Scientific Computation Project 1

Amaury Francou — CID : 01258326

February 4, 2022

Question 1

1(a)

The function `code1` takes a sorted list of integers `L` and an integer element `x` and returns the index corresponding to the position of said `x` in `L`, if this given integer is indeed in the list. Otherwise it returns `-1000` if the integer `x` is not present in the inputed list.

Thus this function is an implementation of a *search* algorithm. In particular, it combines here the use of *linear and binary search algorithms*. In the classic binary search algorithm, half of a sub-array is discarded at each iteration. Here this function performs the binary search method until the current sublist has reached a length that is smaller than `N0` - which is an inputed parameter. Namely, once the iteration-related sublist has reached a length smaller than `N0`, the function performs a linear (naive) search on the remaining elements, by the use of a `for` loop. To implement this "binary search with final linear search" algorithm, the implementation leverage recursive programming - as already seen in the case of classic binary search.

1(b)

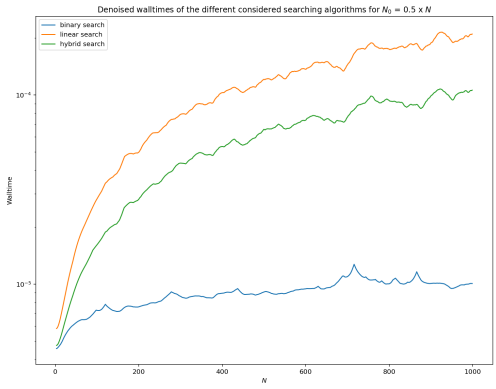
As mentioned previously, the `code1` function performs a binary search approach until the considered sublist is of length smaller than `N0`. In the worst-case scenario, the binary search stage of the considered algorithm does not "point out" the correct value directly, and keeps dividing the list until the length of the sublist is effectively smaller than `N0`. Namely, the condition `x==L[imid]` with `imid = int(0.5*(istart+iend))` is never verified for `iend-istart>N0`.

Consider that the full list is of length $N = 2^n$ and that we have $N_0 = 2^m$. We reach a sublist of length smaller than N_0 for k-half-splits with $2^{n-k} \leq 2^m$. This further gives that we perform $k = n - m$ half-splits. The inner operations (comparisons, assignments) made before calling the function recursively are one-time operations and do not depend on N or N_0. Thus the binary search stage of the considered function has a worst-case scenario of $\mathcal{O}(\log_2(N) - \log_2(N_0))$. The final linear stage step of the algorithm has a worst-case scenario in which the target element is located at the end of sublist (or is not in the list at all), that is to say the one-time inner loop operations are to be made a number N_0 of times. Hence the linear search stage of the considered function has a worst-case scenario of $\mathcal{O}(N_0)$. Finally, we get that the `code1` function has a worst-case scenario of complexity $\mathcal{O}(\log_2(\frac{N}{N_0}) + N_0)$.

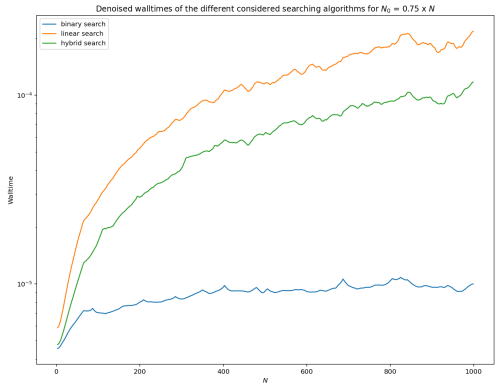
Note that if $N_0 = N$, we retrieve the computational cost of linear search, as this algorithm becomes equivalent to it. Conversely, note that if $N_0 = 1$, the algorithm becomes equivalent to binary search and we well retrieve the corresponding computational cost.

Figures

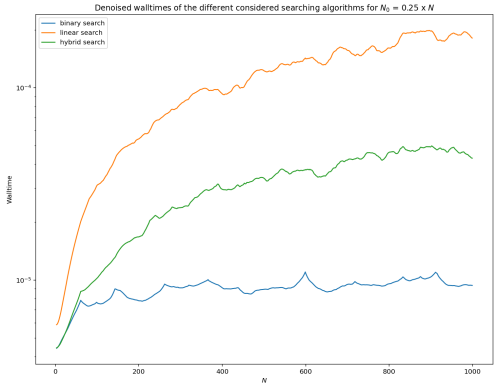
We compute the walltimes required by the function for processing random lists of integers of increasing size N. We analyze the behavior of said walltimes for several values of N_0. Namely, we take $N_0 = 0$ (an `if` statement at the beginning of the function makes this value also equivalent to binary search), $N_0 = 1$ (equivalent to linear search), and $N_0 = \alpha N$ with $\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$.



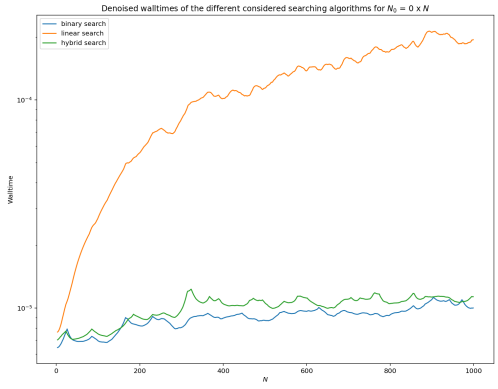
(a) $\alpha = 0.5$



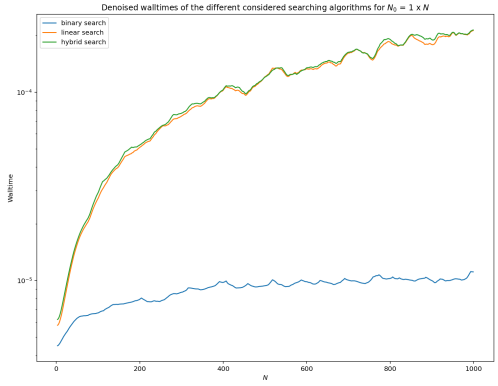
(b) $\alpha = 0.75$



(c) $\alpha = 0.25$



(d) $\alpha = 0$



(e) $\alpha = 1$

Figure 1: Walltimes of the considered search algorithm in log-scale, against N, for $N_0 = 0$ (*binary search*), $N_0 = N$ (*linear search*) and $N_0 = \alpha N$ (*hybrid search*) with $\alpha \in \{0, 0.25, 0.5, 0.75, 1\}$. The walltime values have been denoised. The blue curve is corresponding to binary search, the orange curve to linear search and the green curve to hybrid search.

We notice that linear search walltimes increase linearly with N (having a log trend in log-scale) and give the worst performance in terms of computation time. Moreover, we notice that binary search

walltimes are increasing logarithmically and give the best performance in terms of computation time. The hybrid approach walltimes increase linearly with N (having a log trend in log-scale) and show a performance in computation time that is in between binary and linear search. Indeed, choosing $N_0 = \alpha N$ gives a complexity of $\mathcal{O}(-\log_2(\alpha)+\alpha N)$. The relative performance of the hybrid method depends mainly on the relative size of N_0 compared to N. The best situation occurs when N_0 is small compared to N.

Question 2

2(b)

We design a function that finds the locations of the p amino acid sequences contained in the input list `L_p`. The function first performs a translation of the considered $3m$-length nucleobase sequence contained `L_p`, converting each 3-length codon to the corresponding amino acid using the provided correlation dictionary. Once the current considered sequence of nucleobase has been converted to a sequence of amino acids, the function performs the Rabin Karp algorithm to retrieve the amino acid pattern in the large inputed n-length amino acid sequence `S`. It returns a list of lists `L_out`, containing the index corresponding to the beginning value of the searched amino acid pattern in `S`. The list may be empty if the pattern has never been found in the sequence. In addition, as some codons do not refer to valid amino acids, a one element list containing the value `-1000` may be returned, in the case the considered nucleobase sequence holds a such codon.

The code uses one master `for` loop, linearly going through the p nucleobase sequences of `L_p`. The first translation step is made using a 3-length window capturing the codons jumping 3 by 3 over the elements of the nucleobase sequence. It access the corresponding amino acid immediately evaluating the key of the correlation dictionary, and further concatenates the obtained amino acids. It directly sets the value `-1000` to the output and breaks, if the codon does not refer to a valid amino acid. This stage is made p times, computing m times several one-time operations in the translation sub-loop. Thus, this stage is made in a computation time upper bounded by $\mathcal{O}(pm)$.

The second stage uses the Rabin Karp algorithm, and is made only if a valid amino acid pattern is to be searched. The hash function is called and performs a number of operations that is proportional to m, before entering the rolling hash update $(n-m)$-range subloop. In this sub-loop, if the hash of the target sequence and the hash of the current m-length sub-sequence match, an m-length sequence comparison has to be made. This lead to a running time that does not grow faster than $\mathcal{O}(pnm)$, as $n \gg m$. The overall function is then upper bounded by $\mathcal{O}(pnm)$.

The provided code is memory efficient as it does not make use of large hash tables containing all m-length sublists of consecutive values of `S`. Using the Rabin Karp algorithm, this function is also computationally beneficial - compared to the naive approach - in the many "near-misses" case, that is to say when the searched pattern often *almost* match the m-length sublists of `S`. Finally, some pre-computations of constant parameters have been accordingly made before entering the master loop. For instance see the computations of `bm`, and `hi0`.
