Summary part

1. Explain why your cached Fibonacci function is O(n) instead of O(2^n).

The cached Fibonacci function, often implemented with memoization, is O(n) because it stores the results of previously computed Fibonacci numbers in a cache (usually a dictionary or list). This eliminates the need for redundant calculations. In a naive recursive approach, each call to `fib(n)` results in two more calls (`fib(n-1)` and `fib(n-2)`), leading to an exponential time complexity of O(2^n). However, with caching, each Fibonacci number from 1 to n is computed only once, reducing the time complexity to O(n).

2. Did you notice that your cached Fibonacci function started running slowly on large inputs? Why do you think that is the case?

The cached Fibonacci function can start running slowly on large inputs primarily due to Python's recursion depth limit and the overhead of managing a large cache. For very large n, the function may hit the recursion limit set by the Python interpreter, causing a stack overflow. Additionally, even though each Fibonacci number is computed only once, accessing and storing many values in the cache can introduce overhead, particularly if the cache grows very large.

3. Do you think that the binary search algorithm might benefit from using a hash table in some way?

Binary search operates on sorted arrays and has a time complexity of O(log_n). A hash table could be used in conjunction with binary search for faster lookups, but it is generally unnecessary because binary search is already highly efficient for finding elements in sorted arrays. A hash table could be useful for operations that require frequent insertions, deletions, and lookups, but for the purpose of searching in a sorted array, binary search remains optimal.

4. What about Quicksort? Could it benefit from using a hash table in some way?

Quicksort typically does not benefit from using a hash table because its efficiency comes from partitioning the array into smaller subarrays and recursively sorting them. The partitioning step relies on comparisons rather than lookups, so a hash table does not provide an advantage. In fact, using a hash table could introduce unnecessary complexity and overhead. Quicksort's average-case time complexity is O(n_log_n), and it is highly efficient for in-place sorting. Hash tables are more suited for scenarios where quick key-value pair lookups are needed rather than sorting.

Self-reflection part

What did you learn?

I explored the practical applications and limitations of memoization and how different data structures, like hash tables, can impact algorithm performance.

What went smoothly?

Understanding the theory behind memoization and its practical implementation in recursive functions like the Fibonacci sequence was straightforward.

What was difficult about the content this week?

The challenges arose in understanding the nuances of performance issues in large-scale computations, particularly with recursion limits in Python.

How will you approach things differently next time?

Next time, I will focus more on practical experimentation with edge cases, especially in large-scale computations.

Do you have any feedback about the content for this week?

Good!