Dynamic Programming

Dynamic Programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure.

Characteristics of Dynamic Programming:

1. **Applications**:
    - Used for optimization, counting, and feasibility problems.
    - Examples include finding the longest, maximum, or minimum subsequences or paths in data structures.
2. **Design Approach**:
    - The design often starts with a recursive algorithm that solves the problem by combining the solutions to its subproblems.
3. **Naive Recursive Implementation**:
    - A straightforward recursive implementation can lead to poor running times due to redundant calculations.

Techniques to Improve Running Time:

To enhance the efficiency of a naive recursive algorithm, dynamic programming suggests two main approaches:

1. **Bottom-Up Approach**:
    - This approach starts solving the problem from the simplest possible subproblem and works its way up to the given problem.
2. **Top-Down Approach with Memoization**:
    - This method involves writing the recursive algorithm and storing the results of subproblems to avoid redundant work.

Example: Longest Common Subsequence (LCS)

The LCS problem is a classic example illustrating dynamic programming. It seeks to find the longest subsequence present in two sequences.

Problem Statement:

- Given two sequences `A` and `B`, find the length of the longest subsequence that is common to both.

Example Sequences:

- Sequence `A`: [2, 3, 1, 4, 5]
- Sequence `B`: [1, 2, 3, 4, 5, 7]

Subsequences:

- Subsequences of `A`: [2, 4], [3, 1, 5], etc.
- Subsequences of `B`: [1, 3], [1, 4], [1, 3, 4, 5], etc.

Pseudocode for LCS:

```
// Function to find the LCS length
LCS(A[1..i], B[1..j]) {
    if (i == 0 || j == 0) {
        return 0; // Base case: one of the sequences is empty
    }
    if (A[i] == B[j]) {
        return 1 + LCS(A[1..i-1], B[1..j-1]); // Elements match, part of LCS
    } else {
        return max(LCS(A[1..i-1], B[1..j]), LCS(A[1..i], B[1..j-1])); // Elements don't match,
    }
}
```

Visualization of Subproblem Dependencies:

```
// Recursive tree showing dependencies between subproblems in LCS

A[i] and B[j] subproblems:

   T(n, m)
  /      \
T(n-1, m) T(n, m-1)
/   \        /    \
...  ...    ...   ...

// T(i, j) depends on T(i-1, j) and T(i, j-1)
// Running time is exponential in terms of n and m - which is bad
```

Running Time Considerations:

- The naive recursive approach can potentially explore all `2^n` subsequences, which is highly inefficient.
- Dynamic programming improves this by storing the results of subproblems, reducing the running time to polynomial time.

Dynamic Programming can be implemented in two main ways: Bottom-Up and Top-Down with Memoization.

Bottom-Up Approach

The bottom-up approach iteratively builds the solution for the smallest subproblems and then uses those solutions to construct solutions for larger subproblems.

DP Table (Initially filled with base cases)

```
+---+---+---+---+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | . | . | . | j | . | m |
+---+---+---+---+---+---+---+---+---+---+---+
| 0 | x | x | x | x | x | x | x | x | x | x |
| 1 | x |   |   |   |   |   |   |   |   |   |
| 2 | x |   |   |   |   |   |   |   |   |   |
| 3 | x |   |   |   |   |   |   |   |   |   |
| . | x |   |   |   |   |   |   |   |   |   |
| . | x |   |   |   |   |   |   |   |   |   |
| . | x |   |   |   |   |   |   |   |   |   |
| i | x |   |   |   |   |   |   |   |   |   |
| . | x |   |   |   |   |   |   |   |   |   |
| n | x |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
```

'x' represents base case values or previously computed values.
Blank spaces are filled iteratively from top-left to bottom-right.

In the bottom-up approach, the DP table is filled in an orderly fashion, iteratively solving each subproblem.

Pseudocode for Bottom-Up LCS

```
// Initialize a table dp where dp[i][j] will be storing the length of LCS of A[1..i] and B[1..j
// Note: 'n' and 'm' represent the lengths of the sequences A and B respectively.

for i from 0 to n: dp[i][0] = 0;
for j from 0 to m: dp[0][j] = 0;

for i from 1 to n:
    for j from 1 to m:
        if A[i] == B[j]:
            dp[i][j] = dp[i-1][j-1] + 1;
        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);

return dp[n][m]; // The length of LCS is in dp[n][m].
```

Complexity Analysis

- Time complexity: `O(n * m)` where `n` is the length of sequence A and `m` is the length of sequence B.
- Space complexity: `O(n * m)` for maintaining the `dp` table.

Top-Down Approach with Memoization

The top-down approach with memoization is a recursive approach that saves the result of subproblems to avoid recomputation.

DP Table (Filled on-demand by recursive calls)

```
+---+---+---+---+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | . | . | . | j | . | m |
+---+---+---+---+---+---+---+---+---+---+---+
| 0 | x | x | x | x | x | x | x | x | x | x |
| 1 | x | / | / | / | / | / | / | / | / | / |
| 2 | x | / |   |   |   |   |   |   |   |   |
| 3 | x | / |   |   |   |   |   |   |   |   |
| . | x | / |   |   |   |   |   |   |   |   |
| . | x | / |   |   |   |   |   |   |   |   |
| . | x | / |   |   |   |   |   |   |   |   |
| i | x | / |   | / |   |   |   | / |   |   |
| . | x | / |   | / |   |   |   |   |   |   |
| n | x | / |   | / |   |   |   |   |   | / |
+---+---+---+---+---+---+---+---+---+---+---+
```

'x' represents base case values.
'/' represents values computed on-demand by recursive calls.
Blank spaces represent values not yet computed.

In the top-down approach, cells in the DP table are filled sporadically, based on the needs of the recursive calls, and the rest of the cells remain unfilled until they are needed.

This means the bottom-up fills the entire table, whereas top-down fills the table partially, just enough to compute the answer to the original problem.

Pseudocode for Top-Down LCS with Memoization

```
// This function initializes the memoization table dp where all values are set to -1 initially.
LCS-Main(A[1..n], B[1..m]) {
    for i from 1 to n:
        for j from 1 to m:
            dp[i][j] = -1;

    return LCS(A[1..n], B[1..m], dp);
}

// Recursive function for LCS with memoization.
LCS(A[1..i], B[1..j], dp) {
    if i == 0 or j == 0:
        return 0;
    if dp[i][j] != -1:
        return dp[i][j];
    if A[i] == B[j]:
        dp[i][j] = 1 + LCS(A[1..i-1], B[1..j-1], dp);
    else:
        dp[i][j] = max(LCS(A[1..i-1], B[1..j], dp), LCS(A[1..i], B[1..j-1], dp));

    return dp[i][j];
}
```

Complexity Analysis

- Time complexity: `O(n * m)` since each pair `(i, j)` is solved only once due to memoization.
- Space complexity: `O(n * m)` for maintaining the `dp` table.

Space Optimization in Dynamic Programming

For certain problems, we can optimize the space used in dynamic programming.

Space-Optimized LCS Length

```
// Optimized space complexity for computing the length of LCS.
for i from 1 to n:
    for j from 1 to m:
        if A[i] == B[j]:
            dp[i % 2][j] = dp[(i-1) % 2][j-1] + 1;
        else:
            dp[i % 2][j] = max(dp[(i-1) % 2][j], dp[i % 2][j-1]);

return dp[n % 2][m]; // The length of LCS is in dp[n % 2][m].
```

Complexity Analysis

- Time complexity remains `O(n * m)`.
- Space complexity is optimized to `O(min(n, m))` by only keeping the last row of the `dp` table.

Dynamic Programming Recap

Dynamic Programming is a versatile technique used in algorithm design for solving problems that exhibit the properties of overlapping subproblems and optimal substructure. It is commonly applied in various domains, including optimization, counting, and checking for feasibility.

Recap Points:

1. **Optimization/Counting/Feasible**: Dynamic Programming is well-suited for optimization problems where we seek to find the best solution among many possible ones. It's also useful for counting problems and determining the feasibility of a scenario.
2. **Recursive Design**: The recursive approach forms the foundation of many dynamic programming solutions, often starting with a top-down approach to break down the problem.
3. **Naive Implementation and Issues**: A naive recursive implementation without optimizations can lead to excessive running time due to repeated calculations of the same subproblems.
4. **Improvement Techniques**:
    - *Bottom-Up Approach*: It builds the solution from the simplest cases upward to the problem at hand, improving running time.

- *Top-Down Approach with Memoization*: It involves storing the results of subproblems, which can lead to space optimization.

Sub-array Sum Problem

The sub-array sum problem involves finding a contiguous sub-array within a given array which has the largest sum.

Problem Definition:

- **Optimization**: Find the sub-array with the maximum sum within an array.

Solution Approaches:

- **Divide & Conquer**: Previously covered in divide and conquer section with a time complexity of `O(nlogn)`.

Dynamic Programming Solution:

Naive Recursive Approach:

```
f(i) := maximum sum of subarray ending at i
     = max{A[i], f(i-1) + A[i]}

f(0) = 0 // Base case
```

Example:

- Given array `A` : [1, -2, 3, -5, 6]
- The function `f` calculates the following:
  - `f(1) = 1`
  - `f(2) = -1`
  - `f(3) = 3`
  - `f(4) = -2`
  - `f(5) = 6` (Maximum Sum)

Pseudocode:

```
subarr_sum(A[1...i]) {
    if (i == 0) return 0;
    return max(A[i], subarr_sum(A[1...i-1]) + A[i]);
}

main(A[1:n]) {
    s = -∞;
    for(i = 1...n) {
        s = max(s, subarr_sum(A[1:i]));
    }
    return s;
}
```

Running Time Analysis for Naive Recursive Approach:

- Time complexity: O(2^n) since each function call potentially generates two further calls.
- This approach is highly inefficient due to its exponential running time and the repeated recalculations of the same subproblems.

Top-Down with Memoization:

```
subarr_sum(A[1...i], dp[]) {
    if (i == 0) return 0;
    dp[i] = max(A[i], dp[i-1] + A[i]);
    return dp[i];
}

main(A[1:n]) {
    s = -∞;
    for(i = 1...n) {
        s = max(s, subarr_sum(A[1:i], dp));
    }
    return s;
}
```

Running Time Analysis for Top-Down with Memoization:

- Time complexity: O(n) since each subproblem is only solved once and then stored.
- Space complexity: O(n) for the memoization array.
  This method significantly improves efficiency by avoiding redundant calculations.

Bottom-Up Approach:

```
subarr_sum(A[1:n]) {
    dp[0] = 0;
    for (i = 1...n) {
        dp[i] = max(A[i], dp[i-1] + A[i]);
    }
    return max(dp);
}
```

Running Time Analysis for Bottom-Up Approach:

- Time complexity: O(n) as it iterates through the array once and computes the maximum subarray sum for each element.
- Space complexity: O(n) for storing the dp array which holds the maximum subarray sum ending at each index.
- This approach is efficient and straightforward, building up the solution iteratively.

Final Optimizations:

- **Running Time**: The running time is optimized to `O(n)` with memoization.
- **Space Optimization**: The space is further optimized to `O(1)` in the final version while maintaining the `O(n)` time complexity.

Final Pseudocode:

```
subarr_sum(A[1:n]) {
    overall_max = -∞;
    max_at_i = 0;
    for (i = 1...n) {
        max_at_i = max(A[i], max_at_i + A[i]);
        overall_max = max(overall_max, max_at_i);
    }
    return overall_max;
}
```

Complexity Analysis:

- Time complexity: `O(n)` for the optimized version.
- Space complexity: `O(1)` for the space-optimized version.

0-1 Knapsack Problem

The 0-1 Knapsack problem is a classic problem in combinatorial optimization. The goal is to select a subset of items, each with a value and a weight, to maximize the total value without exceeding the knapsack's capacity.

Problem Definition:

- **Items**: Given `n` items, each with a value `V[i]` and a weight `W[i]`.
- **Capacity**: The knapsack has a maximum capacity `C`.
- **Objective**: Maximize the total value of items that fit within the knapsack capacity `C`.

Greedy Approaches Attempt:

1. **Sort by Value**:
    - Sort items by their value, and then try to fit the most valuable items first.
    - This approach does not guarantee an optimal solution.
2. **Sort by Value-to-Weight Ratio**:
    - Sort items by their value-to-weight ratio `V[i]/W[i]`.
    - This approach, known as the greedy method, does not always yield the optimal solution for the 0-1 Knapsack problem.

Dynamic Programming Solution:

The optimal solution can be found using dynamic programming by defining the function `f(i, j)` which represents the maximum value achievable with the first `i` items and a knapsack capacity of `j`.

Pseudocode:

```
f(i, j) = maximum value using the first i items and capacity j
        = max( f(i-1, j), V[i] + f(i-1, j-W[i]) ) if W[i] <= j
        = f(i-1, j) otherwise

// Base cases:
f(0, j) = f(i, 0) = 0
```

Running Time Analysis:

- Without memoization, the naive recursive approach would have exponential running time, approximately `O(2^n)`, as it explores all combinations of items.

Visualization of Subproblem Dependencies:

Below are text representations for the visualization of subproblem dependencies for the 0-1 Knapsack problem, including both the recursive tree and the DP matrix:

Recursive Tree:

The recursive tree for the 0-1 Knapsack problem can be visualized as follows, where each node represents a state `(i, j)` corresponding to the decision at the `i-th` item with `j` remaining capacity:

```
                 T(n, C)
                /       \
         T(n-1, C)       T(n-1, C-W[n])
          /    \            /         \
    T(n-2, C) T(n-2, C-W[n-1])  ...   ...
     /   \         /   \
    ...  ...      ...  ...
```

This tree has `n` levels corresponding to the `n` items, and each level has twice as many nodes as the previous level, leading to `2^n` possible subproblems.

DP Matrix:

The DP matrix for the bottom-up approach to the 0-1 Knapsack problem can be visualized in a two-dimensional grid. Each cell `(i, j)` contains the maximum value that can be achieved with the first `i` items and a knapsack capacity of `j`.

```
     0   1   2   3   ...   j   ...   C
   +---+---+---+---+---+---+---+---+---+
0  | 0 | 0 | 0 | 0 | ... | 0 | ... | 0 |
   +---+---+---+---+---+---+---+---+---+
1  | 0 |   |   |   | ... |   | ... |   |
   +---+---+---+---+---+---+---+---+---+
2  | 0 |   |   |   | ... |   | ... |   |
   +---+---+---+---+---+---+---+---+---+
3  | 0 |   |   |   | ... |   | ... |   |
   +---+---+---+---+---+---+---+---+---+
...|...|...|...|...| ... |...| ... |...|
   +---+---+---+---+---+---+---+---+---+
i  | 0 |   |   |   | ... | dp[i,j] |   |
   +---+---+---+---+---+---+---+---+---+
...|...|...|...|...| ... |...| ... |...|
   +---+---+---+---+---+---+---+---+---+
n  | 0 |   |   |   | ... |   | ... |   |
   +---+---+---+---+---+---+---+---+---+
```

In this matrix:

- The first row (0th row) and first column (0th column) are initialized to 0, representing the base case where no items are chosen or the knapsack capacity is 0.
- Each cell `(i, j)` is filled in based on the value of the previous items' decisions, and the matrix is filled row by row from top to bottom.
- The value `dp[i, j]` is calculated based on the maximum of not including the current item (`dp[i-1, j]`) or including it (`V[i] + dp[i-1, j-W[i]]` if `W[i] <= j`).

The final answer to the 0-1 Knapsack problem would be found in the cell `dp[n, C]` after the matrix has been completely filled in.

Bottom-Up Approach:

The bottom-up approach fills a table iteratively, considering all weights from 0 to `C` and each item one by one.

Pseudocode:

```
for i from 1 to n:
    for j from 1 to C:
        if (W[i] <= j):
            dp[i, j] = max(dp[i-1, j], V[i] + dp[i-1, j-W[i]]);
        else:
            dp[i, j] = dp[i-1, j];

return dp[n, C];
```

Running Time Analysis:

- The bottom-up dynamic programming approach has a running time of `O(n*C)` , which is polynomial and much more efficient than the naive recursive method.

Space Optimization Trick:

- By using a single-dimensional array and iterating the items in reverse, the space complexity can be reduced to `O(C)` .

```
for i from 1 to n:
    for j from C down to W[i]:
        dp[j] = max(dp[j], V[i] + dp[j-W[i]]);

return dp[C];
```

Complexity Analysis:

- Time complexity: `O(n*C)` for the bottom-up solution.
- Space complexity: `O(C)` using the space optimization trick.

Top-Down Approach to 0-1 Knapsack

The top-down approach employs recursion with memoization to optimize the running time. This technique stores the results of subproblems to avoid redundant calculations.

Pseudocode:

```
function knapsack(V[1...n], W[1...n], i, j, dp[][]) {
    if (i == 0 || j == 0) return 0;
    if (dp[i][j] != -1) return dp[i][j];

    if (W[i] <= j) {
        dp[i][j] = max(knapsack(V, W, i-1, j, dp), V[i] + knapsack(V, W, i-1, j-W[i], dp));
    } else {
        dp[i][j] = knapsack(V, W, i-1, j, dp);
    }

    return dp[i][j];
}

function main(W[], V[], n, C) {
    // Initialize memoization table with -1
    dp[1...n][1...C] = -1;
    return knapsack(W, V, n, C, dp);
}
```

Running Time Analysis:

- The memoized top-down approach reduces the running time to `O(n*C)` , making it efficient for cases where `n` and `C` are not too large.

Knapsack with Repetition (Unbounded Knapsack)

This variation of the Knapsack problem allows for an unlimited number of copies of each type of item.

Problem Definition:

- **Items**: There are `n` different types of items.
- **Infinite Copies**: Each type has an infinite number of copies, with each copy having a value `V[i]` and weight `W[i]` .
- **Objective**: Maximize the total value without exceeding the knapsack capacity `C` .

Recursive Tree Visualization:

The recursive tree for the Unbounded Knapsack problem can be visualized as a tree where each node represents a state `f(i)`, which is the maximum value achievable with knapsack capacity `i`. At each node, we have `n` branches corresponding to the `n` choices of items:

```
                              f(C)
                  /             |             \
            f(C-W[1])    f(C-W[2])    ...    f(C-W[n])
              /   |  \       /   |  \               /   |  \
         ...    ...  ...  ...   ...  ...           ...  ...  ...
```

Each level of the tree represents a decision for an item, and the tree has `C` levels representing each unit of capacity of the knapsack.

Dynamic Programming Solution:

Top-Down Approach:

```
knapsack(V[1...n], W[1...n], i, j, dp[1...n][1...C]) {
    if (i == 0 || j == 0) return 0;
    if (dp[i][j] != -1) return dp[i][j];

    if (W[i] <= j) {
        dp[i][j] = max(knapsack(V, W, i-1, j, dp), V[i] + knapsack(V, W, i-1, j-W[i], dp));
    } else {
        dp[i][j] = knapsack(V, W, i-1, j, dp);
    }

    return dp[i][j];
}

main(W[1...n], V[1...n], n, C) {
    for i from 1 to n:
        for j from 1 to C:
            dp[i][j] = -1;
    return knapsack(W, V, n, C, dp);
}
```

Bottom-Up Approach:

```
knapsack(V[1...n], W[1...n], C) {
    dp[0...C] = 0;
    for i from 1 to C:
        for k from 1 to n:
            if (W[k] <= i) {
                dp[i] = max(dp[i], V[k] + dp[i-W[k]]);
            }
    return dp[C];
}
```

Running Time Analysis:

- The running time for the Unbounded Knapsack problem with memoization (Top-Down Approach) is `O(n*C)` .
- The running time for the Unbounded Knapsack problem with the Bottom-Up Approach is also `O(n*C)` .

Summary of Knapsack Problem with Bottom up and Top Down Solution

Knapsack Problem Type	Approach	Time Complexity	Space Complexity	Notes
Bounded Knapsack	Top-Down	O(n*C)	O(n*C)	Uses recursion with memoization.
Bounded Knapsack	Bottom-Up	O(n*C)	O(n*C)	Iteratively builds up a solution using a table.
Unbounded Knapsack	Top-Down	O(n*C)	O(C)	Similar to the bounded version but allows for infinite copies of each item.
Unbounded Knapsack	Bottom-Up	O(n*C)	O(C)	The same as the bounded bottom-up but with a single-dimensional array for space.

Now let's dive into the pseudocode for each of these:

Bounded Knapsack

Top-Down Pseudocode (Bounded Knapsack):

```
// A function to solve the 0-1 Knapsack problem using top-down approach with memoization
knapsack(V[1...n], W[1...n], i, j, dp[1...n][1...C]) {
    // Base cases: if no items left or no capacity left in knapsack
    if (i == 0 || j == 0)
        return 0;

    // Return the stored value if this subproblem has already been solved
    if (dp[i][j] != -1)
        return dp[i][j];

    // If the item can fit in the remaining capacity, choose the maximum of
    // including the item or not including the item
    if (W[i] <= j)
        dp[i][j] = max(knapsack(V, W, i-1, j, dp), // Not including item i
                       V[i] + knapsack(V, W, i-1, j-W[i], dp)); // Including item i
    else
        // If the item can't fit, proceed without including item i
        dp[i][j] = knapsack(V, W, i-1, j, dp);

    // Return the maximum value for the given subproblem
    return dp[i][j];
}
```

Bottom-Up Pseudocode (Bounded Knapsack):

```
// A function to solve the 0-1 Knapsack problem using bottom-up approach
knapsack(V[1...n], W[1...n], C) {
    // Initialize DP table with zeros
    dp[0...n][0...C] = 0;

    // Build the table dp[][] in a bottom-up manner
    for i from 1 to n:
        for j from 1 to C:
            // If item i can be included in the current capacity j
            if (W[i] <= j)
                // Update the dp table by including the current item
                dp[i][j] = max(dp[i-1][j], // Not including item i
                               V[i] + dp[i-1][j-W[i]]); // Including item i
            else
                // If the item cannot be included, carry forward the value without the item
                dp[i][j] = dp[i-1][j];

    // The final answer will be in dp[n][C]
    return dp[n][C];
}
```

Unbounded Knapsack

Top-Down Pseudocode (Unbounded Knapsack):

```
// A function to solve the Unbounded Knapsack problem using top-down approach with memoization
unboundedKnapsack(V[1...n], W[1...n], C, dp[0...C]) {
    // Base case: no capacity left in knapsack
    if (C == 0)
        return 0;
    // Return the stored value if this subproblem has already been solved
    if (dp[C] != -1)
        return dp[C];
    // Initialize the value for the current capacity C
    dp[C] = 0;
    // Loop through all items to find the maximum value that can be put in a knapsack of capacit
    for i from 1 to n:
        // Check if the item can be included in the current capacity C
        if (W[i] <= C)
            // Update the dp array with the maximum value by including or not including the curr
            dp[C] = max(dp[C], V[i] + unboundedKnapsack(V, W, C-W[i], dp));
    // Return the maximum value for the current capacity C
    return dp[C];
}
```

Bottom-Up Pseudocode (Unbounded Knapsack):

```
// A function to solve the Unbounded Knapsack problem using bottom-up approach
unboundedKnapsack(V[1...n], W[1...n], C) {
    // Initialize a DP array for storing maximum value at each capacity from 0 to C
    dp[0...C] = 0;

    // Iterate over each capacity value starting from 0 up to the maximum capacity C
    for i from 0 to C:
        // Iterate through all items and update dp[i] if the item can fit in the current capacit
        for k from 1 to n:
            if (W[k] <= i)
                // Update the dp array to include the maximum value between the current value ar
                // the value obtained by including the current item k
                dp[i] = max(dp[i], // Current value without including item k
                            V[k] + dp[i - W[k]]); // Value including item k

    // The maximum value that can be obtained with the total weight not more than C is stored ir
    return dp[C];
}
```

Notes:

- The top-down approach for both bounded and unbounded knapsack problems uses recursion and memoization to avoid recalculating overlapping subproblems.
- The bottom-up approach iteratively builds a solution from smaller subproblems, filling in a DP table. For the bounded knapsack, this table is two-dimensional, while for the unbounded knapsack, it is a one-dimensional array, which leads to space optimization.
- The time complexity for both the bounded and unbounded knapsack problems is `O(n*C)` for both top-down and bottom-up approaches.
- The space complexity for the top-down approach includes the stack space due to recursion, which can be significant. The bottom-up approach uses a DP table, and in the case of the unbounded knapsack, it can be optimized to a one-dimensional array.

Longest Increasing Subsequence (LIS)

The Longest Increasing Subsequence problem seeks to find the length of the longest subsequence of a given sequence in which all elements of the subsequence are sorted in increasing order.

Problem Definition:

- **Sequence**: Given an array `A[]` of `n` elements.
- **Objective**: Find a subsequence where each element is greater than the previous one.

Approach:

One way to solve the LIS problem is to reduce it to the Longest Common Subsequence (LCS) problem.

Example:

Given the array `A = [3, 1, 2, 5, 4]` , one can think about reducing it to an LCS problem by sorting `A` to get `B = [1, 2, 3, 4, 5]` and finding the LCS between `A` and `B` .

Running Time Analysis:

- A naive approach to the LIS problem has a running time of `O(n^2)` , but optimizations can improve this.

Reduction:

Reducing the LIS problem to an LCS problem allows for reusing algorithms and techniques already known for solving the LCS.

Dynamic Programming for Longest Increasing Subsequence (LIS)

We define `f(i)` as the length of the longest increasing subsequence (LIS) that ends with the element `A[i]` . To find `f(i)` , we must consider all `f(j)` for `j < i` and `A[j] < A[i]` . The recurrence relation for `f(i)` would be:

```
f(i) = 1 + max(f(j)) for all j < i and A[j] < A[i]
```

If there is no such `j` that satisfies the conditions, then `f(i) = 1` since the LIS ending at `A[i]` is just the element `A[i]` itself.

Visualization of the Dynamic Programming Array:

Let's represent an array `A` with elements `[3, 1, 2, 5, 4]`. We want to visualize the calculation of `f(i)`:

```
Array A: [3, 1, 2, 5, 4]

LIS lengths:
f(1) = 1 (Starting with 3, the LIS is just [3])
f(2) = 1 (Starting with 1, the LIS is just [1])
f(3) = 2 (Starting with 2, the LIS can be [1, 2])
f(4) = 3 (Starting with 5, the LIS can be [1, 2, 5])
f(5) = 3 (Starting with 4, the LIS can be [1, 2, 4])

The array of LIS lengths:
[1, 1, 2, 3, 3]
```

Recursive Tree Visualization for LIS:

Here's how you might visualize the recursive tree for computing `f(5)` in the array `A`:

```
                f(5)
               /  |  \
           f(4) f(3) f(2) f(1)
           /  \
       f(3)  f(2) f(1)
       /
     f(2) f(1)
     /
   f(1)
```

In this tree, each node represents a recursive call to compute `f(i)`, and the branches represent the subproblems that need to be solved to compute `f(i)`. The depth of the tree corresponds to the length of the subsequence being considered.

Bottom-Up DP Table Fill Visualization:

For the bottom-up approach, we fill a DP table to compute the length of LIS:

```
DP Table: [1, 1, 2, 3, 3]

Index:     1  2  3  4  5
Array A:  [3, 1, 2, 5, 4]
```

Here, `DP[i]` stores the length of the longest increasing subsequence ending with `A[i]`. We start with the length of 1 for each element (since each element is an LIS of length 1). We iterate over the array and update `DP[i]` by comparing `A[i]` with all previous elements `A[j]` where `j < i`. If `A[j] < A[i]`, we have the option to extend the subsequence ending at `A[j]` by `A[i]`, and we update `DP[i]` accordingly.

Top-Down Approach:

The top-down approach to LIS uses memoization to store the length of the longest increasing subsequence ending at each element.

Pseudocode (Top-Down LIS):

```
LIS(A[1...n], i, dp[1...n]) {
    if (i == 0) return 0; // Base case: no element to the left
    if (dp[i] != -1) return dp[i]; // Return memoized result

    // Initialize the length of LIS ending at i to 1 (the element itself)
    dp[i] = 1;

    // Check all elements to the left of A[i]
    for (k = 1 to i-1) {
        // If an element to the left is less than A[i], it could be part of LIS ending at A[i]
        if (A[k] < A[i]) {
            // Update the LIS length if including A[k] leads to a longer sequence
            dp[i] = max(dp[i], 1 + LIS(A, k, dp));
        }
    }
    return dp[i]; // Return the length of LIS ending at A[i]
}

main(A[1...n]) {
    dp[1...n] = -1; // Initialize memoization array with -1
    l = 0; // Variable to store the length of the longest subsequence

    // Compute the length of LIS for each element in A
    for (i = 1 to n) {
        l = max(l, LIS(A, i, dp));
    }
    return l; // The length of the longest increasing subsequence
}
```

Bottom-Up Approach:

The bottom-up approach calculates the LIS iteratively, storing the results in a table.

Pseudocode (Bottom-Up LIS):

```
LIS_bottom_up(A[1...n]) {
    dp[1...n] = 1; // Initialize the table with 1, as the LIS of each element is at least the e

    // Iterate through the array A
    for (i = 1 to n) {
        // Check all elements to the left of A[i]
        for (k = 1 to i-1) {
            // If an element to the left is less than A[i], update the dp table
            if (A[k] < A[i]) {
                dp[i] = max(dp[i], dp[k] + 1);
            }
        }
    }
    return max(dp[1...n]); // Return the maximum value in dp table which is the LIS
}
```

Running Time Analysis:

- For both the top-down and bottom-up approaches, the running time is `O(n^2)`. This is because, for each element `i`, the algorithm compares it with all previous elements, leading to a nested loop structure.

The top-down approach uses recursion with memoization to store the LIS ending at each element, thus avoiding the recalculation of subproblems. The bottom-up approach builds the solution iteratively by finding the LIS for each element based on the previously computed values, ensuring that each subproblem is solved only once.

Summary of Dynamic Programming Solutions

Dynamic Programming (DP) is a method used to solve problems by breaking them down into simpler subproblems. Here's a summary of the DP approaches to solving the Longest Common Subsequence (LCS), Longest Increasing Subsequence (LIS), and Knapsack problems:

Longest Common Subsequence (LCS)

Problem Definition

- Input: Two sequences
- Objective: Find the length of the longest subsequence present in both sequences.

DP Formulation

- `f(i, j)` : The length of the LCS of the first `i` characters of sequence `A` and the first `j` characters of sequence `B` .

Recurrence Relation

```
f(i, j) = 1 + f(i-1, j-1) if A[i] == B[j]
f(i, j) = max(f(i, j-1), f(i-1, j)) otherwise
```

Longest Increasing Subsequence (LIS)

Problem Definition

- Input: A single sequence
- Objective: Find the length of the longest subsequence in which each element is greater than the previous one.

DP Formulation

- `f(i)` : The length of the LIS ending with the `i-th` element.

Recurrence Relation

```
f(i) = 1 + max(f(j)) for all j < i and A[j] < A[i]
```

Knapsack Problems

Bounded 0-1 Knapsack

Problem Definition

- Input: `n` items, each with a value and weight, and a knapsack with a capacity limit.
- Objective: Maximize the total value without exceeding the capacity.

DP Formulation

- `f(i, j)` : The maximum value that can be obtained with the first `i` items and a knapsack capacity of `j` .

Recurrence Relation

```
f(i, j) = max(f(i-1, j), f(i-1, j-W[i]) + V[i]) if W[i] <= j
f(i, j) = f(i-1, j) otherwise
```

Unbounded Knapsack

Problem Definition

- Similar to the bounded version, but each item can be chosen multiple times.

DP Formulation

- `f(i)` : The maximum value achievable with a knapsack capacity of `i` .

Recurrence Relation

```
f(i) = max(f(i), f(i-W[j]) + V[j]) for all j with W[j] <= i
```

Running Time Analysis

- The running time for the DP solution of the LCS and LIS problems is `O(n^2)` , where `n` is the length of the input sequence(s).

- The running time for the DP solution of the bounded 0-1 Knapsack problem is `O(n*C)` , where `n` is the number of items and `C` is the knapsack capacity.
- The running time for the DP solution of the Unbounded Knapsack problem is also `O(n*C)` .

Space Optimization

For the knapsack problems, space optimization techniques can be applied to reduce the space complexity from `O(n*C)` to `O(C)` by using a single-dimensional array and updating it in reverse order for the bounded knapsack or directly for the unbounded knapsack.

Matrix Chain Multiplication

Given `n` matrices `A1, A2, ..., An` with dimensions `[P0, P1], [P1, P2], ..., [Pn-1, Pn]` where `Pi-1` and `Pi` represent the number of rows and columns, respectively.

For example, the multiplication of `A1 * A2 * A3` with dimensions `[10,100], [100,10], [10,50]` can have different costs based on the sequence of multiplication. Which sequence results in a lower cost needs to be determined.

Let's consider this from a general matrices multiplication perspective:

- Multiplication of two matrices `P[x, q] * q[r, y]` results in `P[x, y]` :
  - Each dot product: `T(n) = Θ(q)`
  - So multiplication: `T(n) = Θ(pqr)`
- What about addition?
  - Addition of two matrices `P[x, q] + P[q, y]` results in `P[x, y]` :
    - Each dot addition: `T(n) = Θ(q)`
    - So addition: `T(n) = Θ(pq)`

Therefore, addition performs better than multiplication in terms of complexity, and the goal is to achieve minimum cost.

Now, considering the question again: Assuming we have many matrices to multiply `A1 * A2 * ... * An` , we need to design the sequence of multiplication.

Which multiplication should be performed last to minimize the total cost?

- If we choose `Ak * ... * An` as the last multiplication:

  ```
  Cost (A1 * A2 * ... * Ak * ... * An) = 0 + f(2,n) + P0 * Pk * Pn
  ```

If we consider to separate the multiplication into two parts: A1, and A2 _ ... _ An.

Then the cost of A1 is 0. (Single matrice does not cost anything when no multiplication happened.)

The cost of `A2 * ... * An` is complicated, and we call it f(2,n).

Also assume A1 has P0 rows and P1 columns, and the result matrix of `A2 * ... * An` has p1 rows and pn columns.

Then the cost to multiply A1 and the result matrix of `A2 * ... * An` is P0 _ Pk _ Pn, where k is a number between 1 to n - 1.

General Formula

To calculate the minimum multiplication cost, the general formula is:

```
f(1,n) = min{ f(1,k) + f(k+1,n) + Pi-1 * Pk * Pn }
where k = 1...n-1
```

Considering any interval in the sequence `A1 * A2 * ... * Ai * ... * Aj * ... * An` :

```
f(i,j) = min{ f(i,k) + f(k+1,j) + Pi-1 * Pk * Pj }
where k = i...j
```

The base case is for a single matrix `f(i,i) = 0` .

The time complexity `T(1,n) = 2^(n-1)` is very expensive, so let's optimize it from bottom-up.

Optimization

Assume `f(i, j) = min cost of A*i * A*(i+1) * ... * A_j` , where i <= j.

For normal cases in matrices:

```
- f(i, i) + f(i+1, j) + multiplication cost between the result matrices
- f(i, i+1) + f(i+2, j) + multiplication cost between the result matrices
.
.
.
- f(i, j-1) + f(j, j) + multiplication cost between the result matrices
```

f(i, j) is the minimum from the above list.

The pseudocode to fill the normal case is:

```
// fill the column, c is the col#
for (c = 2...n) {
    // fill the row, r is the row#
    for (r = C-1...1) {
        dp[r][C] = ∞;
        for (k = r...C-1) {
            dp[r][C] = min(dp[r][C], dp[r][k] + dp[k+1][C] + P[r-1] * P[k] * P[C]);
        }
    }
}
return dp[1][n];
```

Time complexity `T(n) = Θ(n^3)` .

Smaller Example

Let's see a smaller case example:

0	1	2	3
10	100	10	50

Matrix multiplication: A1 _ A2 _ A3
Dimensions: [10,100], [100,10], [10,50]

The dp table for storing the cost would look like:

	1	2	3
1	0	10,000	15,000
2	0	0	50,000
3			0

From the above, we can derive the cost of multiplication for a smaller subset of matrices.

For A2 * A3, the cost is found at dp[2][3] and so on.

Longest Increasing Path from Matrix

Given a `n x n` matrix with positive integers, find the longest increasing path where you can only move down and to the right.

Matrix A

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	73	51	35	27	23	97	78	93	19
1	35	75	30	6	20	12	78	59	42
2	91	10	39	70	83	7	93	51	62
3	77	89	71	88	94	69	29	81	89
4	81	35	5	79	43	100	2	88	80
5	78	53	38	82	31	64	11	46	61
6	6	90	13	16	43	58	12	45	25
7	68	54	30	55	33	32	59	94	47
8	14	99	100	89	24	95	87	1	89

Define `f(i, j)` as the maximum path length ending with `c[i, j]` at position `(i, j)`.

Consider reversing: `c[i-1, j] --> c[i, j]` and `c[i, j-1] --> c[i, j]`.

Then:

```
f(i, j) = max {
  1 + f(i-1, j) if A[i-1, j] < A[i, j],
  1 + f(i, j-1) if A[i, j-1] < A[i, j]
}
```

Solution is `max { f(i, j) }` for all `i, j`.

Time Complexity: `T(n) = O(n^2)`

Pseudocode for Longest Path in Matrix:

```
for i = 1 to n {
  for j = 1 to n {
    dp[i, j] = 1;  // Initialize dp table
    if (i > 1 && A[i-1, j] < A[i, j]) {
      dp[i, j] = max(dp[i, j], dp[i-1, j] + 1);
    }
    if (j > 1 && A[i, j-1] < A[i, j]) {
      dp[i, j] = max(dp[i, j], dp[i, j-1] + 1);
    }
  }
}
return max(dp);
```

Longest Path Variation

Let's consider a variation of the problem where you can move in all four directions and determine the longest increasing path.

Define `f(i, j)` as the longest path length ending with the position `(i, j)`.

```
f(i, j) = max {
  1 + f(i-1, j) if A[i-1, j] < A[i, j],
  1 + f(i, j-1) if A[i, j-1] < A[i, j],
  1 + f(i+1, j) if A[i+1, j] < A[i, j],
  1 + f(i, j+1) if A[i, j+1] < A[i, j]
}
```

However, this recursion has a circular dependency problem. The circular dependency doesn't hold since this is an enforced increasing path. The issue is that you need to sort from smallest to largest value, which means you need to sort the matrix, which is too complex.

It's easier to use a top-down approach with memoization.

Pseudocode for Top-Down Approach with Memoization:

```
function increasing_path_length(A[], i, j, dp[]) {
  if (dp[i, j] != -1) {
    return dp[i, j];
  }
  // Initialize dp table
  dp[i, j] = 1;
  // Check all four directions
  if (i > 1 && A[i-1, j] < A[i, j]) {
    dp[i, j] = max(dp[i, j], increasing_path_length(A, i-1, j, dp));
  }
  if (j > 1 && A[i, j-1] < A[i, j]) {
    dp[i, j] = max(dp[i, j], increasing_path_length(A, i, j-1, dp));
  }
  // Consider other two directions if you move in all directions
  return dp[i, j];
}

function main(A) {
  dp = initialize to -1 for all elements;
  s = 1;
  for i = 1 to n {
    for j = 1 to n {
      s = max(s, increasing_path_length(A, i, j, dp));
    }
  }
  return s;
}
```

Time Complexity: `T(n) = O(n^2)`

Summary of Dynamic Programming Solutions

Quick Summary: 5 types of recursion

1. **Two Sequences**: e.g., LCS (`Longest Common Subsequence`).
2. **Single Sequence**: e.g., LIS (`Longest Increasing Subsequence`), `Subarray Sum` .
3. **Knapsack [0-1, Unbounded]**.
4. **Interval**.

5. **Position**: e.g., `Increasing Path Length` .

For each type of question, we design recursion differently and consider the implications on time complexity and memoization.

Coin Change

Given unlimited copies of coins with denominations `[1, 5, 10, 25]` , what is the minimum number of coins required to make a certain amount of value?

Steps to solve:

1. **Identify the type of question**: This is an "unbounded knapsack" problem type.
2. **Think about the recursion design**: Define the function `f(i)` as the minimum number of coins needed to make change for value `i` .

Recursion Formula:

```
f(i) = min {
    f(i - d[k]) + 1 for all denominators d[k]
}

f(i) = min {
    f(i - 1) + 1,
    f(i - 5) + 1,
    f(i - 10) + 1,
    f(i - 25) + 1
}
```

DP Table Initialization:

dp	1	2	...	n

Time Complexity:

T(n) = O(n * k)

where `n` is the amount of value and `k` is the number of denominators.

Exam Score Optimization

Given `n` questions, each with a certain points and time required to solve, what is the maximum score achievable within a total time limit `T` for the exam?

Problem Analysis:

1. **Question type**: This is a "0-1 knapsack" problem type.
2. **Define the function**: `f(i, j)` is the maximum score achievable given the first `i` questions and a time limit `j`.

Recurrence Relation:

```
f(i, j) = max {
    f(i-1, j - time[i]) + points[i],
    f(i-1, j)
}
```

Time Complexity:

T(n) = O(n * T)

where `n` is the number of questions and `T` is the total time for the exam.

Alphabet Sequence Problem

Given a mapping of alphabets to numbers where A=1, B=2, ..., Z=26, consider a sequence of digits as a coded message. Count the number of possible alphabet sequences that the message could represent.

For example:

- A=1, B=2, ..., Z=26
- AD represents 145 (A=1, D=4, AD=145)
- NE represents 145 (N=14, E=5)

We have a single sequence type question:

1. Identify the single sequence type question.
2. Design recursion to count the number of alphabet sequences.

The recursion is defined as:

- `f(i)` = number of alphabet sequences given digits [1:i]
- The recursion relies on the previous computed states `f(i-1)` and `f(i-2)`.

Pseudocode for the recursion could look like:

```
dp[1...n]   // dp array to store the number of sequences
f(i) = f(i-1) + f(i-2)  // f(i) depends on the two preceding values
```

The DP array is filled from left to right, where `k` represents the current position being filled. The initial state `f(1)` corresponds to the first character or digit of the sequence.

The running time complexity `T(n)` for this problem is `O(n)` because each state is only solved once and we iterate over the sequence once.

dp // DP array representation

1	**2**	**3**	**4**	**...**	**...**	**k-2**	**k-1**	**k**	**...**	**...**	**n**
1	4	5									

The final result `f(n)` is the total number of possible alphabet sequences for the entire given sequence of digits.

Stone Merge Game

Given `n` piles of stones, each time you can merge two adjacent piles, and the score is the sum of stones in these two piles. What's the maximum score after merging all piles?

Problem Illustration:

```
Piles: [2, 5, 4]

Merge [2, 5] => Score: 7
Merge [7, 4] => Score: 11 (Total: 18)

vs.

Merge [5, 4] => Score: 9
Merge [2, 9] => Score: 11 (Total: 20)
```

DP Strategy:

Define `f(i, j)` as the maximum score to merge `A[i: j]`.

Recurrence Relation:

```
f(i, j) = max {
    f(i, k) + f(k+1, j) + sum(A[i: j])
}
```

where `k` is between `i` and `j`.

Base Case:

```
f(i, i) = 0
```