Lab 03 String Search

For Lab 03 | String Search you will use different algorithms to search for a short DNA sequence (query) in a longer sequence. To complete the assignment read through the introductory material about Python and suffix arrays and answer the questions in this document.

Assignment

Follow the instructions in this document and answer the questions in the cell below each question. Submit your answers by uploading to gradescope: a pdf file Lab03.NETID.pdf where NETID is your NETID. To generate the pdf, first export the notebook as HTML: >Notebook, >Export to ..., >HTML. Then, open the HTML in a browser and use your browser to print to PDF.

Check to make sure all your cells have been run and the **results** displayed in the PDF file. Gradescope only accepts PDF.

Reminder, provide comments for any code you write to ensure partial credit.

Python functions

Python has a number of builtin functions that are always available. A list of them is here: https://docs.python.org/3.7/library/functions.html

Relevant to Lab03: len(), range(), sorted(), and list.sort()

len()

Any kind of Python type that has length, can be passed into this function. The return is always an integer indicating the length of the variable.

In [1]:
```
x = { 1 : 'one', 2 : 'two', 3 : 'three' }
print( len( x ) )

y = [ 5, 8, 12, 24 ]
print( len( y ) )

z = 'Hello world!'
print( len( z ) )
```

```
3
4
12
```

range()

This is not actually a function but a data type, like a string or list, that includes the start, stop and step size of a range of integer values. Input can be either:

range(stop) # assumes start = 0
range(start, stop) # assumes step = 1
range(start, stop, step)
and it returns a range which is iterable in a for loop, but can also be converted to a list, see below.

In [2]:
```
print( list( range(4) ) )
```

```
[0, 1, 2, 3]
```

In [3]:
```
list( range( 5, 10 ) )
```

Out[3]:
```
[5, 6, 7, 8, 9]
```

In [4]:
```
list( range( 0, 10, 2 ) )
```

Out[4]:
```
[0, 2, 4, 6, 8]
```

In [5]:
```
for i in range(2,8,2):
    print(i)
```

```
2
4
6
```

sorted() and **list.sort()**

sorted(iterable, key=None, reverse=False) will return a new sorted list from the items in iterable. The key specifies a function of one argument that is used to extract a comparison key from each element in iterable (for example, key=str.lower). Reverse is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

In [6]:
```
x = [ '10', '4', '6', '7', '2', '1']
print( sorted(x) )
```

```
['1', '10', '2', '4', '6', '7']
```

In [7]:
```
x = [ 10, 4, 6, 7, 2, 1]
print( sorted(x) )
```

```
[1, 2, 4, 6, 7, 10]
```

In [8]:
```
x = [ 10, 4, 6, 7, 2, 1, 'abc', 'ABC']
print( sorted(x, key=str) )
```

```
[1, 10, 2, 4, 6, 7, 'ABC', 'abc']
```

Here, we need to use a str key to convert numbers to strings prior to sort since numbers can't be compared to strings. Also note that 'ABC' is before 'abc'. This is because sorted() uses the Unicode integer representations of the string, and sorts the integers.

When you sort a string, the characters are sorted using the Unicode integers and the order of these integers is listed below. Thus, 'A' comes before 'a'. And similarly 'A' < 'a'

In [9]:
```
str = ''
for i in range(32,127):
    str += chr(i)
print( str )
'A' < 'a'
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno
pqrstuvwxyz{|}~
```
Out[9]:
```
True
```

If you wish to sort a list in-place without returning a new list, you can use the list.sort() method. However, the list.sort() method is only defined for lists. In contrast, the sorted() function accepts any iterable.

In [10]:
```
x = [ '10', '4', '6', '7', '2', '1']
x.sort()
print( x )
```

```
['1', '10', '2', '4', '6', '7']
```

Why is 10 before 2? Objects may be compared to other objects with the same sequence type. The comparison uses lexicographical ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code integer to order individual characters.

Thus, '10' < '2' because '1' < '2'.

Comparisons can be done for strings, or for lists:

In [11]:
```
'GATC' == 'GATC'
```
Out[11]:
```
True
```

In [12]:
```
'GAT' < 'GATC'
```
Out[12]:
```
True
```

In [13]:
```
[1, 2, 3] < [1, 2, 4]
```
Out[13]:
```
True
```

A word of warning on scope

Don't use variable names that are already functions, methods or other keywords in Python. Keywords are highlighted by color in the code cells.

If you assign a list to a variable named 'list' it will conflict with the list class:
`list = ['a', 'b', 'c']`
or assign a string to a variable names 'print' it will conflict with the print function:
`print = 'a string'`
Python is going to give you're variable names local scope, but you will lose their global scope of the *list* class and *print* function.

String search

For this lab, we will use different data structures and search methods to find a shorter DNA sequence (substring) within a longer one (string). We will use *strings*, *dictionaries* and a suffix array to represent DNA sequences. In bioinformatics substrings are often referred to as kmers, where k is a fixed length. They can also be referred to as words, typically implying that they can have different lengths. The type of data structure and algorithm depends on whether you're looking for a fixed or variable length substring.

Strings

One reason strings make a great data structure for DNA is our ability to use substring and take sections of a string easily using the character indexes.

Here is an example of obtaining substrings with Python:

In [14]:
```
sequence = 'ATCGTTCAG'
print( sequence[3:6] )
print( sequence[0:6] )
print( sequence[:4] )
print( sequence[5:] )
```

```
GTT
ATCGTT
ATCG
TCAG
```

Lists

Lists have a number of built in methods that can be applied to them. One of these is sort(), but there are a few others that you may find useful.

If 's' is you list:
`s.append(x)` appends x to the end of the sequence
`s.insert(i, x)` inserts x into s at the index given by i
`s.pop([i])` retrieves the item at i and also removes it from s
`s.reverse()` reverse the order of the list

In [15]:
```
s = list( sequence )
s.reverse()
print( s )
```

```
['G', 'A', 'C', 'T', 'T', 'G', 'C', 'T', 'A']
```

Linear string search

A simple search algorithm is to start at the first index of an array and move through every position in the array asking at each step whether there is a match to a substring of interest. This gives us a compute time of O(n-m), where n is the length of the string and m is the length of the substring. This approach is essentially a sliding window of substring comparisons. Thus, given a substring 'ATG', we ask each each position whether we see the 'ATG' substring.

Question 1

Write a function in Python that takes two strings of DNA sequence (say string1 and string2) as input and returns a list of the position(s) where string2 is present as a substring of string1. If there are no positions an empty list should be returned. When printing positions use 1 as the first position, rather than the 0 used by Python. The output using the above string/substring should be [3, 7]. Print the output of your function to search for the three words below.

(3 points)

In [31]:
```
# Insert your code for Question 1 below
seq = 'CGTATACTAAAACGGACGTTACGATATTGTCTCACTTCATCTTACCACCCTCTATCTTATTGCTGATAGAACA
def linear(seq, sub):
    positions = []

    return( positions )
print( linear( seq,'ATG') )
print( linear( seq,'CAT') )
print( linear( seq,'TAGG') )
```

```
[]
[]
[]
```

Dictionaries

Dictionaries contain *key-value* pairs and are sometimes called “associative arrays”. They have the requirement that the keys are unique (within one dictionary). Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys and are unordered.

Membership test

The operators `in` and `not in` test for membership in dictionaries as well as lists.

In [17]:
```
bases = { 'A': 3, 'G': 0, 'C': 1, 'T': 12 }
print( bases['A'] )
'N' in bases
```

Out[17]:
```
3
False
```

In [18]:
```
bases = [12,5,6,44]
print( bases[3])
5 not in bases
```

Out[18]:
```
44
False
```

Methods and functions

There are a number of methods associated with dictionaries that facilitate their use.

`items()` returns a new view of the dictionary's items ((key, value) pairs. When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.
`keys()` returns a new view of the dictionary's keys.
`values()` returns a new view of the dictionary's values.
`get(key)` Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None.

In [19]:
```
bases = { 'A': 3, 'G': 0, 'C': 1, 'T': 12 }
for k, v in bases.items():
    print(k, v)
print( bases.keys() )
print( bases.values() )
print( bases.get('A'))
print( bases.get('N'))
```

```
A 3
G 0
C 1
T 12
dict_keys(['A', 'G', 'C', 'T'])
dict_values([3, 0, 1, 12])
3
None
```

The `enumerate(iterable, start=0)` function returns a tuple containing a count (from start which defaults to 0) and the values obtained from iterating over iterable. When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

In [20]:
```
for i, v in enumerate(bases):
```

```
    print(i, v)
```

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

Dictionaries and lists

Just as dictionaries can hold single values, they can also hold lists or even other dictionaries. The dictionary below holds numbers, sometimes only one and sometimes a list of two.

In [21]:
```
d = {
    'ATC' : [ 0, 7 ],
    'CAT' : [ 6 ],
    'CGT' : [ 2 ],
    'GTT' : [ 3 ],
    'TCA' : [ 5 ],
    'TCG' : [ 1, 8 ],
    'TTC' : [ 4 ],
}
print( d['CAT'] )
print( d['ATC'] )
d['ATC'].append(33)
print( d['ATC'] )
```

```
[6]
[0, 7]
[0, 7, 33]
```

Accessing entries in a list is done with a second set of brackets.

In [22]:
```
print( d['ATC'][2] )
```

```
33
```

Removal

To remove an item from a dictionary you can use the `del` statement. To clear a value use an empty assignment.

In [23]:
```
d['ATC'] = [ 0, 7 ]
del d['ATC']
print( d )
d['TCG'] = []
print( d )
```

```
{'CAT': [6], 'CGT': [2], 'GTT': [3], 'TCA': [5], 'TCG': [1, 8], 'TTC': [4]}
{'CAT': [6], 'CGT': [2], 'GTT': [3], 'TCA': [5], 'TCG': [], 'TTC': [4]}
```

Kmer dictionary

A dictionary is another data structure that can make string searches very convenient. The advantage of a dictionary is that while there is an initial computing cost to making the dictionary, a

kmer's position(s) can be accessed by nearly instantaneous look up. In contrast, the linear search requires time to iterate through the string for each search that is made. There is however a disadvantage of a dictionary compared to a linear search. Once a dictionary is made, e.g. kmer of length 5 or a 5mer, you can't use that dictionary to search for 6mers without also doing *kmer expansion*, or going back to the positions which match 5mers and then asking whether there is also a 6mer match. Second, the size of the dictionary grows exponentially with its length. For DNA sequences the dictionary size is 4^k, which is about 10^{12} for even just a 20mer. BLAST uses a kmer database to seed hits and then extends these to an alignment. The majority of BLAST compute time is spent in the extension of seeded hits.

Question 2

Write a function in Python that takes a DNA sequence and kmer size (integer) as input, and returns a dictionary of all kmers (keys) in the string with a list of positions as values. The positions should start at 1. Use your function to make a dictionary of the 'seq' string below and print the dictionary.

The following sequence with size = 3 should return:

```
seq = 'ATCGTTCATCG'
kmerdict(seq,3)
{'ATC': [1, 8],
 'CAT': [7],
 'CGT': [3],
 'GTT': [4],
 'TCA': [6],
 'TCG': [2, 9],
 'TTC': [5]}
```

Note that the order in the output is not important.

Use your function and the second string and print the positions of all ATGs

(3 points)

In [32]:

```
seq = 'ATCGTTCATCG'
def kmerdict(sequence, size):
    index = {}

    return index
seq2 = 'CACTTCACTCCATGGCCCATCTCTCATGAATCAGTACCAAATGCACTCACATCATTATGCACGGCACTTGCC
print( "Here are all the ATG positions in seq2: ")
```

```
Here are all the ATG positions in seq2:
```

Question 3

Find the number of times each of the following kmers are found in the lab02.fasta sequence (distributed in Lab02 as well as in the Lab03):
'A','T','G','C','TAA'

You can use either your linear search or your dictionary functions.

If DNA sequences are random, then we'd expect the frequency of the tri-nucleotide sequence 'TAA' to be equal to the product of the frequencies of 'T', 'A' and 'A'. Calculate the expected number of occurrences of 'TAA' based on single nucleotide frequencies.

(3 points)

In [33]:
```
# Your code
```

Suffix arrays

Another data structure that greatly facilitates string searches is a suffix array. A suffix array is a sorted array of all suffixes of a string whose values are the starting positions of each suffix in the original string. A suffix is any substring of a string which includes its last letter, including itself.
https://en.wikipedia.org/wiki/Suffix_array

For example, the suffixes of 'ATCGTTCAG' are:


To facilitate searching, suffix arrays are sorted.


Why use a suffix array? As you can see it's easy to find all the T's in the sequence; the last three entries show they start at positions (0-index) 5, 1 and 4. However, the same array can be used to find kmers of any length. For example TC can be seen to start at position 5 and 1. A dictionary of 1mers can't be used to search for 2mers.

There is a cost to storing the array, namely the array requires more memory than the length of the string. This is why suffix arrays only store the positions of the suffixes in the original string. If you know the position, the suffix can be extracted as a substring from the original sequence.

Thus, the suffix array of the original sequence is:


Thus, given a sorted suffix array and the original string we can get the suffixes.

In [26]:
```
s = 'ATCGTTCAG'
SA = [7, 0, 6, 2, 8, 3, 5, 1, 4]
# SA[3] starts at index 2 and so should be CGTTCAG
s[SA[3]:]
```

Out[26]:
```
'CGTTCAG'
```

Question 4

Write a Python function that will take a string as an input and return a sorted suffix array.

Your function with this input
`suffixArray('ATCGTTCAG')`
should return
`[7, 0, 6, 2, 8, 3, 5, 1, 4]`

(3 points)

In [34]:
```
def suffixArray(s):

    sa = []

    return list(sa)
suffixArray('ATCGTTCAG')
```

Out[34]:
```
[]
```

Searching for a string in a suffix array

To find all the positions of a substring in a suffix array we need to know the first and last suffix which starts with or is equal to the substring. This can be done through string comparison.

In [28]:
```
subs = 'TC'
for i in range( len(SA) ):
    lessthan = subs < s[SA[i]:]
    greaterthan = subs > s[SA[i]:]
    greaterthan2 = subs < s[SA[i]:SA[i]+2]
    greaterthanz = subs+'Z' > s[SA[i]:]
    print(i, lessthan, greaterthan, greaterthan2, greaterthanz, s[SA[i]:])
```

```
0 False True False True AG
1 False True False True ATCGTTCAG
2 False True False True CAG
3 False True False True CGTTCAG
4 False True False True G
5 False True False True GTTCAG
6 True False False True TCAG
7 True False False True TCGTTCAG
8 True False True False TTCAG
```

As you can see, index-6 is the start and index-7 (inclusive) is the end. 'lessthan' found the start but 'greaterthan' didn't find the end; in fact it found the start. This is because 'TC' > 'TCAG' is false.

Two ways to find the end were implemented.

- Ask whether 'TC' is less than just the first two positions of the suffix.
- Ask whether 'TCZ' is less than the suffix, because Z is greater than A, C, G, T.

Binary search

Iterating through a suffix array to find the first and last position that matches a substring is not very efficient and has a complexity of O(n), similar to a simple linear search. However, because the suffix array is sorted, we can use a binary search.

A binary search finds the position of a target value within a sorted array. The steps used:

- Binary search compares the target value to the middle element of the array.
- If the target is greater than the midpoint the target cannot be to the left and so the search continues on the right
- The midpoint of the right side is found, queried against the target and this is repeated until the target is found.

For example, searching for 7 in the sorted array below:

A binary search has a complexity of O(log n), compared to a linear search which is O(n).

To use a binary search with the suffix array, we need to use a version of binary search called **bisect left**. Bisect left does not require the substring exists in the array, and also finds the **leftmost** position it would be inserted.

Thus, bisectleft of 'TC' in the suffix array above should return 6 - this is where it would be inserted.

Below is a bisect left function that takes an array (A) and finds the left most position that the target (T) would be inserted.

In [29]:
```
array = [1,1,2,3,6,6,8,9]
def bisect_left(A, T):
    n = len(A)
    L = 0
    R = n
    while L < R:
        # // Floor division where the number of integers is returned with no rem
        # 9 // 2 = 4
        # 9 // 5 = 1
        # round() and divmod() could also be used
        m = ((L + R) // 2)
        if A[m] < T:
            L = m + 1
        else:
            R = m
    return L
print( bisect_left(array,2) )
print( bisect_left(array,6) )
print( bisect_left(array,10) )
```

```
2
4
8
```

Question 5

Write a function that takes a suffix array, string and substring as input and outputs all the positions (1-based) where the substring occurs in the string. You must use the suffix array and a binary search to receive full credit. Note that the bisect_left function above will not work on a suffix array, it must be modified. Check that your function is working correctly by comparing it to your linear search function and the example below.

(5 points)

In [36]:
```
sequence = 'AATATTCCGCTTTATTTTGTTGCAATTCCTAATTTTTTCATTACATTATCTTGCGAGTACGGAAGCGA
sub = 'TTT'
# Obtain suffix array
SA = suffixArray(sequence)

# Define binary search: bisect_left, bisect_right or one function that outputs b
```

Question 6

a) Suppose you need to find the positions of all 20mers in a genome. If you have already constructed the SA and dictionary, which of your functions do you think would be fastest (Suffix array, dictionary or linear search)?
b) Suppose you need to search a genome for ATG and report all positions. Which method would require the least amount of memory, including memory needed to construct any data structure used in the search?
c) Suppose you need a function that will return the position of any input string in the genome, and many different strings will be searched for. Which of your functions would be fastest?

(3 points)

Answer

Type your answer here.

In [ ]:
```
```