


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.

Search and Sort

The most common search algorithms are sequential search and binary search. If an input array is not sorted, or input elements are accommodated by dynamic containers such as lists, it has to be searched sequentially. If the input is a sorted array, usually the binary search algorithm is a good choice. For example, the problems “Minimum of a Rotated Array” (Question 27), “Turning Number in a Sorted Array” (Question 28), and the “Time of Occurrences in a Sorted Array” (Question 83) can all be solved by the binary search algorithm.

 **Tip** If it is required to search or count a number in a sorted array (even the array is only partially sorted), we may try with the binary search algorithm.

If it is allowed to use auxiliary memory, a hash table might facilitate searching, with which a value can be located in $O(1)$ time with a key. Some common interview questions, such as “First Character Appearing Once in a String” (Question 76) and “Delete Duplication in a String” (Question 79), are all solved with hash tables.

There are many sort algorithms available, such as insert sort, bubble sort, merge sort, and quicksort. Candidates should be familiar with each one’s pros and cons in terms of space consumption, time efficiency on average, and in worst cases.

Many interviewers like to ask candidates to implement their own quicksort. The key step of the quicksort algorithm is to partition an array into two parts. It selects a pivot number and moves numbers less than the pivot to the left side, and it moves numbers greater than the pivot to the right side. The partition method can be implemented in Java, as shown in [Listing 4-6](#).

Listing 4-6. Java Code to Partition an Array

```
int partition(int[] numbers, int start, int end) {
    Random random = new Random();

    int pivot = random.nextInt(end - start + 1) + start;
    swap(numbers, pivot, end);

    int small = start - 1;

    for(int i = start; i <= end; ++i) {
        if(numbers[i] < numbers[end]) {
            ++small;
            if(i != small)
                swap(numbers, small, i);
        }
    }

    ++small;

    if(small != end)
        swap(numbers, small, end);

    return small;
}
```

The method `swap` is to swap two elements in an array. After the array is partitioned into two parts, they are sorted recursively, as shown in [Listing 4-7](#).

Listing 4-7. Java Code for Quicksort

```
void quicksort(int[] numbers, int start, int end) {
    if(start >= end)
        return;

    int index = partition(numbers, start, end);
    quicksort(numbers, start, index);
    quicksort(numbers, index + 1, end);
}
```

The method `quicksort` sorts an array with n elements with the parameter `start` as 0 and `end` as $n-1$.

Quicksort might be the most efficient sort algorithm in general, but it is not the best one for all cases. The time complexity is $O(n^2)$ in the worst cases, when the pivot is the least or greatest number in each round of partition. In order to avoid the worst cases, we randomize the choice of the pivot number in the method `partition`.

The method `partition` is useful not only to sort, but also to find the arbitrary k^{th} number from an array. The interview question “Majorities in Arrays”

(Question 29) and “Minimal k Numbers” (Question 70) are both solved with the `partition` method.

Different sort algorithms work for different scenarios. If it is required to implement an algorithm to sort, it is critical to know the details about the input data as well as limitations about time and space. For example, the following is a piece of dialog during an interview:

Interviewer: How do you implement a sort algorithm in $O(n)$ time?

Candidate: What are the elements to be sorted, and how many?

Interviewer: I am going to sort ages of all employees in our company. The total number is tens of thousands.

Candidate: Does it mean all the elements are in some narrow range?

Interviewer: Yes.

Candidate: Is it allowed to use auxiliary memory?

Interviewer: It depends on how much memory is used. You can only utilize space less than $O(n)$.

It is encouraged for candidates to ask questions of their interviewers. With questions and answers, we know the numbers to be sorted are in a narrow range and an auxiliary space less than $O(n)$ can be employed. Therefore, this problem can be solved with the count sort algorithm, which is implemented in [Listing 4-8](#).

Listing 4-8. Java Code of Count Sort

```
void countSort(int ages[]) {
    int oldestAge = 99;

    int timesOfAge[] = new int[oldestAge+1];

    for(int i = 0; i <= oldestAge; ++ i)
        timesOfAge[i] = 0;

    for(int i = 0; i < ages.length; ++ i) {
        int age = ages[i];

        if(age < 0 || age > oldestAge)
            throw new IllegalArgumentException("Out of range.");

        ++ timesOfAge[age];
    }


    int index = 0;
    for(int i = 0; i <= oldestAge; ++ i) {
        for(int j = 0; j < timesOfAge[i]; ++ j) {
            ages[index] = i;
            ++ index;
        }
    }
}
```

The method `countSort` assumes all ages are in the range between 0 and 99. It counts the occurrence of each age in the array. If an age occurs m times in the array, it writes the age for m time continuously in the array.

It reads and writes the input array once, so the overall time complexity is $O(n)$ to sort an array with size n . No matter how large n is, it only allocates an auxiliary array `timesOfAge` with size 100, so the space complexity is $O(1)$.

All the sort algorithms above are about arrays. Lists are sorted with other algorithms, which are discussed in the section *Sort Lists*.

Binary Search in Partially Sorted Arrays

 **Question 27** When some elements at the beginning of an array are moved to the end, it becomes a rotation of the original array. Please implement a function to get the minimum number in a rotation of an increasingly sorted array. For example, the array {3, 4, 5, 1, 2} is a rotation of array {1, 2, 3, 4, 5}, of which the minimum is 1.

Binary search is suitable for sorted arrays. Let's try to utilize it on a rotation of a sorted array. It is noticeable that a rotation of a sorted array can be partitioned into two sorted sub-arrays, where numbers in the first sub-array are greater than numbers in the second one. Additionally, the minimum is on the boundary of two sub-arrays.

Two pointers P_1 and P_2 are utilized. P_1 references the first element in the array, and P_2 references the last element. According to the rotation rule, the first element should be greater than or equal to the last one.

The algorithm always compares the number in the middle with the numbers pointed to by P_1 and P_2 during binary search. If the middle number is in the first increasingly sorted sub-array, it is greater than or equal to the number pointed to by P_1 . In such cases, the minimal number is behind the middle number in the array. Therefore, it moves P_1 to the middle, which is also in the first sub-array. It continues to search numbers between P_1 and P_2 recursively.

If the middle number is in the second sub-array, it is less than or equal to the number pointed to by P_2 . The minimal number is before the middle number in the array in such cases, so it moves P_2 to the middle. It can continue to search recursively too because P_1 still points to a number in the first sub-array, and P_2 points to a number in the second sub-array.

No matter if it moves P_1 or P_2 for the next round of search, half of the array is excluded. It stops searching when P_1 points to the last number of the first sub-array and P_2 points to the first number of the second sub-array, which is also the minimum of the array.

Let's take the sample array {3, 4, 5, 1, 2} as an example. P_1 is initialized pointing to the first element (with index 0), and P_2 points to the last element (with index 4), as shown in [Figure 4-3\(a\)](#). The middle element 5 (with index 2) is greater than the number pointed by P_1 , so it is in the first increasingly sorted sub-array. Therefore, P_1 is moved to the middle of the array, as shown in [Figure 4-3\(b\)](#).

The middle element 1 (with index 3) at this time is less than the number pointed by P_2 , so it is in the second sub-array. It moves P_2 to the middle and continues the next round of search (Figure 4-3(c)).

Now the distance between two pointers is 1. It means that P_1 already points to the last element in the first sub-array, and P_2 points to the first element in the second sub-array. Therefore, the number pointed to by P_2 is the minimum in the array.

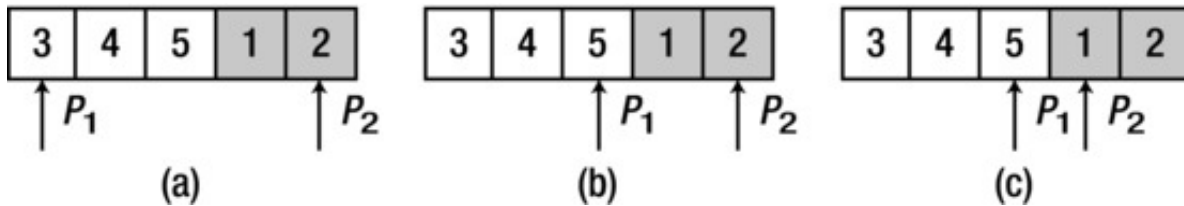


Figure 4-3. Search the minimal element in an array {3, 4, 5, 1, 2}. Elements in the gray background are in the second increasingly sorted sub-array. (a) P_1 points to the first element, and P_2 points to the last element. Since the middle number 5 is greater than the number pointed to by P_1 , it moves P_1 to the middle number for the next round of search. (b) The middle number 1 is less than the number pointed to by P_2 , so it moves P_2 to the middle number for the next round of search. (c) Since P_1 and P_2 point to two adjacent numbers, the number pointed to by P_2 is the minimum.

In the analysis above, the middle element is in the first sub-array if it is greater than the element pointed to by P_1 , and it is in the second sub-array if it is less than the element pointed to by P_2 . However, the middle element and elements pointed to by P_1 and P_2 may be equal when there are duplicated elements in the array. Does binary search work for such cases?

Let's look at other examples. Two arrays {1, 0, 1, 1, 1} and {1, 1, 1, 0, 1} are both rotations of an increasingly sorted array {0, 1, 1, 1, 1}, which are visualized in Figure 4-4.

In these two cases of Figure 4-4, the elements pointed to by P_1 and P_2 , as well as the middle element, are all 1. The middle element with index 2 is in the second sub-array in Figure 4-4(a), while the middle element is in the first sub-array in Figure 4-4(b).

Therefore, the algorithm cannot determine if the middle element belongs to the first or second sub-array when the middle element and the two numbers pointed to by P_1 and P_2 are equal, and it cannot move pointers to narrow the search range. It has to search sequentially in such a scenario.

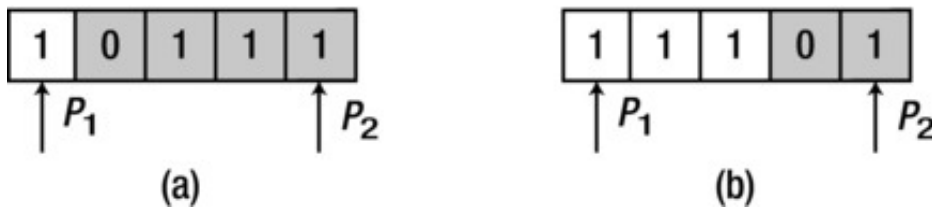


Figure 4-4. Two rotations of an increasingly sorted array {0, 1, 1, 1, 1}: {1, 0, 1, 1, 1} and {1, 1, 1, 0, 1}. Elements with gray background are in the second increasingly sorted sub-array. In both cases, the first, last, and middle numbers are equal. (a) The middle element is in the second sub-array. (b) The middle element is in the first sub-array.

The sample code to get the minimum from a rotation of a sorted array is shown in Listing 4-9.

Listing 4-9. Java Code to Get the Minimum from a Rotation of a Sorted Array

```
int getMin(int numbers[]) {
    int index1 = 0;
    int index2 = numbers.length - 1;
    int indexMid = index1;

    while(numbers[index1] >= numbers[index2]) {
        if(index2 - index1 == 1) {
            indexMid = index2;
            break;
        }

        indexMid = (index1 + index2) / 2;

        // if numbers with indexes index1, index2, indexMid
        // are equal, search sequentially
        if(numbers[index1] == numbers[index2]
            && numbers[indexMid] == numbers[index1])
            return getMinSequentially(numbers, index1, index2);

        if(numbers[indexMid] >= numbers[index1])
            index1 = indexMid;
        else if(numbers[indexMid] <= numbers[index2])
            index2 = indexMid;
    }
}
```

```

        return numbers[indexMid];
    }

    int getMinSequentially(int numbers[], int index1, int index2) {
        int result = numbers[index1];
        for(int i = index1 + 1; i <= index2; ++i) {
            if(result > numbers[i])
                result = numbers[i];
        }

        return result;
    }
}

```

Source Code:

027_ArrayRotation.java

Test Cases:

- Functional cases: rotations of an increasingly sorted array, with/without duplicated numbers
- Boundary cases: An increasingly sorted array (rotating 0 number in the array), an array with only one number

Question 28 A turning number is the maximum number in a unimodal array that increases and then decreases. Please write a function (or a method) that finds the index of the turning number in a unimodal array.

For example, the turning number in the array {1, 2, 3, 4, 5, 10, 9, 8, 7, 6} is 10, so its index 5 is the expected output.

As we know, the binary search algorithm is suitable for search of a number in a sorted array. Since the input array for this problem is partially sorted, we may also try with binary search.

Let's try to get the middle number in an array. The middle number of the array {1, 2, 3, 4, 5, 10, 9, 8, 7, 6} is 5. It is greater than its previous number 4 and less than its next number 10, so it is in the increasing sub-array. Therefore, numbers before 5 can be discarded in the next round of the search.

The remaining numbers for the next round of search are {5, 10, 9, 8, 7, 6}, and the number 9 is in the middle of them. Since 9 is less than its previous number 10 and greater than its next number 8, it is in the decreasing sub-array. Therefore, numbers after 9 can be discarded in the next round of search.

The remaining numbers for the next round of search are {5, 10, 9}, and the number 10 is in the middle. Notice that the number 10 is greater than the previous number 5 and greater than the next number 9, so it is the maximal number. That is to say, the number 10 is the turning number in the input array.

We can see the process above is actually a classic binary search. Therefore, we can implement the required functionality based on the binary search algorithm, as shown in [Listing 4-10](#).

Listing 4-10. Java Code for Turning Number in Array

```

int getTurningIndex(int numbers[]) {
    if(numbers.length <= 2)
        return -1;

    int left = 0;
    int right = numbers.length - 1;

    while(right > left + 1) {
        int middle = (left + right) / 2;
        if(middle == 0 || middle == numbers.length - 1)
            return -1;

        if(numbers[middle] > numbers[middle - 1]
            && numbers[middle] > numbers[middle + 1])
            return middle;
        else if(numbers[middle] > numbers[middle - 1]
            && numbers[middle] < numbers[middle + 1])
            left = middle;
        else
            right = middle;
    }
}

```

```
return -1;
}
```


Source Code:

```
028_TurningNumber.java
```

Test Cases:

- Functional cases: Unimodal arrays with turning numbers
- Boundary cases: A unimodal array with only three numbers

Majorities in Arrays

 **Question 29** How do you find the majority element in an array when it exists? The majority is an element that occurs for more than half of the size of the array. For example, the number 2 in the array {1, 2, 3, 2, 2, 2, 5, 4, 2} is the majority element because it appears five times and the size of the array is 9.

It is easy to get the majority element if the array is sorted because it is not difficult to count the occurrence of each number in a sorted array. It takes $O(n \log n)$ time to sort an array with n elements as well as $O(n)$ time to count, so the time efficiency of this intuitive solution is $O(n \log n)$. Let's explore more efficient solutions.

Based on the Partition Method

The intuitive solution above does not utilize the characteristic of a majority element, which occurs for more than half of the size of an array. If there is a majority element in an array, the majority should occur in the middle of the array when it is sorted. That is to say, the majority of an array is also the median of the array, which is the $(n/2)^{th}$ number in an array with n elements. There is an algorithm available to get the arbitrary k^{th} ($0 \leq k < n$) number in an array in $O(n)$ time.

This algorithm is closely related to the quicksort algorithm, where a pivot is selected to partition an array into two parts. All numbers less than the pivot are located to the left side, and others are located to the right side. If the index of the pivot is $n/2$, it is done because the median is found. If the index is greater than $n/2$, the median should be in the left side of the pivot, so it continues to partition in the left side. Similarly, it continues to partition in the right side if the index is less than $n/2$. It is a typical recursive process, which might be implemented as shown in [Listing 4-11](#).

Listing 4-11. Java Code to Get the Majority (Version 1)

```
int getMajority_1(int[] numbers) {
    int length = numbers.length;

    int middle = length >> 1;

    int start = 0;

    int end = length - 1;

    int index = partition(numbers, start, end);

    while(index != middle) {
        if(index > middle) {
            end = index - 1;

            index = partition(numbers, start, end);
        }
        else {
            start = index + 1;

            index = partition(numbers, start, end);
        }
    }

    int result = numbers[middle];

    if(!checkMajorityExistence(numbers, result))

        throw new IllegalArgumentException("No majority exists.");

    return result;
}
```

The method `partition` was discussed before for quicksort.

The majority is the element that occurs for more than half of the size of the array. How about the scenario where the element occurring most frequently does not meet the bar? That is why a method `checkMajorityExistence` is defined. It is important to handle invalid inputs during interviews, as shown in [Listing 4-12](#).

Listing 4-12. Java Code to Check Existence of Majority

```
boolean checkMajorityExistence(int[] numbers, int number) {
```

```

int times = 0;

for(int i = 0; i < numbers.length; ++i) {

    if(numbers[i] == number)

        times++;

}

return (times * 2 > numbers.length);

}

```

Based on the Definition of Majority

According to the definition of the majority, the occurrence of a majority element is greater than the total occurrences of all other elements. Therefore, this problem can be solved with a new strategy. It scans the array from the beginning to the end, and saves and updates an element of the array as well as a number for occurrences. When an element is visited, the occurrence number is incremented if the currently visited element is the same as the saved one. Otherwise, it decreases the occurrence number when the visited element is different from the saved one. When the occurrence number becomes 0, it saves the currently visited element and sets the occurrence number as 1. The last element that sets the occurrence number to 1 is the majority element.

This solution might be implemented as the code in [Listing 4-13](#), where the method `checkMajorityExistence` is the same as in the preceding solution.

Listing 4-13. Java Code to Get the Majority (Version 2)

```

int getMajority_2(int[] numbers) {

    int result = numbers[0];

    int times = 1;

    for(int i = 1; i < numbers.length; ++i) {

        if(times == 0) {

            result = numbers[i];

            times = 1;

        }

        else if(numbers[i] == result)

            times++;

        else

            times--;

    }

    if(!checkMajorityExistence(numbers, result))

        throw new IllegalArgumentException("No majority exists.");

    return result;

}

```

Comparison

The time efficiencies for these two solutions are both $O(n)$. They differ from each other in whether you are allowed to alter the array. It is noticeable that elements in the input array are reordered in the first solution, so it modifies the input array. Is it allowed to modify the input? It depends on the requirement. Therefore, it is necessary to ask the interviewer for clarification. If it is disallowed to modify, we have to take the second solution or make a copy of the input array and swap numbers in the copy while applying the first solution.

Source Code:

029_MajorityElement.java

Test Cases:

- Functional cases: Arrays with/without majority elements
- Boundary cases: An array with only one element