Scientific Computation Project 1

02216531

October 31, 2024

Part 1

1.

`method1` uses linear (naive) search. It iterates through the list one by one, and if it finds a match, it returns the index. Otherwise, it returns -1000. The worst case is when it has to check every element in the entire list (after which it returns -1000), so the worst-case asymptotic time complexity is $O(n)$ for one search.

`method2` uses binary search. It has a parameter `flag` which controls whether a sorting is needed. For an unsorted list (the default case), it will first sort the list. We can see that `func2A` and `func2B` together implement a merge sort and `func2C` implements a binary search. `method2` first calls `func2B` on the enumerated list `L2` (i.e., `L` paired with index), then performs a binary search with `func2C`. For a sorted list (specified by passing `flag = false`), only the binary search function `func2C` is called. From lecture, we know that the worst-case asymptotic time complexity is $O(n \log n)$ for merge sort and $O(\log n)$ for binary search, so `method2` is $O(n \log n)$ for an unsorted list and $O(\log n)$ for a sorted list.

2.

We are going to measure the time taken by `method1` and `method2` respectively for different n and m values. We can make 4 plots. The first one is when m is fixed as a small number, and we plot with different n values. The second one is the same except that m is large. The third and fourth one fixes n instead of m, with n being small and large respectively.

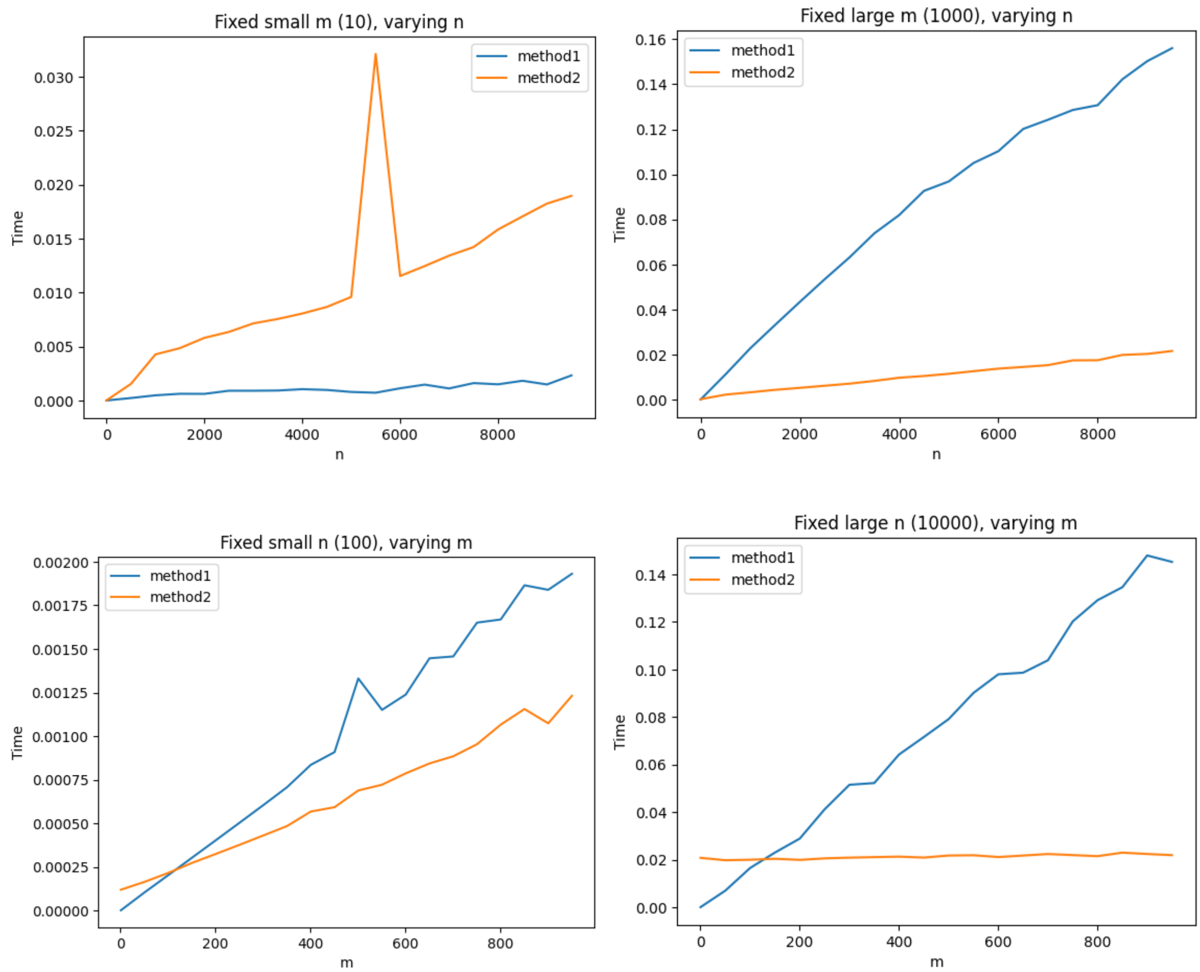
Before we run the code, we would expect that `method1` follow a linear trend throughout, since it has overall time complexity $O(nm)$ and is linear when one of n and m is fixed. For `method2`, we run with `flag = true` on the first query (thereby sorting the list) and with `flag = false` after that. The overall time complexity is therefore $O(n \log n + (m-1) \log n) = O((n+m) \log n)$. When n is fixed, it is also linear with respect to m; when m is fixed, it would be following $n \log n$ trend with a constant term determined by m.

The following plots are generated by running `part1_test`.

From the first 2 plots, we can see that when m is fixed, the time taken by `method1` is shorter than `method2` when m is small, and vice versa. This matches with our expectation: when m is small and fixed, `method1` is approximately $O(n)$, while `method2` is approximately $O(n \log n)$ which is greater. When m is large, however, the m coefficient in front of n for `method1` is more noticeable, yet because $\log n$ is negligible as long as n is not too large, `method2` is still roughly $O(n \log n)$.

Note that there is a sudden spike for the time taken by `method2` around $n = 6000$ in the first plot: this probably indicates that `method2` is not too stable in terms of the time taken, which is an inherent characteristic in merge sort and binary search.

From the last 2 plots, we can see that when n is fixed, the time taken by `method1` is shorter than `method2` when m is small (around $m = 120$), and vice versa. This is independent of whether n is small or large. The results also match with our explanation before.



Part 2

2.

My code is very simple with only 4 added lines. The main idea is to use a nested loop: the first one aims to loop over all pairs in `L`, and the second one aims to ‘move around’ the pair over `A1` and `A2` and see if it find a match. If a match is found, the index is recorded. The first loop should then range from 0 to $l-1$ so as to cover the whole L. The second loop should also start from index 0, but since a match could only be found if there is still a complete pattern, we only need to go as far as $n-m$ (not n). Since `F` is already initialised with empty lists, if no matches are found, the empty list is retained.

As for the time complexity, since there are two loops, one from 0 to $l-1$ and another from 0 to $n-m$, the overall time complexity would be the product of l and $n-m+1$. Inside the loop, there are only 2 comparison and 1 append operations which can be thought as elementary operations. Therefore, the time complexity is $O(l(n-m+1))$, and since we are given that $n \gg m$, the time complexity would be $O(ln)$.