Scientific Computation Project 1

Urte Adomaityte
This is my own work unless stated otherwise.

March 2, 2021

Question 1

1(a)

func1A: it is a recursive function that takes in a list of N length-M lists with elements sorted in ascending order (note that the lists do not have to be of the same length), then checks the length of the list (i.e. number of lists it contains). If it is of length 1, then our work is done as it consists of only one list with elements in ascending order - it returns that sublist. If the length of the list is above 1, then there is work to be done: we merge the second-to-last and last element (that we get by removing it from the list at the same time) of the list into one sorted list using the merge() function, reassigning this to be the last element of the list. Then we call the func1A using the new list of length-1 as we merged and sorted the second-to-last and last sublists. The function repeats this process until we are left with one element in the list which we return as the sorted list.

Computational cost (in the worst case):

1) get length of the list and check if it is 1: 2 FLOPS, O(1)

2) merge two lists: the first one is the length-M second-to-last element of the list and the second one is the popped last length-M element of the list (the cost of pop is O(1)). The cost of merge() grows from O(M) to O(MN) (the way to calculate is in the lectures) as the last element of the list grows in length. We take O(MN) as the worst case scenario.

3) repeat step 1) N times and step 2) N-1 times (as in the last step we are done and just return the sublist).

The total cost is $2O(N) + (N-1)O(MN) \approx O(MN^2)$.

func1B: it is a recursive function that takes in a list of N length-M lists with elements sorted in ascending order. If its length is 1, then our work is done as it consists of only one list with elements in ascending order - it returns that sublist. If its length is 2, then it merges the two elements and returns the merged list M. If it is of larger length, then we apply func1B() again and thus split it until we get to two lists which we merge (as $N = 2$ there), and then build our way back up by merging the lists of sorted elements. The last merge is as follows: if MN is even, we merge two length-$MN/2$; if MN is odd, then the first list is of length $(MN-1)/2$ and the second one is of length $(MN+1)/2$.

Computational cost:

1) get length of the list: O(1)

2) if length of list 1 (1 comparison), return the sublist: O(1)

3) if length of list 2 (1 comparison), merge the two sublists (the sum of number of elements amounts to MN): O(1) + O(MN)

4) else: we start by calling func1B() on the list that is split in two, and keep 'dividing' it by calling func1B() until we reach $N = 2$ and then we start merging:
$\frac{N}{2}$ merges of length-M sublists: cost $\frac{N}{2}O(2M)$, $\frac{N}{4}$ merges of length-$2M$ sublists: cost $\frac{N}{4}O(4M)$, $N/8$ merges of length-$4M$ sublists: cost $\frac{N}{8}O(8M)$, ..., until we get to a list with two elements, i.e., $N = 2$. Importantly, note that we do this recursive call $\log_2 N$ times.

The total cost is $NO(1) + (O(1) + O(MN)) + (\log_2 N)O(MN) \approx O(MN \log_2 N)$. Therefore, func1B() is computationally cheaper than func1A().

1(b)

The cost for func1A() is $O(MN^2)$, and for func1B() is $O(MN \log_2 N)$. We produce three figures to check how the execution time varies for different values of N and M. As we vary N, and keep M fixed, we that the time it takes to compute func1A() increases quadratically while for func1B() it is linear (the $\log_2 N$ is almost negligible for these sizes of N). As we fix N and increase M, we see that the computation time for both functions increases linearly. Finally, when we increase both M and N, we see a cubic increase for func1A() and a quadratic increase for func1B(). We see that the experiments justify our conclusion in part (a) that func1B() is computationally cheaper than func1A(). It is because func1B() only requires $\log_2 N$ levels of recursion, whilst func1A() requires a number that is of order N.

Figures

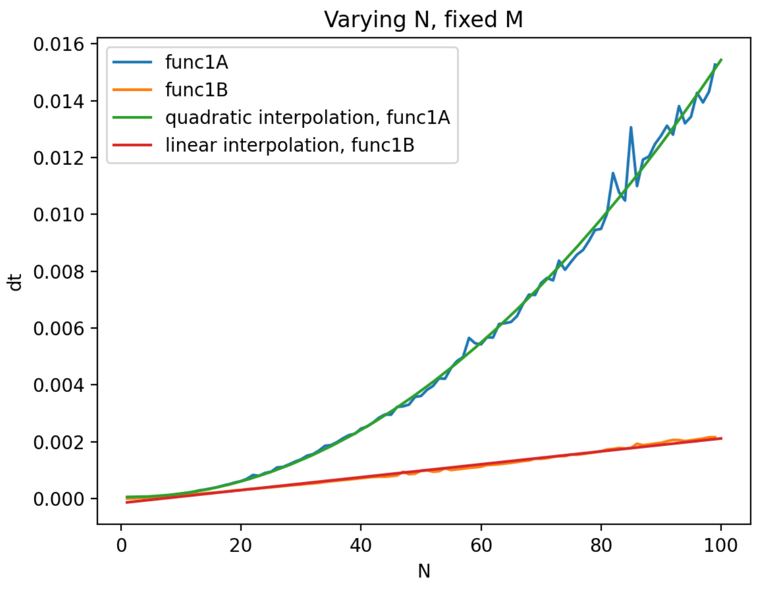

Figure 1: Figure for question 1, varying N

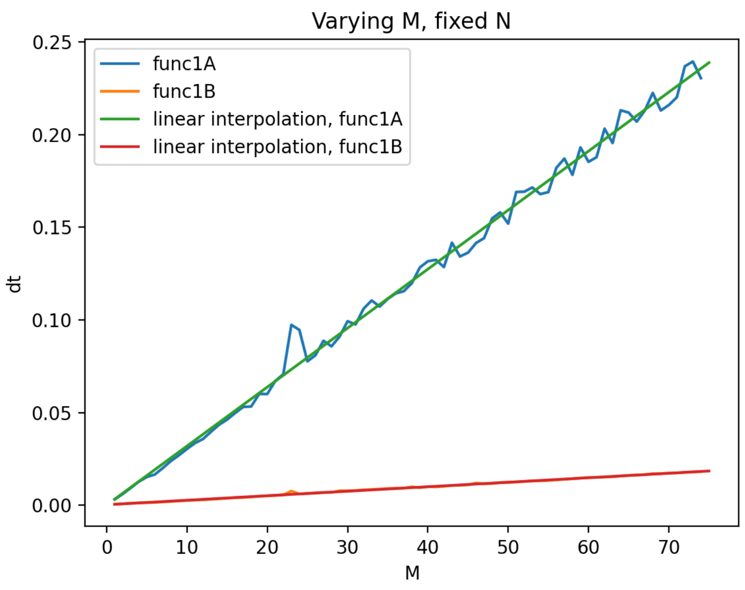

Figure 2: Figure for question 1, varying M

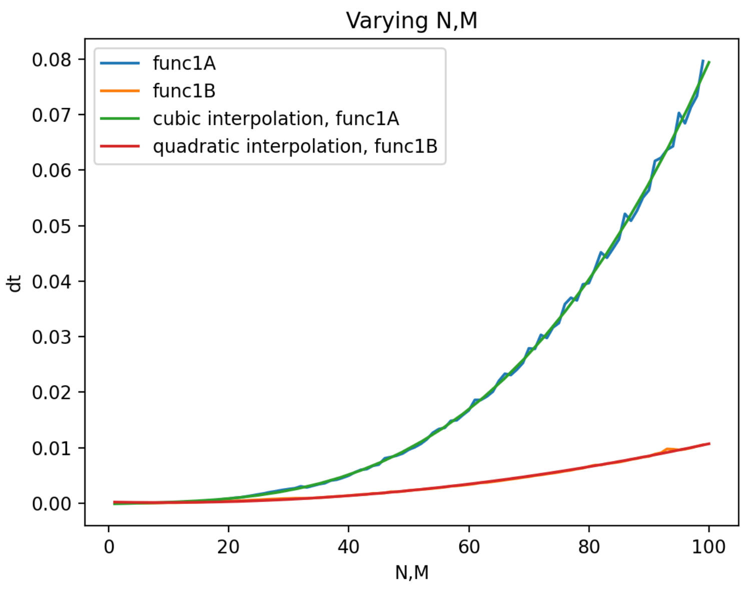

Figure 3: Figure for question 1, varying N and M

Question 2

2(a)

The code is in the file project1.py, the relevant functions are gene1(), which calls h_eval() and char2base() which are defined outside gene1() for easier reading of the code.

2(b)

The function gene1() takes as input the length-N string S that is the gene sequence - a combination of four characters A,C,G,T; L_in which is a list that contains P sequences of length M; and x, the location at which the mutation of each of the P sequences in L_in occurs.

Description of implementation of the function gene1():

- convert S to a list of base 4 integers using char2base4(), get its length
- create an empty list L_out to store the lists that contain the locations of the mutations of each element of L_in
- create a for loop, iterate over each element of L_in: convert the element of L_in into a list of base 4 integers using char2base4(), create an empty list to store its mutations
- now consider x, if it is -1, then append the non-mutated list to the list created to store mutations, of it is not -1, then modify it and append three mutations at location x
- create a list that collects the indices of the mutations, choose base=4, a large prime and apply the Robin Karp algorithm, updating the hash function (as opposed to calculating it at every index when iterating through S) and directly checking the strings when the hash values of the mutation and the substring of S are equal (we have seen direct comparison is quicker in the week 4 lab)
- the lists of indices where the matches occur for each mutation are appended to L_out
- after we have iterated through L_in, we get L_out which is a list of lists that contain three elements that are lists of indices of each of the three mutations in ascending order. We apply func1B() to each element of L_out and thus get the list L_out such that the ith element contains the indices where there are point-x mutations of the ith lentgh-M sequence stored in L_in, and this is the output of the function.

We check the running time in seconds with respect to variable changes: increasing N, that is running it on substrings of S and increasing M, the number of elements of L_in. The graphs below show that the increase in running time is linear in both cases. This suggests that the cost of the algorithm is $O(NP)$.

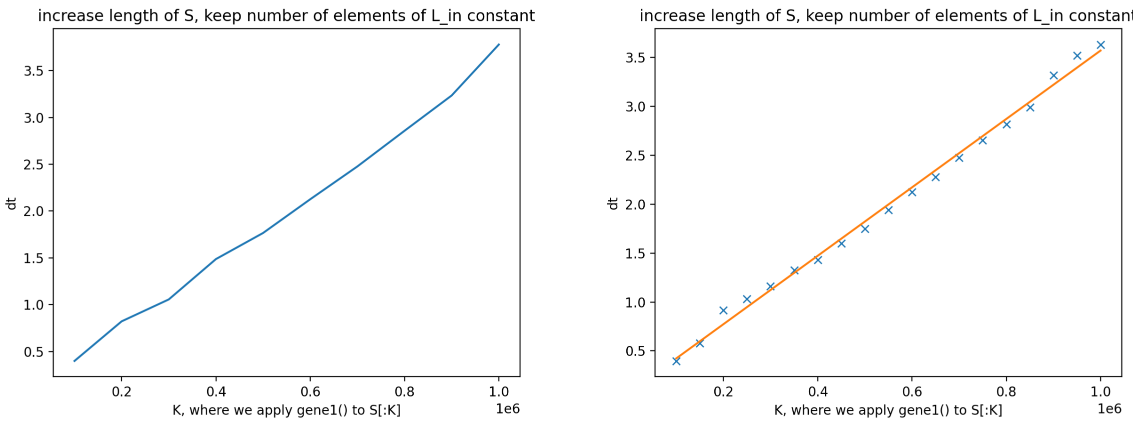

Figure 4: Increasing the length N of S, keeping M and P fixed. On the right the values with linear interpolation.

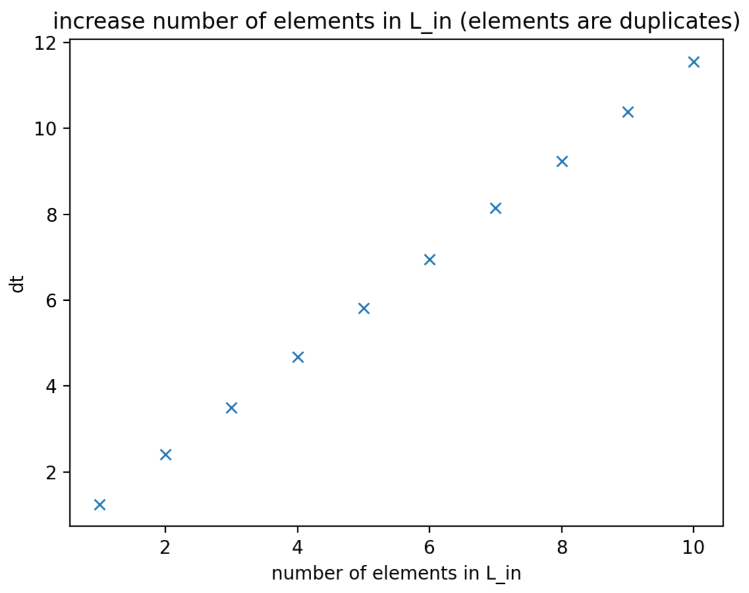

Figure 5: Increasing P, the number of elements of L_in, keeping M and N fixed - linear increase in running time

Below is the running time when we simultaneously increase N and P, which clearly displays a quadratic increase and is verified by a quadratic polynomial fit. This shows that the suggestion that the cost of the algorithm is $O(NP)$ holds.

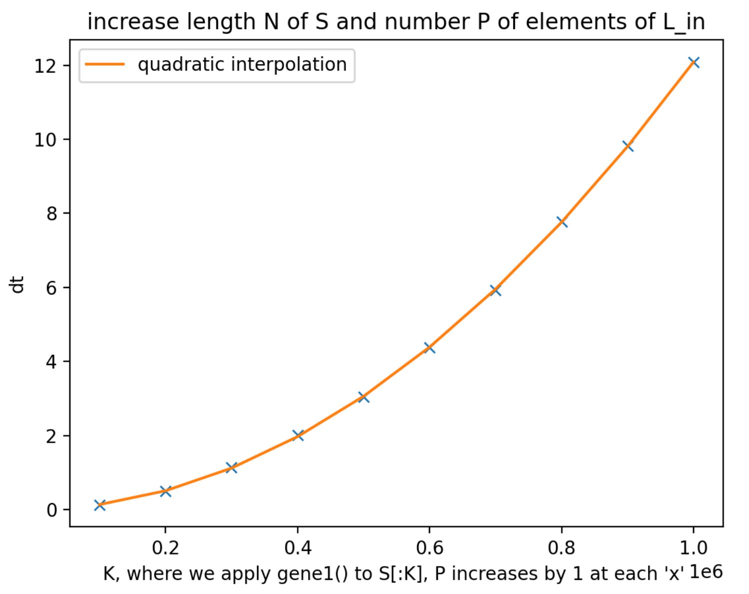

Figure 6: Increasing N and P, observe quadratic growth in computation time

This figure is the variations in running time when increasing the length of the strings we look for, M, and it seems that there is no clear relationship, only perhaps an indication that there may be an optimal range (in terms of running time) for the length of the (non-)mutated substrings that we look for in S.

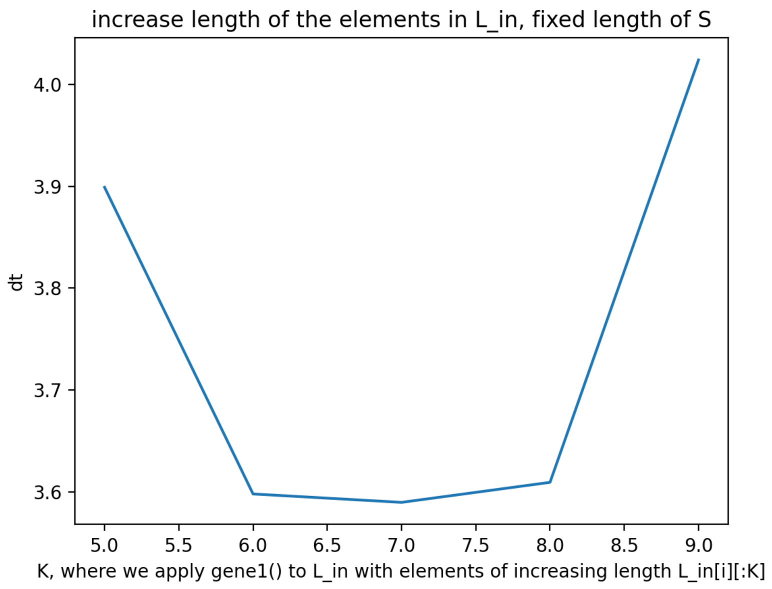

Figure 7: Increasing M, keeping N and P fixed