Advanced Algorithms
Assignment 1

Index No
13440277

Email
erangaeb@gmail.com

Question 1 - Find anagrams(**anagram_finder.py**)

Program implemented with **Python** language and all the implmentation contains in **anagrams_finder.py**
In first part of the program extract the content in **word_anagram.txt** file and add the content to a hash table. Purpose of using hash table to store the text is efficient lookups. Hash table lookup complexity is **O(1)**. Below program shows how to extract the content in text file and creating the hash table.

```
1  import os.path
2  import itertools
3
4
5  # use as a hash table to keep words in the file
6  anagram_dict = {}
7
8
9  def init_anagrams(file_name):
10     """
11     Load text content in 'words_anagram.txt' to hash table, (in to python dict)
12     Purpose of storing in hash table is efficient lookup of strings
13
14     Need to iterate over each and every line of the file and extract the
15     content in to a hash table. There are no key value pairs in the text file.
16     So store this words as keys and line no or may be empty string as a value
17
18     Args:
19         file_name - anagram text file(path to text file)
20     """
21     # keep world list in here
22     anagram_list = []
23
24     if os.path.isfile(file_name):
25         # read file content to a list
26         file = open(file_name, 'r')
27         file_content = file.read()
28         anagram_list = file_content.split()
29         file.close()
30
31         # iterate over word list and add words to dict
32         for index, word in enumerate(anagram_list):
33             anagram_dict[word] = index
```

Figure 1.1

In **line 6** define the hash table that uses to store predefined anagrams. The rest of the code implemented anagram extraction process. Python provide really simple functions to read file content and extract the anagrams form the file.

After constructing the hash table, program starts to find anagram in given input text. This is the second part of this problem. Below code demonstrate how the angram finding part works.

```
def find_anagrams(text):
    """
    Find anagrams of given text and display it for clarification

    First need to identify all character combinations of given string. Then
    iterate through the combination list and check given sting is in anagram
    dictionary

    Args:
        text - input text to find anagrams
    """
    # keep anagrams and count
    anagram_count = 0
    anagram_list = []

    for word in list(map("".join, itertools.permutations(text))):
        # iterate over all combinations of given text
        if word != text:
            # we don't consider text as anagram
            if word in anagram_dict:
                anagram_count += 1
                anagram_list.append(word)

    # display anagrams
    if anagram_count > 0:
        print('text - %s' % text)
        print('anagram count - %d' % anagram_count)
        print('anagram list - %s' % anagram_list)
    else:
        print('No anagrams found')
```

Figure 1.2

In order to find anagrams, program have to have all the combinations of the input text. After getting all the combinations, program iterate through each and every combination and find the combinations which are already in the hash table. Those are the anagrams of given input text

Example

Input text - “opt”
All combinations - “otp”, “opt”, “pto”, “tpo”, “top”
Anagrams - “pot”, “top”

This is a sample output of above execution. In here initially I have entered input text as “opt” and the program finds the anagram count and angram list

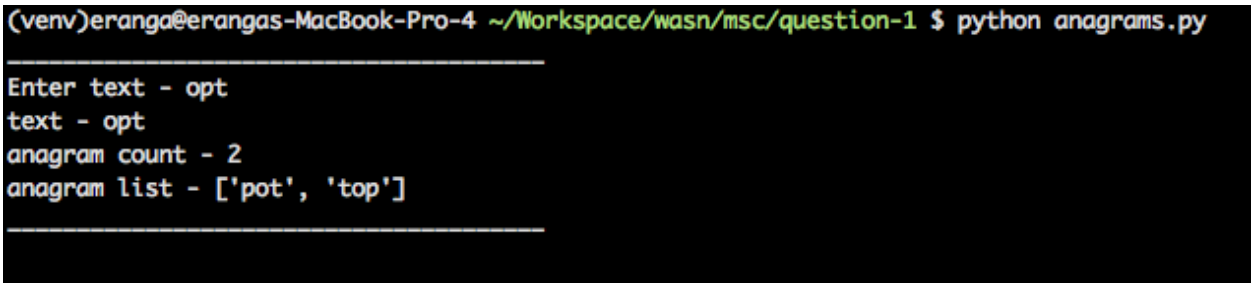

Figure 1.3

Question 2 (**pattern_matcher.py**)

Program implemented in **Python** language and all the implementation contains in **pattern_matcher.py**

I have used **Rabin-Karp** algorithm to implement the solution here. Rabin-karp algorithm use hash functions to match the text with pattern. It calls **rolling hash**(Add hash value of next input letter to already calculated hash and remove hash value of first most letter from already calculated hash). This rolling hash algorithm capable to process incoming letter in a constant amount of time. In the first part of the problem I have implemented the hashing function. Following program shows how to construct hash function of the algorithm.

```
1  #keep hash values of pattern characters
2  hash_table = {}
3
4
5  def init_hash_table(pattern):
6      """
7      Initialize content in the hash table according to the pattern texts, need
8      to iterate over pattern and add the text in to hash table. We use this hash
9      table values when matching text and pattern. Add char as the key and index
10     as the value on hash table
11
12     Args:
13         pattern - pattern to be match
14     """
15     for i in range(0, len(pattern) - 1):
16         # fill hash table
17         hash_table[pattern[i]] = i
```

Figure 2.1

After define the hash function program search the pattern in the incoming text(text growing dynamically by character by character). Below code shows how searching functionality works.

```
20 def search_pattern(text, pattern):
21     """
22     Search a given pattern in the text. Search algorithm is Rabin-Karp.
23     Advantage of using Rabin-Karp is, its capable to process incoming
24     text(characters) in a constant amount of time. Actually Rabin-Karp
25     algorithms hash value generating function should have capable for it
26 
27     Args:
28         text -  incoming text
29         pattern - matching pattern
30     """
31     print('Matching text - %s' % text)
32     print('Matching pattern - %s' % pattern)
33 
34     for i in range(0, len(text) - len(pattern) + 1):
35         # match sub string of pattern
36         if match_pattern_and_text(text[i:len(pattern) + i], pattern):
37             print('Pattern found at index %d' % i)
```

Figure 2.2

```
40 def match_pattern_and_text(text, pattern):
41     """
42     Match the pattern and text, first need to check weather the hash values of
43     pattern and text is equal(according to Rabin-Karp). If they are equal we
44     need to iterate over each and every character in text and check weather
45     text is matching to the pattern
46 
47     Args:
48         text - matching text
49         pattern - matching pattern
50 
51     Returns:
52         matching or not
53     """
54     if get_hash(text) == get_hash(pattern):
55         for i in range(0, len(text)):
56             if text[i] != pattern[i]:
57                 return False
58     else:
59         return False
60 
61     return True
```

Figure 2.3

As described in above code, program initially check the hash value of pattern and text. If both hash values are equal, it moves to further matching of pattern and text. In this way Rabin-Karp algorithm cable to do process incoming letters of the text in constant amount of time. For the preprocessing task algorithm cost **O(m)** time.

Example output

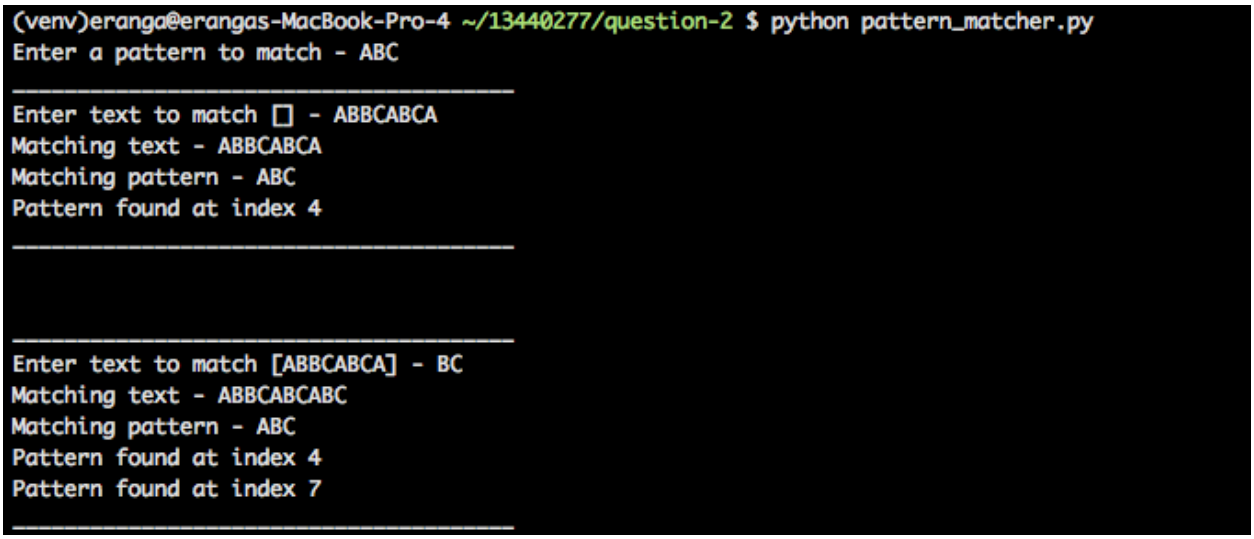

Figure 2.4

Initially program asked to enter the pattern. Entered pattern as "**ABC**". The program starts to listen for the transmitting messages(we can enter transmitting text). When program receives a text it search for the pattern in the input text according to the Rabin-Karp algorithm. In above scenario program will search the pattern "**ABC**" in "**ABBCABCABC**" and give the pattern existing indexes as the output.

When transmitting text is reversed, program needs to match the same pattern in reversed text. In here program can do it by just checking hash values of text and pattern since we already match the pattern(in original text). So program can identifies pattern in reversed text in constant amount of time.

Question 3 - DNA pattern matching (**dna_pattern_matcher.py**)

Program implemented in **Python** language and all the implementation contains in **dna_pattern_matcher.py**
I have used **Knuth-Morris-Pratt** algorithm to implement the solution since it cable to search the the pattern in a text in **O(n+m)** time.

Generate three DNA profiles

In first part of the program I have generating three text files in order to keep different DNA sequences. Following program perform that task

```
29 def init_dna():
30     """
31     Generate three DNA text string from alphabet {A,C,G,T} These texts
32     need to store in three different text file by depicting age groups.
33     Age groups are below
34         1. age >= 50
35         2. 30 >= age > 50
36         3. age < 30
37 
38     Each DNA string must contains at least 10,000 characters. Need to
39     select random item from alphabet and put it in the file
40     """
41     # we only add alphabet characters to file
42     alphabet = ['A', 'C', 'G', 'T']
43 
44     # write to 3 files
45     file1 = open(file_name1, 'w')
46     file2 = open(file_name2, 'w')
47     file3 = open(file_name3, 'w')
48 
49     for i in range(0, 10000):
50         file1.write(random.choice(alphabet))
51         file2.write(random.choice(alphabet))
52         file3.write(random.choice(alphabet))
53 
54     file1.close()
55     file2.close()
56     file3.close()
```

Figure 3.1

This program randomly select letters from alphabet**[A, C, G, T]** and add 10, 000 letters to the each and every text file. Storing file name are “**file1.txt**”, “**file2.txt**” and “**file3.txt**”

Approximating age of given DNA profile

There are three different age categories defined

- ➔ Over 50 category(contains more occurrence of pattern "**ATGGA**" in DNA)
- ➔ 30 to 50 category(contains more occurrence of pattern "**TGGAC**" in DNA)
- ➔ Below 30 category(contains more occurrence of pattern "**CCGT**" in DNA)

In order to find the age, program need to identify the no of occurrences of above patterns. According to the occurrence count program defines the age range.

```
59 def approximate_age(dna_profile):
60     """
61     Approximate the age of give DNA string by analyzing it. Following are the
62     relationships between age and patterns
63         1. over 50 category - more sequences of ATGGA occur in DNA
64         2. 30 to 50 category - more sequences of TGGAC occur in DNA
65         3. below 30 category - more sequence of CCGT occur in DNA
66 
67     Need find the count of above three patterns('ATGGA', 'TGGAC', 'CCGT'),
68     in order to identify the age
69 
70     Args:
71         dna - DNA profile
72     """
73     matchers1 = 0
74     matchers2 = 0
75     matchers3 = 0
76 
77     # iterate over text to match the first two patterns
78     for i in range(0, len(dna_profile) - len(normal_pattern1) + 1):
79         text = dna_profile[i:len(normal_pattern1) + i]
80 
81         if match_pattern_and_text(text, normal_pattern1):
82             matchers1 = matchers1 + 1
83 
84         if match_pattern_and_text(text, normal_pattern2):
85             matchers2 = matchers2 + 1
86 
87     # iterate over text to match the third pattern
88     for i in range(0, len(dna_profile) - len(normal_pattern3) + 1):
89         text = dna_profile[i:len(normal_pattern3) + i]
90 
91         if match_pattern_and_text(text, normal_pattern3):
92             matchers3 = matchers3 + 1
93 
94     # finally get the age category according to the matching count
95     return get_age_category(matchers1, matchers2, matchers3)
```

Figure 3.2

According to the above algorithm program iterate over DNA profile and find the patterns count using **Knuth-Morris-Pratt** algorithm. This pattern count can use to identify the age category of the DNA profile. **Knuth-Morris-Pratt** algorithm use special pre processing task for a pattern. it uses the knowledge of the pattern in order to avoid unnecessary shifts. Pre processing task will identifies proper suffix of the patterns. This function calls prefix function. Below program shows the implementation of prefix function.

```
4  def calculate_prefix(pattern):
5      """
6      Calculate prefix function of the pattern. the function output can be use
7      in KMP shifts.
8
9      Args:
10         pattern - matching pattern
11
12     Returns:
13         list of prefix values
14     """
15     # keep pattern in a list
16     pattern = list(pattern)
17
18     prefixes = [1] * (len(pattern) + 1)
19     shift = 1
20     for pos in range(len(pattern)):
21         while shift <= pos and pattern[pos] != pattern[pos - shift]:
22             shift += prefixes[pos - shift]
23         prefixes[pos + 1] = shift
```

Figure 3.3

<u>Replace the healthy sequence with malignant sequence</u>

The pattern replacement(defined as infecting DNA) also need to be done with age categories. Replacement logic is below

- ➔ if age > 50 replace **ATGGA** with **AATTG**
- ➔ if 30 < age < 50 replace **TGGAC** with **TTACC**
- ➔ if age < 30 replace **CCGT** with **GTTT**

In first part of the program we have calculated the age categories, these age categories can be reuse to infect the DNA. Following program shows the infecting process.

```
98  def infect_dna(dna_profile, age_category):
99      """
100     Scan DNA profile and replace the specific patterns in the text. The
101     replacements are doing according the below criteria
102         1. if age > 50 replace ATGGA with AATTG
103         2. if 30 < age < 50  replace TGGAC with TTACC
104         3. if age < 30 replace CCGT with GTTT
105
106     Args:
107         dna_profile - dna string
108         age_category - three age categories(we simplified categories as below)
109         1 - age > 50
110         2 - 30 < age < 50
111         3 - age < 30
112
113     Returns:
114         infected dna profile
115     """
116     original_pattern = normal_pattern1
117     infecting_pattern = infected_pattern1
118     if age_category == 1:
119         # replace ATGGA with AATTG
120         original_pattern = normal_pattern1
121         infecting_pattern = infected_pattern1
122     elif age_category == 2:
123         # replace TGGAC with TTACC
124         original_pattern = normal_pattern2
125         infecting_pattern = infected_pattern2
126     else:
127         # replace CCGT with GTTT
128         original_pattern = normal_pattern3
129         infecting_pattern = infected_pattern3
130
131     # replace pattern
132     return dna_profile.replace(original_pattern, infecting_pattern)
```

Figure 3.4

Find persons alive/dead status

After inflicting the DNA, program finds the no of occurrences of infected patterns in DNA profile. If the infected pattern count exceed than 200 the person is dead. These infected pattern matching part also done with **Knuth-Morris-Pratt** algorithm.

Example output

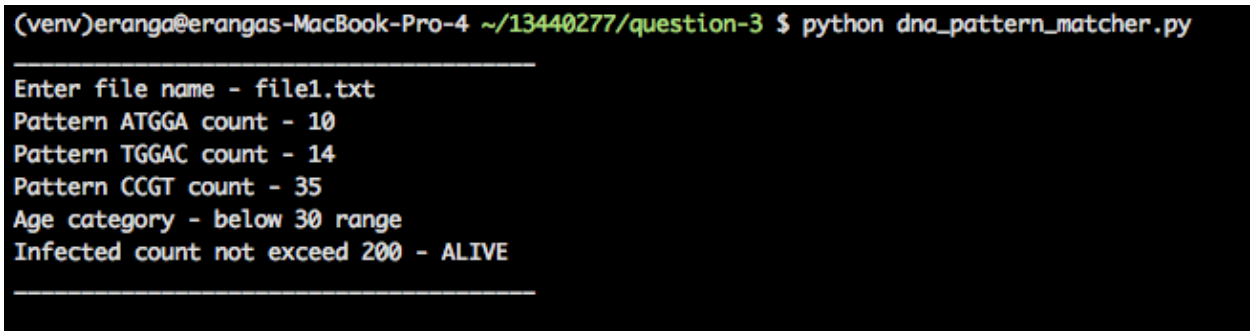

Figure 3.5

Initially program ask to enter DNA text file. I have given “file1.txt”. The program scan the content in DNA profile and give the following informations

1. Pattern count
2. Age category
3. Deal/Alive status

Further discussion about DNA pattern matching

Even though **Knuth-Morris-Pratt** algorithm runs in **O(n+m)**) time, the ideal algorithm to match the DNA profiles might be the **Rabin-Karp** algorithm. Rabin-Karp algorithms works pretty well when finding multiple patterns(with same length) in a text simultaneously. In this problem there are three patterns. Two patterns with same length (**ATGGA**, **TGGAC**) and one with different length (**CCGT**). While iterating over the DNA text we can match same length patterns simultaneously with Rabin-Karp algorithm. So in here we can find the occurrence of first two patterns in same iteration. But in order to match third pattern we need to have additional iteration over DNA profile since its length is different. Sometimes using Rabin-Karp algorithm might be able to increase the performance.

Program executing instructions(Need to have python installed, in order to run the programs)

Question 1

- ➔ Go to "**question1"** directory from terminal
- ➔ Execute command "**python anagram_finder.py**" (program ask for a text to match)

```
(venv)eranga@erangas-MacBook-Pro-4 ~/13440277/question-1 $ python anagram_finder.py
_____________________________________
Enter text - opt
text - opt
anagram count - 2
anagram list - ['pot', 'top']
_____________________________________
```

Question 2 (**Figure 2.4**)

- ➔ Go to "**question2"** directory from terminal
- ➔ Execute command "**python pattern_matcher.py**" (program ask for a pattern and text to match)

```
(venv)eranga@erangas-MacBook-Pro-4 ~/13440277/question-2 $ python pattern_matcher.py
Enter a pattern to match - ABC
_____________________________________
Enter text to match □ - AABC
Matching text - AABC
Matching pattern - ABC
Pattern found at index 1
_____________________________________
```

Question 3 (**Figure 3.5**)

- ➔ Go to "**question3"** directory from terminal
- ➔ Execute command "**python dna_pattern_matcher.py**"
- ➔ Then program ask for a file name. DNA strings stored in files. So give "**file1.txt**", "**file2.txt**" or "**file3.txt**" as an input file

```
(venv)eranga@erangas-MacBook-Pro-4 ~/13440277/question-3 $ python dna_pattern_matcher.py
_____________________________________
Enter file name - file2.txt
Pattern ATGGA count - 5
Pattern TGGAC count - 2
Pattern CCGT count - 28
Age category - below 30 range
Infected count not exceed 200 - ALIVE
_____________________________________
```