Scientific Computation Project 1

01854740

February 7, 2023

Part 1

1.

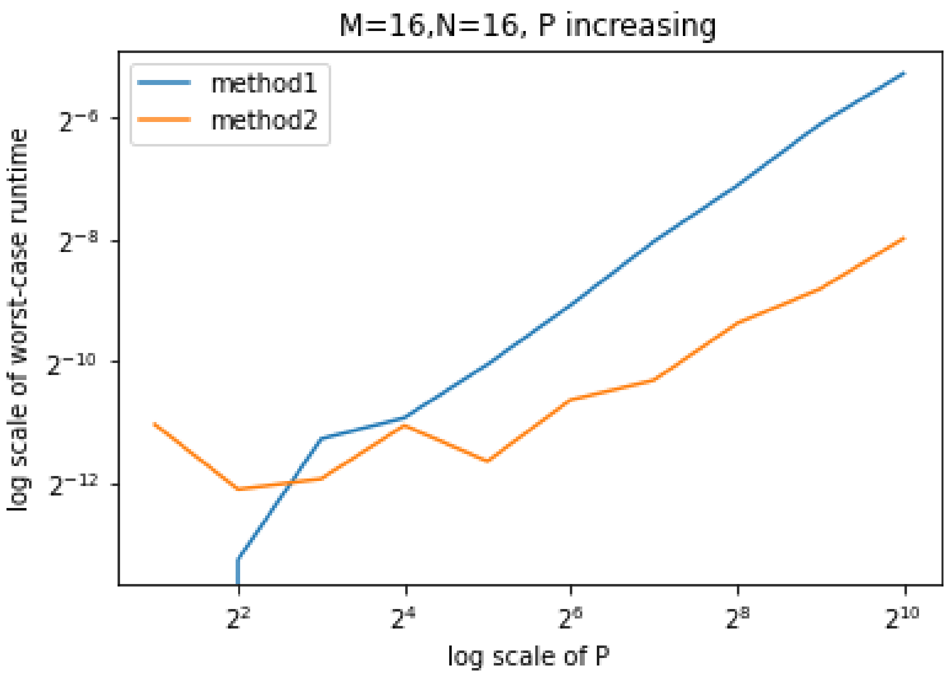
Method 1 iterates through all M sublists of length N and for each of these lists performs binary search. The binary search function returns the index of the target within the list if found, and method1 also returns the index of the list it is found in. However, if the target is not found, the binary search returns -1000 and in this case the next sublist is considered. If the target is found in no list, -1000 is returned. Binary search as we know has worst-case cost of O(logN) and the method1 has worst-case cost of O(MlogN) as if we do not find the target in any of the sublists, we have to iterate through all the M sublists and do binary search for each one. The comparisons made in method1 to -1000 are each O(1) and this is done M times but O(M) is negligeable. Initially finding the length M is also O(1) which is negligeable. Method2: L new is empty in the worst case, hence the lines following the if statement are run. A temporary list L temp is created and we iterate across the input list to method2 and append a tuple of each element and its index in L all to L temp. L temp is now like L all but in place of each value is the value as well as its indexing position within L all. We are performing an O(1) appending operation for each element of a list of length N, for each M, hence this step is O(MN). We also know the merge algorithm has linear worst-case cost, and when this is called in func1, it is called recursively on the left and right halves of L temp, which has length M when func1 is first called. If M s is the size of the sublists at each "level", we have approximately logM "levels" of splitting and at each of these levels, merge is called O(MN/M s) times, each requiring O(M s) operations. Hence, the M s cancel out and we have O(MNlogM) worst-case cost for func1. The merge function is designed to merge in non-decreasing order by the value of the first element of the tuples, and much like a mergesort algorithm, func1 is designed to sort the list of list of tuples by the first element of the tuples, splitting the input into halves recursively, merging when we get down to just 2 length-N sorted sublists (or 1 length-N sublist) and then combining the 2 sublists by merging. Essentially, func1 splits the M sublists into pairs, merges them and then merges the merged length 2N list, and so on. In the end, L new is a list of tuples sorted by value, each tuple containing the value and its index in L all. It should have length MN. We then search for our target on our sorted length MN list using recursive binary search, with a function adapted to look for the target in the first element of the tuples. Bsearch2 hence has worst-case cost of O(logMN) as we are searching through a list of length MN. Finally, the function returns (-1000,-1000) if not found, or the index in the tuple after the value in L new (which is the index in L all). It also returns L new. The overall worst-case cost of method2 is O(MN+MNlogM+logMN) which is O(MNlogM) as logMN and MN are negligeable. This is because the creation of L temp is O(MN) and the recursive calls to func1 are O(MNlogM) and the recursive calls to bsearch2 are O(logMN). Method2 is more costly but also returns the sorted list of tuples.

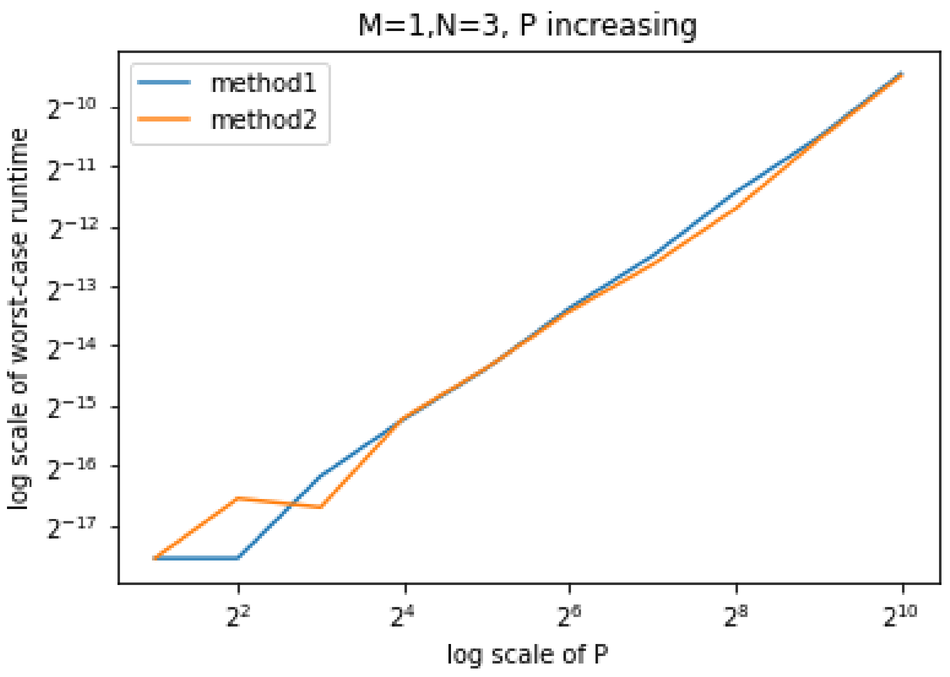
2.

In the worst case for method1, the target each time is different, or not in any sublist, hence to find all P targets, we run the function from the start P times. This is O(PMlogN) as we have to run the function P times and each time it is O(MlogN). The first search with method2 is O(MNlogM) as we have to create L temp and run func1. The second search is O(logMN) as we skip straight to bsearch2. The if and else statements have negligeable cost. The P-1 searches after the first are O(PlogMN) as we skip straight to bsearch2 P-1 times so the overall cost is O(MNlogM+PlogMN). Hence, method1 or method2 being faster depends on P,N and M. If P is small relative to N and M (e.g. around 1), method1 is faster as the O(MNlogM) term in method2 dominates (especially over MlogN). This is like running the function with just one (or few) target(s) where method1 is faster as $MNlogM > MlogN$.

If P is large relative to N and M, the O(PlogMN) term in method2 dominates and the O(MNlogM) is negligeable relative to method1 runtime. Hence, method2 is faster than method1 if $logMN < MlogN$, i.e. as long as $M < N^{(M-1)}$. This is usually the case for most values of M and N, so for large P, method2 is usually faster than method1. The runtimes are similar for large P when N=1, M=3 for example as then $logMN < MlogN$ does not hold.

In the timetest function, I have set M, N and a range over which to vary P, and for each of these P, generated a random input L all of the right dimensions. For each method, I have run the function P times and recorded the time taken, averaged over multiple runs. I have then plotted the time taken for each method against P on a log-log scale. We can see in the first graph with M=N=16 that method2 is slower than method1 for smaller P but for larger P, method2 is faster than method1. In the second graph, with N=1, M=3, neither method is visibly faster for large P because $logMN < MlogN$ does not hold for this case (and for small P method1 is faster as the MNlogM term dominates).





Part 2

2.

The function findGene uses the Rabin Karp algorithm on both sequences in L in at once and checks each of the p patterns in L p against both. Char2base first converts the characters in the sequences to integers 0,1,2,3. Heval then computes a hash value for the input. First, the algorithm computes the hash value for the first m characters of the input, and for each pattern in L p, then compares each pattern to the hash value of each string. Only if the hash value matches both, character by character comparison is performed between the pattern and the subsequences (though in reality I have used direct comparison which is faster but also linear in cost). For the remaining length m subsequences in each sequence, the hash value is calculated using a rolling hash, whereby the hash of a subsequence is computed using the previous subsequence's hash value. Again, each subsequence's hash value is compared to the pattern hash values and only if they match, character by character comparison (direct comparison) is performed. This is done for each pattern and each index in the sequences.

My function is efficient because often we do not need to do character by character comparison (which is O(m) but we only do a constant number of times per sequence and per pattern), we just need to compare the hash values, and they would usually not match. A character by character comparison for subsequences of length m (and patterns of length m) is O(m). Due to the use of a rolling hash the hash value computation is done roughly n-m times per pattern with cost O(1) each time. Also the initial hash value computation (for index 0) and comparison before the rolling hash is used for indices 1 and above is O(m) for each pattern. Hence, the total cost of the hash value computations is O(n) for each pattern and with comparisons and initial computations which are O(m), we have total cost O(m+n) per pattern. Therefore, we get an overall total average cost of O(pm+pn). The fact that we have to compare the pattern to 2 subsequences and work out hash values for 2 subsequences rather than 1 at each step simply multiplies the time taken by 2 but does not change the order of magnitude of the cost compared to standard Rabin Karp.

If the expected number of matches is μ per pattern and if μ times m is large relative to n, we get a large number of hash value comparisons being matches, hence we would have to do character by character comparison for a large number of subsequences for both sequences. Hence, in this case we lose the efficiency of having hash function calculations and the algorithm reverts to naive pattern search, so the code does not work well. In fact, in the worst case, if every pattern matches every subsequence, we must do character by character comparison for every one of lengths m along the whole length n, so get complexity O(pmn).
