Mastery Test 1

Olof Tingskull 2023-02-28

Ad Nauseam

Introduction

The algorithm tackles the problem of sequentially restarting a series of unreliable machines in a factory, each requiring a fixed amount of time for a restart and then operating for a predetermined duration. It starts by calculating the total time needed for all restarts and orders the machines by their operating times in descending order. This prioritization ensures that machines with longer operational periods are restarted first, maximizing operational overlap. The algorithm then checks if each machine, once restarted, will operate long enough to allow for all subsequent restarts. If any machine's operating time isn't sufficient to cover the total restart time, the operator cannot leave work, indicating an endless task. This strategy provides a straightforward method to determine the feasibility of having all machines run simultaneously, thus assessing if completing the restart cycle is possible.

Pseudo Code

For just this problem index is defined to start 1 as it makes the mathimatcal expression more readable.

```
Function ad_nauseam(restart_time, operation_times):
  // Calculate the total time required to restart all machines
  Let n = Length Of operation_times
  Let total_restart_time = n * restart_time

  // Sort the array of functioning times in descending order
  Sort Array operation_times In Descending Order

  // Iterate over the sorted array to check if the task can be completed
  For i = 1 Through n Inclusive:
    // Calculate if the current machines functioning time
    // plus the time required to restart all subsequent machines (including itself)
    // is greater than the total restart time for all machines
    Let operation_time = operation_times Index i
    Let machine_starts_at = i * restart_time

    If operation_time + machine_starts_at Is Smaller Or Equal To total_restart_time:
      // If the condition is not met for any machine, the task cannot be completed
      Return False

  // If the condition is met for all machines, the task can be completed
  Return True
```

Correctness Proof

The algorithm identifies the optimal way to restart machines, which involves restarting them in descending order of operation time, and starting each machine immediately after the previous one. This method is optimal because it adheres to the requirement that once a machine is started, all subsequent machines must be started within that machine's operation time. This requirement can be mathematically expressed as: $f_i + r \cdot i > r \cdot n$, where f_i is the operation time of machine i, r is the restart time, and n is the number of machines. This implies that the operational time requirement decreases linearly for each machine. Therefore, the optimal approach is to start with the machine that has the longest operational time, since it has the highest requirement, and then proceed to the next longest, and so on. It is also evident that the best method is to start the machines as soon as possible, one after another, to save as much time as possible. Since the algorithm checks if the optimal restarting method is feasible, it effectively determines whether the task can be completed.

Caveat Venditor

Introduction

This algorithm employs a dynamic programming approach to minimize the cost of purchasing a certain number of DVDs, n, with the aid of various discount coupons. It builds a solution incrementally, calculating the optimal cost for each possible number of DVDs up to n, by strategically applying the most beneficial coupons. By considering both the addition of a single DVD without new coupons and the application of a new coupon for greater savings, the algorithm explores all avenues for cost reduction. This approach ensures an efficient, polynomial-time solution that leverages the benefits of discount coupons to achieve the lowest possible purchase cost.

Psudo Code

Index start at 0.

```
Define Offer With Properties: free, buy

Define Payment With Properties: pay_for_amount, using_coupons

// Function to find the best price for DVDs using coupons
Function caveat_venditor(total_to_buy, coupons, price_per_movie):
    // Let DP array to find the optimal payment strategy
    Let quantity_payment_dp
        = Array Of Size (total_to_buy + 1) Of Payment (
            pay_for_amount = 0
            using_coupons = Empty Set
        )

    // Iterate through each possible purchase quantity
    For current_quantity = 1 Through total_to_buy Inclusive:
        // Gather all payment options for the current quantity
        Let payment_options = Empty Array Of Payment

        // Incremental purchase without using any new coupons
        no_coupon_payment = quantity_payment_dp Index (current_quantity - 1)
        Increment pay_for_amount In no_coupon_payment By One

        // Include the no-coupon scenario as a valid option
        Append no_coupon_payment To payment_options

        // Evaluate each coupons applicability and benefit
        For Each coupon In coupons:
            // Direct application of an coupon if it covers the current quantity
            If current_quantity <= coupon.buy:
                Let new_payment = New Payment (
                    pay_for_amount = (coupon.buy - coupon.free)
                    using_coupons = Set Including coupon
                )

                Append new_payment To payment_options

            // Combine coupons with previous payments for larger quantities
            Else:
                Let remaining_amount = current_quantity - coupon.buy
                Let remaining_payment = quantity_payment_dp Index remaining_amount

                If coupon Not In remaining_payment.using_coupons:
                    Let new_payment = Copy remaining_payment
                Increment pay_for_amount By (coupon.buy - coupon.free) In new_payment
                Append coupon To using_coupons In new_payment

                Append new_payment to payment_options

        // Select the payment option that minimizes cost
        Let best_payment = Min pay_for_amount in payment_options
        quantity_payment_dp Index current_quantity = best_payment

    // Calculate the total cost based on the optimal payment strategy
    Let best_payment = quantity_payment_dp Index total_to_buy
    Return (pay_for_amount In best_payment) * price_per_movie
```

Correctness Proof

To establish the algorithm's correctness, which minimizes the total cost of purchasing DVDs using various coupons, we define and prove two lemmas.

Lemma 1: The cost of purchasing 0 DVDs is trivially 0, establishing the base case for our dynamic programming algorithm.

Lemma 2: Assuming the least cost for purchasing quantities of DVDs from 0 to $n-1$ is known, the least cost for purchasing n DVDs can be determined by: - Utilizing the optimal costs for quantities 0 to $n-1$ and incorporating the use of an additional coupon, if applicable, or - Adding the cost of one more DVD to the optimal cost for $n-1$ DVDs without using an additional coupon.

Proof of Lemma 2: Given the least costs for up to $n-1$ DVDs, to find the least cost for n DVDs, we examine: 1. **Direct Addition**: The minimal cost for n can be derived from $n-1$ by adding the cost of one DVD. This is because incrementing by one DVD, without additional coupons, extends the optimal solution for $n-1$ in the simplest manner. 2. **Use of an Additional Offer**: Optimal purchasing for n DVDs may involve an coupon not previously optimal for any quantity up to $n-1$. However, due to dynamic programming, only the application of one new coupon needs to be considered for n. This is because the optimal solutions up to $n-1$ already include the best use of coupons, making it redundant to consider multiple new coupons for n.

This approach ensures all strategies for minimizing the cost of n DVDs are explored, avoiding redundancy by leveraging optimal solutions up to $n-1$ and assessing the potential of applying a single new coupon.

By proving Lemma 1 and Lemma 2, we validate the algorithm's ability to compute the least cost for purchasing any quantity of DVDs, ensuring its correctness in optimizing DVD purchases.

Coniunctis Viribus

Introduction

Your algorithm finds the minimum number of trains that can be formed from a sequence of train carriages, ensuring each train's maintenance capacity meets its requirements. It uses dynamic programming to efficiently solve the problem by breaking it down into smaller subproblems, storing their solutions to avoid redundant calculations. The algorithm iteratively explores possible train configurations, updating a table with the optimal number of trains needed up to each point. This approach ensures an efficient and accurate solution by leveraging previous results.

Psudo Code

Index start at 0.

```
// Define Train structure with two properties: start and end indices of a train
Define Train With Properties: start, end

// Main function to calculate the minimum number of trains based on capacities
Function coniunctis_viribus(carriage_capacities):
    // Determine the total number of carriages
```

```
Let n = Length Of carriage_capacities

// Let dynamic programming table for storing optimal train configurations
Let trains_dp = Array With Size n + 1 Of Optional Array Of Train As None
// Base case: no carriages means no trains needed
Set trains_dp Index 0 = Empty Array Of Train

// Iterate through each carriage to explore starting a train from this point
For i = 0 Through n Exclusive:
    // Retrieve the best train configuration up to carriage i
    Let start_at_train = trains_dp[i]
    // Skip if no configuration is found (should not happen after initialization)
    If start_at_train Is None: Continue

    // Reset capacity for the new potential train starting at i
    Let current_capacity = 0

    // Explore extending the train from i to j
    For j = i+1 Through n Inclusive:
        // Accumulate carriage_capacities to support the train from i to j
        Increment current_capacity by carriage_capacities[j-1]
        // Calculate the length and maintenance requirement for the current train
        Let current_train_length = j - i
        Let maintenance = current_train_length ^ 2

        // Continue to next possibility if capacity does not meet maintenance
        If current_capacity Is Less Than maintenance: Continue

        // Create a new train configuration by adding the train from i to j
        Let trains_to_here = Copy start_at_train
        Let new_train = New Train (
            start = i,
            end = j
        )
        Append new_train To trains_to_here

        // Check if this is the shortest configuration for reaching carriage j
        Let so_far_shortest_here = trains_dp[j]
        // Update trains_dp[j] with the current configuration if it's shorter
        If so_far_shortest_here is None Or
            Length Of trains_to_here Less Than Length Of so_far_shortest_here:

            Set trains_dp[j] = trains_to_here

// Return the optimal train configuration for all n carriages
Return trains_dp[n] Is Not None
```

Correctness Proof

This algorithm employs dynamic programming to find the minimum number of trains from n carriages, ensuring each train's maintenance capacity meets its requirement.

Algorithm Strategy:

1. **Initialization**: Sets up a dynamic programming table (`trains_dp`) to store optimal configurations, starting with no carriages equating to no trains.
2. **Iterative Exploration**: Iterates through carriages i from 0 to $n-1$, exploring all potential train configurations

from i to j ($i < j \leq n$), accumulating maintenance capacity and checking it against the required maintenance ($maintenance = (j - i)^2$).

Proof of Correctness:

- **Comprehensive Exploration**: Evaluates every possible starting point and train length, ensuring all configurations are considered.
- **Optimal Substructure**: Maintains the least number of trains for each carriage position, leveraging optimal substructure to ensure solutions to subproblems contribute to the overall problem's solution.

This methodical approach guarantees the algorithm's correctness in identifying the minimum number of trains needed to accommodate all carriages within the given constraints.

Quid Pro Quo

Introduction

The algorithm solves the problem of determining if a player can collect at least one unit of each resource type in a game by trading with NPCs, given initial resource distributions and NPCs' desired resource levels. It employs a recursive strategy to explore all possible trades, leveraging a hash-based mechanism to avoid repeating states. The process involves identifying potential trades that benefit both the player and NPCs, executing these trades in simulation, and recursively exploring subsequent trading opportunities. Success is achieved when the player accumulates at least one of every resource type, with the algorithm returning true; otherwise, it concludes the goal is unattainable and returns false. This approach ensures a comprehensive search for a viable sequence of trades to meet the game's objective.

Psudo Code

Index start 0.

```
Define Structure Trade with properties: with, get, give

Define Structure State with properties:
    player_resources: Array Of Integer
    npc_resources: Array Of Integer
    npc_goals: Array Of Integer

Function recursive_trade(state, explored_states):
    Let (
        player_resources,
        npc_resources,
        npc_goals
    ) = state

    Let n = Length Of player_resources
    Let m = Length Of npc_resources

    Let state_hash = Calculate Hash Of state
    If state_hash Is In explored_states: Return False

    Add state_hash To explored_states

    If All In player_resources Is Greater Than 0:
        Return True

    If Sum Of player_resources Is Less Than n:
        Return False

    Let possible_trades = Empty Array Of Trade
    Let npc_surplus
        = Calculate Array Of (resources - goals) For Each Element

    For i = 0 Through n Exclusive:
        Let player_get = player_resources[i]
        If player_get Is Greater Than 0: Continue

        Let can_trade_this_resource = False

        For j = 0 Through n Exclusive:
            If i Is Equal To j: Continue

            Let player_give = player_resources[j]
            If player_give Is Less Than 1: Continue

            For k = 0 Through m Exclusive:
                Let npc_give_surplus = npc_surplus[k][i]
                Let npc_get_surplus = npc_surplus[k][j]

                If npc_give_surplus Is Less Than 1: Continue
                If npc_get_surplus Is Greater Than -1: Continue

                Let new_trade = New Trade (
                    with = k,
                    get = i,
                    give = j
                )

                Append new_trade To possible_trades
                Set can_trade_this_resource = True

        If Not can_trade_this_resource:
            Return False

    For trade In possible_trades:
        Let (
            with,
            get,
            give
        ) = trade

        Let player_resource_after_trade = Copy player_resources
        Increment player_resource_after_trade[get] By 1
        Decrement player_resource_after_trade[give] By 1

        Let npc_resource_after_trade = Copy npc_resources
        Increment npc_resource_after_trade[with][give] By 1
        Decrement npc_resource_after_trade[with][get] By 1

        Let new_state = New State (
            player_resources = player_resource_after_trade,
            npc_resources = npc_resource_after_trade,
            npc_goals = npc_goals
        )

        Let trade_successful
```

```
            = Run recursive_trade(new_state, explored_states)

        If trade_successful:
            Return True
        Else:
            Continue

    Return false

Function quid_pro_quo(player_resources npc_resources, npc_goals):
    Let state = New State (
        player_resources = player_resources,
        npc_resources = npc_resources,
        npc_goals = npc_goals
    )

    Let explored_states = Empty Set Of Hashes

    Return
        Run recursive_trade(state, explored_states)
```