Q1 (Ishaan)

Problem 1

1. **The FFT of** $(1, 0, 1, -1)$

   Explanation: For a 4-point sequence $x[n] = \{x_0, x_1, x_2, x_3\}$, the DFT $X[k]$ is calculated as:

   $$X[k] = \sum_{n=0}^{3} x[n] \cdot e^{-j2\pi nk/4}$$

   Here, $x = \{1, 0, 1, -1\}$. The twiddle factor is $W_4 = e^{-j\pi/2} = -j$.

   - $X[0] = x[0] + x[1] + x[2] + x[3] = 1 + 0 + 1 - 1 = 1$
   - $X[1] = x[0]W_4^0 + x[1]W_4^1 + x[2]W_4^2 + x[3]W_4^3 = 1(1) + 0(-j) + 1(-1) + (-1)(j) = 1 - 1 - j = -j$
   - $X[2] = x[0]W_4^0 + x[1]W_4^2 + x[2]W_4^4 + x[3]W_4^6 = 1(1) + 0(-1) + 1(1) + (-1)(-1) = 1 + 1 + 1 = 3$
   - $X[3] = x[0]W_4^0 + x[1]W_4^3 + x[2]W_4^6 + x[3]W_4^9 = 1(1) + 0(j) + 1(-1) + (-1)(-j) = 1 - 1 + j = j$

   Final Answer: (1, -j, 3, j)

2. **The worst-case complexity of deterministic quick-sort**

   Explanation: The worst-case for quick-sort occurs when the pivot selection consistently leads to the most unbalanced partition. In a deterministic implementation (ie. always choosing the first or last element), this happens if the array is already sorted. Each partition step divides the array of size n into subarrays of size 0 and $n-1$. The recurrence relation becomes:

   $$T(n) = T(n-1) + \Theta(n)$$

   This recurrence expands to an arithmetic series, resulting in a quadratic time complexity.
   Final Answer: $O(n^2)$

3. **The g.c.d. of two consecutive Fibonacci numbers**

   Explanation: Let the numbers be F_{n+1} and F_n. By the Euclidean algorithm, $\gcd(a, b) = \gcd(b, a \pmod b)$. Since $F_{n+1} = F_n + F_{n-1}$, we have:

   $$\gcd(F_{n+1}, F_n) = \gcd(F_n, F_{n+1} - F_n) = \gcd(F_n, F_{n-1})$$

   Repeating this process gives $\gcd(F_2, F_1) = \gcd(1, 1) = 1$.

   Final Answer: 1

4. **Solving the recurrence** $T(n) = O(n) + \sum_{i=2}^{\lfloor\sqrt{n}\rfloor} T(\lfloor \frac{n}{i^2} \rfloor)$**,** $T(\leq 1) = 1$

   Final Answer: $O(n)$

5. **A Huffman code for `aababcabcdabcdeabcdef`**

   Explanation: Huffman coding uses character frequencies to build an optimal prefix code.

   (a) Frequencies: a:6, b:5, c:4, d:3, e:2, f:1

   (b) Build Tree: Greedily combine the two nodes with the lowest frequencies. One possible tree structure yields the following codes by assigning '0' to left branches and '1' to right branches.

   Final Answer: {a:10, b:01, c:00, d:110, e:1111, f:1110} (Note: Other valid codes are possible).

6. **It is possible to find the median of n numbers in O(_____) time.**

   Explanation: The median can be found in linear time using the "Median of Medians" algorithm. This algorithm provides a deterministic worst-case linear-time performance for the selection problem, which includes finding the median.

   Final Answer: $O(n)$

7. **The edit-distance between SUNNY and SNOWY**

   Explanation:

   (a) 'SUNNY' $\rightarrow$ 'SNNNY' (Substitute U $\rightarrow$ N)

   (b) 'SNNNY' $\rightarrow$ 'SNONY' (Substitute N $\rightarrow$ O)

   (c) 'SNONY' $\rightarrow$ 'SNOWY' (Substitute N $\rightarrow$ W)

   Final Answer: 3

8. **An optimal parenthesization of a matrix-chain product with sequence of dimensions** $\{5, 10, 3, 12, 5, 50, 6\}$

   Explanation: Compute the minimum cost $m[i, j]$ for multiplying matrices $A_i \dots A_j$. By filling out the cost table and a corresponding split table $s[i, j]$, we find the optimal cost is 2010. The s table is then used to reconstruct the parenthesization. The final split for $A_1 \dots A_6$ is at $k = 2$, giving $(A_1A_2)(A_3 \dots A_6)$. The split for $A_3 \dots A_6$ is at $k = 4$, giving $(A_3A_4)(A_5A_6)$.

   Final Answer: $((A_1A_2)((A_3A_4)(A_5A_6)))$

9. **The cut-property states that**

   Final Answer: for any cut of a graph, a minimum-weight edge crossing the cut belongs to some Minimum Spanning Tree.

10. **A language $A \subset \{0,1\}^*$ is said to be mapping-reduced to language $B \subset \{0,1\}^*$ if there exists a computable function f such that for all inputs $w \in \{0,1\}^*$,**

    Explanation: The computable function f must transform any instance w of problem A into an instance $f(w)$ of problem B such that the answer (yes/no, or member/non-member) is preserved.

    Final Answer: $\boldsymbol{w \in A \iff f(w) \in B}$

Q2

Longest Palindromic Subsequence (LPS) – Solution Rubric

1. Definition of DP State (1.5 marks)

Let

$$dp[i][j] = \text{length of the longest palindromic subsequence in substring } s[i..j].$$

2. Recurrence Relation (2 marks)

$$dp[i][j] = \begin{cases} 1 & \text{if } i = j \\ 2 + dp[i+1][j-1] & \text{if } s[i] = s[j] \\ \max(dp[i+1][j],\ dp[i][j-1]) & \text{otherwise.} \end{cases}$$

3. Pseudocode (1.5 marks)

Listing 1: LPS Algorithm

```
function LPS(s):
    n = len(s)
    dp = 2D array of size n x n

    for i in range(n):
        dp[i][i] = 1

    for length in range(2, n+1):
        for i in range(0, n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                dp[i][j] = 2 + dp[i+1][j-1]
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    return dp[0][n-1]
```

4. Proof of Correctness (2 marks)

Claim. The DP value $dp[i][j]$ computed by the recurrence equals the length of the longest palindromic subsequence (LPS) of $s[i..j]$.

Proof (by induction on substring length $L = j - i + 1$).

Base cases.

- $L = 0$ ($i > j$): the empty substring has LPS length 0, and we treat $dp[i][j] = 0$.
- $L = 1$ ($i = j$): a single character is a palindrome of length 1, so $dp[i][i] = 1$.

Inductive hypothesis. Assume for every substring of length $< L$ the DP value equals the true LPS length.

Inductive step. Consider substring $s[i..j]$ with length $L \geq 2$. Let P be an optimal palindromic subsequence of $s[i..j]$ (so $|P|$ is the true LPS length).

- If $s[i] = s[j]$: P uses both $s[i]$ and $s[j]$. Removing those two characters gives a palindrome inside $s[i+1..j-1]$, so

$$|P| = 2 + (\text{some pal. length in } s[i+1..j-1]) = 2 + dp[i+1][j-1],$$

by the inductive hypothesis.

- If $s[i] \neq s[j]$: Any optimal palindrome cannot include both ends, so it mus… $s[i+1..j]$ or $s[i..j-1]$. By IH, the optimum length is

$$\max(dp[i+1][j], dp[i][j-1]),$$

which is the computed recurrence.

Thus, by induction $dp[i][j]$ equals the true LPS length for every substring; hence the algorithm is correct.

5. Time Complexity (1 mark)

We fill an $n \times n$ DP table, each entry in $O(1)$.

$$T(n) = O(n^2)$$

6. Space Complexity (1 mark)

We store an $n \times n$ table.

$$S(n) = O(n^2)$$

7. Reconstruction of Subsequence (1 mark)

Listing 2: Reconstruction of LPS

```
i = 0
j = n - 1
ans_front = ""
ans_back = ""

while i <= j:
    if s[i] == s[j]:
        if i == j:   # middle char
            ans_front += s[i]
        else:
            ans_front += s[i]
```

```
            ans_back = s[j] + ans_back
        i += 1
        j -= 1
    else:
        if dp[i+1][j] >= dp[i][j-1]:
            i += 1
        else:
            j -= 1

LPS = ans_front + ans_back
```

Longest Palindromic Subsequence (LPS) – Using LCS Method

1. Definition of DP State (1.5 marks)

Let

$dp[i][j]$ = length of the longest common subsequence between $s[0..i-1]$ and $s^{rev}[0..j-1]$.

2. Recurrence Relation (2 marks)

$$dp[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + dp[i-1][j-1] & \text{if } s[i-1] = s^{rev}[j-1] \\ \max(dp[i-1][j],\ dp[i][j-1]) & \text{otherwise.} \end{cases}$$

3. Pseudocode (1.5 marks)

Listing 3: LPS via LCS of s and srev

```
function LPS_via_LCS(s):
    n = len(s)
    srev = reverse(s)
    dp = 2D array of size (n+1) x (n+1)

    for i in range(n+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif s[i-1] == srev[j-1]:
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[n][n]
```

4. Proof of Correctness (2 marks total)

Part 1: Proving that $LPS(s) = LCS(s, s^{rev})$ (1 mark)

Claim. The longest palindromic subsequence of s has the same length as the longest common subsequence between s and its reverse s^{rev}.

Proof.

1. **(Every palindromic subsequence is a common subsequence)** Let P be a palindromic subsequence of s. Since P reads the same forward and backward, it will also appear in s^{rev} in the same order. Hence, P is a common subsequence of s and s^{rev}.

   $$\Rightarrow \text{All palindromic subsequences of } s \subseteq \text{Common subsequences of } (s, s^{rev})$$

2. **(Every common subsequence is a palindromic subsequence)** Let Q be a common subsequence of s and s^{rev}. If Q appears in both s and its reverse, then reading Q forward (from s) and backward (from s^{rev}) gives the same sequence. Thus, Q itself must be a palindrome.

   $$\Rightarrow \text{Common subsequences of } (s, s^{rev}) \subseteq \text{Palindromic subsequences of } s$$

3. **(Combining both directions)** Since both sets are equal, their longest subsequences must have the same length.

   $$\boxed{LPS(s) = LCS(s, s^{rev})}$$

□

Part 2: Proving Correctness of LCS DP Algorithm (1 mark)

Claim. The DP table $dp[i][j]$ correctly stores the length of the longest common subsequence between $s[0..i-1]$ and $s^{rev}[0..j-1]$.

Proof (by induction on prefix length).

- **Base Case:** If $i = 0$ or $j = 0$, one of the strings is empty. The longest common subsequence length is 0. Thus, $dp[0][j] = dp[i][0] = 0$ is correct.
- **Inductive Hypothesis:** Assume $dp[a][b]$ is correct for all $a < i$ and $b < j$.
- **Inductive Step:**
  - **Case 1: If** $s[i-1] = s^{rev}[j-1]$ The current characters from both strings match. Since these two characters are equal, they can **extend** the LCS of the previous prefixes $s[0..i-2]$ and $s^{rev}[0..j-2]$. Thus, we include this character and add 1 to the length of the smaller problem:

    $$dp[i][j] = 1 + dp[i-1][j-1].$$

    Whenever characters match, we are simply extending the subsequence that existed before them — just like appending one matching element to the previously best subsequence.

- **Case 2: If** $s[i-1] \neq s^{rev}[j-1]$ The current characters do not match, so one of them cannot be part of the same common subsequence at this position. Therefore, we have two possible options:
  1. Exclude $s[i-1]$ and use $dp[i-1][j]$ (ignore last char of s)
  2. Exclude $s^{rev}[j-1]$ and use $dp[i][j-1]$ (ignore last char of s^{rev})

  We take the better of these two possibilities (whichever yields a longer subsequence):

$$dp[i][j] = \max(dp[i-1][j],\ dp[i][j-1]).$$

By the inductive hypothesis, all smaller subproblems are correctly solved; therefore, $dp[i][j]$ correctly gives the LCS length for prefixes of length i and j.

$$\Rightarrow dp[n][n] = LCS(s, s^{rev}) = LPS(s)$$

□

5. Time Complexity (1 mark)

We fill an $(n+1) \times (n+1)$ DP table with $O(1)$ computation per cell.

$$T(n) = O(n^2)$$

6. Space Complexity (1 mark)

We store an $(n+1) \times (n+1)$ DP table.

$$S(n) = O(n^2)$$

7. Reconstruction of LPS (1 mark)

Listing 4: Reconstructing LPS from dp table

```
i = n
j = n
ans = ""

while i > 0 and j > 0:
    if s[i-1] == srev[j-1]:
        ans = s[i-1] + ans
        i -= 1
        j -= 1
    elif dp[i-1][j] > dp[i][j-1]:
        i -= 1
    else:
        j -= 1

LPS = ans
```

Q3 (Vishal)CLICK ME FOR ANSWER KEY (https://q3-a-key.tiiny.site/)

Q4 (Saiyam)

Given positive integers n and k, along with $p_1, \ldots, p_n \in [0, 1]$, you wish to determine the probability of obtaining exactly k heads when n biased coins are tossed independently at random, where p_i is the probability that the i^{th} coin comes up heads. Give an efficient algorithm for this task. Suppose you wish to simulate an unbiased coin using all/some of the above coin(s), how would you go about it? (7+3=10 marks)

Part 1: Algorithm for Probability of k Heads (7 Marks)

Phase 1: Algorithm Description (3 Marks)
Model Answer

- **State Definition**: This problem can be solved efficiently using **dynamic programming**. Let $DP[i][j]$ represent the probability of obtaining exactly j heads after tossing the first i coins. Our goal is to compute $DP[n][k]$. For a space-optimized solution, a 1D array, $dp[j]$, can store the probability of getting j heads after considering the first i coins.
- **Base Cases**: Before tossing any coins ($i = 0$), the following must be true:
  - The probability of having exactly 0 heads is 1. Therefore, $DP[0][0] = 1$.
  - The probability of having any number of heads greater than 0 is 0. Therefore, $DP[0][j] = 0$ for all $j > 0$.
- **Transition Function (Recurrence Relation)**: When we consider the i^{th} coin with heads probability p_i, the probability of having j heads, $DP[i][j]$, is the sum of two disjoint events:
  1. We had j heads from the first $i-1$ coins (with probability $DP[i-1][j]$) AND the i^{th} coin landed tails (with probability $1 - p_i$).
  2. We had $j-1$ heads from the first $i-1$ coins (with probability $DP[i-1][j-1]$) AND the i^{th} coin landed heads (with probability p_i).

This gives the recurrence relation:

$$DP[i][j] = DP[i-1][j] \times (1 - p_i) + DP[i-1][j-1] \times p_i$$

Marking Criteria

- **(1 Mark)** for a clear **State Definition**.
- **(1 Mark)** for the correct **Transition Function**.
- **(1 Mark)** for correctly identifying both **Base Cases** (0.5 for $DP[0][0] = 1$ and 0.5 for $DP[0][j] = 0$).

Phase 2: Time and Space Analysis (2 Marks)
Model Answer

- **Time Complexity**: The algorithm iterates through each of the n coins. For each coin, it performs calculations for each head count from 0 to k. This results in two nested loops, giving a total time complexity of $\mathbf{O(n \times k)}$.
- **Space Complexity**: The space-optimized version uses a single array of size $k + 1$. Therefore, the space complexity is $\mathbf{O(k)}$.

Marking Criteria

- **(1 Mark)** for correct justification of Time Complexity.
- **(1 Mark)** for correct justification of Space Complexity.

Phase 3: Algorithm Correctness (2 Marks)
Model Answer

- The correctness can be proven by induction on the number of coins, i.
- **Base Case ($i = 0$)**: The initialization ($dp[0] = 1$, $dp[j > 0] = 0$) is correct.
- **Inductive Hypothesis**: Assume after $i-1$ coins, $dp[j]$ correctly stores the probability of getting j heads.
- **Inductive Step (i)**: The transition function is derived from the law of total probability, correctly summing the probabilities of the two mutually exclusive ways to achieve j heads with i coins. Thus, the update is correct.

Marking Criteria

- **(2 Marks)** for a complete and logical correctness argument. Partial marks (1) for a conceptually correct but incomplete argument.

Analysis of Suboptimal Solutions (Graded out of a maximum of 2/7) A brute-force solution would generate all 2^n possible sequences of H/T, calculate the probability of each valid sequence (one with k heads), and sum them up. This is an inefficient $O(2^n)$ algorithm because its runtime grows exponentially with the number of coins, making it impractical for large n. For this reason, it does not receive full marks.

Marking Guide for Suboptimal Answers:

- **(0.5 Mark)** For describing the method of enumerating all 2^n outcomes.
- **(0.5 Mark)** For explaining how to calculate the probability of a single outcome sequence.
- **(0.5 Mark)** For explaining the need to sum the probabilities of all sequences with exactly k heads.
- **(0.5 Mark)** For time and space complexity analysis.

Part 2: Simulating an Unbiased Coin (3 Marks)

Phase 1: How to Simulate
Model Answer

1. **Select** a single coin whose probability of heads, p, is not 0 or 1.
2. **Toss** the coin twice to get a sequence of two outcomes.
3. **Interpret** the result:
   - If the outcome is **Heads-Tails (HT)**, output "Heads".
   - If the outcome is **Tails-Heads (TH)**, output "Tails".
   - If the outcome is **Heads-Heads (HH)** or **Tails-Tails (TT)**, discard and repeat from Step 2.

Phase 2: Why it Works
Model Answer

- The method works because it isolates two mutually exclusive events that occur with identical probability.
- Let the probability of heads be p and tails be $(1 - p)$. The probabilities of the key sequences are:
$$P(\text{HT}) = p \times (1 - p)$$
$$P(\text{TH}) = (1 - p) \times p$$
- Since $P(\text{HT}) = P(\text{TH})$, assigning outputs to these two events and discarding others guarantees a perfectly fair 50/50 result.

Analysis of an Alternative Simulation Method (1 Mark)

Calculate the entire DP table, find a state (i, j) where $dp[i][j] = 1/2$, and then simulate by tossing i coins and checking if the number of heads is j. This is a creative idea, but it is not a feasible or reliable method for two primary reasons:

1. **The Guarantee Problem**: For a general set of biased coins, there is **no guarantee** that any state (i, j) will have a probability of exactly $1/2$. The values in the DP table are polynomials of the input probabilities $\{p_k\}$, and it would be an extraordinary coincidence for them to evaluate to precisely 0.5. For most inputs, no such state (i, j) will exist.

2. **The Efficiency Problem**: Even if such a state (i, j) were found, the simulation itself would be highly inefficient. To produce a single simulated coin flip, one would have to physically toss i coins. In contrast, the von Neumann method only requires tossing one coin (repeatedly). If the found state was, for example, $(50, 20)$, this method would require 50 physical tosses for every one simulated result, which is far less practical.

The von Neumann method is superior because it is **universal** (works for any $p \in (0, 1)$) and generally more **efficient**.

How the Method Works
The proposed simulation operates in a three-step process:

1. **Calculate**: Compute the full DP table to find the probability $dp[i][j]$ for all relevant pairs of i (coins) and j (heads).

2. **Search**: Search the computed table to find a specific state (i, j) for which the probability $dp[i][j]$ is exactly $1/2$.

3. **Simulate**: To perform one simulated flip, toss the first i coins. If the outcome is exactly j heads, register one result (e.g., "Heads"). If the outcome is anything other than j heads, register the other result (e.g., "Tails").

Why the Method Works in Theory
This method is valid *if and only if* a state with probability $1/2$ can be found. Its validity rests on the principle of **complementary events**.

Let's define two events:

- **Event A**: Getting exactly j heads from tossing the first i coins.
- **Event B (The Complement)**: Getting any result *other than* j heads.

By assumption, we have found a case where $P(\text{Event A}) = 1/2$. Since Event A and Event B are mutually exclusive and cover all possibilities, their probabilities must sum to 1:

$$P(\text{Event A}) + P(\text{Event B}) = 1$$

Substituting the known probability gives:

$$1/2 + P(\text{Event B}) = 1 \implies P(\text{Event B}) = 1/2$$

Because both the event and its complement have an equal probability of $1/2$, mapping them to the two sides of a coin creates a perfectly fair simulation.

Q5 (Aryan)

1 Algorithm for Making Change with at Most k Coins

1.1 Problem Statement

Design an algorithm to determine if it is possible to make change for an amount n using at most k coins from a given set of denominations $\{x_1, x_2, \ldots, x_n\}$, assuming an infinite supply of each coin.

1.2 Dynamic Programming Approach

A greedy strategy is not optimal for general coin systems. This problem exhibits optimal substructure and overlapping subproblems, making it a prime candidate for dynamic programming. **[1 mark]**

DP State There are many ways to define the DP state. Let's define a 1D array, `dp[i]`, which stores the minimum number of coins required to make change for amount i. If it's impossible to make change for amount j, we store ∞. The table will have dimensions $(n+1)$. **Note: If you have written a 2D dp state, marks have not been cut. [1 mark]**

DP Initialization An amount of 0 can always be made with 0 coins. Therefore, the base case is:

$$\texttt{dp[0]} = 0$$

[0.5 marks]

All other amounts are impossible to form with zero coin types, hence $\texttt{dp[j]} = \infty$ for all $j > 0$. It is important to write what other dp values are initialized to as well, if it has not been taken care of. **[0.5 marks]**

Recurrence Relation For each state i, we have to take a particular coin to reach that state. Thus, the recurrence will be:

1. We **use the j-th coin (x_j) to reach the ith state**. The solution consists of one x_j coin plus an optimal solution for the remaining amount $i - x_j$. The cost is $1 + dp[i - x_j]$. Since we want to minimize the cost, we take a minimum of all possible values.

We seek the solution with the fewest coins, so we take the minimum of these two cases. The recurrence relation is:

$$\texttt{dp}[i] = \min(\texttt{dp}[i - x_j] + 1) \quad \textbf{[0.5 marks]}$$

Also check for validity of $i - x_j$ (ensure it is ≥ 0). **1 mark is given if the entire dp approach is logical and finds the correct answer.**

Complexity

- **Time Complexity:** $O(n^2)$ here **[0.5 marks]**
- **Space Complexity:** $O(n)$ here **[0.5 marks]**.
- $O(n^2)$ or $O(nk)$ space and time complexity solutions are also accepted. Although, **0.5 marks have been cut** if the solution is $O(n^2k)$ time, or equivalent.

Note: The time and space complexity **has to match the algorithm provided**. Just writing the time and space complexity doesn't fetch you marks, even if it is correct according to the marking scheme.

Final Answer After computing the entire DP table, the value `dp[n]` holds the minimum number of coins required to form the amount n. To solve the problem as stated, we compare this value to k:

- If `dp[n]` $\leq k$, it is possible to make change.
- If `dp[n]` $> k$ (or is ∞), it is not possible.

1.3 Proof of Correctness (DP Algorithm) [1 mark]

The proof is binary. Either you get 0 or 1.

We prove the correctness of the algorithm by induction on the state i.

Inductive Hypothesis Assume that for all states i' such that $i' < i$, the value `dp`$[i']$ correctly stores the minimum number of coins required to make change for amount i'.

Note that there are no cyclic dependencies since $i - x_j < i$ always.

Base Case The base case is `dp[0] = 0`. This is correct because an amount of 0 requires 0 coins, and no positive amount can be formed using zero coin types.

Inductive Step Consider the computation of `dp`$[i]$. Any optimal solution for making change for amount i must use a coin x_j. Let us say that our dp computation does not consider this coin x_j, but rather some other coin.

Since it is possible to use x_j, $i \geq x_j$ is satisfied, else it would not be in the optimal solution. Since $i \geq x_j$, our recurrence considers $dp[i] = (dp[i - x_j] + 1)$, and hence, considers the optimal solution. Here, we arrive at a contradiction and hence, our dp state is correct.

2 Optimality of Greedy Algorithm for Canonical Coin Systems

2.1 Problem Statement

Show that if the set of denominations is of the form $x_i = c^i$ for some integer $c > 1$, a greedy algorithm yields an optimal solution (i.e., one with the minimum number of coins).

Identifying the greedy approach [2 mark]: The greedy approach is to take the largest coin denomination repeatedly such that the total sum does not exceed n. **1 mark has been given if the idea is correct, although what is explained is wrong.**

Proof [2 marks]:

Binary grading. Either 0 or 2.

Most of the errors here were assuming that we can represent k in base c, and hence this is optimal. **This is incorrect way of proving**. You are just taking 1 method and saying since it works here, it will be work everywhere. You need to take care of all the methods (which are infinitely many). Such errors have been given a 0, since it is a conceptual flaw in writing proofs.

Theorem. *For a coin system where denominations are integer powers of $c > 1$, the greedy algorithm for making change is optimal.*

Proof by Contradiction. Let the set of denominations be $D = \{c^0, c^1, \ldots, c^m\}$. The greedy algorithm (G) selects the maximum possible number of the largest available coin for the remaining amount.

Assume G is not optimal. Then there must exist an optimal solution (O) that uses fewer total coins than G.

$$\sum_{i=0}^{m} \text{count}_O(c^i) < \sum_{i=0}^{m} \text{count}_G(c^i)$$

Let j be the largest index for which the coin counts in G and O differ. Since G prioritizes larger-valued coins, it must be that $\text{count}_G(c^j) > \text{count}_O(c^j)$. For all $k > j$, we have $\text{count}_G(c^k) = \text{count}_O(c^k)$.

Since the total value of both solutions is the same, the value provided by coins of denomination c^j or less must be equal in both G and O.

$$\sum_{i=0}^{j} \text{count}_G(c^i) \cdot c^i = \sum_{i=0}^{j} \text{count}_O(c^i) \cdot c^i$$

Rearranging this equation, we get:

$$(\text{count}_G(c^j) - \text{count}_O(c^j)) \cdot c^j = \sum_{i=0}^{j-1} (\text{count}_O(c^i) - \text{count}_G(c^i)) \cdot c^i$$

Let $\Delta = \text{count}_G(c^j) - \text{count}_O(c^j)$. Since $\text{count}_G(c^j) > \text{count}_O(c^j)$, $\Delta \geq 1$. The left side of the equation is therefore at least $1 \cdot c^j$.

Now, let's analyze the right side. The greedy algorithm ensures that the number of coins of denomination c^i used, $\text{count}_G(c^i)$, is always less than c, because if it were c or more, $c \cdot c^i = c^{i+1}$, and a single larger coin would have been used instead. So, $0 \leq \text{count}_G(c^i) \leq c - 1$.

The term $\sum_{i=0}^{j-1} (\text{count}_O(c^i) - \text{count}_G(c^i)) \cdot c^i$ must equal the value on the left, but we can find its maximum possible value. The value is maximized when $\text{count}_G(c^i) = 0$ and $\text{count}_O(c^i)$ is as large as possible. However, any optimal solution must also use fewer than c coins of denomination c^i (otherwise they could be exchanged for one c^{i+1} coin, reducing the coin count). So, $\text{count}_O(c^i) \leq c - 1$. The maximum possible value of the right side is:

$$\begin{aligned}
\sum_{i=0}^{j-1} \text{count}_O(c^i) \cdot c^i &\leq \sum_{i=0}^{j-1} (c-1)c^i \\
&= (c-1) \sum_{i=0}^{j-1} c^i \\
&= (c-1) \left(\frac{c^j - 1}{c - 1} \right) \\
&= c^j - 1
\end{aligned}$$

This brings us to a contradiction. Our equation implies:

$$\underbrace{(\text{count}_G(c^j) - \text{count}_O(c^j)) \cdot c^j}_{\geq c^j} = \underbrace{\sum_{i=0}^{j-1} (\text{count}_O(c^i) - \text{count}_G(c^i)) \cdot c^i}_{< c^j}$$

An integer value that is at least c^j cannot be equal to a value that is strictly less than c^j. Thus, our initial assumption that G is not optimal must be false. The greedy algorithm is indeed optimal for this coin system. ■

Q6

Q7- Harpreet

Full Solution

Definition of Matroids

A **matroid** is a mathematical structure that generalizes the concept of independence from vector spaces to arbitrary sets.

A matroid M is an ordered pair $(E, \mathcal{I})$, where:

- E is a finite set (called the *ground set*),
- $\mathcal{I} \subseteq 2^E$ is a collection of subsets of E (called *independent sets*),

satisfying the following axioms:

1. **Non-emptiness:** $\emptyset \in \mathcal{I}$.
2. **Heredity:** If $A \in \mathcal{I}$ and $B \subseteq A$, then $B \in \mathcal{I}$.
3. **Exchange Property:** If $A, B \in \mathcal{I}$ and $|A| < |B|$, then there exists an element $e \in B \setminus A$ such that $A \cup \{e\} \in \mathcal{I}$.

Graphic Matroid

Given an undirected graph $G = (V, E)$:

- The ground set E is the set of edges.
- A subset of edges is said to be **independent** if it forms a **forest** (i.e., contains no cycles).
- The maximal independent sets correspond to **spanning forests**, and if the graph is connected, these are **spanning trees**.

Thus, the **graphic matroid** captures the cycle-free property of edge sets in a graph.

Greedy Strategy (Kruskal's Algorithm via Matroid View)

1. Sort all edges in non-decreasing order of their weights (for MST). For a maximum spanning tree, sort in non-increasing order.
2. Initialize $T = \emptyset$.
3. For each edge e in sorted order:
   - If $T \cup \{e\}$ is independent in the matroid (i.e., it does not form a cycle), then include e in T.
   - Otherwise, skip e.
4. Stop when $|T| = |V| - 1$.

Here, the independence check corresponds to verifying that adding an edge does not create a cycle — directly using the definition of the graphic matroid.

Why Greedy Works– Informal

According to the **Matroid Greedy Algorithm Theorem**:

> If the feasible solutions of a problem form a matroid, then the greedy algorithm produces an optimal solution for any weight function.

Since the spanning tree problem is modeled by the graphic matroid, the greedy strategy (i.e., Kruskal's algorithm) is guaranteed to produce an optimal minimum or maximum spanning tree.

Marks Distribution

Definition of Matroid and Graph Matroid: 5 marks.
Algorithm and Proof sketch of using Graph Matroid to solve MST: 5 marks.

Exceptions

Mentioned only kruskal/prims: 1 marks.
Some of the informal solutions have also been considered but to an extent.

General Note

Grading has been kept lenient.

Q8 (Karthik Venkat)

Question 8

You are given two multisets of integers, (a) a set of apples $A = \{a_1, a_2, \ldots, a_n\}$ and (b) a set of bananas $B = \{b_1, b_2, \ldots, b_m\}$, where each element lies in the range $[1, k]$.

Define the pair-sum frequency function

$$f : \{2, 3, \ldots, 2k\} \to \mathbb{N}$$

as

$$f(w) = |\{(a, b) \in A \times B \mid a + b = w\}|.$$

That is, $f(w)$ counts the number of pairs consisting of one apple and one banana whose weights sum to exactly w.

Design and analyze an efficient algorithm (using FFT that computes discrete convolutions efficiently) to compute $f(w)$ for all $w \in \{2, \ldots, 2k\}$.

Marking Scheme:

- Algorithm and description: 3 marks
- Pseudocode: 4 marks
- Proof of correctness: 3 marks

Answer

Algorithm and Description (3 marks)

We can interpret $f(w)$ as the discrete convolution of two sequences that represent the frequency of apple and banana weights. Let:

$$A'[i] = \text{number of apples with weight } i, \quad 1 \le i \le k$$
$$B'[i] = \text{number of bananas with weight } i, \quad 1 \le i \le k$$

Then, for all w,

$$f(w) = \sum_{i=1}^{k} A'[i] \cdot B'[w - i]$$

This is precisely the definition of convolution.

To compute $f(w)$ efficiently, we can use the **Fast Fourier Transform (FFT)**:

1. Represent A' and B' as coefficient arrays of two polynomials:

$$P_A(x) = \sum_{i=1}^{k} A'[i]x^i, \quad P_B(x) = \sum_{i=1}^{k} B'[i]x^i$$

2. Compute their product polynomial:

$$P_C(x) = P_A(x) \cdot P_B(x)$$

3. The coefficient of x^w in $P_C(x)$ gives $f(w)$.

4. Compute the multiplication using FFT in $O(k \log k)$ time.

Pseudocode (4 marks)

```
Algorithm ComputePairSumFrequency(A, B, k):

Input:
  - Multisets A = {a1, a2, ..., an}
  - Multisets B = {b1, b2, ..., bm}
  - Range limit k

Output:
  - Array f[2..2k] where f[w] = number of (a,b) pairs with a+b=w

1. Initialize arrays freqA[1..k] = 0, freqB[1..k] = 0
2. For each a in A: freqA[a]++
3. For each b in B: freqB[b]++
4. Let FA = FFT(freqA)
5. Let FB = FFT(freqB)
6. Let FC = FA * FB   // pointwise multiplication
7. Let f = InverseFFT(FC)
8. Round all values of f to nearest integer
9. Return f[2..2k]
```

Proof of Correctness (3 marks)

We show that the algorithm correctly computes $f(w)$ for all w.

Proof. Each apple weight i contributes to sums of the form $i+j$ for all banana weights j. The number of such pairs where $i + j = w$ is $A'[i] \cdot B'[w - i]$.

Summing over all valid i:

$$f(w) = \sum_{i=1}^{k} A'[i] \cdot B'[w - i]$$

This is exactly the definition of convolution of A' and B'.

By the convolution theorem:

$$\mathcal{F}^{-1}(\mathcal{F}(A') \cdot \mathcal{F}(B')) = A' * B'$$

where $\mathcal{F}$ denotes the Fourier Transform.

Hence, applying FFT to both sequences, multiplying pointwise, and then applying the inverse FFT correctly produces the convolution, i.e., the required $f(w)$.

The algorithm's complexity is dominated by FFT and inverse FFT operations, both taking $O(k \log k)$ time, making the overall solution efficient and correct. □

Q9

Q10 (Shail Shah)

Solution: The Longest Common Subsequence (LCS) Algorithm

October 9, 2025

---

1. Algorithm Design (Total: 5 Marks)

The solution uses Dynamic Programming. The design is broken down into initialization, defining the recurrence, computing the length, and constructing the subsequence.

A. Initialization and Recurrence

Initialization (1 Mark)

A 2D table, 'C', of size $(m+1) \times (n+1)$ is created. The first row ($C[0, j]$ for $0 \leq j \leq n$) and the first column ($C[i, 0]$ for $0 \leq i \leq m$) are initialized to **0**.

Recurrence Relation (2 Marks)

For all $i > 0$ and $j > 0$, the value of each cell $C[i, j]$ is computed based on the characters x_i and y_j and previously computed values:

$$C[i, j] = \begin{cases} 1 + C[i-1, j-1] & \text{if } x_i = y_j \\ \max(C[i-1, j], C[i, j-1]) & \text{if } x_i \neq y_j \end{cases}$$

B. Computation and Construction

Finding the Length (1 Mark)

- Iterate through the table from $i = 1..m$ and $j = 1..n$.
- Fill each cell $C[i, j]$ using the recurrence relation.
- The final value in the bottom-right cell, $\mathbf{C[m, n]}$, is the length of the LCS.

Finding the LCS Subsequence (1 Mark)

- Begin the **backtracking** process from cell $C[m, n]$.
- Trace a path backwards: if characters match, move diagonally and prepend the character to the result.
- If they do not match, move to the adjacent cell (up or left) with the larger value.
- Continue until the first row or column is reached.

Note on Evaluation

- **1 mark will be deducted** if the pseudo code for the algorithm is not provided.
- **0.5 mark will be deducted** for any mistake in the pseudo code.

2. Algorithm Analysis (Total: 3 Marks)

A. Proof of Correctness (2 Marks)

We prove that the algorithm correctly computes the length of the LCS. Let $L(i, j)$ be the true length of $LCS(X_i, Y_j)$.

Claim. *The algorithm computes* $C[i, j] = L(i, j)$ *for all* $0 \leq i \leq m$ *and* $0 \leq j \leq n$.

Proof. The proof is by induction on $k = i + j$.

Base Case: For $k = 0$, $i = 0$ and $j = 0$. The LCS of two empty strings has length 0. The algorithm sets $C[0, 0] = 0$, so the claim holds. For any case where $i = 0$ or $j = 0$, the LCS is empty (length 0), matching the initialized table values.

Inductive Hypothesis (IH): Assume that for all pairs (i', j') where $i' + j' < k$, $C[i', j'] = L(i', j')$.

Inductive Step: We prove for pairs (i, j) where $i + j = k$.

- **Case 1:** $x_i = y_j$.
  The true length is $L(i, j) = L(i - 1, j - 1) + 1$. By the IH, $C[i - 1, j - 1] = L(i - 1, j - 1)$. The algorithm calculates $C[i, j] = 1 + C[i - 1, j - 1] = 1 + L(i - 1, j - 1) = L(i, j)$. The claim holds.
- **Case 2:** $x_i \neq y_j$.
  The true length is $L(i, j) = \max(L(i - 1, j), L(i, j - 1))$. By the IH, $C[i - 1, j] = L(i - 1, j)$ and $C[i, j - 1] = L(i, j - 1)$. The algorithm calculates $C[i, j] = \max(C[i - 1, j], C[i, j - 1]) = \max(L(i - 1, j), L(i, j - 1)) = L(i, j)$. The claim holds.

By the principle of induction, the claim is true for all i, j.

B. Efficiency / Complexity (1 Mark)

- **Time Complexity:** $\mathbf{O(m \cdot n)}$.
- **Space Complexity:** $\mathbf{O(m \cdot n)}$.

3. Example Execution (Total: 2 Marks)

A. DP Table Construction (1 Mark)

B. Final Result (1 Mark)