Scientific Computation Project 1

Jonathan Cheung CID:01063446

February 6, 2020

Part 1: Sorting and searching

1.1)

We are given a function 'newSort' that sorts a list of integers. The function 'newSort_test' takes as its input 'length', which determines the length of a randomly generated list, and 'k' which chooses the k threshold for the 'newSort' function which is applied to the list. This function then tests whether the elements of the output from 'newSort' are monotonically increasing. If there is an element that is not, then it returns false, otherwise it returns true. 'newSort_batchtest' will apply 'newSort_test' with a randomly generated k value to a randomly generated list and store this output along with the length and k value into an array. The code 'newSort_batchtest(100) in under if __name__=='__main__' will create 100 random lists and check the accuracy of 'newSort'. This will verify that the function works correctly.

The function 'newSort' has two inputs. We want to test its efficiency as these inputs are varied. We will test the run times to sort randomly generated lists of varying lengths with fixed k, and then we will test the run times for a fixed list of fixed length for varying k. We will use the function 'time_newSort' to test the sorting times. This function has an input 'variety', and setting this to 0 or 1 will cause the function to vary length OR k value (respectively). Before we start we will do a run with list length varying from 1-2000 with the default k value of 0 (fig 1). While we can see a trend, the line is very staggered. This is due to the randomly generated lists which are varying levels of being sorted. To smooth this out, for all future runs, we will input an 'average' value into the function and this will do multiple runs and average the time for each length value. An alternative method to make the trends easier to see would be to take very large list length.

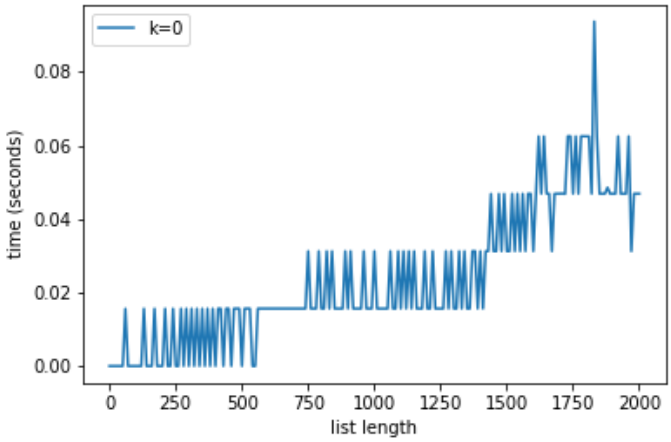


Figure 1: Time to sort list of varying lengths with fixed k and no averaging

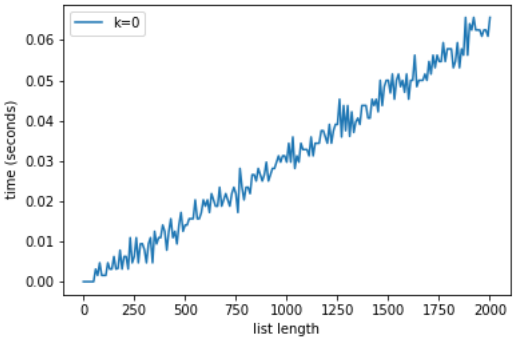
Now, the function 'time_newSort' will produce a series of plots (fig 2). These plots represent the average (averaged for 10 runs per length) run times for sorting lists of lengths varying from 1-2000 with fixed k value. In fig 2a we can see some trend for the time complexity for k=0. At first, we would guess this is a linear relationship between list length and running time. However, upon inspection of the 'newSort' function, we can notice that this function is just a merge sort, when k is set to 0. We also know that merge sort has a complexity of O(nlog$_2$ (n)). So, we can conclude from the graph that 'newSort' with k=0 has this time complexity.

Fig 2b shows the times for when similar lists are sorted with k=5 plotted on top of the previous graph. We can see that choosing k=5 speeds up the sorting time when compared to k=0. Choosing k=10 doesn't seem to reduce the sorting time however (fig 2c). And we can see that k values of 20 and 40 (fig 2d, 2e) start to increase the sorting time. One last thing we can check is to see what happens when we take k such that k is greater or equal to all these lists. When we have such k, 'newSort' acts like an insertion sort and will have complexity O(n^2). The k value is essentially a threshold for which the list and sublists stops being divided into further halves and insertion sort is used.

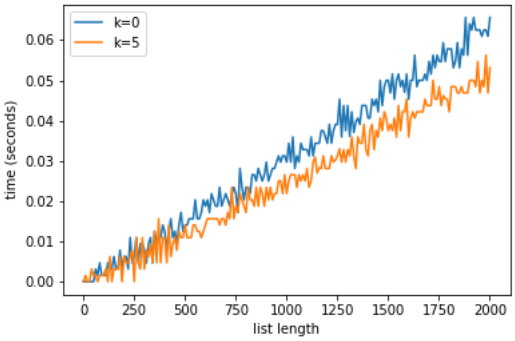
The time complexity for $N > k > 0$ can be calculated by finding the time complexity for the dividing steps, the insertion sorting steps and the remerging steps. This is of order nlog$_2$(n/k) + nk. The optimum value of k seems to lie between 5 and 18 according to this graph. Again, the optimum value is the value which the machine can sort the k length lists faster with insertion sort

than merge sort (and then remerge these lists). To find out when this happens we need to solve xnlog(n) = yn^2 for some constants x and y which are machine dependent. Although we do not know what these constants are, assuming that they are of not too different magnitude, choosing a k value from 4 to 20 does seem to make sense.

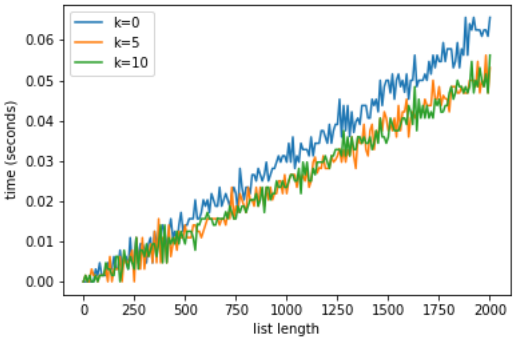
Fig 2f shows the plot when k has been set to 2000. This plot is not as smooth since, the runtimes are significantly longer and calculating an average would take a long time. So these spikes in the graph can be explained as a result of the randomly sorted nature of lists being generated. Nevertheless, we can see some non linear trend here. Also note that y axis scale is a lot larger than in previous graphs, so this non-linear increase is a lot steeper than it appears. Our choice of k=2000 has caused 'newSort' to act like insertion sort which we know has $O(n^2)$ complexity, which agrees with this graph.



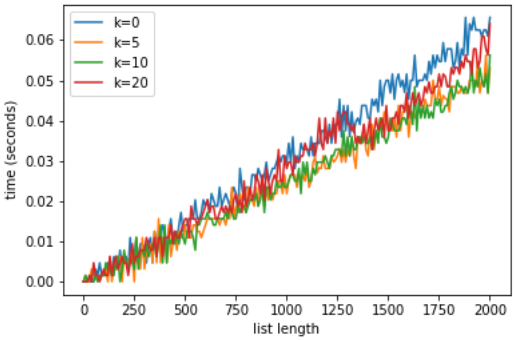
(a) k=0



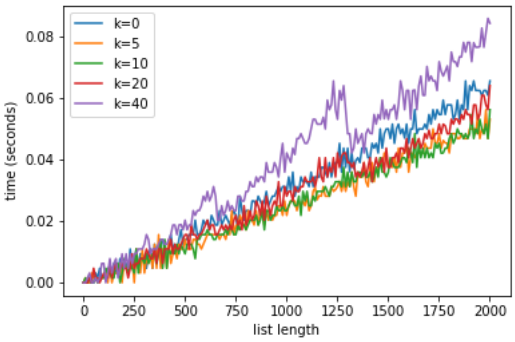
(b) k=0,5



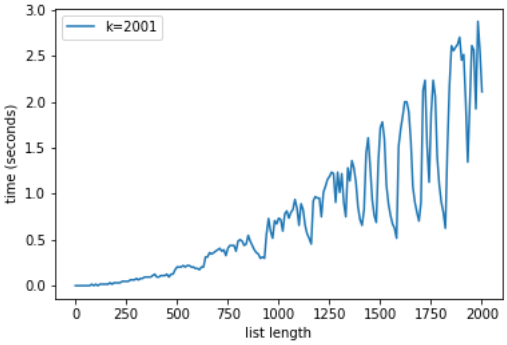
(c) k=0,5,10



(d) k=0,5,10,20



(e) k=0,5,10,20,40



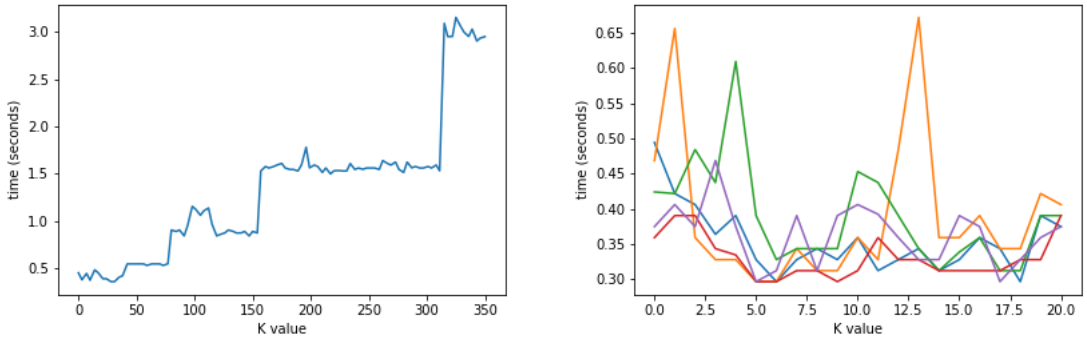
(f) k=2001, not averaged

Figure 2: plotting sorting time over list length for some fixed k

Intuitively we can see that there should exist a choice of k that will optimise the sorting time since there will exists a length of list such that insertion sort is faster than merge sort. We will now plot a graph that compares the sorting time with choice of k-value using a fixed list of length 2000. Fig 3a is a graph that shows the sorting times as k varies from 1 to 350. If we ignore the small fluctuations that are due to fact that the lists is in a randomly sorted, it can be seen that the time steps up when the k value doubles. The steps occur at around 40, 80, 160, and 320. Again, the k value is threshold for when to stop halving the list. For a list of fixed length, increasing the k value only has a significant increase in sorting time if it results in the list being halved one less time, since this means that more insertion sort has to work on larger lists. So this step behaviour can be explained by the fact that the k value has to double for the list to have one less divide and conquer step and more insertion sort steps.

This graph also agrees with our previous graphs; choosing k beyond 20 will cause sorting time to increase. From fig 3a it appears that the optimum k value is somewhere between 0 and 20. So we will now plot the results from a few sorting runs with k varying from 0-20. In fig 3b, each line represents a different randomly generated list and its sorting time for each k value. There doesn't appear to be a single k value that minimises the sorting time for every list. In fact the best choice of k value seems to be anywhere from 5-18.

In short, newSort is identical to mergesort when k=0. As k increases to this 5-20 range, newSort is quicker than merge sort since we have found a size of sublist such that we no longer need to divide and conquer any further. Once we increase k past 20, the sorting time increases and this increases all the way up to k equalling the size of the list and then newSort behaves like insertion sort, which is substantially slower than merge sort.



(a) k up to 350

(b) k up to 20

Figure 3: sorting times when k is varied for a fixed list of length 2000

1.2)

The function 'findTrough' takes as its input a list L of integers and find troughs. These are elements in the list that are less than or equal to the adjacent elements. The function works like a binary search. First we take the middle element of the list and compare it to its neighbours. If this element is a trough, then we are done and this element index is returned. If this element is not a trough, then it will have to be greater than at least one of its adjacent elements. We can now search the half of the list containing the smaller element for a trough recursively. This function will always find a trough. We explain this as follows: We imagine the integers of the list plotted on a graph (with integer value on the y-axis and integer index on the x-axis) and assume without any loss of generality that the element to the right of the middle element is less than the middle element, The line will dip as we move from the middle element to its neighbour on the right. Either the line will eventually reach a point where it moves up/increases again, or it will continue decreasing down towards the last list element. In both these cases there exists a trough. The worst case performance of this function is if we have to halve our list the maximum number of times until we have one element left which is the trough. So we can calculate time complexity by solving $1 = \frac{N}{2^x}$ which happens when x = $\log_2$(N) where N is the length of the list. Hence the asymptotic time complexity of this function is O($\log_2$(N)).

Part 2: working with DNA sequences

2.1)

The function 'DNAtoAA' will convert a string of distrinct amino acids corresponding to the codons in its input 'S' in the order that they first appear in S. This function uses a dictionary from the function 'codontoAA' to hash search. Searching for items in a python dictionary is of constant time so the complexity is O(n) since we loop through (every third element of) a length n list. This is confirmed with the graph below which shows a linear relationship.

Figures

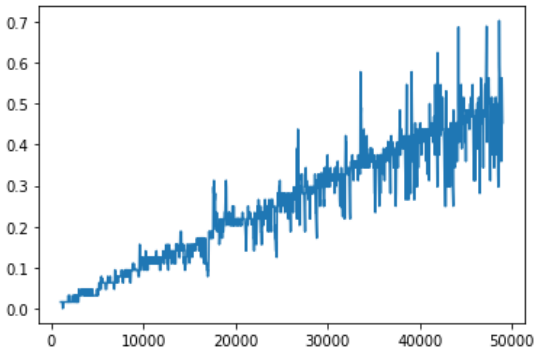


Figure 4: Time to convert genetic sequence of length N into a string of amino acids for different length N

2.2)

The function 'pairSearch' will find all locations of k-mer pairs from 'pairs' in the list 'L' of DNA sequences and return the indexes of the starting location of the pattern in the DNA sequence, the index of the first of the two DNA sequences in 'L', and the index of the pair in 'pairs'. This function uses the Rabin-Karp algorithm. We convert the strings from the pairs into base 4 and then into base 10 hash patterns. Next we compare the pattern of the first pair to the rolling hash pattern of the DNA sequences in 'L'. If there is a match, we first check for a hash collision before checking the second pattern of the pair in the same location of the sequence below. If this matches again we save these indexes to a preallocated list 'locations'. We have used a prime of of 97 to speed up arithmetic. The method implemented here is significantly faster than a naive search. This method loops through N-k+1 patterns (where N is the length of the DNA sequence and K is the length of the pair elements) in L-1 sequences (since the last sequence does need to compared) for p pairs. So we can say that the order is of (N-k+1)(L-1)p. For a naive search, we know that for a

single length K pattern and length N list, the complexity is O(KN). But in this scenario there is a list of p patterns and a list of L sequences. So this complexity is of order KN(L-1)p.