What is Memoization?

Memoization is an **optimization technique** used to improve the performance of recursive functions by storing the results of expensive function calls and reusing them when the same inputs occur again. This prevents redundant calculations, particularly in problems that involve overlapping subproblems.

How Does Memoization Work?

- When a function is called with a specific input, its result is stored in a cache (e.g., a dictionary).
- On subsequent calls with the same input, the cached result is returned instead of recomputing it.

Example: Fibonacci Numbers

The Fibonacci sequence is defined as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Without memoization, the recursive approach recalculates the same Fibonacci values multiple times, leading to exponential time complexity.

Recursive Fibonacci without Memoization

python

Copy code

```
def fibonacci(n): if n <= 1: return n return fibonacci(n-1) + fibonacci(n-2) # Testing the function
print(fibonacci(10)) # Output: 55
```

Problem:

For larger n, this approach becomes very slow because the same values are recomputed repeatedly.

Recursive Fibonacci with Memoization

Here's how memoization solves this problem:

Using a Dictionary for Memoization

python

Copy code

```
def fibonacci_memo(n, memo={}): # Check if the value is already calculated if n in memo: return memo[n] # Base cases
if n <= 1: memo[n] = n return n # Calculate and store in memo memo[n] = fibonacci_memo(n-1, memo) + fibonacci_memo(n-
2, memo) return memo[n] # Testing the function print(fibonacci_memo(10)) # Output: 55
```

Explanation:

- The `memo` dictionary stores results of previously computed Fibonacci values.
- For example:
  - $F(5) = F(4) + F(3)$
  - $F(4)$ and $F(3)$ are computed once and stored in `memo`.
  - When $F(5)$ calls $F(4)$ or $F(3)$, their cached results are used instead of recomputing.

Time Complexity: $O(n)$, **as each Fibonacci number is calculated once.**

Memoization Using `functools.lru_cache`

Python's `functools.lru_cache` provides a built-in way to memoize functions.

python

Copy code

```
from functools import lru_cache @lru_cache(maxsize=None) def fibonacci_lru(n): if n <= 1: return n return
fibonacci_lru(n-1) + fibonacci_lru(n-2) # Testing the function print(fibonacci_lru(10)) # Output: 55
```

Explanation:

- `@lru_cache` automatically stores and retrieves results for the function.
- `maxsize=None` means there is no limit on the size of the cache.

Applications of Memoization

1. **Dynamic Programming Problems**:
   - Fibonacci Sequence
   - Factorial Calculation
   - Knapsack Problem
   - Longest Common Subsequence
   - Coin Change Problem
2. **Graph Traversal**:
   - Shortest path problems (e.g., Dijkstra's algorithm).
3. **String Matching**:
   - Edit distance or Levenshtein distance.
4. **Game Theory Problems**:
   - Minimax algorithms in AI.

Benefits of Memoization

- Reduces redundant computations.
- Improves time complexity (turns exponential problems into linear or polynomial problems).
- Easy to implement with dictionaries or decorators in Python.

Visual Example

Let's compute `fibonacci_memo(5)` step-by-step:

1. $F(5) = F(4) + F(3)$
2. $F(4) = F(3) + F(2)$
3. $F(3) = F(2) + F(1)$
4. $F(2) = F(1) + F(0)$
5. $F(1) = 1, F(0) = 0$

With memoization:

- Store $F(2) = 1$, $F(3) = 2$, $F(4) = 3$, $F(5) = 5$.
- No repeated calculations for $F(3)$, $F(2)$, etc.

Key Takeaway

Memoization is a simple yet powerful technique for optimizing recursive functions, particularly in problems with overlapping subproblems. Use it to improve performance in dynamic programming and other recursive algorithms.