This assignment presents an approach to genome assembly.

Make sure you read the entire exercise and understand what you are supposed
to do before you start coding.

In genome assembly many short sequences (reads) from a sequencing machine are assembled into long sequences – ultimately chromosomes. This is done by ordering overlapping reads so they together represent genomic sequence.
For example given these three reads:

```
AGGTCGTAG
CGTAGAGCTGGGAG
GGGAGGTTGAAA
```

ordering them based on their overlap like this

```
AGGTCGTAG
    CGTAGAGCTGGGAG
             GGGAGGTTGAAA
```

produces the following genomic sequence:

```
AGGTCGTAGAGCTGGGAGGTTGAAA
```

So very briefly, the task is to read in the sequence reads, identify overlaps between them, find the right order of
reads and then reconstruct the genomic sequence from the overlapping reads.
Real genome assembly is of course more sophisticated than what we do here, but the idea is the same. Here we
make two important simplifying assumptions.

1. There are no sequencing errors implying that overlaps between reads are perfect matches.
2. No reads are nested in other reads.

The second assumption imply that overlaps are always of this type:

```
XXXXXXXXXX
        XXXXXXXXXXX
```

and never this type:

```
XXXXXXXXXXXXXX
   XXXXXXXX
```

In this exercise you will be asked to write functions that together solve the problem of assembling a genomic sequence. Some of the functions each solves a small problem, and you will use these inside other function you write, to put together solutions to larger subproblems.

The sequence reads for the mini assembly are in the genome-assembly.txt file.
The format of this file is like this:

1:ATGCG...
2:AGGCG...
3:TGAAG...

Each line represents a read. The first field on each line is the name of the read and the second field is the read sequence itself. The two filelds are separated by a ':'. So for the first line '1' is the name and 'ATGCG...' is the sequence.

Problem 1

The first task is to read and parse the input data. Write a function

```
def readDataFromFile( fileName ):
```

that takes a string fileName as argument. To read in genome-assembly.txt you need to put it in the same folder as the file with python code you are writing. The function must return a dictionary where keys are read names and values are the associated read sequences. Both keys and values must be strings.
Example usage:
readDataFromFile('genome-assembly.txt ')

should return a dictionary with the following content (maybe not with key-value pairs in that order)

```
{'1':'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC',
 '3':'GTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTG AACACATCAGT',
 '2':'CTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGATTCCAGGCTCCCCACGGG',
 '5':'CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC',
 '4':'TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCG',
 '6':'TGACAGTAGATCTCGTCCAGACCCCTAGCTGGTACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGT'
}
```

Problem 2

Often there are too many reads to look at them manually. We want to know what the mean length of the reads is.
Write a function

```
def meanLength(fileName):
```

that takes a string fileName containing the name of an input file like the one above. The function must return a float being the mean sequence length.

Problem 3

Next thing is to figure out which reads overlap each other. To do that we need a function that takes two read sequences and computes their overlap. In the input data none of the

reads are completely nested in another read. So given two reads that overlap, the 3' end of the left read overlaps the 5' end of the right read.
We know that there are no sequencing errors, so in the overlap the sequence match will be perfect. You need to loop over all possible overlaps honoring that one sequence is the left one and the other is the right one. In the for loop, start with the largest possible overlap (`min(len(left), len(right))`) and evaluate smaller and smaller overlaps until you find an exact match.

Hint: In each iteration of the loop you want to to compare the last ibases of the left sequence with the first *i* bases of the right sequence. You can slice out the latter using right[:i].

Write a function:
`def getOverlap(left,right):`

that takes two string arguments, left and right, each containing a read sequence. The function must return the overlapping sequence. If there is no overlap it should return an empty string. So with these two reads:

```
s1 = "CGATTCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC"
s2 = " GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGC "
```

`getOverlap(s1, s2)` should return `'GGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTC'` and `getOverlap(s2, s1)` should return `'C'`.

So in that case it seems that s1 and s2 overlap and that s1 is the left one and s2 is the right one. Treating s2 as the left one and s1 as the right one only gives an overlap of one base (we expect a few bases of overlap even for unrelates sequences).

Problem 4

Now that we can evaluate the overlap between two reads in some orientation we can compute overlaps between all pairs of reads in both left-right and right-left orientations.

Write a function:
`def getAllOverlaps(reads):`

that takes a dictionary argument reads as produced by readDataFromFile. The function must return a dictionary of dictionaries containing the number of overlapping bases for a pair of reads in a specific orientation. Overlap of a read to itself is meaningless and must not be included. If the overlap between read 2 in left position and 5 in right position is 21 then d['2']['5'] must be 21 (if the dictionary was called d). Make sure you understand how this data structure represents the overlaps before you go on.

Hint: d['2']will be a dictionary where keys (d['2'].keys()) are the names of reads that have an overlap with read '2' when '2' is put in the left position, and values (d['2'].values()) are the number of overlapping bases for those reads.

Example usage:

```
getA llOverlaps ( reads )
```
should return a dictionary containing (maybe not with key-value pairs in that order):

```
{'1': { '3': 0, '2': 1 , '5': 1 , '4': 0 , '6': 29} ,
'3': { '1': 0, '2': 0 , '5': 0 , '4': 1 , '6': 1} ,
'2': { '1': 13 , '3': 1, '5': 21 , '4': 0, '6': 0} ,
'5': { '1': 39 , '3': 0, '2': 1 , '4': 0, '6': 14} ,
'4': { '1': 1, '3': 1 , '2': 17 , '5': 2 , '6': 0} ,
'6': { '1': 0, '3': 43 , '2': 0 , '5': 0, '4': 1}}
```

Hint: To generate all combinations of reads you need two for-loops. One looping over reads in left positions and another (inside the first one) looping over reads in right position.

Problem 5

The dictionary returned by getAllOverlaps is a little messy to look at. We want to print it in a nice matrix-likeformat so we can better see which pairs overlap in what orientations.
Write a function

```
def prettyPrint (overlaps):
```

that takes a dictionary argument overlaps containing a dictionary of dictionaries produced by getAllOverlaps. The function should not return anything but must print a matrix exactly as shown in the example below with nicely aligned and right-justified columns. First column must hold names of reads in left conformation. The top row holds names of reads in right conformation. Remaining cells each hold the number of overlapping bases for a left-right read pair. The diagonal corresponds to overlaps to the read itself. You must put dashes in these cells.

Example usage:
```
prettyPrint ( overlaps )
```
should print exactly
```
    1   2   3   4   5   6
1   -   1   0   0   1   29
2   13  -   1   0   21  0
3   0   0   -   1   0   1
4   1   17  1   -   2   0
5   39  1   0   0   -   14
6   0   0   43   1   0  -
```

Hint: To print integer xpadded to fill three characters use string formatting like this:
print (" %3d "% x , end = ' ')

The option end='' suppresses the trailing newline that is added by default.
Notice that the overlaps are not either zero or a large number. A lot of the overlaps are 1 or 2. This is because you often find a few bases of random overlap between any two reads. So in the following we must distinguish true (significant) overlaps from random ones. We decide that true overlaps are the ones with an overlap larger than two (>2).

Problem 6

Now that we know how the reads overlap we can chain them together pair by pair from left to right to get the order in which they represent the genomic sequence. To do this we take

the first (left-most) read and identify which read has the largest overlap to its right end. Then we take that read and find the read with the largest overlap to the right end of that – and so on until we reach the rightmost (last) read.
The first thing you need to do is to identify the first (leftmost) read so we know where to start. This read is identified as the one that only has a significant (>2) overlap to its right end (it only has a good overlap when positioned to the left of other reads). In the example output from prettyPrint above the first read would be read '4' because the '4' column has no significant overlaps (no one larger than two).

Write a function
```
def findFirstRead(overlaps):
```

that takes a dictionary argument overlaps returned from getAllOverlaps. It must return a string containing the name of the first read.

Example usage: findFirstRead(overlaps) should return 4.

Problem 7

Now that we have the first read we can recursively find the correct ordering of reads. We want a list with the read names in the right order. To see how this can be solved using recursion, consider that the task can be broken into small identical tasks of finding the read that has the largest overlap to the right end of the current read (that is the recursive case). You just keep doing this until you reach the last (right-most) read that does not have any significant (>2) overlap to its right end (that is the base case).
In the recursive case you need to identify which read that is next. I.e. which read has the largest overlap to the right end of the current read. Here we use our dictionary of overlaps. If the first read is '4' then overlaps[`4'] is a dictionary of reads with overlap to the right end of read '4'. So to find the name of the read with the largest overlap you must write a function that finds the key associated with the largest value in a dictionary. We do that first:
Write a function
```
def findKey ForLargestValue ( d ) :
```

that takes a dictionary argument dand returns the key associated with the largest value. Use that function as a tool in the next function.

Write a function
```
def findOrder ( name , overlaps ) :
```
that takes a string argument name containing the name of the first read identified in problem 6, and a dictionary argument overlaps returned from getAllOverlaps. The function must return a list of read names in the order in which they represent the genomic sequence.

Hint: You know the first read is given by the name argument, you also know that you can find the next read in the chain of overlapping reads by using the findKeyForLargestValue
function, and you know that you should keep adding on reads to the chain as long as the overlap is larger than two (you can use a while loop).

Example usage: findOrder('4', overlaps) should return ['4','2','5','1','6','3'].

Make sure you understand why this is the right list of read names before you try to implement the function.

Problem 8

Now that you have the number of overlapping bases between reads and the correct order of the reads you can reconstruct the genomic sequence.
Write a function

```
def assembleGenome ( readOrder , reads , overlaps ):
```

that takes a list argument readOrder containing the order of reads, a dictionary argument reads returned from readDataFromFile and a dictionary argument overlaps returned from getAllOverlaps. The function must return a string with the genomic sequence:

```
TGCGAGGGAAGTGAAGTATTTGACCCTTTACCCGGAAGAGCGGGACGCTGCCCTGCGCGAT
TCCAGGCTCCCCACGGGGTACCCATAACTTGACAGTAGATCTCGTCCAGACCCCTAGCTGG
TACGTCTTCAGTAGAAAATTGTTTTTTTCTTCCAAGAGGTCGGAGTCGTGAACACATCAGT
```

Hint: iterate over the reads in order and use the overlap information to extract and join the appropriate parts of the reads.

Once you have all the functions you can use them, as an example, in the following way to produce the genome:

```
fileName='genome_assembly-input.txt'
reads=readDataFromFile(fileName)
print(meanLength(fileName))
overlaps=getAllOverlaps(reads)
prettyPrint(overlaps)
name=findFirstRead(overlaps)
order=findOrder(name,overlaps)
genome=assembleGenome(order,reads,overlaps)
print(genome)
```