Scientific Computation Project 1

Matthew Cowley, CID 1059624

February 6, 2020

Part 1

1.1)

a) and b)

newSort works in the say way as merge sort apart from when it gets to a sublist of size k it swaps elements to find the order, instead of merging lists. The way it sorts these is via selection sort. newSort is the same as merge sort for k=0,1.

$$\text{operations in selection sort} = \frac{n(n-1)}{2} \implies O(n^2)$$

$$\text{operations in merge sort} = n + n\log_2(n) \implies O(n\log_2(n))$$

But for large N merge sort is far better performing at $O(n\log_2(n))$ compared to selections sort's $O(n^2)$. So we expect k to be small, where smaller order terms will have more effect, where selection sort is better than merge sort. Thus for some small optimum k we expect Newsort to slightly outperform merge sort, but both still have $O(n^2)$.

Literature suggests that selection sort is better than merge sort for n between 7 and 50. Solving number of operations above, gives you 9.494. So the optimum k should be around 9.

c) and d)

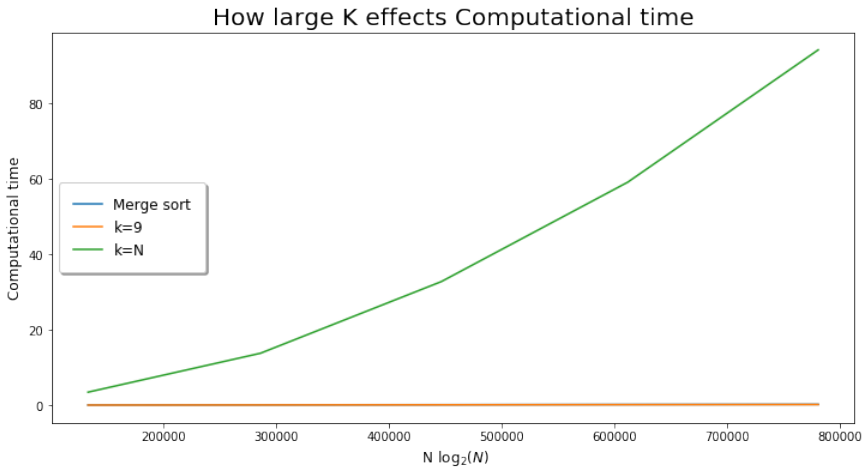

Figure 1: How newSort performs with k=N, as $N \log_2(N)$ varies

newSort does not operate to $O(n \log_2(n))$, for k=N, represented by the curved line.

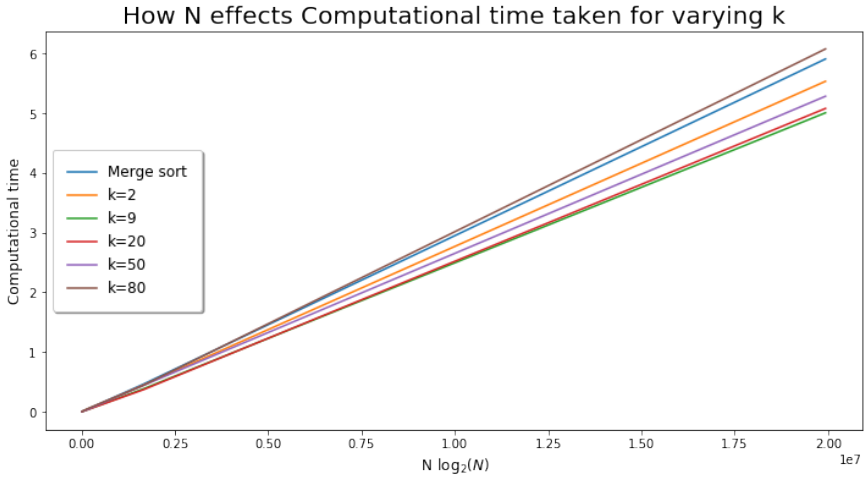

Figure 2: How varying k effects computational time

All results appear to be $O(n \log_2(n))$. This would suggest for low k we have $O(n \log_2(n))$, represented by the straight lines, agreeing with the analysis. For K=80 we get worse results then merge sort result which is expected and the best results are given by k=9, which agrees with the analysis in a) and b).

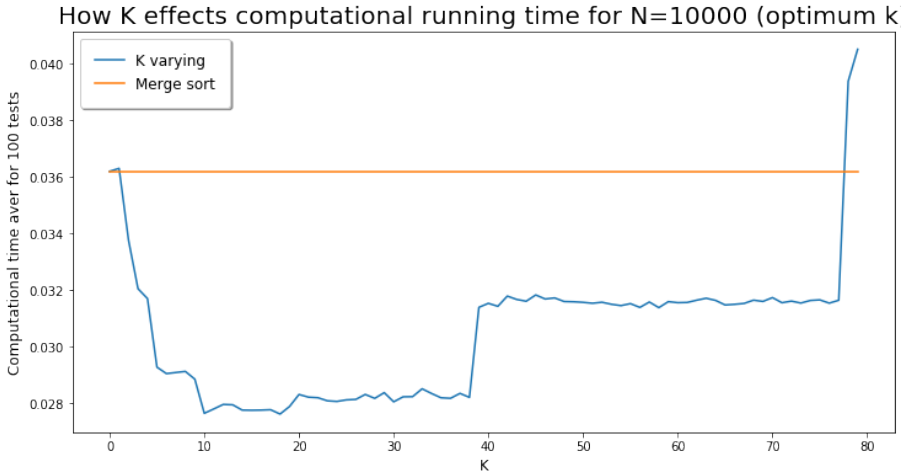

Figure 3: How k effects computational time for N=10000

Interestingly the optimum appears to be 9 or 18, which does agree with the literature, interestingly Newsort is better than Merge sort all the way up to k=78, which is higher than expected.

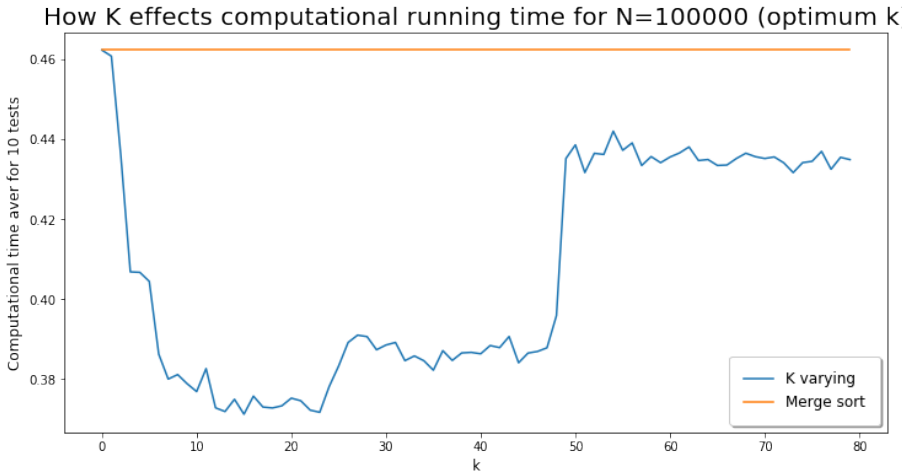

Figure 4: How k effects computational time for N=100000

When we increase N, k appears to have a more random effect. But what is very interesting, is for k= 80, newSort is better than merge sort, which is unexpected. Perhaps there is link between increasing N and how k effects the algorithm? it could perhaps be the down to the way the random list of integers is created. Further analysis would be required however.

1.2)

In the context of the problem you are essentially trying to find a local minimum of a list N integers, so if you find a minimum have solved the problem. The min function in python has Θ(N), So O should at least be N.

The Algorithm I have implemented first checks the number at start and end of the list are not troughs. It then search's through the index's using binary search logic, checking if a index is trough, in the process choosing an appropriate direction (to reduce size of list effectively). The appropriate direction is the smallest adjacent integer toa given index. The algorithm breaks when it has found a trough. This algorithm has roughly $6 \log_2 N$ operations in the worst case (when the list is small enough it search's through it using a naive approach). So for large N, it has $O(\log_2 N)$.

The benefit it has over binary search is the best case scenario, as for binary search it would be $\Omega(\log_2 N)$, where as for this new one it is $\Omega(1)$. It will very rarely be of order $\log_2 N$, generally being lower.

Part 2

2.1)

It iterates through the list, 3 letters at time, converting 3 letter string into a amino acid, using a codonToAA (which uses a hashtable, which is constant in time). It checks if this amino acid is in a set (another python hash table), if not add it to the set and add to a string which will be the final output.

Going through the list takes $\frac{n}{3}$ operations and searching the hash table is negligible for large N, as it has max size of 64. The overall number of operations is approximately $64 \times 64 \times \frac{n}{3}$ in the worst case, so for large N is $O(N)$.

2.2)ii)

Notation let n be the number of sequences and N length of sequence. Let m be number of pairs and k the length of pair.

My function works by first converting the sequence in L, into integers 0,1,2,3 instead of strings. This takes N*n operations and is effectively constant in time.

It then converts the pairs list into 2 dictionarys, where the key is the hash value calculated using base 4 used on sequences. e.g P1=[0,2,3,0] would be (k=4 here) :

$$0 * 4^{k-1} + 2 * 4^2 + 3 * 4 + 0$$

It then searches through one sequence at a time using rabin karp method omitting modulo arithmetic as we are interested in large N and the problem is in base 4, if in base 26 or k large it would be beneficial to implement modulo part.

However introducing the modulo arithmetic means we would have to check for hash collisions.

If a given hash value is in of the dictionary of pair 1 , it updates an index dictionary storing its hash value, with the key being an index. If a given hash value is in dictionary of pairs 2, it checks previous sequences dictionary of indexing, if they have an agreeing item, then pairs are aligned and it appends relevant info to locations.

Iterating through the lists takes and calculating hash values takes roughly n*(N+k)*k operations.

Checking if each hash value is pairs list takes m*2 and then searching index list depends on how many meaningful amino acids are detected in previous sequence, but this is a dictionary, so should be constant in time. Then calculating intersection of sets should be negligible (proportional to how many items in pairs list are repeated e.g [('TAG','TGA'),('TAG','GTA')] would increase this time). But overall. All of these operations use dictionary's which should not be too large, so should be constant in time. But for approximation, let us assume they take operations of size 2*m, ignoring operations taken by index and intersections to be negligible (will not always happen anyway).

So overall number of operations is n*(N+k)*k*m*2 giving O(n*k*(N+k)*m), however we are assuming that $N, n >> k$. The method would break down for large k, as computer wouldn't be able to store large numbers calculated in the hash value, (would have to use modulo Rabin Karp but then would have to check for hash collisions so might as well use naive search.)

But as result this method takes $O(N * n * m)$ but generally $m << N$, so we get O(N*M) which is similar to naive approach. The number of operations for naive approach would be n*N*k*k*m*2 instead of n*(N+k)*k*m*2.
