Dynamic Programming

## Classic Dynamic Programming

**Easy/Medium Difficulty**

* Fibonacci Sequence: Find the nth number in the Fibonacci sequence (where each number is the sum of the previous two). A classic example for understanding recursion and memoization.
* Knapsack Problem: Given a set of items with values and weights, and a knapsack with a weight capacity, determine the maximum value you can carry without exceeding the capacity.
* Coin Change Problem: Find the minimum number of coins needed to make up a specific amount of change, given a set of coin denominations.
* Longest Common Subsequence (LCS): Find the longest shared subsequence between two strings (not necessarily contiguous).
* Longest Increasing Subsequence (LIS): Find the longest subsequence in an array where elements are in strictly increasing order.

**Intermediate/Hard Difficulty**

* Matrix Chain Multiplication: Given a sequence of matrices, find the most efficient way (minimum number of multiplications) to multiply them together.
* Word Break Problem: Given a string and a dictionary, find if the string can be segmented into a space-separated sequence of words from the dictionary.
* Maximum Sum Subarray: Find the contiguous subarray within an array that has the largest sum (Kadane's algorithm).
* Rod Cutting Problem: Given a rod and a list of prices for different lengths, determine how to cut the rod to maximize profit.
* Optimal Binary Search Tree: Construct a binary search tree that minimizes the expected search cost, given a set of keys and their search frequencies.

**Tips for Learning with Dynamic Programming**

* Start Small: Begin with simpler problems like Fibonacci or the Knapsack problem.
* Visualize Subproblems: Break down larger problems into overlapping smaller subproblems. Draw tables or diagrams if it helps.
* Recursion First, then Memoization/Tabulation: First approach problems recursively to understand the structure, then improve the solution with memoization (storing results) or tabulation (bottom-up approach).
* Practice Pattern Recognition: Solving multiple DP problems helps identify common patterns in how to break down problems and the types of solutions that apply.

## Dynamic Programming

Dynamic Programming: The case of overlapping subproblems. Any instance where a larger problem can be decomposed into smaller subproblems AND the algorithm has patterns of an overlapping substructure.

* Memoization (Top-Down)
* Tabulation (Bottom-Up)

While Recursion, Memoization and Tabulation are great starting points, think of them as tools instead of the final endpoint in optimization. Additional tools include:

* State & Transitions
  + Examples (Longest Common Subsequence)
  + Focus on how the state (current information) of a subproblem contributes to other states and how computed values contribute to values that have not yet been determined.
  + Can this transition be made more efficient?
* Optimization Strategies
  + Space Complexity
    - Rolling Arrays
      * Do you only need the last few computed values? Can you reuse the same space?
    - State Compression
      * Can you reduce the information stored within a state to make the DP structure smaller?
  + Exploit Problem Structure
    - Precompute useful values like known base cases and edge cases.
    - Subproblem Dependencies
      * + States: dp[i] represents the ith Fibonacci number with the constraint.
        + Transitions: dp[i] = dp[i-2] + dp[i-3]
        + Base Cases: dp[0] = 0, dp[1] = 1, dp[2] = 1
      * In the bigger picture, are the subproblems truly ALL dependent or can subproblems be solved sporadically or in a sequence if there’s a specific ordering? (Ex. Coin Change)
      * Look for repeated work to determine which order is optimal and solves the most subproblems correctly.
      * Overlapping Subproblems:
        + The problem breaks down into smaller subproblems with shared solutions.
      * Optimal Substructure:
        + The optimal solution to the larger problem depends on the optimal solutions to its subproblems.
      * Subproblem Dependencies: (ex. House Robber)
        + A subproblem might not depend on all previous subproblems, but only a select few. Smartly identifying these dependencies allows us to compute only the necessary subproblems, improving efficiency.
        + Recurrence Relation

dist[newIndex] = Math.min(1 + dist[i], dist[newIndex]);

* + - * Graph Subproblem Visualization
        + Nodes: Each node represents a subproblem in our dynamic programming formulation.
        + Edges: A directed edge from node A to node B signifies that the solution to subproblem B depends on the solution to subproblem A. In other words, we need A's answer before we can calculate B's.
        + Directed Acyclic Graphs (DAGs) are key:

If the dependency graph is a DAG (no cycles), topological sorting can reveal an optimal order of computation, ensuring we only solve the necessary prerequisites for a subproblem.

* + - Specialized Data Structures
      * Could the algorithm be sped up by transitioning from an array to a priority queue, a set, or some other special data structure?
    - Greedy or Divide & Conquer Considerations
      * Can either of these techniques be utilized to solve the problem or can they replace the DP solution entirely?
  + Algorithmic Tricks
    - Bitmasking
      * Represents sets of choices compactly and allows for efficient exploration.
    - Convex Hull Optimization
      * Look for techniques to drastically prune the search space.
    - Sliding Window
      * Maintain a relevant window to obtain the best result efficiently.

Types of DP Problems

1. Calculate the ith Fibonacci number.
2. How many ways can you escape a maze? Count the number of different ways to move through a 6x9 grid.
3. Given a set of coins, how can we make 27 cents with the least amount of coins?
4. Given a set of substrings, what are the possible ways to reconstruct the word ‘potentpot’?

# Fibonacci Numbers

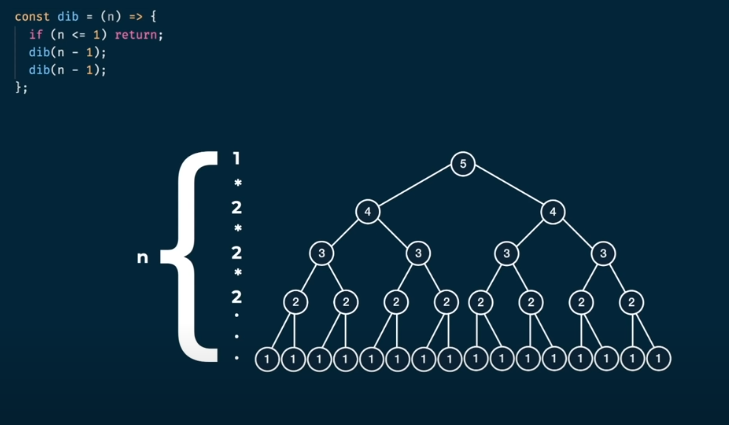
Generate the next number in the Fibonacci Sequence.

Recursive Algorithm

Runtime: O(2^N), Space O(N)

Fib(n) = Fib(n – 1) + Fib(n – 2), Fib(0) = 0 and Fib(1) = 1

Given a large enough value of N, the recursive solution executes too slowly.

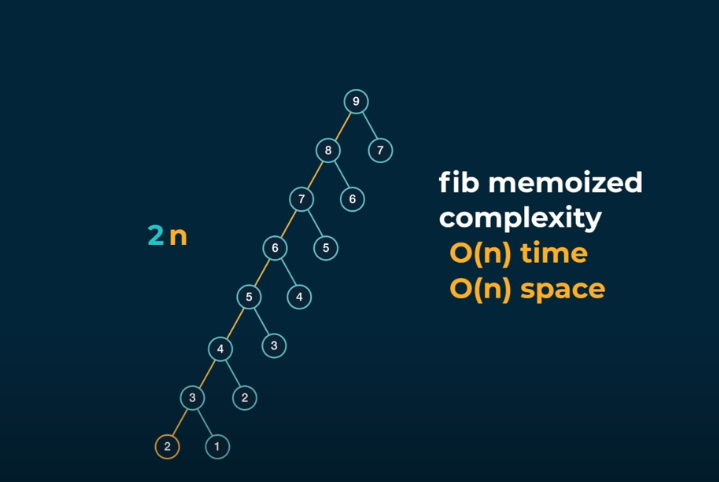


Dynamic Programming Fibonacci

Runtime: O(N), Space O(N)

To solve this problem, a Memoization table needs to be implemented to save the solutions to duplicate subproblems.

The KEY to access the value in the memorization table is THE ARGUMENT passed into the function for easy ID of a repeat problem being solved.



Data Structures for Memoization

1. HashTable
2. Array

# Grid Traveler

How many ways can you travel to a goal in a grid of NxM?

* It is best to break this problem down into base cases first.
  + What does success look like?
  + What does failure look like?
  + What should I return in the case of failure or success?

The Recursive solution quickly gets Reduced to the Recursive Fibonacci where problems are repeating.

Runtime: O(2^(N + M))

Space: O(M + N)

A diagram of a mathematical equation

Description automatically generated with medium confidence

Base Cases

* 0x\* or \*x0 => 0 ways to travel from start to finish (Out of Bounds)
* 1x1 Grid => 1 way to get from start to finish (Don’t move)
* 2x3 Grid => 3 ways to get from start to finish
  + Down, Right, Right
  + Right, Down, Right
  + Right, Right, Down
* 3x3 Grid => 3 + 3 = 6
  + Down => 2x3 Grid Reduction
  + Right => 3x2 Grid Reduction

Memoization Grid Traveler

By saving the results from grids of the same size, we can save time when running the algorithm.

* We MUST check all pairs of M \* N before we can be sure that the repeated calls are ALL in the MEMO table.
  + Runtime: O(M \* N)
* A SINGLE recursive branch will be AT MOST height M + N
  + Space: O(M + N)



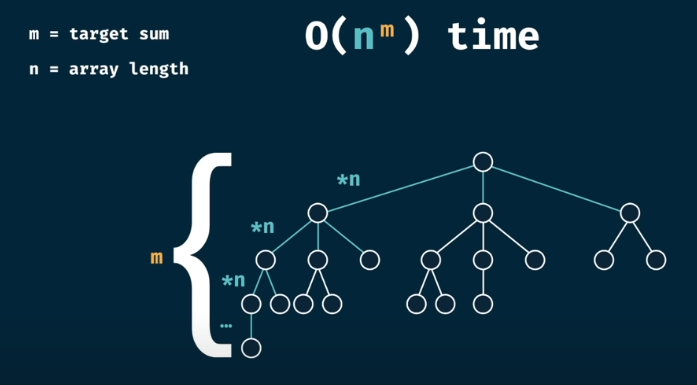
# Memoization Recipe

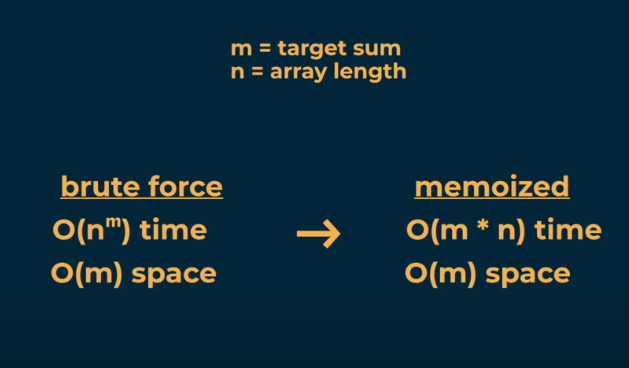
1. Make it work.
   1. Recursion / Brute force is okay.
2. Make it efficient.
   1. Identify the duplicated work that’s being done and save the results in a Hash Table.
   2. Add memo base case.
   3. ONLY MEMOIZE THE INPUT AND NOT THE CHILDREN OF THE INPUT!!!

# Additional DP Problems

Solution: Memoization

* Can Sum?
  + Given a target sum and a list of numbers, return true if the numbers in the numbers array can sum to the target sum and false otherwise.
  + Constraints:
    - The numbers in the array are strictly positive.
    - Th numbers in the array can be used infinitely.

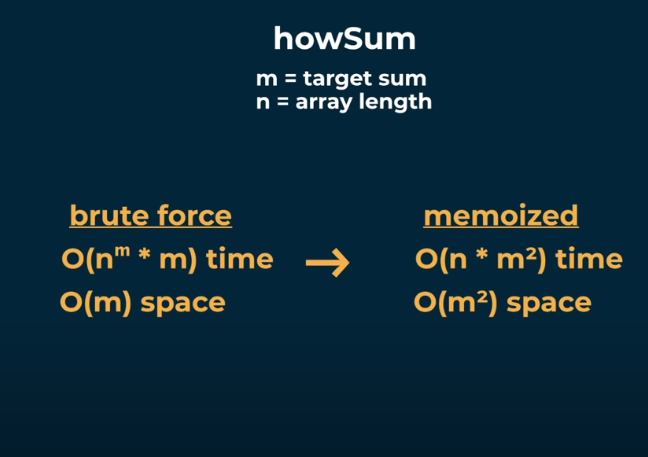




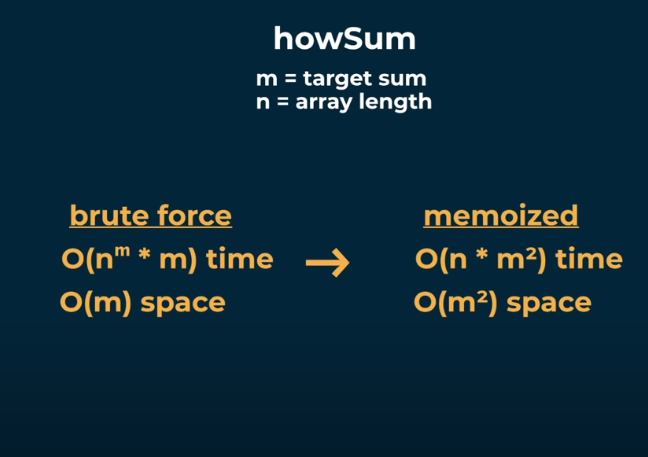
* How Sum
  + Given a target sum and an array of numbers, return the list of numbers in the array that sum up to the target node, if they exist. If not, return an empty array.
    - Constraints
      * All numbers in the array are positive.
      * All numbers in the array may be used more than once.
      * If there are multiple combinations, return any of them.

Consider the Base Cases

* HowSum(0, array) = [], HowSum(targetSum < 0, array) = null
* Add the branch that got us to 0.



* Best Sum
  + Generate a list that sums up to the target Sum, but is also the shortest length of all possible sums.



# Dynamic Programming Progression

A blue screen with yellow text

Description automatically generated

* CanSum -> Decision Problem
* HowSum -> Combinatorics Problem
* BestSum -> Optimization Problem

# Substrings

* canConstruct
  + Given a target string and an array of words, return a Boolean if it is possible to concatenate any combination of strings in the array to construct the target String.
  + Constraints
    - Words in the array can be reused.

Thought Process

* The easiest String to create with the elements in the array is the empty string (BASE CASE)
* Since I need to reconsider all of the array elements, I should use a loop
* countConstruct
  + Given a target String and an array of words, count the number of ways that the target string can be constructed.
* allConstruct
  + Given a target string and an array of words, return a 2D array of all the ways that the target string can be created using words in the word bank

Tabulation

The technique of building out a table for the DP. But instead of doing it recursively, we do it iteratively with a table.

* Key Insight:
  + Instead of working from the parent recursively, Tabulation can work by starting from the children when a base case has already been established (True, 1, 0, False, [], [[]], etc.)
* From that child, identify a pattern of dependencies for them and iterate.
  + Runtime w/ Tabulation O(m\*n), Space w/ Tabulation: O(m \* n)

# Tabulation Recipe

* Visualize the problem as a grid / table.
* Size the table based on the inputs.
* Initialize the table with default values.
* SEED the table with the BASE CASES / Trivial answers.
* FILL FURTHER POSITIONS, with the current position.

HowSum(7, [5, 4, 3])

HowSum(targetSum, nums)

Array[targetSum + 1]

Array[0] = []

For int I = 0; i <= targetSum; i++

If (Array[i] != null)

For num in nums:

Int[] append = Array[i]

Append += [num]

Array[I + num] = append

bestSum(targetSum, nums)

Array[targetSum + 1]

Array[0] = []

For (int I = 0; I <= targetSum; i++) {

If (Array[i] != null) {

For num in nums

Int[] numsArr = Array[i] + [num]

If Array[I + num] == null || numsArr.length < Array[I + num].length

Array[I + num] = numsArr

A screenshot of a computer

Description automatically generated

Explanation of Tabulation Logic

* In the above picture, the first index represents the empty string.
  + The next few indexes indicate every character in the target string.
* At position 4, there are two ways to create the string ‘purp’ using the word array.’
  + The algorithm is updating position 6 for two reasons:
    - The first reason is that ‘le’, the string that can be appended to ‘purp’ is of length 2.
    - The second reason is that by adding the content of position 4 to the content of position 6, the algorithm is saying that given the two ways that I already know how to make ‘purp’, they can both be completed by adding the substring of ‘le’.
* Note: Had there been an additional way to get from ‘purpl’ to ‘purple’ then the final index value would have been 3.

# **Final Thoughts**

When it comes to any Dynamic Programming Problem (Memoization or Tabulation) follow this thought process:

* Identify the most trivial input / BASE CASE and decide what it should return.
  + Success AND Failure
* Decide how to bring the more complicated cases down to the BASE CASES.
  + Are arguments reused? => (Loop) ELSE Knapsack (RECURSIVE)
* Decide whether to Memoize or to Tabulate the results.
* When MEMOIZING, always use the current argument to store the latest values, NOT THE CHILD CALLS!
* TROUBLE FINDING THE PATTERN? Look away from the current method.
  + Is the pattern further back?
  + What’s similar about the inputs and how can that translate to a simpler output?
  + Is the base case too complicated?

# Additional Runtime and Space Notes

* If the number of recursive calls is growing by a factor of N at each LEVEL of the tree and the HEIGHT of the recursive tree in the worse case is of height M, then the runtime will be EXPONENTIAL in the realm of O(N^M)
  + N more work at every LEVEL
  + M LEVELS total
  + Each recursive call is 1 ADDITIONAL STEP (MULTIPLIED BY) ADDITIONAL WORK done at EACH STEP
* SPACE != RUNTIME
* Space Complexity == the MAXIMUM HEIGHT of the tree (MULTIPLIED BY) any additional space needed at any LEVEL!
* EVEN AFTER MEMOIZATION, The Runtime will be proportional to the most work done at any LEVEL (MULTIPLIED BY) The number of UNIQUE MEMO KEYS!

# Things To Work On

* Finding and storing all the substrings into an array.
* Tabulation
  + Substrings
  + ETC.
  + Longest Increasing Subsequence

## References

Longest Increasing Subsequence:

public class LIS {

public static int[][] lisTable(String str1, String str2) {

int m = str1.length();

int n = str2.length();

// Create a table to store LIS lengths

int[][] dp = new int[m + 1][n + 1];

// Initialize first row and column to 0

for (int i = 0; i <= m; i++) {

dp[i][0] = 0;

}

for (int j = 0; j <= n; j++) {

dp[0][j] = 0;

}

// Fill the dp table

for (int i = 1; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if (str1.charAt(i - 1) == str2.charAt(j - 1)) {

dp[i][j] = dp[i - 1][j - 1] + 1; // Match, take diagonal + 1

} else {

dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]); // Not a match, take max from above or left

}

}

}

return dp;

}

public static void main(String[] args) {

String str1 = "ace";

String str2 = "abcde";

int[][] lis = lisTable(str1, str2);

// Print the LIS table

for (int i = 0; i < lis.length; i++) {

for (int j = 0; j < lis[i].length; j++) {

System.out.print(lis[i][j] + " ");

}

System.out.println();

}

}

}

