DESIGN AND ANALYSES OF ALGORITHMS

DYNAMIC PROGRAMMING ALGORITHMS

ABSTRACT:

Many important applied problems in Computer Science involve finding the best way to accomplish some task. Often this involves finding the maximum or minimum value of some function. Such problems come under the class of Optimization Problems. Brute Force methods to solve such problems do exist but are often very inefficient and give exponential time complexities. In order to come up with efficient solutions to such problems, we make use of dynamic programming. Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its sub problems, a property commonly referred to as Optimal Substructure. To understand the workings of DP and check its efficiency, we implemented several algorithms using the DP approach and tested them on a variety of randomly generated valid inputs. Our results verified the correctness of these algorithms and thus, we were able to conclude that DP was a powerful programming technique that could be used for solving optimization problems in polynomial time.

INTRODUCTION:

A lot of real life problems can be solved efficiently with the help of DP. Among the more popular ones, we implemented the following:

a. ***Longest Common Subsequence***:
Given two strings, find the length of the longest common subsequence between them. A common subsequence in a pair of strings is a sequence that follows the same order of characters, but the sequence does not necessarily have to be contiguous.
Input: The algorithm will take two strings, i.e., str1 and str2, as input. Strings can be of variable length, even empty too
Output: The algorithm should return an integer representing the length of the longest common subsequence.

b. ***Shortest Common Super sequence*** :

Given two strings str1 and str2, the task is to find the length of the shortest string that has both str1 and str2 as subsequences.
Input: The algorithm will take two strings, i.e., str1 and str2, as input. Strings can be of variable length, even empty too
Output: The algorithm should return an integer representing the length of the longest common subsequence.

c. Levenshtein Distance (edit-distance):

Given two strings, str1 and str2, find the minimum number of operations required to be operated on str1 to convert it into str2. There can be three kinds of operations: i) insertion of a character at some specific position, ii) deletion of a character at some specific position, or iii) changing a character at some specific position into some other character.
Input: The algorithm will take as input two strings, str1 and str2.
Output: The algorithm should output the minimum cost of converting str1 into str2 or minimum cost of aligning both strings.

d. Longest Increasing Subsequence:

The problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.
Input: The algorithm takes as input a list of unsorted integers.
Output: The algorithm outputs the length of the longest increasing subsequence.

e. Matrix Chain Multiplication:

You are given a chain of matrices to be multiplied. You have to find the least number of primitive multiplications possible to evaluate the result
Input: The algorithm will take as input a list denoting the dimensions of the matrices to be multiplied
Output: algorithm would return the least number of primitive multiplications required to multiply the chain of matrices whose dimensions are given to you

f. 0-1-knapsack-problem:

Given a list of weights and a list of costs, find the optimal subset of things that form the highest cumulative price bounded by the capacity of the knapsack.
Input: As input, the algorithm will get a list of weights, a list of prices, and an integer capacity denoting the total weight the knapsack can carry. weights and prices are aligned to each other, i.e., weight and price of the first object are given by weights [0] and prices [0] respectively.
Output: The algorithm will return the maximum possible profit, i.e., the sum of prices of goods selected based on weights and prices bound by capacity

g. Partition-problem:

You are required to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same.
Input: The algorithm takes as input a list of integers
Output: The algorithm outputs a boolean value (true or false) based on whether the partition is possible or not.

h. Rod Cutting Problem:

You are given a rod of length n meters. You want to sell the rod and earn revenue. You can cut the rod into multiple smaller pieces of sizes 1 through n and sell them too. You are given the price of each size of the rod. Since you want to earn more, you will have to look for the optimal cuts on the rod that would earn the most revenue.
Input: The algorithm would get as input n, the original length of the rod. You can cut the rod into different pieces of sizes 1, 2, up to n. The price for each of these pieces is given in a list of size n called prices.
Output: The algorithm should return the optimal amount of revenue you can get out of the rod provided n and prices.

i. Coin Change Making:

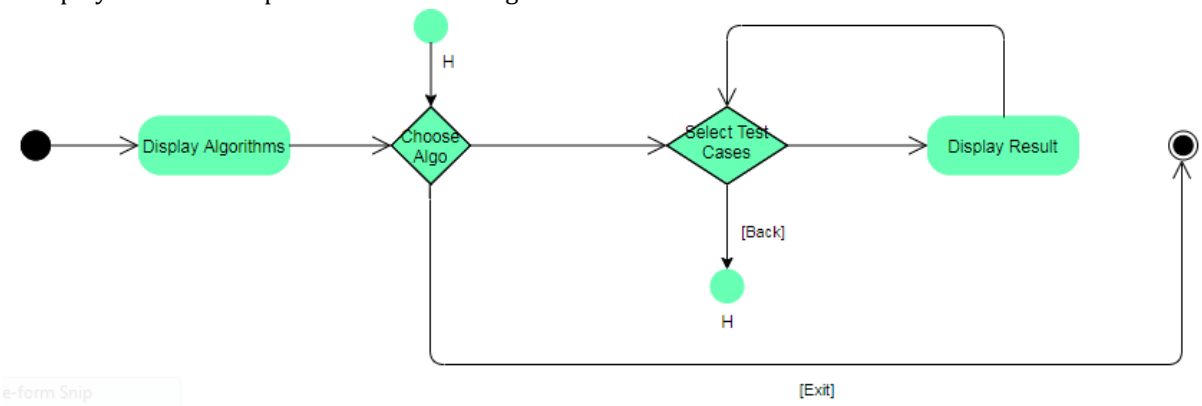
Given coins of different denominations and a total amount of money amount, you are required to compute the fewest number of coins that you need to make up that amount.
Input: The algorithm takes as input a list of integers representing the various values of coins and the total amount
Output: The algorithm outputs the least number of coins needed to make up the amount.

j. Word Break Problem:

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.
Input: The algorithm takes as input a given input string str1 and a dictionary of words dict.
Output: The algorithm outputs a Boolean value (true or false) based on whether the segmentation is possible or not.

PROPOSED SYSTEM:

We have built a small application on React for the UI. On running the application, the user is given a set of algorithms to choose from. The user can click on any of the algorithm, upon which he is displayed a set of "Input" test cases corresponding to that chosen algorithm. After the user clicks on a given test case, its result is displayed in the "Output" column on the right side of the screen.



EXPERIMENTAL SETUP:

The input dataset was primarily constructed by making use of an inbuilt function to generate random numbers and strings. The generated test cases can also be stored in the form of a text file if the user chooses to download them.

A. **LONGEST COMMON SUBSEQUENCE**: Two sequences by using alphabets of your name in random order and repetition with the random length from 30 to 100 characters.

EXAMPLE:

- *Name* = Yusha Arif
- *str1* = sAYrraaYrhrhhssuAArhYYiAiuaffsurYiriiAAsihYiaYaAsa
- *str2* = YaasrrAaYhhYsfuhffarffhrAaAYisAruhAAAiYiYsruiaahfY

B. **SHORTEST COMMON SUPER SEQUENCE**: Same as a.

C. LEVENSHTEIN DISTANCE (EDIT-DISTANCE): Same as a.

D. LONGEST INCREASING SUBSEQUENCE: A sequence of n random numbers was generated in the range 0 to 100 (n varied from 30 to 100).

EXAMPLE:

arr [] = *62 23 18 29 78 2 22 50 85 2 75 54 73 54 64 43 78 11 91 33 32 47 10 26 88 60 34 48 52 68 36 85 35 77 70 19 55 38 25 8 26 51 13*

E. MATRIX CHAIN MULTIPLICATION Same as d.

F. 0-1-KNAPSACK-PROBLEM: A set of n points (n is a random number varying from 10 to 100) with weights and values ranging from 1 to 100 was generated. The capital W was the last three digits of our roll number.

EXAMPLE:

- *Weights = 5 77 26 82 46 54 12 17 59 49 77 53 39 51 57 48 39 81 15 56 90 87 81 63 51 99 21 79 45 58 18 29 5 66 83 60 54 43 25 82 18 30 4 99 94*
- *Values = 97 60 96 25 25 69 76 74 56 8 14 33 68 87 46 46 41 56 36 12 17 55 68 13 89 78 77 76 31 84 53 69 55 6 21 52 20 89 33 29 49 30 53 4 46*
- *Capacity* = 1289 (roll no: 18K-1289)

G. PARTITION-PROBLEM: Same as d.

H. ROD CUTTING PROBLEM: Same as f with *rod length* = 1289, weights as lengths and values as price.

I. COIN-CHANGE-MAKING-PROBLEM: Same as d wi *amount = 1289*

J. WORD BREAK PROBLEM: A set of randomly generated strings from alphabets a to z was generated along with our full name, in small letters and without space, as Input for word break.

Type your text

EXAMPLE:

- *Input Word*: *yushaarif*
- *Dict [ ] = { m, hweu, de ,a, tog, qvcf, mbl ,gv, bphvt, gjm, t, qiutp, ltb, luego, aaegf, k, dxssi, c, rkjl ,z ,l ,p ,xwqo, s, kgf, hs, tv, pqb ,hoqv, tdzke }*

RESULTS:

	Algorithm	Inputs	Output
1	Longest Common Subsequence	Str1: AAasYfYYurhsAhYisarrhYhssYfifarsuiiuYrAhaYrrriahif Str2: YAussYYuiifsrusAhsuuAhiuuruAhurhaifhshYAaaAafaAafY	17
2	Shortest Common Super sequence	Str1: iahushAYYiYuYAffrsaussYiYYhAAiYashrufAusuhfAauAiYA Str2: ffsrrYrsYaAsYaufsAYYfihuuYfYYffYYhAAusaYYAfYiufAua	91
3	Levenshtein Distance	Str1: afrhYsYYihssiAiAiYrshhiuAAafufaffraAfaAaiYiAssuuYh Str2: iYiaYarrasirYhsAuarhiiAAfuAfYurruAaihsfAaaYhuYusrY	49
4	Longest Increasing Subsequence	Arr[] = 66 87 6 40 2 24 28 80 18 63 6 95 26 73 58 56 49 15 63 14 79 35 56 14 15 69 97 87 37 71 17 24 36 88 8 75 87	9
5	Matrix Chain Multiplication	Dim[] = 51 79 92 43 47 74 76 44 60 96 56 73 23 42 92 21 45 8 79 30 60 48 79 8 29 18 32 50 21 46	602048
6	0-1-knapsack-problem	Weights[] = 52 36 94 40 62 70 5 1 78 73 82 Values[] = 52 36 94 40 62 70 5 1 78 73 82 Capacity = 296	476
7	Partition-problem	Arr[] = 45 83 79 26 83 19 55 98 43 83 7 77 32 92 42 53 71 85 68 57 98 37 74 46 48 71 95 92 36 77 10 22 15	False
8	Rod Cutting Problem	Length[] = 96 31 81 50 64 82 86 96 13 77 86 45 92 89 35 Price[] = 71 99 47 95 32 86 80 47 11 89 98 45 95 44 16 Rod Length = 296	440
9	Coin-change-making-problem	Change[] = 89 49 86 46 31 22 68 8 58 88 37 43 15 33 85 89 47 25 97 22 42 3 26 10 75 40 83 4 28 82 23 30 31 46 8 86 42 84 1 30 Amount = 296	4
0	Word Break Problem	Input Word: yushaarif Dict [] = { m, hweu, de ,a, tog, qvcf, mbl ,gv, bphvt, gjm, t, qiutp, ltb, luego, aaegf, k, dxssi, c, rkjl ,z ,l ,p ,xwqo, s, kgf, hs, tv, pqb ,hoqv, tdzke }	False

CONCLUSION:

As amazing as dynamic programming is, you cannot use it on every kind of problem. There are a number of problems that entail an exponential complexity, but their complexity cannot be reduced even after using dynamic programming. In fact, a problem needs to meet very strict prerequisites before it can be solved using dynamic programming. Precisely, a problem must have an optimal substructure, meaning that an optimal solution to a problem can be built using the optimal solutions to that problem's sub problems, and should have overlapping sub problems in order to avoid recomputations through the use of techniques like memorization and tabulation. For problems that satisfy these criteria, dynamic programming provides the tools to a programmer to come up with elegant solutions that are efficient and accurate.