**Department of Computer Science and Engineering**

**2024-2025**

**Even Semester**

**DESIGN AND ANALYSIS ALGORITHMS**

**(24CS2203)**

**ALM – PROJECT BASED LEARNING**

**Exploring Memoization Techniques in Dynamic Programming Algorithms**

**V. Sahith Reddy 2420030766**

**COURSE INSTRUCTOR**

**Dr. J Sirisha Devi**

**Professor**

**Department of Computer Science and Engineering**

**Exploring Memoization Techniques in Dynamic Programming Algorithms**

**Introduction:**

In algorithm design, **performance optimization** is a critical factor, especially when solving problems that involve **overlapping subproblems** and **recursive computations**. One powerful technique to address such inefficiencies is **memoization** — a top-down dynamic programming method that stores the results of already solved subproblems and reuses them instead of recomputing.

This case study focuses on understanding how memoization enhances the performance of dynamic programming algorithms, particularly analyzing problems like the **Fibonacci sequence** and the **0/1 Knapsack Problem**. We will explore how memoization:

* Reduces redundant computations
* Minimizes execution time
* Achieves better time complexity compared to naive recursive solutions

**Problem Statement:**

Recursive algorithms solve problems by breaking them into smaller subproblems. However, many classical problems revisit the same subproblems multiple times, leading to an exponential increase in time complexity.

Example: Fibonacci Sequence

If we compute F(5) using a naive recursive algorithm:

* F(3) is computed **twice**, F(2) **three times**.
* This leads to unnecessary duplicate work.

For n = 40, the number of function calls becomes extremely large (in the order of billions), making the algorithm inefficient.

**Objective:**

The main objective is to show how **memoization** can:

* Improve **execution efficiency** by storing and reusing results.
* Reduce **time complexity** from exponential to linear or polynomial.
* Be implemented in problems like Fibonacci and 0/1 Knapsack easily without changing the basic logic of recursion.

**Real-world Examples of Problems Benefiting from Memoization:**

* **Fibonacci Numbers** – converting O(2^n) to O(n)
* **0/1 Knapsack Problem** – converting O(2^n) to O(n\*W)
* **Coin Change Problem**
* **Longest Common Subsequence**
* **Matrix Chain Multiplication**

In all these problems, overlapping subproblems occur, and memoization can be used to store and reuse intermediate results.

**ALGORITHM / PSEUDO CODE**

**1.Understanding the Memoization Approach**

Memoization involves:

* Using a **data structure (array, map, or table)** to store computed values.
* Checking the memo structure before solving a subproblem.
* If the value is already present, use it directly.
* Otherwise, compute and store it for future use.\

**2.** **Example: Fibonacci Sequence (Step-by-Step)**

**Step 1: Naive Recursive Algorithm**

fib(n):

if n <= 1:

return n

return fib(n-1) + fib(n-2)

* Solves subproblems repeatedly.
* Exponential time complexity.

**Step 2: Adding Memoization**

fib(n, memo):

if n <= 1:

return n

if memo[n] is not empty:

return memo[n]

memo[n] = fib(n-1, memo) + fib(n-2, memo)

return memo[n]

**Step 3: Dry Run Example for n = 5**

1. fib(5) → memo[5] is empty → compute fib(4) + fib(3)
2. fib(4) → memo[4] is empty → compute fib(3) + fib(2)
3. fib(3) → memo[3] is empty → compute fib(2) + fib(1)
4. fib(2) → memo[2] is empty → compute fib(1) + fib(0) = 1 + 0 = 1
   * memo[2] = 1
5. fib(1) → base case = 1
6. memo[3] = fib(2)+fib(1)=1+1=2
7. memo[4] = fib(3)+fib(2)=2+1=3
8. fib(3) is already stored → no recomputation
9. memo[5] = fib(4)+fib(3)=3+2=5

Result: 5 function results stored → No repeated calls.  
Naive recursion would have made **15+ calls** for the same input.

**TIME COMPLEXITY**

**1. Fibonacci Without Memoization**

* Recursive calls:
* T(n) = T(n-1) + T(n-2) + O(1)
* This recurrence solves to:
* T(n) = O(2^n)
* For n = 40, the number of calls ≈ 2^40 ≈ 1 trillion calls
* Impractical for large inputs.

**2. Fibonacci With Memoization**

* Each subproblem (from 0 to n) is solved only once.
* For every n, we make:
  + One computation
  + Two lookups in memo table (O(1) each)
* Total operations:
* T(n) = O(n)
* Example: n = 40 → 41 computations only.

**3. Knapsack Without Memoization**

* Each item has two choices (include or exclude), so:
* T(n) = O(2^n)
* Example: n = 10 → 1024 calls, and grows rapidly as n increases.

**4. Knapsack With Memoization**

* There are n items and capacity W.
* Each subproblem (n, W) is solved only once.
* Total subproblems = n \* W
* T(n, W) = O(n \* W)
* Much more efficient than O(2^n).

**SPACE COMPLEXITY**

**Space Complexity Table**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Problem** | |  | | --- | | **Memoization Space** |  |  | | --- | |  | | |  | | --- | | **Recursion Stack** |  |  | | --- | |  | | | **Total Space** | | --- |  |  | | --- | |  | |
| |  | | --- | | Fibonacci |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | |
| Knapsack | |  | | --- | | O(n\*W) |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | | O(n\*W) |

Although memoization uses extra memory, this **trade-off between space and time** leads to massive performance improvements.

**Comparative Performance Table**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Problem Type** |  |  | | --- | |  | | |  | | --- | | **Input Size** |  |  | | --- | |  | | |  | | --- | | **Naive Recursion Time** |  |  | | --- | |  | | |  | | --- | | **Memoization Time** |  |  | | --- | |  | | | **Space Used** | | --- |  |  | | --- | |  | |
| |  | | --- | | Fibonacci (n=40) |  |  | | --- | |  | | 40 | |  | | --- | | Very High (2^40 ops) |  |  | | --- | |  | | |  | | --- | | Very Low (40 ops) |  |  | | --- | |  | | |  | | --- | | O(n) |  |  | | --- | |  | |
| |  | | --- | | Knapsack (n=10, W=50) |  |  | | --- | |  | | 10, 50 | |  | | --- | | O(2^10) = 1024 calls |  |  | | --- | |  | | |  | | --- | | O(500) calls |  |  | | --- | |  | | O(n\*W) |

**Conclusion**

Memoization is a **powerful technique** that:

* Converts **inefficient recursive solutions** into **efficient DP solutions**.
* Reduces **time complexity** drastically by storing and reusing results.
* Involves additional space usage but results in much faster execution.
* Is widely used in optimization problems in computer science and competitive programming.

**Key Takeaways:**

* Fibonacci problem goes from **O(2^n)** → **O(n)**.
* Knapsack problem goes from **O(2^n)** → **O(n\*W)**.
* Memoization is best used in problems with **overlapping subproblems** and **optimal substructure**.