2140. Solving Questions With Brainpower

Dynamic Programming

Problem Description

Medium Array

questions[i] represents a question described by two integers - points_i and brainpower_i. The first integer points_i is the amount of points you'd earn by solving the ith question, and the second integer brainpower_i is the number of subsequent questions you must skip if you decide to solve this question. The rules of the exam are such that you must make decisions on questions in order, deciding to either solve a question (earning

In this LeetCode problem, you are presented with a 0-indexed 2D integer array called questions, where each element

points and skipping the next brainpower_i questions) or to skip it (allowing a decision on the next question). Your goal is to maximize the points you can earn by the end of the exam. To put it into a scenario, assume questions = [[3, 2], [4, 3], [4, 4], [2, 5]]:

• However, if you skip the first question and solve the second one, you'll earn 4 points but will have to skip the last two questions.

• If you solve the first question, you'll get 3 points but miss out on the next two questions.

- Hence, you are looking for a strategy that allows you to accumulate the maximum number of points possible by the end of the

To approach this problem, we need to think about each decision's consequences and explore different scenarios to come up with

decision on the following question.

A naive approach could involve trying all combinations of solving and skipping questions to find the optimal solution, but that would take an enormous amount of time for large arrays. Instead, we look for a <u>Dynamic Programming</u> (DP) approach that helps

The intuition behind the DP solution is to use recursion with memoization to keep track of the maximum points we can earn from any question i onward. At each question, we consider the maximum points of two scenarios: if we solve this question or if we skip it. We use a recursive function that takes an index i and returns the maximum points starting from question i. The base case is

when we go beyond the last question, which earns us 0 points by default. By caching the results of our recursive calls, we greatly reduce the number of calculations, since we ensure that each subproblem is solved exactly once. This makes the solution time-efficient enough to handle large input sizes.

previously computed results to speed up the recursive exploration. dfs(i) represents the maximum points we can earn starting from question i. We take the max between solving the current question and proceeding with dfs(i + brainpower[i] + 1) and skipping the current question with dfs(i + 1). With this approach, we can ensure that we are considering every possible scenario and collecting the maximum points without exhaustively checking each combination, thereby arriving at an optimal solution.

Therefore, the @cache decorator above the recursive function (called dfs in the code) plays a critical role in memoizing

Let's dissect the mostPoints function and the nested dfs function: **<u>Dynamic Programming</u>** via Recursion: The dfs function embodies the concept of dynamic programming by breaking down

the problem into smaller subproblems. At each step, we are essentially asking: "What's the maximum points I can get from

this question onwards?" This leads to a recursive relation because the answer to this question depends on the maximum

points from subsequent questions.

question, there are no more points to earn.

us the maximum score we can achieve from the exam.

Recursive Case: At each question **1**, we consider two scenarios:

question and dfs(i + 1) represents the path where we skip it.

Memoization with @cache: The @cache decorator is a Python built-in decorator from the functools module that automatically saves the results of the dfs function calls into memory so that the next time the same function call is made

• Solving the question: We earn points[i] and skip the next brainpower[i] questions. This is done by recursing with dfs(i + brainpower[i] + 1). • Skipping the question: We move to the next question by calling dfs(i + 1) without earning any points. We take the max between these two scenarios. The call to dfs(i + b + 1) represents the path where we solve the current

Recursion Stack: Recursion uses a stack implicitly where each recursive call adds a frame to the stack. Once a call finishes, it

is removed from the stack and the function returns to the previous call. This means we explore each branch (solve or skip)

- Starting the Recursion: After defining the dfs function, we initiate the recursion by calling dfs(0), which evaluates the maximum points starting from the first question.
- Let's illustrate the solution approach with a small example. Given the array questions = [[2, 1], [3, 1], [5, 2]], let's walk through the dynamic programming steps to find the maximum points: Call dfs(0) to start the recursion with the first question.

Solving question 2: ■ Earn 5 points. There's no question 3, so dfs(2 + brainpower[2] + 1) is dfs(5) which returns 0 (base case). ■ The total score for this path is 2 + 5 = 7. Skipping question 2:

The recursive calls, leveraging memoization, ensure that dfs(2) is not recomputed when accessed through different paths.

At the end of this process, dfs(0) returns the answer, which is 7 points - the maximum number of points we can earn for this set

■ Now compute dfs(1): Solving question 1: ■ Earn 3 points and move to dfs(1 + brainpower[1] + 1) which is dfs(3) and returns 0.

class Solution:

Skipping question 0:

Example Walkthrough

■ The total score for skipping question 0 is 5 points. Take the max between solving (dfs(0) = 7) and skipping (dfs(0) = 5) the first question, which is 7 points.

The total score for this path is 3 points.

Move to dfs(2) which, as already computed, gives 5 points.

Move to dfs(3) which also returns 0 (base case).

The total score for this path remains 2 points.

■ Take the max for dfs(2) which is 7 points (solving question 2).

The total score for solving question 0 is 7 points.

■ Move to dfs(0 + 1) which is dfs(1).

Skipping question 1:

Python from typing import List

from functools import lru_cache # Importing `lru_cache` for memoization

Use lru cache to memoize the recursive function results for efficiency

Unpack points (p) and bonus (b) from the current question

Recursive case: Choose the maximum between two options:

return max(points + dfs(index + bonus + 1), dfs(index + 1))

2. Skipping the current question to try the next one

Start the depth-first search from the first question (index 0)

return 0; // No more points can be earned, return 0

long long mostPoints(std::vector<std::vector<int>>& guestions) {

if (index >= numQuestions) {

// Start the calculation from the first question

return dp[index];

const numQuestions = questions.length;

return memoization[index]; // Return the cached result

def mostPoints(self, questions: List[List[int]]) -> int:

@lru cache(maxsize=None)

return 0

return dfs(0)

class Solution {

};

public:

def dfs(index: int) -> int:

if index >= len(questions):

public long mostPoints(int[][] questions) {

points, bonus = questions[index]

of questions by optimally choosing whether to solve or skip each of them.

- Java class Solution { private int numOuestions: // Total number of questions
 - return dfs(0); // Start the depth-first search from the first question // Recursive method using Depth-First Search to calculate the maximum points private long dfs(int index) {
- C++ #include <vector> // Include for using vectors #include <cstring> // Include for using memset

int numQuestions = questions.size(); // Get the total number of questions

// Function to recursively calculate the max points starting from question i

std::function<long long(int)> calculatePoints = [&](int index) -> long long {

- return calculatePoints(0); **}**; **TypeScript** function mostPoints(questions: number[][]): number { // n represents the total number of questions.
- **}**; return findMaxPoints(0);

@lru cache(maxsize=None)

return 0

def dfs(index: int) -> int:

if index >= len(questions):

const [points, bonus] = questions[index];

// Calculate the maximum points by choosing either:

def mostPoints(self, questions: List[List[int]]) -> int:

// 2. Skipping the current question and checking the next question.

Use lru cache to memoize the recursive function results for efficiency

Unpack points (p) and bonus (b) from the current question points, bonus = questions[index] # Recursive case: Choose the maximum between two options: # 1. Earning points for the current question and jumping over the bonus questions # 2. Skipping the current question to try the next one return max(points + dfs(index + bonus + 1), dfs(index + 1))

Base case: If the index is beyond the questions list, no points can be earned

The time complexity of the code primarily depends on two factors: 1. The number of subproblems to solve, which is O(n), where n is the number of questions. 2. The computation done for each subproblem, which is in this case 0(1) since we are only making a choice between two options: solving the

current question and jumping to the one after the bonus, or moving to the next question directly.

- Each state of the dfs function is uniquely defined by the index i of the current question, and since each question gives us two choices, in the worst case, the function could be called for every index from 0 to n-1. However, due to memoization, each state is
- only computed once, and subsequent calls for the same state i return the cached result. Therefore, the time complexity is O(n).
- **Space Complexity**
- 2. The space used by the memoization cache, which stores a result for each subproblem, resulting in O(n) space. Since these two factors are additive, the total space complexity is 0(n) as well. Each question index i is visited only once due to

In conclusion, both the time complexity and the space complexity of the mostPoints function are O(n).

last question.

Intuition

the optimal strategy. The essence of the problem has us consider two options for each question i: solving or skipping it. When solving question i, we earn points[i] and skip the next brainpower[i] questions. If we skip it, we simply move on to making a

us store the results of subproblems to avoid redundant calculations.

The solution is implemented using a recursive function with memoization, two key elements that are often utilized in dynamic programming solutions.

Solution Approach

with the same arguments, the cached result is returned immediately without recomputing. This feature helps in reducing the time complexity from exponential to polynomial. Base Case: If i >= len(questions), the recursion stops (base case) and returns 0 because once we move past the last

one at a time while maintaining only the current path in the stack, which is a depth-first search pattern. By combining these elements we arrive at an efficient solution that systematically explores each decision's consequences and

appropriately tabulates them to avoid unnecessary re-computation. The result is that the first call to dfs(0) will eventually give

Compute dfs(0): Solving question 0: ■ Earn 2 points and move to dfs(0 + brainpower[0] + 1) which is dfs(2). ■ Now compute dfs(2):

The total score for this path is 5 points. ■ Take the max for dfs(1) which is 5 points (skipping question 1).

Solution Implementation

Base case: If the index is beyond the questions list, no points can be earned

1. Earning points for the current question and jumping over the bonus questions

// Public method to start the process and return the maximum points that can be obtained

if (index >= numQuestions) { // Base case: if the index exceeds the number of questions

if (memoization[index] != null) { // If the result for this index is already computed

- private Long[] memoization; // Cache to store solutions to sub-problems private int[][] questionsData; // 2D array representing questions and their points/bonus
- numQuestions = questions.length; // Initialize number of questions memoization = new Long[numQuestions]; // Initialize the cache array questionsData = questions: // Reference to the original questions array
- int points = questionsData[index][0]; // Points for the current question int bonus = questionsData[index][1]; // Bonus (number of questions to skip) for the current question // Recur in two scenarios: either answer the current question and jump over the bonus questions, // or move to the next guestion. Take the maximum of these two choices. return memoization[index] = Math.max(points + dfs(index + bonus + 1), dfs(index + 1));
- return OLL; // If we have already calculated the result for this index, use it and avoid recomputation if (dp[index] != 0) { return dp[index]; // Points for current question and break before next question can be taken int points = questions[index][0], breakTime = questions[index][1]; // Recursively calculate the max points by taking this question or skipping to the next dp[index] = std::max(points + calculatePoints(index + breakTime + 1), calculatePoints(index + 1));

std::vector<long long> dp(numQuestions, 0); // Dynamic programming array to store the max points upto each question

// If the question index is beyond the number of questions, return 0 as there are no more points to collect

// Create an array 'memo' to store the maximum points that can be obtained starting from each question. const memo = Array(numQuestions).fill(0); // Define a recursive function 'findMaxPoints' to find the maximum points from a given question index 'index'. const findMaxPoints = (index: number): number => { // If we have gone past the last question, no points can be scored, return 0. if (index >= numQuestions) { return 0; // If we have already computed the answer for this question, return the stored result. if (memo[index] > 0) { return memo[index];

// Extract the points 'points' and the bonus 'bonus' for skipping questions from the current question.

return (memo[index] = Math.max(points + findMaxPoints(index + bonus + 1), findMaxPoints(index + 1)));

// 1. Solving the current question and adding the result of the next available question.

- // Start finding maximum points from the first question. from typing import List from functools import lru_cache # Importing `lru_cache` for memoization class Solution:
- # Start the depth-first search from the first question (index 0) return dfs(0) Time and Space Complexity

The given Python function mostPoints uses memoization (@cache) to optimize the recursive depth-first search (dfs) function to

solve a dynamic programming problem. It involves making a decision at each question whether to solve it or to skip to the next

one.

Time Complexity

1. The space used by the recursion stack, which, in the worst case, could be 0(n) if we go question by question without jumping through bonuses.

The space complexity is determined by:

caching, which stores a constant amount of information (the maximum points from that question to the end).