Phân tích và Giải thuật bài toán Pokémon Arena

1 Phân tích bài toán

Bạn sở hữu n Pokémon, mỗi Pokémon có m thuộc tính và chi phí thuê c_i. Ban đầu Pokémon số 1 đứng trong đấu trường. Mục tiêu: đưa Pokémon số n vào đấu trường với chi phí nhỏ nhất.

Có hai loại thao tác:

- Tăng thuộc tính $a_{i,j}$ lên k với chi phí k.
- Thuê Pokémon i để đấu thuộc tính j với chi phí c_i, nếu $a_{i,j} \geq$ thuộc tính Pokémon đang đứng trong đấu trường thì thắng và thay thế vào sân.

Ta cần tối ưu chi phí để có thể chuyển từ Pokémon 1 sang Pokémon n qua chuỗi thao tác bất kỳ.

2 Mô tả thuật toán

Giải pháp dựa trên việc mô hình hóa bài toán dưới dạng bài toán đường đi ngắn nhất trong đồ thị có trọng số không âm.

Mỗi Pokémon được xem như một nút trong đồ thị. Tuy nhiên, thao tác tăng thuộc tính khiến rằng mỗi thuộc tính j cần được mô phỏng bằng **các nút ảo** thể hiện trạng thái "đã tăng đủ để vượt qua các Pokémon khác theo thuộc tính đó".

Thuật toán xây dựng:

- Với mỗi thuộc tính j, ta sắp xếp tất cả Pokémon theo $a_{i,j}$.
- Với mỗi Pokémon theo thứ tự tăng dần, sinh ra:
  - Nút ảo đi lên (up-node) mô phỏng khả năng tiến tới đối thủ mạnh hơn mà không tốn chi phí tăng.
  - Nút ảo đi xuống (down-node) mô phỏng khả năng tăng điểm thuộc tính với chi phí đúng bằng lượng chênh lệch.
- Mỗi node ảo liên kết với node Pokémon tương ứng với chi phí thuê c_i.

Sau khi xây dựng xong toàn bộ đồ thị, chạy Dijkstra từ nút đại diện Pokémon 1 và tìm đường đi ngắn nhất đến Pokémon n.

3 Phương pháp thiết kế

Giải pháp sử dụng:

- **Mô hình hóa bài toán → đồ thị có trọng số**.
- **Dijkstra** cho đường đi ngắn nhất với trọng số không âm.
- **Tạo các lớp node ảo** cho từng thuộc tính để mô phỏng đầy đủ hai loại thao tác.

4 Tính phù hợp của phương pháp

- Hai thao tác trong đề đều có chi phí không âm → phù hợp tuyệt đối với Dijkstra.
- Bài toán yêu cầu tìm chi phí tối ưu để chuyển từ trạng thái ban đầu sang trạng thái đích → bản chất là bài toán đường đi tối ưu.
- Các thuộc tính nhiều chiều (m thuộc tính) khiến việc mô phỏng trực tiếp trở nên phức tạp; tạo node ảo theo từng thuộc tính cho phép biến đổi bài toán thành dạng tuyến tính dễ xử lý.
- Tổng số node ảo là $O(nm)$, nằm trong giới hạn đề (tổng $nm \leq 4 \cdot 10^5$).

Do đó phương pháp Dijkstra trên đồ thị mở rộng hoàn toàn phù hợp và hiệu quả.

5 Phân tích độ phức tạp

5.1 Độ phức tạp thời gian

- Mỗi thuộc tính tạo ra $2n$ node ảo và các cạnh tương ứng.
- Tổng node $= n + 2nm \leq 8 \cdot 10^5$.
- Tổng số cạnh cũng tỷ lệ $O(nm)$.
- Chạy Dijkstra có độ phức tạp:

$$O(E \log V) = O(nm \log(nm))$$

Vì tổng nm tối đa trên toàn bộ test là $4 \cdot 10^5$, nên thời gian vẫn bảo đảm chạy tốt.

5.2 Độ phức tạp không gian

- Lưu node thật: $O(n)$.
- Lưu node ảo: $O(nm)$.
- Lưu cạnh: $O(nm)$.
- Mảng khoảng cách cho Dijkstra: $O(nm)$.

Tổng quan:

$$S(n, m) = O(nm)$$

6 Code Python chuẩn

```
import sys
import heapq


sys.setrecursionlimit(200000)

def solve():

    def input_generator():

        for line in sys.stdin:
            for token in line.split():
                yield token


    iterator = input_generator()

    try:

        t_str = next(iterator)
        t = int(t_str)
    except StopIteration:
        return

    results = []

    for _ in range(t):
        try:
            n = int(next(iterator))
            m = int(next(iterator))

            c = []
            for _ in range(n):
                c.append(int(next(iterator)))

            a = []
            for _ in range(n):
                row = []
                for _ in range(m):
                    row.append(int(next(iterator)))
                a.append(row)

        except StopIteration:
            break


        total_nodes = n + 2 * n * m
        adj = [[] for _ in range(total_nodes)]
        current_virtual_id = n
```

```
        for j in range(m):
            sorted_pokes = []
            for i in range(n):
                sorted_pokes.append((a[i][j], i))
            sorted_pokes.sort()

            up_start = current_virtual_id
            down_start = current_virtual_id + n
            current_virtual_id += 2 * n

            for k in range(n):
                val, u = sorted_pokes[k]
                up_node = up_start + k
                down_node = down_start + k

                if k + 1 < n:
                    adj[up_node].append((up_node + 1, 0))
                adj[u].append((up_node, 0))
                adj[up_node].append((u, c[u]))

                if k > 0:
                    prev_val = sorted_pokes[k-1][0]
                    diff = val - prev_val
                    adj[down_node].append((down_node - 1, diff))
                adj[u].append((down_node, 0))
                adj[down_node].append((u, c[u]))


        INF = 10**18
        dist = [INF] * total_nodes
        dist[0] = 0
        pq = [(0, 0)]
        min_cost = -1

        while pq:
            d, u = heapq.heappop(pq)
            if d > dist[u]: continue
            if u == n - 1:
                min_cost = d
                break
            for v, w in adj[u]:
                if dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    heapq.heappush(pq, (dist[v], v))


        print(min_cost)

if __name__ == '__main__':
    solve()
```