Hands - on 4

1002166689

* fibonacci Sequence

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

print( fib(5) )
```

* The above algorithm takes value 5 for input and call fib_seq function, the following call stack provide an overview for function call

-> start with fib(5)

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

for fib(4)

fib(4); calls fib(3) and fib(2)

for fib(3)

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

fib(2)

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

for fib(1)

fib(1) returns 1

for fib(0)

fib(0) return 0

Back to fib(2)

fib(2) returns `fib(1) + fib(0)' = 1+0 = 1

Back to fib(3)

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

Back to fib(4)

`fib(4) returns `fib(3) + fib(2) = 2+1 = 3

Back to fib(5)

`fib(5) returns `fib(4) + fib(3) = 3+2 = 5

* problem 1:

→ The function takes a vector of vectors 'array' as input, where each inner vector represented a sorted array.

→ The function use iterative approach to merge the sorted array. it continues the process untill all arrays are empty. in each iteration, it finds the minimum element among the current position in all arrays

→ Two variables 'min_val' and 'max_val' are used to track the minimum element and its array index.

→ a loop iterates through all arrays, checkin the first element of each non-empty array. if an array is non-empty and its first element is smaller than the current minimum ('min_val') update 'min_val' and 'min_index'.

→ adding minimum element to result: if a min element is found ('min_index is not -1), it is added to the result vector. The first element of the corresponding array array is removed. if the array becomes empty, it is removed from arrays vector.

→ The process is repeated until all arrays are merged. The loop continues as long as the 'array' vector is not empty.

Time Complexity: The time complexity is $O(N \cdot K)$, where N is the size of each array and K is the number of arrays. in each iteration the algorithm finds the minimum element among the current position in all arrays.

- The overall time complexity is influenced by the input size, specifically the number of element across all arrays.

* Optimization: The current implementation uses iterative approach for merging sorted arrays instead we can use a priority queue (min heap) as discussed in class. To keep track of the minimum element from each array efficiently. this can reduce time complexity to $O(K)$ and $O(\log K)$ in each iteration.

* problem 2:

→ The 'removeduplicate' takes a vector of integers as a refrence and modifies it in-place to remove duplicates.

→ The function uses two pointer 'i' and 'j', to iterate through the vector. The variable 'j' represents the position to store the next non-duplicate element

→ a loop iterates through the vector, and if the current element is different from next element, it is stored at the position pointed by 'j'. and the vector is resized to 'j' to remove the duplicate element

Time complexity: The time complexity of the function is O(n), where n is the size of the input vector. The function iterates through the vector once, and each iteration involves constant time operations

→ The overall time complexity is linear with respect to size of the vector.

* optimization: The current algorithm for removing duplicates from a sorted array is already quite efficient with time complexity of O(n), where n is the size of the array. Since the array is sorted, the linear approach is optimal.

→ if the array is not sorted and we want to remove duplicates, we can use a hash set to keep track of unique elements while iterating through the array. and the time complexity will be O(n)