High Score

You play a game consisting of n rooms and m tunnels. Your initial score is 0, and each tunnel increases your score by x where x may be both positive or negative. You may go through a tunnel several times.

Your task is to walk from room 1 to room n. What is the maximum score you can get?

Input

The first input line has two integers n and m: the number of rooms and tunnels. The rooms are numbered 1,2,...,n.

Then, there are m lines describing the tunnels. Each line has three integers a, b and x: the tunnel starts at room a, ends at room b, and it increases your score by x. All tunnels are one-way tunnels.

You can assume that it is possible to get from room 1 to room n.

Output

Print one integer: the maximum score you can get. However, if you can get an arbitrarily large score, print -1.

Constraints

1 <= n <= 2500
1 <= m <= 5000
1 <= a,b <= n
10^-9 <= x <= 10^9

Example

Input:

```
4 5
1 2 3
2 4 -1
1 3 -2
3 4 7
1 4 4
```

Output:

```
5
```

Problem Breakdown

Lets understand the problem first

You are standing at city 1.
You want to travel to city n.
You have a bunch of one-way roads between cities, and each road gives you a certain score.

Your goal?
👉 Maximize your score by choosing the best possible path from city 1 to city n.

But there's a twist!
If you can keep looping infinitely and keep increasing your score,
then that means the score is unbounded—and we must print -1.

So this is not a shortest path problem.
It's a longest path problem on a directed graph, and it can have negative or positive weights.
That means Dijkstra is off the table.
So what can we use?

Why Not Dijkstra?

Dijkstra's algorithm fails with negative weights

Dijkstra assumes once you reach a node with the shortest path, you don't need to revisit it.
This assumption breaks with negative or positive cycles.

Also, even if we try to run Dijkstra for every node to handle queries, it becomes too slow for large graphs.

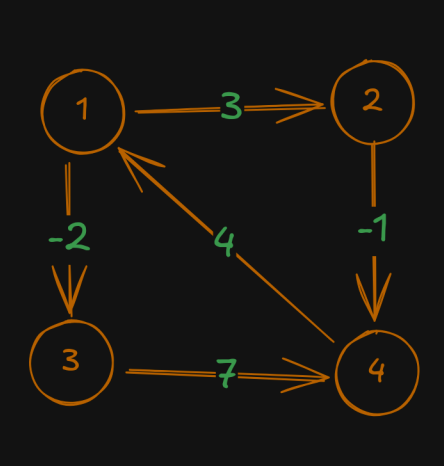
Bellman-Ford Algorithm

Bellman-Ford is a single-source shortest path algorithm that:

- Works with weighted directed/undirected graphs
- Can handle negative weights
- Detects negative weight cycles
- Is simpler than Dijkstra's but has higher time complexity

How It Works - Step by Step

Let's use this example graph with 4 vertices (1-4):



Step 1: Initialize Distances

Set distance to source (A) = 0

Set distance to all other nodes = ∞

Initial distances:
1: 0, 2: ∞, 3: ∞, 4: ∞

Step 2: Relax All Edges Repeatedly

The algorithm relaxes all edges (V-1) times where V = number of vertices.

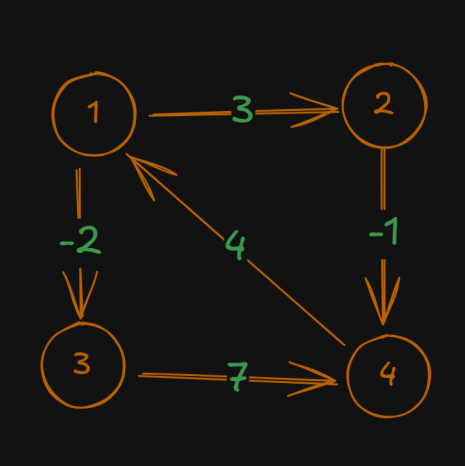
Relaxation Rule

For every edge (u → v) with weight w:

```
if distance[v] > distance[u] + w:
    distance[v] = distance[u] + w
```

Iteration 1:

1 -> 2 : dis[2] = min(0+3, ∞) = 3
1 -> 3 : dis[3] = min(0-2, ∞) = -2
2 -> 4 : dis[4] = min(3-1, ∞) = 2
3 -> 4 : dis[4] = min(-2+7, 2) = 2
4 -> 1 : dis[1] = min(0, 2) = 2



In ith iteration we get <= i path length answers

like if we have i = 3 then we will get answers for 1,2,3 path length

Why (v-1) iterations?

Consider these points:

→ Longest Possible Path Without Cycles:

In a graph with V nodes, the longest path without cycles has V-1 edges

Example: Path A→B→C→D in a 4-node graph has 3 edges

→ Relaxation Property:

Each iteration relaxes all edges

After 1st iteration: Finds shortest paths with at most 1 edge

After 2nd iteration: Finds shortest paths with at most 2 edges

...

After V-1 iterations: Finds shortest paths with at most V-1 edges

→ Guaranteed Completion:

By V-1 iterations, the algorithm must have considered all possible shortest paths

If distances still change after V-1 iterations, there must be a negative cycle

Negative Cycle Detection

The V-th iteration is used to detect negative cycles:

If any distance can still be improved after V-1 iterations

This means there's a negative cycle that can be traversed infinitely to reduce the path weight

Part 2: Modifying Bellman-Ford for Our Problem (High Score)

So what's the trick?

We flip the logic:

🔁 Instead of minimizing, we maximize:

Use dist[i] = -∞ (not ∞)

Set dist[1] = 0

Instead of checking if dist[u] + wt < dist[v],
we do if dist[u] + wt > dist[v] ✅

💡 But we also need to detect positive weight cycles:

That means cycles where you can loop and increase score infinitely.

Same strategy:

Do n-1 iterations of relaxation to find best distances

In the n-th iteration, if you can still improve, you've found a positive cycle

Why not just return -1 on positive cycle?

Because:

You only return -1 if that positive cycle is reachable from 1
AND you can go from that cycle to node n

So we do two BFS traversals:

From node 1, see which cycle nodes are reachable

From those cycle nodes, see if node n is reachable

Only if both are true → return -1

Algorithm

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

typedef long long ll;
typedef pair<int, pair<int, int>> Edge;

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

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

    vector<Edge> edges(m);
    vector<vector<int>> graph(n + 1);
```

→ Input and Graph Initialization

```cpp
    for (int i = 0; i < m; ++i) {
        int u, v, wt;
        cin >> u >> v >> wt;
        edges[i] = {u, {v, wt}};
        graph[u].push_back(v);
    }
```

→ Edge Reading

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

    // Bellman-Ford to find longest path
    for (int i = 0; i < n - 1; ++i) {
        for (auto [u, vw] : edges) {
            int v = vw.first, wt = vw.second;
            if (dist[u] != LLONG_MIN && dist[u] + wt > dist[v]) {
                dist[v] = dist[u] + wt;
            }
        }
    }
```

→ Bellman-Ford for Longest Path

```cpp
    // Detect positive cycles
    set<int> cycle_nodes;
    for (auto [u, vw] : edges) {
        int v = vw.first, wt = vw.second;
        if (dist[u] != LLONG_MIN && dist[u] + wt > dist[v]) {
            cycle_nodes.insert(v);
        }
    }

    // BFS from node 1 to find reachable cycle nodes
    vector<bool> visited(n + 1, false);
    queue<int> q;
    q.push(1);

    queue<int> cycle_q;

    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (visited[u]) continue;
        visited[u] = true;

        if (cycle_nodes.count(u)) {
            cycle_q.push(u);
        }

        for (int v : graph[u]) {
            if (!visited[v]) {
                q.push(v);
            }
        }
    }
```

→ BFS 1: Reach cycle from node 1

```cpp
    // Check if any cycle node can reach node n
    visited.assign(n + 1, false);
    bool can_reach_n = false;

    while (!cycle_q.empty()) {
        int u = cycle_q.front(); cycle_q.pop();
        if (visited[u]) continue;
        visited[u] = true;
        if (u == n) {
            can_reach_n = true;
            break;
        }
        for (int v : graph[u]) {
            if (!visited[v]) {
                cycle_q.push(v);
            }
        }
    }
```

→ BFS 2: Can we reach node n from any cycle node?

```cpp
    if (can_reach_n)
        cout << "-1\n";
    else
        cout << dist[n] << "\n";

    return 0;
}
```

Time Complexity

Bellman-Ford: O(n * m)

Two BFS: O(n + m)

Space Complexity

O(n + m) for graph, distances, visited arrays