Scientific Computation Project 1

02027072

November 1, 2023

Part 1

1.

The strategy *part1* uses is a hybrid sorting algorithm, where initially the elements of the list are sorted using an insertion sort for an index $i \leq istar$. Then, a variation of the insertion using a binary search to replace the reverse linear search determines where to insert the subsequent elements with an index $i > istar$.

Considering the number of comparisons, the worst-case scenario for an insertion sort of length n is when the list is sorted in reverse, and by lecture slides, this has an $\mathcal{O}(n^2)$ time complexity. Hence for $i \leq istar$, the worst-case computational cost is $\mathcal{O}(istar^2)$.

For the case where $i > istar$, a binary search is performed on the elements up to i to determine its location in the sorted list of size i. The worst-case scenario for a binary search of size i has a time complexity of $\mathcal{O}(\log(i))$ as seen by lecture slides. This search is performed on i from $istar + 1$ up to $N - 1$, so the overall complexity is $\log(istar + 1) + \log(istar + 2) \ldots + \log(N - 1)$. Hence the worst-case computational cost of this part of the algorithm is $\mathcal{O}(N \log(N))$.

Combining both parts of the sorting algorithm means the worst-case computational cost for the comparisons in the algorithm overall is $\mathcal{O}(istar^2 + N \log(N))$, and choosing $istar = N - 1$ would lead to the worst-case computational cost of $\mathcal{O}(N^2)$. By this conclusion, setting $istar = 0$ would minimise the computational cost to $\mathcal{O}(N \log(N))$ for the worst-case input i.e. when the list is sorted in reverse (as this is also a worst-case scenario for binary search).

For $i \leq istar$, the best-case scenario would be a list sorted in ascending order, and the complexity would be $\mathcal{O}(istar)$. For the remaining i up to $N - 1$, the best-case scenario would require a specifically designed list where every subsequent element only required one comparison to find its correct index at the halfway point using the binary search, hence the complexity would be $\mathcal{O}(N)$. So overall, the best-case computational cost for the comparisons would be $\mathcal{O}(istar + N)$.

Now considering the swaps in the algorithm, we assume that performing a slice assignment is proportional to the size of the slice. Assuming the worst case for each iteration of the algorithm, assigning a slice of size i would have a complexity of $\mathcal{O}(i)$, so overall, the worst-case computational cost would be $\mathcal{O}(N^2)$ regardless of $istar$.

2.

In all four plots, we see a clear positive trend between time and N. For this section, we use the words 'ascending' and 'non-decreasing' interchangeably, as well as 'descending' and 'non-increasing'. We test the algorithm for three different cases, a list in ascending order, a list in descending order, and a list sampled from random integers between 0 and $2N$ inclusive.

In the first plot, we plot time against N for different choices of $istar$ when sorting a list in descending order. We observe the lower the $istar$ the faster the wall time. This is expected as the binary search part of the algorithm is faster than the reverse linear search for the worst-case scenario. For $istar = N - 1$, we can also see the timings resemble a quadratic trend, which supports our conclusion of the $\mathcal{O}(N^2)$ complexity.

In the second plot, we plot time against N for different choices of $istar$ when sorting a list in ascending order. We observe the greater the $istar$ the faster the wall time. This is expected as for a list in ascending order, the reverse linear search part of the algorithm has a complexity of $\mathcal{O}(1)$ compared to the slower complexity of the binary search $\mathcal{O}(log(N))$.

In the third plot, we observe that for $istar = 0$ (entirely using a binary search modified insertion sort), plotting time against $N \log N$ exhibits an almost linear relationship for a random list. We also observe that a sorted list in ascending order is faster than a random list and a list in descending order. This is possibly due to the swapping part of the algorithm, as for a list in ascending order, this assignment is of $\mathcal{O}(1)$ complexity.

In the fourth plot, we observe that for $istar = N - 1$ (entirely using a classic insertion sort), plotting time against N^2 exhibits an almost linear positive relationship for a list in descending order and a random list. This supports our calculation of a $\mathcal{O}(N^2)$ worst-case time complexity. We also observe that an ascending list is significantly faster than a random list, which is faster than a descending list.

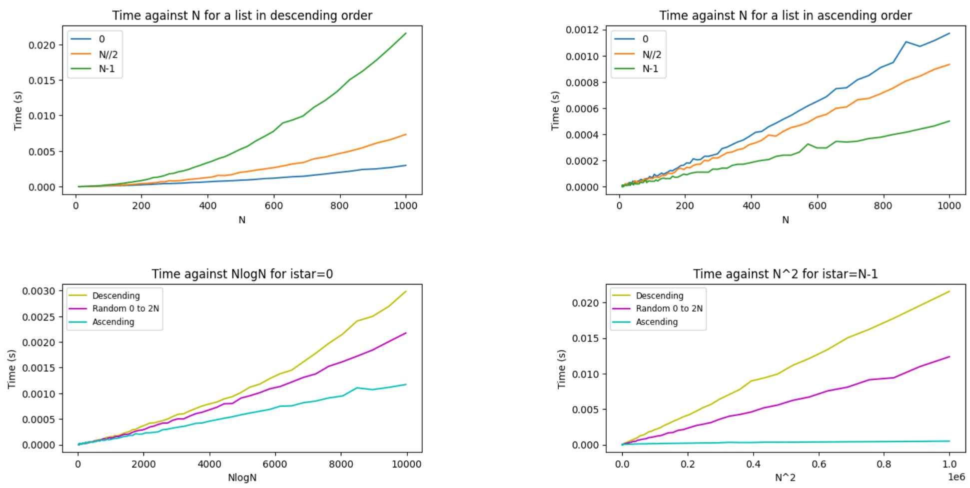

Figure 1: Plots for Part 1 Q2.

Part 2

2.

For *part2*, the strategy I used was the Rabin Karp algorithm to compare each length-m sub-string in T to each length-m sub-string in S. I implemented the *char2base4* and *heval* functions from lecture slides to calculate the two initial base hashes for the first length-m sub-string of S and of T and perform a comparison between them. I then implemented the first Rabin Karp algorithm to calculate a hash using a rolling hash function for every length-m sub-string of T. Each hash is stored as the key in a dictionary, with the value being a list of indexes indicating the locations of each hash in T. I also compare each hash to the base hash of S.

With everything initialised, I implemented the second Rabin Karp algorithm. Using a rolling hash function, a hash is calculated for every length-m sub-string of S, and then checked against the T hash dictionary. If found, every index in the dictionary value is then checked with a character comparison and the S index is added to L if there's a match. The character comparison is to protect against hash collisions.

The use of a dictionary over a list to store the hashes for T is to make the hash-matching process more efficient. The cost of a dictionary lookup is $\mathcal{O}(1)$, compared to iterating through a list of hashes for T which has a cost of $\mathcal{O}(l)$.

In the worst-case scenario, there are many hash collisions i.e. the dictionary has few unique keys with long lists of indexes. Looping through each sub-string of S has a complexity of $\mathcal{O}(n)$, looping through each T hash dictionary list of indexes would have a complexity of $\mathcal{O}(l)$, and we assume the complexity of a character comparison is $\mathcal{O}(m)$, hence my algorithm would have computational cost of $\mathcal{O}(lmn)$. This would be equivalent to the naive case of double looping through every length-m sub-string of S and T and performing a character-by-character check. However, this case is unlikely for large l, m, and n, and by choosing a large prime, fewer hash collisions will occur (at the expense of memory usage).

In the best-case scenario, there would be no hash matches, so the lists in the T hash dictionary and character comparisons would not have to occur, and my algorithm would have a computational cost of $\mathcal{O}(l + n)$ for simply calculating each hash value.