Priklady vystupov aj s vysvetlenim.

Case #1:

Materials: 2

```
 AA
B+A
 C
```

Case #2:

Materials: 2

```
 A
 AA  B
 |   B
 C   B
 CCC-B
```

Case #3:

Materials: 8

```
 A
 AA  B
C+--BB
 D   B
 DDD-B
 | D |
 |   EE
 F   EE
 F
```

Kompletny vystup:

2

2

8

224

229

173

268

233

248

283

0

222

214

279

256

203

233

189

223

222

224

209

198

295

228

236

185

-1

203

220

199

253

217

214

217

269

280

199

234

171

233

164

244

260

244

265

227

235

169

-1

-1

227

153

211

297

26

191

205

205

235

186

224

174

208

305

249

162

214

-1

269

217

187

248

265

202

207

279

338

179

173

217

272

250

231

251

230

219

205

197

248

255

246

142

234

220

209

210

227

209

215

141

-1

333

Riesenie

```
from typing import Dict, List, Set, Tuple

Coordinate = Tuple[int, int]
IslandID = int

INPUT_PATH = "test.in"
BRIDGE_LENGTH = 3


class Bridge:
    def __init__(self, islands, coordinates, length):
        self.islands = islands
        self.coordinates = coordinates
        self.length = length
```

```
    islands: Set[IslandID]
    coordinates: Set[Coordinate]
    length: int


def vector_diff(v1, v2):
    return (v1[0] - v2[0], v1[1] - v2[1])


def vector_add(v1, v2):
    return (v1[0] + v2[0], v1[1] + v2[1])


def normalize_vector(v):
    return tuple(
        map(int, (v[0] / abs(v[0]) if v[0] else 0, v[1] / abs(v[1]) if v[1] else 0))
    )


def ray_trace(start, end):
    direction = vector_diff(end, start)
    normalized_direction = normalize_vector(direction)

    v = vector_add(start, normalized_direction)

    while v != end:
        yield v
        v = vector_add(v, normalized_direction)
```

```
def id_generator():
    i = 0
    while True:
        yield i
        i += 1


def neighbors(row, col):
    return [(row - 1, col), (row + 1, col), (row, col - 1), (row, col + 1)]


def flood_island(row, col, island_id, grid):
    queue = [(row, col)]
    map = {(row, col): island_id}

    while queue:
        v = queue.pop(0)
        for neighbour in neighbors(*v):
            n_row, n_col = neighbour
            try:
                if neighbour not in map and grid[n_row][n_col] == "1":
                    map[neighbour] = island_id
                    queue.append(neighbour)
            except IndexError:
                pass

    return map


def read_test_case(inp):
    grid: List[List[str]] = []
```

```
    height, width = map(int, inp.readline().split())

    for _ in range(height):
        grid.append(list(inp.readline().strip()))

    return grid, height, width


def create_map(grid, height, width) -> Dict[Coordinate, IslandID]:
    id_iterator = id_generator()
    result: Dict[Coordinate, IslandID] = {}

    for row in range(height):
        for col in range(width):
            if grid[row][col] == "1" and (row, col) not in result:
                result.update(flood_island(row, col, next(id_iterator), grid))

    return result


def find_available_bridges(islands_map) -> Set[Bridge]:
    result: Set[Bridge] = set()

    for coordinate, island_id in islands_map.items():
        for neighbour in neighbors(*coordinate):
            if neighbour not in islands_map:
                direction = vector_diff(neighbour, coordinate)

                for i in range(BRIDGE_LENGTH):
                    neighbour = vector_add(neighbour, direction)
```

```
            if neighbour in islands_map and islands_map[neighbour] != island_id:
                result.add(
                    Bridge(
                        {island_id, islands_map[neighbour]},
                        {coordinate, neighbour},
                        i + 1,
                    )
                )
                break

    return result


def find_minimum_spanning_tree(bridges) -> Set[Bridge]:
    tree_islands: Set[IslandID] = {0}
    result: Set[Bridge] = set()

    while bridges:
        if not any(b.islands & tree_islands for b in bridges):
            return result

        l = list(bridges)
        l.sort(key=lambda b: b.length)

        for bridge in l:
            if sum(island_id in tree_islands for island_id in bridge.islands) == 1:
                result.add(bridge)
                tree_islands.update(bridge.islands)
                bridges.difference_update(
                    filter(lambda b: b.islands <= tree_islands, bridges.copy())
                )
```

```
            break

    return result


def main():
    with open(INPUT_PATH, "r") as inp:
        T = int(inp.readline())

        for _ in range(T):
            grid, height, width = read_test_case(inp)

            islands_map: Dict[Coordinate, IslandID] = create_map(grid, height, width)
            islands = set(islands_map.values())

            available_bridges: Set[Bridge] = find_available_bridges(islands_map)

            final_bridges: Set[Bridge] = find_minimum_spanning_tree(available_bridges)

            materials = sum(b.length for b in final_bridges)

            if len(final_bridges) != len(islands) - 1:
                print(-1)
            else:
                print(materials)


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