史莱姆的弹跳板 ☆☆☆

题目描述

你是一只史莱姆，被困在了一个 $N \times M$ 的网格迷宫中。你的目标是到达出口 E。迷宫包含以下元素：

- `.`：平地，可以通行。
- `#`：墙壁，无法通行，也无法停留。
- `S`：你的起点。
- `E`：出口。
- `T`：弹跳板。

移动规则：

1. **普通移动**：向上下左右移动一格，花费 1 单位时间。
2. **弹跳移动**：若当前站在 `T` 上，可选择向上下左右一次性跳跃 2 格。跳跃可以**跨越**障碍（如`#`），但落脚点必须合法（非墙壁、不越界）。花费 1 单位时间。

请计算从起点 S 到出口 E 的最少时间。如果无法到达，输出 -1。

输入格式

第一行包含两个整数 N 和 M。接下来 N 行，每行包含一个长度为 M 的字符串。

输出格式

一个整数，表示最短时间 t。如果不通，输出 -1。

输入输出样例

输入	输出
5 5 S.#.. T.#.. #.... ...#. ...E.	3

算法分析与代码实现

本题是在二维网格图上的最短路问题。由于无论是“普通移动”还是“弹跳移动”，花费的时间都是固定的 1 单位，因此 广度优先搜索 (BFS) 是解决此类的最优算法。

1. **基础框架与方向数组**

   我们需要定义方向数组来简化上下左右的移动代码。同时，我们需要一个 `dist` 数组来记录起点到每个点的最短距离，初始化为 -1 表示未到达。

```
const int MAXN = 1005;
char mp[MAXN][MAXN];
int dist[MAXN][MAXN];
int n, m;

// 方向数组：上、下、左、右
int dx[4] = {-1, 1, 0, 0};
int dy[4] = {0, 0, -1, 1};

struct Node { int x, y; };
queue<Node> q;
```

2. **BFS 初始化与普通移动**

   从起点 S 开始入队。对于每一个状态 (x, y)，首先尝试普通的“走一步”。

   注意：BFS 的核心在于“第一次到达某点时，步数一定是最少的”，所以如果 `dist[nx][ny]` 已经被更新过（不为 -1），则不需要重复走。

```
    while (!q.empty()) {
        Node curr = q.front(); q.pop();
        int x = curr.x, y = curr.y;

        // 1. 尝试四个方向的普通移动（距离 1）
        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i];
            int ny = y + dy[i];

            // 越界或撞墙检查
            if (nx < 1 || nx > n || ny < 1 || ny > m) continue;
            if (mp[nx][ny] == '#') continue;

            // 如果该点未被访问过
            if (dist[nx][ny] == -1) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
```

3. **特殊机制：弹跳板逻辑**

   在处理当前点 (x, y) 时，如果当前点字符是 $'T'$ ，则可以额外进行一次判定：向四个方向跳跃 **2** 格。

   关键点：跳跃可以跨越障碍，但落脚点必须合法。这意味着我们不需要检查中间那一个格子是什么，只需要检查落脚点 (nnx, nny)。

```
        // 2. 如果脚下是弹跳板 T，尝试弹跳移动（距离 2）
        if (mp[x][y] == 'T') {
            for (int i = 0; i < 4; i++) {
                // 坐标变化量乘 2
                int nnx = x + dx[i] * 2;
                int nny = y + dy[i] * 2;

                // 检查落脚点是否越界
                if (nnx < 1 || nnx > n || nny < 1 || nny > m)
                    continue;
                // 检查落脚点是否是墙（中间可以跨越，不用管）
                if (mp[nnx][nny] == '#') continue;

                if (dist[nnx][nny] == -1) {
                    dist[nnx][nny] = dist[x][y] + 1;
                    // 弹跳也只花 1 时间
                    q.push({nnx, nny});
                }
            }
        }
    }
```

拓展思考

1. 如果弹跳板有方向限制？

如果是“超级马里奥”风格的弹跳板，规定了只能向特定方向弹射（例如 `U` 表示只能向上弹），代码该如何修改？我们需要修改 `mp[x][y] == 'T'` 的判断逻辑，根据当前字符决定遍历方向数组的哪些索引，或者建立字符到方向的映射表。

2. 边权变化：Dijkstra 算法

如果题目改为：普通移动花费 1 秒，但启动弹跳板需要**蓄力花费 3 秒**。此时边权不再全部为 1（有 1 和 3 两种边权），简单的 BFS 队列性质失效（先入队的未必距离更近）。这时候需要使用 **Dijkstra** 算法来求解。