Scientific Computation Project 1

Leon Wu 01190736

February 6, 2020

Part 1

1.1)

Correctness and efficiency of newSort

Case (1):
For a given input X, with $n = len(X)$, and for the case $n <= k$, the algorithm iterates through each integer (apart from the last one, since the iteration on the penultimate integer means that the last integer is already 'sorted' (it is the largest integer). At each integer i, the algorithm iterates again starting from the next integer to the last integer in the list, comparing it to integer i. This loop finds the smallest integer after integer i and swaps the smallest into the place of integer i. In this way, the sorted list is built up from the start systematically and the list will be sorted correctly. The i loop has n-1 iterations, and inside there is 1 assignment, and a swapping of numbers in a list (unless integer i is already sorted). The j loop has $(n - i - 1)$ iterations (on average about $(n - 1)/2$), and contains a comparison and an assignment in some cases. Overall, cases (1) contains (in the worst case) $(n-1)(3+(n-1)/2*3) = \frac{3}{2}n^2 - 3n + \frac{3}{2}$ operations. Asymptotically as n gets large, we have that the worst case time complexity is $\mathcal{O}(n^2)$, since the lower order terms become insignificant. The average and best case complexity is the same, since the algorithm still has to perform both loops if the list is already sorted, with the only time save being that ind_min is never reassigned to j.

Case (2):
For the case that $k = 0$, this is just merge sort, since the algorithm splits the list into two halves recursively until the length of the lists are all 1. Then, each pair of lists are merged so that they are sorted, to make lists of length 2. This process is repeated until the final two lists are merged into the final sorted list, which is returned by the function. Each merge is achieved in $\mathcal{O}(n)$, and the merge must be performed about $log_2(n)$ times, so we can see that merge sort has a worst-case time complexity of $\mathcal{O}(nlog(n))$. The best and average case

time complexity is the same, since no significant time savings are made.

Case (3): For the case that $0 < k < n$, the algorithm will do something 'in-between' the sorting that I described in the first case, and merge sort. The algorithm in this case will split the list in half, calling newSort() on these smaller lists. These smaller lists will either be split again if the lengths of the smaller lists are still larger than k, or sorted as in the first case if not. Cases such as the empty list and negative entries clearly work with this algorithm, and were tested (with k positive). With all this being said, this algorithm can guarantee that this algorithm will correctly return a sorted list (as long as the input is of sensible size and type that we expect). We could also implement assertions to check whether the inputs are of the expected size and type to make this algorithm more robust. The worst-case time complexity for this case will be between between $\mathcal{O}(nlog(n))$ and $\mathcal{O}(n^2)$ (same for average and best case).

Clearly, this newSort algorithm has a worse worst-case asymptotic time complexity than mergesort, unless $n = k$ when newSort is merge sort. However in some cases, a small value of k makes the algorithm quicker (see figures below). This is because with small lists, the algorithm implemented for case 1 could be faster in some cases, since the lower order terms in the number of operations become more significant. (For numpy arrays, which were used in this case, the performance increase is harder to see than for python lists for small k.)

Discussion of Figures

The following figures support the discussion above.

- Figure 1 shows how the running time varies for increasing N. We can see that for merge sort, the time increases with N. For k=N and k=N/2, the graph increases with N at a rate that is faster than linear, with the line when k=N increase the fastest

- Figure 2 shows the quadratic relationship between N and the running time with k=N. The plotted line fits very well to a quadratic line so it is consistent with the analysis.

- Figure 3 shows the given relationship between N and the running time with k = 0. The plotted line again fits very well to the reference line that scales with $Nlog(N)$ so it is consistent with the analysis.

- Figure 4 shows how the running time varies as k is increases. Note that it levels out when $k >= N$ since increasing k past this point has no influence in the behaviour of the algorithm. The 'intermediate' plateaus are due to the fact that similar values of k do not influence the behaviour of the algorithm unless the value of k matches the lengths of the lists at some layer of the algorithm. Clearly, as k increases, the time increases since we move from case 2 towards case 1.

- For low values of k, say $1 <= k <= 9$, the algorithm is actually slightly faster then with $k = 0$, as discussed.

Figures

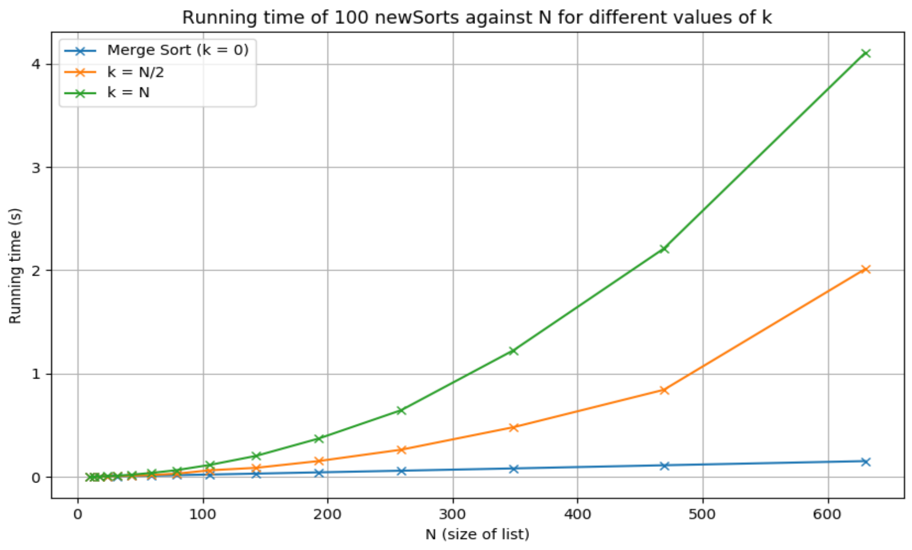

Figure 1: Running time of newSort against N

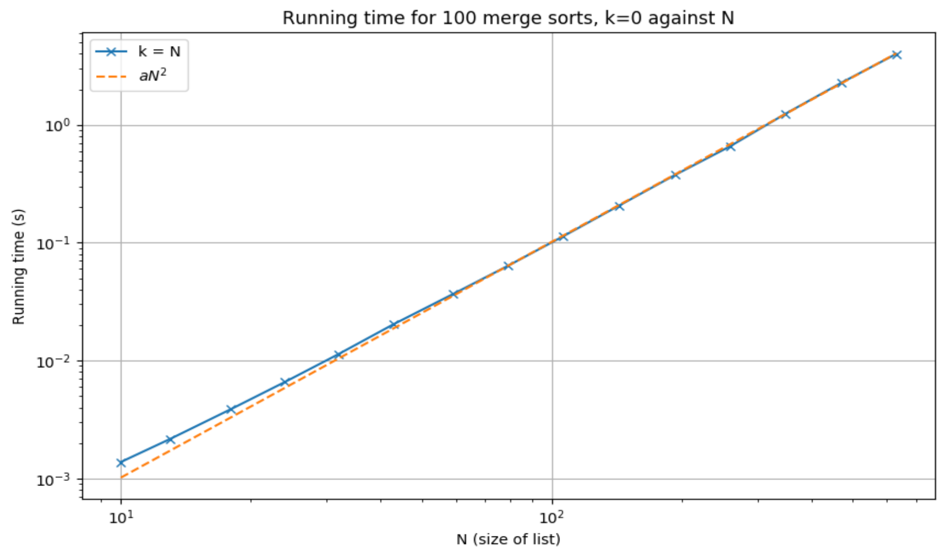

Figure 2: Running time of newSort against N with reference

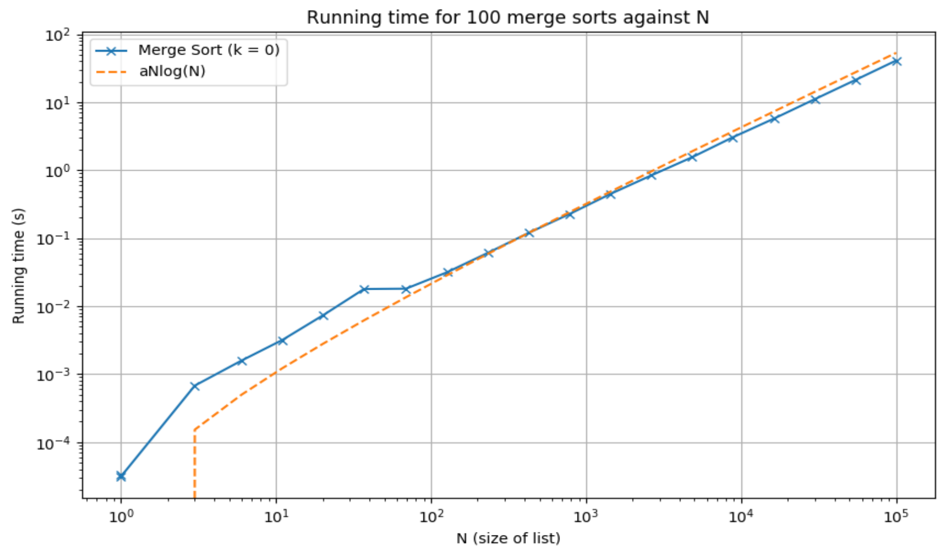

Figure 3: Running time of newSort against N with reference

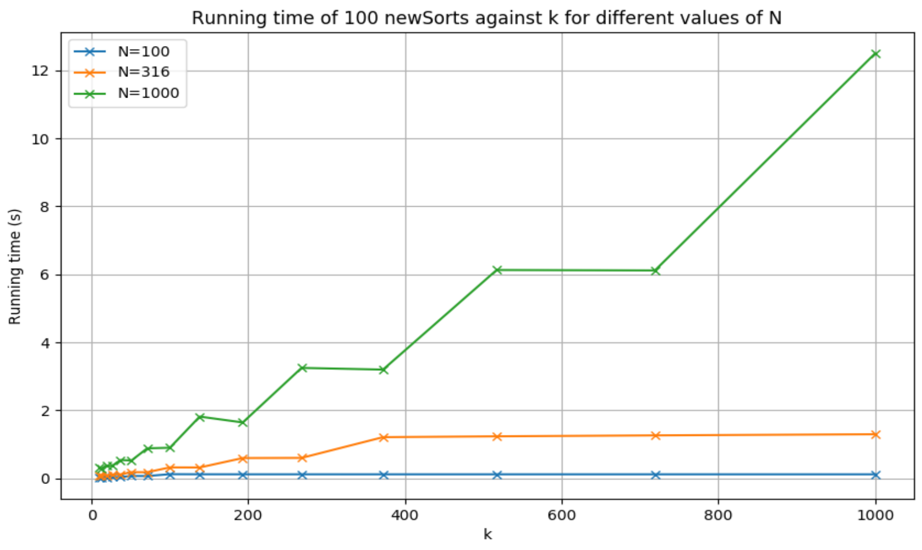

Figure 4: Running time of newSort against large range of k

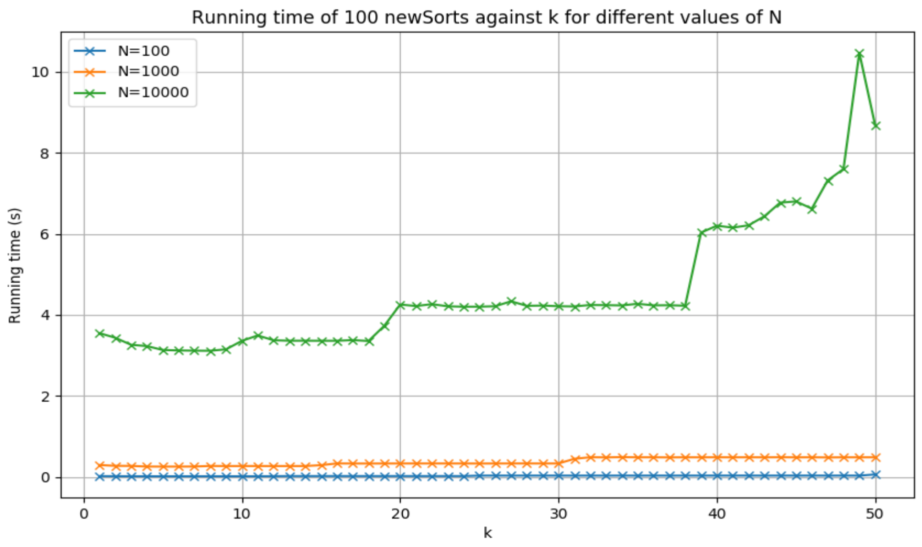

Figure 5: Running time of newSort against small range of k

1.2)

Note first that for $N >= 2$ there must exist a trough. This can be seen by considering a list with no troughs. Then the second element of L must be less than the first otherwise the first would be a trough. This argument continues so that the sequence is strictly decreasing. But then the last element in the list sequence is clearly a trough (assuming N is finite). In the cases $N = 0$ and $N = 1$ the function returns -(N+1).

The algorithm uses an (iterative) binary search in order to find the solution. One of the troughs will always be found, and is implemented in the following way:

- Initialize istart and iend indices at the extremes of the list
- If $iend <= 0$, then the list is either size 0 or 1, so return -(N+1)
- Perform a binary search. Check the middle number and the number next in the sequence.
- If the sequence is increasing, then there exists a trough on the left side of the sequence (which follows from the argument I gave above), so reduce the search to the left side
- Similarly if the sequence is decreasing, then there exists a trough on the right side, so reduce the search to the right side.
- If the sequence is flat, then there exists a trough on both sides, so reduce the search to either side (the left side is chosen in this case)

At each iteration, the search space is reduced by half. So then the worst-case time complexity for the algorithm is $\mathcal{O}(log_2(n))$. This is the same for the average and best case time-complexity, since the algorithm never terminates before the search space has been fully reduced (except for the cases where the list is of length 0 or 1). We could very roughly count the operations in each iteration; there is an assignment, a calcualtion of imid, a comparison and another assignment, so that the number of operations is roughly $log_2(n) * 4$.

(On a side note, if there were a high frequency of troughs (for example if the numbers in the list were randomly generated independently from a uniform distribution between -1000, 1000), then a linear search would have a significantly better average case efficiency compared to binary search, however its worse case is $\mathcal{O}(n)$ which is worse than binary search, and this is generally what we are interested in.)

Part 2

2.1)

The checking of amino acids already in the string AA can be achieved in constant time by using a hash table. In this case, I used a python set. The adding of letters to the string AA and also to the set is of constant time. The implementation loops through the input string S in jumps of 3. (I am assuming that this string slice operation is $\mathcal{O}(1)$, which is supported by the figure below, at least for $N < 3,000,000$). The amino acid is found with codonToAA, indexing the correct 3 letters. Then the amino acid is looked up in the set (in constant time). If it doesn't exist, the amino acid is added to the set and the end of the output string. The algorithm is therefore of worst-case time complexity $\mathcal{O}(N)$, where N is the length of the string. The best and average case time-complexity is the same, since the only time save is when repeat amino acids are found and don't have to be added to the set and the string.

The number of operations can be roughly calculated as $N/3*(3+1+1+1) = 2N$, since there are $N/3$ iterations, a string slice (assumed to be 3 operations), a condition check and an addition to both the set and the string.

A quick figure illustrates the linear time complexity in N:

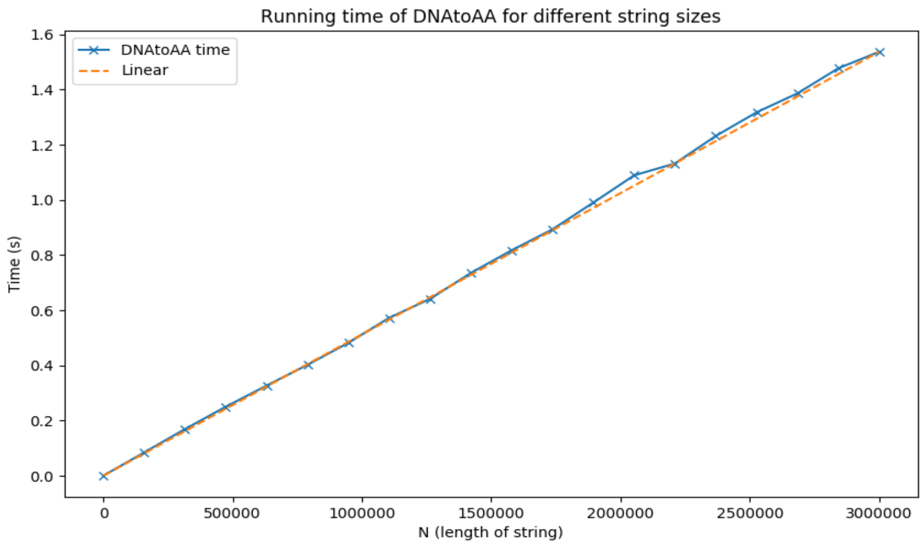

Figure 6:

2.2)

A brief outline of the algorithm is as follows, with L the list of M N-length sequences, and pairs the list of J K-length pairs:

- Convert all sequences and pairs into base 4 by mapping [A,C,G,T] onto [0,1,2,3]. **Running time: $\mathcal{O}(MN + JK)$ since the dictionary lookup is of constant time to convert a letter into a digit.**

- Create a dictionary, where the keys are the hashes of the first of each pair, and the values are dictionaries consisting of {second of the pairs : list of pair indices}. In each case, the hashes are checked in the dictionary and are created/added to. This hash function converts base 4 integers into base 10 integers. There is no need to mod the hashes since there in no performance penalty associated with large integers, and also this means there are no hash collisions possible since each K-length string uniquely maps to its hashed representation. **This part is $\mathcal{O}(JK)$, since again the checking of the hashed pair and adding to the dictionary is done in constant time, and since the hashing function runs in $\mathcal{O}(K)$ and is repeated J times.**

- The hashes for first K-mers in all sequences are calculated. $\mathcal{O}(MK)$. Then, Iterate down each sequence, calculating the rolling hash as in the Rabin-Karp method. For each sequence, check whether the hash exists in the pairs dictionary. If so, check all corresponding pairs in the next sequence, and if there is a match, append to the output. There is no need to check for hash collisions since K is constant, as discussed earlier. The code is arranged so that the K-mers in the sequences are only hashed once to avoid any duplicate calculations. Roughly, this is $\mathcal{O}((N-K)MJ)$ since the rolling hash is calculated in constant time, the matches are checked in a dictionary so is also constant time. The dependence on J here is actually eliminated in the case where there are no exact repeated pairs, such that index_list is of length 1, which will be assumed.

- Overall, we have a worst-case run time of $((N-K)M+MN+JK+MK)$. Asymptotically as N and M get large, and since $N >> K$ and $M >> K$, we have an asymptotic worst-case run time of $\mathcal{O}(NM + J)$. The naive approach discussed in lectures would loop through each sequence, and check one character at a time for matches with all the pairs. This would be $\mathcal{O}((NM * K * J))$, as discussed in lectures for the worst case with many near misses, but in this case M sequences must be checked for J patterns. Asymptotically as N and M get large, and since $N >> K$ and $M >> K$, we have an asymptotic worst-case run time of $\mathcal{O}(NMJ)$.

- In summary, naive search runs in $\mathcal{O}(MNJ)$ whereas my algorithm runs in $\mathcal{O}(MN + J)$, with the smaller terms omitted. Clearly the latter is preferred since the J term is additive rather than multiplicative. Also, my algorithm has no NK term whereas the naive search does, which could add significant time to the naive search algorithm compared to mine.