Problem 0 - fibonacci

b) The time complexity of fibonacci algorithm implemented this way is exponential, specifically $O(2^n)$, because each call branches into two recursive calls. This results in binary tree of recursive calls, with each node spwaning two children, leading to a total of 2^n function calls.

c) To improve implementation, we could use memorization or dynamic programming to avoid redundant calculations. This would reduce time complexity to $O(n)$.

recursive calls when the value(n) is 5

debug_fib(5) calls fib(5)

fib(5) calls fib(4) and fib(3)

fib(4) calls fib(3), fib(2)

fib(3) calls fib(2), fib(1)

fib(2) calls fib(1), fib(0)

fib(1) return 1

fib(0) return 0

fib(2) return 1

fib(3) returns fib(2) + fib(1)

= 1+1 = 2

fib(2) returns 1

fib(4) returns fib(3) + fib(2)

= 2 + 1 = 3

fib(3) calls fib(2) and fib(1)

fib(2) calls fib(1) and fib(0)

fib(1) returns 1

fib(0) returns 0

fib(2) returns 1

fib(3) returns fib(2) + fib(1)

= 1+1 = 2

fib(5) returns fib(4) + fib(3)

= 3 + 2 = 5

Hence the function call stack for fib(5) is

fib(5) → fib(4) → fib(3) → fib(2) → fib(1) → fib(0)

Then it returns back.

fib(1) → fib(2) → fib(3) → fib(2) → fib(1) → fib(0) → fib(3)

→ fib(4) → fib(5)

Hands-On 4

Problem 1

Proving Time Complexity:

The time complexity of the algorithm can be analyzed as follows:

Building the initial min heap: The initial heap construction takes O(K*log(K)), where K is the number of arrays.

Merging the arrays: In the worst case scenario, each element in each array only needs to be handled once. The time complexity for this step is O(NK*log(K)), where N is the size of each array, because each push and pop operation on the heap requires O(log(K)) and there are NK elements in total. As a result, the algorithm's overall time complexity is O(NKlog(K)), where K is the number of arrays and N is their respective sizes.

Possible Improvements:

Use Alternative Data Structure: Taking into account other data structures such as sorted lists or priority queues in place of a min heap may help lower the algorithm's time complexity.

Optimize Heap Operations: The overhead related to using the built-in heapq module may be decreased by implementing custom heap data structures or by optimizing the heap's operations.

Parallel Processing: By investigating methods for merging arrays in parallel, particularly for big and numerous arrays, processing times may be reduced overall.

Space Optimization: Reducing needless data duplication or taking into account in-place merging techniques could optimize space complexity.

Problem 2

Time Complexity Analysis:

Let's analyze the time complexity of the remove_duplicates function: The function iterates through the input array once, so the time complexity of the loop is O(N), where N is the size of the array. Inside the loop, each element is compared with its previous element (except for the first element). Comparisons take constant time, so the overall time complexity remains O(N). Therefore, the time complexity of the remove_duplicates function is O(N).

Possible Improvements:

Possible improvements for removing duplicate elements from a sorted array include using a set for efficient tracking of unique elements, implementing in-place removal to minimize space usage, applying binary search for time-efficient duplicate identification, optimizing comparison methods, and considering pre-sorting if the array is unsorted initially. Each approach has trade-offs in terms of time and space complexity.