Báo Cáo CS112

1 Phát biểu bài toán

Bài toán yêu cầu tìm chi phí nhỏ nhất để đưa Pokémon thứ n ra sân đấu, với trạng thái ban đầu chỉ có Pokémon thứ 1 đứng sân. Mỗi Pokémon có m thuộc tính, ký hiệu $a_{i,j}$.

Để thay đổi Pokémon trên sân, ta có thể:

- **Thuê Pokémon** i để đấu với Pokémon hiện tại dựa trên thuộc tính j. Pokémon i thắng nếu:
$$a_{i,j} \geq a_{\text{current},j}.$$
- **Tăng chỉ số thuộc tính** bất kỳ với chi phí k để đảm bảo điều kiện thắng khi cần.

Mục tiêu: tìm tổng chi phí nhỏ nhất để cuối cùng Pokémon số n đứng trên sân.

2 Chiến lược phát triển thuật toán

Đây là bài toán tìm **chuỗi thao tác tối ưu**, thuộc nhóm bài toán đường đi ngắn nhất có trạng thái. Ta phát triển thuật toán qua các bước:

- **Cách tiếp cận ngây thơ:** So sánh trực tiếp mọi cặp Pokémon theo mọi thuộc tính dẫn tới $O(n^2m)$, không thể dùng vì không gian thao tác quá lớn.
- **Transform and Conquer:** Thay vì xét quan hệ trực tiếp $n \times n$, ta chuyển bài toán thành việc xây dựng một *đồ thị trạng thái* dựa trên việc **sắp xếp** giá trị thuộc tính. Điều này giảm số cạnh từ $O(n^2)$ xuống còn $O(nm)$.

3 Mô hình hoá và thiết kế chi tiết

3.1 Ý tưởng chính

Khó khăn lớn nhất nằm ở thao tác *tăng thuộc tính* để chiến thắng. Thay vì trực tiếp tính chi phí tăng, ta mô hình hoá quá trình này thành các cạnh của đồ thị.

Ta xây dựng một đồ thị có các node đặc biệt:

- **Pos(u):** Pokémon u đang đứng trên sân.
- **State(u, t):** trạng thái trung gian xem xét Pokémon u dưới thuộc tính t.

3.2 Sorting theo từng thuộc tính

Với mỗi thuộc tính t, ta sắp xếp n Pokémon theo $a_{i,t}$. Điều này tạo ra một chuỗi xếp hạng từ nhỏ đến lớn, từ đó ta có thể mô hình hóa việc tăng chỉ số bằng các cạnh giữa những phần tử liền kề.

3.3 Thiết kế đồ thị

Dựa trên mã nguồn cài đặt, ta xây dựng các loại cạnh sau:

1. **Upgrade Edge (nâng thuộc tính):**
   Với thuộc tính t, xét Pokémon u ở hạng r và v ở hạng $r+1$:
   - $v \to u$: chi phí 0 (Pokémon mạnh hơn đánh xuống).
   - $u \to v$: chi phí $a_{v,t} - a_{u,t}$ (chi phí nâng để theo kịp).
2. **Selection Edge (chọn thuộc tính):**
   $$\text{Pos}(u) \to \text{State}(u,t) \quad \text{(chi phí 0)}$$
3. **Battle Edge (thuê và đấu):**
   $$\text{State}(u,t) \to \text{Pos}(u) \quad \text{(chi phí } c_u\text{)}$$

3.4 Thuật toán Conquer: Dijkstra

Sau khi xây dựng xong đồ thị, ta chạy thuật toán Dijkstra:

- Khởi tạo nút nguồn S nối với tất cả $\text{State}(1,t)$.
- Sử dụng Priority Queue để mở rộng trạng thái có chi phí nhỏ nhất.
- Đáp án cuối cùng:
  $$\min_t \Big(\text{dist}(\text{State}(n,t)) + c_n\Big).$$

4 Phân tích độ phức tạp

4.1 Độ phức tạp thời gian

- **Sorting:** Sắp xếp n Pokémon theo từng thuộc tính:
  $$O(m \cdot n \log n).$$
- **Dijkstra:**
  $$V = n \cdot m, \quad E \approx 4nm.$$
  $$O(E \log V) = O(nm \log(nm)).$$

Với ràng buộc $nm \le 4 \cdot 10^5$, thuật toán chạy tốt trong thời gian cho phép.

4.2 Độ phức tạp không gian

$$O(V+E) = O(nm)$$

Bao gồm lưu trữ đồ thị, các bảng thứ hạng và mảng khoảng cách. Dung lượng này hoàn toàn phù hợp với giới hạn 512MB.

5 Lý do lựa chọn phương pháp

1. **Tính đúng đắn:** Mọi thao tác nâng chỉ số và thuê Pokémon đều được mô hình hoá chính xác bằng cạnh đồ thị. Dijkstra đảm bảo tìm được chi phí thấp nhất toàn cục.
2. **Tối ưu hóa:** Nhờ sorting, quan hệ so sánh không còn là $O(n^2)$ mà giảm thành $O(n)$ trên từng thuộc tính.
3. **Hiệu quả thực tế:** Đồ thị dù lớn nhưng vẫn xử lý được nhờ số nút và cạnh tuyến tính theo nm, phù hợp với bộ nhớ và thời gian.

6 Cài đặt Python

```
import sys
import heapq

input = sys.stdin.readline

INF = 10**30

def solve():
    T = int(input())
    for _ in range(T):
        n, m = map(int, input().split())
        c = [0] + list(map(int, input().split()))

        a = [[0]*(m+1) for _ in range(n+1)]
        b = [[] for _ in range(m+1)]

        for i in range(1, n+1):
            row = list(map(int, input().split()))
            for j in range(1, m+1):
                a[i][j] = row[j-1]
                b[j].append((row[j-1], i))

        rank = [[0]*(m+1) for _ in range(n+1)]
        dec  = [[0]*(n+1) for _ in range(m+1)]

        for j in range(1, m+1):
            arr = b[j]
            arr.sort()
            for r, (val, pid) in enumerate(arr, start=1):
                rank[pid][j] = r
                dec[j][r] = pid

        def State(x, t):
            return (x-1)*m + t

        def Pos(x):
            return n*m + x
```

```
STATE_CNT = n*m
S = STATE_CNT + n + 1
N_total = S

adj = [[] for _ in range(N_total + 1)]

def add_edge(u, v, w):
    adj[u].append((v, w))

for t in range(1, m+1):
    for r in range(1, n+1):
        x = dec[t][r]
        u = State(x, t)
        if r < n:
            add_edge(u, State(dec[t][r+1], t), 0)
        if r > 1:
            z = dec[t][r-1]
            add_edge(u, State(z, t), a[x][t] - a[z][t])

for x in range(1, n+1):
    px = Pos(x)
    base = (x-1)*m
    for t in range(1, m+1):
        add_edge(base + t, px, 0)

for x in range(1, n+1):
    px = Pos(x)
    base = (x-1)*m
    for t in range(1, m+1):
        add_edge(px, base + t, c[x])

for t in range(1, m+1):
    add_edge(S, State(1, t), 0)

dist = [INF] * (N_total + 1)
dist[S] = 0
pq = [(0, S)]

while pq:
    d, u = heapq.heappop(pq)
    if d != dist[u]:
        continue
    for v, w in adj[u]:
        nd = d + w
        if nd < dist[v]:
            dist[v] = nd
            heapq.heappush(pq, (nd, v))

ans = INF
base = (n-1)*m
for t in range(1, m+1):
```

```
            v = base + t
            if dist[v] < INF:
                ans = min(ans, dist[v] + c[n])

        print(ans)

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