Recursion, Backtracking and Dynamic Programming

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

What is Dynamic Programming?

- **Dynamic Programming (DP):** Solve problems by breaking into overlapping subproblems.
- **Key Idea:** Avoid recomputation by **storing results** of subproblems.
- Two approaches:
  - **Top-down (Memoization: recursion + caching)**
  - **Bottom-up (Iterative, table-filling)**

DP Workflow (Memoization)

1. Define recursive relation
2. Identify base cases
3. Store answers in a cache (array)
4. Before computing, check cache
5. Save result into cache

Example 1: Fibonacci Numbers

Recursive definition:

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

Fibonacci Code (with Cache)

```c
#define MAX 50
int memo[MAX];

int fib(int n) {
    if (memo[n] != -1) return memo[n];
    if (n == 0) return memo[n] = 0;
    if (n == 1) return memo[n] = 1;
    return memo[n] = fib(n-1) + fib(n-2);
}
```

⚡ From exponential time → linear time.

Example 2: Grid Paths

Count ways to reach bottom-right of an `m x n` grid, moving only right or down.

Recurrence:

$$ways(m, n) = ways(m-1, n) + ways(m, n-1)$$

Base cases:

- `ways(0,n) = 1`
- `ways(m,0) = 1`

Grid Paths Code

```c
#define MAX 20
int memo[MAX][MAX];

int paths(int m, int n) {
    if (memo[m][n] != -1) return memo[m][n];
    if (m == 0 || n == 0) return memo[m][n] = 1;
    return memo[m][n] = paths(m-1, n) + paths(m, n-1);
}
```

Example 3: Longest Common Subsequence (LCS)

Problem: Given strings `X` and `Y`, find length of longest subsequence common to both.

Recurrence:

$$LCS(i,j) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & X[i-1] = Y[j-1] \\ \max(LCS(i-1, j), LCS(i, j-1)) & \text{otherwise} \end{cases}$$

LCS Code (Memoized)

```c
#define MAX 100
int memo[MAX][MAX];

int LCS(char *X, char *Y, int i, int j) {
    if (i == 0 || j == 0) return 0;
    if (memo[i][j] != -1) return memo[i][j];

    if (X[i-1] == Y[j-1])
        return memo[i][j] = 1 + LCS(X, Y, i-1, j-1);

    return memo[i][j] =
        (LCS(X, Y, i-1, j) > LCS(X, Y, i, j-1) ?
         LCS(X, Y, i-1, j) : LCS(X, Y, i, j-1));
}
```

Example 4: 0/1 Knapsack Problem

Problem: Given `n` items with weights `w[i]` and values `v[i]`, and capacity `W`, maximize total value.

Recurrence:

$$K(n, W) = \begin{cases} 0 & n = 0 \text{ or } W = 0 \\ K(n-1, W) & w[n-1] > W \\ \max(v[n-1] + K(n-1, W - w[n-1]),\ K(n-1, W)) & \text{otherwise} \end{cases}$$

Knapsack Code (Memoized)

```c
#define MAXN 100
#define MAXW 1000
int memo[MAXN][MAXW];

int knap(int n, int W, int w[], int v[]) {
    if (n == 0 || W == 0) return 0;
    if (memo[n][W] != -1) return memo[n][W];

    if (w[n-1] > W)
        return memo[n][W] = knap(n-1, W, w, v);

    int include = v[n-1] + knap(n-1, W - w[n-1], w, v);
    int exclude = knap(n-1, W, w, v);

    return memo[n][W] = (include > exclude ? include : exclude);
}
```

Why Use Memoization?

- Simple to implement (just add a cache)
- Improves performance drastically
- Natural extension of recursion
- Good stepping stone to iterative DP

Practice Problems

1. Fibonacci (memoized)
2. Grid Paths (m x n grid)
3. Longest Common Subsequence
4. Minimum Coin Change
5. Knapsack Problem
6. Edit Distance

Summary

- DP = Recursion + Caching
- Identify overlapping subproblems
- Store answers in arrays
- Examples: Fibonacci, Grid Paths, LCS, Knapsack
- Powerful tool for optimization problems