ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Lớp: CS112.Q11.KHTN

Môn học: Phân Tích Và Thiết Kế Thuật Toán

Luyện Tập Thiết Kế Thuật Toán
Nhóm 7

Nhóm 4:
Lê Văn Thức
Bảo Quý Định Tân

Giảng viên:
Nguyễn Thanh Sơn

Lời giải chi tiết cho bài toán *Pokémon*

1. Mô hình hóa bài toán

Gọi N là số lượng Pokémon và M là số lượng thuộc tính. Mục tiêu là tìm chi phí nhỏ nhất để chuyển từ trạng thái Pokémon 1 đang đứng sân sang Pokémon N.

Chi phí khi thay Pokémon u bằng v thông qua thuộc tính j được xác định bởi:

$$\text{Cost}(u \to v) = c_v + \max(0,\, a_{u,j} - a_{v,j}).$$

Trông ý tưởng khá giống đi từ u đến v bằng thuộc tính j trên đồ thị với chi phí cạnh là $\max(0, a_{u,j} - a_{v,j})$.

Nên ta xây dựng một **đồ thị có hướng có trọng số** $G = (V, E)$ để mô phỏng quá trình chuyển đổi giải quyết bài toán này là tìm đường đi ngắn nhất từ đỉnh **pokemon 1** đến đỉnh **pokemon n**, phương pháp này có thể gọi là **Transform and conquer** (Representation Change), ta sử dụng phương pháp này bởi vị nó giúp ta đơn giản hoá bài toán và dễ dàng giải quyết về sau hơn.

Chúng ta vẫn giải bài toán "Tìm đường đi ngắn nhất"(bài toán không đổi -> không phải Problem Reduction). Dữ liệu đầu vào không bị cắt bớt hay đơn giản hóa theo kiểu chia nhỏ (không phải Instance Simplification).

Và cuối cùng ta sử dụng thuật toán **Dijkstra** được xây dựng trên phương pháp tham lam (tối ưu cục bộ sẽ dẫn đến toán cục).

Chứng minh chi phí cạnh

Ta dễ dàng nhận xét được rằng nếu ở u ta chưa từng sử dụng thao tác loại một để tăng giá trị của thuộc tính nào lên hết thì công thức trên kia đúng. Bởi vì nếu $a_{v,j} \geq a_{u,j}$ thì $cost = 0$, ngược lại ta chỉ cần thực hiện thao tác một để tăng $a_{v,j}$ lên vừa đủ bằng $a_{u,j}$ tốn mất $k = a_{u,j} - a_{v,j}$ để v đánh bại u.

Nhưng nếu trước đó ở u ta có sử dụng thao tác một để tăng giá trị của thuộc tính j nào đó lên để đánh bại một Pokemon x trước đó nữa, thì ta đã tốn $k = a_{u,j} - a_{x,j}$ để đưa a_{u_j} lên bằng $a_{x,j}$ thì theo công thức trên ta sẽ không biết được điều này mà vẫn cho rằng $a_{u,j}$ vẫn chưa thay đổi, thì ta có thể nghĩ việc này sẽ làm sai đáp án nhưng lại **không**:

- **Nếu** $a_{v,j} < a_{u,j}$: tổng chi phí sẽ là $a_{u,j} - a_{v,j}$ cộng thêm chi phí hồi trước là $a_{x,j} - a_{u,j} = a_{u,j} - a_{v,j} + a_{x,j} - a_{u,j} = a_{x,j} - a_{v,j}$ thì không khác gì ta đi từ x đến v mà không qua u, nên vẫn hợp lý không ảnh hưởng sai đáp án.

- **Nếu** $a_{v,j} \geq a_{u,j}$: tổng chi phí sẽ là $a_{x,j} - a_{u,j}$ thôi nhưng ta lại thấy rằng bây giờ $a_{v,j} \geq a_{u,j} \Leftrightarrow -a_{v,j} \leq -a_{u,j} \Leftrightarrow a_{x,j} - a_{v,j} \leq a_{x,j} - a_{u,j}$. Hay có nghĩa là đi trực tiếp từ x đến v sẽ lời hơn, nên từ đầu ta có thuật toán đi tối ưu thì sẽ tự triệt tiêu đi trường hợp này nên vẫn hợp lý và không ảnh hưởng sai đáp án.

- **Nếu** $a_{v,j} \geq a_{x,j}$: trường hợp này thì quá rõ rồi, ta đi trực tiếp từ x đến v luôn lời hơn nên vẫn không ảnh hưởng đáp án.

Vậy từ đây ta có thể thấy rằng công thức cạnh để đi bằng một thuộc tính như trên là hợp lý để ta xây dựng đồ thị.

Tập đỉnh

Đồ thị gồm hai loại đỉnh:

- **Đỉnh Pokémon**: $P_1, P_2, \ldots, P_N$.
- **Đỉnh thuộc tính ảo**: Với mỗi thuộc tính $j \in [1, M]$, lấy dãy

$$(a_{1,j}, a_{2,j}, \ldots, a_{N,j})$$

sắp xếp tăng dần thành:

$$v_{j,1} \leq v_{j,2} \leq \cdots \leq v_{j,N}.$$

Tạo N đỉnh tương ứng $T_{j,1}, T_{j,2}, \ldots, T_{j,N}$.

Tổng số đỉnh là:

$$|V| = N + NM.$$

Tập cạnh và trọng số

Ta thêm các cạnh sau:

1. **Cạnh vào:** từ Pokémon u tới đỉnh thuộc tính ứng với giá trị của nó:

$$P_u \to T_{j,k} \quad \text{nếu } v_{j,k} = a_{u,j}, \qquad w = 0.$$

và với mỗi P_u chỉ nối đến một đỉnh đến duy nhất (và đỉnh kia cũng phải chưa có $P_{u'}$ nào nối đến. Có ý nghĩa là ta chọn pokemon u đang đứng trên sàn đấu và sẽ bị đánh.

2. **Cạnh ra:** từ đỉnh thuộc tính đến Pokémon v:

$$T_{j,k} \to P_v \quad \text{nếu } v_{j,k} = a_{v,j}, \qquad w = c_v.$$

và với mỗi $T_{j,k}$ chỉ nối đến một đỉnh đến duy nhất (và đỉnh kia cũng phải chưa có $T_{j,k'}$ nào nối đến. Việc đi ra này giống như cho v đánh u hay ta thực hiện thao tác loại hai nên chi phí sẽ như trên là $w = c_v$.

3. **Cạnh chuyển tiếp cùng thuộc tính:**

- *Đi lên:*

$$T_{j,k} \to T_{j,k+1}, \qquad w = 0.$$

- *Đi xuống:*

$$T_{j,k+1} \to T_{j,k}, \qquad w = v_{j,k+1} - v_{j,k}.$$

Các cạnh này mô phỏng chính xác công thức:

$$\max(0,\, a_{u,j} - a_{v,j}).$$

Việc đi lên dĩ nhiên là không tốn gì, tương đương với việc có giá trị cao hơn sẽ đánh được luôn mà không cần sử dụng tới thao tác loại một. Còn việc đi xuống sẽ tương đương với sử dụng thao tác loại một để nâng giá trị lên rồi mới đánh nên mỗi lần đi xuống ta lại cộng thêm chi phí còn thiếu để đánh được.

2. Thuật toán

Bài toán trở thành tìm đường đi ngắn nhất từ P_1 đến P_N trên đồ thị G.

1. **Tiền xử lý:** Với mỗi thuộc tính j, tạo danh sách các cặp $(a_{i,j}, i)$ và sắp xếp tăng dần để phục vụ việc tạo đồ thị.
2. **Dựng đồ thị:** Tạo các cạnh theo mô tả ở mục trước.
3. **Chạy Dijkstra:** Khởi tạo:

$$d[u] = \infty, \quad d[P_1] = 0.$$

   Dùng hàng đợi ưu tiên để tìm $d[P_N]$.

Kết quả:

$$d[P_N] = \text{chi phí nhỏ nhất.}$$

3. Phân tích độ phức tạp

Gọi $K = N \cdot M$ là tổng số giá trị thuộc tính trong ma trận A. Ta giả sử các phép toán sơ cấp (so sánh số, gán, đọc/ghi ô mảng) có chi phí $O(1)$.

Bước tiền xử lý & sắp xếp để tạo cạnh

Mục tiêu. Với mỗi thuộc tính j (một cột của A) ta cần:

1. thu thập danh sách các cặp $(a_{i,j}, i)$ cho $i = 1 \dots N$,
2. sắp xếp danh sách này theo giá trị $a_{i,j}$,
3. xác định vị trí (chỉ số k) của mỗi $a_{i,j}$ trong dãy đã sắp xếp để biết đỉnh ảo tương ứng.

Phân tích.

- Việc thu thập các cặp cho tất cả các cột xử lý tổng cộng K phần tử: chi phí $O(K)$.
- Với một cột có N phần tử, sắp xếp tốn $O(N \log N)$. Với M cột tổng là

$$\sum_{j=1}^{M} O(N \log N) = O(M \cdot N \log N) = O(K \log N).$$

- Gán chỉ số vị trí (ví dụ: duyệt một lần danh sách đã sắp xếp và lưu bản đồ từ chỉ số Pokémon sang chỉ số đỉnh ảo) tốn $O(N)$ cho mỗi cột, tổng $O(K)$.

Kết luận bước sắp xếp.

$$T_{\text{sort}} = O(K \log N).$$

Bước dựng đồ thị (xây dựng danh sách kề)

Mục tiêu. Tạo các cạnh loại:

- *Cạnh vào:* cho mỗi cặp (i, j) một cạnh $P_i \to T_{j,k}$ (trọng số 0) — tổng K cạnh.
- *Cạnh ra:* cho mỗi cặp (i, j) một cạnh $T_{j,k} \to P_i$ (trọng số c_i) — tổng K cạnh.
- *Cạnh chuỗi trong mỗi thuộc tính:* với mỗi thuộc tính j có $N-1$ cạnh đi lên và $N-1$ cạnh đi xuống, tổng khoảng $2(N-1)M \approx 2K$ cạnh.

Đếm đỉnh và cạnh.

$$|V| = N + NM = N + K, \qquad |E| \approx K + K + 2K = 4K.$$

Phân tích.

- Mỗi cạnh được thêm vào danh sách kề bằng một thao tác $O(1)$ (append). Vì vậy dựng đồ thị tốn $O(|E|) = O(K)$.

Kết luận bước dựng đồ thị.

$$T_{\text{build}} = O(K), \qquad |E| = \Theta(K).$$

Bước chạy Dijkstra (tìm đường ngắn nhất)

Thiết lập. Ta chạy Dijkstra trên đồ thị có $|V|$ đỉnh và $|E|$ cạnh; cài đặt chuẩn sử dụng hàng đợi ưu tiên (binary heap).

Số thao tác heap (ước lượng).

- Mỗi đỉnh khi được khám lần đầu sẽ bị push vào heap; tuy nhiên do cơ chế relax, số lần push có thể lên đến $O(|E|)$ trong trường hợp tệ nhất (mỗi cạnh gây một lần relax thành công và do đó một lần đẩy mới).
- Mỗi lần pop (lấy đỉnh có khoảng cách nhỏ nhất) xảy ra tối đa $O(|V|)$ lần (một pop hợp lệ cho mỗi đỉnh); nếu có các bản sao cũ trong heap thì số pop có thể lớn hơn nhưng vẫn bị giới hạn bởi số push, do đó tổng số pop là $O(|V|+(\text{số push không hợp lệ})) = O(|E|)$ trong phân tích thô.

Độ phức tạp thời gian của các thao tác heap. Với binary heap, mỗi thao tác push/pop tốn $O(\log |V|)$. Vì vậy tổng chi phí cho heap là

$$O\big((\text{push} + \text{pop}) \cdot \log |V|\big) = O\big((|E| + |V|) \log |V|\big).$$

Kết hợp với việc duyệt các cạnh để thử relax (chi phí tuyến tính $O(|E|)$), ta có

$$T_{\text{Dijkstra}} = O\big((|E| + |V|) \log |V|\big).$$

Thay số xấp xỉ. Vì $|E| = \Theta(K)$ và $|V| = N + K = \Theta(K)$ (vì $K = N \cdot M$ lớn hơn hoặc bằng N trong các trường hợp thực tế của bài), ta có thể viết gọn:

$$T_{\text{Dijkstra}} = O(K \log K).$$

3.1. Tổng hợp thời gian

$$T_{\text{total}} = T_{\text{sort}} + T_{\text{build}} + T_{\text{Dijkstra}} = O(K \log N) + O(K) + O(K \log K).$$

Vì $\log K \geq \log N$, hạng lớn nhất là $O(K \log K)$. Do đó:

$$\boxed{T(N, M) = O(K \log K).}$$

Ước lượng số cho $K = 4 \cdot 10^5$.

- $|E| \approx 4K \approx 1.6 \times 10^6$ cạnh.
- $|V| \approx K \approx 4 \times 10^5$ đỉnh, nên $\log_2 |V| \approx 19$.
- Số thao tác heap nghiệm cho Dijkstra xấp xỉ $(|E| + |V|) \log |V| \approx (1.6 \times 10^6 + 4 \times 10^5) \cdot 19 \approx 3.8 \times 10^7$ phép toán log-heap (thao tác so sánh/hoán vị trong heap).

3.2. Phân tích bộ nhớ (space)

Yêu cầu lưu trữ chính.

- **Danh sách kề (adjacency list):** lưu cho mỗi cạnh một cặp $(\text{to}, \text{weight})$. Với $|E| \approx 4K$, chiếm $O(K)$ ô lưu.
- **,**Mảng khoảng cách dist$[\cdot]$: một số thực hoặc số nguyên cho mỗi đỉnh, chiếm $O(|V|) = O(K)$.
- **Bảng trạng thái/visited và các mảng phụ (parent, index mapping)**: tổng $O(K)$.
- **Priority queue:** trong tệ nhất có thể chứa $O(|E|)$ phần tử (nhiều bản sao), do đó bộ nhớ tạm cho heap là $O(K)$.

Kết luận về bộ nhớ.

$$\boxed{S(N, M) = O(K).}$$

4. Kết luận

Bằng việc sử dụng **đồ thị phụ** gồm các đỉnh thuộc tính ảo, ta giảm số cạnh từ $O(N^2 M)$ xuống $O(NM)$, cho phép giải bài toán trong thời gian khả thi. Thuật toán chiếm thời gian $O(K \log K)$ và bộ nhớ $O(K)$, phù hợp với các ràng buộc của đề bài.

5. Code python

```python
import sys
import heapq

input = sys.stdin.read

def solve():
    data = input().split()
    if not data:
        return
```

```
    iterator = iter(data)
    try:
        num_test_cases = int(next(iterator))
    except StopIteration:
        return

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

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

            # a[i][j]
            raw_a = []
            for _ in range(n * m):
                raw_a.append(int(next(iterator)))

        except StopIteration:
            break

        adj_head = [[] for _ in range(n + n * m)]
        adj_weight = [[] for _ in range(n + n * m)]

        node_counter = n

        for j in range(m):
            col_vals = []
            for i in range(n):
                val = raw_a[i * m + j]
                col_vals.append((val, i))

            col_vals.sort(key=lambda x: x[0])

            start_node = node_counter

            for k in range(n):
                val, pokemon_idx = col_vals[k]
                u_virtual = start_node + k

                adj_head[pokemon_idx].append(u_virtual)
                adj_weight[pokemon_idx].append(0)

                adj_head[u_virtual].append(pokemon_idx)
                adj_weight[u_virtual].append(c[pokemon_idx])

                if k > 0:
                    prev_virtual = start_node + k - 1
                    prev_val = col_vals[k-1][0]

                    adj_head[prev_virtual].append(u_virtual)
                    adj_weight[prev_virtual].append(0)

                    diff = val - prev_val
                    adj_head[u_virtual].append(prev_virtual)
```

```
                    adj_weight[u_virtual].append(diff)

            node_counter += n

        # Dijkstra
        dist = [-1] * node_counter
        pq = [(0, 0)]
        dist[0] = 0

        target = n - 1

        min_cost = -1

        while pq:
            d, u = heapq.heappop(pq)

            if d > dist[u] and dist[u] != -1:
                continue

            if u == target:
                min_cost = d
                break

            neighbors = adj_head[u]
            weights = adj_weight[u]

            for i in range(len(neighbors)):
                v = neighbors[i]
                w = weights[i]
                new_dist = d + w

                if dist[v] == -1 or new_dist < dist[v]:
                    dist[v] = new_dist
                    heapq.heappush(pq, (new_dist, v))

        print(min_cost)

if __name__ == '__main__':
    sys.setrecursionlimit(200000)
    solve()
```