Python for Computer Science and Data Science 2 (CSE 3652)

MAJOR ASSIGNMENT-2: RECURSION, SEARCHING, SORTING AND BIG O

Recursion

1. Write a recursive function to compute the sum of the first n natural numbers. Then, analyze its time complexity using Big O notation.
   Tasks:

   (a) Implement a recursive function `recursive_sum(n)` that computes the sum of the first n natural numbers using recursion.

   (b) Determine the base case and recursive step.

   (c) Write a function `iterative_sum(n)` that computes the sum using a loop.

   (d) Compare the time complexity of both functions using Big O notation.

   (e) Explain why recursion is less efficient in this case compared to iteration.

   Deliverables:

   (a) Python code for `recursive_sum(n)` and `iterative_sum(n)`.

   (b) A written explanation of the time complexity of both functions.

   (c) A comparison between recursion and iteration in this problem.

2. The **Josephus Problem** is a famous theoretical problem related to recursion and combinatorics. It is defined as follows:

   - There are n people standing in a circle, numbered from 0 to $n-1$.
   - Every k-th person is eliminated in a circular manner until only one person remains.
   - Your task is to implement a recursive function to find the position of the last remaining person (zero-based index).

   Tasks:

   (a) Implement a recursive function `josephus(n, k)` that returns the position of the last person remaining.

   (b) Identify the base case and the recursive formula.

   (c) Analyze the time complexity of your recursive implementation using Big O notation.

   (d) Implement an iterative version and compare its efficiency with the recursive version.

   (e) Test the function with different values of n and k.

   Deliverables:

   - Python code for `josephus(n, k)` using recursion.
   - Explanation of the recursive logic and base case.
   - Big O complexity analysis.
   - Comparison of recursion vs. iteration for solving the problem.

3. You need to implement an **efficient recursive function** to compute a^b (where a is the base and b is the exponent) using **Exponentiation by Squaring**, which reduces the number of recursive calls significantly.

   Tasks:

   (a) Implement a recursive function `fast_power(a, b)` that computes a^b using **divide and conquer**.

   (b) Identify the base case and the recursive formula:

   - If b is even, use the identity:
   $$a^b = (a^{b/2})^2$$
   - If b is odd, use the identity:
   $$a^b = a \times a^{b-1}$$

   (c) Compare the time complexity of the standard recursive method ($O(b)$) and the optimized exponentiation by squaring method ($O(\log b)$).

   (d) Implement an iterative version of exponentiation by squaring and compare it with the recursive implementation.

   (e) Test your function with large values of b (e.g., $a = 2, b = 1000000$) and observe the performance.

   Deliverables:

   - Python code for `fast_power(a, b)` using recursion.
   - Explanation of divide-and-conquer logic used in exponentiation by squaring.
   - Big O complexity analysis for both naive recursion and optimized recursion.
   - Comparison with an iterative version of the algorithm.

Phantom tracking

A notorious hacker known as **The Phantom** has been manipulating digital transactions in Bhubaneswar to move stolen money through multiple accounts.

The Cyber Crime Unit has intercepted a list of transactions (in thousands of rupees) from different accounts.

However, the transactions are unordered, some are duplicates, and The Phantom has hidden key details among normal transactions.

Your task is to analyze the data step by step, finally leading to the criminal.

Objective 1) Identifying Suspicious Transactions
The police believe that certain high-value transactions (above a given threshold) are linked to the criminal network.

Task: Use **Linear Search** to extract all suspicious transactions above a certain amount.

Given Data:
transactions = [120, 45, 300, 220, 90, 600, 130, 75, 800, 500, 350, 40]
threshold = 250

Your goal is to identify transactions greater than ₹250,000 to narrow the suspect list.

Once you extract the high-value transactions, you will use them for the next step.

Objective 2) Organizing the Transaction Data
The extracted suspicious transactions from Objective 1 need to be sorted in ascending order for further analysis.

Task:
Use **Selection Sort** to sort the suspicious transactions and return a sorted list of transactions.
Once sorted, the police will use these transactions to identify frequent recipients.

Objective 3) Finding a Specific Transaction
A forensic accountant identified a key transaction—one of these amounts was transferred to an account linked to The Phantom.

Task:
Use **Binary Search** to check if a specific suspicious transaction exists in the sorted list.

$search\_amount$ = 500

Return True if the ₹500,000 transaction is in the suspicious list, otherwise return False.

If this transaction exists, it will lead to a secret offshore account linked to The Phantom.

Objective 4) Reconstructing the Criminal's Full Transaction History
The police have recovered all transaction records from The Phantom's secret bank. However, they are completely unorganized and need to be sorted for further analysis.

Task:
Use Merge Sort to sort all transactions.

Input: transactions = [120, 45, 300, 220, 90, 600, 130, 75, 800, 500, 350, 40]

Your goal is to return a fully sorted list so the police can analyze the transaction history and track The Phantom.

Once sorted, the police notice a repeating pattern of money being sent to a specific name— The Phantom's real identity!