TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 1: Dynamic Programming

Session Overview

In this session, students will learn how to use a dynamic programming approach and memoization to optimize solutions by breaking down problems into overlapping subproblems and storing results to avoid redundant calculations. Dynamic programming is a key technique for improving the efficiency of algorithms, especially in optimization problems that would otherwise have high time complexity due to repeated work.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution

- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▼ Standard Problem Set Version 1

Problem 1: Aang's Meditation for Energy Balance

Aang from *Avatar: The Last Airbender* needs to balance his spiritual energy through meditation sessions to prepare for upcoming battles. Each day, the energy he gains from meditation is the sum of the energy gained on the two previous days.

On days 1 & 2, he gains 1 unit of energy each. Write a function `energy_on_nth_day()` that accepts an integer `n` and returns the total amount of energy Aang has gained on day `n`. Use a dynamic programming approach in your solution.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def energy_on_nth_day(n):
    pass
```

Example Usage:

```
print(energy_on_nth_day(1))
print(energy_on_nth_day(2))
print(energy_on_nth_day(5))
print(energy_on_nth_day(7))
```

Example Output:

```
1
1
5
13
```

▶ 💡 **Hint: Dynamic Programming**

Problem 2: Toph's Earthbending Training

Toph is training her earthbending skills by climbing a staircase made of rock steps. Each step requires a certain amount of energy to climb, represented by an integer array `cost` where `cost[i]` is the energy cost to step on the `i`-th step.

Once Toph pays the energy cost for a step, she can either climb one or two steps. She can start from either the first step (index `0`) or the second step (index `1`). Help Toph minimize the total energy required to reach the top of the staircase.

Write a function `toph_training()` that accepts the integer array `cost` and returns the minimum energy required to reach the top.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def toph_training(cost):
    pass
```

Example Usage:

```
print(toph_training([10, 15, 20]))
print(toph_training([1, 100, 1, 1, 1, 100, 1, 1, 100, 1]))
```

Example Output:

```
15
Example 1 Explanation: Toph will start at index 1, pay 15 units of energy, and jump
two steps to the top. Total energy: 15.
6
Example 2 Explanation: Toph will minimize energy by taking steps in a pattern that u
least energy, resulting in a total energy cost of 6.
```

Problem 3: Aang and Zuko's Elemental Duel

Aang and Zuko take turns in a strategic duel to control the elements, with Aang going first. The duel begins with a number `n` representing the strength of the elements on the battlefield.

On each turn, a player must:

1. Choose any `x` such that `0 < x < n` and `n % x == 0`.
2. Reduce the battlefield's strength by `x`, replacing `n` with `n - x`.

If a player cannot make a move, they lose the duel.

Write a function `aang_wins()` that accepts an integer `n` and returns `True` if Aang wins the duel (assuming both Aang and Zuko play optimally), and `False` otherwise.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
def aang_wins(n):
    pass
```

Example Usage:

```
print(aang_wins(2))
print(aang_wins(3))
```

Example Output:

```
True
Example 1 Explanation: Aang reduces the strength by 1, and Zuko has no more moves.

False
Example 2 Explanation: Aang reduces the strength by 1, then Zuko does the same, leav
Aang with no moves.
```

Problem 4: Aang's Training Sequence

As part of his training, Aang needs to practice bending techniques in a specific sequence. A string `sequence` represents the flow of different bending techniques. A string `move` represents a specific bending technique that Aang practices multiple times in a row.

A bending technique is considered k-repeating if the technique, repeated `k` times, appears as a substring in the training sequence. The technique's *maximum k-repeating value* is the highest value of `k` for which this is true. If the technique is not a substring of the sequence at all, the maximum k-repeating value is `0`.

Write a function `max_k_repeating()` that uses a dynamic programming approach to find the maximum `k-repeating` value of `move` in the `sequence`.

```
def max_k_repeating(sequence, move):
    pass
```

Example Usage:

```
print(max_k_repeating("airairwater", "air"))
print(max_k_repeating("fireearthfire", "fire"))
print(max_k_repeating("waterfire", "air"))
```

Example Output:

```
2
Example 1 Explanation: "airair" is a substring in "airairwater".

1
Example 2 Explanation: "fire" is a substring in "fireearthfire", but "firefire" is n

0
Example 3 Explanation: "air" is not a substring in "waterfire".
```

Problem 5: Zuko's Redemption Mission

Zuko is on a redemption mission to gather various supplies for the Earth Kingdom. He has several types of supply tokens, represented by an integer array `tokens`, where each type of token can be exchanged for different amounts of supplies. Zuko needs to gather an exact total of `amount` units of supplies.

Write a function `zuko_supply_mission()` that returns the fewest number of tokens Zuko needs to collect to gather exactly `amount` units of supplies. If it is impossible for Zuko to gather the exact amount, return `-1`.

You can assume Zuko has an infinite supply of each type of token.

```
def zuko_supply_mission(tokens, amount):
    pass
```

Example Usage:

```
print(zuko_supply_mission([1, 2, 5], 11))
print(zuko_supply_mission([2], 3))
print(zuko_supply_mission([1], 0))
```

Example Output:

```
3
Example 1 Explanation: Zuko can gather 11 units of supplies with 5 + 5 + 1 tokens.

-1
Example 2 Explanation: It's impossible for Zuko to gather exactly 3 units of supplie

0
Example 3 Explanation: Zuko doesn't need any tokens to gather 0 units of supplies.
```

Problem 6: Toph and Katara's Training Synchronization

Toph and Katara are practicing their bending skills together. Both have different training sequences, but they want to synchronize their moves as much as possible. A synchronized sequence is a common subsequence that appears in both Toph's and Katara's training routines while preserving the order of the moves.

Given two strings `katara_moves` and `toph_moves`, return the length of their longest common subsequence. If there is no common subsequence, return `0`.

A subsequence of a string is a new string generated from the original string by deleting some characters (without changing the relative order of the remaining characters).

Using dynamic programming, write a function `training_synchronization()` to calculate the longest common subsequence.

```
def training_synchronization(katara_moves, toph_moves):
    pass
```

Example Usage:

```
print(training_synchronization("waterbend", "earthbend"))
print(training_synchronization("bend", "bend"))
print(training_synchronization("fire", "air"))
```

Example Output:

```
5
Example 1 Explanation: The longest common subsequence is "bend" and its length is 5.

4
Example 2 Explanation: The longest common subsequence is "bend" and its length is 4.

0
Example 3 Explanation: There is no common subsequence between "fire" and "air", so t
```

▶ 💡 **Hint: 2-D Dynamic Programming**

▾ Standard Problem Set Version 2

Problem 1: Counting Pikachu's Thunderbolt Charges

Pikachu is preparing for an epic battle, and Ash needs to keep track of how many Thunderbolt charges Pikachu can store in his electric pouch. Pikachu's electric power can be represented in binary form, and the number of Thunderbolt charges corresponds to the number of `1`s in the binary representation of each number.

Given an integer `n`, return an array `ans` of length `n + 1` such that for each number `i` `(0 <= i <= n)`, `ans[i]` is the number of Thunderbolt charges (`1`'s in the binary representation) Pikachu can store.

Write a function `thunderbolt_charges()` to calculate the number of Thunderbolt charges for each number from 0 to `n`.

```
def thunderbolt_charges(n):
    pass
```

Example Usage:

```
print(thunderbolt_charges(2))
print(thunderbolt_charges(5))
print(thunderbolt_charges(10))
```

Example Output:

```
[0, 1, 1]
Example 1 Explanation:
- 0 in binary is `0`, so Pikachu stores `0` Thunderbolt charges.
- 1 in binary is `1`, so Pikachu stores `1` Thunderbolt charge.
- 2 in binary is `10`, so Pikachu stores `1` Thunderbolt charge.

[0, 1, 1, 2, 1, 2]
Example 2 Explanation:
- 0 in binary is `0`, so Pikachu stores `0` Thunderbolt charges.
- 1 in binary is `1`, so Pikachu stores `1` Thunderbolt charge.
- 2 in binary is `10`, so Pikachu stores `1` Thunderbolt charge.
- 3 in binary is `11`, so Pikachu stores `2` Thunderbolt charges.
- 4 in binary is `100`, so Pikachu stores `1` Thunderbolt charge.
- 5 in binary is `101`, so Pikachu stores `2` Thunderbolt charges.

[0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2]
Example 3 Explanation:
- 0 in binary is `0`, so Pikachu stores `0` Thunderbolt charges.
- 1 in binary is `1`, so Pikachu stores `1` Thunderbolt charge.
- 10 in binary is `1010`, so Pikachu stores `2` Thunderbolt charges.
```

▶ 💡 **Hint: Dynamic Programming**

Problem 2: Gary's Pokédollar Trading Strategy

Gary Oak is always looking to maximize his sPokédollar! Each day, the prices of Pokéballs fluctuate, and Gary wants to maximize his profit by buying Pokéballs on one day and selling them on another future day for more Pokédollar.

Given an array `prices` where `prices[i]` is the trade value in Pokédollars on the `ith` day, your task is to help Gary find the maximum profit he can achieve by choosing a single day to buy and a different future day to sell.

If Gary cannot make any profit, return `0`.

Write a function `max_pokedollar_profit()` that calculates the maximum profit Gary can make from these trades.

```
def max_pokedollar_profit(prices):
    pass
```

Example Usage:

```
print(max_pokedollar_profit([7, 1, 5, 3, 6, 4]))
print(max_pokedollar_profit([7, 6, 4, 3, 1]))
```

Example Output:

```
5
Example 1 Explanation: Gary should buy on day 2 (price = 1 Pokédollar) and sell on da

0
Example 2 Explanation: In this case, Gary cannot make any profitable trades, so the m
```

Problem 3: Caterpie's Evolution Sequence

Caterpie is training to evolve into a Butterfree! To evolve, Caterpie must practice completing sequences of increasing steps. Each step sequence can be represented as a contiguous subarray of steps, and Caterpie must complete sequences that are strictly increasing in difficulty. Your task is to help Caterpie count how many strictly increasing step sequences there are.

You are given an array `steps` consisting of positive integers, where each element represents the difficulty of a training step. A subarray is considered strictly increasing if every step in that subarray is greater than the previous one.

Using a dynamic programming approach, implement a function `count_increasing_sequences()` to calculate the number of strictly increasing subarrays.

```
def count_increasing_sequences(steps):
    pass
```

Example Usage:

```
print(count_increasing_sequences([1, 3, 5, 4, 4, 6]))
print(count_increasing_sequences([1, 2, 3, 4, 5]))
```

Example Output:

```
10
Example 1 Explanation:
The strictly increasing subarrays are:
- Subarrays of length 1: `[1], [3], [5], [4], [4], [6]`.
- Subarrays of length 2: `[1,3], [3,5], [4,6]`.
- Subarrays of length 3: `[1,3,5]`.
The total number of subarrays is 6 + 3 + 1 = 10.

15
Example 2 Explanation:
Every subarray is strictly increasing. There are 15 possible subarrays to take.
```

Problem 4: Ash's Team Battle Strategy

Ash Ketchum is preparing for a series of Pokémon battles, but he needs to assemble a team that alternates between different Pokémon types. He has a list of Pokémon names, `pokemon`, and an array of `types` where each type is represented as either `0` (for physical attackers) or `1` (for special attackers). His goal is to select the longest subsequence of Pokémon such that their types alternate (i.e., no two consecutive Pokémon on the team have the same type).

Help Ash find the longest alternating subsequence of Pokémon for his team. The selected team should be as long as possible, and if there are multiple valid solutions, you may return any of them.

Using a dynam9ic programming approach, determine the longest alternating subsequence and return a list with the names of the selected Pokémon.

```
def team_battle_strategy(pokemon, types):
    pass
```

Example Usage:

```
print(team_battle_strategy(["Pikachu", "Bulbasaur", "Charmander"], [0, 0, 1]))
print(team_battle_strategy(["Squirtle", "Pidgey", "Rattata", "Gengar"], [1, 0, 1, 1]
```

Example Output:

```
['Pikachu', 'Charmander']
Example 1 Explanation:
Ash can choose a subsequence like `["Pikachu", "Charmander"]` because types[0] != ty
Another valid subsequence is `["Bulbasaur", "Charmander"]`. The length of the longes

['Squirtle', 'Pidgey', 'Rattata']
Example 2 Explanation:
Ash can select `["Squirtle", "Pidgey", "Rattata"]` because types[0] != types[1] and
types[1] != types[2]. Another valid subsequence is `["Squirtle", "Pidgey", "Gengar"]
The length of the longest subsequence is 3.
```

Problem 5: Team Rocket's Heist Plan

Team Rocket is planning their next big heist along a row of Pokémon Centers. Each Pokémon Center has a certain amount of rare Pokéballs stashed. However, there's a catch! The security systems in adjacent Pokémon Centers are linked, so if they try to rob two centers next to each other on the same night, Officer Jenny will be alerted.

Given an integer array `pokeballs` representing the amount of rare Pokéballs in each Pokémon Center, return the maximum number of Pokéballs Team Rocket can steal without triggering the security system.

Write a function `heist_plan()` to calculate the maximum amount of Pokéballs Team Rocket can steal.

```
def heist_plan(pokeballs):
    pass
```

Example Usage:

```
print(heist_plan([1, 2, 3, 1]))
print(heist_plan([2, 7, 9, 3, 1]))
```

Example Output:

```
4
Example 1 Explanation:
Team Rocket should rob Pokémon Center 1 (Pokéballs = 1) and then rob Pokémon Center
Total Pokéballs stolen = 1 + 3 = 4.

12
Example 2 Explanation:
Team Rocket should rob Pokémon Center 1 (Pokéballs = 2), Pokémon Center 3 (Pokéballs
Total Pokéballs stolen = 2 + 9 + 1 = 12.
```

Problem 6: Mewtwo's Genetic Fusion

Professor Oak has tasked you with a challenging experiment: fusing the DNA of two Pokémon, Mew and Ditto, to create a new species! The fusion process is governed by specific rules, where the genetic sequences of Mew (`dna1`) and Ditto (`dna2`) must be interwoven to form a new DNA sequence (`dna3`). Your goal is to determine if `dna3` can be formed by interleaving the sequences of `dna1` and `dna2` .

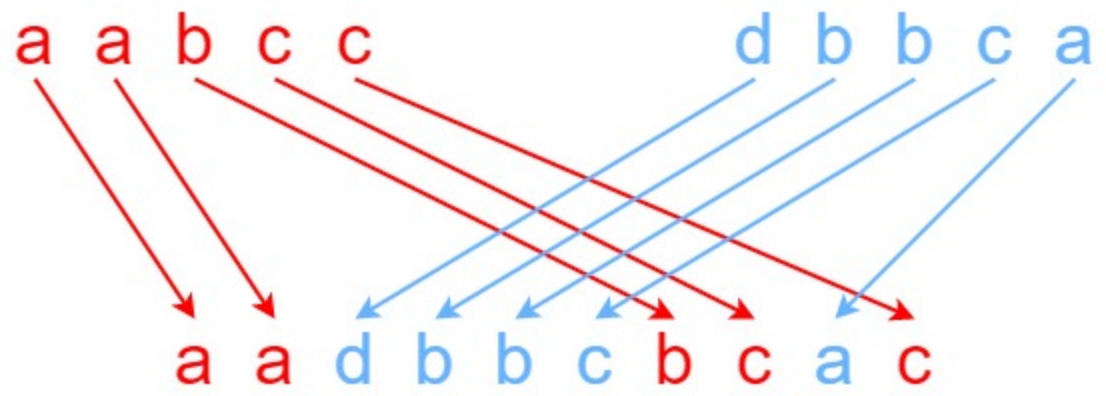
An interleaving of two genetic sequences `g1` and `g2` is a configuration where the sequences are divided into smaller substrings, and then combined alternately without changing the order of the substrings.

Help Professor Oak figure out if `dna3` can be obtained by interleaving `dna1` and `dna2` by implementing a function `genetic_fusion()` that returns `True` if the fusion is successful and `False` otherwise.

```
def genetic_fusion(dna1, dna2, dna3):
    pass
```

Example Usage:

Example 1:




```
print(genetic_fusion("aabcc", "dbbca", "aadbbcbcac"))
print(genetic_fusion("aabcc", "dbbca", "aadbbbaccc"))
print(genetic_fusion("", "", ""))
```

Example Output:

```
True
Example 1 Explanation:
One way to obtain `dna3` is:
Split `dna1` into "aa" + "bc" + "c", and `dna2` into "dbbc" + "a".
Interleaving the two sequences gives "aa" + "dbbc" + "bc" + "a" + "c" = "aadbbcbcac"
Since this forms `dna3`, the result is `True`.

False
Example 2 Explanation:
It is impossible to interleave `dna2` with any other string to form `dna3`.

True
Example 3 Explanation:
Empty strings can always be interleaved to form an empty string.
```

▶ 💡 **Hint: 2-D Dynamic Programming**

Close Section

▶ Advanced Problem Set Version 1

▶ Advanced Problem Set Version 2