BioinoRmatika

1. INTRODUCTION

This is R notebook.

First, let's learn few elementary things in R programming language.

This is a chunck. Here is where we will write a lines of code which will be executed. We will execute chunks by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*. We can add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl+Alt+I*

```
# This is comment. It serves as explanation for the source code. It is not meant to be execut
ed.

my_first_variable = 13 # This is my first variable. Variables are used to store information f
or its further use. For example, numbers (integers) can be stored in variables.

my_second_variable = "D" # Letters can also be stored in variables in form of strings. String
 is composed of characters.


my_third_variable = "String"
my_fourth_variable = "13" # Numbers can also written as strings.
```

```
my_first_variable + 6
```

```
## [1] 19
```

```
#my_fourth_variable + 6 # But, then we can not perform mathematical operations, such as addit
ion, with said variables.
```

```
my_first_vector = c(1, 2, 3) # Vector is a collection of elements. We can store integers in v
ectors.

my_second_vector = c("GV", "cf", "!.") # And strings as well.

my_first_vector[2] # We can fetch vector elements by indexing (putting one or more indexes in
 brackets)
```

```
## [1] 2
```

```
# If we want to print variable's value
my_first_vector
```

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

```
# Adding new elements to the vectors:

my_first_vector = c(my_first_vector, 123)
```

2. BIOLOGICAL SEQUENCES

Next, we will focuse in type of variables and operations used for storage of biological sequences.

We have previously installed packages called Biostrings and GenomicRanges using following code:

```
#source("http://bioconductor.org/biocLite.R")
#biocLite("Biostrings")
#biocLite("GenomicRanges")
```

Biostrings defines containers and provides functions for genome sequence data and GenomicRanges handles genomic interval sets.

Next, we will load packages:

```
require(Biostrings)
```

```
## Loading required package: Biostrings
```

```
## Loading required package: BiocGenerics
```

```
## Loading required package: parallel
```

```
##
## Attaching package: 'BiocGenerics'
```

```
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
```

```
## The following objects are masked from 'package:stats':
##
##     IQR, mad, sd, var, xtabs
```

```
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, cbind, colMeans,
##     colnames, colSums, do.call, duplicated, eval, evalq, Filter,
##     Find, get, grep, grepl, intersect, is.unsorted, lapply,
##     lengths, Map, mapply, match, mget, order, paste, pmax,
##     pmax.int, pmin, pmin.int, Position, rank, rbind, Reduce,
##     rowMeans, rownames, rowSums, sapply, setdiff, sort, table,
##     tapply, union, unique, unsplit, which, which.max, which.min
```

```
## Loading required package: S4Vectors
```

```
## Loading required package: stats4
```

```
##
## Attaching package: 'S4Vectors'
```

```
## The following object is masked from 'package:base':
##
##     expand.grid
```

```
## Loading required package: IRanges
```

```
## Loading required package: XVector
```

```
##
## Attaching package: 'Biostrings'
```

```
## The following object is masked from 'package:base':
##
##     strsplit
```

```
require(GenomicRanges)
```

```
## Loading required package: GenomicRanges
```

```
## Loading required package: GenomeInfoDb
```

We can store DNA sequence in variable “my_string” as a string:

```
my_string = "CTGT"

my_string
```

```
## [1] "CTGT"
```

However, in package Biostrings we can defined object of DNA String type:

```
my_first_dna_string = DNAString("CTGTCGCGCTGCGTGCGTTGTGGGGCGTGCGTGCCCCCGCGGCG") # function is
 called DNAString

my_first_dna_string
```

```
##   44-letter "DNAString" instance
## seq: CTGTCGCGCTGCGTGCGTTGTGGGGCGTGCGTGCCCCCGCGGCG
```

We use toString function to compute string from DNA String:

```
toString(my_first_dna_string)
```

```
## [1] "CTGTCGCGCTGCGTGCGTTGTGGGGCGTGCGTGCCCCCGCGGCG"
```

Allowed symbols are letters from IUPAC code for incomplete nucleic acid specification. In addition, some other symbols are allowed. This is one of the differences between DNA Strings and other strings.

```
#my_dna_string = DNAString("QXZ")
```

If we type in following line, we can see symbols which use in allowed in DNA Strings:

```
DNA_ALPHABET
```

```
##  [1] "A" "C" "G" "T" "M" "R" "W" "S" "Y" "K" "V" "H" "D" "B" "N" "-" "+"
## [18] "."
```

Following line allows us to see symbols which use in allowed in AA Strings (protein sequences):

```
AA_ALPHABET
```

```
##  [1] "A" "R" "N" "D" "C" "Q" "E" "G" "H" "I" "L" "K" "M" "F" "P" "S" "T"
## [18] "W" "Y" "V" "U" "O" "B" "J" "Z" "X" "*" "-" "+" "."
```

Now, we can try to use additional symbols (for example “-”) in DNA Strings:

```
my_second_dna_string = DNAString("TTGAAA-CTCTGCGTCT")

my_second_dna_string
```

```
##   17-letter "DNAString" instance
## seq: TTGAAA-CTCTGCGTCT
```

In continuation, we will use function from Biostrigs package to gather information about defined DNA Strings and we will manipulate DNA Strings.

Function alphabetFrequency will provide information about composition.

```
alphabetFrequency(my_first_dna_string) # Given a biological sequence, computes the frequency
 of each letter
```

```
##  A  C  G  T  M  R  W  S  Y  K  V  H  D  B  N  -  +  .
##  0 15 20  9  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

If we define argument baseOnly as TRUE, result will include only nucleotides (A, C, G, T for DNA) (other symbols will be in “other” category). In addition, with argument as.prob, values will be written in percentages.

```
alphabetFrequency(my_first_dna_string, baseOnly = TRUE, as.prob = TRUE)
```

```
##         A         C         G         T     other
## 0.0000000 0.3409091 0.4545455 0.2045455 0.0000000
```

```
length(my_first_dna_string) # length of the sequence
```

```
## [1] 44
```

Subsetting of DNA String:

```
my_first_dna_string[1] # base at first position in the DNA String (as DNA String)
```

```
##   1-letter "DNAString" instance
## seq: C
```

```
my_first_dna_string[1:3] # from the first to the third position
```

```
##   3-letter "DNAString" instance
## seq: CTG
```

Other way:

```
# it is recomended to use subseq

subseq(my_first_dna_string, 1, 1)
```

```
##   1-letter "DNAString" instance
## seq: C
```

```
subseq(my_first_dna_string, 1, 3)
```

```
##   3-letter "DNAString" instance
## seq: CTG
```

Storage of multiple DNA Strings:

```
my_first_dna_string_set = DNAStringSet(c("ACGTAAATCTC", "GTCAATCTA", "GCTATCTCATC")) # a set
 of DNA strings
my_first_dna_string_set
```

```
##   A DNAStringSet instance of length 3
##     width seq
## [1]    11 ACGTAAATCTC
## [2]     9 GTCAATCTA
## [3]    11 GCTATCTCATC
```

Subsetting of DNA String Set:

```
my_first_dna_string_set[2] # as DNA String Set instance(of length 1)
```

```
##   A DNAStringSet instance of length 1
##     width seq
## [1]     9 GTCAATCTA
```

```
my_first_dna_string_set[[2]] # as DNA String instance
```

```
##   9-letter "DNAString" instance
## seq: GTCAATCTA
```

```
codons(my_first_dna_string_set[[2]]) #codon of the second sequence in DNA String Set
```

```
##   Views on a 9-letter DNAString subject
## subject: GTCAATCTA
## views:
##     start end width
## [1]     1   3     3 [GTC]
## [2]     4   6     3 [AAT]
## [3]     7   9     3 [CTA]
```

Basic operation with DNA Strings:

Reverse complement

```
reverseComplement(my_first_dna_string)
```

```
##   44-letter "DNAString" instance
## seq: CGCCGCGGGGGCACGCACGCCCCACAACGCACGCAGCGCGACAG
```

Task no. 1

Read in following DNA sequence as DNA String: ATGTACGTCGTCGGTAATTTCATTTAA. Determine the frequency of nucleotide A in the sequence (tip: search the information about the letterFrequency function).

```
DNA_seq_1 = DNAString("ATGTACGTCGTCGGTAATTTCATTTAA")

?letterFrequency # finding information about said function
```

```
## starting httpd help server ... done
```

```
letterFrequency(DNA_seq_1, "A", as.prob = TRUE)
```

```
##         A
## 0.2592593
```

Task no. 2

Transcribe DNA sequence from 1. task into RNA. Then, translate given sequence into protein.

```
RNA_seq_1 = RNAString(DNA_seq_1) # turns tymine (T) into uracil (U)
RNA_seq_1
```

```
##   27-letter "RNAString" instance
## seq: AUGUACGUCGUCGGUAAUUUCAUUUAA
```

However, function “translate” can operate on both DNA and RNA Strings:

```
translate(RNA_seq_1)
```

```
##   9-letter "AAString" instance
## seq: MYVVGNFI*
```

```
translate(DNA_seq_1)
```

```
##   9-letter "AAString" instance
## seq: MYVVGNFI*
```

We can try to write our own code which will performe translation of DNA sequence to protein.

In order to do that, we use information about genetic code.

```
GENETIC_CODE
```

```
## TTT TTC TTA TTG TCT TCC TCA TCG TAT TAC TAA TAG TGT TGC TGA TGG CTT CTC
## "F" "F" "L" "L" "S" "S" "S" "S" "Y" "Y" "*" "*" "C" "C" "*" "W" "L" "L"
## CTA CTG CCT CCC CCA CCG CAT CAC CAA CAG CGT CGC CGA CGG ATT ATC ATA ATG
## "L" "L" "P" "P" "P" "P" "H" "H" "Q" "Q" "R" "R" "R" "R" "I" "I" "I" "M"
## ACT ACC ACA ACG AAT AAC AAA AAG AGT AGC AGA AGG GTT GTC GTA GTG GCT GCC
## "T" "T" "T" "T" "N" "N" "K" "K" "S" "S" "R" "R" "V" "V" "V" "V" "A" "A"
## GCA GCG GAT GAC GAA GAG GGT GGC GGA GGG
## "A" "A" "D" "D" "E" "E" "G" "G" "G" "G"
## attr(,"alt_init_codons")
## [1] "TTG" "CTG"
```

```
GENETIC_CODE [["TTT"]] # here is how substract information about amino acid coded by codon TT
T
```

```
## [1] "F"
```

```
protein = ""

for(i in seq(1, length(DNA_seq_1), 3)){ # we will read DNA sequence in codons (triplets of nu
cleotides) in the loop

    codon = subseq(DNA_seq_1, i, i+2) # storage of codon (start: ith position, end: ith + 2 p
osition)
    # or codon = DNAseq[i, i + 2]

    codonString = toString(codon) # we have to tranform DNAString to string
    amino_acid = GENETIC_CODE [[codonString]] # turning codon into amino acid
    protein = paste(protein, amino_acid, sep = "") # writing amino acids into protein
}

protein
```

```
## [1] "MYVVGNFI*"
```

```
#AAString(protein)
```

3. SEARCHING FOR MOTIFS

Finding occurences of given pattern in chosen biological sequence is the next tasks.

```
matchPattern("CGC", my_first_dna_string) # function matchPattern gives positions (start, end
 and width) of the pattern (motif)
```

```
##   Views on a 44-letter DNAString subject
## subject: CTGTCGCGCTGCGTGCGTTGTGGGGCGTGCGTGCCCCCGCGGCG
## views:
##     start end width
## [1]     5   7     3 [CGC]
## [2]     7   9     3 [CGC]
## [3]    38  40     3 [CGC]
```

```
matchPattern("CGC", my_first_dna_string, max.mismatch = 1) # among other things, it is possib
le to define maximal number of mismatches
```

```
##   Views on a 44-letter DNAString subject
## subject: CTGTCGCGCTGCGTGCGTTGTGGGGCGTGCGTGCCCCCGCGGCG
## views:
##      start end width
##  [1]     5   7     3 [CGC]
##  [2]     7   9     3 [CGC]
##  [3]    10  12     3 [TGC]
##  [4]    12  14     3 [CGT]
##  [5]    14  16     3 [TGC]
##  ...   ... ...   ... ...
## [14]    36  38     3 [CCC]
## [15]    38  40     3 [CGC]
## [16]    40  42     3 [CGG]
## [17]    41  43     3 [GGC]
## [18]    43  45     3 [CG ]
```

```
countPattern("CGC", my_first_dna_string) # if we only want to know number of occurences of ch
osen pattern, we use function countPattern
```

```
## [1] 3
```

Task no. 3

Find position on the motif “GGT” in the DNA_seq_1 and its reverse. Maximal number of mismatches is 1.

```
motif = DNAString("GGT") # defining motif as DNA string

max.mismatch = 1 # defining maximal number of mismatches

fwd = matchPattern(motif, DNA_seq_1, max.mismatch = max.mismatch) # finding motif position in
 forward strand

DNAseq_rev = reverse(DNA_seq_1) # storing reverse strand

rev = matchPattern(reverse(motif), DNAseq_rev, max.mismatch = max.mismatch) # finding motif p
osition in reverse strand

complete = c(fwd, rev) # merging two results
complete
```

```
##   Views on a 27-letter DNAString subject
## subject: ATGTACGTCGTCGGTAATTTCATTTAA
## views:
##     start end width
## [1]     2   4     3 [TGT]
## [2]     6   8     3 [CGT]
## [3]     9  11     3 [CGT]
## [4]    13  15     3 [GGT]
## [5]    13  15     3 [GGT]
## [6]    17  19     3 [ATT]
## [7]    20  22     3 [TCA]
## [8]    24  26     3 [TTA]
```

Task no. 4

Find the open reading frames in the DNA sequence:
ATGCAATGGGGAAATGTTACCAGGTCCGAACTTATTGAGGTAGACAGATTTAA.

Each open reading frame starts with start codon and ends with stop. For this task, we must first find positions of start and stop codons.

```
# function for finding positions of start codon:

findStartCodons = function(sequence){
    codon = "ATG" # defining start codon
    occurences = matchPattern(codon, sequence)
    positions = start(occurences) # open reading frame starts with first position of start co
don, so we want to store only this information
    sorted_positions = sort(positions) # sort positions (in ascending order)
    return(sorted_positions ) # function output
}


# similary, function for finding positions of stop codons:

findStopCodons = function(sequence){
    codons = c("TAA", "TAG", "TGA") # defining stop codons
    stop_codon_positions = c() # defining an empty vector

    for (codon in codons){ # loop: finding position of each of the three codons
        occurences = matchPattern(codon, sequence)
        positions = start(occurences) # open reading frame ends with first position of stop c
odon, so we want to store only this information
        stop_codon_positions = c(stop_codon_positions, positions) # storing positions in vect
or called stop_codon_positions
    }

    sorted_stop_codon_positions = sort(stop_codon_positions)
    return(sorted_stop_codon_positions)
}
```

```
DNA_seq_2 = DNAString("ATGCAATGGGGAAATGTTACCAGGTCCGAACTTATTGAGGTAAGACAGATTTAA")

findStartCodons(DNA_seq_2)
```

```
## [1]  1  6 14
```

```
findStopCodons(DNA_seq_2)
```

```
## [1] 36 41 52
```

```
findORF = function(sequence){
    position_start = findStartCodons(sequence) # find positions of start codons
    position_stop = findStopCodons(sequence) # find positions of stop codons


    orfstarts = c() # vector with start positions of orfs
    orfstops = c() # vector with end positions of orfs
    orflengths = c() # vector with lengths of orfs


    if (length(position_start) >= 1 && length(position_stop) >= 1){ # if we find more than on
e start and more than one stop position

        for (start in position_start){
            start_used = FALSE # first we mark start codon as unused

            for (stop in position_stop){
                if ((stop - start)%% 3 == 0){ # finding open reading frame: length between fi
rst position of stop codon and first position of start codon devided by 3 is 0 (because codon
s are triplets of nucleotides)

                    start_used = TRUE # once we find reading frame, start is marked as used

                    orfstarts = c(orfstarts, start) # we want to store start positions of orf
s
                    orfstops = c(orfstops, stop) # end positions
                    orflengths = c(orflengths, stop - start) # and length

                }
                if(start_used == TRUE){ # if we already used the start codon, that start codo
n can not longer be used for finding open reading frames
                    break() # and we end the loop
                }
            }

        }

    return(list(orfstarts, orfstops, orflengths)) # output is a start, end and length of orfs

    }
}
```

```
findORF(DNA_seq_2)
```

```
## [[1]]
## [1]  1  6 14
##
## [[2]]
## [1] 52 36 41
##
## [[3]]
## [1] 51 30 27
```

4. THE DISTANCE BETWEEN TWO SEQUENCES

We will ilustrate distance problem on the folowing DNA sequences (s1 and s2):

```
s1 = DNAString("ACTGTACCAGAATCGCTATTAGCCCACCTTAGGCGAGTGAAATAACCAAATAAACAAGTGGTGAGGGGAATTGTCCC
CACCGTTGCGTTTATGGAGGGGGTGGAAGTGGCCACGAACTGCCAGGTGTCGCCAAACGGAAGACTTCGGGCTTTAGATCCGACTTAACTAAC
ATTTTTCCACCATGAAAGGAGCAATTCAAAGCAACGTAAGGTACTTGCCTGGCCAGGTTGATAAAAGATGCGGACGTCTGATGATGTACGATG
ATCTTGGCGAGTCAAACCCGGGGACCCCGAGCCGTGACCTAGAGATTGCAATACAGTAAGTAGCCAGGAAAGGAGGATACGATATAAATTAGG
GTCACTGTACCCGTTCCGCCTTTCTGCGGCCAAAGACCCGCACGACACATGGACGCCACAGAGGCTATTTGGACCGATGACTCAGGATCATCA
AGGGCGACGACGTTAGTCAGTTATATCTGACATTGGATATGTTATAAATAAAACTGGTAACCCACAACGATCCCGGTAGTGGGGACACTGGCC
AGGCTTCTAAGCAGATGCGAGGCACAGACACAAACCGGCCGTATGTCAGAGGCAGTACTGAAGTCTAACTTTATCCACGGCAGACGCGTTACA
TGGCAATCTTGAGCGGGGCGAAGTTAGAGACGTTAAGCTATATGAAACACACTCGGCGTAGCCAATAGCCCATCTGCCTCATAAGGATGGCTG
GTTCATTTGTAAAATACTGTATCAGGCGGGGGTAACCTCCCGCGCTCAGGTAATATAATGAGACTGGTACCCATAACACGTTTTCGTCAGTAA
TAAAAGCGCGATCATTCAAGGGGACGATAGCAGACCTTCAATGCGGAATGGTTTTGCGCCTCTAATAACTGAGAGCACTATAATAGAAGTGAG
TGTATTGTTATGCCATCCT")
```

```
s2 = DNAString("ACTGGAGCGAACTGGACATCAACCCTACTAAGGGAAGAAAATTGGAATAATCATCAGGTACTGAGAACACACGACCC
CACCGTTGAGGTTTCGACAGCTTGAATTCTTACAAGGGCTAGCCGTTGCTGGCTAGCCATTTGTCAGAGTGTCTAAGAGCAGACTAAACTACC
CTCGTTCCTATATAAACGAAGCTACTACCAGCAAGGTCCGGAACGCAACTCTCAGGATTGATGGGATGTGCACATTTCGTTTGGAGTTGCCGG
ATAGACGCTCGCAATCCTCTTCGACCACACGGAATGACGCGCCGGTCGCTCCAAATATAGTAGCCCGGGGCGGAAGAGGCGAAACTATATCAG
CTTCCGGAACAGATGTCGTTTCTATGGCGCCTTCAAAGAGTGCGACGCCATGCAATCGTACATGCGCCTAGAACTCCTGGATGAGGATCTTTA
AGAACGACTGGGTAAGCTAGATACATTTCAGTCTGGTTATGGTCTAAGTAGAACAGGTAACCCATGTACATTCAAATGTATGAGTGCCGGTCC
ATGCTTATCGGTTAACATGACCAACATCGAAACTATGGTGCGAGTTTAAGTGCAATACCAAAGGCCAAATGAGCGCACTCCATAGGCTGACCC
GCATAATAATGGTCTGGTCAAAGGATTAGAAGTAAGATGCTACGATAAGCTTCCCGCGGTCACGATGACCCTTTAGCCTCACCTTATTAAATG
GCTAAATTTTTTATAACTGCTGCGGGCAGGGGAACCCAGCCGAGCTCCGTGGATTTACCGAGGCCGCTTACAGTCACATGTTTATGTCAACAA
CTTGTGCTCGAGTATGCGAGGGCCCTATTGAATCTCGGAAATGTGGCTTGGTTGTCGACCTCTCTAGTCCCGTAGTTCTCAACTCGGAATGGA
TGGCTGAAACATACGTCCA")
```

First, we will write the function which computes Hamming distance between two sequences:

```
hamming_distance = function (sequence1, sequence2){
    dis = 0 # distance is inicialy zero

    for (i in 1:length(s1)){
        if (s1[i] != s2[i]){ # we compere the same positions of two different sequences
            dis = dis + 1 # if they are different, distance increases by 1
        }
    }
    return(dis)
}

result_ham_distance = hamming_distance(s1, s2)
result_ham_distance
```

```
## [1] 465
```

However, we can also use function stringDist from the package Biostrings to compute Hamming distance:

```
ss = DNAStringSet(list(s1, s2))
stringDist(ss, method = "hamming") # we have to define method as hamming
```

```
##     1
## 2 465
```

```
stringDist(ss, method = "levenshtein") #other comonly used method is levenshtein
```

```
##     1
## 2 422
```

Task no. 5

Find the distance between two sequences if transition are penalized by 1 and transversions by 2.

```
pu = DNAStringSet(c("A","G")) # defining purin bases
py = DNAStringSet(c("C","T")) # defining pyrimidin bases


tt_distance = function (s1, s2){
    dis = 0 # distance is incially zero
    for (i in 1:length(s1)){
        b1 = s1[i] # base at ith position in first sequence
        b2 = s2[i] # base at ith position in first sequence

        if (b1 != b2){ # if bases are differnt at the same position:

            if ( ((b1 %in% pu) && (b2 %in% pu)) || ((b1 %in% py) && (b2 %in% py))){ # if firs
t and second base is purin OR
                                                    first and second base is pyrimidin, it is tr
ansition
                dis = dis + 1 # and it is penalized by 1
            }
            else{
                dis = dis + 2 # else, it is transversion and it is penalized by 2
            }

        }
    }

    return(dis)
}

result_tt_distance = tt_distance(s1, s2)
result_tt_distance
```

```
## [1] 769
```

Global alingment is method which tries to align all bases of two sequences.

```
matrix = nucleotideSubstitutionMatrix(match = 1, mismatch = -1, baseOnly = TRUE) # we can def
ine scoring system in substitution matrix: for example, we can score matches with +1 and mism
atches with -1


# global alingment of sequences ACTCGCAC and ATAGAC

pairwiseAlignment(pattern = "ACTCGCAC", subject = "ATAGAC", gapOpening = -1, substitutionMatr
ix = matrix, type = "global") # for global alingment, we use pairwiseAlignment function (type
 = "global")
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] ACTCGCAC
## subject: [1] A-TAG-AC
## score: -6
```

```
# pattern can be multiple sequences, and subject is one sequence
```

We can also used pairwiseAlignment function to solve the 5th task. However, we must define substitution matrix with given scoring system.

```
tt_matrix = matrix(data = c(0, 2, 1, 2,
                            2, 0, 2, 1,
                            1, 2, 0, 2,
                            2, 1, 2, 0), nrow = 4, ncol = 4) # matrix is defined by the data
 and its dimentions: number of rows (nrow) and columns
                        (ncol)

rownames(tt_matrix) = c("A", "C", "G", "T")
colnames(tt_matrix) = c("A", "C", "G", "T")

tt_matrix
```

```
##   A C G T
## A 0 2 1 2
## C 2 0 2 1
## G 1 2 0 2
## T 2 1 2 0
```

```
pairwiseAlignment(s1, s2, gapOpening = 10000000, substitutionMatrix = tt_matrix, type = "glob
al") # we choose global aligment and high gap opening penatly to insure that gaps are not for
med (because we want to compare bases at the same positions)
```

```
## Global PairwiseAlignmentsSingleSubject (1 of 1)
## pattern: [1] ACTGTACCAGAATCGCTATTAGCCCACCTTAG...TAATAGAAGTGAGTGTATTGTTATGCCATCCT
## subject: [1] ACTGGAGCGAACTGGACATCAACCCTACTAAG...AACTCGGAATGGATGGCTGAAACATACGTCCA
## score: 769
```

The score is the same as in previous solution.

5. GENOMIC RANGES

Biological sequence can also be presented in ranges. Biological question can reflect range-based quaeries. For example, we can propose a question which of given motive is the closest to the expressed gene.

We can diferenetiate IRanges and GRanges.

Each range has start and end coordinates and presents closed interval.

Here are few examples of how to define object of IRanges type.

```
ir1 = IRanges(start = 1:10, width = 10:1) # here we define ranges with the start coordinates
 and with the width
ir1
```

```
## IRanges object with 10 ranges and 0 metadata columns:
##            start       end     width
##        <integer> <integer> <integer>
##    [1]         1        10        10
##    [2]         2        10         9
##    [3]         3        10         8
##    [4]         4        10         7
##    [5]         5        10         6
##    [6]         6        10         5
##    [7]         7        10         4
##    [8]         8        10         3
##    [9]         9        10         2
##   [10]        10        10         1
```

```
ir2 = IRanges(start = c(1, 25), width = 3) # one width can also be defined for several ranges
ir2
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         3         3
##   [2]        25        27         3
```

```
ir3 = IRanges(start = c(1, 1, 4, 10), end = c(6, 3, 8, 10)) # here we define ranges with the
 start and end coordinates
ir3
```

```
## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         6         6
##   [2]         1         3         3
##   [3]         4         8         5
##   [4]        10        10         1
```

Now, we will perform several operation on the range ir3. Following operations represent inter-interval operations.

```
reduce(ir3, min.gapwidth = 1) # merging redundant ranges
```

```
## IRanges object with 2 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         8         8
##   [2]        10        10         1
```

```
disjoin(ir3) # fragmenting into the widest ranges where the set of overlapping ranges is the
 same
```

```
## IRanges object with 4 ranges and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         1         3         3
##   [2]         4         6         3
##   [3]         7         8         2
##   [4]        10        10         1
```

```
gaps(ir3) # ranges that fall between the ranges in the input
```

```
## IRanges object with 1 range and 0 metadata columns:
##           start       end     width
##       <integer> <integer> <integer>
##   [1]         9         9         1
```

GRanges are used to store genomic intervals. Genomic coordinates consist of chromosome(sequence name), position, and potentially strand information.

```
gr1 = GRanges(c("seq1", "seq2", "seq3"),
              IRanges(start = c(15, 18, 1233), width = 2), # we define ranges with IRanges (s
tart and width is defined in this case)
              strand = c("-", "+", "+"))

gr1
```

```
## GRanges object with 3 ranges and 0 metadata columns:
##       seqnames       ranges strand
##          <Rle>    <IRanges>  <Rle>
##   [1]     seq1 [  15,   16]      -
##   [2]     seq2 [  18,   19]      +
##   [3]     seq3 [1233, 1234]      +
##   -------
##   seqinfo: 3 sequences from an unspecified genome; no seqlengths
```

Task no. 6

Find the number and position of occurrences of the patern in “AGTGCT” in genome of the species E. coli.

BSgenome and other genome data packages provide full genome sequences for many species.

```
#source("http://bioconductor.org/biocLite.R")
#biocLite("BSgenome.Ecoli.NCBI.20080805") # load the genome data package
require(BSgenome.Ecoli.NCBI.20080805)
```

```
## Loading required package: BSgenome.Ecoli.NCBI.20080805
```

```
## Loading required package: BSgenome
```

```
## Loading required package: rtracklayer
```

```
Ecoli_genome = Ecoli$NC_000913 # we will use strain K-12 substrain MG1655 (defined by the num
ver NC_000913) at store it in variable Ecoli_genome

Ecoli_genome
```

```
##   4639675-letter "DNAString" instance
## seq: AGCTTTTCATTCTGACTGCAACGGGCAATATGTC...ACCAAATAAAAAACGCCTTAGTAAGTATTTTTC
```

```
freqPattern = countPattern("AGTGCT", Ecoli_genome) # number of occurrences
freqPattern
```

```
## [1] 864
```

```
positionPattern = matchPattern("AGTGCT", Ecoli_genome) # with matchPattern we will get positi
on of the pattern

positionPattern
```

```
##   Views on a 4639675-letter DNAString subject
## subject: AGCTTTTCATTCTGACTGCAACGGGCAATATG...CAAATAAAAAACGCCTTAGTAAGTATTTTTC
## views:
##         start     end width
##   [1]    3000    3005     6 [AGTGCT]
##   [2]   12516   12521     6 [AGTGCT]
##   [3]   22090   22095     6 [AGTGCT]
##   [4]   22228   22233     6 [AGTGCT]
##   [5]   26242   26247     6 [AGTGCT]
##   ...     ...     ...   ... ...
## [860] 4602891 4602896     6 [AGTGCT]
## [861] 4603172 4603177     6 [AGTGCT]
## [862] 4624627 4624632     6 [AGTGCT]
## [863] 4629358 4629363     6 [AGTGCT]
## [864] 4631517 4631522     6 [AGTGCT]
```