Laura González Antiga
MCs Bioinformatics in Health Science UPF

PERL 2018-19

PERL FINAL PROJECT

Getting the Topmost Scoring Sequences from Position Weight Matrices

Introduction

For this project, the 2 below documents are passed as input to the program (separately), and the program should return a small overview of the data (in the terminal) and the list of 10000 sequences with their score on a text document.

```
AC  M00034
XX
ID  V$P53_01
XX
DE  tumor suppressor p53
XX
BF  T00671; p53; Species: Homo sapiens.
BF  T01806; p53; Species: Mus musculus.
XX
P0      A      C      G      T
01      4      0     13      0      G
02      5      0     12      0      G
03     15      0      2      0      A
04      0     17      0      0      C
05     17      0      0      0      A
06      0      0      0     17      T
07      0      0     17      0      G
08      0     13      0      4      C
09      0     17      0      0      C
10      0     17      0      0      C
11      0      0     17      0      G
12      0      0     17      0      G
13      2      0     15      0      G
14      0     17      0      0      C
15     17      0      0      0      A
16      0      0      0     17      T
17      0      0     17      0      G
18      0      2      0     15      T
19      0     13      0      4      C
20      0      7      2      7      Y
XX
//
```

```
AC  M00097
XX
ID  V$PAX6_01
XX
DE  Pax-6
XX
BF  T00681; Pax-6; Species: Mus musculus.
BF  T01122; Pax-6; Species: Homo sapiens.
XX
P0      A      C      G      T
01     15      7      6     10      N
02     21      9      3     10      N
03     10      9     10     18      N
04      8     14      9     16      N
05      3      2      4     38      T
06      2      0      1     44      T
07      3     29      1     14      C
08     40      5      1      1      A
09      3     39      0      5      C
10      1      0     44      2      G
11      1     36      7      2      C
12     23      2      1     21      W
13      1      4      0     42      T
14      2     13     26      3      G
15     40      1      6      0      A
16     14     11     15      7      N
17      2      4      3     37      T
18      1      0     20     25      K
19     13     17      9      4      N
20     14      8      4      6      N
21      4     12      3      9      N
XX
//
```

The inputs are passed as the first argument of @ARGV and the program stores the data in green in different variables. There were two main issues to solve for this project:

1. Which kind of variable was better to store the data, either the input and the output.
2. Which kind of function would allow the program to iterate each combination of nucleotides possible.

For the first issue, perl has 2-dimensional arrays (arrays of arrays) which would had allowed us to work with coordinates of each score ($array[$i][$j]). But, in

the end, as the process was developing, the best approach was to store the data in a hash of lists, were the key is, in all the hashes, was the first column of the input (the number of the row).

That decision allowed us to always keep track of the rows we were working on and it was easier to sort either rows and columns.

For the function, the answer was a function that calls itself within it. A function that, for each nucleotide of each position checks if there are nucleotides in the lower row, and, if there are, it calls itself.

But we will describe the function in detail when we get to it. Let's see the process step by step.

Throughout the function

Storing input data

As most programs, our script starts getting the data from an input. After declaring most of the program's variables, the scripts opens a FILEHANDLE which refers to the first argument on @ARGV or dies a warning (and a message for the programmer to pass the input to the $ARGV[0].
And then, with the filehandle INPUT, it stores the data:

- The header as a vector of 4 elements (the order of the nucleotides in the original data, that we will use to change the scores to letters and vice-versa)
- And all rows that start with a digit inside a hash (%data) of the form $data{$number_row} = [@row_scores]

The header is kept a part because in following steps we will use the matrix repeatedly and all "non-score rows" would be a burden. Or at least it would had required an extra step to filter those rows in each step.

Sorting loop

So, the program saves the header and the score in those two variables and prints them (just a first check, but it's always useful to know if the problem is how it stored the data). Then, the sorting loop sorts all the values (scores) of each row (key) of the hash %data decreasingly and, at the same time, stores in a different hash (%indexes) the new order of the columns.
This step is crucial because once reordered the arrays storing the scores, we won't have the relation column-letter of the header anymore and we won't have either the same order or columns in each row, so it was really important to store that order in another matrix. So, %indexes stores, for each number of row, the index from 0 to 1 corresponding to the original column position of each score.

The order vector

This step sorts the matrix decreasingly by the first element of each array and stores the key of the hash (the number of row) in that order. This vector will set the order of row processing in the function.

But, this order is a matrix order, but, as arrays in perl are indexem from 0, in this step we also store in a @positions vector the same vector but with all the values decreases in one unit.

Score-nucleotide equivalence

The last step to manipulate the input data is to change every score of every position of each row by the corresponding nucleotide.
It could seem easier to have done that at the beginning, because all columns 0 would be one nucleotide, all columns 1 would be another, and so on. But if we had done that at the beginning, we couldn't had sorted by score.

On the other hand, once sorted the scores for each row, with the score-nucleotide equivalence loop we could substitute each matrix[$i][$j], were $i and $j represent rows and columns, respectively, by the value in header corresponding to the position indicated by the %indexes matrix ($header[$indexes[$i][$j]).
This matrix will be the input of the function. And the result is a hash %matrix with the number of row as key and the nucleotides ordered by score as arrays of each key.

Executing the function

The main step of the script is the function permutator. It requires the previous matrix of nucleotides we just generated as input, alongside with the @order vector, an empty counter and the threshold (10000 as the exercise demands) and returns the threshold amount of sequences created and the $counter, which will tell us how many sequences it has return (and should be equal to the threshold)

PERMUTATOR subroutine

@It stores the matrix as %input and the @order vector, and, if the counter is smaller than the threshold it initializes the permutation.

All the function is based on the @order vector size. If the vector is > 1 it takes out the first element. As the vector contains the number of rows, ordered, each time it will take away one row. And that number of row are used as $key for the hash %input. This means that in each round of the loop, one row will be set as key for the hash, in order.

@current stores the row of nucleotides for the $key in each round of the loop. And, for each nucleotide, an internal loop runs the function again, but it passes as input the order vector, the %input matrix, the $$counter and the $ $threshold modified each row, so %input and $$threshold will remain the same, but @order and $$counter will decrease and increase by one each round, respectively.

Inside the foreach $nucleotide of @current, which calls the function again, there is another foreach, that adds that $nucleotide of the outer loop to all the strings ($string) resulting of the function.

These 2 loops work one time for each nucleotide, so, the second will add each nucleotide at the beginning of each started string. We will explain this better later, but that will mean that each step will make 4^n steps. These 2 foreach return an array @allstrings with all the possible combinations of nucleotides and the counter.

All of the previous explaining was for @order vector size>1. But it is reduced by one each round. So, when the @order vector just has one value (always corresponding to the number of the row with lower scores) it goes to the else.

The else first adds +4 to the counter, because, as mentioned before, the function will concatenate each of the four nucleotides of a row in every round.
Then it returns the reference to the @current vector and the $$counter.
As @current is a row of nucleotides of the matrix %input under the key $order[0], and else will just be performed when @order has just one value (the number of the row with lower scores), else will return the four nucleotides of the last row of the input matrix (the nucleotides matrix ordered by score) and it will concatenate them to an empty array, so the first round will always return the last row of the matrix (e.g.: (A, C, T, G)).
But, the next round, for each nucleotide of the previous row (because this function works backwards thanks to calling itself inside the loop) will concatenate it to all the previous stored strings, so, if the n-1 row would be (C, T, A, G), the second round will retunr:

(CA, CC, CT, CG, TA, TC, TT, TG, AA, AC, AT, AG, GA, GC, GT, GG)

The previous vector corresponds to all the possible combinations of 2 nucleotides, ordered by score, of the 2 last rows of the input matrix, and, as said, it hash 4^2 elements. The two foreach loops will return an array containing all the sequences obtained and the counter, so the loop will continue until the counter equals or surpasses the threshold.

<u>After the function</u>
As the function works with multiples of 4, the first thing we made was to take out all the sequences that we did not want. The counter of that is the $residue.
And, as the function returns a string, a splice is performed over the $result in order to have all sequences separated as single elements in an array (@total).

The only thing left was to calculate the score of each of the sequences generated and print them, but, as they were many, we opened another FILEHANDLE, RESULT, so that we could print there.

<u>Calculating the score</u>
The last step is, indeed, calculating the score of each sequence. We used a loop over each element of the @total array. We then separated each sequence into single nucleotides, as elements of another array, @list.
Then, for each element of each sequence (each $i in each @line) we added to an empty counter ($score), the value corresponding to the equivalent row and column position of the original matrix.

At the end of the loop, each @line_sorted will be @line sorted by @positions, which stored the order vector but with array indexes (starting by 0), and the

loop prints either the output seuqence of the function (with nucleotides ordered by highest score, and the correct ordered sequence, followed by its score.

Results and discussion

The results are the demanded amount of sequences and its score, but there is a mistake we didn’t manage to solve.

When getting the @order vector, the best approach is to order all values decreasingly, but the script works with the first (the highest of all) sorted elements. If 2 rows would have different maximum values, and the second of the row was different, it wouldn’t affect, because the best score will come from the sorted rows we get.

But if 2 maximums scores were similar, by default the others are ordered in increasing order, so, for two rows like:
a: (8,4,3,0) and b: (8,7,0,0), we would like b to come first, but the programs sets it after a.

It also calculates correctly all the scores checked by hand, but for some reason the nucleotides sometimes don’t correspond, so we deduce there might be some referencencing mistake, but we weren’t able to find it.
But it works and it should need some minor correction to work.