Scientific Computation Project 1

Marwan Riach - CID: 01349928

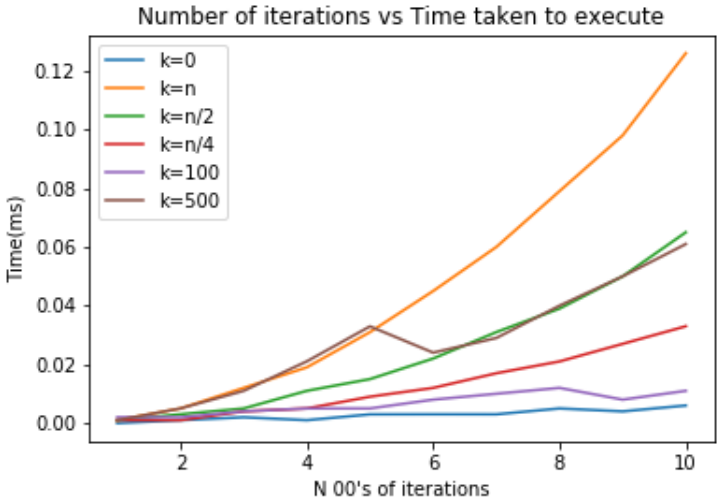
February 6, 2020

Part 1

1.1)

The idea behind the new sort algorithm is that it combines the selection sort and merge sort algorithms, with the extent of each algorithm being dictated by the k value. If k is greater than or equal to the length, N, of the list that is inputted, then the entire algorithm is selection sort. Similarly, is k is equal to 0, then the entire algorithm is merge sort. Any value that lies in between will result in the inputted list being broken down into sorted lists via selection sort. Merge sort will then play its part and arrange the sorted sublists into a long sorted list.

We know that merge sort is of order $O(n \log_2 n)$. This is because whether it be worst case or average case the merge sort just divides the array into two halves at each stage which gives it $\log_2(n)$ component and the other n component comes from its comparisons that are made at each stage.



Similarly, the selection sort algorithm performs a number of iterations that is equivalent to the $(n-1)th$ triangular number. This is equal to $\frac{1}{2}(n-1)(n-2)$ and hence it is $O(n^2)$.

The blue line at the bottom represents the merge sort algorithm and the orange line represents the selection sort algorithm. For constant values of k, what can be seen is that the merge sort component kicks in once the length of the sublists are all of values less than or equal to k. For example, the $k = 500$ line begins to behave quicker than $O(n^2)$ once N exceeds 500.

Furthermore, merge sort only works on ordered lists and hence it is the selection sort's job to provide the merge sort algorithm with ordered lists. In the extreme case where $k = 0$, merge sort is simply applied to N lots of lists with only one element, thus taking order $O(n \log_2 n)$.

The conclusion to be drawn here from the graph plot and knowledge of computational time order is that merge sort is always as efficient as new sort.

1.2)

The original algorithm I created started by considering the edge cases where only two comparison needed to be made. If a trough was found, the function would break and return the location of the trough. Otherwise, it would loop through all of the cases between $i = 1$ and $i = N - 2$. If a trough was found, then the location is returned and the for loop is broken. If this was not the case, then $-(N+1)$ is returned. This was of order N and it hence had an asymptotic running time of order N.

I thought I could do better than this so I decided to use recursion to locate a trough. It is worth noting that every non-empty finite list has a minimum value and hence a trough always exists. This means that $-(N + 1)$ would only ever be returned for inputs that were the empty.

Recursion was used. If the middle element of the list is less than its right neighbour then we know a trough exists in the first half of the list. This is because we have now found a sublist where the edge case is not a minimum. If the middle element is greater than its right neighbour, a trough exists in the second half and we repeat this until one element is outputted. This gives us an algorithm of $O(\log_2 n)$ which is the theoretical time that is taken asymptotically.

When running the algorithm, it is important to input the list you are using twice as the second argument is used as a reference point for the return of the index of a trough.

Part 2

2.1)

I converted the inputted string to a list to make it easier to perform with. I then iterated through the codons using the dictionary as defined in the codontoAA function. During the function's execution, if the length of our codons is 20, then

we know there aren't anymore amino acids and hence we can stop the function and return the required string, AA. I made use of the "not in" function to make sure repeats are avoided, as required.

In the worst case scenario, this function is $O(n)$, however let us consider the expected time until we elapse.

If we assume that each codon is independently and uniformly distributed in the string we are given, then we encounter an alteration to the coupon collector problem: namely, given an $n - sided$ fair die, what is the expected number of rolls in order to roll every number at least once? In our case, $n = 20$, since this is the number of distinct amino acids there are. The formula for the expected value given n is the sum of the reciprocals of the natural numbers between 1 and n inclusive multiplied by n. In this case, we obtain 71.95479314287364, or 72 with rounding.

This means that on average, the function is of $O(1)$, and that $O(n)$ is a rare worst case scenario.

Furthermore, the dictionary could be reduced in size to increase the speed of the codontoAA function since the three elements denoted by an underscore represent the end of a sequence of codons and this is not part of the string we wish to output.

2.2)

For this section, I made use of the functions char2base4 and heval as provided in the fourth and fifth lecture in order to write my strings in base 4 (since there are 4 distinct nucleotides) and produce a hash function respectively.

I chose 101 as my prime. Number theory tells us that the hash value each string is assigned is approximately uniformly distributed, meaning that the probability of two randomly colliding hashes is $\frac{1}{101}$ which is just below 1%.

This significantly reduces my computational time as compared with the naive search method since in the naive method, all strings are directly compared, whereas with a rolling hash function, strings are only directly compared if the hash values are identical.

Of course, the worst case scenario is that every hash value collides, although with a reasonably large prime, this becomes near-enough impossible making the algorithm I have written more efficient than the naive search algorithm.

Furthermore, if I call the length of my list of strings, L, $len(L)$ and the length of my list of pairs,P, $len(P)$ with M representing the length of the $k-mer$ pairs and N representing the length of the strings in the list then the following can be said about the computational time:

The average time for the Rabin-Karp algorithm to be executed is $O(M + N)$, and so the average computational time for the algorithm I have written is $O(len(P) * len(L) * (M + N))$. We are told that $M << N$ so this can be simplified to give $O(len(P) * len(L) * N)$, since in most cases with the rolling hash function, only 4 iterations are required, instead of M.

With the naive search algorithm, the same cannot be said. Every set of strings that are aligned are compared against one another, and so the average

computational time is $O(len(P) * len(L) * MN)$, making it much less time-wise efficient than the modified Rabin-Karp algorithm which I wrote.

One of the big reasons as to why the modified Rabin-Karp algorithm is more efficient is because the rolling hash function is mostly $O(1)$, except for when a match is found. In that case, it becomes $O(M)$, however as I discussed earlier, the prime I chose is such that in the vast majority of cases (circa 99%), the computational time is $O(1)$ as opposed to $O(M)$, thus expediting the process.