Scientific Computation Project 1

01724711

February 10, 2023

Part 1

1.

Method 1 is one of the simpler approaches to this problem.

1. The method loops over each of the M lists of length N.
2. For each list it uses a binary search, 'bsearch1', function to search the current list for the target.
3. If the target is present in the list, the function returns the tuple (i, ind), where i is the index of the current iteration of lists and ind is the index of the target in that current list.
4. In the case where the target is not found in any of the lists, the function returns (-1000,-1000).

For each list $i \in 0, \dots, M-1$ we perform binary search, 'bsearch1', which we know from lectures to be $\mathcal{O}(log_2(N))$ where N is the length of each list. If the target is not present in any of the lists then we have to search all of the M lists of length N. Hence the worst-case time complexity for this method is $\mathcal{O}(Mlog_2(N))$. In the method we do not store additional data that depends on M and each iteration of the binary search we know from lectures to have constant space cost. So the space complexity for this method is $\mathcal{O}(1)$.

Method 2 is a more creative approach to solving the problem by using a merging technique to reduce the number of search operations.

1. It first creates a new list, L_temp, of tuples by looping over all M lists and all N values in each list
2. Each tuple consists of the value from one of the lists in L_all, specifically L_all$[i][j]$ and its corresponding index (i, j).
3. Then, it uses a function, func1, to use the merge function to implement a recursive form of merge sort on the lists in L_temp, based on first value of each tuple, into a single sorted list, L_new.
4. After creating L_new, the function uses binary search, 'bsearch2' which performs binary search based on the first value of the tuple element, to search for the target value, x.
5. If the target is found, the function returns the indices of the target as a tuple (i,j), where i is the index of the list and j is the index of the target in the list as well as returning the sorted by first element list of tuples consisting of element value in original list and its indices, L_new.
6. If the target is not found, the function returns (-1000,-1000) as well as the sorted by first element list of tuples consisting of element value in original list and its indices, L_new.

We begin by looping over each list $i \in 0, \dots, M-1$ in L_all and looping over each value in the current list, appending the value as well as the 'coordinate' tuple of that value, in a tuple to the newly created list L _temp. This is of time complexity $\mathcal{O}(MN)$ since both lookup in L_all and appending to the back of each list in L_new is $\mathcal{O}(1)$. This is of space complexity $\mathcal{O}(MN)$ since for each value we are storing 3 values of information in our new list L_temp, namely, the value and its 'coordinates', and there are MN total values in the initial list, L_all. Next we use func1 and merge to recursively perform merge sort on the L_temp based on the first value of each tuple, which is of course the corresponding value in the original list, L_all. From lectures we had a very similar recursive merge sort implementation. In fact this part of the code has the exact same worst-time complexity as the recursive merge sort from lectures since the only alteration is a lookup of the first value in the tuple which is $\mathcal{O}(1)$. Hence this recursive merge sort in method2 is also worst-case time complexity $\mathcal{O}(MNlog_2(MN))$ since we are sorting L_temp which has length MN. The space complexity of this part is also $\mathcal{O}(MN)$ since we are creating a new list, L_new, in which we hold the sorted L_temp which of same size as L_temp which is $\mathcal{O}(MN)$. The final part of the code is performing 'bsearch2' on L_new which again has the same worst-time complexity as the binary search implementation from lectures, since the only difference is a $\mathcal{O}(1)$ lookup of the first element in the tuple, so we know this to be $\mathcal{O}(log_2(MN))$ since we are searching L_new which is length MN. In the binary search we do not assign any more storage so this part has $\mathcal{O}(1)$ space complexity. This means that for the whole of method2 the worst-time complexity is

$$\mathcal{O}(MN) + \mathcal{O}(MNlog_2(MN)) + \mathcal{O}(log_2(MN)) = \mathcal{O}(MNlog_2(MN))$$

since as $M, N \to \infty$ the $\mathcal{O}(MNlog_2(MN))$ will be largest. The space complexity for method2 is

$$\mathcal{O}(MN) + \mathcal{O}(MN) + \mathcal{O}(1) = \mathcal{O}(MN)$$

since as $M, N \to \infty$ the $\mathcal{O}(MN)$ will be largest.

2.

Since each search is independent and we are not able to reuse any information with method1 then for every of the P targets we have to repeat the method fully. For each target $0, \ldots, P-1$ we loop over each of the M lists of length N and perform binary search on the current list for the target. So the worst-case asymptotic running time is in the case where none of the targets are present anywhere in the lists and we have to fully search all of them for every target. This would mean we have worst-time complexity of

$$\mathcal{O}(PMlog_2(N))$$

using the worst-case complexity of $\mathcal{O}(Mlog_2(N))$ for each target from part 1. The difference with method2 is that we are able to reuse information. Specifically, we are able to reuse L_new, the list of tuples where the first element in the tuple corresponds to a value in the original list we are searching and the second element is the 'coordinates' tuple of where this value was in the original list. This mean we are not required to create L_temp and then perform recursive merge sort for every of the P targets, rather only once. We are able to reuse this information since we are searching for all of the targets in the same original list. We are only required to carry out the binary search part of the method on L_new for each of the $P-1$ targets after the first target. To create L_new we have worst-case time complexity of $\mathcal{O}(MN) + \mathcal{O}(MNlog_2(MN)) = \mathcal{O}(MNlog_2(MN))$ due to the recursive merge sort algorithm and creation of L_temp part. Since for each target we perform binary search this gives worst-case time complexity of $\mathcal{O}(Plog_2(MN))$. Combining these, to search for the P targets using method2 we have worst-case time complexity

$$\mathcal{O}(MNlog_2(MN)) + \mathcal{O}(Plog_2(MN)) = \mathcal{O}((P + MN)log_2(MN))$$

We see that if the number of targets to search for is relatively small then method1 is better here since we do not need to go the relatively large cost of creating the 'coordinates' tuples list as in method2. However, if the number of targets to search for is high then not reusing any previous work and starting again for each target as we do in method1 becomes more work overall than doing the initial intensive work of the 'coordinates' tuples list and reusing this information for each target as we do in method2 in a computationally less strenuous task, 'bsearch2'. So in this case method2 would be better. The function time_test() is designed to see the relationship between walltimes and the number of targets P for three different pairs of values of M, N. The function runs creates a random list of dimensions (M, N) to search over and then times how long method1 and method2 take to complete. We do repeat this for $P \in (1, 2, 4, 8, \ldots, 1024)$ and repeat this 50 times to get an average walltime for each method for each P. We also plot a least squares fit for both methods based on our analysis of the worst-case time complexity to verify that this is indeed the case. We choose $(M, N) = (4, 64), (64, 4), (64, 64)$ to have a plot for each case of $M < N, M > N, M = N$. We restrict M, N to powers of 2 and under 1024 for convenience of running time and for convenient analysis of the time complexities due to the log_2 term and to be able to observe the general behaviour in the limit of P. We can verify that the least squares fit for both method1 and method2 are as predicted by the time complexity analysis and we observe that the curves cross at roughly where we would expect them to by solving $(P + MN)log_2(MN) = PMlog_2(N) \Rightarrow P = \frac{MNlog_2(MN)}{Mlog_2(N) - log_2(MN)}$. We are able to observe that the curves become parallel as P becomes large. This is as expected since the time complexities will tend to straight lines as the P factor dominates the others and on the logscale these straight lines will appear parallel. Overall, the graphs support the analysis that method2 becomes a more time efficient algorithm as P increases beyond the crossing point that we outlined above since we can see that after this point method2 takes less time.

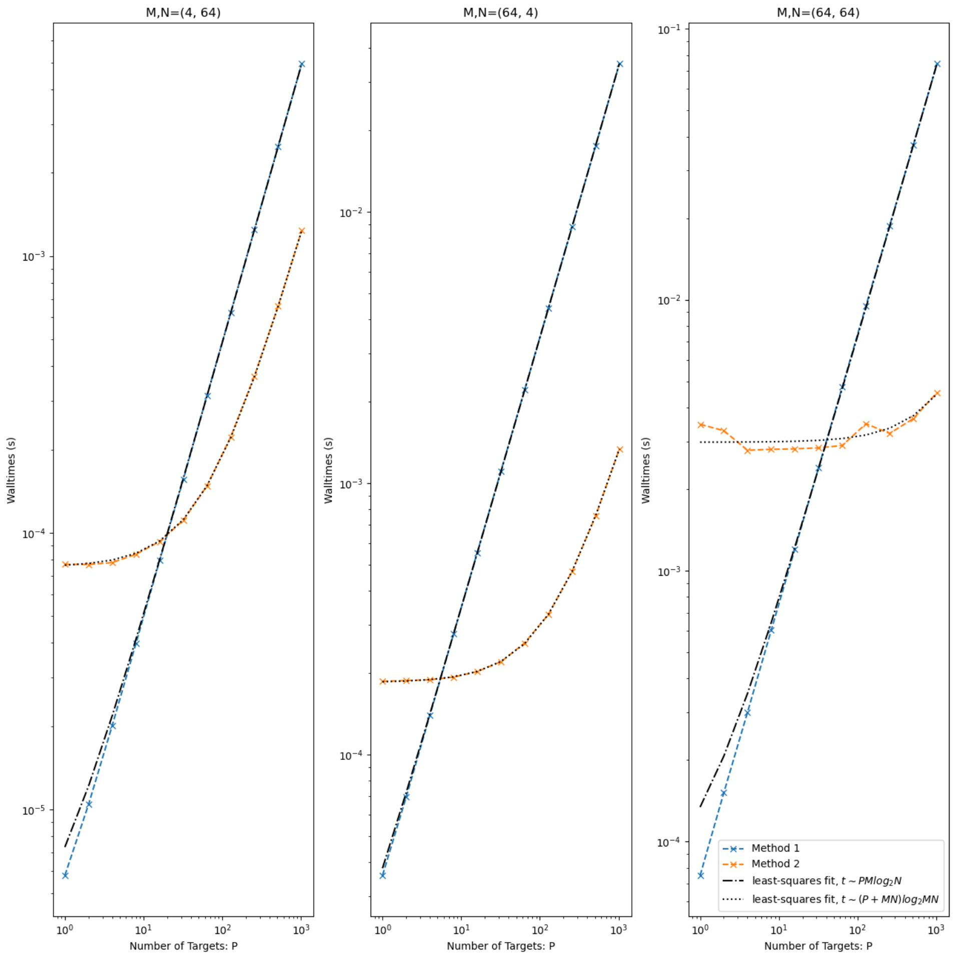

Figure 1: Walltimes for method1 and method2 for set M, N across values of P

Part 2

2.

The idea of the code is to maximise efficiency by pre-computing as much as possible. Specifically, precomputing the hashes of the all the length m substrings of X1 and X2. We also use the Rabin Karp Algorithm with a rolling hash function for efficiency. We first define a few helper functions that we will utilise in the algorithm. The match function checks if two lists are equal. The char2base4 function maps a genome sequence to its base 4 equivalent. The heval function evaluates the hash of a mapped to base 4 genome sequence efficiently using Horner's method.
We begin by mapping the first genome sequence S1 to *base* 4 as well as pre-computing bm to $base^m$ (I do not compute hashes modulo a prime since we have assumed no costs associated with calculations involving large integers). We then pre-compute the hashes, using the rolling hash algorithm, for each length m mapped to base 4 substring of S1 (which we name X1) and store this information in a dictionary hi_dic_1. We do this by looping over each substring in X1 and setting the key to it's hash and the value to a list containing the index of the substring. If the hash is already in the dictionary then we add the

index to the value of the corresponding hash. So we end up with a map of every hash that we get from the substrings of X1 to a list of locations of the corresponding hash. We do exactly the same for S2. We then loop find the intersection of these two dictionaries so that we have hashes as keys and the values as the indices where that hash occurred in both S1 and S2. Then we loop over each pattern in L_p and compute that pattern hash. We obtain the indices where this pattern occurs in X1 and X2 using the lookup in hi_dic. Since we do not set use a modular prime in this algorithm, we do not have to verify for each index that we do not have a hash collision by checking that substring in the original S1 to the pattern since there can be no hash collisions as we have unique hashes. If this is the case then we append this index to the corresponding pattern list in L_out. The complexity of this algorithm is a trade off between space and time. We choose by the assumption to prioritise time complexity and not use a prime for modular use. This means that the space complexity of the code is not $\mathcal{O}(N+M+q)$ but $\mathcal{O}(N+M)$ since we do not limit the dictionary size to q. The only scalable containers we store are X1, X2 which are size N, Y which is of size M and the dictionary hi_dic which is of size $N-M$. We can now analyse the time complexity. First the construction of X1 and X2 is $\mathcal{O}(N)$ as N is the length of S1. The initial pre-calculation of the X1, X2 substring hashes dictionaries is $\mathcal{O}(N-M)$ since we have $N-M$ substrings of length M to loop over. The intersection of the two dictionaries is also $\mathcal{O}(\mu(N-M))$, where μ is the expected number of matches for a pattern in S1, since we are utilising $\mathcal{O}(1)$ lookup in the dictionaries and each list will be roughly of length μ and so the intersection of each hash value list will be $\mathcal{O}(\mu)$. Then we loop over each pattern $0,\ldots,P-1$. We create Y and also evaluate it's hash in $\mathcal{O}(M)$. If we were to include a q then in the extremely unlikely case that we have total hash collisions and all substrings have equal hashes then we have to loop over $N-M$ indices. Then again in worst case scenario we have to fully loop over each substring in the match function if all substrings are indeed the pattern, which would take $\mathcal{O}(M)$. This would mean overall that the worst-case time complexity for the algorithm would be $\mathcal{O}(PNM)$ which is equivalent to the naive search. However, since we do not include a q we do not have to account for the worst case of hash collisions and so the worst-case time complexity becomes $\mathcal{O}(\mu N + PM)$. In the case where there is not a large number of patterns to search for and not a large number of expected matched and since we assume that $N >> M$ then we have approximate running time $\sim \mathcal{O}(N)$. In terms of memory usage we can have the largest possible hash to be roughly 4^m which is of length $mlog_{10}(4)+1$. We can make a strong assumption that the usage for memory of length l lists and an integer is comparable and so we can see that the memory requirement for storing the precomputed hashed is about the same as needed for L_p. So the memory usage for this task is $\mathcal{O}(N+\mu P)$ since L_p has P lists of approximate length μ and the memory usage of the dictionaries is $\mathcal{O}(N)$.
The code can be considered efficient since we do not re-compute any unnecessary information and beat the running time of naive search. We are able to pre-compute the hashes for all the substrings of X1 and X2 and so utilise $\mathcal{O}(1)$ lookup for the hash of the pattern. Also, in the hash calculations we use the rolling hash concept to reuse information from the previous hash using the Rabin Karp algorithm as well as the efficient Horner's method to evaluate the hash. We have also not largely increased memory usage beyond what is needed by the input. The time complexity will be affected if $\mu M >> N$ since we will have $\mu >> \frac{N}{M}$ and so we will have $\mathcal{O}(\frac{N^2}{M}+PM)$. So the time complexity will depend on the ration between N and M. If they are relatively similar then the time complexity will not be heavily affected but if $N >> M$ then we may end up with a large detriment to the running time as we could get worst-case quadratic. The memory usage of the task may be increased under this assumption since if $P\mu >> P\frac{N}{M}$ and if we assume again that $N >> M$ then we will have an increase in approximate memory usage which will be larger than $\mathcal{O}(N+P\frac{N}{M})$. So under this assumption, if we assume that $M \approx N$ then we are largely unaffected and the code will still perform well but if $N >> M$ then we will have a drop in complexity performance as a whole. So if we are in the scenario that the patterns we are searching are under some prior belief going to be be more common than on average then we can expect to see some drop in performance. If the patterns we are searching for are believed to be quite rare then we will not have any detriment to performance.