

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.

Knowledge Migration Skill


One of the learning skills that is tested is the ability to apply knowledge to new domains in order to learn new things and solve unprecedented problems. Usually, new technology is developed based on old one technology. Take learning programming languages as an example. If we understand the object-oriented mechanisms of C++, it is not difficult for us to learn Java and other object-oriented programming languages. Similarly, if we master the garbage collection mechanism of Java, it is easy to learn other managed languages such as C#.

Since the ability to migrate knowledge is important for software engineers, many IT companies pay attention to it when interviewing candidates. Interviewers like to ask coding questions that are actually variants of classic algorithms. Interviewers expect candidates to find the similarities between the problem and the classic algorithms, and solve the problem accordingly. For example, usually the binary search algorithm is used to find a number in a sorted array. How do you count an element in a sorted array (Question 83)? If we can find the first and last occurrence of the element in the sorted array with the binary search algorithm, we can get the count of the element.

Many candidates try lots of exercises during interview preparation. However, they cannot anticipate all coding interview problems, and it is unavoidable for them to meet new ones during interviews. Therefore, it is more important for them to summarize the characteristics of a solution when they solve a problem and try to reapply the solution when a similar problem is met. For example, in order to solve the problem to reverse the order of words in a sentence, all characters in the sentence are reversed and then characters in each word are reversed. The idea of reversing characters multiple times can be reapplied to rotate a string. Please refer to the section *Reversing Words and Rotating Strings* for more details.

As you have already seen, similar coding interview problems are grouped in a section in this book. Readers may analyze the similarities among a group of problems in order to improve the ability to draw inferences about other cases from a known instance.

Time of Occurrences in a Sorted Array

 **Question 83** Please implement a function to find how many times a number occurs in a sorted array.

For instance, the output is 4 when the inputs are an array {1, 2, 3, 3, 3, 3, 4, 5} and the number 3 because 3 occurs 4 times in the given array.

In a sorted array, it is natural to utilize the binary search algorithm. In order to count the occurrences of the number 3 in the given sample array, we can find any number with value 3 using the binary search and then scan its two sides sequentially to get the first 3 and the last 3. The target number may occur $O(n)$ times in an array with size n , so the time complexity is still $O(n)$ and it is not better than the linear search. It looks like the binary search algorithm does not help much here. Let's utilize the binary search algorithm thoroughly.

Assume the target value is k . Most of the time in the solution above is spent locating the first and last k . Let's explore more efficient solutions to locate the first and last k .

A better solution always compares the number m in the middle with the target k . If m is greater than k , k can only appear in the first half of the array, and the second half can be ignored in the next round of search. Similarly, k can only appear in the second half when m is less than k , and the first half can be ignored in the next round of search.

How do you continue searching when m equals k ? If the number prior to the middle one is not k , the middle number is the first k . If the number before the middle number is also k , the first k should be in the first half of the array, and the second half can be ignored in the next round of search.

It is easy to implement code to find the first k based on recursion, as shown in [Listing 8-1](#).

Listing 8-1. Java Code to Get the First k in a Sorted Array

```
int getFirst(int[] numbers, int start, int end, int k) {

    if(start > end)

        return -1;

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

    if(numbers[middle] == k) {

        if((middle > 0 && numbers[middle - 1] != k)

            || (middle == 0))

            return middle;

        end = middle - 1;

    }

    else if(numbers[middle] > k){

        end = middle - 1;

    }

    else {

        start = middle + 1;

    }

    return getFirst(numbers, start, end, k);

}
```

The method `getFirst` returns -1 if there is not a k in the array; otherwise, it returns the index of the first k .

It finds the last k utilizing a similar process. If the middle number m is greater than k , k can occur only in the first half of the array. If m is less than k , k can occur only in the second half. When m equals k , it checks whether the number next to the middle one is also k . If it is not, the number in the middle is the last k ; otherwise, the last k occurs in the second half of the array.

The recursive code to get the last k is shown in [Listing 8-2](#).

Listing 8-2. Java Code to Get the Last k in a Sorted Array

```
int getLast(int[] numbers, int start, int end, int k) {

    if(start > end)

        return -1;

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

    if(numbers[middle] == k) {

        if((middle < numbers.length - 1 && numbers[middle + 1] != k)

            || (middle == numbers.length - 1))

            return middle;

        start = middle + 1;

    }

    else if(numbers[middle] > k){

        end = middle - 1;

    }

    else {

        start = middle + 1;

    }

    return getLast(numbers, start, end, k);

}
```

Similar to `getFirst`, the method `getLast` returns -1 when there is not a k in the array; otherwise, it returns the index of the last k .

When the first k and last k are found, it gets the occurrence numbers of k based on their indexes, as shown in [Listing 8-3](#).

Listing 8-3. Java Code to Count k in a Sorted Array

```
int countOccurrence(int[] numbers, int k) {

    int first = getFirst(numbers, 0, numbers.length - 1, k);

    int last = getLast(numbers, 0, numbers.length - 1, k);

    int occurrence = 0;

    if(first > -1 && last > -1)

        occurrence = last - first + 1;

    return occurrence;

}
```

It utilizes the binary search algorithm to get the first and last k , which costs $O(\log n)$ time, so the overall time complexity of the method `countOccurrence` is $O(\log n)$.


Source Code:

083_Occurrence.java

Test Cases:

- Functional Test Cases (A sorted array with/without the target value; the target value occurs once or multiple times in a sorted array)
- Boundary Test Cases (The target value is at the beginning/end of the sorted array; all numbers in the array are duplicated; there is only one number in the array)

Application of Binary Tree Traversals

 **Question 84** How do you get the k^{th} node in a binary search tree in an incremental order of values? For example, the third node in the binary search tree in [Figure 8-1](#) is node 4.

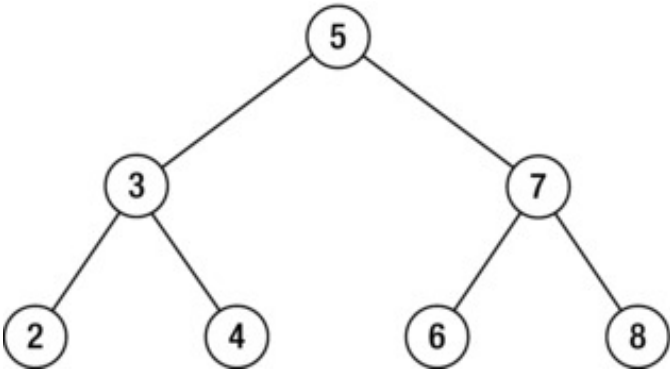


Figure 8-1. A sample binary search tree with seven nodes, where the third node has value 4

If a binary search tree is scanned with the in-order traversal algorithm, the traversal sequence is increasingly sorted. For example, the in-order traversal sequence of the binary tree in Figure 8-1 is {2, 3, 4, 5, 6, 7, 8}. Therefore, we can count the visited nodes during traversal and get the k^{th} node with the code in Listing 8-4.

Listing 8-4. C++ Code to Get the k^{th} Node in a Binary Search Tree

```
BinaryTreeNode* KthNode(BinaryTreeNode* pRoot, unsigned int k) {

    if(pRoot == NULL || k == 0)

        return NULL;

    return KthNodeCore(pRoot, k);

}

BinaryTreeNode* KthNodeCore(BinaryTreeNode* pRoot, unsigned int& k) {

    BinaryTreeNode* target = NULL;

    if(pRoot->pLeft != NULL)

        target = KthNodeCore(pRoot->pLeft, k);

    if(target == NULL) {

        if(k == 1)

            target = pRoot;

        k--;
    }

    if(target == NULL && pRoot->pRight != NULL)

        target = KthNodeCore(pRoot->pRight, k);

    return target;

}
```

Source Code:

084_KthNodeInBST.cpp

Test Cases:

- Normal Test Cases (Get the k^{th} node out of a normal binary search tree)
- Boundary Test Cases (The input k is 0, 1, or the number of nodes in the binary tree)
- Robustness Test Cases (The pointer to the root node is `NULL` ; special binary search trees, including those whose nodes only have right subtrees or left subtrees)

Question 85 How do you get the depth of a binary tree? Nodes from the root to a leaf form a path. Depth of a binary tree is the maximum length of all paths. For example, the depth of the binary tree in Figure 8-2 is 4, with the longest path through nodes 1, 2, 5, and 7.

We have discussed how to store nodes of a path in a stack while traversing a binary tree in the section *Paths in Binary Trees*. The depth of a binary tree is the length of the longest path. This solution works, but it is not the most concise one.

The depth of a binary tree can be gotten in another way. If a binary tree has only one node, its depth is 1. If the root node of a binary tree has only a left subtree, its depth is the depth of the left subtree plus 1. Similarly, its depth is the depth of the right subtree plus 1 if the root node has only a right subtree. What is the depth if the root node has both left subtree and right subtree? It is the greater value of the depth of the left and right subtrees plus 1.

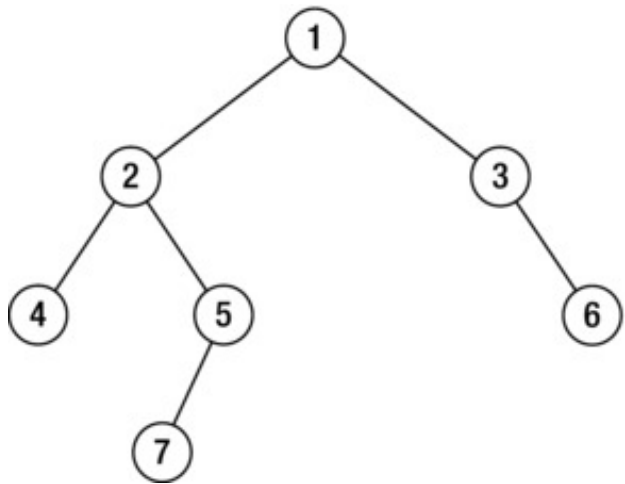


Figure 8-2. A binary tree with depth 4

For example, the root node of the binary tree in Figure 8-2 has both left and right subtrees. The depth of the left subtree rooted at node 2 is 3, and the depth of the right subtree rooted at node 3 is 2, so the depth of the whole binary tree is 4; 1 plus the greater value of 3 and 2.

It is easy to implement this solution recursively, with little modification on the post-order traversal algorithm, as shown in Listing 8-5.

Listing 8-5. C++ Code to Get Depth of a Binary Tree

```
int TreeDepth(BinaryTreeNode* pRoot) {  
  
    if(pRoot == NULL)  
  
        return 0;  
  
  
    int nLeft = TreeDepth(pRoot->m_pLeft);  
  
    int nRight = TreeDepth(pRoot->m_pRight);  
  
  
    return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);  
  
}
```

Source Code:

085_TreeDepth.cpp

Test Cases:

- Normal Test Cases (A normal binary tree with one level or multiple levels)
- Robustness Test Cases (The pointer to the root node is `NULL` ; special binary search trees, including those whose nodes only have right subtrees or left subtrees)

Question 86 How do you verify whether a binary tree is balanced? If the depth difference between a left subtree and right subtree of any node in a binary tree is not greater than 1, it is balanced. For instance, the binary tree in Figure 8-2 is balanced.

Visiting Nodes for Multiple Times

According to the definition of balanced binary trees, this problem can be solved by getting the depth difference between the left and right subtrees of every node. When a node is visited, the function `TreeDepth` is invoked to get the depth of its left and right subtrees. If the depth different is 1 at most for all nodes in a binary tree, it is balanced. This solution can be implemented based on the discussed in the preceding problem, as shown in Listing 8-6.

Listing 8-6. C++ Code to Verify Balanced Binary Trees (Version 1)

```
bool IsBalanced_Solution1(BinaryTreeNode* pRoot) {  
  
    if(pRoot == NULL)  
  
        return true;  
  
  
    int left = TreeDepth(pRoot->m_pLeft);  
  
    int right = TreeDepth(pRoot->m_pRight);  
  
    int diff = left - right;  
  
    if(diff > 1 || diff < -1)  
  
        return false;  
  
  
    return IsBalanced_Solution1(pRoot->m_pLeft)  
  
    && IsBalanced_Solution1(pRoot->m_pRight);  
  
}
```

This solution looks concise, but it is inefficient because it visits some nodes for multiple times. Take the binary tree in [Figure 8-2](#) as an example. When the function `TreeDepth` takes the node 2 as a parameter, it visits nodes 4, 5, and 7. When it verifies whether the binary tree rooted at node 2 is balanced, it visits nodes 4, 5, and 7 again. Obviously, we could improve performance if nodes are visited only once.

Visiting Every Node Only Once

If a binary tree is scanned with the post-order algorithm, its left and right subtrees are traversed before the root node. If we record the depth of the currently visited node (the depth of a node is the maximum length of paths from the node to its leaf nodes), we can verify whether the subtree rooted at the currently visited node is balanced. If any subtree is unbalanced, the whole tree is unbalanced.

This new solution can be implemented as shown in [Listing 8-7](#).

Listing 8-7. C++ Code to Verify Balanced Binary Trees (Version 2)

```
bool IsBalanced_Solution2(BinaryTreeNode* pRoot) {

    int depth = 0;

    return IsBalanced(pRoot, &depth);

}

bool IsBalanced(BinaryTreeNode* pRoot, int* pDepth) {

    if(pRoot == NULL) {

        *pDepth = 0;

        return true;

    }

    int left, right;

    if(IsBalanced(pRoot->m_left, &left)

        && IsBalanced(pRoot->m_right, &right)) {

        int diff = left - right;

        if(diff <= 1 && diff >= -1) {

            *pDepth = 1 + (left > right ? left : right);

            return true;

        }

    }

    return false;

}
```

After verifying left and right subtrees of a node, the solution verifies the subtree rooted at the current visited node and passes the depth to verify its parent node. When the recursive process returns to the root node finally, the whole binary tree is verified.

Source Code:

```
086_BalancedBinaryTree.cpp
```

Test Cases:

- Normal Test Cases (A normal binary tree is or is not balanced)
- Robustness Test Cases (The pointer to the root node is `NULL` ; special binary search trees, including those whose nodes only have right subtrees or left subtrees; there is only one node in a binary tree)

Sum in Sequences



Question 87 Given an increasingly sorted array and a number s , is there a pair of two numbers in the array whose sum is s ?

For example, if the inputs are an array {1, 2, 4, 7, 11, 15} and the number 15, there are two numbers, 4 and 11, whose sum is 15.

Let's first have a try selecting two numbers (n_1 and n_2) from the input array. If their sum equals to s , it is lucky because two required numbers have been found. If the sum is less than s , it replaces n_1 with its next number because the array is increasingly sorted and the next number of n_1 should be greater than n_1 . If the sum is greater than s , the number n_2 can be replaced with its preceding number in the sorted array, which should be less than n_2 .

Take the array {1, 2, 4, 7, 11, 15} and the number 15 as an example. At the first step, n_1 is the first number (also the least one) 1 and n_2 is the last number (also the greatest one) 15. It moves n_2 backward to the number 11 because their sum 16 is greater than 15.

At the second step, the two numbers are 1 and 11, and their sum, 12, is less than 15. Therefore, it moves the n_1 forward and n_1 becomes 2.

The two numbers are 2 and 11 at the third step. Since their sum, 13, is less than 15, it moves n_1 forward again.

Now the two numbers are 4 and 11 and their sum is 15, which is the expected sum.

The process is summarized in [Table 8-1](#).

Table 8-1. The Process to Find a Pair of Numbers Whose Sum Is 15 out of the Array {1, 2, 4, 7, 11, 15}

Step	n_1	n_2	$n_1 + n_2$	Comparing $n_1 + n_2$ with s	Operation
1	1	15	16	Greater	Select the preceding number of n_2
2	1	11	12	Less	Select the next number of n_1
3	2	11	13	Less	Select the next number of n_1
4	4	11	15	Equal	-

It is not difficult to write code with the detailed analysis above, as shown in [Listing 8-8](#).

Listing 8-8. Java Code to Get a Pair with a Sum

```
boolean hasPairWithSum(int numbers[], int sum) {  
  
    boolean found = false;  
  
  
    int ahead = numbers.length - 1;  
    int behind = 0;  
  
    while(ahead > behind) {  
  
        int curSum = numbers[ahead] + numbers[behind];  
  
  
        if(curSum == sum) {  
            found = true;  
            break;  
        }  
  
        else if(curSum > sum)  
            ahead --;  
  
        else  
            behind ++;  
    }  
  
    return found;  
}
```


In this code, `ahead` is the index of n_2 , and `behind` is the index of n_1 . The time complexity is $O(n)$ for an array with n elements because it only scans the input array once.

Source Code:

087_TwoNumbersWithSum.java

Test Cases:

- Normal Test Cases (An array with/without a pair of numbers whose sum is the target sum)
- Boundary Test Cases (There is only one element in the array; there are two elements in the array)

 **Question 88** Given an array, please check whether it contains three numbers whose sum equals 0.

This problem is also required to find some numbers with a given array and sum, so it is similar to the previous problem. We may get some hints from the solution above.

The solution above is based on an increasingly sorted array, so first, the input array is sorted increasingly, too. Second, the sorted array is scanned from beginning to end. When the i^{th} number with value a_i is scanned, we try to find a pair of numbers whose sum is $-a_i$ in the array excluding the i^{th} number.

Let's modify the method `hasPairWithSum` above, as demonstrated in [Listing 8-9](#).

Listing 8-9. Java Code to Get a Pair with a Sum Excluding a Number

```
boolean hasPairWithSum(int numbers[], int sum, int excludeIndex) {  
  
    boolean found = false;  
  
  
    int ahead = numbers.length - 1;  
    int behind = 0;  
  
    while(ahead > behind) {
```

```

    if(ahead == excludeIndex)

        ahead--;

    if(behind == excludeIndex)

        behind++;

    int curSum = numbers[ahead] + numbers[behind];

    if(curSum == sum) {

        found = true;

        break;

    }

    else if(curSum > sum)

        ahead --;

    else

        behind ++;

}

return found;

}

```

It checks whether there are two numbers whose sum is `sum` in `numbers` excluding the number with index `excludeIndex`. It then checks whether there are three numbers in an array with sum 0 with the method in [Listing 8-10](#).

Listing 8-10. Java Code to Get Three Numbers with Sum 0

```

boolean hasTripleWithSum0(int numbers[]) {

    boolean found = false;

    if(numbers.length < 3)

        return found;

    Arrays.sort(numbers);

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

        int sum = -numbers[i];

        found = hasPairWithSum(numbers, sum, i);

        if(found)

            break;

    }

    return found;

}

```


It contains two steps in the function `hasTripleWithSum0`. It costs $O(n \log n)$ time to sort n numbers in its first step. At the second step, it costs $O(n)$ time for each number to call `hasPairWithSum`, so it costs $O(n^2)$ time in the `for` loop. Therefore, its overall time complexity is $O(n^2)$.

Source Code:

088_ThreeNumbersWithSum.java

Test Cases:

- Normal Test Cases (An array with/without three numbers whose sum is 0)
- Boundary Test Cases (There are only two or three elements in the array)

 **Question 89** Given an array, please check whether it contains a subset of numbers (with one number at least) whose sum equals 0.

A subset of an array is a combination of numbers in the array. We have discussed two different algorithms to get combinations of characters in a string in the section *Permutations and Combinations*. It is a similar process to get combinations of an array. [Listing 8-11](#) shows the sample code based on bit operations to get subsets of an array.

Listing 8-11. Java Code to Get a Subset with Sum 0


```
boolean hasSubsetWithSum0(int numbers[]) {  
  
    BitSet bits = new BitSet(numbers.length);  
  
    while(increment(bits, numbers.length)) {  
  
        int sum = 0;  
  
        boolean oneBitAtLeast = false;  
  
        for(int i = 0; i < numbers.length; ++i) {  
  
            if(bits.get(i)) {  
  
                if(!oneBitAtLeast)  
  
                    oneBitAtLeast = true;  
  
  
                sum += numbers[i];  
  
            }  
  
        }  
  
  
        if(oneBitAtLeast && sum == 0)  
  
            return true;  
  
    }  
  
    return false;  
}
```

The method `increment` is the same as the method in the section *Permutations and Combinations* to get combinations of a string.
Source Code:

089_SubsetWithSum.java

Test Cases:

- An array with one, two, three, four, or more numbers whose sum is 0
- The sum of all numbers in an array is 0
- There is not a subset of numbers whose sum is 0

 **Question 90** Given a positive value s , print all sequences with continuous numbers (with two numbers at least) whose sum is s . Take the input $s=15$ as an example. Because $1+2+3+4+5=4+5+6=7+8=15$, three continuous sequences should be printed: 1~5, 4~6, and 7~8.

A continuous sequence is specified with two numbers at its two ends. The least number in the sequence is denoted as *small*, and the greatest number is denoted as *big*. At first, *small* is initialized as 1, and *big* is initialized as 2. If the sum of numbers from *small* to *big* is less than the given s , *big* is increased to include more numbers; otherwise, *small* is increased to exclude some numbers if the sum of numbers in the sequence is greater than s . Since there are two numbers at least in the sequence, the process stops when *small* is $(1 + s)/2$.

Let's take the sum 9 as an example. First, *small* is initialized as 1, and *big* is initialized as 2. The sequence between *small* and *end* is {1, 2}, and the sum of sequence is 3, which is less than 9, so more numbers should be included into the sequence. *big* is increased to 3, and there are three numbers {1, 2, 3} in the sequence. The sum of the sequence is 6, and it is still less than 9, so *big* is increased again. The sum of the sequence {1, 2, 3, 4} is 10, and it is greater than 9, so *small* is increased to exclude a number. The sum of sequence {2, 3, 4} is 9, which is equal to the given sum, so the sequence should be printed.

In order to get other sequences, *big* is increased again. Similarly to the steps above, another sequence {4, 5} with sum 9 can be found. The whole process is summarized in Table 8-2.

Table 8-2. The Process to Find Ccontinuous Sequences with Sum 9

Step	<i>small</i>	<i>big</i>	Sequence	Sum	Comparison	Next Step
1	1	2	1, 2	3	Less	Increasing <i>big</i>
2	1	3	1, 2, 3	6	Less	Increasing <i>big</i>
3	1	4	1, 2, 3, 4	10	Greater	Increasing <i>small</i>
4	2	4	2, 3, 4	9	Equal	Printing, and Increasing <i>big</i>
5	2	5	2, 3, 4, 5	14	Greater	Increasing <i>small</i>
6	3	5	3, 4, 5	12	Greater	Increasing <i>big</i>
7	4	5	4, 5	9	Equal	Printing

It is time to write code after we have clear ideas about the process. The sample code is found in [Listing 8-12](#).

Listing 8-12. Java Code to Get a Sequence with a Given Sum

```
void findContinuousSequence(int sum) {

    if(sum < 3)

        return;

    int small = 1;

    int big = 2;

    int middle = (1 + sum) / 2;

    int curSum = small + big;

    while(small < middle) {

        if(curSum == sum)

            printContinuousSequence(small, big);

        while(curSum > sum && small < middle) {

            curSum -= small;

            ++small;

            if(curSum == sum)

                printContinuousSequence(small, big);

        }

        ++big;

        curSum += big;

    }

}

void printContinuousSequence(int small, int big){

    for(int i = small; i <= big; ++i)

        System.out.print(String.valueOf(i) + " ");

    System.out.println("");

}
```

There is a detail in the code in [Listing 8-12](#) that is worthy of attention. Usually, it takes $O(n)$ time to calculate the sum of a sequence with n numbers. However, it only costs $O(1)$ time here. Only one number is inserted or removed at each step, so all numbers except one in a sequence are the same compared with the sequence at the preceding step. Therefore, the sum of the sequence can be gotten based on the sum of the preceding sequence with fewer calculations.


Source Code:

090_ContinuousSequenceWithSum.java

Test Cases:

- Functional Test Cases (The sum s with only one continuous sequence, such as 6; the sum s with multiple continuous sequences, such as 9 and 100; the sum s without a continuous sequence, such as 4)
- Boundary Test Cases (The minimum sum 3 with a continuous sequence)

Reversing Words and Rotating Strings

 **Question 91** How do you reverse the order of words in a sentence, but keep words themselves unchanged? Words in a sentence are separated by blanks. For instance, the reversed output should be “student. a ma I” when the input is “I am a student.”.

This is a very popular interview question. It can be solved with two steps. First, we reverse all characters in a sentence. If all characters in the sentence “I am a student.” are reversed, they become “.tneduts a ma I”. Not only the order of words is reversed, but also the order of characters inside each word is reversed. Secondly, we reverse characters in every word. We can get “student. a ma I” from the example input string with these two steps.

The key of our solution is to implement a function to reverse a string, which is shown as the `Reverse` function in [Listing 8-13](#).

Listing 8-13. C++ Code to Reverse a Segment in a String

```
void Reverse(char *pBegin, char *pEnd) {

    if(pBegin == NULL || pEnd == NULL)
```

```

    return;

    while(pBegin < pEnd) {

        char temp = *pBegin;

        *pBegin = *pEnd;

        *pEnd = temp;

        pBegin ++, pEnd --;

    }

}

```

Now we can reverse the whole sentence and each word based on this `Reverse` function with the code in [Listing 8-14](#).

Listing 8-14. C++ Code to Reverse Words in a Sentence

```

char* ReverseSentence(char *pData) {

    if(pData == NULL)

        return NULL;

    char *pBegin = pData;

    char *pEnd = pData;

    while(*pEnd != '\0')

        pEnd ++;

    pEnd--;

    // Reverse the whole sentence

    Reverse(pBegin, pEnd);

    // Reverse every word in the sentence

    pBegin = pEnd = pData;

    while(*pBegin != '\0') {

        if(*pBegin == ' ') {

            pBegin ++;

            pEnd ++;

        }

        else if(*pEnd == ' ' || *pEnd == '\0') {

            Reverse(pBegin, --pEnd);

            pBegin = ++pEnd;

        }

        else {

            pEnd ++;

        }

    }

    return pData;

}

```

Since words are separated by blanks, we can get the beginning position and ending position of each word by scanning blanks. In the second phase to reverse each word in the previous sample code, the pointer `pBegin` points to the first character of a word, and `pEnd` points to the last character.

Source Code:

091_ReverseWordsInSentence.cpp

Test Cases:

- Functional Test Cases (A sentence with multiple words)
- Boundary Test Cases (A sentence with only a word; an empty string; a sentence with only blanks)
- Robustness Test Cases (The pointer to the input string is `NULL`)

Question 92 Left rotation of a string is to move some leading characters to its end. Please implement a function to rotate a string to the left. For example, if the input string is “abcde~~fg~~” and a number 2, two characters are rotated and the result is “cde~~fg~~ab”.

It looks difficult to get rules of left rotation on a string. Fortunately, we get some hints from the solution of the previous problem.

If the input string of the preceding problem is “hello world”, the reversed result should be “world hello”. Note that the result “world hello” can be viewed as a rotated result of “hello world”. It becomes “world hello” when we move some leading characters of the string “hello world” to its end. Therefore, this problem is quite similar to the previous problem.

Let’s take a string “abcde~~fg~~” as another example. We divide it into two parts: the first part contains the two leading characters “ab”, and the second part contains all other characters “cde~~fg~~”. We first reverse these two parts separately, and the whole string becomes “bag~~f~~edc”. It becomes “cde~~fg~~ab” if the whole string is reversed, which is the expected result of left rotation of the string “abcde~~fg~~” with two characters.

According to the analysis above, we can see that left rotation of a string can be implemented by invoking the `Reverse` function three times to reverse a segment or the whole string. The sample code is shown in [Listing 8-15](#).

Listing 8-15. C++ Code for Left Rotation in a String

```
char* LeftRotateString(char* pStr, int n) {
    if(pStr != NULL) {
        int nLength = static_cast<int>(strlen(pStr));

        if(nLength > 0 && n > 0 && n < nLength) {
            char* pFirstStart = pStr;
            char* pFirstEnd = pStr + n - 1;
            char* pSecondStart = pStr + n;
            char* pSecondEnd = pStr + nLength - 1;

            // Reverse the n leading characters
            Reverse(pFirstStart, pFirstEnd);

            // Reverse other characters
            Reverse(pSecondStart, pSecondEnd);

            // Reverse the whole string
            Reverse(pFirstStart, pSecondEnd);
        }
    }

    return pStr;
}
```

Source Code:

092_LeftRotateString.cpp

Test Cases:

- Functional Test Cases (Rotate a string with n characters to the left side for $0/1/2/n-1/n/n+1$ characters respectively)
- Robustness Test Cases (The pointer to the input string is `NULL`)

Maximum in a Queue

Question 93 Given an array of numbers and a sliding window size, how do you get the maximum numbers in all sliding windows?

For example, if the input array is {2, 3, 4, 2, 6, 2, 5, 1} and the size of the sliding windows is 3, the output of maximums are {4, 4, 6, 6, 6, 5}, as illustrated in [Table 8-3](#).

Table 8-3. Maximums of All Sliding Windows with Size 3 in an Array {2, 3, 4, 2, 6, 2, 5, 1}. A pair of Brackets Indicates a Sliding Window.

Sliding Windows in an Array	Maximums in Sliding Windows
[2, 3, 4], 2, 6, 2, 5, 1	4
2, [3, 4, 2], 6, 2, 5, 1	4
2, 3, [4, 2, 6], 2, 5, 1	6
2, 3, 4, [2, 6, 2], 5, 1	6
2, 3, 4, 2, [6, 2, 5], 1	6
2, 3, 4, 2, 6, [2, 5, 1]	5

It is not difficult to get the brute-force solution: scan numbers in every sliding window to get the maximum value. The overall time complexity is $O(nk)$ if the length of array is n and the size of the sliding windows is k .
The naive solution is not the best solution. Let's explore alternatives.

Maximum Value in a Queue

A window can be viewed as a queue. When it slides, a number is pushed into its end and the number at its beginning is popped off. Therefore, the problem is solved if we can get the maximum value of a queue.

There are no straightforward approaches to getting the maximum value of a queue. However, there are solutions to get the maximum value of a stack, which is similar to the solution to "Stack with Min Function." Additionally, a queue can also be implemented with two stacks (details are discussed in the section *Build a Queue with Two Stacks*).

If a new type of queue is implemented with two stacks in which a function `max` is defined to get the maximum value, the maximum value in a queue is the greater number of the two maximum numbers in the two stacks.

This solution works. However, we may not have enough time to write all code to implement data structures for our own queue and stack during interviews. Let's continue exploring a more concise solution.

Saving the Maximum Value into a Queue

Instead of pushing every number inside a sliding window into a queue, we try to only push the candidates of maximum into a double-ended queue. Let's take the array {2, 3, 4, 2, 6, 2, 5, 1} as an example to analyze the solution step-by-step.

The first number in the array is 2, and it is pushed into a queue. The second number is 3, which is greater than the previous number, 2. The number 2 should be popped off because it is less than 3 and it has no chance to be the maximum value. There is only one number left in the queue when 2 is removed and then 3 is inserted. The operations are similar when the next number, 4, is inserted. There is only a number 4 remaining in the queue. Now the sliding window already has three elements and the maximum value is at the beginning of the queue.

Table 8-4. The Process to Get the Maximum Number in all Sliding Windows with Window Size 3 in the Array {2, 3, 4, 2, 6, 2, 5, 1}. In the Column Indexes in Queue, the Number inside a Pair of Parentheses Is the Number Indexed by the Number before It in the Array.

Step	Pushed Number	Sliding Window	Indexes in Queue	Maximum
1	2	2	0(2)	N/A
2	3	2, 3	1(3)	N/A
3	4	2, 3, 4	2(4)	4
4	2	3, 4, 2	2(4), 3(2)	4
5	6	4, 2, 6	4(6)	6
6	2	2, 6, 2	4(6), 5(2)	6
7	5	6, 2, 5	4(6), 6(5)	6
8	1	2, 5, 1	6(5), 7(1)	5

We continue to push the fourth number. It is pushed to the end of queue because it is less than the previous number 4 and it might be a maximum number in the future when the previous numbers are popped off. There are two numbers, 4 and 2, in the queue, and 4 is the maximum.

The next number to be pushed is 6. Since it is greater than the existing numbers 4 and 2, these two numbers can be popped off because they have no chance of being the maximum. Now there is only one number in the queue, which is 6, after the current number is pushed. Of course, the maximum is 6.

The next number is 2, which is pushed to the end of the queue because it is less than the previous number 6. There are two numbers in the queue, 6 and 2, and the number 6 at the beginning of the queue is the maximum value.

It is time to push the number 5. Because it is greater than the number 2 at the end of the queue, 2 is popped off and then 5 is pushed. There are two numbers in the queue, 6 and 5, and the number 6 at the beginning of the queue is still the maximum value.

Now, let's push the last number 1. It can be pushed into the queue. Note that the number 6 at the beginning of the queue is beyond the scope of the current sliding window, and it should be popped off. How do we know whether the number at the beginning of the queue is out of the sliding window? Rather than storing numbers in the queue directly, we can store indexes instead. If the distance between the index at the beginning of queue and the index of the current number to be pushed is greater than or equal to the window size, the number corresponding to the index at the beginning of queue is out of the sliding window.

The analysis process above is summarized in Table 8-4. Note that maximum numbers of all sliding windows are always indexed by the beginning of the queue.

We can implement a solution based on this analysis. Some sample code in C++ is shown in Listing 8-16 that utilizes the type `deque` in the STL.

Listing 8-16. C++ Code for Maximums in Sliding Windows

```

vector<int> maxInWindows(const vector<int>& numbers, int windowSize) {
    vector<int> maxInSlidingWindows;

    if(numbers.size() >= windowSize && windowSize >= 1) {
        deque<int> indices;

        for(int i = 0; i < windowSize; ++i) {
            while(!indices.empty() && numbers[i] >= numbers[indices.back()])
                indices.pop_back();

            indices.push_back(i);
        }

        for(int i = windowSize; i < numbers.size(); ++i) {
            maxInSlidingWindows.push_back(numbers[indices.front()]);

            while(!indices.empty() && numbers[i] >= numbers[indices.back()])
                indices.pop_back();

            if(!indices.empty() && indices.front() <= i - windowSize)
                indices.pop_front();

            indices.push_back(i);
        }

        maxInSlidingWindows.push_back(numbers[indices.front()]);
    }

    return maxInSlidingWindows;
}


```

Source Code:

093_MaxInSlidingWindows.cpp

Test Cases:

- Functional Test Cases (Various n and k where n is the size of the array and k is the size of sliding windows; various arrays, including increasingly and decreasingly sorted arrays)
- Boundary Test Cases (The size of sliding window k is 1, or the same as the size of the array; the size of sliding window k is less than 1, or greater than the size of the array)

 **Question 94** Define a queue in which we can get its maximum number with a function `max`. In this stack, the time complexity of `max`, `push_back`, and `pop_front` are all $O(1)$.

As we mentioned before, a sliding window can be viewed as a queue. Therefore, we can implement a new solution to get the maximum value of a queue based on the second solution to get the maximums of sliding windows.

[Listing 8-17](#) shows the sample code.

Listing 8-17. C++ Code to Get Maximum in Queue

```

template<typename T> class QueueWithMax {
public:
    QueueWithMax(): currentIndex(0) {
    }

    void push_back(T number) {
        while(!maximums.empty() && number >= maximums.back().number)
            maximums.pop_back();

        InternalData internalData = {number, currentIndex};
        data.push_back(internalData);
    }
}

```

```

        maximums.push_back(internalData);

        ++currentIndex;
    }

    void pop_front() {
        if(maximums.empty())
            throw new exception("queue is empty");

        if(maximums.front().index == data.front().index)
            maximums.pop_front();

        data.pop_front();
    }

    T max() const {
        if(maximums.empty())
            throw new exception("queue is empty");

        return maximums.front().number;
    }

private:
    struct InternalData {
        T number;
        int index;
    };

    deque<InternalData> data;
    deque<InternalData> maximums;
    int currentIndex;
};

```

Since this solution is similar to the second solution to get maximums of sliding windows, we will not analyze the process step-by-step but leave it as an exercise if you are interested.

Source Code:

094_QueueWithMax.cpp

Test Cases:

- Insert elements into an empty queue and then delete them
- Insert elements into a non-empty stack and then delete them
- Push and pop multiple elements continuously