# Project 2 - Scientific Computing

Letizia D'Achille

## Description of the implementations

`kmer_search`

We first iterate over all the $k$-letter substrings within the string $S$. We progressively construct a dictionary where the `key` is the string corresponding to a $k$-mer, and the corresponding `value` is the list of all its locations. For each $k$-letter substring, we check whether we had already found it. When we find a new $k$-mer, we add a new pair `(key, value)` to the dictionary. The `value` is initialized with a list containing one element only, that is the location of the newly found $k$-mer. Otherwise, when we find an already seen $k$-mer, we update the `value` of the corresponding entry of the dictionary, appending the location of the current substring to the list.
We then iterate over all the entries of the dictionary, checking if the number of occurrences of the $k$-mer is greater or equal to the fixed frequency. In this case, we append the list of locations to the list `L1`. This is ordered since we scanned the string linearly. Moreover, we count the number of point-$x$ mutations by considering the strings obtained by modifying the $k$-mer in the position $x$ (three possible modifications), and accessing them in the dictionary. Whenever a corresponding entry in the dictionary is found, the number of occurrences of the string is added to a counter (initialized to 0). Note that `L1` will indeed be ordered with respect to the first occurrence of the $k$-mers since Python guarantees that the order of items in a dictionary is preserved (from version 3.7).

`spet_location`

We first construct a dictionary where the `key` is the string corresponding to a $k$-mer in the range $[p - 2k, p - k)$, and the corresponding `value` is a list of two elements. The first of the two is the closest location of the $k$-mer to $p$, while the second is the number of occurrences of the $k$-mer in the whole string. To achieve this, we iterate over the substrings of $S$ starting from the index $p - k - 1$, going backwards to $p - 2k$. When we find a new $k$-mer, we add a new entry to the dictionary. We save in the `value` the location of the substring, that will indeed be the closest to $p$ since we are iterating backwards, and we initialize the number of occurrences to 1. Every time we find an already seen $k$-mer, we update this counter. Then we iterate over the remaining positions in $S$, without saving any new $k$-mers, and updating the counter whenever we find an already seen $k$-mer.
Subsequently, for each $k$-mer in the dictionary we verify whether it is a SPET location by checking its frequency and the proportion of the letters "C" and "G" in the string.

We save the location and the frequency of the first SPET location found. Then, for every other admissible $k$-mer, we check whether the frequency is strictly lower that the saved one, and in this case we update the saved location and frequency.
The $q$ found at the end of the algorithm will be a SPET location since we only saved indexes of strings in the range $[p-2k, p-k)$ and we checked that the necessary conditions were met. Moreover, if there is more than one SPET location, we indeed find a $k$-mer with the lowest frequency. Additionally, among them, we save the location closest to $p$ since the dictionary was defined by scanning the substrings starting from the closest to $p$ and going backwards.

`get_my_pqs`

The pair $(p, q)$ where $q$ is None was found by considering a random position $p$, applying the function `spet_location`, then iterating this procedure until we found an admissible pair. The same was done in order to find the pair $(p, q)$ where $q$ is a number. Finally, we set $p = 22400$ (the corresponding date was 24/02/00, hence we used the format `MM/DD/YY`) and found the corresponding $q$. In Table 1 we present the three pairs.

| | p | q | $k$-mer |
|---|---|---|---|
| 1 | 87852 | `None` | \ |
| 2 | 130849 | 130808 | `GTTCCTTCTTCCATAACCAGAGCTCGGAGAAGGAAGAGAT` |
| 3 | 22400 | 22359 | `TAAAATGCTTAGCCAATTCCACGCGTCTAACCAAAAAGTC` |

Table 1: Results of `get_my_pqs`()

## Analysis of algorithms' asymptotic complexity

`kmer_search`

We decided not to use a rolling-hash function and Rabin-Karp algorithm for multiple patterns since we could assume that any access to a dictionary is $O(k)$, when the key is a string of length $k$. Indeed Rabin-Karp would require us to compare the hash of each substring with all previously computed hashes, that can be at most $\sim n - k$. A rough computation would give us a complexity of $O((n - k)k)$.
On the other hand, in our case, the lines 30-34 consist of $n-k+1$ iterations of operations of reading or writing accesses to a dictionary with keys of length $k$, that would give us a total complexity of $O((n - k)k)$.
The lines 37-46 consist of $n - k + 1$ iterations at most (the dictionary can contain at most all the possible $k$-substrings of $S$ if there are no repetitions [1]). These lines contain an inner cycle that loops only four times, that is a costant number of times with respect to the parameters $n$ and $k$, therefore the cost of the inner lines remains $O(k)$ (costant number of reading accesses to the dictionary). The cost of this part (in the worst case)

[1]Note that the number of possible $k$-mers are $4^k$, therefore the number of entries in the dictionary cannot exceed this value. Hence, the number of iterations can be at most $\min\{n - k + 1, 4^k\}$. Moreover, one could analyze the probability of appearance of $k$-mers based on the frequency of the codons in the considered genetic material, to determine the expected value of the number of $k$-mers that actually appear. For simplicity, we approximate it with $n - k + 1$, since the total cost would remain the same.

is $O((n-k)k)$. The choice of modifying the $k$-mer in the position $x$ and then searching it in the dictionary, instead of scanning the dictionary and checking whether the keys were modifications of the $k$-mer, was made to reduce the complexity and again exploit the fact that accesses to the dictionary given a key only cost $O(k)$.
The total complexity is then given by the sum, that is again $O((n-k)k)$.

`spet_location`

Even if in this case we have to save only $k$ keys, we again decided not to use Rabin-Karp algorithm. Note that the latter would require at most $k$ checks (one for each saved hash) for each $k$-substring of S, giving us a complexity of $O((n-k)+k^2)$, using hash tables. A test was performed to determine whether this solution was faster than ours, but the implementation of dictionaries in Python seem to suit our algorithm better. Therefore, even if the theoretical complexity is slightly worse, we chose the algorithm presented above.
The lines 131-141 comprehend three loops that collectively range from 0 to $n-k$. Each loop contains consecutive reading and writing accesses to the dictionary that cost $O(k)$, for an overall cost of $O((n-k)k)$.
The lines 143-151 consist of $k$ iterations at most (in any case less than $n-k$ [2]). The inner loop costs $O(k)$ since it ranges over all the characters of a string of length $k$ and it contains costant operations. The other operations are consecutive accesses to the dictionary, that cost $O(k)$ again. The complexity is then (in the worst case) $O(k^2)$. The various checks, made to determine whether an index fulfils the conditions for being a SPET location, are performed in this second loop since, in this way, they do not add to the complexity of the algorithm, that remains lower than the first part.
The total complexity is then again $O((n-k)k)$.

## Extra observations

Table 2 reports the results of the application of the function `spet_location` on the five distinct DNA data sets suggested in the description of the project. In particular, over four distinct values of $k$, we counted how many times the algorithm failed to find a suitable SPET location $q$ over 100 attempts with random $p$.

| | `len`$(S)$ | $k=10$ | $k=25$ | $k=50$ | $k=100$ |
|---|---|---|---|---|---|
| `NC_007898.3` | 155461 | 24 | 16 | 13 | 16 |
| `U00096.3` | 4641652 | 30 | 2 | 3 | 3 |
| `BK006941.2` | 1090940 | 14 | 11 | 7 | 9 |
| `CM001069.3` | 43960406 | 98 | 27 | 32 | 34 |
| `CM000683.2` | 40088619 | 93 | 14 | 14 | 8 |

Table 2: Number of failures over 100 attempts for each DNA data set

[2] Note that the number of iterations does not exceeds $n-k$ since the range $[0, n-k+1)$ always contains $[\max(0, p-k-1), \max(-1, p-2*k-1))$. For the overall computational cost we can therefore take $n-k+1$ as an upper bound for this quantity.

The higher values of failures of the last two datasets for $k = 10$ follow from the high probability that a substring of length 10 repeats itself more than 5 times in a string of length $\sim 10^7$ since $4^{10} \sim 10^6$.
Looking at the significant values, we can note that for some strings of nucleotides (i.e. `U00096.3` - Escherichia coli bacteria) SPET locations are easier to find than in others (i.e `CM001069.3` - Tomato chromosome 6). This might reveal lower genetic diversity in some specific genetic material with respect to other. Indeed, when the algorithm fails to find a SPET location, either all the $k$-mers repeated themselves more than 5 times, or all the $k$-mers in the considered range had not a suitable proportion of `"C"` and `"G"` within them. Therefore there must be an high occurrence of identical $k$-mers, or identical nucleotides.