This week's problem set focuses on tasks solvable using *graph theory*, a topic that is prevalent in computer science and extremely common in competitive programming.

We believe that it would be instructive for you to find an introduction to graph theory, and then try to explicitly model each structure in these problems as a graph/network. What are the vertices and what are the edges? What is the problem asking you to do, in pure graph theoretic terms? Viewing it through the lens of general graphs will help you pick up skills that are applicable to other problems.

A

Just count how many of the (u, v) pairs in the input have $u = 1$ or $v = 1$.

Even though n and m are fixed values, we still encourage you to solve this problem using a for loop, since it's so much cleaner than copy-pasting some code 8 times.

```python
n, m = map(int, input().split())
count_one = 0
for _ in range(m):
    u, v = map(int, input().split())
    if u == 1 or v == 1:
        count_one += 1
print(count_one)
```

B

You might be tempted to use the code from Problem A *for each* value x from 1 to n. In other words, for each x from 1 to n, loop over all m of the (u, v) pairs and count how many pairs have $u = x$ or $v = x$.

Unfortunately, this solution will get you a Time Limit Exceeded. Looping over all m pairs, for every x from 1 to n, means that we have a running time of $\mathcal{O}(mn)$. With $n = m = 10^5$, we plug this into our running time function and get an estimate of 10^{10} operations needed in the worst case, which is too many.

The fix is simple—we need to ensure we only make *one pass* over the (u, v) pairs. Here's what we can do. Make a *frequency array* `freq` that is initially all 0; ultimately, we want `freq[x]` to count the number of pairings with x.

To do this, for each (u, v) pair, apply `freq[u] += 1` and `freq[v] += 1`.

You can use a `std::map` or `dict` for your implementation. Actually, because the elements are guaranteed to be integers from 1 to n, we can also just use an array of size $n + 1$.

The running time is $\mathcal{O}(n + m)$, to make (and later output) the frequency array of size n, and then make one pass over the pairing list of size m.

```python
n, m = map(int, input().split())
freq = [0 for _ in range(n+1)]
for _ in range(m):
    u, v = map(int, input().split())
    freq[u] += 1
    freq[v] += 1

print(*freq[1:])
```

K

Prerequisite: You should learn a graph traversal technique such as Breadth First Search (BFS) or Depth First Search (DFS). We believe that it is possible to come up with a solution *ad hoc*, i.e., without formally invoking graph theory. However, for education value, it would be most instructive for the tutorial to be phrased in terms of graph theory.

Set up a graph as follows. Let the vertices be the squares in the grid without rubble. Then, draw an edge between two vertices if the corresponding cells are within the surrounding 8 cells of the other one.

Here's a neat way to programmatically find the cells which surround some cell (i, j), without copy-pasting. Let di and dj be values such that $-1 \le di \le 1$ and $-1 \le dj \le 1$. Iterate over all such possible pairs of di and dj (using two nested for loops, perhaps) and other than when $di = dj = 0$, the cells of the form $(i + di, j + dj)$ will exactly correspond to the 8 cells that surround (i, j).

To eliminate literal edge cases of possibly going out of bounds, you can also surround the given grid with a "padding" border of impassable rubble. This has the added benefit of allowing the coordinates to be truly 1-indexed.

Two squares are reachable from each other if their corresponding vertices lie in the same connected component. So, run a BFS/DFS from Propesor Pugita's starting location and see if he can reach the hosts. Then, run a BFS/DFS from Gagamboy's starting location and see if he can reach the hosts.

You could also source the BFS/DFS from the location of the hosts, and check if Gagamboy and Propesor Pugita are reachable from them (since edges always go both ways in this model). Ultimately though, it doesn't really matter too much.

Output the appropriate message depending on the results of these two checks.

If you implement the BFS/DFS correct, the running time should just be linear in the size of the graph, which is $\mathcal{O}(rc)$ (since there are $< 8rc/2$ edges).

```python
r, c = map(int, input().split())
grid = [
    ['#']*(c+2)
    for i in range(r+2)
]
for i in range(1, r+1):
    grid[i][1:c+1] = list(input())

def find_hosts(grid, r, c):
    for i in range(1, r+1):
        for j in range(1, c+1):
            if grid[i][j] == '!':
                return i, j

si, sj = find_hosts(grid, r, c)

vis = [
    [False]*(c+2)
    for i in range(r+2)
]
vis[si][sj] = True

frontier = [(si, sj)]
ptr = 0

pugita_reachable, gagamboy_reachable = False, False

```

```python
while ptr < len(frontier):
    i, j = frontier[ptr]
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if (di, dj) != (0, 0):
                if not vis[i + di][j + dj] and grid[i + di][j + dj] != '#':

                    if grid[i + di][j + dj] == '8':
                        pugita_reachable = True
                    if grid[i + di][j + dj] == 'G':
                        gagamboy_reachable = True

                    vis[i + di][j + dj] = True
                    frontier.append((i + di, j + dj))

    ptr += 1

if pugita_reachable:
    if gagamboy_reachable:
        print("SAVED THE DAY")
    else:
        print("GAGAMBOY FAILED")
else:
    print("NO CRISIS")
```

Prerequisite: You should learn a graph traversal technique such as Breadth First Search (BFS) or Depth First Search (DFS). We believe that it is possible to come up with a solution *ad hoc*, i.e., without formally invoking graph theory. However, for education value, it would be most instructive for the tutorial to be phrased in terms of graph theory.

Set up the graph in the same way as was described in the tutorial for Problem K. Now, what we want is a fast way to check if two nodes belong to the same connected component.

Here's an analogy for what we'll do. In K, we use a "paint bucket" to mark all cells that are reachable from e.g. Gagamboy, then check if the target was painted by the operation. For D, what we'll do is paint each connected component *a different color*, then our query will just check if the two cells have the same color. If they do have the same color, then they are reachable from each other.

In addition to a `vis` array which tracks which cells have been visited, let's make a `color` array, as well as a global variable `current_color`, initialized to 0, which we will use to denote the current color. When you start a *new* BFS or DFS, add a line that sets `color[u] = current_color` for all u in this connected component; when you are done, increment `current_color += 1` so that the next connected component gets painted a different color.

The BFS/DFS still only takes $\mathcal{O}(rc)$ time, but now each query can be answered in just $\mathcal{O}(1)$.

```python
r, c = map(int, input().split())
grid = [
    ['#']*(c+2)
    for i in range(r+2)
]
for i in range(1, r+1):
    grid[i][1:c+1] = list(input())

color = [
    [None]*(c+2)
    for i in range(r+2)
]

def paint_bucket(si, sj, current_color, grid):
    frontier = [(si, sj)]
    color[si][sj] = current_color
    ptr = 0

    while ptr < len(frontier):
        i, j = frontier[ptr]
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if (di, dj) != (0, 0):
                    if color[i + di][j + dj] is None and grid[i + di][j + dj] != '#':
                        color[i + di][j + dj] = current_color
                        frontier.append((i + di, j + dj))

        ptr += 1

current_color = 0
for i in range(1, r+1):
    for j in range(1, c+1):
        if grid[i][j] != '#' and color[i][j] is None:
            paint_bucket(i, j, current_color, grid)
            current_color += 1

for q in range(int(input())):
    i1, j1, i2, j2 = map(int, input().split())
    print(
        "CARNAGE"
        if color[i1][j1] == color[i2][j2]
        else "ANTICLIMACTIC"
    )

```