A3_Report_123090612

Q1 Tree

①What are the possible solutions for the problem?

1. Depth-First Search (DFS) Approach

The solution provided in the original code already implements this approach, which uses DFS to traverse the tree. This is an effective solution for this problem due to the tree structure and the nature of the problem.

- **Tree Structure**: Since the city is represented as a tree, there are no cycles, and each path from a leaf node to the root is unique.
- **DFS Traversal**: Traverse the tree starting from the root node `t`. For each node, calculate how much health point (HP) the warrior would lose as they pass through each edge on the path from leaf nodes to the root.
- **Edge Analysis**: As the warrior moves from one node to its parent, the magic point (MP) decreases by 1. For each edge, the warrior's HP is reduced by the formula `max(0, w - k)` where `w` is the attack power of the monster and `k` is the MP at that point.

2. Dynamic Programming on Trees

Another possible solution would be to apply dynamic programming (DP) to the tree to calculate the minimum health points required for each node, starting from the root and moving downwards to the leaves.

- **Recursive DP Calculation**: The basic idea is to compute the minimum HP needed at each node starting from the root. For each node, we compute the HP required for each of its children and propagate the values upwards to the root.

3. Breadth-First Search (BFS) with Modified State Tracking

A BFS approach could be used, but it would require modifications to track the state of the warrior's magic points (MP) and health points (HP) as they travel along the tree.

- **State Tracking**: Instead of simply visiting nodes, we would track the current state of both the warrior's MP and HP as they traverse the tree.
- **Level-by-Level Traversal**: Use BFS to explore the tree level by level, updating the warrior's state as they move along each edge.

②How do you solve this problem?

1. Input Parsing and Initial Setup

```python
# Read the number of nodes, edges, and the root node t
n, m, t = map(int, input().split())

# Initialize adjacency list and degrees list
adj = [[] for _ in range(n + 1)]  # Using 1-based indexing
degrees = [0] * (n + 1)
```

- **Input Parsing**: The first line of the input contains three integers: `n` (number of nodes), `m` (number of edges), and `t` (the root node). These values are read using `map` and split into respective variables.
- **Adjacency List**: A list of lists `adj` is created to represent the tree, where each index corresponds to a node, and the sublist holds tuples of adjacent nodes and the edge weight. We initialize it with `n + 1` empty lists, as node indexing starts from 1.
- **Degree List**: An array `degrees` is initialized to store the degree (number of neighbors) of each node. This will help in identifying leaf nodes later.

2. Building the Tree from Edges

```python
# Read the edges and build the tree
for _ in range(m):
    u, v, w = map(int, input().split())
    adj[u].append((v, w))
    adj[v].append((u, w))
    degrees[u] += 1
    degrees[v] += 1
```

- **Edge Reading**: This part reads the edges from the input. Each edge is described by two nodes `u` and `v`, and the attack power `w` of the monster on the edge between them. For each edge:
  - The nodes `u` and `v` are connected bidirectionally, meaning `u` is connected to `v` and vice versa. This is stored as a tuple `(v, w)` in the adjacency list of `u` and `(u, w)` in the adjacency list of `v`.
- **Degree Update**: For each node `u` and `v`, we increment their degrees by 1, because both nodes are now connected by an edge.

3. Initialization for Depth-First Search (DFS)

```python
# Initialize necessary arrays
parent = [0] * (n + 1)
depth = [0] * (n + 1)
w_edge = [0] * (n + 1)
visited = [False] * (n + 1)
```

- **Parent Array**: This array `parent` will store the parent node of each node during the DFS traversal.
- **Depth Array**: The `depth` array will store the depth of each node from the root node `t`. Depth corresponds to how many edges the node is away from the root.
- **Edge Weight Array**: The `w_edge` array will store the weight (attack power) of the edge between each node and its parent during the DFS traversal.
- **Visited Array**: The `visited` array is used to mark whether a node has been visited during DFS.

4. Depth-First Search (DFS) Implementation

```
# Depth-First Search to compute parent, depth, and edge weights
def dfs(u, p):
    visited[u] = True
    for v, w in adj[u]:
        if not visited[v]:
            parent[v] = u
            depth[v] = depth[u] + 1
            w_edge[v] = w  # Weight of edge from v to parent[v]
            dfs(v, u)

dfs(t, 0)
```

- **DFS Traversal**: The function `dfs(u, p)` recursively visits all the nodes in the tree starting from node `u`, with `p` being the parent of `u`. During the traversal:
  - We mark the current node `u` as visited.
  - For each neighboring node `v`, if it has not been visited yet:
    - We set the parent of `v` to `u`.
    - We set the depth of `v` to `depth[u] + 1`, which means that `v` is one level deeper than `u`.
    - We store the attack power `w` of the edge between `v` and its parent `u` in `w_edge[v]`.
    - We recursively call DFS on node `v`.
- **Start DFS from the Root**: We call `dfs(t, 0)` to begin the DFS traversal from the root node `t`. The parent of the root is set to 0 (a placeholder).

5. Identifying Leaf Nodes

```
# Collect all leaf nodes except the root node t
leaf_nodes = []
for u in range(1, n + 1):
    if degrees[u] == 1 and u != t:
        leaf_nodes.append(u)
```

- **Leaf Node Identification**: This loop iterates over all nodes from 1 to `n`. If a node has a degree of 1 (i.e., it has only one neighbor) and is not the root node `t`, it is a leaf node. These leaf nodes are collected in the `leaf_nodes` list.

6. Calculating the Minimum Health Points

```
max_total_HP = 0  # Initialize the maximum HP required

# Compute the total HP required for each leaf node
for u in leaf_nodes:
    total_HP = 0
    k = depth[u]  # Starting MP for this path
    curr = u
    while curr != t:
        w = w_edge[curr]  # Weight of the edge from curr to parent[curr]
        HP_consumed = max(0, w - k)
        total_HP += HP_consumed
        k -= 1  # Decrease MP by 1 as we move to the next node
        curr = parent[curr]
    max_total_HP = max(max_total_HP, total_HP)
```

- **Total HP Calculation**: This part computes the total health points required for the warrior to safely travel from each leaf node to the root node `t` while ensuring that the magic points (MP) and health points (HP) both reach zero at the root:
  - For each leaf node `u`, we start from that leaf and move upwards to the root, calculating the health points consumed at each edge.
  - The starting MP for each path is the depth of the leaf node, as this represents the initial MP at that point.
  - For each edge the warrior crosses:
    - The attack power `w` of the monster on the edge is stored in `w_edge[curr]`.
    - The health points consumed is the difference `max(0, w - k)` where `k` is the current MP before crossing the edge.
    - The warrior's MP is reduced by 1 as they move up the tree (to the parent node).
    - The total HP consumed for the path is accumulated in `total_HP`.
- After calculating the HP for each leaf node, we keep track of the maximum `total_HP` among all leaf nodes, because the warrior needs enough HP to handle the worst-case scenario.

7. Output the Result

```
# Output the minimum health point required at the beginning
print(max_total_HP)
```

③Why is your solution better than others?

For this problem, the **DFS approach** is likely the most efficient and straightforward solution, as it directly exploits the tree structure and handles the health point calculation in a clean, recursive manner.

Q2 Find the Maximum Subinterval Sum with Unequal Shelf Contributions

①What are the possible solutions for the problem?

1. **Brute-force Solution:**
    - Check all possible subintervals of items, ensuring that each one satisfies the constraints.
    - This approach may be slow for larger inputs, as it tries every possible subinterval.
2. **Sliding Window with Optimized Checking:**
    - Use a sliding window approach combined with a hashmap or set to ensure unique item counts per shelf.
    - This is more efficient than brute force, as it reduces redundant checks.
3. **Circular Array Handling:**
    - Take advantage of the circular shelf arrangement by extending the shelf array to simulate a ring.
    - This avoids the complexity of handling wrap-around conditions manually, especially when checking for continuous subintervals.
4. **Dynamic Programming Approach (Advanced):**
    - This is an extension of the sliding window approach, where dynamic programming can be used to store previously computed results for certain subintervals.
    - This avoids recalculating the values and item counts for overlapping subintervals, improving performance.

②How do you solve this problem?

1. Reading Inputs

```
n, k, bag_size = map(int, input().split())
n = int(n)
k = int(k)
bag_size = int(bag_size)
```

Explanation:

- This part reads three integers: `n`, `k`, and `bag_size` from the user input.
    - `n`: The total number of items in the supermarket.
    - `k`: The number of shelves, where the items are distributed (shelves are arranged in a ring).
    - `bag_size`: The maximum number of items the user can take in a continuous subinterval.

2. Processing Item Data

```
# List to store item data and track maximum decimal places
items_data = []
max_decimal_places = 0
# Read items and determine the maximum number of decimal places
for _ in range(n):
    id_str, value_str = input().split()
    id = int(id_str)
    # Determine the number of decimal places
    if '.' in value_str:
        integer_part, decimal_part = value_str.split('.')
        num_decimal_places = len(decimal_part)
    else:
        integer_part = value_str
        decimal_part = ''
        num_decimal_places = 0
    # Update maximum decimal places
    if num_decimal_places > max_decimal_places:
        max_decimal_places = num_decimal_places
    # Store item data
    shelf_num = id % k
    items_data.append({
        'id': id,
        'value_str': value_str,
        'integer_part': integer_part,
        'decimal_part': decimal_part,
        'num_decimal_places': num_decimal_places,
        'shelf': shelf_num
    })
```

Explanation:

- The code reads the information for each item: its `id` and `value`.
    - For the value, the program checks if it contains a decimal part and calculates the number of decimal places (`num_decimal_places`).
    - It keeps track of the maximum decimal places seen in any item (`max_decimal_places`) to standardize the precision later.
    - It calculates which shelf an item belongs to using the hash function `id % k` and stores the item data in `items_data`.

3. Converting Item Values to Scaled Integers

```
# Convert item values to scaled integers
for item in items_data:
    # Pad decimal part with zeros to match maximum decimal places
    decimal_part_padded = item['decimal_part'].ljust(max_decimal_places, '0')
    value_int_str = item['integer_part'] + decimal_part_padded
    item['value_int'] = int(value_int_str)
```

Explanation:

- The program converts the item values into integers by padding the decimal part to match the maximum number of decimal places.
    - This ensures that floating-point precision issues don't affect the calculations later on.
    - It forms a new integer representation of the item value (`value_int`), which is used for accurate comparison during value summation.

4. Organizing Items by Shelf

```
# Assign items to shelves
shelves = [[] for _ in range(k)]
for item in items_data:
    shelves[item['shelf']].append(item)
# Sort items on each shelf in descending order of IDs
for shelf in shelves:
    shelf.sort(key=lambda x: -x['id'])
```

Explanation:

- The items are grouped into `k` shelves based on their `shelf` number (which is determined by `id % k`).
- Each shelf's items are sorted in **descending** order of their IDs. This ensures that the highest-ID items come first when considering the items to take.

5. Extending Items Array for Ring Structure

```
# Build the items array in order of shelves in the ring
items_array = []
shelf_indices = []
for i in range(k):
    shelf = shelves[i]
    for item in shelf:
        items_array.append(item)
        shelf_indices.append(i)
# Extend the items array to simulate the circular nature
items_array_extended = items_array + items_array
```

Explanation:

- The program creates a `items_array` by concatenating all items from all shelves, maintaining the shelf order.
- Then, it extends this array by duplicating it. This extension simulates the **ring structure** of the shelves (where the first and last shelves are connected), allowing for easier handling of subintervals that wrap around the end of the array.

6. Precomputing Shelf Information

```
max_total_value_int = 0  # Initialize maximum total value as integer
# Precompute whether shelves have items
shelf_has_items = [len(shelves[i]) > 0 for i in range(k)]
```

Explanation:

- `max_total_value_int` : Initializes a variable to track the maximum sum of item values that can be obtained under the given conditions (in integer form).
- `shelf_has_items` : A list of booleans indicating whether each shelf has items, which helps when checking for gaps (empty shelves) between selected items.

7. Helper Function for Formatting the Total Value

```
# Function to format the total value for output
def format_total_value(total_value_int, max_decimal_places):
    total_value_str = str(total_value_int)
    if max_decimal_places == 0:
        return total_value_str
    # Ensure the string has enough digits
    if len(total_value_str) <= max_decimal_places:
        total_value_str = total_value_str.rjust(max_decimal_places + 1, '0')
    integer_part = total_value_str[:-max_decimal_places]
    decimal_part = total_value_str[-max_decimal_places:]
    return integer_part + '.' + decimal_part
```

Explanation:

- This function converts the total value (stored as an integer) back to a string, formatted with the correct number of decimal places.
    - If no decimal places are needed (`max_decimal_places == 0`), it simply returns the integer as a string.
    - If decimals are required, it ensures that the integer value is padded with zeros as necessary and splits it into an integer and decimal part.

8. Finding the Maximum Total Value

```
# Main loop to find the maximum total value
for start in range(n):
    shelves_taken = {}
    total_items_taken = 0
    total_value_int = 0
    empty_shelf_skipped = False
    prev_shelf = None
    i = start

    while total_items_taken < bag_size and i < start + n:
        item = items_array_extended[i]
        shelf = item['shelf']

        # Check for empty shelves between the current and previous shelf
        if prev_shelf is not None:
            dist = (shelf - prev_shelf) % k
            if dist != 0:
                empty_shelves_between = 0
                for s in range(1, dist):
                    inter_shelf = (prev_shelf + s) % k
                    if not shelf_has_items[inter_shelf]:
                        empty_shelves_between += 1
                if empty_shelves_between > 1 or (empty_shelf_skipped and empty_shelves_between >= 1):
                    break  # Cannot skip more than one empty shelf
                if empty_shelves_between >= 1:
                    if empty_shelf_skipped:
                        break
                    empty_shelf_skipped = True
        prev_shelf = shelf

        shelves_taken[shelf] = shelves_taken.get(shelf, 0) + 1
        total_items_taken += 1
        total_value_int += item['value_int']
        i += 1

        # Check if the counts of items from each shelf are unique
        counts_list = list(shelves_taken.values())
        if len(counts_list) == len(set(counts_list)):
            if total_value_int > max_total_value_int:
                max_total_value_int = total_value_int
```

Explanation:

- This is the main logic of the program that iterates over all possible starting points (`start`) of a subinterval of items.
    - For each starting point, it selects items in a **continuous sequence** while respecting the constraints:
        - The total number of items selected must not exceed `bag_size` .
        - The number of items taken from each shelf must be **unique**.
        - If there are empty shelves between two selected shelves, it handles them based on the rules (only one empty shelf can be skipped, multiple empty shelves between selected shelves are not allowed).
    - If a valid subinterval is found, it updates the `max_total_value_int` if the total value of that subinterval is larger than the current maximum.

9. Output the Result

```
# Output the maximum total value formatted correctly
print(format_total_value(max_total_value_int, max_decimal_places))
```

③Why is your solution better than others?

My solution strikes a balance between **efficiency** and **correctness**:

- It uses a **sliding window** to quickly find valid subintervals while maintaining the constraints.
- It handles the **circular nature** of shelves and allows for skipping **empty shelves** in an optimized manner.
- By tracking **unique counts** and **maximizing value**, it guarantees an optimal solution in linear time, making it much more efficient than brute-force approaches or those that require recalculating values for each subinterval repeatedly.