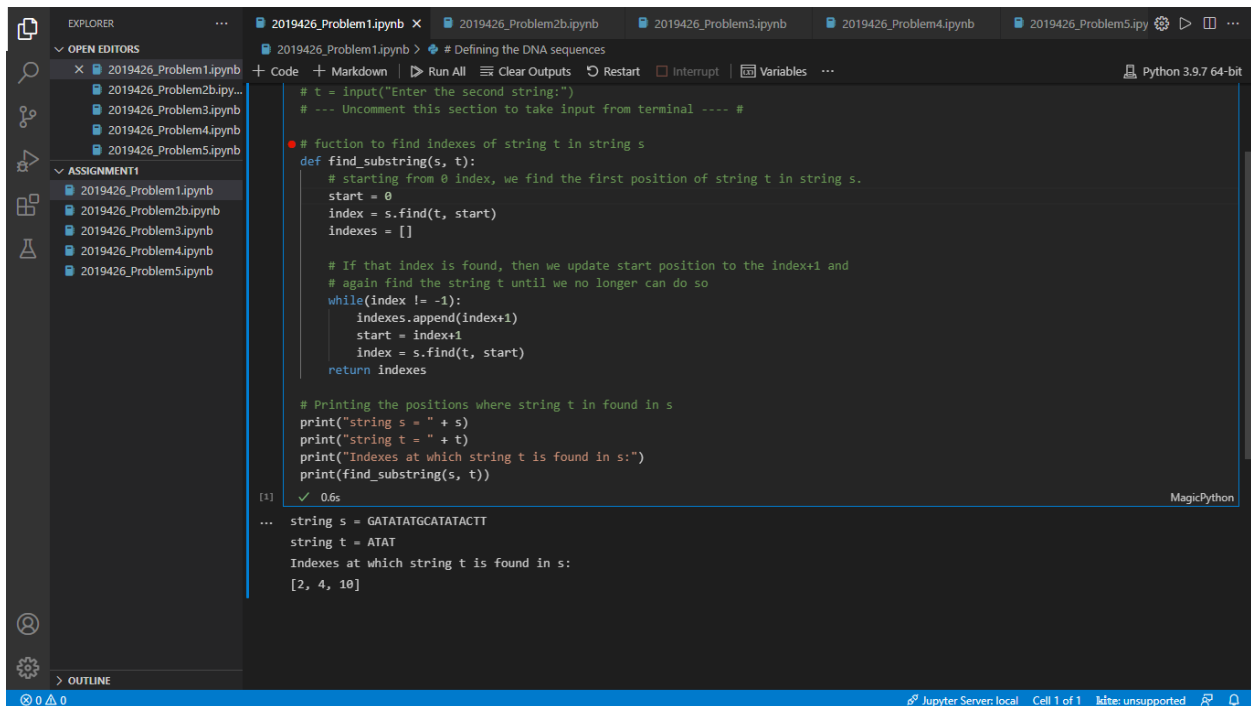


## Note:

- Each question has a commented input section at the top. Uncomment it to take variable input/terminal input.
- Each input is case-sensitive, so AA and aa will behave differently.
- Each output has lines that you can comment for long input sequences to view output properly.

## Screenshots:

### 2019426\_problem1



The screenshot shows a Jupyter Notebook environment with a dark theme. The left sidebar displays the 'EXPLORER' view with a file tree containing several Jupyter Notebook files named '2019426\_Problem1.ipynb' through '2019426\_Problem5.ipynb'. The main editor area shows the code for '2019426\_Problem1.ipynb'. The code defines a function 'find\_substring(s, t)' that finds the starting indices of a substring 't' within a string 's'. It uses a while loop to iterate through the string 's' and the 'find' method to locate the substring. The output of the code is displayed in the console, showing the string 's' as 'GATATATGCATATACTT', the string 't' as 'ATAT', and the resulting indices as '[2, 4, 10]'. The status bar at the bottom indicates 'Jupyter Server: local', 'Cell 1 of 1', and 'Idle: unsupported'.

```
# Defining the DNA sequences
# t = input("Enter the second string:")
# --- Uncomment this section to take input from terminal ---- #

# fuction to find indexes of string t in string s
def find_substring(s, t):
    # starting from 0 index, we find the first position of string t in string s.
    start = 0
    index = s.find(t, start)
    indexes = []

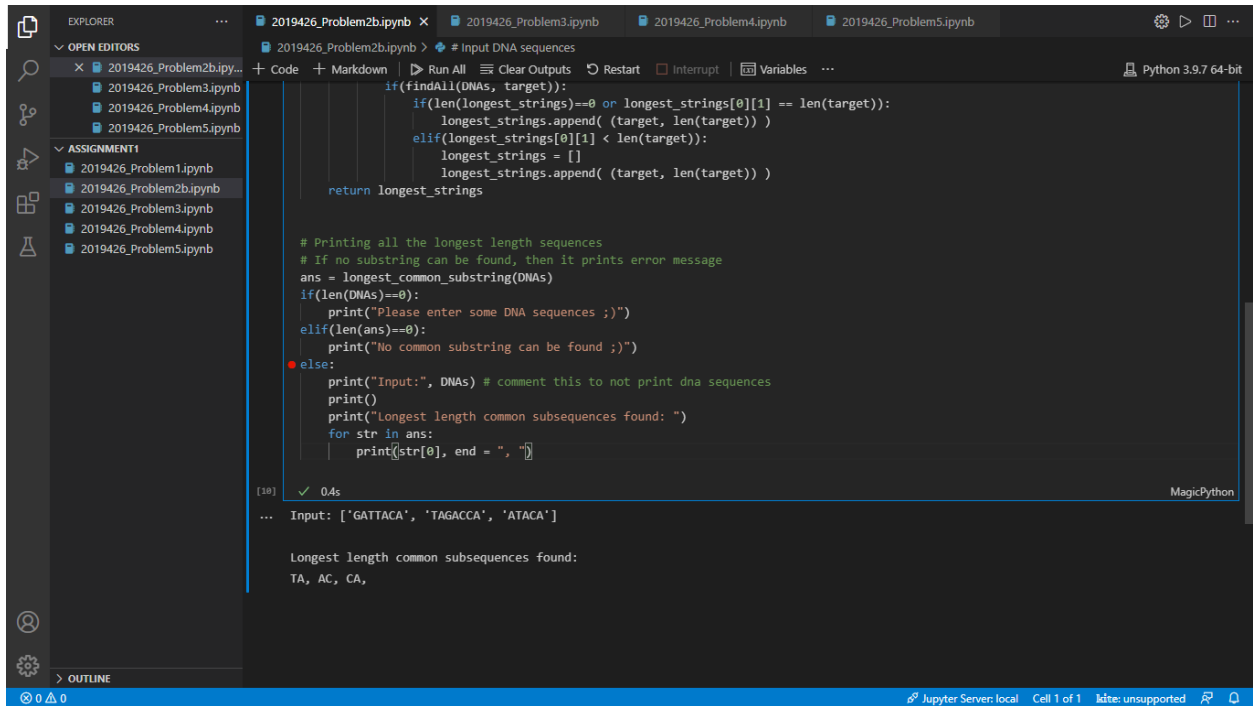
    # If that index is found, then we update start position to the index+1 and
    # again find the string t until we no longer can do so
    while(index != -1):
        indexes.append(index+1)
        start = index+1
        index = s.find(t, start)
    return indexes

# Printing the positions where string t is found in s
print("string s = " + s)
print("string t = " + t)
print("Indexes at which string t is found in s:")
print(find_substring(s, t))

[1]: ✓ 0.6s

...
string s = GATATATGCATATACTT
string t = ATAT
Indexes at which string t is found in s:
[2, 4, 10]
```

## 2019426\_problem2b



The image shows a Jupyter Notebook interface with the file explorer on the left displaying a list of files under 'ASSIGNMENT1'. The main editor shows a Python script for finding the longest common substring. The code includes a function `findAll(DNAs, target)` that returns a list of longest substrings. The main script prompts the user for DNA sequences and prints the results. The output shows the input sequences ['GATTACA', 'TAGACCA', 'ATACA'] and the longest common subsequences found: TA, AC, CA.

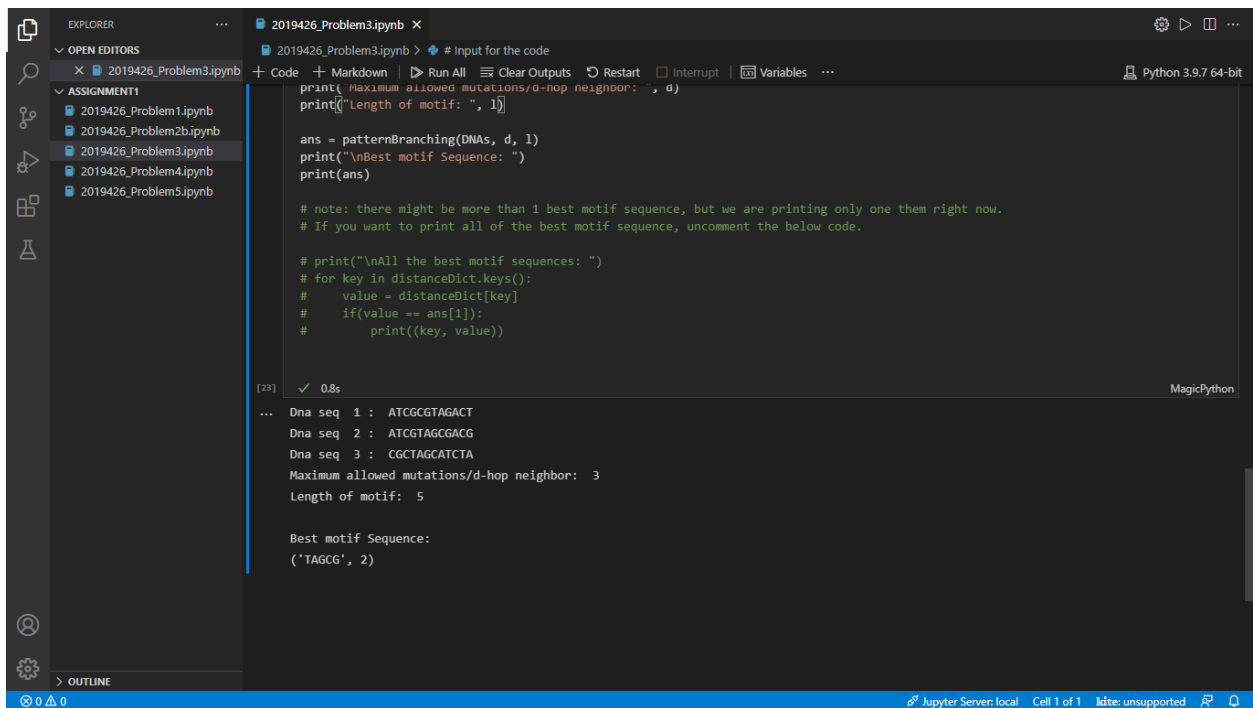
```
# Input DNA sequences
def findAll(DNAs, target):
    if(len(longest_strings)==0 or longest_strings[0][1] == len(target)):
        longest_strings.append( (target, len(target)) )
    elif(longest_strings[0][1] < len(target)):
        longest_strings = []
        longest_strings.append( (target, len(target)) )
    return longest_strings

# Printing all the longest length sequences
# If no substring can be found, then it prints error message
ans = longest_common_substring(DNAs)
if(len(DNAs)==0):
    print("Please enter some DNA sequences ;)")
elif(len(ans)==0):
    print("No common substring can be found ;)")
else:
    print("Input:", DNAs) # comment this to not print dna sequences
    print()
    print("Longest length common subsequences found: ")
    for str in ans:
        print(str[0], end = ", ")

[10]: ✓ 0.4s
... Input: ['GATTACA', 'TAGACCA', 'ATACA']

Longest length common subsequences found:
TA, AC, CA,
```

## 2019426\_problem3



The image shows a Jupyter Notebook interface with the file explorer on the left displaying a list of files under 'ASSIGNMENT1'. The main editor shows a Python script for finding the best motif sequence. The code includes a function `patternBranching(DNAs, d, l)` that returns the best motif sequence. The main script prompts the user for DNA sequences, the maximum allowed mutations/d-hop neighbor, and the length of the motif. The output shows the input sequences, the maximum allowed mutations/d-hop neighbor (3), the length of the motif (5), and the best motif sequence: ('TAGCG', 2).

```
# Input for the code
print("Maximum allowed mutations/d-hop neighbor: ", d)
print("Length of motif: ", l)

ans = patternBranching(DNAs, d, l)
print("\nBest motif Sequence: ")
print(ans)

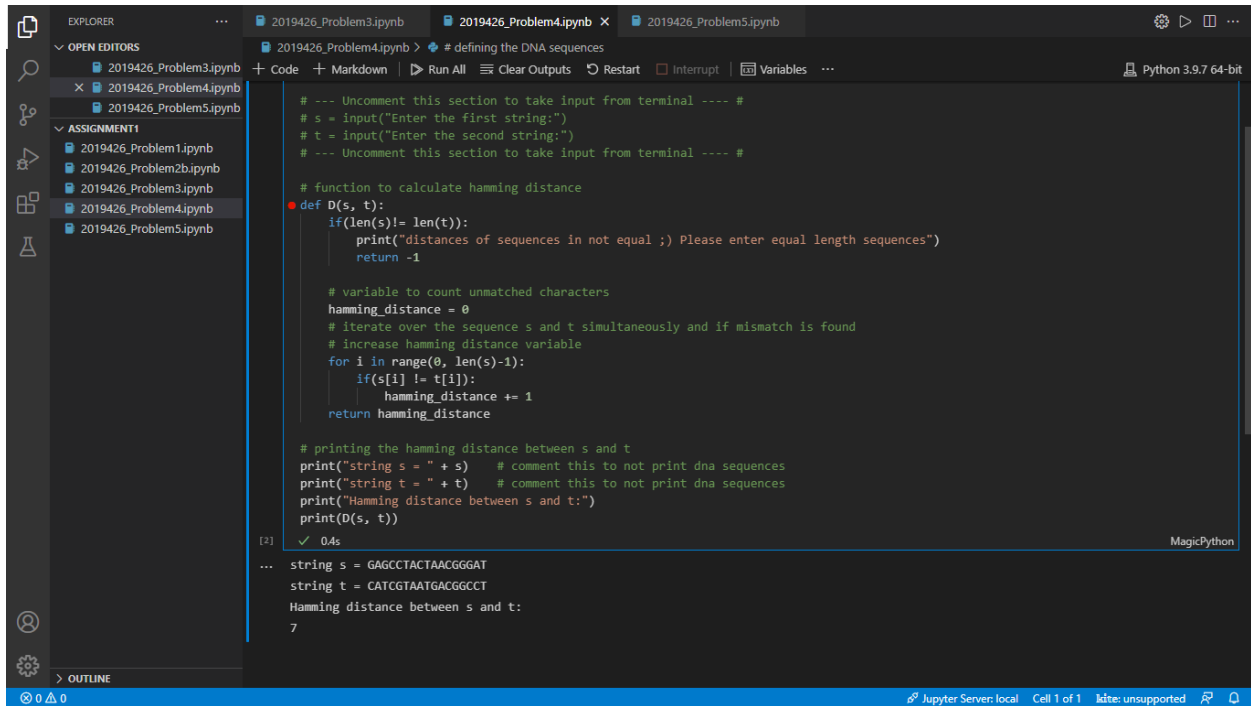
# note: there might be more than 1 best motif sequence, but we are printing only one them right now.
# If you want to print all of the best motif sequence, uncomment the below code.

# print("\nAll the best motif sequences: ")
# for key in distanceDict.keys():
#     value = distanceDict[key]
#     if(value == ans[1]):
#         print((key, value))

[23]: ✓ 0.8s
... Dna seq 1 : ATCGCGTAGACT
Dna seq 2 : ATCGTAGCGACG
Dna seq 3 : CGCTAGCATCTA
Maximum allowed mutations/d-hop neighbor: 3
Length of motif: 5

Best motif Sequence:
('TAGCG', 2)
```

## 2019426\_problem4



The image shows a Jupyter Notebook interface with a dark theme. The Explorer panel on the left shows a file tree with '2019426\_Problem4.ipynb' selected. The main editor area displays Python code for calculating the Hamming distance between two strings. The code includes comments, a function definition, and a test case. The output shows the Hamming distance between 'GAGCCTACTAACGGGAT' and 'CATCGTAATGACGGCCT' is 7.

```
# --- Uncomment this section to take input from terminal --- #
# s = input("Enter the first string:")
# t = input("Enter the second string:")
# --- Uncomment this section to take input from terminal --- #

# function to calculate hamming distance
def D(s, t):
    if(len(s)!= len(t)):
        print("distances of sequences in not equal ;) Please enter equal length sequences")
        return -1

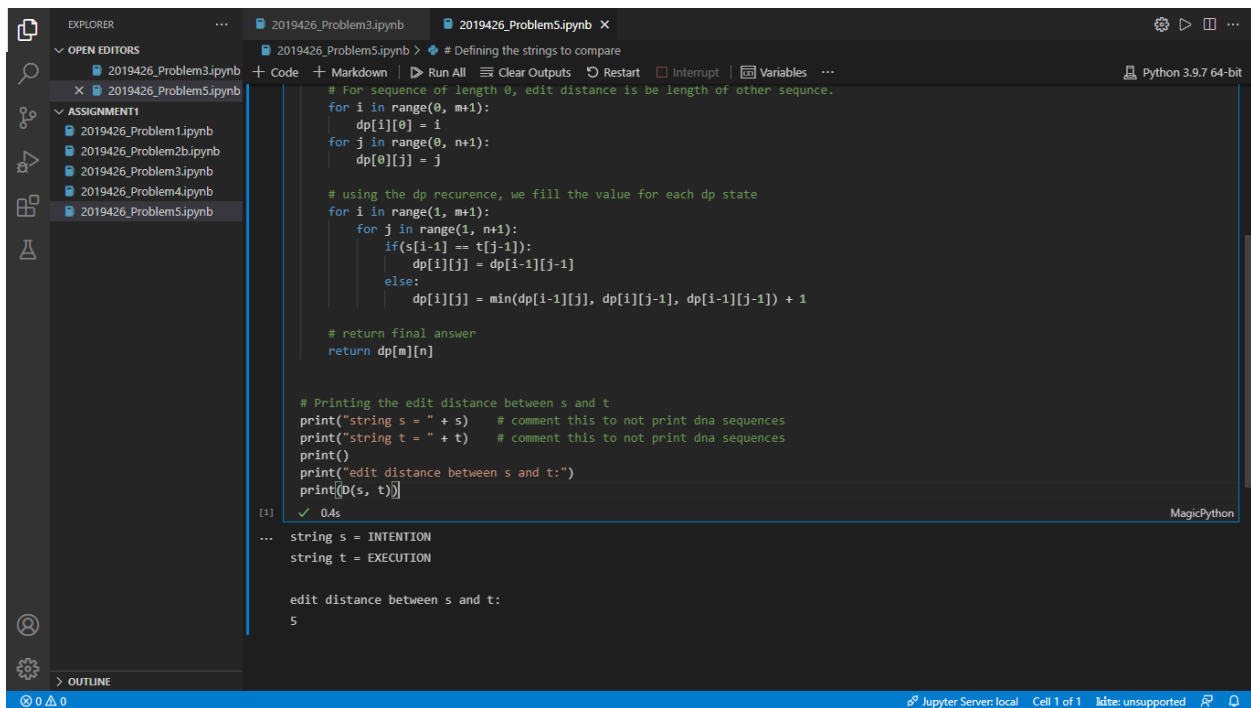
    # variable to count unmatched characters
    hamming_distance = 0
    # iterate over the sequence s and t simultaneously and if mismatch is found
    # increase hamming distance variable
    for i in range(0, len(s)-1):
        if(s[i] != t[i]):
            hamming_distance += 1
    return hamming_distance

# printing the hamming distance between s and t
print("string s = " + s) # comment this to not print dna sequences
print("string t = " + t) # comment this to not print dna sequences
print("Hamming distance between s and t:")
print(D(s, t))

[1] ✓ 0.4s

...
string s = GAGCCTACTAACGGGAT
string t = CATCGTAATGACGGCCT
Hamming distance between s and t:
7
```

## 2019426\_problem5



The image shows a Jupyter Notebook interface with a dark theme. The Explorer panel on the left shows a file tree with '2019426\_Problem5.ipynb' selected. The main editor area displays Python code for calculating the edit distance between two strings using dynamic programming. The code includes comments, a function definition, and a test case. The output shows the edit distance between 'INTENTION' and 'EXECUTION' is 5.

```
# For sequence of length 0, edit distance is be length of other sequence.
for i in range(0, m+1):
    dp[i][0] = i
for j in range(0, n+1):
    dp[0][j] = j

# using the dp recurrence, we fill the value for each dp state
for i in range(1, m+1):
    for j in range(1, n+1):
        if(s[i-1] == t[j-1]):
            dp[i][j] = dp[i-1][j-1]
        else:
            dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1

# return final answer
return dp[m][n]

# Printing the edit distance between s and t
print("string s = " + s) # comment this to not print dna sequences
print("string t = " + t) # comment this to not print dna sequences
print()
print("edit distance between s and t:")
print(D(s, t))

[1] ✓ 0.4s

...
string s = INTENTION
string t = EXECUTION

edit distance between s and t:
5
```

2019426\_problem2a

Check the attached answer on the next page.

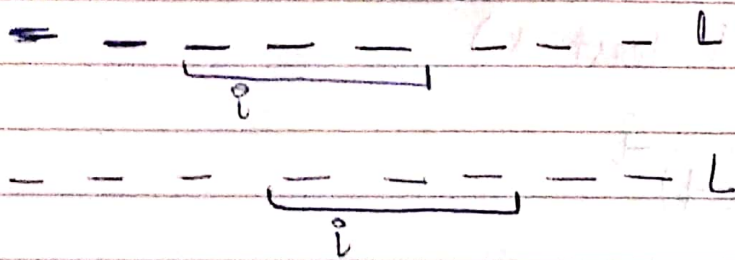
2a) Given: length of each sequence A and B = L

$P(\text{common substring of length at least } k \text{ nucleotides})$

$$= P(\text{common substring of length } k) + \dots + P(\text{common substring of length } L)$$

$$= \sum_{i=k}^{i=L} P(\text{common substring of length } i)$$

Probability of <sup>common</sup> substring of length  $i$  =



There are  $n-i+1$  sequences of length  $i$  in A and B.  
So, we have a total of  $(n-i+1) \times (n-i+1)$  combinations to choose two  $i$ -length sequences.

$$P(\text{common substring of length } i) = \text{no of ways to choose 2 strings of length } i \times P(\text{substrings of length } i \text{ to be equal})$$

$$= (n-i+1)(n-i+1) P(\text{equal substrings of length } i)$$

$$P(\text{equal substrings of length } i) = P(\text{of one pos match}) \times \text{no of pos} \\ = \left(\frac{4}{16}\right)^i \times (1) = \left(\frac{4}{16}\right)^i = \left(\frac{1}{4}\right)^i$$

# You can



# Reasoning behind last step is you have 4 bases and probability of each is  $1/4$ . Once a base is fixed in first string, you only have 1 choice for base to have same base at that position.

—  $\{A, T, G, C\}$  ~~same as~~  
 — {same as upper string}

$$P(\text{common substring of } i \text{ len}) = (n-i+1) \times (n-i+1) \times (1/4)^i$$

$$= (n-i+1)^2 (1/4)^i$$

$P(\text{common substring of length at least } k)$

$$= \sum_{i=k}^{i=L} (n-i+1)^2 (1/4)^i$$