Hands On 4

Problem 0: Fibonacci Sequence

Breakdown off how each recursive step works for fib(5)

fib(5): Invokes fib(4) and fib(3)
fib(4): Invokes fib(3) and fib(2)
fib(3): Invokes fib(2) and fib(1)
fib(2): Invokes fib(1) and fib(0)
fib(1): Reaches the base case and returns 1
fib(0): Reaches the base case and returns 0
Returning to fib(2): fib(2) = fib(1) + fib(0) = 1 + 0 = 1

Returning to fib(3): calls fib(1) again

fib(1): Reaches the base case again and returns 1

Returning to fib(3): fib(3) = fib(2) + fib(1) = 1 + 1 = 2

Returning to fib(4): calls fib(2) again

fib(2): Invokes fib(1) and fib(0)
fib(1): Returns 1 (base case)
fib(0): Returns 0 (base case)

Returning to fib(2): fib(2) = fib(1) + fib(0) = 1 + 0 = 1

Returning to fib(4): fib(4) = fib(3) + fib(2) = 2 + 1 = 3

Returning to fib(5): calls fib(3) again

fib(3): Invokes fib(2) and fib(1)
fib(2): Invokes fib(1) and fib(0)
fib(1): Returns 1 (base case)
fib(0): Returns 0 (base case)

Returning to fib(2): fib(2) = fib(1) + fib(0) = 1+ 0 =1

Returning to fib(3): calls fib(1)

fib(1): Returns 1 (base case)

Returning to fib(3): fib(3) = fib(2) + fib(1) = 1+ 1 = 2

Returning to fib(5): fib(5) = fib(4) + fib(3) = 3+ 2 = 5

Problem 1: Merging K sorted Arrays

Time Complexity Analysis:

The time complexity of the approach is O(N * K log K), where:

- N is the number of elements in each array
- K is the number of arrays.

The priority queue operations take log k time, and we perform these operations for every element in all araay, making the overall complexity O(N * K log K)

Possible Improvements:

- Parallel Processing: if the input arrays are extremely large, parallelizing the merging of subarrays can improve performance on multi-core systems.
- Scape Optimization: Currently, we store all elements in the result array , consuming O(N * K) scape. We might try to reduce space usage by processing elements in pl or streaming the output directly if that's feasible.

Problem 2: Removing Duplicates from a Sorted Array

Time Complexity Analysis:

The time complexity of this algorithm is O(N), where N is the number of elements in the array. The algorithm scans through the array once, comparing each element to its predecessor.

- Iteration: Looping through the array takes O(n) time.
- Slicing: Removing duplicates and slicing the array takes O(n) time.
- Total Time Complexity: T(n) = O(n)

Possible Improvements:

- Memory Optimization: The algorithm currently modified the array in place, so it is already space-efficient with O(1) extra space. However, we could explore using two-pointer technique more explicitly to make it cleaver.
- Early Termination: if we encounter large blocks of repeated elements, we could terminate early once the rest of the array contains duplicates, optimizing for certain cases.