A

One possible solution is to just make an if-else chain that acts on the input number.

```
n = int(input())

if n == 1:
    print('A')
elif n == 2:
    print('B')
elif n == 3:
    print('K')
elif n == 4:
    print('D')
else:
    print('Impossible case')
```

Technically, the `print('Impossible case')` part is not necessary because the input constraints guarantee that n is one of 1, 2, 3, or 4, so this case will never happen anyway. But explicitly spelling out the condition that leads to each outcome helps more clearly communicate the intent of our code, and what else would we put in that final case?

Another solution, which is less tedious and also less error-prone, is to create a sequence of the letters A, B, K, and D, in order, using a list or string. We treat the input n as an index (**after subtracting** 1 **from it**). By definition of how lists work, this will get the nth letter in the sequence, which is exactly what we want.

```
n = int(input())
codigo = 'ABKD'
print(codigo[n-1])

```

B

Again, one possible solution is to use four copy-pasted if statements. For each string in the input, we check if it contains the letter A; if it does, we increment a counter that stores the number of people that have solved problem A. Repeat for letters B, K, and D, with their own counters.

```
n = int(input())

count_A = 0
count_B = 0
count_K = 0
count_D = 0

for i in range(n):
    solved = input()
    if 'A' in solved:
        count_A += 1
    if 'B' in solved:
        count_B += 1
    if 'K' in solved:
        count_K += 1
    if 'D' in solved:
        count_D += 1

print(count_A, count_B, count_K, count_D)
```

This is a perfectly valid solution to the problem and we could stop here. We can design this better, though. Everything that follows is enrichment.

Let's aim for a solution that doesn't have so much copy-pasted code. Reducing the amount of copy-pasting done helps us make the code cleaner, easier to read, and less prone to sneaky bugs.

We want to establish a mapping of letter → counter. We can use a data structure like a dictionary to manage this mapping. In the Python code below, we use a `dict`. In C++, you would use an `std::map`.

```
n = int(input())

keys = 'ABKD'
tallies = dict()
for key in keys:
    tallies[key] = 0

for i in range(n):
    solved = input()
    for c in solved:
        tallies[c] += 1

print(tallies['A'], tallies['B'], tallies['K'], tallies['D'])

```

Look at how much neater that looks already!

If we want to lean into Python's features, we can make a few more minor tweaks to clean up the code even further.

```
n = int(input())

keys = 'ABKD'
tallies = {c: 0 for c in keys}

for i in range(n):
    solved = input()
    for c in solved:
        tallies[c] += 1

print(*tallies.values())
```

- Use a dictionary comprehension syntax to initialize the key-value pairs in the `tallies` dictionary
- Use `tallies.values()` to return a sequence containing the values in the dictionary.
    - The asterisk * "unpacks" the sequence, giving the same behavior as if we passed each of its elements as comma-separated arguments to the print function.
    - As of Python 3.6, the standard `dict` maintains insertion order by default. So, `tallies.values` gives the counts in the same order as the tallies were initialized, which is A, then B, then K, then D, which is the behavior that we want.

K

The challenge for this problem is that we want to sort some data according to *decreasing* order of one property, and then break ties in *increasing* order of some other property. We will show two solutions: One solution that only uses the default `sort()` function, and then another that uses an idea called *custom comparators*. Only the first part of "Sort by a key" is strictly necessary to solve the problem; the rest is enrichment.

Sort by a key

Comparison between sequences is usually done in lexicographic order (same as strings, which makes sense, because strings are just sequences of characters). That is to say, we compare sequences by their first element, breaking ties by considering the second element, then breaking *further* ties by considering the third element, and so on.

- For each contestant, create a sequence (like a `tuple` in Python or an `std::pair` in C++) where the first element is the score (the primary sorting property), and the second element is the name (the secondary tie-breaking property).
- We want the score to be sorted in the *opposite* direction that we are sorting the names. Since the score is a number, a common hack is to use the *negative* of the score as the basis for comparison.
    - With this scheme, $-400 < -300$, so 400 will come before 300 in the sequence, as desired.
    - Just remember to negate the values again when we read the sorted list of contestants.

```
n = int(input())
contestants = []
for _ in range(n):
    name, score = input().split()
    contestants.append((-int(score), name))

contestants.sort()

for neg_score, name in contestants:
    print(name, -neg_score)

```

In Python specifically, we can even apply this technique without "destroying" the original data by passing the `key` kwarg (keyword argument) to `sort()`.

```
n = int(input())
contestants = []
for _ in range(n):
    name, score = input().split()
    # Here, the original data's order
    #  and values are "preserved"
    contestants.append((name, int(score)))

def decreasing_score_then_increasing_name(contestant):
    return (-contestant[1], contestant[0])
contestants.sort(key=decreasing_score_then_increasing_name)

# The data is still in its "original" form
#  but it has been sorted properly!
for name, score in contestants:
    print(name, score)
```

Here, the function passed to `key` is applied to each element of the sequence. Then, the *original* elements of the sequence are sorted using these outputs to determine the order.

The above code can be further cleaned up by using an anonymous function (called a lambda in Python) instead of declaring `decreasing_score_then_increasing_name`. Since the function is not used anywhere else anyway, it's neater if it's not polluting the namespace.

```
n = int(input())
contestants = []
for _ in range(n):
    name, score = input().split()
    contestants.append((name, int(score)))

contestants.sort(key=lambda contestant: (-contestant[1], contestant[0]))

for name, score in contestants:
    print(name, score)
```

If you dislike `contestant[0]` and `contestant[1]` because they don't clearly communicate intent when read, you can always either define a `class Contestant` or use a `namedtuple` (from Python's `collections` library) so that the properties are dot-accessed as e.g. `contestant.name` and `contestant.score`. That's straying a bit too far from the main point, so we won't show that here.

Sort with a comparator

Here is another approach for sorting data in some non-standard order *without* "destroying" it. Instead of modifying the *data*, let's modify *what it means for something to be "in order" in the first place.*

For the rest of this subsection, do not read $x < y$ as "x is less than y". In the context of sorting, it's more instructive to interpret it as "x *must come before* y". Similarly, $x > y$ means "x must come after y", and $x = y$ means "the order of x and y is interchangeable."

There are standard interpretations of what it means for $x < y$ in sorting; for example, if x and y are real numbers, then $x < y$ if x is less than y. If we want `<` to mean something else, then we just need to *tell* the computer what we want it to mean.

If we *tell* the computer what ordering scheme we want, then it can sort the data (without destroying it) using that scheme. In Python 3, we use the `key` kwarg. Other languages do this using a *custom comparator*, where we directly provide the function `cmp(x, y)` that gives meaning to $x < y$.

This feature doesn't exist any more in Python 3, as it has been replaced by `key`. However, it's still worth mentioning because many other languages like C++, Java, and JavaScript use comparators for custom sorting. The sample code for the remainder of this section is in C++.

When using `sort` in C++, we can pass a function `bool cmp(Type x, Type y)` as a *third* argument, which will serve as the `<` comparison when sorting. To be precise, the behavior of `cmp` must be as follows:

- returns `True` if x must come before y
- returns `False` otherwise

```
#include <iostream>
#include <vector>
#include <algorithm>  // sort
using namespace std;
using Contestant = pair<string, int>;  // alias for convenience
```

```cpp
int main() {
    int n; cin >> n;
    vector<Contestant> contestants(n);
    for (Contestant& contestant : contestants) {
        cin >> contestant.first >> contestant.second;
    }

    sort(
        contestants.begin(),
        contestants.end(),
        [](Contestant x, Contestant y) {
            // first is name, second is score
            if (x.second != y.second) {
                return x.second > y.second;
            } else {
                return x.first < y.first;
            }
        }
    );

    // unpacking only usable in C++17 onwards
    for (auto&[name, score] : contestants) {
        cout << name << " " << score << endl;
    }
}
```

To describe, in words, what the custom comparator is doing:

- If the scores are different, then x must come before y if the score of x is greater than the score of y.
- Otherwise, x must come before y if the name of x is alphabetically earlier than the name of y.

Note that we give the custom comparator as an anonymous function, which keeps our code clean, since the custom comparator is not used elsewhere in the code anyway.

If you dislike the use of `x.first` and `x.second` because it does not clearly communicate intent when read, then you can define a `struct Contestant` or `class Contestant` instead of just aliasing `std::pair<std::string, int>`. But again, we won't show that here because it's straying from the main point.

As a final technical caveat, **not all orderings** defined by custom comparators are valid. Sometimes it can define an "ordering" that's paradoxical or self-contradictory. An invalid custom comparator can, in the worst case, cause your program to crash due to an infinite loop.

In C++, a comparator $x < y$ is valid if all the following (hopefully familiar) properties are satisfied[1]:

- $x < x$ is **false**
- If $x < y$ is true, then $y < x$ is **false**
- If $x < y$ and $y < z$ are true, then $x < z$ is also true
- Let $x = y$ if $x < y$ and $y < x$ are both false. If $x = y$ and $y = z$, then $x = z$.

These should all be familiar properties if x, y, z are real numbers. In general, if your `<` satisfies these properties, then it forms what is called a *strict weak ordering*. You can convince yourself that if any of these properties don't hold, then this concept of "order" is nonsensical since we can run into contradictions.

[1] `https://codeforces.com/blog/entry/72525`

In particular, **note that $x \leq y$ is not a valid custom comparator** because it fails the second condition (if $x = y$, then $x \leq y$ and $y \leq x$ both hold). Custom comparators in C++ use a **strict** ordering!

Java, JavaScript, and Python 2 have slightly different syntax, but the idea is the same, except your comparator function `cmp(x, y)` should return:

- Any negative value, if $x < y$,
- 0, if $x = y$,
- Any positive value, if $x > y$.

But those languages aren't commonly used in competitive programming, so we won't give an example of that here anymore.

D

In this problem, writing a correct program is straightforward. However, this time, we have to be careful about our *efficiency*. A tutorial with more examples will be uploaded on the NOI.PH website, but you should hopefully be able to follow along by keeping in mind these two main points:

- Counting operations is hard and imprecise, so we use the language of Big O Notation when estimating the running time of our algorithms. Essentially, something like $\mathcal{O}(n^2)$ just means "*approximately* n^2 if n is large". In general $\mathcal{O}(f(n))$ just means "*approximately* $f(n)$ if n is large".
    - We use the word "approximately" in lieu of a more precise technical definition using limits because that technical definition isn't important for how we're going to use it.
- A program can reasonably do somewhere on the order of magnitude of $\approx 10^7$ to 10^8 primitive operations in 1 second, depending on the programming language.
    - The actual running time is dependent on a lot of other factors that get swept under the rug in the approximations we use for Big O Notation. But we can still reason about the general ballpark efficiency of a program.

Now, consider the following solution.

```
n = int(input())
sorted_contestants = []
for _ in range(n):
    name, score = input().split()
    sorted_contestants.append((name, score))

s = int(input())
for _ in range(s):
    name = input()
    for i in range(n):
        if sorted_contestants[i][0] == name:
            print(i+1)
            break
```

This solution is clearly correct, but is it always efficient? Let's see...

- For some search request, suppose that `name` corresponds to the very last person in the list.
- Then the inner loop would have to iterate all the way to the final index before the name is found.
- There are n contestants, so in the worst case where we have to go through all of them before finding the one with the right name, it would take $\mathcal{O}(n)$ operations to answer a search request.
    - This reasoning also applies if `name` is not last in the list, but also just *somewhere near the end* of the list.
- There are s search requests, so in the worst case where all queries are "bad", our program would have to do $\mathcal{O}(s \times n)$ operations in order to answer all of them.

But the problem constraints tell us that s and n can be up to 10^5. So the amount of "stuff" that our program has to do can go to up to $\approx 10^{10}$ operations in the worst case, which is *way* beyond our safety thresholds. This solution will not run under the time limit.

In order to make this solution more efficient, we need to remove redundant work. The bottleneck is the inner loop, where we have to iterate over the *entire* list for *every* search request. A lot of information is getting thrown away and then recomputed anyway, which is what is causing the slowness. Let's try to iterate over all the contestants *only once*.

We can do this using a data structure like a `dict` or `std::map`. In one pass, read every contestant's name, and *store* the corresponding index for each name. Then, answering each search request is just a matter of accessing the correct pre-computed value from the dictionary.

```
n = int(input())
sorted_contestants = []
for _ in range(n):
    name, score = input().split()
    sorted_contestants.append((name, score))

position_of = {}
for i in range(n):
    position_of[contestants[i][0]] = i+1

s = int(input())
for _ in range(s):
    name = input()
    print(position_of[name])
```

Or, actually, we might notice that we don't even need to store the contestants in a list, since they are already sorted. We can insert the contestant's names directly into the dictionary as we read them from input.

```
n = int(input())

position_of = {}
for i in range(n):
    name, score = input().split()
    position_of[name] = i+1

s = int(input())
for _ in range(s):
    name = input()
    print(position_of[name])
```

Dictionary access is fast. In Python, it runs in $\mathcal{O}(1)$ on average (i.e. basically constant time). In C++, it runs in $\mathcal{O}(1)$ on average or $\mathcal{O}(\log_2(n))$ in the worst case, depending on if you're using an `std::unordered_map` or an `std::map`.

We won't go into the specific details here as to why. As builtin data structures, the specific implementation of how they work is interesting, but not necessary to understand, as long as you remember that dictionary lookup is *fast*.

Answering each query in $\mathcal{O}(1)$ on average means that our program runs in $\mathcal{O}(n + s)$ to read the contestants and then answer all the queries. This running time is well within the acceptable limits, so we expect this program to be Accepted.