

**Username:** Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## Arrays

Arrays might be the most simple data structure. Elements are sequentially stored in continuous memory in arrays. When an array is created, its size should be specified. Even though it may only store one element at first, the size is required because we have to allocate memory for all of the elements. Since there may be vacancies in arrays, they are not efficient in memory utilization.

In order to improve space efficiency, dynamic arrays were developed. The class `vector` in the standard template library (STL) of C++ is one such example. Memory is allocated for a few elements in dynamic arrays at first. When the number of elements is greater than the capacity of a dynamic array, more memory is allocated (the capacity doubles when it has to enlarge the capacity of a `vector` in C++), existing elements are copied to the newly allocated space, and the previous memory is released. It reduces waste in memory, but many extra operations are required to enlarge capacity, so it has negative impact on time efficiency. Therefore, it is better to reduce the times needed to enlarge the capacity of dynamic arrays. The type `ArrayList` in both C# and Java is similar to `vector` in C++.

Because memory allocation for arrays is sequential, it only costs  $O(1)$  time to access to an element based on its index, and it is very efficient. A simple hash table can be implemented with an array to utilize its advantage of time efficiency. Each index is treated as a key and every element in an array is treated as a value, so an index and its corresponding element form a pair of key and value. Many problems can be solved with such a hash table, and examples are illustrated in the section *Hash Tables for Characters*. It is a practical solution especially when built-in hash tables are not available in some programming languages such as C/C++.

Arrays and pointers are closely related to each other in C/C++ and also different from each other. The C code in [Listing 3-1](#) shows the relationship between them. What is the output of this code?

**Listing 3-1.** C Code about Arrays and Pointers

```
int GetSize(int data[]) {
    return sizeof(data);
}

int _tmain(int argc, _TCHAR* argv[]) {
    int data1[] = {1, 2, 3, 4, 5};
    int size1 = sizeof(data1);

    int* data2 = data1;
    int size2 = sizeof(data2);

    int size3 = GetSize(data1);


    printf("%d, %d, %d", size1, size2, size3);
}
```

The output should be “20, 4, 4” in a 32-bit system. `data1` is an array, and `sizeof(data1)` gets its size. There are five integers in the array, and each integer occupies four bytes, so the total size of array is 20 bytes.

The name of an array is the address of the first element in the array, so `data2` points to the first element of an array with the statement `data2 = data1`. `data2` is declared as a pointer, and the `sizeof` operator returns 4 for any pointers in a 32-bit system.

When an array is passed as a parameter in C/C++, the compiler treats it as a pointer. What gets passed is the address of the first element in an array, rather than the whole array. Therefore, the result of `sizeof(data)` is also 4 in the function `GetSize` even though `data` is declared as an array in the parameter list.

## Duplication in an Array

 **Question 5** An array contains  $n$  numbers ranging from 0 to  $n-2$ . There is exactly one number duplicated in the array. How do you find the duplicated number? For example, if an array with length 5 contains numbers {0, 2, 1, 3, 2}, the duplicated number is 2.

Suppose that the duplicated number in the array is  $m$ . The sum of all numbers in the array, denoted as  $sum1$ , should be the result of  $0+1+\dots+(n-2)+m$ . It is not difficult to get the sum result of  $0+1+\dots+(n-2)$ , which is denoted as  $sum2$ . The duplicated number  $m$  is the difference between  $sum1$  and  $sum2$ . The corresponding code in Java is shown in [Listing 3-2](#).

**Listing 3-2.** Java Code to Get a Duplicated Number in an Array

```
int duplicate(int numbers[]) {
    int length = numbers.length;

    int sum1 = 0;
    for(int i = 0; i < length; ++i) {
        if(numbers[i] < 0 || numbers[i] > length - 2)
```

```

        throw new IllegalArgumentException("Invalid numbers.");
    }

    sum1 += numbers[i];
}

int sum2 = ((length - 1) * (length - 2)) >> 1;

return sum1 - sum2;
}

```

Source Code:

005\_Duplication.java

Test Cases:

- Normal case: an array with size  $n$  has a duplication
- Boundary case: an array  $\{0, 0\}$  with size 2
- Some numbers are out of the range of 0 to  $n-2$  in an array of size  $n$

**Question 6** An array contains  $n$  numbers ranging from 0 to  $n-1$ . There are some numbers duplicated in the array. It is not clear how many numbers are duplicated or how many times a number gets duplicated. How do you find a duplicated number in the array? For example, if an array of length 7 contains the numbers  $\{2, 3, 1, 0, 2, 5, 3\}$ , the implemented function (or method) should return either 2 or 3.

A naive solution for this problem is to sort the input array because it is easy to find duplication in a sorted array. As we know, it costs  $O(n \log n)$  time to sort an array with  $n$  elements.

Another solution is the utilization of a hash set. All numbers in the input array are scanned sequentially. When a number is scanned, we check whether it is already in the hash set. If it is, it is a duplicated number. Otherwise, it is inserted into the set. The data structure `HashSet` in Java is quite helpful in solving this problem. Even though this solution is simple and intuitive, it has costs:  $O(n)$  auxiliary memory to accommodate a hash set. Let's explore a better solution that only needs  $O(1)$  memory.

Indexes in an array with length  $n$  are in the range 0 to  $n-1$ . If there were no duplication in the  $n$  numbers ranging from 0 to  $n-1$ , we could rearrange them in sorted order, locating the number  $i$  as the  $i^{\text{th}}$  number. Since there are duplicate numbers in the array, some locations are occupied by multiple numbers, but some locations are vacant.

Now let's rearrange the input array. All numbers are scanned one by one. When the  $i^{\text{th}}$  number is visited, first it checks whether the value (denoted as  $m$ ) is equal to  $i$ . If it is, we continue to scan the next number. Otherwise, we compare it with the  $m^{\text{th}}$  number. If the  $i^{\text{th}}$  number equals the  $m^{\text{th}}$  number, duplication has been found. If not, we locate the number  $m$  in its correct place, swapping it with the  $m^{\text{th}}$  number. We continue to scan, compare, and swap until a duplicated number is found.

Take the array  $\{2, 3, 1, 0, 2, 5, 3\}$  as an example. The first number 2 does not equal its index 0, so it is swapped with the number with index 2. The array becomes  $\{1, 3, 2, 0, 2, 5, 3\}$ . The first number after swapping is 1, which does not equal its index 0, so two elements in the array are swapped again and the array becomes  $\{3, 1, 2, 0, 2, 5, 3\}$ . It continues to swap since the first number is still not 0. The array is  $\{0, 1, 2, 3, 2, 5, 3\}$  after swapping the first number and the number with index 3. Finally, the first number becomes 0.

Let's move on to scan the next numbers. Because the following three numbers, 1, 2 and 3, are all equal to their indexes, no swaps are necessary for them. The following number, 2, is not the same as its index, so we check whether it is the same as the number with index 2. Duplication is found since the number with index 2 is also 2.

With an understanding of the detailed step-by-step analysis, it is time to implement code. Sample code in Java is shown in [Listing 3-3](#).

**Listing 3-3.** Java Code to Get a Duplicated Number in an Array

```

int duplicate(int numbers[]) {
    int length = numbers.length;

    for(int i = 0; i < length; ++i) {
        if(numbers[i] < 0 || numbers[i] > length - 1)
            throw new IllegalArgumentException("Invalid numbers.");
    }

    for(int i = 0; i < length; ++i) {
        while(numbers[i] != i) {
            if(numbers[i] == numbers[numbers[i]]) {
                return numbers[i];
            }

            // swap numbers[i] and numbers[numbers[i]]
            int temp = numbers[i];

```

```

        numbers[i] = numbers[temp];

        numbers[temp] = temp;

    }

}

throw new IllegalArgumentException("No duplications.");
}

```

It throws two exceptions in the code to make the code complete and robust. If there are any numbers out of the range between 0 and  $n-1$ , the first exception is thrown. If there is no duplication in the array, the second exception is thrown. It is important for candidates to write complete and robust code during interviews. Source Code:

006\_Duplication.java

Test Cases:

- Normal cases: an array with size  $n$  has one or more duplicated numbers
- Boundary cases: the array {0, 0} with size 2
- Some numbers are out of the range from 0 to  $n-1$  in an array of size  $n$
- No duplication in the array

## Search in a 2-D Matrix

**Question 7** In a 2-D matrix, every row is increasingly sorted from left to right, and the last number in each row is not greater than the first number of the next row. A sample matrix follows. Please implement a function to check whether a number is in such a matrix or not. It returns `true` if it tries to find the number 7 in the sample matrix, but it returns `false` if it tries to find the number 12.

```

1 3 5
7 9 11
13 15 17

```

There are many solutions for this problem. The naive solution with brute force is to scan all numbers in the input matrix. Obviously, it costs  $O(mn)$  time if the size of the matrix is  $m \times n$ .

Since each row in the matrix is sorted and the first number of a row is guaranteed to be greater than or equal to the last number of the preceding row, the matrix can be viewed as a 1-D sorted array. If all rows in the sample matrix are concatenated in top down order, it forms a sorted array {1, 3, 5, 7, 9, 11, 13, 15, 17}. The binary search algorithm is suitable for such a scenario, as shown in [Listing 3-4](#).

**Listing 3-4.** Java Code to Search in a Sorted Matrix

```

boolean find(int matrix[][], int value) {

    int rows = matrix.length;

    int cols = matrix[0].length;

    int start = 0;

    int end = rows * cols - 1;

    while (start <= end) {

        int mid = start + (end - start) / 2;

        int row = mid / cols;

        int col = mid % cols;

        int v = matrix[row][col];

        if (v == value)

            return true;

        if (v > value)

            end = mid - 1;

        else

            start = mid + 1;

    }

    return false;
}

```

```

}

```

If there are  $m$  rows and  $n$  columns in a matrix, the time efficiency for the binary search algorithm is  $O(\log mn)$ .

Source Code:

```
007_FindInSortedMatrix.java
```

Test Cases:

- The matrix contains the target value (including cases where the target value is the maximum or minimum in the matrix)
- The matrix does not contain the target value (including cases where the target is larger than the maximum or less than the minimum)
- Special matrices, including matrices with only one row, only one column, or with only one element

**Question 8** In a 2-D matrix, every row is increasingly sorted from left to right, and every column is increasingly sorted from top to bottom. Please implement a function to check whether a number is in such a matrix or not. For example, all rows and columns are increasingly sorted in the following matrix. It returns `true` if it tries to find number 7, but it returns `false` if it tries to find number 5.

```

1 2 8 9
2 4 9 12
4 7 10 13
6 8 11 15

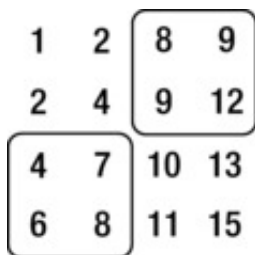
```

Different from the previous problem, the first number in a row may be less than the last number of the preceding row. For instance, the first number in the second row (the number 2) is less than the last number of the first row (the number 9). Therefore, we cannot utilize the binary search algorithm on the 2-D matrix as a whole.

Since each row is sorted, it improves efficiency if a binary search is utilized. It costs  $O(\log n)$  time for a binary search on  $n$  numbers, so the overall time efficiency is  $O(m \log n)$  for an  $m \times n$  matrix. This solution does not fully take advantage of characteristics of an input matrix: all rows are sorted, and all columns are also sorted. More efficient solutions can be found if we fully utilize these characteristics.

### Binary Search on a Diagonal

Because all rows and all columns in an input matrix are sorted, numbers on the diagonal from the top left corner to the bottom right corner are also sorted. Therefore, the binary search algorithm can be applied on numbers on the diagonal. If the target value is on the diagonal, it is done. Otherwise, it gets the greatest number on the diagonal that is less than the target value.



**Figure 3-1.** Find 7 in a 2-D Matrix. The number 4 on the diagonal is located first, which is the greatest number less than the target 7. The whole matrix is split into four sub-matrices by the number 4, and it continues to search in two of them in the rounded rectangles.

The greatest number that is less than the target value on the diagonal splits the whole matrix into four sub-matrices, and it continues to search in two of them. Searching in a sub-matrix is similar to searching in a matrix, so it can be solved recursively.

As shown in Figure 3-1, the target value 7 is not on the diagonal from the top left corner to the bottom right corner. The greatest number less than 7 on the diagonal (the number 4) is found first. Since the target number is greater than 4, it may appear in the northeast area or southwest area (numbers in rounded rectangle). All numbers in the northwest area should be less than 7, and numbers in the southeast area should be greater than 7.

Listing 3-5 provides sample code in Java based on the solution just discussed.

**Listing 3-5.** Java Code to Search in a Partially Sorted Matrix (Version 1)

```

boolean find_solution1(int matrix[][], int value) {
    int rows = matrix.length;
    int cols = matrix[0].length;
    return findCore(matrix, value, 0, 0, rows - 1, cols - 1);
}

boolean findCore(int matrix[][], int value, int row1, int col1, int row2, int col2) {
    if(value < matrix[row1][col1] || value > matrix[row2][col2])
        return false;

    if(value == matrix[row1][col1] || value == matrix[row2][col2])
        return true;

    int copyRow1 = row1, copyRow2 = row2;

```

```

int copyCol1 = col1, copyCol2 = col2;

int midRow = (row1 + row2) / 2;

int midCol = (col1 + col2) / 2;

// find the last element less than value on diagonal
while((midRow != row1 || midCol != col1)
      && (midRow != row2 || midCol != col2)) {
    if(value == matrix[midRow][midCol])
        return true;

    if(value < matrix[midRow][midCol]) {
        row2 = midRow;
        col2 = midCol;
    }
    else {
        row1 = midRow;
        col1 = midCol;
    }

    midRow = (row1 + row2) / 2;
    midCol = (col1 + col2) / 2;
}

// find value in two sub-matrices
boolean found = false;
if(midRow < matrix.length - 1)
    found = findCore(matrix, value, midRow + 1, copyCol1, copyRow2, midCol);
if(!found && midCol < matrix[0].length - 1)
    found = findCore(matrix, value, copyRow1, midCol + 1, midRow, copyCol2);

return found;
}

```

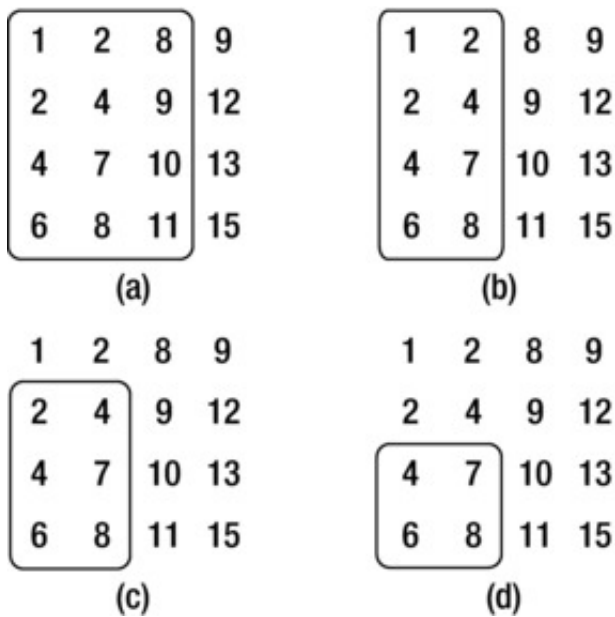
If the diagonal length of the input matrix is  $l$ , the time efficiency of the solution can be calculated with the equation  $T(l) = 2T(l/2) + \log l$ . According to the master theory,  $T(l) = O(l)$ . Additionally, the length  $l$  in a  $m \times n$  matrix is  $l = O(m+n)$ . Therefore, the time efficiency of the solution above is  $O(m+n)$ .

### Removing a Row or a Column at Each Step

When complicated problems are encountered during interviews, an effective method is to use examples to simplify complexity. We can also start from some examples to solve this problem. Let's analyze the step-by-step process to find the number 7 out of the sample matrix in the problem description.

First, we choose the number 9 at the top right corner of the matrix. Since 9 is greater than 7, and 9 is the first number, also the least one, in the fourth column, all numbers in the fourth column should be greater than 7. Therefore, it is not necessary to search in the last column any more, and it is safe to just focus on the other three columns, as shown in [Figure 3-2\(a\)](#).

The number at the top right corner of the remaining matrix is 8, which is also greater than 7, so the third column can also be removed. Let's just focus on the remaining two columns, as shown in [Figure 3-2\(b\)](#).



**Figure 3-2.** Find 7 in a 2-D Matrix. Numbers in the rounded rectangle are the focus of the next round of searching. (a) The number 9 at the top right corner is greater than the target value 7, so the column containing 9 is removed. (b) 8 is also greater than 7, so we remove the column containing 8. (c) The number 2 at the top right corner is less than 7, so the row containing 2 is removed. (d) 4 is less than 7 too, so the row containing 4 is removed.

The number 2 is at the top right corner of the remaining matrix with only two columns. Since 2 is less than 7, the target number 7 may be at the right side of 2 or below 2 according to the sorting rules. All columns at the right side of the number 2 have been removed, so it is safe to ignore them. Therefore, the target number 7 should be below the number 2, and the row containing 2 can also be removed. It is only necessary to search in a 3x2 sub-matrix (Figure 3-2(c)).

Similarly to the previous step, the row containing the number 4 can be removed as well because 4 is less than 7. The sub-matrix left with two rows and two columns is shown in Figure 3-2(d).

The number 7 at the top right corner of the remaining 2x2 sub-matrix equals the target number, so the target has been found and we stop searching here.

The following rules can be summarized based on the detailed analysis step-by-step. The number at the top right corner is selected in each round of searching, and it is compared with the target value. When it is the same as the target value, it stops to search. If it is greater than the target value, the last column in the remaining sub-matrix is removed. If it is less than the target value, the first row in the remaining sub-matrix is removed. Therefore, it reduces the sub-matrix by a row or a column if the target value is not at the top right corner.

It is not difficult to develop code after clearly understanding the searching process. Some sample code in Java is found in Listing 3-6.

**Listing 3-6.** Java Code to Search in a Partially Sorted Matrix (Version 2)

```
boolean find_solution2(int matrix[][], int value) {
    boolean found = false;
    int row = 0;
    int col = matrix[0].length - 1;

    while(row < matrix.length && col >= 0) {
        if(matrix[row][col] == value) {
            found = true;
            break;
        }

        if(matrix[row][col] > value)
            --col;
        else
            ++row;
    }

    return found;
}
```

Since a row or a column is removed in each round of searching, it costs  $O(m+n)$  time for a matrix with  $m$  rows and  $n$  columns.

In the previous analysis, the number at the top right corner is selected in each round of searching. Similarly, we can also select the number at the bottom left corner. Please try using the numbers at the bottom left corner if you are interested. However, numbers at the top left corner or bottom right corner are not appropriate choices. Let's take numbers at the top left corner as a quick example. The number 1 is at the top left corner of the original matrix. Since the target value 7 is greater than 1, it may be at the right side of 1 or be in rows below 1. Neither a row nor a column can be removed based on this comparison.

Source Code:

```
008_FindInPartiallySortedMatrix.java
```

Test Cases:

- The matrix contains the target value (including cases where the target value is the maximum or minimum in the matrix)
- The matrix does not contain the target value (including cases where the target is larger than the maximum or less than the minimum)
- Special matrices, including matrices with only one row, only one column, or with only one element