A

Just use a switch statement, or an if-else chain, or a dictionary, in order to handle all the cases.

```
c = input()
if c == 'I':
    print('#.')
    print('#.')
    print('#.')
    print('#.')
elif c == 'O':
    print('##')
    print('##')
    print('..')
    print('..')
elif c == 'J':
    print('.#')
    print('.#')
    print('##')
    print('..')
elif c == 'L':
    print('#.')
    print('#.')
    print('##')
    print('..')
elif c == 'S':
    print('#.')
    print('##')
    print('.#')
    print('..')
elif c == 'T':
    print('#.')
    print('##')
    print('#.')
    print('..')
elif c == 'Z':
    print('.#')
    print('##')
    print('#.')
    print('..')
else:
    print('Impossible case!')
```

B

Let there be R rows and C columns in the grid.

Recall that a 2D grid can be represented as a list of lists. In this particular case, since each cell is represented by one character, we can represent the given 2D grid as a *list of strings*.

So, read this list of strings. We want to remove all the filled lines. So *just do that*: remove all elements that are equal to a filled row (a string of $C = 10$ `#` characters). For a more efficient solution, you can just *create another list* that consists only of the strings in the original list that are *not* equal to filled up rows. This reduces the running time from $\mathcal{O}(R^2)$ down to $\mathcal{O}(R)$. But R is always equal to 20, which is small, so this is not strictly necessary.

Then, just count the number of removed lines. We can do that, for example, by just looking at the different in length between the original and post-removal lists.

Now we just output that many blank lines first, followed by outputting all the original not-removed lines. Then make another switch statement / if-else chain to decide on what message to output in the end.

```
R, C = 20, 10
grid = [input() for _ in range(R)]
uncleared_lines = [line for line in grid if line != '#' * C]
cleared_count = R - len(uncleared_lines)
for _ in range(cleared_count):
    print('.' * C)
for line in uncleared_lines:
    print(line)

if cleared_count > 4:
    print("SUPER QUAD!!!1!")
elif cleared_count == 4:
    print("QUAD!!!!")
elif cleared_count == 3:
    print("TRIPLE!!!")
elif cleared_count == 2:
    print("DOUBLE!!")
elif cleared_count == 1:
    print("SINGLE!")


```

K

Let's try to make some observations. First, when is the task clearly impossible? Well, all our pieces are tetrominos, which have 4 squares. So if the area $R \times C$ is not divisible by 4, then the task is definitely impossible.

That doesn't *necessarily* mean that the task is always possible if $R \times C$ *is* divisible by 4, but at least it tells us that we only need to consider a few cases. When is $R \times C$ divisible by 4?

- It could be that R is divisible by 4. But if that's the case, then the task is always solvable: tile the entire board with vertical I pieces.

  ```
  aba
  aba
  aba
  aba
  bab
  bab
  bab
  bab
  ```

- Similarly, if C is divisible by 4, then it's also always solvable: tile the entire board with *horizontal* I pieces.

  ```
  aaaabbbbaaaa
  bbbbaaaabbbb
  aaaabbbbaaaa
  ```

- Finally, if R and C are both *not* divisible by 4, then $R \times C$ can only be divisible by 4 if **both** of them are divisible by 2. But even in this case, the board is always solvable: tile the entire board with O pieces.

  ```
  aabbaa
  aabbaa
  bbaabb
  bbaabb
  aabbaa
  aabbaa
  ```

As for how to color the pieces—the simplest way to do it is in a "checkerboard" pattern. We leave this last part to you as a guided exercises.

If we wanted a *pure* checkerboard pattern, where each 1×1 square is a different color from the squares next to it, then a common scheme is to color (i, j) black if $i + j$ is even, and white if $i + j$ is odd. That's because taking a step in any NSEW direction will cause the value of $i + j$ to change by ± 1, thereby changing the parity of $i + j$ from even to odd or vice versa, thereby also ensuring that adjacent squares have different colors.

Now you just need to think about how this system can be tweaked so that we're coloring I pieces / O pieces in a checkerboard pattern. Or, you know, you can also just look at the sample solution code and try to figure out why those methods of tweaking the $i + j$ idea are correct for each case!

Bonus: Consider the problem *Pampamilyang Pasalubong* from the NOI.PH 2022 Eliminations Round. The problem is exactly the same as this one, except **two tetrominos of the same type are not allowed to touch each other** (they may not share an edge; they may share borders). The problem is much more difficult, but also very interesting! Think about it!

```python
R, C = map(int, input().split())

if R % 4 == 0:
    print("PC found")
    for i in range(R):
        print(''.join('a' if (i//4 + j) % 2 == 1 else 'b' for j in range(C)))

elif C % 4 == 0:
    print("PC found")
    for i in range(R):
        print(''.join('a' if (i + j//4) % 2 == 1 else 'b' for j in range(C)))

elif R % 2 == 0 and C % 2 == 0:
    print("PC found")
    for i in range(R):
        print(''.join('a' if (i//2 + j//2) % 2 == 1 else 'b' for j in range(C)))

else:
    print("No PC here")

```

D

Suppose we maintain a list h, initialized to all 0, which tracks the height of each column.

Note that the `I` and `-` events can be done in constant time, because they only require you to +4 to *one* column, or +1 to *four* columns. The only thing that could cause us to TLE is the `G` event: literally applying +1 to all but one column will take $\mathcal{O}(c)$ time per `G` event. An $\mathcal{O}(ce)$ solution is too slow because of the **huge** number of events.

Let's consider an easier version of the problem instead: What if `G` caused garbage to be sent to **all** columns, not just all-but-one?

Well, since all columns increase by the same amount, we can create a separate variable called, say, `bonus`, which we initialize to 0. When we receive a `G` event, **instead** of applying +1 to all columns, just increment `bonus` by 1, i.e. `bonus += 1`.

Now, when you want to find the height of some column q, instead of just `h[q]`, the answer will instead be `h[q] + bonus`. In other words, track the contribution of the `G` events separately, but make sure to include them in the final computation in the end.

But what about the actual version of the problem, where a `G` event causes *all except* column g_i to increase in height by 1? Well, break that down into two events:

1. Increase the height of **all** columns by 1.
2. **Decrease** the height of column g_i by 1.

This has the same behavior as the described `G` event, but rephrased in terms of a problem that we already know how to solve quickly!

Initializing the list of heights takes $\mathcal{O}(c)$ time. Since there are e events and each event can be processed in $\mathcal{O}(1)$, processing all of them takes just $\mathcal{O}(e)$ time.

```
m, n = map(int, input().split())

def iterate(A):
    if A <= 1:
        return -1
    else:
        # find the smallest k that _fails_ the test
        k = 2
        while k*k < A:
            k += 1
        return k-1

A = m
t = 1
while t != n:
    if A == -1:
        break
    else:
        A = iterate(A)
    t += 1

print(A)
```

In common comp prog parlance, this technique is often called using a *global delta*, where `delta` is the name given to the separate variable instaed of `bonus`.