Lời giải bài toán: Pokémon

Nhóm 2: Nguyễn Trường Giang – Nguyễn Ngọc Hưng

Ngày 3 tháng 12 năm 2025

1 Mô tả bài toán

Có n Pokémon, Pokémon thứ i có:

- chi phí thuê c_i;
- m chỉ số (stats) $a_{i,1}, a_{i,2}, \ldots, a_{i,m}$.

Ta xuất phát từ Pokémon 1 và muốn đến được Pokémon n. Khi đang ở một Pokémon x, ta có thể "gọi" Pokémon i ra chiến đấu bằng chỉ số j:

- Nếu chỉ số $a_{i,j}$ chưa đủ lớn, ta có thể tăng chỉ số này lên cho phù hợp.
- Cụ thể, khi so sánh với Pokémon hiện tại x, ta phải đảm bảo chỉ số của Pokémon mới không kém hơn về chỉ số j. Điều này dẫn tới chi phí tăng chỉ số.
- Mỗi lần gọi Pokémon i, ta phải trả chi phí thuê c_i.

Chi phí khi đi từ Pokémon x sang Pokémon i bằng chỉ số j có dạng:

$$\text{cost}(x \to i, j) = c_i + \max(0,\, a_{x,j} - a_{i,j}).$$

Yêu cầu: với mỗi test, hãy tìm chi phí nhỏ nhất để đi từ Pokémon 1 đến Pokémon n với các thao tác như trên.

2 Phương pháp thiết kế thuật toán

2.1 Mô hình hoá bài toán bằng đồ thị

Ta mô hình hoá bài toán thành một **đồ thị có trọng số** rồi tìm **đường đi ngắn nhất** trên đồ thị đó.

- Mỗi Pokémon i tương ứng với một **đỉnh chính** (node chính) có số hiệu i.
- Với mỗi cặp (i, j) (Pokémon i, chỉ số j), ta tạo thêm một **đỉnh phụ** ký hiệu là (i, j). Trong cài đặt, ta đánh số đỉnh phụ là

$$id = i + j \cdot n.$$

- Các đỉnh phụ này dùng để tách chi phí thuê Pokémon và mô phỏng việc tăng/giảm chỉ số theo từng cột j.

Cách dựng cạnh:

1. Tách chi phí thuê Pokémon.
Với mỗi Pokémon i và chỉ số j, ta thêm hai cạnh:

$$i \to (i,j) \text{ với chi phí } 0, \quad (i,j) \to i \text{ với chi phí } c_i.$$

Ý nghĩa:

- cạnh $i \to (i,j)$: từ trạng thái "đang dùng Pokémon i" chọn "dùng Pokémon i với chỉ số j" mà không tốn thêm chi phí;
- cạnh $(i,j) \to i$: sau khi đã điều chỉnh chỉ số j phù hợp, ta "khóa" lựa chọn Pokémon i và trả chi phí thuê c_i.

2. Chuỗi hoá theo từng chỉ số j.
Với mỗi chỉ số j:

1. Ta xét các cặp $(a_{i,j}, id)$ với $id = i + jn$ và sắp xếp tăng dần theo $a_{i,j}$.
2. Giả sử sau khi sắp xếp, ta được dãy

$$(v_1, val_1), (v_2, val_2), \ldots, (v_n, val_n)$$

với $val_1 \le val_2 \le \cdots \le val_n$.

3. Ta thêm các cạnh:
   - $v_i \to v_{i+1}$ với chi phí 0 (đi từ stat nhỏ lên stat lớn không tốn gì);
   - $v_{i+1} \to v_i$ với chi phí $val_{i+1} - val_i$ (đi ngược lại phải trả đúng phần chênh lệch).

Với cách này, nếu đang ở đỉnh phụ (x,j) và muốn đến (i,j):

- nếu $a_{x,j} \le a_{i,j}$, ta đi xuôi trên chuỗi, toàn cạnh chi phí 0;
- nếu $a_{x,j} > a_{i,j}$, ta đi ngược, tổng chi phí của các cạnh ngược chính là $a_{x,j} - a_{i,j}$.

Khi kết hợp với cạnh $(i,j) \to i$ có chi phí c_i, ta thấy chi phí đường đi

$$x \to (x,j) \to (i,j) \to i$$

bằng đúng

$$c_i + \max(0, a_{x,j} - a_{i,j}),$$

trùng với công thức chi phí của đề bài.

2.2 Thuật toán trên đồ thị (Dijkstra)

Sau khi dựng xong đồ thị:

1. Số đỉnh tổng cộng là $N = (m+1)n$ (gồm n đỉnh chính và mn đỉnh phụ).
2. Ta sử dụng **thuật toán Dijkstra** trên đồ thị có trọng số dương, xuất phát từ đỉnh 1 (Pokémon đầu tiên).
3. Giá trị cần tìm là khoảng cách nhỏ nhất từ đỉnh 1 đến đỉnh n.

Thuật toán tổng quát:

- Với mỗi test:
  1. Đọc n, m, các giá trị c_i và ma trận $a_{i,j}$.
  2. Dựng đồ thị với các cạnh:
     - $(i \leftrightarrow (i,j))$ để tách chi phí thuê,
     - các cạnh trên chuỗi cho từng cột j.
  3. Chạy Dijkstra từ đỉnh 1.
  4. In ra kết quả là khoảng cách dist$[n]$.

3 Lý do phương pháp phù hợp

Tránh duyệt thô/ quy hoạch động tốn kém

Một cách nghĩ ngây thơ là: từ mỗi Pokémon, ta thử tất cả Pokémon khác và tất cả chỉ số, tính trực tiếp chi phí

$$c_i + \max(0, a_{x,j} - a_{i,j})$$

để đi tới đó. Cách làm này dẫn tới độ phức tạp $O(n^2 m)$, không phù hợp với ràng buộc $\sum n \cdot m \leq 4 \cdot 10^5$.

Quy hoạch động dạng "đang ở Pokémon i với vector chỉ số hiện tại" cũng không khả thi vì trạng thái sẽ nổ số chiều (mỗi chỉ số có thể rất lớn).

Ưu điểm của mô hình đồ thị + Dijkstra

Phương pháp hiện tại có những điểm mạnh:

- **Mô hình hoá đúng bản chất bài toán**: Mỗi thao tác "gọi Pokémon mới bằng chỉ số j" tương ứng với một đường đi cụ thể trong đồ thị, với chi phí đúng bằng chi phí mà đề yêu cầu. Không bỏ sót và không đếm trùng.
- **Giảm số cạnh nhờ "chuỗi hoá" theo chỉ số**: Thay vì tạo cạnh trực tiếp giữa mọi cặp Pokémon theo từng chỉ số ($O(n^2 m)$ cạnh), ta sắp xếp theo từng cột j và chỉ nối các Pokémon lân cận trong thứ tự sắp xếp. Từ đó, bất kỳ Pokémon nào cũng có thể đến Pokémon khác trong cùng cột j thông qua một đường đi với chi phí đúng bằng hiệu chỉ số, mà tổng số cạnh chỉ là $O(nm)$.

- **Áp dụng Dijkstra một cách tự nhiên**: Tất cả trọng số cạnh đều không âm (chi phí thuê c_i và chênh lệch chỉ số đều ≥ 0), nên Dijkstra là thuật toán chuẩn và tối ưu để tìm đường đi ngắn nhất.
- **Cấu trúc trạng thái rõ ràng**: Việc tách các đỉnh phụ (i, j) giúp ta tách chi phí thuê Pokémon khỏi việc di chuyển trên trục chỉ số, làm cho đồ thị vừa dễ hiểu vừa dễ cài đặt.

Nhờ những lý do trên, mô hình đồ thị + Dijkstra không chỉ đúng về mặt logic mà còn đáp ứng tốt yêu cầu về hiệu năng của bài toán.

4 Phân tích độ phức tạp

Độ phức tạp thời gian

Xét một test với n Pokémon và m chỉ số:

- Số đỉnh:
$$V = (m + 1)n$$
gồm n đỉnh chính và mn đỉnh phụ.
- Số cạnh:
  - Từ mỗi cặp (i, j), ta thêm 2 cạnh giữa i và $(i, j) \Rightarrow 2nm$ cạnh.
  - Với mỗi chỉ số j, ta sắp xếp n Pokémon theo $a_{i,j}$ và nối chuỗi bằng $2(n - 1)$ cạnh $\Rightarrow 2m(n - 1)$ cạnh.

  Do đó, tổng số cạnh
$$E = O(nm).$$
- Chạy Dijkstra trên đồ thị với hàng đợi ưu tiên có độ phức tạp:
$$O(E \log V) = O\big(nm \log((m + 1)n)\big).$$

Vì đề bài đảm bảo $\sum n \cdot m \leq 4 \cdot 10^5$ trên toàn bộ test, nên tổng thời gian chạy là chấp nhận được.

Độ phức tạp bộ nhớ

Các cấu trúc dữ liệu sử dụng:

- Mảng c_i và ma trận $a_{i,j}$: $O(nm)$.
- Danh sách kề của đồ thị: lưu $O(E) = O(nm)$ cạnh.
- Mảng khoảng cách `dist`, hàng đợi ưu tiên trong Dijkstra: $O(V) = O(nm)$.

Vì vậy, tổng độ phức tạp bộ nhớ là:
$$O(nm),$$
hoàn toàn phù hợp với giới hạn bộ nhớ của bài.

5 Code C++

```cpp
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
const ll INF = (ll)4e18;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int T; cin >> T;
    while (T--) {
        int n, m;
        cin >> n >> m;

        vector<ll> c(n + 1);
        for (int i = 1; i <= n; ++i) {
            cin >> c[i];
        }

        vector<vector<ll>> a(n + 1, vector<ll>(m + 1));
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= m; ++j) {
                cin >> a[i][j];
            }
        }

        int N = n * (m + 1);                    // tong so dinh
        vector<vector<pair<int,ll>>> g(N + 1);

        auto add_edge = [&](int u, int v, ll w) {
            g[u].push_back({v, w});
        };

        // 1) Tach chi phi thue pokemon: i <-> (i,j)
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= m; ++j) {
                int id = i + j * n;             // dinh phu (i,j)
                add_edge(i, id, 0);             // chon chi so j
                add_edge(id, i, c[i]);          // tra phi c[i]
            }
        }

        // 2) Chuoi hoa theo tung chi so j
        vector<pair<ll,int>> B(n + 1);
        for (int j = 1; j <= m; ++j) {
            for (int i = 1; i <= n; ++i) {
                int id = i + j * n;
                B[i] = {a[i][j], id};
```

```
            }
            sort(B.begin() + 1, B.begin() + n + 1);
            for (int i = 1; i < n; ++i) {
                int u = B[i].second;
                int v = B[i + 1].second;
                ll val_u = B[i].first;
                ll val_v = B[i + 1].first;
                // nho -> lon: cost 0
                add_edge(u, v, 0);
                // lon -> nho: cost chenh lech
                add_edge(v, u, val_v - val_u);
            }
        }

        // 3) Dijkstra tu 1 den n
        vector<ll> dist(N + 1, INF);
        priority_queue<pair<ll,int>, vector<pair<ll,int>>,
                       greater<pair<ll,int>>> pq;
        dist[1] = 0;
        pq.push({0, 1});

        while (!pq.empty()) {
            auto [d, u] = pq.top();
            pq.pop();
            if (d != dist[u]) continue;
            for (auto [v, w] : g[u]) {
                if (dist[v] > d + w) {
                    dist[v] = d + w;
                    pq.push({dist[v], v});
                }
            }
        }

        cout << dist[n] << '\n';
    }

    return 0;
}
```