1. a) Заметим, что если $A[i][j] > key$, то все элементы $A$ в строки $i$ и столбцах $j+1, j+2, \ldots, n-1$ тоже больше key, т.е.

$$A[i][j] > key \Rightarrow \left(\forall k \in \overline{j+1, n-1} \Rightarrow A[i][k] > key\right)$$

множество $\{j+1, j+2, \ldots, n-1\}$

Также, если $A[i][j] > key$, то любой элемент в столбце $j$ и строках $i-1, \ldots, 0$ тоже больше key, т.е.

$$A[i][j] > key \Rightarrow \left(\forall k \in \overline{0, i-1} \Rightarrow A[k][j] > key\right)$$

b) Опишем алгоритм, который решает задачу для матрицы размера $n \times m$ (и, в частности, для матрицы размера $n \times n$)

Пройдёмся циклом по $j$ от 0 до $m-1$ $(0 \le j < m)$ по столбцам матрицы и на каждой итерации будем поддерживать индекс строки $i$, такой что это первая строка в матрице, для которой $A[i][j] \le key$. Пусть при $j = p$ такой индекс $i = q$, тогда точно известно, что $A[q][p] \le key$, и $q = 0$ или $A[q-1][p] > key$. Если $A[q][p] \ne key$, то на следующей итерации, при $j = p+1$,

| $A$ | $p$ | $p+1$ |
|---|---|---|
| $q-1$ | $>key$ | $>key$ |
| $q$ | $\leq key$ | $>key$ |
| $q+1$ | $<key$ | $>key$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $q+d-1$ | $<key$ | $>key$ |
| $q+d$ | $<key$ | $\leq key$ |

Будет увеличивать индекс строки $i$, пока он не достигнет $n-1$ или пока $i$ не станет равным $q+d$ таким, что $A[q+d][p+1] \leq key$. Если $i = n-1$, то элемента $key$ нет в $A$. Если $A[q+d][p+1] == key$, то алгоритм вернёт пару $\{q+d, p+1\}$.

Иначе если $A[q+d][p+1] \neq key$, будет выполнена следующая итерация цикла по $j$. Если $j = n$, то цикл завершается, $key \notin A$.

3) Рассмотрим на примере поиска ключа $key = 17$ в матрице $A =$

$$\begin{bmatrix} 4 & 14 & 18 & 21 & 38 \\ 3 & 11 & 13 & 20 & 25 \\ 2 & 6 & 9 & 17 & 22 \\ 1 & 5 & 8 & 12 & 16 \end{bmatrix}$$

$n = 4,\ m = 5$

Изначально $i=0$. На первой итерации при $j=0$ увеличиваем $i$, пока $i<n$ и $A[i][0] > 17$

$$\begin{array}{cc} & j=0 \\ i=0 \rightarrow & \begin{bmatrix} 4 & 14 & 18 & 21 & 38 \\ 3 & 11 & 13 & 20 & 25 \\ 2 & 6 & 9 & 17 & 22 \\ 1 & 5 & 8 & 12 & 16 \end{bmatrix} \end{array} \longmapsto \begin{array}{cc} & j=0 \\ i=0 \rightarrow & \begin{bmatrix} 4 & 14 & 18 & 21 & 38 \\ 3 & 11 & 13 & 20 & 25 \\ 2 & 6 & 9 & 17 & 22 \\ 1 & 5 & 8 & 12 & 16 \end{bmatrix} \end{array}$$

$i$ не увеличилось. $(i<n) \wedge (A[i][j] \neq 17) \Rightarrow$

$\Rightarrow$ продолжаем работу цикла по $j$.

Т.к. $A[0][0] < key$, то key точно не может быть в нулевом столбце $\Rightarrow$
$\Rightarrow$ при увеличении $j$ на $1$ мы не пропустим key.

При $j = 1$ аналогично $A[0][1] = 14 < key \Rightarrow$
$\Rightarrow$ $i$ не изменяется, key точно не содержится в столбце с индексом $1$, можно увеличить $j$.

При $j = 2$ $A[i][j] = A[0][2] = 18 > 17 = key \Rightarrow$
$\Rightarrow$ key точно не может быть в строке с индексом $0$. Увеличиваем $i$, пока $(i < n) \wedge (A[i][j] > key)$. Получим, что $i = 1$

На первых двух итерациях мы убедились что key не может быть в столбцах с индексами $0$ и $1$, на данной итерации убедились в том, что key не может быть в строке с индексом $0$.
Выделим зелёным область, в которой может быть key.

$j=2$ (над третьим столбцом), $i=0 \rightarrow$ (у первой строки)

| | | | | |
|---|---|---|---|---|
| 4 | 14 | 18 | 21 | 38 |
| 3 | 11 | 13 | 20 | 25 |
| 2 | 6 | 9 | 17 | 22 |
| 1 | 5 | 8 | 12 | 16 |

$\mapsto$

$j=2$ (над третьим столбцом), $i=1 \rightarrow$ (у второй строки)

| | | | | |
|---|---|---|---|---|
| 4 | 14 | 18 | 21 | 38 |
| 3 | 11 | 13 | 20 | 25 |
| 2 | 6 | 9 | 17 | 22 |
| 1 | 5 | 8 | 12 | 16 |

$A[i][j] = A[1][2] = 13 < 17 = key \Rightarrow$ key точно не содержится в столбце с индексом $j=2$ $\Rightarrow$ можно увеличить $j$ на 1 и мы не пропустим элемент key.

На следующей итерации, при $j=3$, $A[i][j] = A[1][3] = 20 > 17 = key \Rightarrow$ в строке с индексом $i=1$ нет ключа key и можно увеличивать $i$ аналогично прошлой итерации.

Получим $i=2$. $A[i][j] == key \Rightarrow$ алгоритм вернёт пару $\{i,j\} = \{2,3\}$.

$$\begin{array}{r|ccccc|}
 & & & & \downarrow j=3 & \\
\hline
 & 4 & 14 & 18 & 21 & 38 \\
i=1 \rightarrow & 3 & 11 & 13 & 20 & 25 \\
 & 2 & 6 & 9 & 17 & 22 \\
 & 1 & 5 & 8 & 12 & 16 \\
\hline
\end{array}
\quad \mapsto \quad
\begin{array}{r|ccccc|}
 & & & & \downarrow j=3 & \\
\hline
 & 4 & 14 & 18 & 21 & 38 \\
 & 3 & 11 & 13 & 20 & 25 \\
i=2 \rightarrow & 2 & 6 & 9 & 17 & 22 \\
 & 1 & 5 & 8 & 12 & 16 \\
\hline
\end{array}$$

Таким образом, алгоритм поддерживает границы области в которой может быть ключ key, и на каждом шаге уменьшает либо $i$, границу верхней строки, либо $j$, границу самого левого столбца области.

Реализация алгоритма на языке C++ может быть такая:

Полный код этой реализации алгоритма находится конце файла.
Работа функции проверялась на версии языка 2020 и 2023 годов
с компилятором g++ версии 12.2.0

```cpp
template <typename T>
using matrix_t = std::vector<std::vector<T>>;

/// @brief Searches for the key in the matrix (with special order of the elements).
/// @tparam T type of the element.
/// @param matrix matrix.
/// @param key key to find.
/// @return pair of indexes { row, column } if element is found. Otherwise, returns { (size_t)-1, (size_t)-1 }.
template <typename T>
requires requires (const T& a, const T& b) {
    { a < b } -> std::same_as<bool>;
    { a == b } -> std::same_as<bool>;
}
std::pair<size_t, size_t> FindKey(const matrix_t<T>& matrix, const T& key) {
    std::pair<size_t, size_t> ans{ static_cast<size_t>(-1), static_cast<size_t>(-1) };
    if (matrix.empty()) {
        return ans;
    }

    size_t n = matrix.size();
    size_t m = matrix[0].size();
    size_t i = 0;
    for (size_t j = 0; j < m; j++) {
        while (i < n && key < matrix[i].at(j)) {
            i++;
        }

        if (i == n) {
            return ans;
        }

        if (key == matrix[i][j]) {
            ans.first = i;
            ans.second = j;
            return ans;
        }
    }

    return ans;
}
```

2. Докажем, что для данной задачи, когда A - квадратная матрица, $T(n) = O(n)$
Укажем в таблице, какие вычислительные затраты нужны на каждой строке и в течении какого количества итераций они выполняются.

| Код | затраты | итераций |
|---|---|---|
| `std::pair<size_t, size_t> ans{ static_cast<si` | $c_1$ | $1$ |
| `if (matrix.empty()) {` | $c_2$ | $1$ |
| `return ans;` | $c_3$ | $1$ |
| `}` | | |
| `size_t n = matrix.size();` | $c_4$ | $1$ |
| `size_t m = matrix[0].size();` | $c_5$ | $1$ |
| `size_t i = 0;` | $c_6$ | $1$ |
| `for (size_t j = 0; j < m; j++) {` | $c_7$ | $m$ |
| `while (i < n && key < matrix[i].at(j)) {` | $c_8$ | $m+n$ |
| `i++;` | $c_9$ | $n-1$ |
| `}` | | |
| `if (i == n) {` | $c_{10}$ | $m-1$ |
| `return ans;` | $c_{11}$ | $m-1$ |
| `}` | | |
| `if (key == matrix[i][j]) {` | $c_{12}$ | $m-1$ |
| `ans.first = i;` | $c_{13}$ | $m-1$ |
| `ans.second = j;` | $c_{14}$ | $m-1$ |
| `return ans;` | $c_{15}$ | $m-1$ |
| `}` | | |
| `}` | | |
| `return ans;` | $c_{16}$ | $1$ |

☐ Тело внутреннего цикла while по i выполнится не более $n-1$ раз, т.к. $0 \le i < n$, а одно из обязательных условий выполнения тела этого цикла – $i < n$. Проверки в цикле while по i выполнятся не более $n+m$ раз, т.к. хотя бы одна проверка выполняется на каждой итерации внешнего цикла по j, а при каждой успешной проверке (==true) i увеличивается на 1.

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 m + c_8(m+n) + c_9(n-1) + c_{10}(m-1) +$$
$$+ c_{11}(m-1) + c_{12}(m-1) + c_{13}(m-1) + c_{14}(m-1) + c_{15}(m-1) + c_{16} =$$
$$= C_1 + C_2 m + C_3 n, where$$
$$C_1 = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 - c_9 - c_{10} - c_{11} - c_{12} - c_{13} - c_{14} - c_{15} + c_{16}$$
$$C_2 = c_7 + c_8 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15}$$
$$C_3 = c_8 + c_9$$
$$\implies T(n) = \mathcal{O}(n+m) = \mathcal{O}(\max(n,m))$$

По условию входная матрица A – квадратная

$$\implies n = m \implies T(n) = \mathcal{O}(\max(n,n)) = \mathcal{O}(n) \quad \blacksquare$$

Полный код реализации алгоритма.

```cpp
#include <cstdint>  // size_t
#include <concepts> // std::same_as<T>
#include <utility>  // std::pair<T, U>
#include <vector>   // std::vector<T, ...>

template <typename T>
using matrix_t = std::vector<std::vector<T>>;

/// @brief Searches for the key in the matrix (with special order of the elements).
/// @tparam T type of the element.
/// @param matrix matrix.
/// @param key key to find.
/// @return pair of indexes { row, column } if element is found. Otherwise, returns { (size_t)-1, (size_t)-1 }.
template <typename T>
requires requires (const T& a, const T& b) {
    { a < b } -> std::same_as<bool>;
    { a == b } -> std::same_as<bool>;
}
std::pair<size_t, size_t> FindKey(const matrix_t<T>& matrix, const T& key) {
    std::pair<size_t, size_t> ans{ static_cast<size_t>(-1), static_cast<size_t>(-1) };
    if (matrix.empty()) {
        return ans;
    }

    size_t n = matrix.size();
    size_t m = matrix[0].size();
    size_t i = 0;
    for (size_t j = 0; j < m; j++) {
        while (i < n && key < matrix[i].at(j)) {
            i++;
        }

        if (i == n) {
            return ans;
        }

        if (key == matrix[i][j]) {
            ans.first = i;
            ans.second = j;
            return ans;
        }
    }

    return ans;
}
```