Flight Discount

Your task is to find a minimum-price flight route from Syrjälä to Metsälä. You have one discount coupon, using which you can halve the price of any single flight during the route. However, you can only use the coupon once.

When you use the discount coupon for a flight whose price is x, its price becomes x/2 (it is rounded down to an integer).

Input

The first input line has two integers n and m: the number of cities and flight connections.
The cities are numbered 1,2,...,n. City 1 is Syrjälä, and city n is Metsälä.

After this there are m lines describing the flights.
Each line has three integers a, b, and c: a flight begins at city a, ends at city b, and its price is c. Each flight is unidirectional.

You can assume that it is always possible to get from Syrjälä to Metsälä.

Output

Print one integer: the price of the cheapest route from Syrjälä to Metsälä.

Constraints

2 <= n <= 10^5
1 <= m <= 2.10^5
1 <= a,b <= n
1 <= c <= 10^9

Example

Input:
3 4
1 2 3
2 3 1
1 3 7
2 1 5

Output:
2

Problem Breakdown

Lets understand the problem first

You're flying from city 1 to city n.
There are m flights with costs.
You're allowed to apply a 50% discount to exactly one flight.
Your goal is to find the cheapest total cost to go from city 1 to city n.

Why Dijkstra?

We're trying to minimize cost → shortest path.

All edge weights are positive, so Dijkstra is valid.

But we have a twist: exactly one discount can be used. This introduces state transitions.

Idea of States

We model each node with 2 states:

state = 0: We haven't used the discount yet.

state = 1: We have used the discount.

So dist[node][0] → min cost to reach node without using discount,
and dist[node][1] → min cost to reach node having used the discount.

State Transitions

From a node u to neighbor v via edge weight w:

If at state 0

We can either not use discount:

```
dist[v][0] = min(dist[v][0], dist[u][0] + w)
```

Or use discount now:

```
dist[v][1] = min(dist[v][1], dist[u][0] + w / 2)
```

If at state = 1:

We cannot use discount again, only go forward:

```
dist[v][1] = min(dist[v][1], dist[u][1] + w)
```

Algorithm

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

#define ll long long
#define pii pair<ll, int>
const ll INF = 1e18;

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

```
    int n, m;
    cin >> n >> m;

    vector<vector<pair<int, int>>> graph(n + 1);

    for (int i = 0; i < m; i++) {
        int a, b, c;
        cin >> a >> b >> c;
        graph[a].push_back({b, c});
    }
```

→ Input and Graph Initialization

```
    vector<vector<ll>> dist(n + 1, vector<ll>(2, INF));
    dist[1][0] = 0;
```

→ Dist array for storing result of states

```
    priority_queue<tuple<ll, int, int>, vector<tuple<ll, int, int>>, greater<>> pq;
    pq.push({0, 1, 0}); // cost, node, state (0 = not used discount, 1 = used)
```

→ Priority Queue for Dijkstra

```
    while (!pq.empty()) {
        auto [currDist, node, used] = pq.top();
        pq.pop();

        if (currDist > dist[node][used]) continue;  Skip this if we already found a better way to reach this node in this state.

        for (auto [nbr, w] : graph[node]) {
            // No discount used yet
            if (used == 0) {
                // Go to nbr without using discount
                if (currDist + w < dist[nbr][0]) {
                    dist[nbr][0] = currDist + w;
                    pq.push({dist[nbr][0], nbr, 0});
                }
                // Use discount now
                if (currDist + w / 2 < dist[nbr][1]) {
                    dist[nbr][1] = currDist + w / 2;
                    pq.push({dist[nbr][1], nbr, 1});
                }
            } else {
                // Already used discount, must pay full price
                if (currDist + w < dist[nbr][1]) {
                    dist[nbr][1] = currDist + w;
                    pq.push({dist[nbr][1], nbr, 1});
                }
            }
        }
    }
```

```
    cout << dist[n][1] << "\n";
    return 0;
}
```

Time Complexity

O((n + m) * log(n)) - each state visited at most once in Dijkstra.

Space Complexity

O(2n) for dist[][].

Common Questions

1. Why two distances per node?

Because we have two states: "used discount" or "not used yet".

2. Why not just halve all edges?

You can use discount only once, so brute forcing that is inefficient.

3. Can we use BFS here?

No. Edge weights vary → Dijkstra is required.

4. What if we can use discount more than once?

Add more states like dist[node][k] where k is how many discounts you've used.

5. Why w / 2 not w * 0.5?

Safer with integers; this also ensures we stay in integer arithmetic. Use w / 2LL to avoid float precision issues.