Hands-on 4

Problem 0

Implement the Fibonacci sequence

```
x = fib(n)
   if n == 0
      return 0
   if n == 1
       return 1
   return fib(n-1) + fib(n-2)
```

Fibonacci.py

Debug the code and "step into" the function for fib(5). I want you to step into all recursive calls and list out the the function call stack ex. fib(5) -> fib(4) -> fib(3) ?.... that you observe.

Function Call Stack

```
fib(5)
   fib(4)
      fib(3)
         fib(2)
            fib(1) returns 1
            fib(0) returns 0
            return 1 + 0 = 1  (Return of fib(2) = fib(1) + fib(0))
         fib(1) returns 1
         returns 1 + 1 = 2 (Return of fib(3) = fib(2) + fib(1))
      fib(2)
         fib(1) returns 1
         fib(0) returns 0
         return 1 + 0 = 1  (Return of fib(2) = fib(1) + fib(0))
      returns 2 + 1 = 3   (Return of fib(4) = fib(3) + fib(2))
   fib(3)
      fib(2)
         fib(1) returns 1
         fib(0) returns 0
         return 1 + 0 = 1  (Return of fib(2) = fib(1) + fib(0))
      fib(1) returns 1
      returns 1 + 1 = 2   (Return of fib(3) = fib(2) + fib(1))
   returns 3 + 2 = 5   (Return of fib(5) = fib(4) + fib(3))
```

fib(5) -> fib(4) -> fib(3) -> fib(2) -> fib(1) -> fib(0) -> fib(1) -> fib(2) -> fib(1) -> fib(0) -> fib(3) -> fib(2) -> fib(1) -> fib(0) -> fib(1)

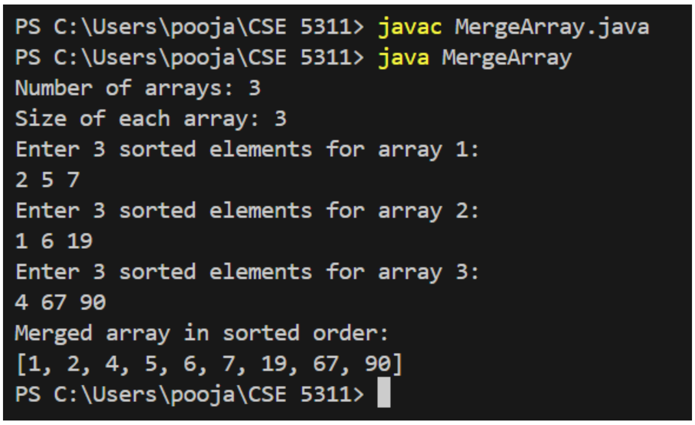
fib(5) returns **5**

Problem 1

Given K sorted arrays of size N each, the task is to merge them all maintaining their sorted order.

Week 4/DuplicateArray.java

Output:



```
PS C:\Users\pooja\CSE 5311> javac MergeArray.java
PS C:\Users\pooja\CSE 5311> java MergeArray
Number of arrays: 3
Size of each array: 3
Enter 3 sorted elements for array 1:
2 5 7
Enter 3 sorted elements for array 2:
1 6 19
Enter 3 sorted elements for array 3:
4 67 90
Merged array in sorted order:
[1, 2, 4, 5, 6, 7, 19, 67, 90]
PS C:\Users\pooja\CSE 5311>
```

Time Complexity

mergeKArrays recursively splits K arrays into halves, leading to **O(log K)** recursive levels.
mergeArrays merges all K arrays (each of size N) at each level and takes **O(NK)** time.
Total Complexity: Since merging happens at **O(log K)** levels, the final time complexity is:
O(N K log K)

Comment on way's you could improve your implementation

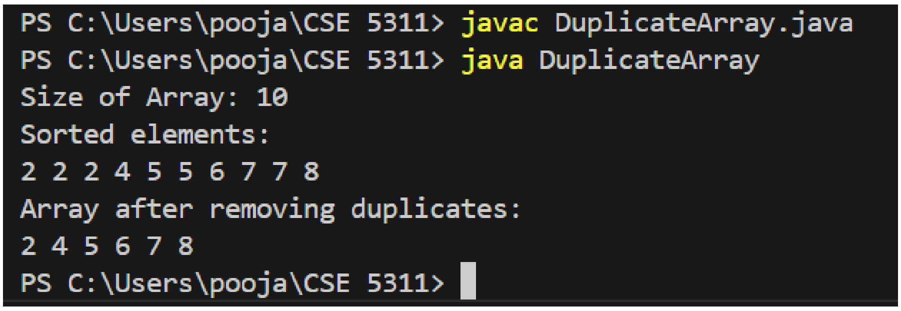
- Min-Heap (PriorityQueue) could have been used to reduce recursion overhead and achieve O(NK log K) complexity.
- Space complexity can be reduced by avoiding extra arrays (out1, out2) and merging directly into output.
- Use iterative pairwise merging can be used instead of recursion to prevent deep recursion overhead.
- Input sorting can be optimized by checking if arrays are pre-sorted instead of using Arrays.sort().

Problem 2

Given a sorted array array of size N, the task is to remove the duplicate elements from the array.

Week 4/MergeArray.java

Output:



```
PS C:\Users\pooja\CSE 5311> javac DuplicateArray.java
PS C:\Users\pooja\CSE 5311> java DuplicateArray
Size of Array: 10
Sorted elements:
2 2 2 4 5 5 6 7 7 8
Array after removing duplicates:
2 4 5 6 7 8
PS C:\Users\pooja\CSE 5311>
```

Time Complexity

The loop runs once through N elements → **O(N)**
No extra loops or nested operations, each element is compared and moved atmost once → **O(N)**.

all operations inside the loop take constant time **O(1).**
Total time complexity – **O(N)**

Ways to Improve Implementation

- Using ArrayList instead of modifying the array in-place to dynamically adjust the size and avoid unused elements.
- Using HashSet for unsorted arrays to remove duplicates in O(N) time but with O(N) space**.**
- Implementing parallel processing (multi-threading) for large inputs to enhance performance in high-volume data scenarios.