# Project 2
# Scientific Computing

Rebecca Selvaggini

April 2022

## 1 Task 1

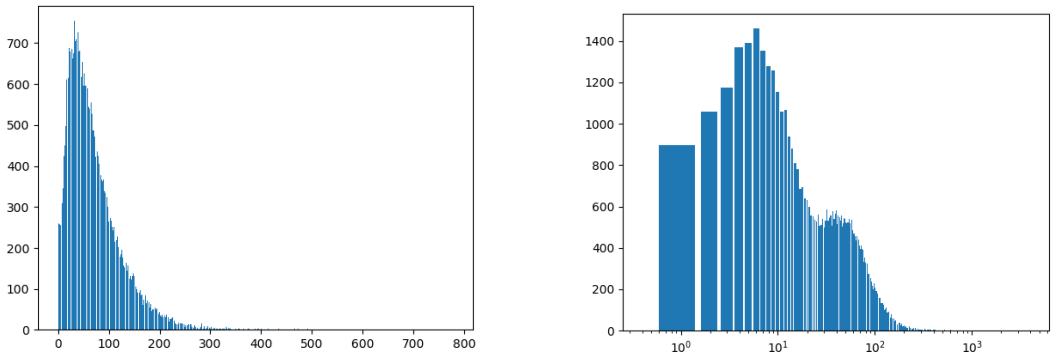
In the function `kmer_hist` we take in input a string `S` and an integer `k`, at first we initialize a dictionary `d` as an empty dictionary. Then we start considering all the substrings of `S` of length `k` and we add them as element in `d` using the substring as a key and the number of times it appears in `S` as the correspondent value. We use the command `count = d.get(kmer,0)` so that if `kmer` is already a key in the dictionary we get the correspondent value (i.e. the number of times the substring `kmer` has already been found in `S`) and we increment it by 1, otherwise we get 0 and we insert a new couple `(kmer, 1)` to the dictionary. After we have construct the dictionary we consider the maximal value in `d` (`m = max(d.values())`) and we initialize `h` as a list of length `m+1` of all zeros (so that the last element `h[m]` is the number of strings with maximal frequency), and `mfkmers` as the empty list. Now for all the keys in `d` (i.e. for all the distinct substrings of `S`) `d[kmer]` is the frequency of the string `kmer` in `S`, so if we find `d[kmer] = i` we increment the value of `h[i]` by 1. Then if `i` coincides with the maximal frequency `m` we append the string `kmer` to the list `mfkmers`. At the end of the loop we have construct the two lists as requested.
Now we want to estimate the complexity of the algorithm assuming that any reading or writing access to a dictionary in Python is O(k). The main cost of the algorithm is given by the two `for` loops. In the first one we compute for `n-k+1` times a reading and a writing access to a dictionary, so we have a cost of $(n-k+1)*(2k+1) \approx O((n-k)k)$. In the second loop we perform a maximum of $n-k$ iteration (the maximum number of distinct substrings of length `k` in `S`); at each iteration we perform a reading access to the dictionary and we modify the two lists `h` and `mfkmers`, so also in this case we get a cost of $\approx O((n-k)k)$. Then the complexity of the entire algorithm is $O(n-k)k + O((n-k)k) = O((n-k)k)$.

## 2 Task 2

We include in Figure 1 the k-mer spectra for the e-coli and human chromosome 21 genomes for $k = 8$. The first image has been generated by the following code:

```
import matplotlib.pyplot as plt
with open("sequence.fasta") as f: text = f.read()
S = "".join((c if c in ["A", "T", "C", "G"] else "") for c in text)
k = 8
h,m = kmer_hist(S,k)
plt.bar(range(0,len(h)), h)
plt.show()
```

The code for the second figure is basically identical, the logarithmic scale on the x-axes is obtained by `plt.gca().set_xscale("log")`. In Table 1 you can find the most frequent k-mers in the two sets of sequencing data for k = 3, 8, 15, 25, 35, together with their frequency.




(a) *Escherichia coli genome* (b) *Human chromosome 21 genome*

Figure 1: $k$-mer spectra for $k = 8$

| Genome | k | frequency | $k$-mers |
|---|---|---|---|
| Escherichia coli | 3 | 115734 | CGC |
| | 8 | 778 | CGCTGGCG |
| | 15 | 71 | ACGCCGCATCCGGCA |
| | 25 | 39 | GGATAAGGCGTTCACGCCGCATCCG |
| | 35 | 22 | GTAGGCCGGATAAGGCGTTTACGCCGCATCCGGCA |
| Human chromosome 21 | 3 | 142696 | TTT |
| | 8 | 3916 | TTTTTTTT |
| | 15 | 822 | TTTTTTTTTTTTTTT |
| | 25 | 290 | TCTCTCTCTCTCTCTCTCTCTCTCT |
| | 35 | 107 | TTTCTTTCTTTCTTTCTTTCTTTCTTTCTTTCTTT |

Table 1: Most frequent k-mers in two sets of sequencing data

# 3 Task 3

The implementation of the function `kmer_search` follows the idea of the Rabin-Karp algorithm and the use of rolling hash functions as seen during the lectures of the course. The functions `extend_by_one`, `remove_left` and `hash_value` give the implementation of the rolling hash function used. The function `kmer_search` take in input a sequence `S` of length `n` and a list `L` of `m` sequences of length `k`. We initialize the two output variables `freq = 0` and `pos = -1`. Then we compute a list of the hash values for any sequence in `L` (`hp = [hash_value(L[i]) for i in range(m)]`), in this case we cannot use the property of the rolling hash function, so that we have to use `hash_value` for each of them. Then we perform a `for` loop computing all the hash values for the substrings of length `k` in `S`. In this case we can use `hash_value` for the first string, and then use `remove_left` and `extend_by_one` to compute the other hash values with less effort. Then we start a loop searching for hash matching: for each string in `L` we initialize `f, p = 0, -1` and we compare the hash value `hp[j]` with any hash value in `S`, if we find a match, and the strings are equal we update the value of `f`(each time) and the value of `p` just in the first occurrence, so that `p` is the first position in `S` where you find `L[j]`. After considering all the string in `S` we compare the values `f` and `freq`, we save in `freq` the greater value, so that at the end of the algorithm it is the maximal frequency; in the case `f == freq` we save the values with the minimal position. At the end if we have `freq == 0` then no string in `L` has been found in `S` then we return `None, 0`.
Now we consider the cost of the algorithm: the initialization phase (computing the hash values) has a cost of $O(mk)$ for the list `L`, for the sequence `S` we have a cost of $O(k)$ for the first value, and a cost of $O(1)$ for all the others, for a total cost of $O(k + n - k) = O(n)$. Then the main cost of the function is given by the two `for` loops: we have an $m$-loops with an inner $(n - k)$-loop, in the inner loop we have a cost of $O(k)$ in the worst case (when we compare all the strings `S[i:i+k]` and `L[j]`). So in the worst case the cost is of $O(mnk)$. The global cost of the function, in the worst case, is $O(mk) + O(n) + O(mnk)$.