1:length(subs)) {
```

```
  ix <- which(subs[i]==dna)
  if(is.null(ind) ) {
    ind[i] <- min(ix)
  } else {
    ix2 <- which(which(subs[i]==dna) > ind[i-1])
    ind[i] <- ix[min(ix2)]
  }
}
print(ind, quote = FALSE)
write(ind, "output.txt", ncolumns = length(ind))
```

Alternatively

```
library(seqinr)
#import file
file <- read.fasta("rosalind_sseq.txt")
myseq <- file[1]
motif <- file[2]

#x, y, z = counters for sequence index, motif index, and index of identity for x and y.
x <- 1
y <- 1
z <- 1
motif.index <- matrix()

#tests for identity between motif and seq indices. if true, then seq index is stored in a matrix
#if not identical, then seq index is increased. loop breaks upon reaching the end of the motif.
repeat{
  if (identical(myseq[[1]][x], motif[[1]][y])){
    motif.index[z] <- x
    x <- x + 1
    y <- y + 1
    z <- z + 1
  }
  if (!(identical(motif[[1]][y], myseq[[1]][x]))){
      x <- x + 1
  }
  if (y > length(motif[[1]])){
      break
  }
}
```

Problem descriptions sourced from rosalind.info