Time Efficiency

No one likes to wait for a delayed response from a software application, so companies spend a lot of resources on performance optimization before software releases. That is why many interviewers pay so much attention to time efficiency in candidates' code. Besides coding skills, interviewers also examine candidates' passion on performance tuning.

Programming habits greatly impact performance because there are many details that could affect execution efficiency. For instance, C/C++ programmers should get into the habit of passing instances of complex types with references or pointers in argument lists, especially when a parameter is a container such as `vector` with lots of elements. A copy-by-value parameter costs an unnecessary copy, and it should be avoided as much as possible. Take strings in C# as another example. It is not a good practice to utilize the '+' operator repeatedly to concatenate strings because a temp instance of string is returned for each '+' operation, which wastes both time and space. A better choice is to utilize the method `StringBuilder.Append` to concatenate strings.

Recursive and iterative implementations of an algorithm may differ significantly from the perspective of performance. When a complex problem is divided into manageable subproblems and subproblems are solved recursively, it is important to check whether there are overlaps among subproblems. The complexity may grow exponentially if subproblems overlap each other. In order to solve this kind of problem, we can analyze them with recursion but implement iterative code with arrays or matrices to cache resolutions of subproblems. Dynamic programming algorithms are usually applied with these two steps.

Performance optimization requires deep understanding of data structures and algorithms. The costs to insert, delete, and search elements in various data containers are quite different. It is important to choose appropriate data structures to solve coding problems during interviews. For example, arrays, lists, and binary trees are not as efficient as heaps to get the median from a stream. Additionally, it costs O(n) time to search from n numbers sequentially. If the numbers are sorted (or partially sorted), we may try the binary search algorithm, which reduces the time complexity to O(log$n$). We can improve the efficiency to O(1) if a hash table is built in advance.

It is important to demonstrate passion for performance optimization during interviews. Often interviewers ask for more efficient solutions after candidates have already proposed several solutions. In such difficult times, candidates should show a positive attitude and try their best to solve problems from various perspectives. Sometimes interviewers may not expect candidates to find a perfect solution in only 30 or 40 minutes. What they are looking for is the attitude and passion to try new solutions and pursue perfection. Usually, they believe candidates who give up easily are not qualified to be outstanding engineers.

Median in a Stream

Question 69 How do you find the median from a stream of numbers? The median of a set of numbers is the middle one when they are arranged in order. If the count of numbers is even, the median is defined as the average value of the two numbers in the middle.

Since numbers come from a stream, the count of numbers is dynamic and increases over time. If a data container is defined for the numbers from a stream, new numbers will be inserted into the container when they are deserialized. Let's find an appropriate data structure for such a data container.

An array is the simplest choice. If the array is not sorted, the median can be found based on the `partition` method, which is discussed in the section *Search and Sort* for the quicksort algorithm. It costs O(1) time to add a number into an unsorted array and O(n) time to get the median. More details about this solution are available in the section *Majorities in Arrays* to get the majority element from an array.

We can also keep the array sorted while adding new numbers. Even though it only costs O(log$n$) time to find the right position with the binary search algorithm, it costs O(n) time to insert a number into a sorted array because O(n) numbers will be moved if there are n numbers in the array. It is very efficient to get the median since it only takes O(1) time to access a number in an array with an index.

A sorted list is another choice. It takes O(n) time to find the appropriate position to insert a new number. Additionally, the time to get the median can be optimized to O(1) if we define two pointers that point to the central one or two elements.

A better choice is a binary search tree because it only costs O(log$n$) on average to insert a new node. However, the time complexity is O(n) for the worst cases when the binary search tree is extremely unbalanced and becomes similar to a sorted list. To get the median number from a binary search tree, auxiliary data to record the number of nodes in subtrees are necessary for each node. It also requires O(log$n$) time to get the median node on average, but O(n) time for the worst cases.

We may utilize a balanced binary search tree, AVL, to avoid the worst cases for normal binary search trees. Usually, the balance factor of a node in AVL trees is the height difference between its right subtree and left subtree. We may modify the balance factor a little bit here: we define the balance factor as the count difference of nodes between the right subtree and left subtree. It costs O(log$n$) time to insert a new node into an AVL, and O(1) time to get the median for all cases.

An AVL is efficient, but unfortunately it is not implemented in libraries of the most common programming languages. It is also very difficult for candidates to implement the left/right rotation of AVL trees in dozens of minutes during interviews. Let's look for better solutions.

As shown in Figure 7-1, if all numbers are sorted, the numbers that are related to the median are indexed by P_1 and P_2. If the count of numbers is odd, P_1 and P_2 point to the same central number. If the count is even, P_1 and P_2 point to two numbers in the middle.

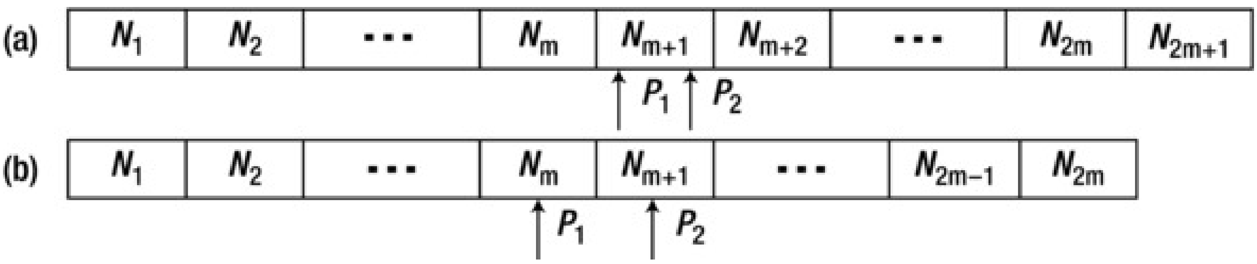

Figure 7-1. *Elements in an array are divided into two parts by one or two numbers in the center. (a) The count of elements is odd. (b) The count of elements is even.*

Median can be calculated with the numbers pointed to by P_1 and P_2. Note that the sequence of numbers is divided into two parts. The numbers in the first half are less in value than the numbers in the second half. Moreover, the number indexed by P_1 is the greatest number in the first half, and the number indexed by P_2 is the least one in the second half.

If numbers are divided into two parts and all numbers in the first half are less than the numbers in the second half, we can get the median with the greatest number of the first part and the least number of the second part, even if numbers in the first and second half are not sorted. How do you get the greatest number efficiently? Utilizing a max heap. A min heap is also an efficient way to get the least number.

Therefore, numbers in the first half are inserted into a max heap, and numbers in the second half are inserted into a min heap. It costs O(log*n*) time to insert a number into a heap. Since the median can be gotten or calculated with the root of a min heap and a max heap, it only takes O(1) time.

The comparisons of the solutions above with an array (sorted or unsorted), a sorted list, a binary search tree, an AVL tree, as well as a min heap and a max heap, are summarized in Table 7-1.

Let's consider the implementation details with heaps. All numbers should be evenly divided into two parts, so the count difference of numbers in the min heap and max heap should be 1 at most. To achieve such a division, a new number is inserted into the min heap if the count of existing numbers is even; otherwise, it is inserted into the max heap.

We also should make sure that the numbers in the max heap are less than the numbers in the min heap. Suppose that the count of existing numbers is even; a new number will be inserted into the min heap. If the new number is less than some numbers in the max heap, it violates our rule that all numbers in the min heap should be greater than numbers in the max heap.

***Table 7-1.** Time Efficiency Comparisons of Solutions with a Sorted Array, a Sorted List, a Binary Search Tree, an AVL Tree, as well as a Min Heap and a Max Heap*

Type of Data Container	Time to Insert	Time to Get Median
Unsorted Array	O(1)	O(*n*)
Sorted Array	O(*n*)	O(1)
Sorted List	O(*n*)	O(1)
Binary Search Tree	O(log *n*) on average, O(*n*) for the worst case	O(log *n*) on average, O(*n*) for the worst case
AVL	O(log*n*)	O(1)
Max and Min Heap	O(log*n*)	O(1)

In such a case, we can insert the new number into the max heap first, then pop the greatest number from the max heap, and finally push it into the min heap. Since the number pushed into the min heap is the former greatest number in the max heap, all numbers in the min heap are still greater than numbers in the max heap even with the newly moved number.

The situation is similar when the count of existing numbers is odd and the new number to be inserted is greater than some numbers in the min heap. Please analyze the insertion process carefully by yourself.

The code in Listing 7-1 is sample code in C++. Even though there are no types for heaps in STL, we can build heaps with vectors utilizing the functions `push_heap` and `pop_heap`. Comparison functors `less` and `greater` are employed for max heaps and min heaps respectively.

***Listing 7-1.** C++ Code to Get the Median from a Stream*

```
template<typename T> class DynamicArray {
public:
    void Insert(T num) {
        if(((minHeap.size() + maxHeap.size()) & 1) == 0) {
            if(maxHeap.size() > 0 && num < maxHeap[0]) {
                maxHeap.push_back(num);
                push_heap(maxHeap.begin(), maxHeap.end(), less<T>());

                num = maxHeap[0];

                pop_heap(maxHeap.begin(), maxHeap.end(), less<T>());
                maxHeap.pop_back();
            }

            minHeap.push_back(num);
            push_heap(minHeap.begin(), minHeap.end(), greater<T>());
        }
        else {
            if(minHeap.size() > 0 && minHeap[0] < num) {
                minHeap.push_back(num);
                push_heap(minHeap.begin(), minHeap.end(), greater<T>());

                num = minHeap[0];

                pop_heap(minHeap.begin(), minHeap.end(), greater<T>());
                minHeap.pop_back();
            }
```

```
            maxHeap.push_back(num);
            push_heap(maxHeap.begin(), maxHeap.end(), less<T>());
        }
    }

    int GetMedian() {
        int size = minHeap.size() + maxHeap.size();
        if(size == 0)
            throw exception("No numbers are available");

        T median = 0;
        if(size & 1 == 1)
            median = minHeap[0];
           else
            median = (minHeap[0] + maxHeap[0]) / 2;

        return median;
    }

private:
    vector<T> minHeap;
    vector<T> maxHeap;
};
```

In the code above, the function `Insert` is used to insert a new number deserialized from a stream into the heaps, and `GetMedian` is used to get the median of all existing numbers.

Source Code:

`069_MedianStream.cpp`

Test Cases:

- Functional Test Cases (Get the median of a stream with an even/odd number of numbers)
- Boundary Test Cases (Get the median of a stream with only one or two numbers; get the median of a stream with duplicated numbers)

Minimum *k* Numbers

Question 70 Please find the smallest *k* numbers (in value) out of *n* numbers. For example, if given an array with eight numbers {4, 5, 1, 6, 2, 7, 3, 8}, please return the least four numbers 1, 2, 3, and 4.

The naive solution is to sort the *n* input numbers increasingly and to have the least *k* numbers be the first *k* numbers. Since it needs to sort, its time complexity is O(*n*log*n*). Interviewers will probably ask to explore more efficient solutions.

O(*n*log*k*) Time Efficiency

We want to create a container that will contain the least *k* numbers out of n input numbers. A data container with capacity *k* is first created to store the *k* numbers and then numbers are read out of the *n* input numbers one by one, compared, and entered into *k* if they meeting the following criteria:

- If the container has less than *k* numbers, the number read in the current round (denoted as *num*) is inserted into the container directly.
- If it contains *k* numbers already, *num* cannot be inserted directly any more. An existing number in the container should be replaced with *num* in such a case. We find the maximum number out of the existing *k* numbers in the container and compare that with *num*. If *num* is less than the maximum number, the maximum number is replaced with *num*. Otherwise *num* is discarded since there have been *k* numbers in the container that are all less than *num* and *num* cannot be one of the least *k* numbers.

Three steps may be required when a number is read and the container is full. The first step is to find the maximum number; in the second, the maximum number may be deleted; and third, the new number may be inserted. The second and third steps are optional and depend on the comparison between *num* in the current round and the maximum number in the container.

We have different choices for the data container. If the data container is implemented as a binary search tree, it costs O(log*k*) time for these three steps on average, but costs O(*k*) time in the worst cases. Therefore, the overall time complexity is O(*n*log*k*) for *n* input numbers on average and O(*nk*) in the worst cases.

Since we need to get the maximum number out of k numbers, a maximum heap might be a good choice. In a maximum heap, its root is always greater than its children, so it costs O(1) time to find the maximum number. However, it takes O(log$k$) time to insert and delete a number. In Java, the default implementation of type `PriorityQueue` is for minimum heaps. A new comparator has to be defined because a maximum heap is needed here, and `ReversedComparator` shown in Listing 7-2 is such an example.

Listing 7-2. *Java Code to Get Minimum k Numbers (Version 1)*

```
void getLeastNumbers_1(int[] input, int[] output) {
    ReversedComparator comparator = new ReversedComparator();
    PriorityQueue<Integer> maxQueue = null;
    maxQueue = new PriorityQueue<Integer>(1, comparator);

    getLeastNumbers(input, maxQueue, output.length);

    Iterator<Integer> iter = maxQueue.iterator();
    for(int i = 0; i < output.length; ++i) {
        output[i] = iter.next();
    }
}

void getLeastNumbers(int[] input, PriorityQueue<Integer> output, int k) {
    output.clear();

    for(int i = 0; i < input.length; ++i) {
        if(output.size() < k)
            output.add(new Integer(input[i]));
        else {
            Integer max = output.peek();
            Integer number = new Integer(input[i]);
            if(output.comparator().compare(number, max) > 0) {
                output.poll();
                output.add(number);
            }
        }
    }
}

class ReversedComparator implements Comparator<Integer> {
    public int compare(Integer int1, Integer int2) {
        int num1 = int1.intValue();
        int num2 = int2.intValue();

        if(num1 < num2)
            return 1;
        else if (num1 == num2)
            return 0;
        else
            return -1;
    }
}
```

O(n) Time Efficiency

We can also utilize the method `partition` in quicksort to solve this problem, assuming that n input numbers are contained in an array. If it takes the k^{th} number as a pivot to partition the input array, all numbers less than the k^{th} number should be at the left side and other greater ones should be at the right side. The k numbers at the left side are the least k numbers after the partition. We can develop the code in Listing 7-3 according to this solution.

Listing 7-3. *Java Code to Get Minimum k Numbers (Version 2)*

```
void getLeastNumbers_2(int[] input, int[] output) {
    int start = 0;
```

```
        int end = input.length - 1;

        int k = output.length;

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

        while(index != k - 1) {

            if(index > k - 1) {

                end = index - 1;

                index = partition(input, start, end);

            }

            else {

                start = index + 1;

                index = partition(input, start, end);

            }

        }


        for(int i = 0; i < k; ++i)

            output[i] = input[i];

    }
```

Comparison between the Two Solutions

The second solution, based on the method `partition`, costs only $O(n)$ time, so it is more efficient than the first one. However, it has two obvious limitations. One limitation is that it needs to load all input numbers into an array, and the other is that we have to reorder the input numbers.

Even though the first solution takes more time, it has some advantages. Reordering the input numbers (`input` in the previous code) is not required. We read a number from data at each round, and all write operations are taken in the container `maxQueue` . It does not require loading all input numbers into memory at one time, so it is suitable for huge-sized data sets. If an interviewer asks us get the least k numbers from a huge-size input, obviously we cannot load all data with huge size into limited memory at one time. We can read a number from auxiliary space (such as disks) at each round with the first solution and determine whether it should be inserted into the container `maxQueue` . It works once memory can accommodate `maxQueue` , so the first option is especially suitable for scenarios where n is huge and k is small.

The comparison of these two solutions can be summarized in Table 7-2.

Table 7-2. Pros and Cons of Two Solutions

Criteria	First Solution	Second Solution
Time complexity	$O(n\log k)$	$O(n)$
Reorder input numbers?	No	Yes
Suitable for huge-size data?	Yes	No

Since each solution has its own pros and cons, candidates should ask interviewers for more detailed requirements to choose the most suitable solution, including the input data size and whether it is allowed to reorder the input numbers.

Source Code:

`070_KLeastNumbers.java`

Test Cases:

- Functional Test Cases (An input array with/without duplicated numbers)
- Boundary Test Cases (The input k is 1, or the length of the input array)

Intersection of Sorted Arrays

Question 71 Please implement a function that finds the intersection of two sorted arrays. Assume numbers in each array are unique.
For example, if the two sorted arrays as input are {1, 4, 7, 10, 13} and {1, 3, 5, 7, 9}, the output is an intersection array with numbers {1, 7}.

An intuitive solution for this problem is to check whether every number in the first array (denoted as *array1*) is in the second array (denoted as *array2*). If the length of *array1* is m, and the length of *array2* is n, its overall time complexity is $O(mn)$ based on linear search. We have two better solutions.

With $O(m+n)$ Time

Note that the two input arrays are sorted. Supposing that a number *number1* in *array1* equals a number *number2* in *array2*, the numbers after *number1* in *array1* should be greater than the numbers before *number2* in *array2*. Therefore, it is not necessary to compare the numbers after *number1* in *array1* with numbers before *number2* in *array2*. This improves efficiency since many comparisons are eliminated.

The sample code for this solution is shown in Listing 7-4.

***Listing 7-4.** C++ Code for Intersection of Arrays (Version 1)*

```
void GetIntersection_solution1(const vector<int>& array1,
                 const vector<int>& array2,
                 vector<int>& intersection) {
    vector<int>::const_iterator iter1 = array1.begin();
    vector<int>::const_iterator iter2 = array2.begin();

    intersection.clear();

    while(iter1 != array1.end() && iter2 != array2.end()) {
        if(*iter1 == *iter2) {
            intersection.push_back(*iter1);
            ++ iter1;
            ++ iter2;
        }
        else if(*iter1 < *iter2)
            ++ iter1;
        else
            ++ iter2;
    }
}
```

Since it only requires scanning two arrays once, its time complexity is O(m+n).

With O(*n*log*m*) Time

As we know, a binary search algorithm requires O(log$m$) time to find a number in an array with length m. Therefore, if we search each number of an array with length n from another array with length m, its overall time complexity is O($n$log$m$). If m is much greater than n, O($n$log$m$) is actually less than O(m+n). Therefore, we can implement a new and better solution based on binary search in such a situation, as shown in Listing 7-5.

***Listing 7-5.** C++ Code for Intersection of Arrays (Version 2)*

```
/* === Supposing that array2 is much longer than array1 === */
void GetIntersection_solution2(const vector<int>& array1,
                 const vector<int>& array2,
                 vector<int>& intersection) {
    intersection.clear();

    vector<int>::const_iterator iter1 = array1.begin();
    while(iter1 != array1.end()) {
        if(binary_search(array2.begin(), array2.end(), *iter1))
            intersection.push_back(*iter1);
    }
}
```

There are some implementation details worthy of attention. The parameters `array1` and `array2` are passed as references. If they were passed by values, it would cost O(m+n) time to copy them.

Source Code:

`071_ArrayIntersection.cpp`

Test Cases:

- Functional Test Cases (Two arbitrary arrays with/without intersected numbers)
- Boundary Test Cases (Two arrays are identical to each other; one or two arrays are empty; the intersecting numbers are the first/last number of the input arrays)

Greatest Sum of Sub-Arrays

Question 72 Given an integer array containing positive and negative numbers, how do you get the maximum sum of its sub-arrays? Continuous numbers form a sub-array of an array.

For example, if the input array is {1, -2, 3, 10, -4, 7, 2, -5}, the sub-array with the maximum sum is {3, 10, -4, 7, 2} whose sum 18.

During interviews, many candidates can solve this problem by enumerating all sub-arrays and calculating their sums. An array with n elements has $n(n+1)/2$ sub-arrays. It costs $O(n^2)$ time (at least) to calculate their sums. Usually, the intuitive and brute-force solution is not the most efficient one. It is highly possible for interviewers to ask for better solutions.

Analyzing Numbers in the Array One by One

Let's accumulate each number in the sample array from beginning to end. Our solution initializes *sum* as 0. In the first step, it adds the first number 1, and *sum* becomes 1. And then if it adds the second number -2, *sum* becomes -1. At the third step, it adds the third number 3. Notice that the previous *sum* is less than 0, so the new *sum* will be 2 and it is less than the third number 3 itself. Therefore, the previous accumulated *sum* -1 should be discarded.

The key point here is that when the *sum* becomes a negative number or zero, adding this *sum* to the following array element will not be greater than the element itself, so the new sub-array will start from the next element.

It continues accumulating from the next number with *sum* 3. When it adds the fourth number 10, *sum* becomes 13, and it decreases to 9 when it adds the fifth number -4. Notice that the *sum* with -4 is less than the previous *sum* 13 because of the negative number -4. It saves the previous *sum* 13 since it might be the max sum of sub-arrays.

At the sixth step, it adds the sixth number 7 and *sum* becomes 16. Now *sum* is greater than the previous max sum of sub-arrays, so the max sum is updated to 16. It is similar when it adds the seventh number 2. The max sum of sub-arrays is updated to 18. Lastly it adds -5 and *sum* becomes 13. Since it is less than the previous max sum of sub-arrays, the final max sum of sub-arrays remains 18, and the sub-array is {3, 10, -4, 7, 2} accordingly. The whole process is summarized in Table 7-3.

Table 7-3. *TheProcess to Calculate the Maximum Sum of All Sub-Arrays in the Array {1, -2, 3, 10, -4, 7, 2, -5}*

Step	Operation	Accumulated Sum	Maximum Sum
1	Add 1	1	1
2	Add -2	-1	1
3	Discard sum -1, add 3	3	3
4	Add 10	13	13
5	Add -4	9	13
6	Add 7	16	16
7	Add 2	18	18
8	Add -5	13	18

The code in Listing 7-6 is the sample code according to the step-by-step analysis just discussed.

Listing 7-6. *Java Code to Get Maximum Sum of Sub-Arrays*

```
int getGreatestSumOfSubArray(int[] numbers) {
    int curSum = 0;
    int greatestSum = Integer.MIN_VALUE;
    for(int i = 0; i < numbers.length; ++i) {
        if(curSum <= 0)
            curSum = numbers[i];
        else
            curSum += numbers[i];

        if(curSum > greatestSum)
            greatestSum = curSum;
    }

    return greatestSum;
}
```

Dynamic Programming

If a candidate is familiar with dynamic programming, he or she might analyze this problem in a new way. If a function $f(i)$ stands for the maximum sum of a sub-array ending with the i^{th} number, the required output is $max[f(i)]$. The function $f(i)$ can be calculated with the following recursive equation:

$$f(i) = \begin{cases} number[i] & i = 0 \text{ or } f(i-1) \le 0 \\ f(i-1) + number[i] & i \ne 0 \text{ and } f(i-1) > 0 \end{cases}$$

If the sum of the sub-array ending with the $(i\text{-}1)^{th}$ number is negative or zero, the sum of the sub-array ending with the i^{th} number should be the i^{th} number itself (for example, the third step in Table 7-3). Otherwise, it gets the sum of the sub-array ending with the i^{th} number by adding the i^{th} number and the sum of the sub-array ending with the $(i\text{-}1)^{th}$ number.

Even though it analyzes the problem recursively, it is usually implemented based on iteration. The iterative code according to the equation above should be the same as the code of the first solution. The variable `curSum` is the $f(i)$ in the equation, and `greatestSum` is $max[f(i)]$. Therefore, these two solutions are essentially identical to each other.

Source Code:

`072_GreatestSumOfSubarrays.java`

Test Cases:

- Functional Test Cases (An array with positive numbers, negative numbers, or zeros; all numbers in the input array are positive/negative/zero)
- Boundary Test Cases (There is only one number in the input array)

Digit 1 Appears in Sequence from 1 to *n*

Question 73 How many times does the digit 1 occur in the integers from 1 to *n ?*

For example, if *n* is 12, there are four numbers from 1 to 12 containing the digit 1, which are 1, 10, 11, and 12, and the digit 1 occurs five times.

Straightforward but Inefficient

The intuitive way to solve this problem is to get a count of digit 1 in each number. The least important digit in a decimal number can be calculated with the modulo operator (%). If the number is greater than 10 with two digits at least, it is divided by 10 and another modulo operation is applied to calculate the last digit again. The process is repeated until the number is less than 10, which can be implemented as the function `NumberOf1` in Listing 7-7.

Listing 7-7. *C Code to Get the Number of 1 in Consecutive Integers (Version 1)*

```
int NumberOf1Between1AndN_Solution1(unsigned int n) {
    int number = 0;
    unsigned int i;

    for(i = 0; i <= n; ++ i)
        number += NumberOf1(i);

    return number;
}

int NumberOf1(unsigned int n) {
    int number = 0;
    while(n) {
        if(n % 10 == 1)
            number ++;

        n = n / 10;
    }

    return number;
}
```

It takes a modulo operation and a division on each number to get the count of all 1 digits. A number *m* has O(log*m*) digits, so it costs O(*n*log*n*) time to count all 1 digits in numbers from 1 to *n*. Let's explore more efficient solutions.

Based on Divide and Conquer

We may have a try to analyze how the 1 digit occurs in numbers. Let's take a somewhat large *n* as an example, such as 21345. Numbers from 1 to 21345 are divided into two ranges. The first one contains numbers from 1 to 1345 (the number excluding the first digit of 21345), and the other contains numbers from 1346 to 21345.

We first count the 1 digit in the range from 1346 to 21345. Let's focus on the most significant digit in numbers with five digits. The 1 digit occurs 10000 (10^4) times in the 10000 numbers from 10000 to 19999 in the first digit.

It should be noticed that there are cases where the 1 digit occurs less than 10000 times in numbers with five digits. For example, when the most significant digit of the input *n* is 1, such as 12345, the digit 1 occurs in the most significant digit 2346 times in numbers from 10000 to 12345 in the first digit.

Let's move on to focus on digits except the most significant one. The 1 digit occurs for 2000 times in the last four digits of numbers from 1346 to 21345. Numbers from 1346 to 21345 are divided into two ranges, 10000 numbers in each range: from 1346 to 11345, and from 11346 to 21345. The last four digits in numbers in these two ranges can be viewed as digit permutations. One of the digits is 1, and the other three digits can be any digit from 0 to 9. According to the principle of permutation, the 1 digit occurs 2000 (2×10^3) times.

How do you get the count of the 1 digit in numbers from 1 to 1345? It can be gotten with recursion. The reason we divide numbers from 1 to 21345 into two ranges (from 1 to 1345 and from 1346 to 21345) is that we get 1345 when the first digit is removed from the number 21345. Similarly, numbers from 1 to 1345 are divided into two ranges: from 1 to 345 and from 346 to 1345. It is a typical divide-and-conquer strategy, and it can be implemented as shown in Listing 7-8. A number is converted into a string for coding simplicity.

Listing 7-8. *C Code to Get the Number of 1 in Consecutive Integers (Version 2)*

```
int NumberOf1Between1AndN_Solution2(int n) {
    char strN[50];

    if(n <= 0)
        return 0;

    sprintf(strN, "%d", n);
    return NumberOf1InString(strN);
}

int NumberOf1InString(const char* strN) {
    int first, length;
    int numOtherDigits, numRecursive, numFirstDigit;

    if(!strN || *strN < '0' || *strN > '9' || *strN == '\0')
        return 0;

    first = *strN - '0';
    length = (unsigned int)(strlen(strN));

    if(length == 1 && first == 0)
        return 0;

    if(length == 1 && first > 0)
        return 1;

    // If strN is 21345, numFirstDigit is the number of digit 1
    // in the most signification digit in numbers from 10000 to 19999
    numFirstDigit = 0;
    if(first > 1)
        numFirstDigit = PowerBase10(length - 1);
    else if(first == 1)
        numFirstDigit = atoi(strN + 1) + 1;

    // numOtherDigits is the number of digit 1 in digits except
    // the most significant digit in numbers from 01346 to 21345
        numOtherDigits = first * (length - 1) * PowerBase10(length - 2);

    // numRecursive is the number of digit 1 in numbers from 1 to 1345
    numRecursive = NumberOf1InString(strN + 1);

    return numFirstDigit + numOtherDigits + numRecursive;
}

int PowerBase10(unsigned int n) {
    int i, result = 1;
    for(i = 0; i < n; ++ i)
        result *= 10;

    return result;
}
```

A digit is removed at each step, so the recursion depth is the same as the count of digits in the input integer n. An integer n has $O(\log n)$ digits, so the time complexity is $O(\log n)$, and it is more efficient than the preceding solution.

Source Code:

073_NumberOf1.c

Test Cases:

- Functional Test Cases (Input 5, 10, 55, 99, ...)
- Boundary Test Cases (Input 0, and 1)
- Performance Test Cases (Input some large numbers, such as 10000, 21235)

Concatenate an Array to Get a Minimum Number

Question 74 Big numbers can be formed if numbers in an array are concatenated together. How do you print the minimum concatenated number of a given array?
For example, if the input array is {3, 32, 321}, there are six concatenated numbers and the minimum one is 321323.

An intuitive solution for this problem is to get all permutations of the given array at first and then concatenate each permutation together. The minimum concatenated number is selected after all concatenated permutations are compared. We have discussed permutation in the section *Permutations and Combinations*.

As we know, there are $n!$ permutations for an array with n element, so this intuitive solution costs $O(n!)$ time. Let's explore more efficient solutions.

It is essential to find a sort rule to reorder elements in an array in order to get the minimum concatenated number. We have to compare numbers in order to sort an array. That is to say, when given two numbers m and n, it is necessary to find a rule to compare them and place one of them before the other.

Numbers mn and nm are concatenated from two numbers m and n. (In this section, a number mn stands for a concatenated number of m and n, rather than m×n.) If $mn < nm$, the output is mn, in which m is placed ahead of n. A new operator $\vee$ is defined. When m is placed ahead of n, it is denoted as $m \vee n$. Similarly, it is denoted as $m \wedge n$ when m is placed behind n. In such a case, $mn > nm$ and the output is nm. When $mn = nm$, it does not matter which number is placed ahead of the other. It is denoted $m \equiv n$ for this case.

In the analysis above, we have found a new way to compare two numbers m and n according to their concatenated numbers mn and nm. Therefore, the array can be reordered with a new comparator `NumericComparator`, as shown in Listing 7-9.

***Listing 7-9.** Java Code to Reorder an Array to Get Minimum Concatenated Number*

```
void PrintMinNumber(int numbers[]) {
    String strNumbers[] = new String[numbers.length];
    for(int i = 0; i < numbers.length; ++i) {
        strNumbers[i] = String.valueOf(numbers[i]);
    }

    Arrays.sort(strNumbers, new NumericComparator());

    for(int i = 0; i < numbers.length; ++i)
        System.out.print(strNumbers[i]);
    System.out.print("\n");
}

class NumericComparator implements Comparator<String>{
    public int compare(String num1, String num2) {
        String str1 = num1 + num2;
        String str2 = num2 + num1;
        return str1.compareTo(str2);
    }
}
```

In the code above, it converts integers to strings to concatenate numbers. The reason for the conversion is to avoid overflow problems. When two numbers m and n are in the range for integers, their concatenated numbers mn and nm may be beyond the range. Numbers mn and nm can be compared lexicographically in their converted strings because they have the same number of digits.

If there are n elements in the input array, the program sorts the array with method `Arrays.sort in` $O(n\log n)$ time. Therefore, it is much more efficient than the intuitive solution with $O(n!)$ time complexity.

A new rule to compare two numbers is defined in the solution above. Is the comparison rule valid? A valid comparison rule is reflective, symmetric and transitive. Let's prove it:

- Reflexivity. It is obvious that $aa=aa$, so $a \equiv a$.
- Symmetry. If $a \vee b$, $ab<ba$. Therefore, $ba>ab$, and $b \wedge a$.
- Transitivity. If $a \vee b$, $ab<ba$. Supposing that there are l and m digits in decimal numbers a and b. Therefore, $ab=a\times10^m+b$, and $ba=b\times10^l+a$.
  $ab<ba \rightarrow a\times10^m+b<b\times10^l+a \rightarrow a\times10^m-a<b\times10^l-b\ ) \rightarrow a\times(10^m-1)<b\times(10^l-1) \rightarrow a/(10^l-1)<b/(10^m-1)$

If $b \vee c$, $bc<cb$. Supposing that there are n digits in the decimal number c, it can be proven that $b/(10^m-1)<c/(10^n-1)$ with similar steps as above.

Because $a/(10^l-1)<b/(10^m-1)$ and $b/(10^m-1)<c/(10^n-1)$, $a/(10^l-1)<c/(10^n-1) \rightarrow a\times(10^n-1)<c\times(10^l-1) \rightarrow a\times10^n+c<c\times10^l+a \rightarrow ac<ca \rightarrow a \vee c$

We have demonstrated that our rule to compare two numbers fulfills the requirements of reflexivity, symmetry, and transitivity, so it is a valid comparison rule.

Let's move on to prove that the number concatenated from the array sorted with our comparison rule is a minimum. We are going to prove it by contradiction.

Supposing that n numbers in an array are sorted, and the concatenated number is $A_1A_2A_3 \ldots A_n$. If the concatenated number was not the minimum, there were two numbers x and y ($0<x<y\leq n$) at least. When x and y are swapped, $A_1A_2 \ldots A_y \ldots A_x \ldots A_n < A_1A_2 \ldots A_x \ldots A_y \ldots A_n$. Additionally, since n numbers are sorted with the comparison rule above, $A_x \vee A_{x+1} \vee A_{x+2} \ldots A_{y-2} \vee A_{y-1} \vee A_y$.

Because $A_{y-1} \vee A_y$, $A_{y-1}A_y<A_yA_{y-1}$. If A_{y-1} and A_y are swapped in $A_1A_2 \ldots A_{y-1}A_y \ldots A_n$, $A_1A_2 \ldots A_{y-1}A_y \ldots A_n<A_1A_2 \ldots A_yA_{y-1} \ldots A_n$. If numbers ahead of A_y are swapped with A_y, $A_1A_2 \ldots A_x \ldots A_{y-1}A_y \ldots A_n<A_1A_2 \ldots A_x \ldots A_yA_{y-1} \ldots A_n<A_1A_2 \ldots A_x \ldots A_yA_{y-2}A_{y-1} \ldots A_n< \ldots <A_1A_2 \ldots A_yA_x \ldots A_{y-2}A_{y-1} \ldots A_n$.

Similarly, $A_xA_{x+1}<A_{x+1}A_x$ because $A_x \vee A_{x+1}$. If numbers A_x and A_{x+1} are swapped in $A_1A_2 \ldots A_yA_xA_{x+1} \ldots A_{y-2}A_{y-1} \ldots A_n$, $A_1A_2 \ldots A_yA_xA_{x+1} \ldots A_{y-2}A_{y-1} \ldots A_n<A_1A_2 \ldots A_yA_{x+1}A_x \ldots A_{y-2}A_{y-1} \ldots A_n$. When numbers behind A_x are swapped with A_x, $A_1A_2 \ldots A_yA_xA_{x+1} \ldots A_{y-2}A_{y-1} \ldots A_n<A_1A_2 \ldots A_yA_{x+1}A_x \ldots A_{y-2}A_{y-1} \ldots A_n< \ldots <A_1A_2 \ldots A_yA_{x+1}A_{x+2} \ldots A_{y-2}A_{y-1}A_x \ldots A_n$.

Therefore, $A_1A_2 \ldots A_x \ldots A_y \ldots A_n<A_1A_2 \ldots A_y \ldots A_x \ldots A_n$, and it is contradicted to our assumption $A_1A_2 \ldots A_y \ldots A_x \ldots A_n<A_1A_2 \ldots A_x \ldots A_y \ldots A_n$. Our initial assumption must be false, and our algorithm is correct.

Source Code:

074_SortArrayForMinNumber.java

Test Cases:

- Functional Test Cases (Input an array with a few numbers)
- Boundary Test Cases (Input an array with only a number; numbers in an input array have duplicated digits)
- Performance Test Cases (The size of an input array is somewhat big, such as 50)