# Scientific Computation Project 1

*CID: 02099078*

November 2, 2023

## Part 1

### 1.

The function part1 combines insertion sort with both linear search and binary search. It takes a list of integers and an integer istar as inputs. It iterates through the elements in the list. The variables i and x represent the current index and the value at that index respectively. If i is less than or equal to istar, a linear search is performed in order to find the correct index for the element x. It searches for the first index where x is greater than or equal to the element at that index, and then updates the ind variable to the index after it. If i is greater than istar, a binary search is performed in order to find the correct index for the element x. In the code, a and b represent the range of indexes that are being searched, and c is the index in the middle of both. The value at index c is compared with x. if x is larger, then the binary search is continued in the upper half, whereas if x is smaller or equal to the value at index c, the binary search is continued in the lower half. This is repeated until the right index is found, and then the ind variable can be updated to it. After this, the function inserts x into a sorted list by shifting elements to the right to make space. This process is repeated for every element in the original list, resulting in the list being sorted in non-decreasing order.

To analyse the worst-case computational cost of part1, we look at the different parts of the function. In the searching part of the function, either linear search or binary search is used. Linear search has a worst-case time complexity of O$(N)$, where it has to compare each element to x. Binary search has a worst-case time complexity of O$(log(N))$, when the maximum number of comparisons are made due to the element being either the smallest or largest in the sorted list. For the insertion part of the function, insertion sort is used, which has a worst-case time complexity of O$(N^2)$, when the list is in decreasing order. Therefore, the insertion sort is the most influential factor in determining the time complexity of the function in the worst-case scenario. If $istar = N - 1$, then the insertion sort is performed on the entire list. This means that overall, the worst-case computational cost of the function part1 is O$(N^2)$. The asymptotic time complexity in this case is only dependent on the size of the list, N. When $istar < N - 1$, insertion sort is not performed on the entire list. In this case, the linear search and binary search are more influential in determining the asymptotic time complexity of the code. For these values of istar, it is approximately O(N) since $N > log(N)$. This is again dependent on the size of the list, N.

My analysis shows that if there is no information regarding the list that needs sorting, a very low value for istar will be more efficient as binary search has a better time complexity than linear search in the average case, and so increasing the use of binary search is beneficial. However, if there is information that the list is partially sorted, the efficiency of the sorting algorithm can be improved by varying the amount of linear and binary search that the code implements in relation to the parts of the list that are sorted.

**2.**

My first test investigates how changing the size of the list, N, affects the wall time of the function part1 for three different values of istar. Using a similar approach, my second test examines the dependence of the wall time on the value of istar. The first test begins by initialising an empty list of wall times. Random input data is then generated for part1 to be tested with. The time module is used to record the time before and after calling part1, allowing the wall time to be measured by calculating the difference between these two times. The resulting value is appended to the list of wall times and the process is repeated for each list size, N. After the data has been collected, a least-squares fit is computed in order to estimate the time complexity of the function. The data is then plotted on a logarithmic scale.

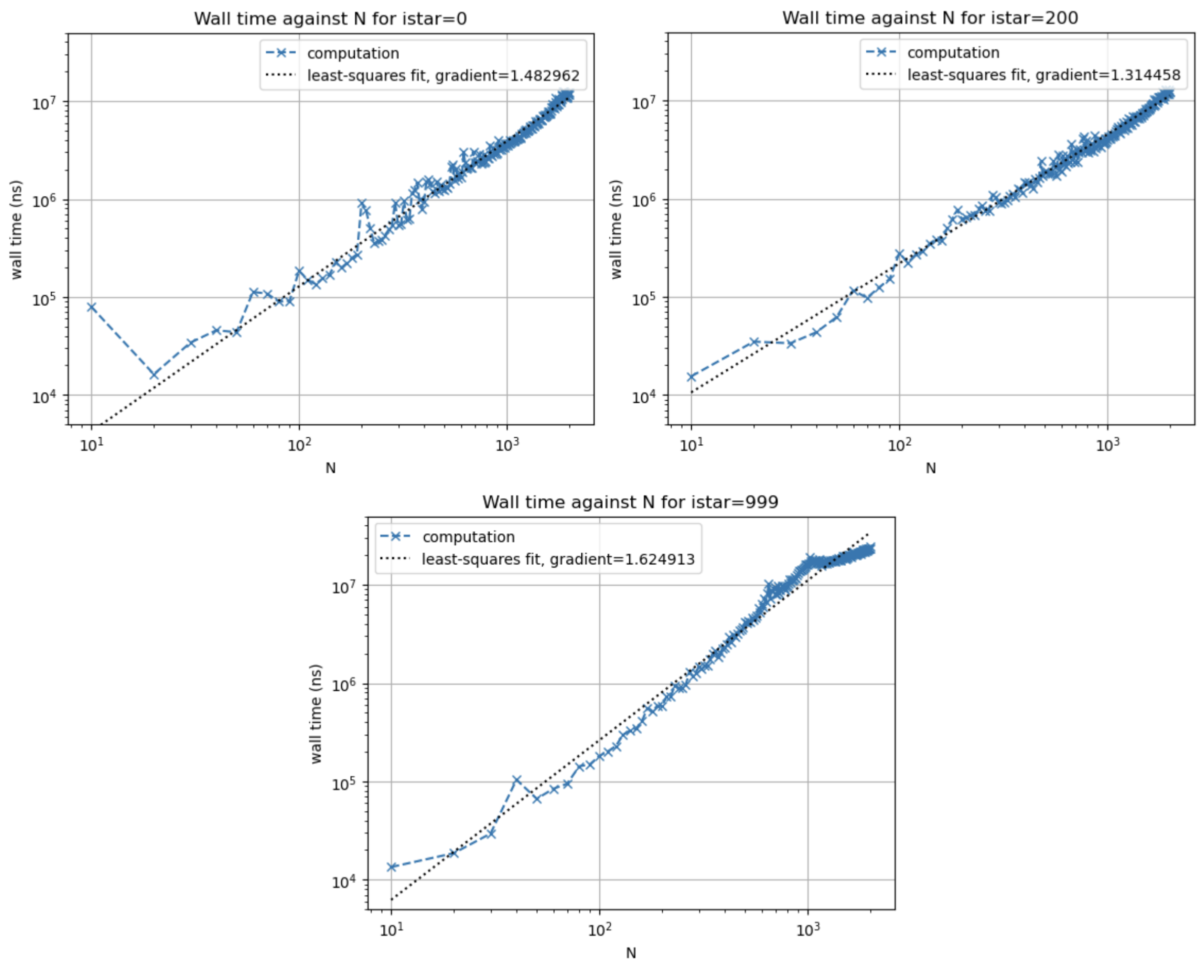

Figure 1: Wall time against N for different values of istar

Figure 1 shows that for all three values of istar, there is a strong positive correlation between wall time and the size of the list, N ($N = 1000$ in this case). The gradient is largest for $istar = N - 1$, and smallest for $istar = 200$. This tells us that the time complexity of the part1 function is better for lower values of istar, which agrees with what we would expect based on the analysis of the computational cost.

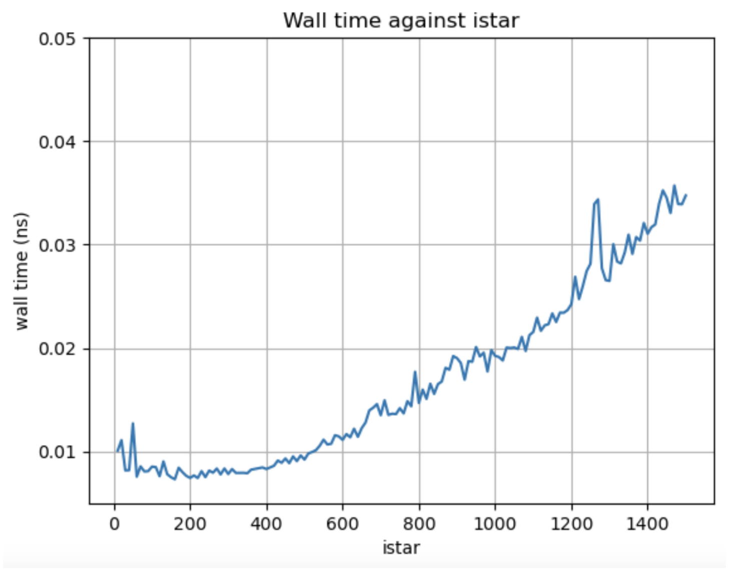

Figure 2: Wall time against istar

Figure 2 shows the dependence of wall time on the value of istar. The plot shows that in general, increasing istar will increase wall time. However, the minimum of the graph is at approximately $istar = 175$. This is interesting as it implies that it is not advantageous for istar to be as low was possible and for the function to only utilise binary search. Overall, the figures 1 and 2 largely agree with our computational cost analysis, but I was surprised to see such large variations in time complexity as istar was altered.

## Part 2

### 2.

My code uses hashing to provide an effective method for substring searching within a larger DNA sequence. I first implement two functions; char2base4 which converts a string representing a DNA sequence into a list of integers, and heval which is utilised when calculating hash values. The latter converts our list to a base-10 number modulo a prime, allowing us to perform standard operations such as addition and multiplication easily. I then define part2, which is the main function that performs the pattern search. The code first computes the hash values of the target patterns from the sequence T, and stores a mapping from the hash values to the patterns' starting indexes in the dictionary d. It calculates the rolling hash value hi for the first m characters in the sequence S and checks if any of the patterns match. Then, it iterates through S character by character, updating the rolling hash and checking if any of the patterns match. If a match is found, the starting index is stored in the list L. The function returns this list, where L[i] contains all the starting indexes in the sequence S where the length-m sequence from T[i] can be found.

To analyse the time complexity of my code, we look at the different parts of the function. Computing the hash values for the target patterns has time complexity O(l-m), where l is the length of T and m is the length of the patterns. Iterating through S character by character to update the rolling hash and check if any of the patterns match has time complexity O(n-m) since the loop iterates n-m times, where n is the length of S. The updating of hash values and checking if patterns match has time complexity O(1) due to the use of dictionaries. Since dictionaries use hash tables, searching through them has constant time complexity. Due to the fact that n-m is greater than l-m, the loop that iterates through S character by character is more influential when determining my code's time complexity. This means that overall, my code has time complexity O(n-m), which is linear.

My code should be considered to be efficient for various reasons. Using dictionaries is efficient as it allows for lookup of hash values and their corresponding patterns with constant time complexity, whilst lists perform the same task with linear time complexity. The use of the rolling hash function ensures that hash values do not have to be recomputed for every substring. Instead, the code updates the hash value of a substring based on the previous substring's hash value, which has constant time complexity. In addition, my code matches hash values before performing character comparisons. This is efficient since character comparisons are slower than hash value calculations, and so only comparing characters when they are required is optimal.

---