Assignment 2

Jasraj Singh Johal - 400434346

3 Feb, 2025

Part 1

Q 3

Given boxes with weights $w_1, w_2, \ldots, w_n$ and truck capacity W, the greedy algorithm loads boxes sequentially, dispatching a truck when the next box does not fit.

Algorithm:

- Boxes are loaded into the current truck until the next box does not fit.
- When a box does not fit, the truck is dispatched, and a new truck is started.
- This process continues until all boxes are shipped.

Let:

- G as the number of trucks used by the greedy algorithm.
- O as the number of trucks used by an optimal solution (one that minimizes the number of trucks).

We aim to prove that $G = O$, meaning the greedy algorithm is optimal.

We prove that $G \leq O$ at every step.

Base Case: Initially, $G(0) = O(0) = 0$.
Inductive Hypothesis: Assume $G(i) \leq O(i)$ for some i. We prove it for $i + 1$.

Case 1: Box $i + 1$ fits in the current truck. Then $G(i + 1) = G(i)$, maintaining $G(i + 1) \leq O(i + 1)$.
Case 2: Box $i + 1$ does not fit. The greedy algorithm dispatches the truck and starts a new one, so $G(i + 1) = G(i) + 1$. The optimal solution must also start a new truck, ensuring $O(i+1) \geq O(i)+1$. By induction, $G(i) \leq O(i)$ implies $G(i+1) \leq O(i+1)$.

Thus, $G(n) = O(n)$, proving the greedy algorithm is optimal.

Q 5

Given houses along a road at positions $x_1, x_2, \ldots, x_n$, we must place base stations to ensure each house is within 4 miles of one.

Algorithm: The greedy algorithm works as follows:

1. Start at the westernmost house.
2. Place a base station at the easternmost location that covers this house (i.e., at $x_i + 4$).
3. Remove all houses covered by this base station (those within 4 miles of it).
4. Repeat until all houses are covered.

Let G be the number of base stations used by the greedy algorithm and O the optimal number.

Proof

We prove that $G \leq O$ at every step.

Base Case: Initially, $G(0) = O(0) = 0$, as no base stations are placed.

Inductive Hypothesis: Suppose after placing i base stations, the greedy algorithm has placed no more stations than the optimal solution, i.e., $G(i) \leq O(i)$. We now prove this holds for $i + 1$.

- The greedy algorithm places the $(i + 1)$-th base station at the farthest position possible while still covering all houses left uncovered by previous base stations.
- Let $G(i + 1)$ be the position chosen by the greedy algorithm and $O(i + 1)$ be the position chosen in an optimal solution.
- Since the greedy approach places each station as far east as possible while still ensuring that all uncovered houses are within range, it ensures that $G(i + 1) \geq O(i + 1)$.
- Because $G(i+1)$ is placed at the latest valid position possible, it guarantees that no additional base stations are needed beyond what an optimal solution would require.

By induction, $G(i) \leq O(i)$ implies $G(i + 1) \leq O(i + 1)$, and thus $G(n) = O(n)$.

Therefore, the greedy algorithm produces an optimal solution.

Time Complexity The algorithm processes each house at most once, placing a base station and skipping covered houses. Since each house is visited only once, the total time complexity is $O(n)$.

Q 6

We have to find the optimal sequence to schedule the contestants in a mini-triathlon so that the competition finishes as early as possible. Each contestant has a swimming time S_i, a biking time B_i, and a running time R_i. The contestants swim one at a time, and after finishing swimming, they begin biking and running. Biking and running can happen concurrently.

Algorithm:

1. **Input**: A list of contestants, each with swimming time S_i, biking time B_i, and running time R_i.
2. **Sort** the contestants by the sum of their biking and running times $B_i + R_i$ in **decreasing order**.
3. **Output**: The sorted list of contestants, which gives the optimal order of their starts.

The reason behind sorting by $B_i + R_i$ in decreasing order is that we allow contestants with the longest biking and running times to start earlier, therefore reducing the waiting time for other contestants.

Correctness Proof
We will prove that the greedy algorithm produces an optimal schedule.

Assumption: Assume that we have an optimal solution $T = \{t_1, t_2, \ldots, t_n\}$ where the contestants are not ordered by decreasing $B_i + R_i$. This means there exist two contestants i and j such that i is scheduled before j, but $B_i + R_i < B_j + R_j$.

1. Swap Contestants
Now, we swap the positions of contestants i and j in the schedule. In the new schedule, contestant i will be scheduled after j.

2. Effect of Swap

- When we swap, contestant j will finish their entire triathlon earlier because they now start earlier in the schedule. Since $B_j + R_j > B_i + R_i$, j's larger biking and running times were wasting time when they were scheduled later. By moving j earlier, their finish time improves.
- Contestant i will still finish swimming before j because i starts swimming earlier. However, i's biking and running times will now "run in parallel" with j's tasks. Since $B_i + R_i < B_j + R_j$, i will finish earlier in the new schedule.

Therefore
After swapping contestants i and j, we find that the new schedule does not have a later finish time than the original one. Therefore, the completion time of the new schedule is at most the completion time of the original schedule.

By continuing this process for all inversions (pairs of contestants where $B_i + R_i < B_j + R_j$), we eventually achieve a schedule where contestants are ordered by decreasing $B_i + R_i$, and this schedule has no greater completion time than any optimal solution.

Therefore, the greedy algorithm that orders contestants by decreasing $B_i + R_i$ is optimal.

Time Complexity

Sorting the contestants by $B_i + R_i$ in decreasing order takes $O(n \log n)$, where n is the number of contestants.

Thus, the overall time complexity of the algorithm is: $O(n \log n)$

Q 16

We are given two sets:

- A set of account events $\{x_1, x_2, \ldots, x_n\}$, where each x_i represents the time of an event on a particular account.
- A set of suspicious transactions, where each transaction is represented by an interval $[t_j - e_j, t_j + e_j]$, corresponding to a transaction occurring at time t_j with an error margin e_j.

We need to determine whether it is possible to match each account event x_i to a distinct suspicious transaction interval $[t_j - e_j, t_j + e_j]$ such that for each x_i, $t_j - e_j \leq x_i \leq t_j + e_j$, i.e., each event must lie within its corresponding interval.

Algorithm

1. **Sort the Account Events**: Sort the account events $x_1, x_2, \ldots, x_n$ in increasing order.
2. **Sort the Intervals**: For each suspicious transaction interval $[t_j - e_j, t_j + e_j]$, sort these intervals in increasing order of their *end time* $t_j + e_j$.
3. **Greedy Matching**: For each account event x_i (processed in sorted order), do the following:
   (a) Find the unmatched interval $[t_j - e_j, t_j + e_j]$ such that x_i lies within the interval, i.e., $t_j - e_j \leq x_i \leq t_j + e_j$.
   (b) Among all intervals that contain x_i, choose the one that ends the earliest (i.e., the one with the smallest value of $t_j + e_j$).
   (c) If no such interval is found, terminate the algorithm. (FALSE)
   (d) If such an interval is found, match x_i to this interval and remove it from the list of unmatched intervals.
4. If all account events are matched to distinct intervals, we can output TRUE, indicating a perfect matching exists. Otherwise, terminate and output FALSE.

Correctness Proof

We prove the correctness of the greedy algorithm by contradiction. Suppose there exists a valid perfect matching G between account events and intervals, but the greedy algorithm fails to find it.

Let i be the largest index where the greedy algorithm makes a different match compared to G, and suppose it mismatches account event x_i with an interval $[t_k - e_k, t_k + e_k]$, while G matches it with $[t_j - e_j, t_j + e_j]$.

Since the greedy algorithm selects the interval with the earliest end time that still contains x_i, we know that $t_k + e_k \leq t_j + e_j$. Moreover, since x_i lies within the interval in G, we have $t_j - e_j \leq x_i \leq t_j + e_j$.

Therefore, swapping the two intervals, $[t_k - e_k, t_k + e_k]$ and $[t_j - e_j, t_j + e_j]$, still results in a valid matching. This implies the greedy algorithm would have found the correct match for x_i, contradicting the assumption that it failed.

Thus, if a perfect matching exists, the greedy algorithm will find it.

Time Complexity

- Sorting the account events takes $O(n \log n)$ time.
- Sorting the suspicious transaction intervals by their end times also takes $O(n \log n)$ time.
- For each account event, we search through the intervals to find the best match. Each for loop takes $O(n)$ time per account event. Since there are n account events, this matching step takes $O(n^2)$ time.

Thus, the overall time complexity of the algorithm is $O(n^2)$.

Q 24

We are given a complete binary tree where each edge has a positive length. The goal is to adjust edge lengths so that all root-to-leaf paths have the same total length while minimizing the total increase in edge lengths.

We know that the longest root-to-leaf path sets the target length for all other paths. Any shorter path must be extended to match this length, and our task is to do so while keeping the total added length as small as possible.

Algorithm

- **Find the longest root-to-leaf path:**
  - Compute the maximum path length ($L_{\max}$) from the root to any leaf.
  - This represents the target length for all root-to-leaf paths.
- **Balance the tree bottom-up:**
  - Start from the leaves and move upward through the tree.
  - At each internal node:
    1. Determine the longest and shortest root-to-leaf path in its subtrees.
    2. Extend edges in the shorter subtree so that both paths match.
  - Continue this process until reaching the root.

Correctness

- Since all root-to-leaf paths must be equalized, the longest path determines the required length.
- The greedy approach works optimally because:
  - At each node, we extend the shorter subtree by the minimum required amount.
  - No unnecessary adjustments are made, ensuring minimal total length increase.
- The bottom-up approach ensures that each subtree is fully balanced before making changes at higher levels. Basically so that we don't have to backtrack.

Time Complexity

- Finding the longest root-to-leaf path takes $O(n)$ (a single depth-first search).
- Adjusting edge lengths bottom-up takes $O(n)$ (another DFS traversal).
- The overall complexity is $O(n)$, where n is the number of nodes.

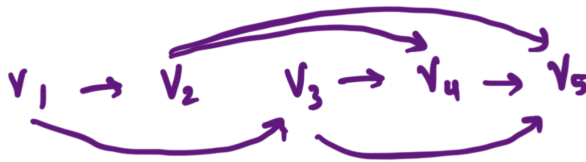
Since each node is processed at most twice (once for computing path lengths and once for balancing), the algorithm runs efficiently in linear time.

By processing the tree from the leaves to the root, we ensure that all paths reach the same length while minimizing the total cost.

Part 2

Q 3

(a)



The correct optimal path from $V_1 - V_5$ is of len(3)

$V_1 \to V_2 \to V_4 \to V_5$

But the algorithm will follow

$V_1 \to V_2 \to V_5$ of len(2)

(b)

An efficient algorithm to find the length of the longest path from v_1 to v_n in an ordered graph can be implemented using DP:

```
longest_path_length(G):
    n = len(G)  # Number of nodes in the graph
    dp = [0] * n  # Initialize dp array where dp[i] is the length of the longest path

    for i = 1 to n-1:  # Iterate through each node
        for j = 0 to i-1:  # Consider all previous nodes
            if (vj, vi) is an edge in G:  # Check if there is an edge from vj to vi
                dp[i] = max(dp[i], dp[j] + 1)  # Update the dp array

    return dp[n-1]  # Return the length of the longest path to vn
```

Algorithm:

- Initialize a dp array where dp[i] represents the length of the longest path from v_1 to v_{i+1}.
- For each node v_i (i from 1 to $n-1$), consider all previous nodes v_j (j from 0 to $i-1$).
- If there's an edge from v_j to v_i, update $dp[i]$ to be the maximum of its current value and $dp[j]+1$.

- The final answer is $dp[n-1]$, which represents the longest path to v_n.

The time complexity of this algorithm is $O(n^2)$, where n is the number of nodes in the graph.

Q 5

We are given a string $y = y_1 y_2 \ldots y_k$ and a function `quality(x)` that assigns a numerical score to any substring x. Our goal is to find a segmentation of y that maximizes the sum of quality scores for the chosen words.

To solve this problem, we use DP:

- Define $dp[i]$ as the **maximum possible quality score** for segmenting the substring $y[0:i]$ (i.e., the first i characters).
- To compute $dp[i]$, we consider all possible words ending at position $i-1$ and take the one that gives the best segmentation:

$$dp[i] = \max_{j<i}(dp[j] + \texttt{quality}(y[j:i]))$$

- The **base case** is $dp[0] = 0$, since an empty string has zero quality.
- We also maintain a **backtracking array** to reconstruct the segmentation.

Pseudocode:

```
1. Initialize dp[0] = 0 and dp[i] = -inf for all i > 0
2. For i from 1 to n:
       a. For each j < i:
              - Let word = y[j:i]
              - If quality(word) is valid:
                     - Update dp[i] = max(dp[i], dp[j] + quality(word))
                     - Store j in a backtrack array to reconstruct the segmentation
3. Reconstruct the optimal segmentation using the backtrack array
```

Correctness Proof

We prove correctness using **mathematical induction**.

- **Base Case:** For $i = 0$, the best segmentation is the empty string, and $dp[0] = 0$, which is correct.
- **Inductive Hypothesis:** Assume that for all $k < i$, $dp[k]$ correctly stores the maximum quality segmentation for $y[0:k]$.
- **Inductive Step:** At index i, we examine all possible segmentations where the last word ends at $i-1$. The recurrence formula:

$$dp[i] = \max_{j<i}(dp[j] + \texttt{quality}(y[j:i]))$$

  ensures that we always select the segmentation with the **highest possible quality**. Since subproblems are solved **optimally** before being used, $dp[i]$ always stores the best possible segmentation score.

Time Complexity

The algorithm iterates over all i from 1 to n, and for each i, it checks all j values from 0 to $i-1$, leading to a worst-case complexity of $O(n^2)$ (assuming `quality(x)` runs in $O(1)$ as given in the question). Thus, the overall runtime is **quadratic**, $O(n^2)$.

Q 7

We are given the prices of a stock for n consecutive days. Our task is to find the best day to buy the stock (let's call it day i) and the best day to sell it (day $j > i$), in order to maximize the profit, $p(j) - p(i)$. If no profit is possible (i.e., the prices are always decreasing), we should return 0.

The previous book/chapter solution involving checking all pairs of days, leads to an $O(n^2)$ time complexity.

DP Algorithm:

1. Initialize two variables:

- `min_price` $= \infty$ (or prices[0]) to track the lowest price encountered so far.
- `max_profit` $= 0$ to track the highest possible profit.

2. Loop through the list of prices:

- For each price $p(i)$, calculate the potential profit if sold at this price: `profit` $=$ $p(i) -$ `min_price`.
- If this profit is greater than the current `max_profit`, update `max_profit`.
- Update `min_price` to the minimum of `min_price` and $p(i)$ (to track the best day to buy).

3. After looping through all the prices, `max_profit` will contain the largest possible profit. If it's greater than 0, return it, otherwise return 0.

Pseudocode:

```
def max_profit(prices):
    min_price = float('inf')
    max_profit = 0

    for price in prices:
        # Calculate potential profit if we sell at this price
        profit = price - min_price
        # Update max_profit if we found a better profit
        max_profit = max(max_profit, profit)
        # Update min_price to the lowest value seen so far
        min_price = min(min_price, price)

    return max_profit
```

Correctness Proof:

We prove the correctness of the algorithm by the following reasoning:

- Initially, `min_price` is set to a very large value (or the first price), and `max_profit` is 0, which is correct for an empty case or no profit.
- As we iterate through the list, for each price $p(i)$, we calculate the potential profit by subtracting the `min_price`. This ensures that we are always considering the best day to sell, given the best day to buy (the minimum price seen so far).
- Since we update `min_price` only when we find a lower price, and `max_profit` is updated whenever a higher profit is found, the algorithm guarantees that the best possible segmentation is selected.
- This approach only requires a single pass through the list of prices, ensuring that we are always making the optimal decision at each step.

Thus, the algorithm always computes the maximum possible profit in a single pass over the prices.

Time Complexity:
The algorithm iterates through the list of prices exactly once. Therefore, the time complexity is $O(n)$, where n is the number of prices (days).

Q 10

(a) Counterexample Showing the Algorithm Is Incorrect

The algorithm makes decisions based on immediate gains and does not consider future consequences of switching machines.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	50
B	5	1	20	20

Why the Given Algorithm Fails:
The algorithm may select A for minute 1 (since a1>b1), assuming local optimizations, but does not account for the benefit of switching to B in the subsequent minutes, missing the optimal solution.

- Minute 1: Choose A (10 steps).
- Minute 2: Move to B (0 steps).
- Minute 3: Stay on B (20 steps).
- Minute 4: Stay on B (20 steps).
- Total = 10 + 0 + 20+ 20 = 50.

[A,move,B,B]

The algorithm greedily picks the machine with the higher number of steps for each minute, but it does not take into account the cost of switching or the future possibilities of each machine's processing power.

The correct approach should look for the global optimal solution, considering all possible transitions, not just local choices based on immediate gains.

Correct Optimal Plan:

- Start with A (10 steps in minute 1).
- Stay in A the whole time; Minute 2, 3, 4 (1 + 1 + 50 = 52 steps)
- Total = (10 + 52) = 62 steps.

[A,A,A,A]

(b)

We can use dynamic programming (DP) to solve this problem efficiently. Here's the idea behind the approach:

Let $dp_A[i]$ be the maximum number of steps achieved up to minute i if we are using machine A at minute i. Let $dp_B[i]$ be the maximum number of steps achieved up to minute i if we are using machine B at minute i.

Transition Relations:

At each minute i, we have two options for each machine:

- **Staying on the same machine**: If we were already on machine A or B at the previous minute, we can stay on the same machine. We just add the number of steps available for that minute on the current machine to the total.
- **Switching machines**: If we were on one machine and want to switch to the other, it costs us one minute of time (no steps), so we consider the best result from two minutes ago.

For minute i:

- If we're on machine A, then:

$$dp_A[i] = \max(dp_A[i-1] + a_i, dp_B[i-2] + a_i)$$

- Similarly, if we're on machine B, then:

$$dp_B[i] = \max(dp_B[i-1] + b_i, dp_A[i-2] + b_i)$$

Base Cases:

- $dp_A[0] = 0$: No steps have been executed yet if we haven't started.
- $dp_B[0] = 0$: Same for machine B.

Final Result:

After we've processed all n minutes, the optimal result will be the maximum value between $dp_A[n]$ and $dp_B[n]$, which represents the maximum steps we can execute, whether we end on machine A or machine B.

By storing the best possible number of steps at each minute for both machines, we ensure that we are always making the best decision based on all previous possibilities.

This way, we avoid recalculating the results for previous minutes repeatedly [memoization]. We efficiently compute the maximum number of steps possible by looking at each decision in terms of staying or switching, while considering the costs involved.

Time Complexity:

The time complexity of this solution is $O(n)$, because we are iterating through the minutes and performing constant time operations to compute $dp_A[i]$ and $dp_B[i]$ at each step.

Q 17

Part (a): The given algorithm fails for the following example:

$$P = [10, 5, 8, 3, 6, 7]$$

Correct longest rising trend: $[10, 5, 8]$ or $[10, 3, 6, 7]$, length = 3.
Algorithm's output: 2.

- Initialize: i = 1, L = 1 (L represents the length of the rising trend)
- j = 2: P[2] = 5 ¡ P[1] = 10, so no change
- j = 3: P[3] = 8 ¿ P[1] = 10, so i = 3, L = 2
- j = 4: P[4] = 3 ¡ P[3] = 8, so no change
- j = 5: P[5] = 6 ¡ P[3] = 8, so no change
- j = 6: P[6] = 7 ¡ P[3] = 8, so no change
- j = 7: End of loop

The algorithm returns L = 2, corresponding to the subsequence [10, 8].

The algorithm only updates when it finds a price higher than the current highest (stored at index i). It fails to consider non consecutive elements that could form a longer rising trend.

Part (b): Efficient Algorithm to Find the Longest Rising Trend

Using dynamic programming (DP) to avoid unnecessary recalculations [memoization].

Algorithm:

- Initialize a DP array dp where $dp[i]$ represents the length of the longest rising trend that ends on day i.
- Set $dp[1] = 1$, as the longest rising trend starting at day 1 is just day 1 itself.
- Set $dp[i] = 1$ for all i.
- Iterate through all pairs of days:
  - For each i from 2 to n, check all previous days j (where $j < i$).
  - If $P[i] > P[j]$, update $dp[i]$ to be $\max(dp[i], dp[j] + 1)$.

- Return the maximum value in the DP array, which represents the length of the longest rising trend.

Time Complexity:
The time complexity is $O(n^2)$, where n is the number of days (or length of the price sequence). This is because we have a nested loop that compares each pair of days.

Collaborator names: Avyya Singh (singa274), Aditya Lahiri (Lahira2)