1 Introduction

This report contains a description of the code made by Stefano Costa for the second project of the scientific computing course. In the report, individual functions written in the code are analyzed. In order not to report excessively long codes, long comments on the functions have been eliminated, which are then described. For comments and code details refer to the delivered file (also because minor changes have been made).

Complexity is calculated for each function, rather than for the algorithm used. This choice was made because the programming techniques in this project greatly affect performance. For example, the choice of data structure, methods used, and implementations written are very affecting on performance. In general, the complexity of the code is taken from the site: python - Time complexity. For operations with dictionaries, as suggested by assignment, what is referred to in the site as the worst case is always used: $O(k)$, with k being the length of the key. In the complexity analysis part, only the lines of code that significantly affect in the number of operations are considered.

The extra parts were not done because the DNA sequences are very long, to the point where my computer cannot handle them.

Two important observations are the following:

Remark 1 - order in dictionaries

The code relies on the fact that dictionaries in python maintain the order in which elements are inserted (from version 3.6). This is critical in task 1 when generating the L1 list.

A good overview of this characteristic, complete with sources and documentation, can be found at the following question posed on Stack Overflow

Remark 2 - sorting dictionary

The code exploits the `sorted()` function, which has complexity of $n \cdot log(n)$.

Since it was not seen in class in detail, it is explained how it is used: `sorted_kmers_tuples = sorted(kmers_dict.items(), key=lambda item: item[1])`

`sorted()` allows to define a `key` function that is applied before sorting. In this way lists or tuples of elements can be sorted by a specific index. In the case used in the code, it is sorted according to the dictionary value, so the element at index 1. Hence the `key` function simply returns the second element of the pair `(key, value)`.

For more information see: python - sorted()

2 kmers_dict_freq_index(S, k, start_index, end_index, compute_indexes)

Explanation

The function finds all the k-mers in the string `S` and saves indices in which they appear within the range `[start_index, end_index]`. It is possible to ask the function not to find indexes by setting `compute_indexes` to `False`.

The function returns a dictionary of frequencies of k-mers present in `[start_index, end_index]`, and (if asked) a dictionary of indexes of k-mers present in the same interval.

The function was programmed by modifying a simple function for counting words in a string.

Code

```
def kmers_dict_freq_index(S, k, start_index=0, end_index=-1, compute_indexes = True):
    # ---
    # Set ending index
    # ---
    lenS = len(S)
    if end_index == -1:
        end_index = lenS

    # ---
    # Initialize dictionaries
    # ---
    kmers_dict = {}
    kmers_index_dict = {}

    # ---
    # Populate dictionaries
    # ---
    for i in range(0, lenS-k+1):  # cycle in all S in search of k-mers
        substr = S[i:i+k]   # avoid computations

        if substr in kmers_dict:
            kmers_dict[substr] += 1      # add to frequency

            # If the kmer is in the interval add also the index
            if compute_indexes and i >=start_index and i <= end_index-k:
                if substr in kmers_index_dict:
                    kmers_index_dict[substr].append(i)
                else:
                    kmers_index_dict[substr] = [i]

        else:
            kmers_dict[substr] = 1       # set first appearance

```

```
34              # If the kmer is in the interval add also the index
35              if compute_indexes and i >=start_index and i <= end_index-k:
36                  if substr in kmers_index_dict:
37                      kmers_index_dict[substr].append(i)
38                  else:
39                      kmers_index_dict[substr] = [i]
40
41
42      if not compute_indexes:  # haven't computed the indexes
43          return kmers_dict
44
45      # ---
46      # Delete kmers not in the interval [start_index, end_index]
47      # ---
48      else:
49          kmers_interval_dict = {}
50          for kmer in kmers_index_dict.keys():
51              kmers_interval_dict[kmer] = kmers_dict[kmer]
52
53          return kmers_interval_dict, kmers_index_dict
```

Complexity

Calling l the the width of the interval `[start_index, end_index]`, complexity is analyzed based on lines of code. First the lines of code executed regardless of the value of `compute_indexes`.

- Line 18
  The `for` cycles through a list of exactly `len(S)-k` elements.
- Line 19
  The slicing has a complexity of $O(k)$.
- Lines 22, 32
  The dictionary operation has complexity of $O(k)$, since k is the length of `substr`.

So without calculating the indices, the complexity of the function is $O((\texttt{len(S)} - k) \cdot 2k)$
When indexes are also to be saved, one must also add:

- Lines 25-29, 35-39
  These lines of code are executed only when the k-mer is in the range, so only l times. There is access to a dictionary, so complexity of $O(k)$.
- Lines 50-53
  The call `dict.keys()` has complexity $O(1)$, the `for` performs at most l repetitions (because there might be k-mers repetitions), there are two accesses to dictionaries that altogether have complexity $O(2k)$.

So in this case, adding the operations mentioned above, we arrive at a complexity of $O((\texttt{len(S)} - k) \cdot 2k + 3 \cdot l \cdot k)$.

Comparison between estimated complexity and average time

To compare the theoretical estimate with the average case, random DNA sequences of lengths from 1 to 100 are created; 50000 strings are created for each length.

For each string is called `kmers_dict_freq_index(S, k, end_index=l)`, where $k = \texttt{len}(S) \cdot 0.2 + 2$ casted as `int` (where the $+2$ is to avoid degenerate situations) and $l = \texttt{len}(S) \cdot 0.1 + 2$ casted as `int` (again $+2$ is to avoid degenerate situations), the mean and standard deviation are calculated. The empirical results are shown in figures 1 and 2.

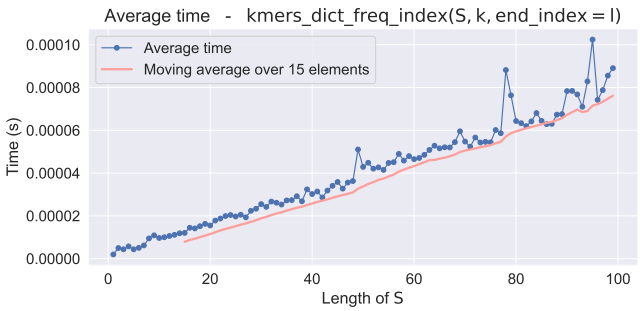

Figure 1: Average time required by `kmers_dict_freq_index(S, k, end_index=l)` for different length of S.

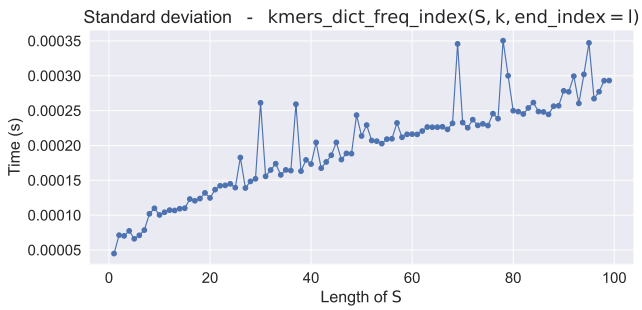

Figure 2: Standard deviation of time required by `kmers_dict_freq_index(S, k, end_index=l)` for different length of S.

It can be seen immediately that the standard deviation has fluctuations and generally increases as the length of the string increases. This is due to the fact that the 50000 strings considered for time averaging are a small percentage of the possible DNA sequences longer than 10.

Despite this, it can be said that the mean time overall follows an almost linear trend, leaving out fluctuations. This trend can be seen particularly from the moving average.

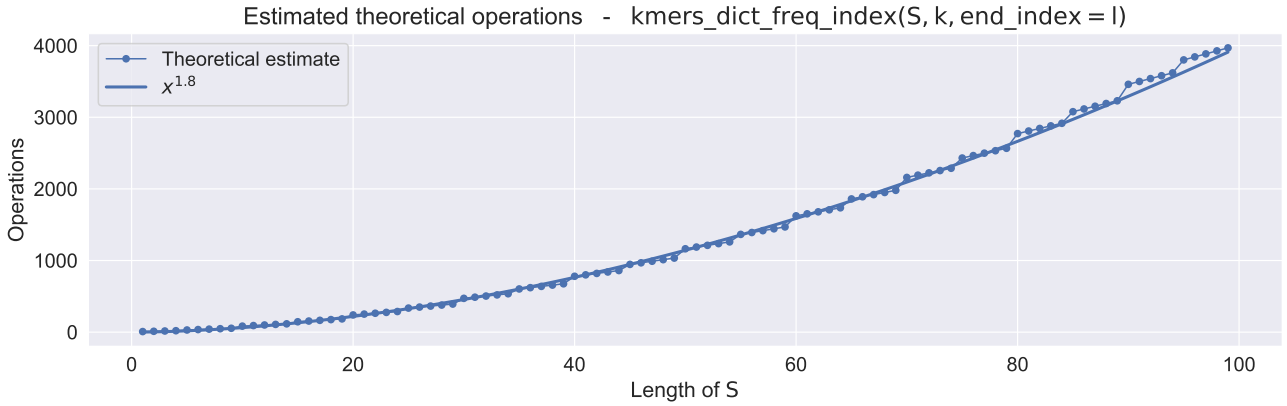

Figure 3: Hypothesized number of operations required by `kmers_dict_freq_index(S, k, end_index=l)` for different length of S.

This trend is also described by the complexity assumed and plotted in figure 3. It can be seen that the theorized complexity grows very precisely as $x^{1.8}$.

It can be seen that the theoretical trend follows the empirical trend well in the first data, those without fluctuations. As seen in figures 4 and 5, the trend is quite precise.

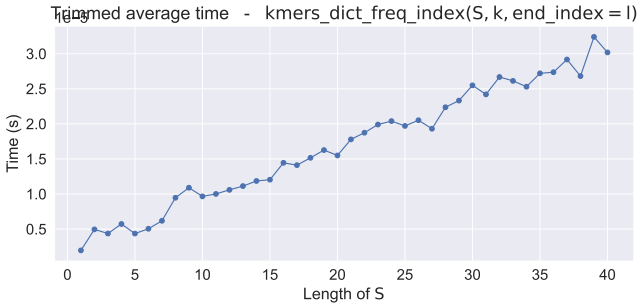

Figure 4: Average time required by `kmers_dict_freq_index(S, k, end_index=l)` for length of S up to 40.

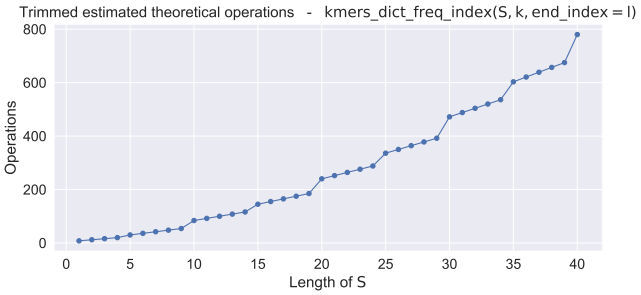

Figure 5: Hypothesized number of operations required by `kmers_dict_freq_index(S, k, end_index=l)` for length of S up to 40.

3 kmer_search(S, k, freq, x)

Explanation

The function relies on `kmers_dict_freq_index(S, k, start_index, end_index, compute_indexes)` to find the frequency of k-mers and indices (without setting `start_index` and `end_index`, then working on the whole string).

To find frequent k-mers, all frequency values from the appropriate dictionary are consulted in a simple `for`.

To find the point-x mutations, all k-mers are compared with each other. Since many consultations of the same k-mer with its values in the proper dictionaries are required in this process, proper lists are first initialized to speed up the retrieve of information. Two indexes in the list of all k-mers are used to compare all k-mers with each other. In this way, for each k-mer, all subsequent k-mers are taken into account. For each pair of k-mers it is checked whether they are point-x mutations using a function. If yes, the mutation dictionary is updated appropriately.

Once all the dictionaries are arranged, the two lists `L1` and `L2` are generated, which are then returned.

Code

```
def kmer_search(S, k, freq, x):
    # ---
    # Check if two strings are point-x mutations
    # ---
    def point_x_mutation_control(s1,s2,x):
        # Assuming s1, s2 of the same length
        l = len(s1)

        for i in range(l):
            if i==x:
                if i+1 in range(l): # To check whether I am at the last index
                    i+=1
                else:
                    break
            if s1[i]!=s2[i]:
                return False

        return True

    # ---
    # Initialize lists and dictionaries
    # ---
    L1 = [] # list of indexes of frequent k-mers
    L2 = [] # point-x mutations

    kmers_dict = {}   # Dictionary of kmers + appearance
    kmers_index_dict = {} # Dictionary of index appearances
    kmers_mutation_dict = {} #Dictionary of mutations

    # ---
    # Find frequencies and indexes of kmers
    # ---
    kmers_dict, kmers_index_dict = kmers_dict_freq_index(S, k)

    # ---
    # Initialize temporary support lists
    # ---
    freq_kmer_list = [] # Frequent kmers list
    for val in kmers_dict.values():
        if val >= freq:
            freq_kmer_list.append(True)
        else:
            freq_kmer_list.append(False)

    tot_kmers_list = list(kmers_dict.keys()) # All kmers list
    tot_kmers_list_len = len(tot_kmers_list) # How many total kmers
    kmers_freq_list = list(kmers_dict.values()) # Frequency of all kmers list

    # ---
    # Find point-x mutation
    # ---
    for i, kMer in enumerate(tot_kmers_list):
        j=i+1
        while j < tot_kmers_list_len:
            comparing_kMer = tot_kmers_list[j]

            if point_x_mutation_control(kMer, comparing_kMer, x): # they are point-x mutations

                # Update kMer mutations
                if freq_kmer_list[i]: # is a frequent k-mer
                    increment = kmers_freq_list[j]
                    if kMer in kmers_mutation_dict:
                        kmers_mutation_dict[kMer] += increment
                    else:
                        kmers_mutation_dict[kMer] = increment

```

```
67                 # Update comparing_Kmer mutations
68                 if freq_kmer_list[j]: # is a frequent k-mer
69                     increment = kmers_freq_list[i]
70                     if comparing_kMer in kmers_mutation_dict:
71                         kmers_mutation_dict[comparing_kMer] += increment
72                     else:
73                         kmers_mutation_dict[comparing_kMer] = increment
74             j+=1
75
76     # ---
77     # Generate the output
78     # REMARK: using list lowers the computational costs
79     # ---
80     for i in range(tot_kmers_list_len): # cycle in all kmers
81         if freq_kmer_list[i]:  # frequent kmer
82             kmer = tot_kmers_list[i]
83
84             L1.append(kmers_index_dict[kmer])
85             if kmer in kmers_mutation_dict:
86                 L2.append(kmers_mutation_dict[kmer])
87             else:
88                 L2.append(0)
89
90     return L1, L2
```

Complexity

Again the complexity of each line is analyzed.

- Line 5
  The function `point_x_mutation_control(s1, s2, x)` checks the equality of strings index by index skipping one, so it has complexity $O(\texttt{len}(s_1) - 1)$
- Line 33
  As seen in the previous section, the function call has complexity $O((\texttt{len}(S) - k) \cdot 2k)$
- Line 39
  The `for` repeats $\leq \texttt{len}(S) - k$ times operations of complexity $O(1)$, so we can say it has complexity $O(\texttt{len}(S) - k)$.
- Lines 45, 47
  Casting `list(...)` performs a copy, so it has complexity $O(\texttt{len}(S) - k)$, which is the maximum number of k-mers that S can contain (obviously without counting repetitions...). So the two lines have complexity of $O(2 \cdot (\texttt{len}(S) - k))$.
- Line 52
  The `for` repeats $\texttt{len}(S) - k$ times (again not counting possible repetitions of k-mers)
- Line 57
  As seen previously, the call `point_x_mutation_control(kMer, comparing_kMer, x)` has complexity $O(k - 1)$
- Lines 62-65
  The operation on the dictionary has complexity $O(k)$ while the operator `in` has complexity $O(k)$; since it is an `if`, that block has complexity $O(2k \cdot \rho \cdot \chi)$, where ρ is the number of frequent k-mers and χ the numbers of point-x mutations, because of the `if`s.
- Lines 70-73
  As the above block, this block has complexity of $O(2k \cdot \rho \cdot \chi)$.
- Line 80
  The `for` repeats at most $\texttt{len}(S) - k$ times.
- Line 84
  The dictionary operation has complexity $O(k)$
- Line 86
  The dictionary operation has complexity $O(k)$

Overall, the complexity is $O((\texttt{len}(S) - k) \cdot (5k + 2) + 4k \cdot \chi \cdot \rho)$. Since χ and ρ depend linearly on k and since there is a linear dependence in k also on $\texttt{len}(S)$ (which we assume to be very large, much more than χ and ρ), we can eliminate the part with the unknown constants and estimate: $O((\texttt{len}(S) - k) \cdot (5k + 2))$

Comparison between estimated complexity and average time

To compare the theoretical estimate with the average case, random DNA sequences of lengths from 1 to 100 are created; 50000 strings are created for each length.

For each string is called `kmer_search(s, k, freq, x)`, where $k = \texttt{len}(S) \cdot 0.2 + 2$ (where the $+2$ is to avoid for degenerate situations), $x = 0$ (the value of x does not change the runtime) and[1] `freq = 2`, the mean and standard deviation are calculated.

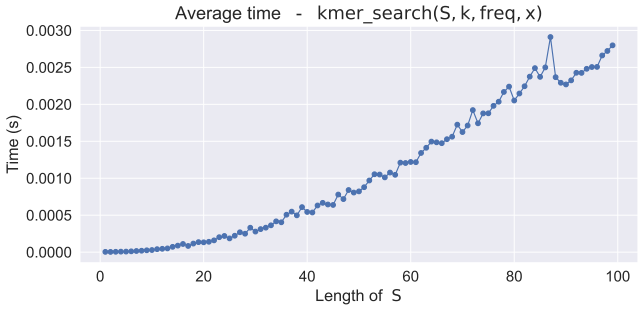

Figure 6: Average time required by `kmer_search(S, k, freq, x)` for different length of S.

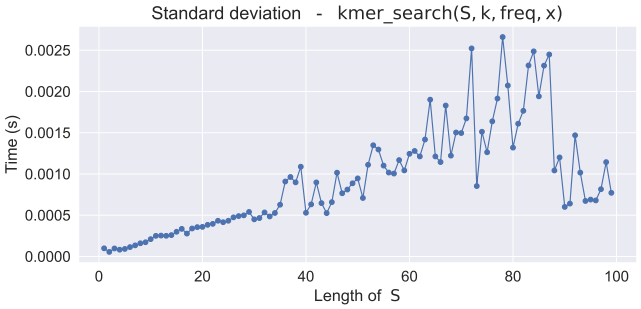

Figure 7: Standard deviation of time required by `kmer_search(S, k, freq, x)` for different length of S.

The empirical results are shown in figures 6 and 7.

It can be seen that as the length of `S` increases, the standard deviation fluctuates more and more. This is due to the fact that, particularly for very long strings, 50000 is only a very small percentage of all possible DNA sequences. Already starting with `S` 10 characters long we are talking about a million possible DNA sequences, of which those examined in the report are only 5 percent.

It can be seen, however, that the mean time trend is fairly regular, although it fluctuates. Conjecturing a little, we could interpret this regularity as a sign that the $k \cdot \chi \cdot \rho$ term is small enough that it does not count for much. However, a more detailed analysis is needed to draw firm conclusions about it.

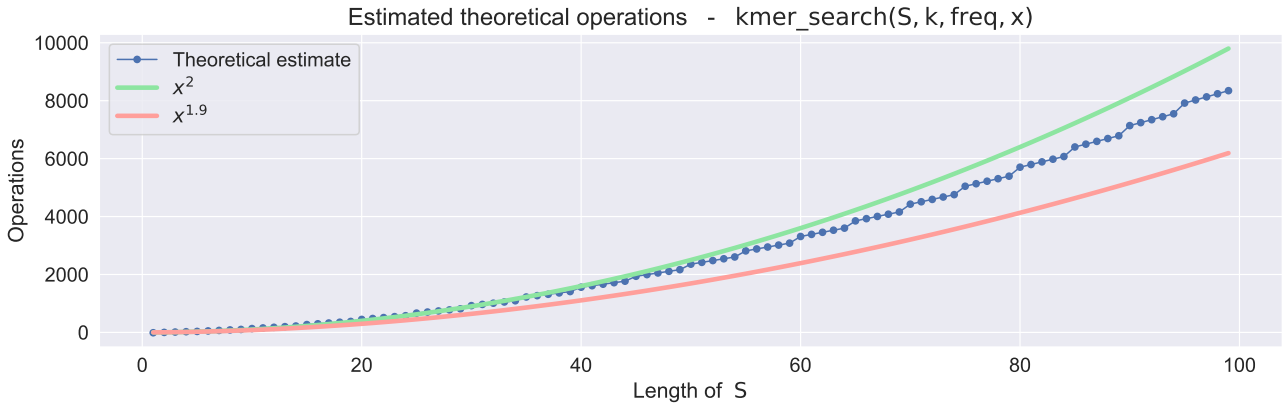

Figure 8: Hypothesized number of operations required by `kmer_search(S, k, freq, x)` for different length of S.

The theoretical results shown in figure 8 represent the mean time trend well and shows that the number of operations follows the trend of x^2.

[1] In retrospect, I realize that perhaps I should have increased the frequency value so that the ρ value would be smaller. In spite of this, however, the theorized complexity emulates the mean time trend well.

4 spet_location(S, k, p)

Explanation

This function also relies on `kmers_dict_freq_index(S, k, start_index, end_index, compute_indexes)` to find the frequency and indices of the k-mers in the interval $[p - 2 \cdot k, p - k)$, appropriately setting `start_index` and `end_index`.

It also relies on that function to find the ratio of "C" and "G", looking for 'k-mers' consisting of one letter (i.e., individual bases).

Once the k-mers that meet the requirements are found, they are first sorted using the `sorted()` function as explained in the introduction.

Finally, the distances to p of the k-mers with the lowest frequency (thus those appearing at the beginning of the ordered list) are compared, and the index of the nearest k-mer is returned.

Code

```
def spet_location(S, k, p):
    # ---
    # Dictionary of kmers and indexes of kmer not 'too distant' from p
    # ---

    # Define 'not too distant'
    inf_index = p-2*k
    sup_index = p-1 #because of how the function kmers_dict_freq_index works

    kmers_dict, kmers_index_dict = kmers_dict_freq_index(S, k, inf_index, sup_index)

    # Delete kmers that appear >5 times in the interval
    kmers_dict = {k:v for k,v in kmers_dict.items() if v <= 5}

    # ---
    # Case: no kmer with less than 5 appearances
    # ---
    if not kmers_dict:
        return None

    # ---
    # check C-G ratio
    # ---
    kmers_list = list(kmers_dict.keys())
    for key in kmers_list:

        # Get dictionary of characters (DNA basis) of kmer
        char_dict = kmers_dict_freq_index(key, 1, compute_indexes=False)

        # get C,G frequency
        C_freq = 0
        G_freq = 0
        if "C" in char_dict:
            C_freq = char_dict["C"]
        if "G" in char_dict:
            G_freq = char_dict["G"]

        # Check ratio
        if 0.35*k > C_freq + G_freq or C_freq + G_freq > 0.65*k:
            del kmers_dict[key]
    # ---
    # Case: no kmer with correct C-G ratio
    # ---
    if not kmers_dict:
        return None

    # ---
    # Case: only one kmer admissible
    # ---
    number_kmers = len(kmers_dict)

    if number_kmers == 1:  # If I have only one kmer
        only_kmer = list(kmers_dict.keys())[0]
        only_kmer_index = kmers_index_dict[only_kmer][0]
        return only_kmer_index

    # REMARK: Now in kmers_dict there are only kmers (>1) that appears less than 5 times AND with the
    right percentage of C and G

    # ---
    # Order the kmer by frequency
    # ---
    sorted_kmers_tuples = sorted(kmers_dict.items(), key=lambda item: item[1])

    # ---
    # Function to compare the distance from p
    # ---
```

```
67      def compare_min_distance(min_distance, kmer):
68          temp_q = -2
69          temp_min_dist = min_distance
70
71          index_list = kmers_index_dict[kmer]
72          for ind in index_list: # check all the indexes of the kmer
73              kmer_distance = abs( ind - p )  # check distance from p
74              if min_distance >= kmer_distance: # if it is closer to p set it as the closest
75                  temp_q = ind
76                  temp_min_dist = kmer_distance
77
78          return temp_min_dist, temp_q
79
80      # ---
81      # Search kmer with lowest frequency and closest to p
82      # ---
83
84      # First definition of min_distance and q is from the lowest frequency kmer
85      min_distance, q = compare_min_distance(2*k, sorted_kmers_tuples[0][0]) # 2k is always >= of any
        distance
86
87      # Check only lowest frequency kmers
88      for i in range(1, number_kmers):
89          if sorted_kmers_tuples[i][1] == sorted_kmers_tuples[i-1][1]: # if they still have the lowest
        frequency
90              temp_min_dist, temp_q = compare_min_distance(min_distance, sorted_kmers_tuples[i][0])
91              if temp_q != -2:
92                  q = temp_q
93                  min_distance = temp_min_dist
94          else:
95              break
96
97      return q
```

Complexity

The complexity of the next function is very difficult to assess, as most cycles repeat code according to how many k-mers respect certain properties. Having an estimate of how many these k-mers are is almost impossible as it depends from situation to situation. Hence the estimation of complexity will be very rough.

- Line 10
  As seen above, the function has complexity $O((\texttt{len}(S) - k) \cdot 2k + 6k^2)$ (where here we have the value of l that is $(p-1) - (p-2k) = 2k - 1 \sim 2k$)

- Line 13
  The creation of the dictionary has complexity $O(n)$ where n is the number of elements. Here the number of elements is the number of kmers with length ≤ 5. This value depends on several parameters: for example, a small value of k means that there are probably many repeating k-mers, thus lowering the number of k-mers present less than 5 times; a shorter S may instead increase this number... From now on, this value is called γ.

- Line 24
  The line of code generates a copy, which requires γ operations, i.e. the length of the dictionary.

- Line 25
  the `for` is repeated γ times.

- Line 28
  As stated above, the call has complexity of $O(2(k-1))$.

- Lines 39-40
  One enters the `if` only when a k-mer does not respect the correct ratio of "C" and "G". The dictionary operation has complexity $O(k)$. So these two lines of code have complexity $O(\sigma \cdot k)$ where σ is the number of k-mer with frequency less than 5 that do not respect the ratio of the bases "C" and "G". Again, estimation of this value is almost impossible.

- Line 62
  The sorting operation has complexity $O(n \cdot log(n))$ with n being the number of elements. In this case the number of elements corresponds to the number of admissible k-mers, i.e. elements with frequency ≤ 5 and which respect the ratio between bases. This number is called θ, so the complexity is $O(log(\theta) \cdot \theta)$. Again, estimating this value is very difficult.

- Line 67
  The dictionary operation has complexity $O(k)$ while everything else we can say is constant (even the `for`, since it repeats operations for every index where the k-mer appear, but we can assume those indexes are few). So overall, the function `compare_min_distance(min_distance, kmer)` has complexity $O(k)$.

- Line 85
  As stated above, the function has complexity of $O(k)$.

- Lines `88, 90`
  The `for` is repeated θ times. Line `90` is accessed only for the least frequent k-mers and, as stated above, has complexity $O(k)$. Assuming that line of code is accessed for every k-mers (also because there is no way to know exactly how many meet the entry condition...), overall this block of code has complexity of $O(k \cdot \theta)$.

Overall, there is a complexity of $O\left(k \cdot (2\texttt{len}(S) + 1) + 4k^2 + k(2\gamma + \gamma\sigma + \theta) + \theta\log(\theta)\right)$. It can therefore be seen that there is dependence on the unknown constants γ, σ and θ linearly in k, negligible since there is also a quadratic term in k. The term $\theta\log(\theta)$ may not be negligible. So we can estimate the complexity as $O\left(k \cdot (2\texttt{len}(S) + 1) + 4k^2 + \theta\log(\theta)\right)$.

Comparison between estimated complexity and average time

To compare the theoretical estimate with the average case, random DNA sequences of lengths from 1 to 100 are created; 50000 strings are created for each length.

For each string is called `spet_location(S, k, p)`, where $k = \texttt{len}(S) \cdot 0.2 + 2$ (where the +2 is to avoid degenerate situations) and $p = i/2$ casted as `int`, the mean and standard deviation of time required by every call has been calculated.

The emprirical results are shown in figures 9 and 10.

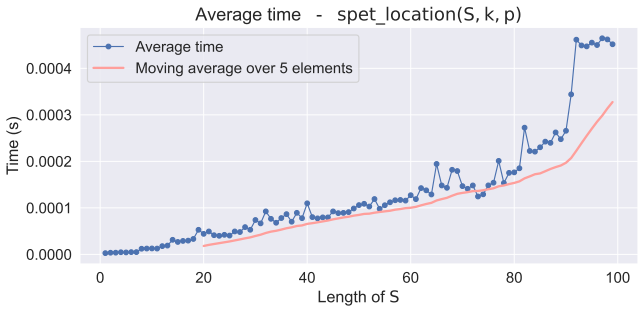

Figure 9: Average time required by `spet_location(S, k, p)` for different length of S.

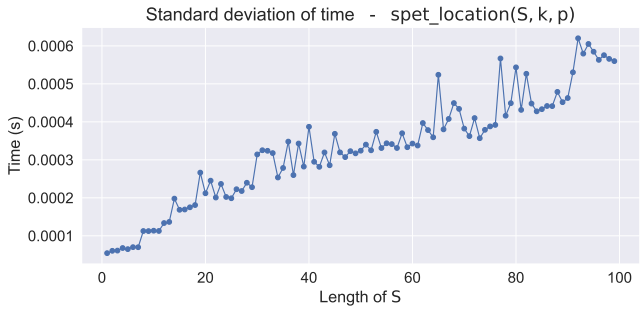

Figure 10: Standard deviation of time required by `spet_location(S, k, p)` for different length of S.

As a first thing, the fluctuations in both mean and standard deviation are observed. This is due to the fact that 50.000 strings is a low percentage of all possible DNA strings from the length of 15 bases onwards (Remark: $4^{20} \sim 10^{12}$).

Because of this we can expect the theoretical predictions not to coincide precisely, but to represent the asymptotic trend. We also expect inaccuracies since we neglect the θ term, and in fact the theoretical results shown in figure 11 confirm this.

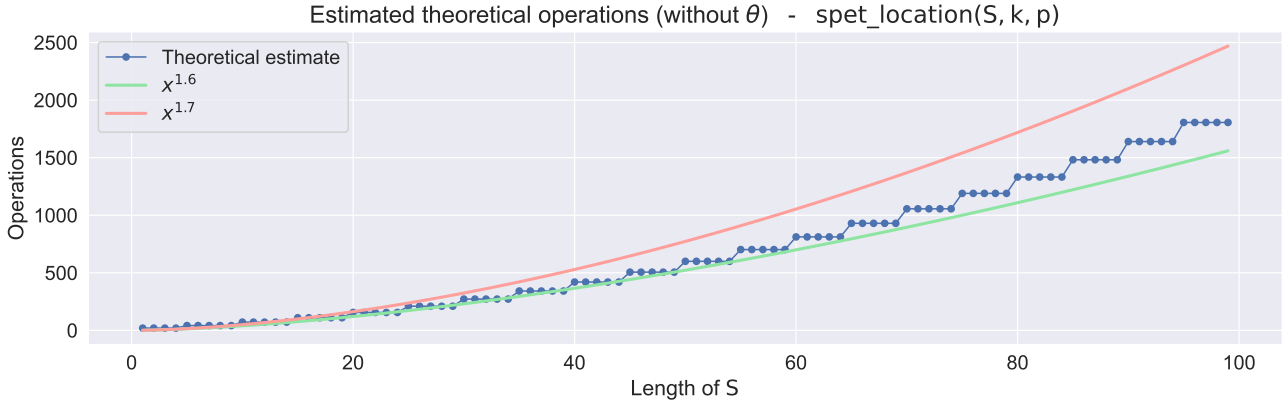

Figure 11: Hypothesized number of operations required by `spet_location(S, k, p)` for different length of S.

Note that although neglecting the θ term, comparing the trend with the moving average in figure 9, we can say that the theoretical curve succeeds in mimicking the average number of operations required. Obviously there will be more accuracy in prediction for small lengths of the string S, as shown in figures 12 and 13.

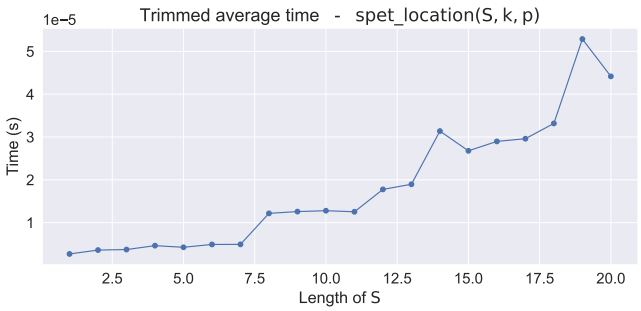

Figure 12: Trimmed average time required by `spet_location(S, k, p)` for length of S up to 20.

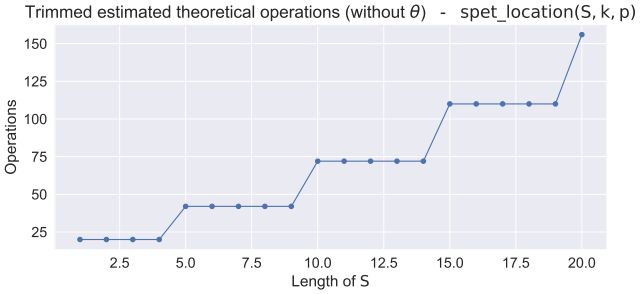

Figure 13: Trimmed hypothesized number of operations required by `spet_location(S, k, p)` for length of S up to 20.

For short lengths, the mean time also tends to jump as in theory. Since already at length 12 the possible DNA sequences become on the order of 16 million, the average time begins to be irregular and the theorized number begins to become inaccurate. We could also conjecture that as the length of `S` increases, with the `k` and `p` values set, the unknown theta becomes larger and its weight in the number of operations becomes greater and greater, still justifying the differences between the graphs. However, a deeper analysis is needed to confirm this hypothesis.

5 Task 3 results

p	q	k-mer starting at q
10	None	
1508	1467	GCCGTCAATAAGGTAGGGATCATCAAAACACCAAACCATC
140500	140459	GCTTTACGCAATCGATCGGATGAGATAGATATCCCTTCAA

Table 1: Results of application of `spet_location(S, k, p)` to chloroplast of the tomato plant.