IBS-3

Assignment 3

KAUSHIK.M
CB.EN.U4AIE19036

DATE:28.08.2020

Julia
GATTTTCCCACGGCCACGCCCGGATATGAGGTAATTTTGGGCGGTTGCAAATAAA
ATTAGGACATGGTGGCG

Q1) Write a code to create a k-mer composition for the given nucleotide sequence?

CODE:

```
#Function to find all the possible kmers from the string
function pattern(String, k)
    #Creating a list to handle kmers obtained
    dict1 = Dict();
    idx = 1;
    #Pushing them into the list
    for i in 1:length(String)-k+1
        dict1[idx] = String[i:i+k-1];
        idx += 1;
    end
    #Returning the deictionary
    return dict1;
end
```

```
pattern (generic function with 1 method)
```

For k = 3:

```
#Storing the list and dictionary obtained from the function
k = 3;
String = "GATTTTCCCACGGCCACGCCCGGATATGAGGTAATTTTGGGCGGTTGCAAATAAAATTAGGACATGGTGGCG";
kmer1 = pattern(String,k)
```

```
Dict{Any,Any} with 70 entries:
  68 => "TGG"
  2  => "ATT"
  11 => "CGG"
  39 => "GGG"
  46 => "TGC"
  25 => "TAT"
  55 => "AAT"
  42 => "CGG"
  29 => "AGG"
  58 => "TAG"
  66 => "GGT"
  59 => "AGG"
  8  => "CCA"
  57 => "TTA"
  20 => "CCG"
  14 => "CCA"
  31 => "GTA"
  70 => "GCG"
  33 => "AAT"
  18 => "GCC"
  52 => "TAA"
  69 => "GGC"
  26 => "ATG"
  35 => "TTT"
  17 => "CGC"
  ⋮  => ⋮
```

For k = 4:

```
#Storing the list and dictionary obtained from the function
k = 4;
String = "GATTTTCCCACGGCCACGCCCGGATATGAGGTAATTTTGGGCGGTTGCAAATAAAATTAGGACATGGTGGCG";
kmer1 = pattern(String,k)
```

```
Dict{Any,Any} with 69 entries:
  68 => "TGGC"
  2  => "ATTT"
  11 => "CGGC"
  39 => "GGGC"
  46 => "TGCA"
  25 => "TATG"
  55 => "AATT"
  42 => "CGGT"
  29 => "AGGT"
  58 => "TAGG"
  66 => "GGTG"
  59 => "AGGA"
  8  => "CCAC"
  57 => "TTAG"
  20 => "CCGG"
  14 => "CCAC"
  31 => "GTAA"
  33 => "AATT"
  18 => "GCCC"
  52 => "TAAA"
  69 => "GGCG"
  26 => "ATGA"
  35 => "TTTT"
  17 => "CGCC"
  64 => "ATGG"
  ⋮  => ⋮
```

Q2) Write a code to rearrange the k-mer composition into its lexicographic order.

CODE:

```
function lexico_order(kmer)
    sort_kmer = Dict();
    s = [];
    #Pushing all the kmers into a list for changing it to a lexicographic manner
    for(key1,val1) in kmer
        s = push!(s, val1);
    end
    #Sorting takes place in these for loops
    for i in 1:length(s)
        for j in i:length(s)
            if(cmp(s[i], s[j]) >= 0)
                #Swapping takes place
                swap = s[i];
                s[i] = s[j];
                s[j] = swap;
            end
        end
    end
    #New dictionary created based on the lexicographic order
    idx = 0;
    for i in 1:length(s)
        idx = idx + 1;
        sort_kmer[idx] = s[i];
    end
    #Returning the sorted dictionary and sorted list of kmers
    return sort_kmer, s;
end
```

```
lexico_order (generic function with 1 method)
```

```
#Calling lexico order function
dictK, K = lexico_order(kmer1);
```

For k = 3:

```
#Printing the list in lexicographic order
println(K)
```

```
Any["AAA", "AAA", "AAA", "AAT", "AAT", "AAT", "ACA", "ACG", "ACG", "AGG", "AGG", "ATA", "ATA", "ATG", "ATG", "ATT", "ATT", "AT
T", "CAA", "CAC", "CAC", "CAT", "CCA", "CCA", "CCC", "CCC", "CCG", "CGC", "CGG", "CGG", "CGG", "GAC", "GAG", "GAT", "GAT", "GC
A", "GCC", "GCC", "GCG", "GCG", "GGA", "GGA", "GGC", "GGC", "GGC", "GGG", "GGT", "GGT", "GGT", "GTA", "GTG", "GTT", "TAA", "TA
A", "TAG", "TAT", "TCC", "TGA", "TGC", "TGG", "TGG", "TGG", "TTA", "TTC", "TTG", "TTG", "TTT", "TTT", "TTT", "TTT"]
```

For k = 4:

```
#Printing the list in lexicographic order
println(K)
```

```
Any["AAAA", "AAAT", "AAAT", "AATA", "AATT", "AATT", "ACAT", "ACGC", "ACGG", "AGGA", "AGGT", "ATAA", "ATAT", "ATGA", "ATGG", "AT
TA", "ATTT", "ATTT", "CAAA", "CACG", "CACG", "CATG", "CCAC", "CCAC", "CCCA", "CCCG", "CCGG", "CGCC", "CGGA", "CGGC", "CGGT", "G
ACA", "GAGG", "GATA", "GATT", "GCAA", "GCCA", "GCCC", "GCGG", "GGAC", "GGAT", "GGCC", "GGCG", "GGCG", "GGGC", "GGTA", "GGTG",
"GGTT", "GTAA", "GTGG", "GTTG", "TAAA", "TAAT", "TAGG", "TATG", "TCCC", "TGAG", "TGCA", "TGGC", "TGGG", "TGGT", "TTAG", "TTCC",
"TTGC", "TTGG", "TTTC", "TTTG", "TTTT", "TTTT"]
```

Q3) Write a code to obtain the original nucleotide sequence.

For k = 3:

```
#Function to create adjacency matrix (a directed graph)
function create_AdjMat(kmer)
    kmer1 = copy(kmer);
    Adj = zeros(length(kmer1),length(kmer1));
    for (key1,val1) in kmer1
        for (key2,val2) in kmer1
            if(cmp(val1, val2) != 0 && cmp(val1[1:k-1],val2[2:k])==0)
                Adj[key1, key2] = 1;
            end
        end
    end
    return Adj;
end
```

```
create_AdjMat (generic function with 1 method)
```

```
#Function to find the starting and ending of the string
function start_end(dictK)
    start = "";
    ending = "";
    a2 = create_AdjMat(dictK);
    dict_pos = Dict();
    #Computing outdegree from adjacency matrix
    outdegree = [sum(a2[:, i] for i in 1:size(a2)[1])];
    #Computing indegree from adjacency matrix
    indegree = [sum(a2[i, :] for i in 1:size(a2)[2])];
    for i in 1:length(outdegree[1])
        if(outdegree[1][i] == minimum(outdegree[1]))
            start = string(start,dictK[i]);
        elseif(indegree[1][i] == minimum(indegree[1]))
            ending = string(ending,dictK[i]);
        end
    end
    return start,ending;
end
start, ending = start_end(dictK)
```

```
("TCC", "TTC")
```

Observation –

We see that starting and ending kmer are not matching with the original string.

For k = 4:

```
#Function to create adjacency matrix (a directed graph)
function create_AdjMat(kmer)
    kmer1 = copy(kmer);
    Adj = zeros(length(kmer1),length(kmer1));
    for (key1,val1) in kmer1
        for (key2,val2) in kmer1
            if(cmp(val1, val2) != 0 && cmp(val1[1:k-1],val2[2:k])==0)
                Adj[key1, key2] = 1;
            end
        end
    end
    return Adj;
end
```

```
create_AdjMat (generic function with 1 method)
```

```
#Function to find the starting and ending of the string
function start_end(dictK)
    start = "";
    ending = "";
    a2 = create_AdjMat(dictK);
    dict_pos = Dict();
    #Computing outdegree from adjacency matrix
    outdegree = [sum(a2[:, i] for i in 1:size(a2)[1])];
    #Computing indegree from adjacency matrix
    indegree = [sum(a2[i, :] for i in 1:size(a2)[2])];
    count1 = 0;
    count2 = 0;
    for i in 1:length(outdegree[1])
        if(outdegree[1][i] == minimum(outdegree[1]) && count1 < 1)
            start = string(start,dictK[i]);
            count1 = count1 + 1;
        elseif(indegree[1][i] == minimum(indegree[1]) && count2 < 1)
            ending = string(ending,dictK[i]);
            count2 = count2 + 1;
        end
    end
    return start,ending;
end
start, ending = start_end(dictK)
```

```
("ACAT", "ACGC")
```

Observation –

We see that starting and ending kmer are not matching with the original string.
The code is for reconstruction is continued on the next page.

```
#Function to retrieve the Prefix
function Prefix(Str)
    return Str[1:k-1];
end
#Function to retrieve the Suffix
function Suffix(Str)
    return Str[2:k];
end
```

```
Suffix (generic function with 1 method)
```

```
#Function used to order the kmers in sequence based on the prefix and suffix, also remove the starting and ending point.
function kmer_reorder(kmer)
    list = [];
    #Count used to remove the starting and the ending kmer only once
    count = 0;
    for (key1,val1) in kmer
        if(count == 1)
            list = push!(list, val1);
        elseif(val1 != start || val1!= ending)
            list = push!(list, val1);
        else
            count = count + 1;
        end
    end
    #Ordering the list based on Prefix and Suffix
    for i in 1:length(list)-1
        if((cmp(Prefix(list[i]), Prefix(list[i+1]))==0) && cmp(Prefix(list[i]),Suffix(list[i+1]))==0)
            temp = list[i];
            list[i] = list[i+1];
            list[i+1] = temp;
        end
    end
    #Returning the list
    return list;
end
#Calling and obtaining the list
ord_kmer = kmer_reorder(dictK);
```

```
#Function to reconstruct the sequence
function reconstruct(ordlist)
    #Starting appended to string
    str = string("",start);
    #Intermediate string to check for prefix with the next kmer
    inter_str = start;
    for i in 1:length(ordlist)
        for j in 1:length(ordlist)-1
            if(Prefix(ordlist[j]) == Suffix(inter_str))
                #Appending to the string
                str = string(str, ordlist[j][k]);
                #println(str);
                #Changed to lowercase inorder to reduce the repetition of a kmer
                ordlist[j] = lowercase(ordlist[j]);
                #Updating inter_str
                inter_str = str[end-k+1:end];
            end
        end
    end
    #Appending the ending kmer's last index value.
    str = string(str,ending[k]);
    #Returning the string
    return str;
end
```

```
reconstruct (generic function with 1 method)
```

For k = 3:

```
#Calling reconstruct function to retrieve the original string
String_Reconstructed = reconstruct(ord_kmer);
String_Reconstructed
```

```
"TCCCCACACGGTAATGGTGAGGATTCCAATTTGGGACGCGGCCGGTTTATAAAATAGGCGC"
```

```
#Checking whether both the strings are equal
String == String_Reconstructed
```

```
false
```

For k = 4:

```
#Calling reconstruct function to retrieve the original string
String_Reconstructed = reconstruct(ord_kmer);
String_Reconstructed
```

```
"ACATCAAACATGCCGGCGCCGACAGAGGGATAGATTGCAAGGGCGTAAGTGGGTTGTAGGTATGTCCCTGAGTGCATTAGTTCCGGACATC"
```

```
#Checking whether both the strings are equal
String == String_Reconstructed
```

```
false
```

Observation-

We see that the reconstructed kmers are not matching with the original sequence for k=3, 4. It is seen that upon increasing the value of k, at k = 9, I am able to reconstruct the original sequence. The output for the same is given below.

K-mer List:

```
#Storing the list and dictionary obtained from the function
k = 9;
kmer1 = pattern("GATTTTCCCACGGCCACGCCCGGATATGAGGTAATTTTGGGCGGTTGCAAATAAAATTAGGACATGGTGGCG",k)
```

```
Dict{Any,Any} with 64 entries:
  2  => "ATTTTCCCA"
  11 => "CGGCCACGC"
  39 => "GGGCGGTTG"
  46 => "TGCAAATAA"
  25 => "TATGAGGTA"
  55 => "AATTAGGAC"
  42 => "CGGTTGCAA"
  29 => "AGGTAATTT"
  58 => "TAGGACATG"
  59 => "AGGACATGG"
  8  => "CCACGGCCA"
  57 => "TTAGGACAT"
  20 => "CCGGATATG"
  14 => "CCACGCCCG"
  31 => "GTAATTTTG"
  33 => "AATTTTGGG"
  18 => "GCCCGGATA"
  52 => "TAAAATTAG"
  26 => "ATGAGGTAA"
  35 => "TTTTGGGCG"
  17 => "CGCCCGGAT"
  64 => "ATGGTGGCG"
  49 => "AAATAAAAT"
  44 => "GTTGCAAAT"
  4  => "TTTCCCACG"
  ⋮  => ⋮
```

K-mer Lexicographic Order:

```
#Printing the list in lexicographic order
println(K)
```

```
Any["AAAATTAGG", "AAATAAAAT", "AAATTAGGA", "AATAAAATT", "AATTAGGAC", "AATTTTGGG", "ACATGGTGG", "ACGCCCGGA", "ACGGCCACG", "AGGACATGG", "AGGTAATTT", "ATAAAATTA", "ATATGAGGT", "ATGAGGTAA", "ATGGTGGCG", "ATTAGGACA", "ATTTTCCCA", "ATTTTGGGC", "CAAATAAAA", "CACGCCCGG", "CACGGCCAC", "CATGGTGGC", "CCACGCCCG", "CCACGGCCA", "CCCACGGCC", "CCCGGATAT", "CCGGATATG", "CGCCCGGAT", "CGGATATGA", "CGGCCACGC", "CGGTTGCAA", "GACATGGTG", "GAGGTAATT", "GATATGAGG", "GATTTTCCC", "GCAAATAAA", "GCCACGCCC", "GCCCGGATA", "GCGGTTGCA", "GGACATGGT", "GGATATGAG", "GGCCACGCC", "GGCGGTTGC", "GGGCGGTTG", "GGTAATTTT", "GGTTGCAAA", "GTAATTTTG", "GTTGCAAAT", "TAAAATTAG", "TAATTTTGG", "TAGGACATG", "TATGAGGTA", "TCCCACGGC", "TGAGGTAAT", "TGCAAATAA", "TGGGCGGTT", "TTAGGACAT", "TTCCCACGG", "TTGCAAATA", "TTGGGCGGT", "TTTCCCACG", "TTTGGGCGG", "TTTTCCCAC", "TTTTGGGCG"]
```

String Reconstruction:

```
#Calling reconstruct function to retrieve the original string
String_Reconstructed = reconstruct(ord_kmer);
String_Reconstructed
```

```
"GATTTTCCCACGGCCACGCCCGGATATGAGGTAATTTTGGGCGGTTGCAAATAAAATTAGGACATGGTGGCG"
```

```
#Checking whether both the strings are equal
String == String_Reconstructed
```

```
true
```

So, for a real-life scenario, what we can do is from a sequence, we can divide the sequences into smaller windows, find the kmers, reconstruct that particular window and then reconstruct the whole string with the help of these windows. This would help us find the required kmers as well from the real genome sequences which are of millions in length.