Rosalind problems 31-40

Erik Sundberg

7/1/2019

Contents

---

31. Transitions and transversions

For DNA strings s_1 and s_2 having the same length, their transition/transversion ratio $R(s_1,s_2)$ is the ratio of the total number of transitions to the total number of transversions, where symbol substitutions are inferred from mismatched corresponding symbols as when calculating Hamming distance (see “Counting Point Mutations”).

Given: Two DNA strings s_1 and s_2 of equal length (at most 1 kbp).

Return: The transition/transversion ratio $R(s_1,s_2)$.

Solution:

```r
library(seqinr)
tran <- read.fasta("rosalind_tran.txt")
s1 <- toupper(tran[[1]])
s2 <- toupper(tran[[2]])

ti <- 0
tv <- 0
for(i in 1:length(s1)) {
  if(s1[i]=='A' & (s2[i]=='T' || s2[i]=='C')) {
    tv = tv+1
  } else if(s1[i]=='T' & (s2[i]=='A'|| s2[i]=='G')) {
    tv = tv+1
  } else if(s1[i]=='C' & (s2[i]=='A'|| s2[i]=='G')) {
    tv = tv+1
  } else if(s1[i]=='G' & (s2[i]=='T' || s2[i]=='C')) {
    tv = tv+1
  } else if((s1[i]=='C' & s2[i]=='T') || (s1[i]=='T' & s2[i]=='C')) {
    ti = ti+1
  } else if((s1[i]=='A' & s2[i]=='G') || (s1[i]=='G' & s2[i]=='A')) {
    ti = ti+1
  }
}

ratio <- as.character(signif(ti/tv, 12))
write(ratio, "output.txt")
```

Alternatively

```
#!/usr/bin/Rscript
library(seqinr);
a <- read.fasta("rosalind_tran.txt");
transit <- sum((chartr("agct","rryy",a[[1]]) == chartr("agct","rryy",a[[2]]))[!(a[[1]] == a[[2]])])
transv <- sum((chartr("agct","rryy",a[[1]]) != chartr("agct","rryy",a[[2]]))[!(a[[1]] == a[[2]])])
transit / transv
```

---

32. Completing a tree

Given: A positive integer n (n1000) and an adjacency list corresponding to a graph on nnodes that contains no cycles.

Return: The minimum number of edges that can be added to the graph to produce a tree.

Solution:

```
#scan file in working directory
hmm <- scan("rosalind_tree.txt")
n <- hmm[1] #nr of nodes
al <- hmm[-1] #adjacency list

#nr of pairs in the adjacency list:
pree = length(al)/2
#total nr of edges in a tree of n nodes:
tote = n-1
#edges to add:
adde = tote-pree

#or simply...
edad = n-1-length(al)/2
```

---

33. Catalan numbers and RNA secondary structures

In this problem, we will consider counting noncrossing perfect matchings of basepair edges. As a motivating example of how to count noncrossing perfect matchings, let $\mathrm{c_n}$ denote the number of noncrossing perfect matchings in the complete graph K_{2n}. After setting c_0=1, we can see that c_1 should equal 1 as well. As for the case of a general n, say that the nodes of K_{2n} are labeled with the positive integers from 1 to 2n. We can join node 1 to any of the remaining 2n−1 nodes; yet once we have chosen this node (say m), we cannot add another edge to the matching that crosses the edge {1,m}. As a result, we must match all the edges on one side of {1,m} to each other. This requirement forces m to be even, so that we can write m=2k for some positive integer k.

There are 2k−2 nodes on one side of {1,m} and 2n−2k nodes on the other side of {1,m}, so that in turn there will be c_{k-1} c_{n-k} different ways of forming a perfect matching on the remaining nodes of K_{2n}. If we let m vary over all possible n−1 choices of even numbers between 1 and 2n, then we obtain the recurrence relation $c_n = \sum_{n=1}^{k} c_{k1} * c_{nk}$. The resulting numbers c_n counting noncrossing perfect matchings in K_{2n} are called the Catalan numbers, and they appear in a huge number of other settings. See Figure 4 for an illustration counting the first four Catalan numbers.

Given: An RNA string s having the same number of occurrences of 'A' as 'U' and the same number of occurrences of 'C' as 'G'. The length of the string is at most 300 bp.

Return: The total number of noncrossing perfect matchings of basepair edges in the bonding graph of s, modulo 1,000,000.

Solution:

```r
solve <- function(rna) {

    #An input RNA consisting of {A, U, C, G}
    #The number of non-overlapping perfect
    #matchings.

    return(helper(rna, 1, nchar(rna), list() ))
}

helper <- function(rna, lo, hi, dp) {

    mapping <- vector(mode="list", length=4)
    names(mapping) <- c("A", "U", "G", "C")
    mapping[[1]] <- "U"; mapping[[2]] <- "A"; mapping[[3]] <- "C"; mapping[[4]] <- "G"

    characters = hi - lo + 1

    # if there are an odd number of nucleotides,
    # this is an invalid matching.
    if(characters %% 2 == 1) {
        return(0)
    }

    # handles tricky edge cases.
    if(lo >= hi || lo >= nchar(rna) || hi < 0) {
        return(1)
    }

    # return answer if it is memoized.
    if( all(c(lo, hi) %in% dp) ) {
        return(dp[c(lo, hi)])
    }
    else {
        curr = unlist(strsplit(rna,""))[lo]
        target = mapping[[curr]]
        acc = 0
        for(i in seq( (lo + 1), (hi + 1), 2) ) {
            if(unlist(strsplit(rna, "") )[i] == target) {
                left = helper(rna, (lo + 1), (i - 1), dp)
                right = helper(rna, (i + 1), hi, dp)
                acc = (acc+(left * right) ) %% 1000000
            }
        dp[[lo]] = acc
        dp[[hi]] = acc
        return(acc)
        }
    }
}

rna = "CAUAUG"
solve(rna)

print(solve(trimws(rna)) %% 1000000)
```

Alternatively, in python

```
#python code
def solve(rna):
    """
    An input RNA consisting of {A, U, C, G}
    The number of non-overlapping perfect
    matchings.
    """
    return helper(rna, 0, len(rna) - 1, {})


def helper(rna, lo, hi, dp):

    mapping = {
    "A" : "U",
    "U" : "A",
    "G" : "C",
    "C" : "G"
    }
    characters = hi - lo + 1

    # if there are an odd number of nucleotides,
    # this is an invalid matching.
    if characters % 2 == 1:
        return 0

    # handles tricky edge cases.
    if lo >= hi or lo >= len(rna) or hi < 0:
        return 1

    # return answer if it is memoized.
    if (lo, hi) in dp:
        return dp[(lo, hi)]
    else:
        curr = rna[lo]
        target = mapping[curr]
        acc = 0
        for i in xrange(lo + 1, hi + 1, 2):
            if rna[i] == target:
                left = helper(rna, lo + 1, i - 1, dp)
                right = helper(rna, i + 1, hi, dp)
                acc += (left * right) % 1000000
        dp[(lo, hi)] = acc
        return acc


rna = "UGUUCGCGCCGGAGGACGAUUAUGCCGCGCUACGGCUCAUUAGAUACAGCAUUACCAACGUUCGGGCCUCGACUUAAUGCAGAUCGCCGGCGGUGC

print solve(rna.strip()) % 1000000
```

---

34. Error correction in reads

As is the case with point mutations, the most common type of sequencing error occurs when a single nucleotide from a read is interpreted incorrectly.

Given: A collection of up to 1000 reads of equal length (at most 50 bp) in FASTA format. Some of these reads were generated with a single-nucleotide error. For each read s in the dataset, one of the following applies:

- *s* was correctly sequenced and appears in the dataset at least twice (possibly as a reverse complement);
- *s* is incorrect, it appears in the dataset exactly once, and its Hamming distance is 1 with respect to exactly one correct read in the dataset (or its reverse complement).

Return: A list of all corrections in the form "[old read]->[new read]". (Each correction must be a single symbol substitution, and you may return the corrections in any order.)

Solution:

```
## utility functions
us <- function(..., sep='') { unlist( strsplit( ..., sep ) ) }

reverse_compliment <- function( sequence ) {
  nucs <- c("A", "C", "G", "T")
  return( paste( rev( kReplace( us(sequence), nucs, rev(nucs) ) ), collapse="" ) )
}

kReplace <- function( vec, orig, out=names(orig) ) {
  tmp <- out[ match( vec, orig ) ]
  tmp[ is.na(tmp) ] <- vec[ is.na(tmp) ]
  tmp
}

## actual code
```

```r
dat <- scan( what=character(), "rosalind_corr.txt" )

dat_split <- strsplit( dat, "" )

dat_rc <- c(dat, sapply( dat, reverse_compliment) )
names(dat_rc) <- NULL

reads <- table(dat_rc)
good_reads <- names( reads[ reads >= 2 ] )
good_reads_list <- strsplit( good_reads, "" )

bad_reads <- reads[ reads == 1 ]
bad_reads <- names( bad_reads[ names(bad_reads) %in% dat ] )
ix = which(nchar(bad_reads)==14)
bad_reads <- bad_reads[-ix]

for( read in 1:length(bad_reads) ) {
  x <- us(bad_reads[read])
  dists <- sapply( good_reads_list, function(xx) {
    length(xx) - sum( xx == x )
  })
  if(any(dists==1)) {
  cat( paste0(
    paste(x, collapse=""),
    "->",
    paste(good_reads_list[[which(dists==1)]], collapse=""),
    "\n"
    ), file="output.txt", append = TRUE)
  }
}
```

Alternatively, in python

```python
from Bio import SeqIO
from Bio.Seq import Seq
from Bio.Alphabet import generic_dna
reads = []
handle = open('rosalind_corr.txt', 'r')
for record in SeqIO.parse(handle, 'fasta'):
    reads.append(str(record.seq))
handle.close()

right = []
wrong = []
for i, j in enumerate(reads):
    read = Seq(j, generic_dna)
    rev_read = read.reverse_complement()
    for k in range(i + 1, len(reads)):
        if read == reads[k] or rev_read == reads[k]:
            if read not in right and rev_read not in right:
                right.append(str(read))
                right.append(str(rev_read))
```

```python
for l in reads:
    if l not in right:
        wrong.append(l)

for incorrect in wrong:
    for correct in right:
        hamming = 0
        for nt1, nt2 in zip(incorrect, correct):
            if nt1 != nt2:
                hamming += 1
                if hamming > 2:
                    break
        if hamming == 1:
            with open('answer.txt', 'a') as textfile:
                print(incorrect, '->', correct, sep='', file=textfile)
```

---

35. Counting phylogenetic ancestors

A binary tree is a tree in which each node has degree equal to at most 3. The binary tree will be our main tool in the construction of phylogenies.

A rooted tree is a tree in which one node (the root) is set aside to serve as the pinnacle of the tree. A standard graph theory exercise is to verify that for any two nodes of a tree, exactly one path connects the nodes. In a rooted tree, every node v will therefore have a single parent, or the unique node w such that the path from v to the root contains {v,w}. Any other node x adjacent to v is called a child of v because v must be the parent of x; note that a node may have multiple children. In other words, a rooted tree possesses an ordered hierarchy from the root down to its leaves, and as a result, we may often view a rooted tree with undirected edges as a directed graph in which each edge is oriented from parent to child. We should already be familiar with this idea; it's how the Rosalind problem tree works!

Even though a binary tree can include nodes having degree 2, an unrooted binary tree is defined more specifically: all internal nodes have degree 3. In turn, a rooted binary tree is such that only the root has degree 2 (all other internal nodes have degree 3).

Given: A positive integer n (3n10000).

Return: The number of internal nodes of any unrooted binary tree having n leaves.

Solution:

```python
#nr of leaves
n = 7135
#nr of internal nodes:
ino = n-2
```

---

35. k-Mer composition

For a fixed positive integer k, order all possible k-mers taken from an underlying alphabet lexicographically.

Then the k-mer composition of a string s can be represented by an array A for which A[m] denotes the number of times that the mth k-mer (with respect to the lexicographic order) appears in s.

Given: A DNA string s in FASTA format (having length at most 100 kbp).

Return: The 4-mer composition of s.

Solution:

```r
library(seqinr)
kmer = read.fasta("rosalind_kmer.txt") #file in wd
kmers = toupper(sapply(kmer, paste, collapse=""))

nt = c('A','C','G','T')
#make a matrix w all possible 4-mers:
mertrix = matrix(nrow = 4^4, ncol = 4)
for(i in 1:ncol(mertrix) ) {
  mertrix[,i] = rep(rep(nt, each=4^(4-i) ), by=4^(i-1) )
}
#put all 4-mers into a vector:
mers = c()
for(i in 1:dim(mertrix)[1]) {
mers[i] = paste(mertrix[i,1], "(?=", paste(mertrix[i,2:4],collapse=""), ")", sep="")
}
#for each 4-mer, count occurrences in the string:
merc = c()
for(i in 1:length(mers) ) {
merc[i] = length(gregexpr(mers[i], kmers, perl = TRUE)[[1]] )
  if(gregexpr(mers[i], kmers, perl = TRUE)[[1]][1] == -1) {
    merc[i] = 0
  }
}
writeLines(paste(merc, collapse=" "), "output.txt")
```

Alternatively

```r
library(seqinr)
kmer = read.fasta("rosalind_kmer.txt")
cat(unname(count(kmer[[1]],4)))
```

---

36. Speeding up motif finding

A prefix of a length n string s is a substring s[1:j]; a suffix of s is a substring s[k:n].

The failure array of s is an array P of length n for which P[k] is the length of the longest substring s[j:k] that is equal to some prefix s[1:k−j+1], where j cannot equal 1 (otherwise, P[k] would always equal k). By convention, P[1]=0.

Given: A DNA string *s* (of length at most 100 kbp) in FASTA format.

Return: The failure array of s.

Solution:

```r
start = Sys.time()
library(seqinr)
s = read.fasta("rosalind_kmp.txt")[[1]] #w file in wd
```

```
#Failure array:
P = c(); P[1] = 0 #by convention
j = 2
for(k in 2:length(s) ) {
  #Index (k) for the final letter in each motif.
  for(i in j:k) {
    #Index (i) for the first letter in each motif.
    if(all(s[i:k] == s[1:(k-i+1)]) ) {
      #Add length of longest prefix-matching motif ending at k:
      P[k] = k-i+1
      #Next motif gets read from beginning of previous matching motif (i):
      j = i
      break #Move on to next k.
    } else { P[k] = 0; j = k  } #If no prefix-match, add 0.
  }
}
writeLines(paste(P, collapse=" "), "output.txt")
Sys.time()-start
```

---

37. Finding a shared spliced motif

A string u is a common subsequence of strings s and t if the symbols of u appear in order as a subsequence of both s and t. For example, "ACTG" is a common subsequence of "AACCTTGG" and "ACACTGTGA".

Analogously to the definition of longest common substring, u is a longest common subsequence of s and t if there does not exist a longer common subsequence of the two strings. Continuing our above example, "ACCTTG" is a longest common subsequence of "AACCTTGG" and "ACACTGTGA", as is "AACTGG".

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

Return: A longest common subsequence of s and t. (If more than one solution exists, you may return any one.)

Solution:

```
begin = Sys.time()
library(seqinr)
lcsq = read.fasta("rosalind_lcsq.txt")
len = sapply(lcsq, length)
maxix = which(len == max(len))
minix = which(len == min(len))
X = toupper(lcsq[[maxix]])
Y = toupper(lcsq[[minix]])

m = length(X)+1
n = length(Y)+1

#Calculating longest LCS:
#LCSLength = function(X, Y) {
    C = matrix(ncol = length(X)+1, nrow = length(Y)+1)
    for(i in 1:(length(Y)+1) ) {
       C[i,1] = 0 }
    for(j in 1:(length(X)+1) ) {
       C[1,j] = 0 }
    for(i in 2:(length(Y)+1) ) {
        for(j in 2:(length(X)+1) ) {
            if(X[j-1] == Y[i-1]) {
                C[i,j] = (C[i-1,j-1] + 1)
            } else {
                C[i,j] = max(C[i,j-1], C[i-1,j])
            }
        }
    }
#    return(C[length(Y)+1, length(X)+1])
#}
#LCSLength(X,Y)


spliced_motif = ''
x = length(X)+1
y = length(Y)+1
while (x != 1 && y != 1) {
    if(C[y,x] == C[y - 1,x]) {
        y = y-1 }
    else if( C[y,x] == C[y, x-1] ) {
        x = x-1 }
    else {
        spliced_motif = paste(Y[y - 1], spliced_motif, sep="")
        x = x-1
        y = y-1
    }
}
cat(spliced_motif)

Sys.time() - begin
```

Alternatively, in python

```
S, T = open('rosalind_lcsq.txt').read().splitlines()
cur = [''] * (len(T) + 1) #dummy entries as per wiki
for s in S:
    last, cur = cur, ['']
    for i, t in enumerate(T):
        cur.append(last[i] + s if s==t else max(last[i+1], cur[-1], key=len))
print cur[-1]
```

---

38. Ordering strings of varying lengths lexicographically

Say that we have strings $s=s_1s_2\ s_m$ and $t=t_1t_2\ t_n$ with m<n. Consider the substring t =t[1:m]. We have two cases: 1. If s=t , then we set $s<_{\text{Lex}}t$ because s is shorter than t (e.g., APPLE<APPLET). 2. Otherwise, s t . We define $s<_{\text{Lex}}t$ if $s<_{\text{Lex}}t$ and define $s>_{\text{Lex}}t$ if $s>_{\text{Lex}}t$ (e.g., APPLET$<_{\text{Lex}}$ARTS because APPL$<_{\text{Lex}}$ARTS).

Given: A permutation of at most 12 symbols defining an ordered alphabet A and a positive integer n ($n \leq 4$).

Return: All strings of length at most n formed from A, ordered lexicographically. (Note: As in "Enumerating k-mers Lexicographically", alphabet order is based on the order in which the symbols are given.)

Solution:

```
begin = Sys.time()
lexv = readLines("rosalind_lexv.txt")
n = as.numeric(lexv[2])
chrs = unlist(strsplit(lexv[1], " "))

alf = LETTERS[1:length(chrs)]

k = 1
ord = alf
while(k < n) {
  ix = which(nchar(ord) == k)
  for(i in ix) {
    for(j in 1:length(alf)) {
      ord[length(ord)+1] = paste(ord[i], alf[j], sep="")
    }
  }
  k = k+1
}

ord2 = ord[order(ord)]
lord = chartr(paste(alf, collapse=""), paste(chrs, collapse=""), ord2)

writeLines(lord, "output.txt")
```

---

39. Maximum matchings and RNA secondary structures

A maximum matching of basepair edges will correspond to a way of forming as many base pairs as possible in an RNA string, as shown in Figure 3.

Given: An RNA string s of length at most 100.

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

Solution:

```
library(seqinr)
library(gmp)
mmch = read.fasta("rosalind_mmch.txt")
s = as.factor(mmch[[1]])
#count base that occurs most (n) and least (k), for A/U and C/G respectively
n1 = max(summary(s)[c("u","u")])
k1 = min(summary(s)[c("a","u")])
k2 = min(summary(s)[c("c","g")])
n2 = max(summary(s)[c("g","g")])

(factorialZ(n1)/factorialZ(n1-k1))*(factorialZ(n2)/factorialZ(n2-k2))
```

---

40. Creating a distance matrix

For two strings s_1 and s_2 of equal length, the p-distance between them, denoted $d_p(s_1,s_2)$, is the proportion of corresponding symbols that differ between s_1 and s_2.

For a general distance function d on n taxa $s_1,s_2,\ldots,s_n$ (taxa are often represented by genetic strings), we may encode the distances between pairs of taxa via a distance matrix D in which $D_{i,j}=d(s_i,s_j)$.

Given: A collection of n ($n \leq 10$) DNA strings $s_1,\ldots,s_n$ of equal length (at most 1 kbp). Strings are given in FASTA format.

Return: The matrix D corresponding to the p-distance d_p on the given strings. As always, note that your answer is allowed an absolute error of 0.001.

Solution:

```r
library(seqinr)
pdst = read.fasta("rosalind_pdst.txt")
D = matrix(nrow = length(pdst), ncol = length(pdst))
len = length(pdst[[1]])

for(i in 1:length(pdst) ) {
  for( j in 1:length(pdst) ) {
    D[j,i] = sum(pdst[[i]] != pdst[[j]])/len
  }
}

D = formattable(D, digits = 5, format = "f")
write.table(as.character(D), "output.txt", quote = FALSE, col.names = FALSE, row.names = FALSE)
```

Problem descriptions sourced from rosalind.info