CHAPTER 8

Skills for Interviews

Interviews are chances for candidates to demonstrate their communication skills. Sometimes interviewers ask coding questions with new concepts to verify candidates' learning skills. Candidates might solve new interview problems with skills that reapply solutions to classic problems or solve problems abstracted from daily life and work with mathematical modeling skills. Divergent thinking skills are necessary for candidates when conventional solutions are disallowed by interviewers. All of these skills are discussed in detail with several sample coding interview questions in this chapter.

Communication and Learning Skills

Communications Skills

Good communication skills are essential for career success. Engineers have to communicate with developers, testers, and project managers during their daily work, so communication greatly impacts productivity. That is why interviewers like to evaluate candidates' communication skills, which include the following types:

- *Verbal Communication.* If a candidate is able to express his or her project experience and solutions to coding interview problems clearly, confidently, and concisely, he or she demonstrates effective spoken communication skills. Note that communication is not talkativeness with empty words. Candidates should be honest and sincere during interviews.
- *Nonverbal Communication.* Body language means a lot during communications. It is a good practice for a candidate to face his or her interviewer with open, attentive, upright posture and maintain good eye contact (look at the interviewer a lot, but don't stare all the time), smile, and nod from time to time.
- *Listening and asking questions actively.* When candidates are asked a question, they should reflect on the question and state in their own words what they understand the question to be. Sometimes interviewers do not provide enough information about their questions, and they wait for candidates to ask for more clarification. The process of asking for clarification and eliminating misunderstanding is a good indicator of a candidate's communication skills.

H. He, *Coding Interviews*

■ **Tip** An interview is a bidirectional communication. Interviewers ask candidates questions and candidates can also ask interviewers questions. The ability to ask questions to get more information and eliminate misunderstanding is a demonstration of effective communication skills.

Learning Skills

Technology has been changing at a fast pace for decades. Software engineers have to have strong learning skills or they will not be able to keep up with technology progress. Outstanding engineers will only stay sharp if they have strong learning ability and aspirations to continuously learn about and evaluate new technologies. Therefore, interviewers in IT companies pay a lot of attention to candidates' learning abilities.

There are two strategies available for interviewers to examine candidates' learning skills. The first one is to ask candidates what books they have recently read or what they learned from their most recent project. If an engineer is eager to learn new things at all times, he or she has learning skills that might be an advantage over other candidates, that is, from the interviewer's perspective.

The other strategy is to ask questions about some new concepts and examine whether a candidate can absorb them in a short period of time. There are many new concepts about coding interview questions covered in this book, such as rotation of arrays, mirrored images of a binary trees, ugly numbers, and reversed pairs in an array. A candidate demonstrates his or her learning skills if he can master new concepts in a few minutes and then solve the problems.

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.

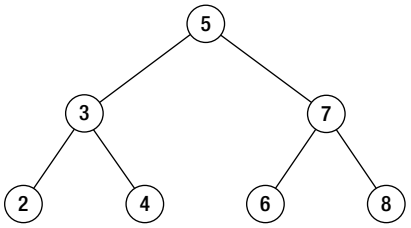

Figure 8-1. *A sample binary search tree with seven nodes, where the third node has value 4*

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* kth *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.

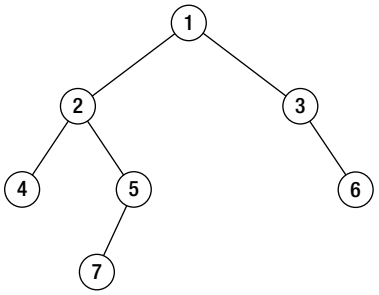

Figure 8-2. *A binary tree with depth 4*

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 a 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 `Depth` 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 `TreeDepth` 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_pLeft, &left)
        && IsBalanced(pRoot->m_pRight, &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	*small*	*big*	Sequence	Sum	Comparison	Next Step
1	1	2	1, 2	3	Less	Increasing *big*
2	1	3	1, 2, 3	6	Less	Increasing *big*
3	1	4	1, 2, 3, 4	10	Greater	Increasing *small*
4	2	4	2, 3, 4	9	Equal	Printing, and Increasing *big*
5	2	5	2, 3, 4, 5	14	Greater	Increasing *small*
6	3	5	3, 4, 5	12	Greater	Increasing *big*
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_ContinousSequenceWithSum.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 am 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 am 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 "abcdefg" and a number 2, two characters are rotated and the result is "cdefgab".

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 "abcdefg" 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 "cdefg". We first reverse these two parts separately, and the whole string becomes "bagfedc". It becomes "cdefgab" if the whole string is reversed, which is the expected result of left rotation of the string "abcdefg" 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 $0/1/2/n\text{-}1/n/n{+}1$ 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 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

Mathematical Modeling Skill

The computer is a tool to solve problems in our daily life and work, and the duty of programmers is to abstract mathematical models from real problems and solve them with programming languages. Therefore, the mathematic modeling skill is important for software engineers.

Candidates should select the appropriate data structure to model a problem. The problems in our life and work are various, but there are only a few common data structures. Candidates could make choices according to properties of the problem as well as performance and difficulties of development. For instance, candidates may model a circle with a set of numbers as a looped linked list in order to solve the problem "Last Number in a Circle" (Question 96).

Candidates should also analyze the hidden rules or patterns inside the problem and implement them with appropriate algorithms. For example, the interview problem of "Probabilities of Dice Points" (Question 95) is essentially equivalent to calculate a sequence of $f(n)=f(n\text{-}1)+f(n\text{-}2)+f(n\text{-}3)+f(n\text{-}4)+f(n\text{-}5)+f(n\text{-}6)$. Additionally, in order to find the minimum number of moves to sort a set of cards, we have to find the longest increasing subsequence at first. If candidates are familiar with algorithms, they should be able to find the longest increasing subsequence with dynamic programming.

Probabilities of Dice Points

■ **Question 95** Given a number n, which stands for n dice, please print probabilities of all possible sums of dice points.

For example, if there are two dice, there are two ways to get three points: (1) one die has one point, and the other has two points; and (2) one die has two points, and the other has one. However, there is only one way to get twelve points, with six points on each die. Therefore, the possibility to get three points is higher than to get twelve points with two dice.

A die is often a rounded cube, with each of its faces showing a different number from 1 to 6. The minimum sum of n dice is n, and the maximum is $6n$. Additionally, points shown by n dice have 6^n permutations.

A function $f(i)$ is defined for the number of situations for dice to show i points. When we get $f(i)(0<n\leq 6n)$, the probability to get i points with n dice is $f(i)/6^n$.

When there is only one die, $f(1)$, $f(2)$, …, $f(6)$ are initialized to 1. A die is added at each step. If there are $f(i)$ situations to get i points at a certain step, there are $f(i)=f(i\text{-}1)+f(i\text{-}2)+f(i\text{-}3)+f(i\text{-}4)+f(i\text{-}5)+f(i\text{-}6)$ with one more die at the next step. This is a typical recursive equation. However, the complexity grows exponentially if it is solved recursively because there are overlapped calculations. A better choice is

based on iteration. More discussions about the efficiency difference between recursion and iteration are available in the section *Fibonacci Sequence*.

The iterative solution can be implemented as shown in Listing 8-18.

Listing 8-18. *C++ Code to Get Probabilities of Dice Points*

```
void PrintProbability(int number) {
    if(number < 1)
        return;

    const int maxValue = 6;
    int* pProbabilities[2];
    pProbabilities[0] = new int[maxValue * number + 1];
    pProbabilities[1] = new int[maxValue * number + 1];
    for(int i = 0; i < maxValue * number + 1; ++i) {
        pProbabilities[0][i] = 0;
        pProbabilities[1][i] = 0;
    }

    int flag = 0;
    for (int i = 1; i <= maxValue; ++i)
        pProbabilities[flag][i] = 1;

    for (int k = 2; k <= number; ++k) {
        for(int i = 0; i < k; ++i)
            pProbabilities[1 - flag][i] = 0;

        for (int i = k; i <= maxValue * k; ++i) {
            pProbabilities[1 - flag][i] = 0;
            for(int j = 1; j <= i && j <= maxValue; ++j)
                pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];
        }

        flag = 1 - flag;
    }

    double total = pow((double)maxValue, number);
    for(int i = number; i <= maxValue * number; ++i) {
        double ratio = (double)pProbabilities[flag][i] / total;
        printf("%d: %e\n", i, ratio);
    }

    delete[] pProbabilities[0];
    delete[] pProbabilities[1];
}
```

Two arrays, `pProbabilities[0]` and `pProbabilities[1]`, are defined in the code above. The i^{th} element in these two arrays stands for the number of situations for dice to show i points. These two arrays are swapped (modifying the variable `flag`) for the next step to add one more die.

Source Code:

`095_DicesProbability.cpp`

Test Cases:

- Normal Test Cases (Probabilities of points for 1, 2, 3, 4 dice)
- Boundary Test Cases (Input 0)
- Performance Test Cases (Somewhat big numbers for the count of dice, such as 11)

Last Number in a Circle

■ **Question 96** A circle is composed of *n* numbers, 0, 1, …, *n*-1. The m^{th} number is removed every time, counting from the number 0 for the first removal. What is the last number left?
For example, a circle is composed of five numbers 0, 1, 2, 3, and 4 (Figure 8-3). If the third number is removed from the circle repeatedly, four numbers are deleted in the sequence of 2, 0, 4, and 1, and the last number remaining in the circle is 3.

This is the classic Josephus problem, and there are two solutions for it. The first solution is to simulate a circle with a looped list, and the other one is to analyze the pattern of deleted numbers at each step.

Simulating a Circle with a Looped List

Since this problem is about a circle, an intuitive solution is to simulate the circle with a looped list. A loop with *n* nodes is created first, and then the m^{th} node is deleted from it at each step.

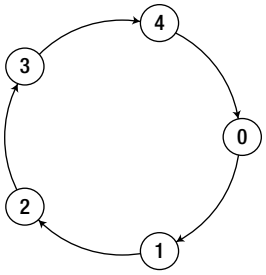

Figure 8-3. *A circle with five numbers*

The code in Listing 8-19 is based on the type `list` in the C++ standard template library. An instance of `std::list` is not looped. When the iterator reaches the tail node, it returns back to the head node, so the list is traversed in a circled sequence. This solution can be implemented as the code in C++ as shown in Listing 8-19.

Listing 8-19. *C++ Code for the Last Number Remaining in a Circle (Version 1)*

```
int LastRemaining_Solution1(unsigned int n, unsigned int m) {
    if(n < 1 || m < 1)
        return -1;

    list<int> numbers;
    for(unsigned int i = 0; i < n; ++ i)
        numbers.push_back(i);

    list<int>::iterator current = numbers.begin();
    while(numbers.size() > 1) {
        for(int i = 1; i < m; ++ i) {
            current ++;
            if(current == numbers.end())
                current = numbers.begin();
        }

        list<int>::iterator next = ++ current;
        if(next == numbers.end())
            next = numbers.begin();

        -- current;
        numbers.erase(current);
        current = next;
    }

    return *(current);
}
```

This solution has to traverse the circled list many times. It costs O(m) time to delete a node, so its overall complexity is O(mn) on a circle with n nodes originally. Additionally, it consumes O(n) space to create a looped list. Let's explore a more effective solution.

Analyzing the Pattern of Deleted Numbers

A function $f(n,m)$ is defined for the last remaining number in a circle with n numbers from 0 to n-1 when the m^{th} number is deleted at every step.

Among the n numbers in the circle, the first deleted number is (m-1)%n. (m-1)%n is denoted as k for simplicity. When k is deleted, there are n-1 numbers (0, 1, …, k-1, k+1, …, n-1) left, and it counts from k+1 to delete the next m^{th} number. Therefore, numbers can be reordered as k+1, …, n-1, 0, 1, …, k-1 in order to place k+1 as the first number. The pattern of the reordered sequence is different from the original sequence since it does not start from 0. Another function $f'(n\text{-}1,m)$ is defined for the last remaining number in these n-1 numbers, and $f(n,m)=f'(n\text{-}1,m)$.

The sequence with n-1 numbers k+1, …, n-1, 0, 1, …, k-1 can be projected to a new sequence from 0 to n-2:

$k+1 \rightarrow 0$
$k+2 \rightarrow 1$

…
n-1 → n-k-2
0 → n-k-1
1 → n-k
…
k-1 → n-2

If the projection is defined as p, $p(x)=(x-k-1)\%n$. The reversed projection is $p^{-1}(x)=(x+k+1)\%n$.

Since the projected sequence starts from 0 and has a pattern similar to the original sequence from 0 to n-1, the last number left in the projected sequence with n-1 numbers also follows the function f, and it is $f(n-1,m)$.

Therefore, the last number remaining in the sequence before the projection is $f(n-1,m)=p^{-1}[f(n-1,m)]=[f(n-1,m)+k+1]\%n$. Because $k=(m-1)\%n$, $f(n,m)=f(n-1,m)=[f(n-1),m)+m]\%n$.

We have found a recursive equation to get the last remaining number from n numbers when the m^{th} number is deleted at each step. If there is only one number in the original circle, it is obvious that the last remaining number is 0. Therefore, $f(n,m)$ is defined as follows:

$$f(n,m)=\begin{cases}0 & n=1\\ (f(n-1,m)+m)\%n & n>1\end{cases}$$

The equation can be implemented easily with both recursion and iteration. The code in Listing 8-20 is based on iteration.

Listing 8-20. *C++ Code for the Last Number Remaining in a Circle (Version 2)*

```
int LastRemaining_Solution2(unsigned int n, unsigned int m) {
    if(n < 1 || m < 1)
        return -1;

    int last = 0;
    for (int i = 2; i <= n; i ++)
        last = (last + m) % i;

    return last;
}
```

Even though this analysis is very complicated, the implementation code looks quite concise. More importantly, it is more efficient because its time complexity is O(n) and space complexity is O(1).

Source Code:

```
096_LastNumberInCircle.cpp
```

Test Cases:

- Normal Test Cases (The input m is less than n, such as to delete the 3rd number at each step from a circle with 5 numbers originally; the input m is greater than or equal to n, such as to delete the 6th or 7th number at each step from a circle with 6 numbers originally)
- Boundary Test Cases (The circle is empty)

- Performance Test Cases (The inputs *m* and *n* are somewhat big, such as to delete the 997th number at each step from a circle with 4000 numbers originally)

Minimum Number of Moves to Sort Cards

■ **Question 97** You are required to sort *n* cards numbered from 1 to *n*. You can choose one card and move it to any place you want (insert to any place, not swap). Given a sequence, please implement a function to return the minimum move count to sort these cards.

For example, given a sequence {1, 2, 5, 3, 4, 7, 6}, you can move 5 and insert it between 4 and 7, and the sequence becomes {1, 2, 3, 4, 5, 7, 6}. This is one move. If 7 is moved behind 6, the whole sequence gets sorted. Therefore, it needs two steps at least to sort cards in the sequence of {1, 2, 5, 3, 4, 7, 6}.

In order to sort a sequence with minimum number of moves, we first find the longest increasing subsequence, and then it is only necessary to insert numbers out of the longest increasing subsequence into the appropriate places. For example, one of the longest increasing subsequences of {1, 2, 5, 3, 4, 7, 6} is {1, 2, 3, 4, 6}. If the two numbers 5 and 7 are inserted into right places of {1, 2, 3, 4, 6}, the whole sequence is sorted.

Therefore, if we get the length of the longest increasing subsequence (which is not necessarily to be continuous), it is easy to know the minimum number of moves, as shown in Listing 8-21.

Listing 8-21. *Java Code for the Minimum Number of Moves to Sort a Sequence*

```
int minMoveCount(int[] seq) {
    return seq.length - longestIncreasingLength(seq);
}
```

It is a classic problem to get the length of the longest increasing subsequence, and there are two solutions available for it.

Based on Dynamic Programming Costing O(n^2) Time

A function $f(i)$ is defined to indicate the maximum length of all subsequences ending with the i^{th} number in the input array (denoted as $A[i]$). We define another function $g(i,j)$, which stands for the maximum length of incremental subsequences ending with $A[i]$, whose preceding number in the incremental subsequence is $A[j]$.

The required output is $max[f(i)]$ where $0 \le i < n$ and n is the length of array. We can get $f(i)$ with the following equation:

$$f(i) = max(g(i,j)) \text{ where } g(i,j) = \begin{cases} 1 & A[i] < A[j] \\ f(j)+1 & A[i] > A[j] \end{cases}, 0 \le j < i$$

Take the sample sequence {1, 2, 5, 3, 4, 7, 6} as an example. The longest increasing subsequence ending with the third element is the subsequence {1, 2, 5} itself, and the length is 3.

Let's move on to analyze the longest increasing subsequence ending with the fourth element. The increasing subsequence may come from the first element and the length of {1, 3} is 2.

Similarly, the increasing subsequence may come from the second element, and the length of {1, 2, 3} is 3. However, the increasing subsequence cannot come from the third element because {1, 2, 5, 3} is not incremental. Therefore, the maximum length of the increasing subsequence ending with the fourth element is 3.

The maximum lengths of increasing subsequences ending with the last three elements are 4, 5, and 5 respectively. Therefore, the maximum length of all increasing subsequences is 5.

Even though this problem is analyzed recursively, it is more efficient to implement a solution iteratively with an array, as demonstrated in Listing 8-22.

Listing 8-22. *Java Code for the Length of Longest Increasing Subsequence (Version 1)*

```
int longestIncreasingLength(int[] seq){
    int len = seq.length;
    int[] lookup = new int[len];

    for(int i = 0; i < len; ++i) {
        int longestSoFar = 0;
        for(int j= 0; j < i; ++j) {
            if(seq[i] > seq[j] && lookup[j] > longestSoFar)
                longestSoFar = lookup[j];
        }
        lookup[i] = longestSoFar + 1;
    }

    int longestLength = 0;
    for(int i = 0; i < len; ++i) {
        if(lookup[i] > longestLength)
            longestLength = lookup[i];
    }

    return longestLength;
}
```

The i^{th} number in the auxiliary array `lookup` is for $f(i)$ in the equation above.

It has to compare each number with O(n) numbers ahead of it in an array with n numbers, so it costs O(n^2) to get maximum length of all increasing subsequences.

Based on Binary Search Costing O($n$log$n$) Time

An auxiliary array is utilized to store ending elements of the increasing subsequences while we scan the input sequence and analyze its elements one by one. When the first number 1 of the sequence {1, 2, 5, 3, 4, 7, 6} is scanned, the longest increasing subsequence so far only has one element, and we insert the number 1 at the end into the auxiliary array, and the array becomes {1}.

Similarly, when the second and third numbers are scanned, the longest increasing subsequence so far has three elements and the auxiliary array contains {1, 2, 5}.

The next number to be scanned is 3. The longest increasing subsequence so far has three elements, which are {1, 2, 5} or {1, 2, 3}. At this time, the new end number 3 is not added into the auxiliary array directly, but it replaces the last number 5 because it is less than 5 and so the array becomes {1, 2, 3}.

The reason the number 5 was replaced by 3 is that for the step to insert, the number 4 has no effect as the number 5 is the last number in the subsequence {1, 2, 5}, and the next number 4 cannot be appended to the subsequence because 4 is less than 5. However, when the number is replaced with 3, the array still keeps increasing after the number 4 is inserted, and a longer increasing subsequence {1, 2, 3, 4} with four numbers is found.

When a number in the input sequence is scanned, it is compared with the greatest number in the auxiliary array. If the newly scanned number is greater, it is inserted into the array directly. Otherwise, it replaces the least number of those greater than it in the array in order to allow a wider range of values to be added after it. In order to find the greatest number in the array as well as to find the first number greater than a target value efficiently, the array keeps sorted.

When the next number 7 is scanned, it is inserted into the auxiliary array and the array becomes {1, 2, 3, 4, 7}. And then the array becomes {1, 2, 3, 4, 6} after the last number 6 is scanned. The previous end number 7 is replaced because it is greater than 6.

The maximum length of all increasing subsequences of {1, 2, 5, 3, 4, 7, 6} is five because there are five elements in the auxiliary array {1, 2, 3, 4, 6} finally.

It should be noticed that the auxiliary array is not guaranteed to be the longest increasing subsequence. For example, the auxiliary array is {1, 3, 5, 6} after all numbers in the sequence {2, 4, 1, 5, 6, 3} are handled. The auxiliary array is not the longest increasing subsequence, but its length is the same as the longest increasing subsequence, which is {2, 4, 5, 6}.

Because the auxiliary array is sorted, the binary search algorithm is applied to find the first number greater than a target. This solution can be implemented as shown in Listing 8-23.

Listing 8-23. *Java Code for the Length of Longest Increasing Subsequence (Version 2)*

```
int longestIncreasingLength(int[] seq) {
    int len = seq.length;
    int[] lookup = new int[len];

    lookup[0] = seq[0];
    int longestLength = 1;
    for(int i = 1; i < len; ++i) {
        if(seq[i] > lookup[longestLength - 1]) {
            longestLength++;
            lookup[longestLength - 1] = seq[i];
        }
        else {
            int low = 0;
            int high = longestLength - 1;
            while(low != high) {
                int mid = (low + high) / 2;
                if(lookup[mid] < seq[i]) {
                    low = mid + 1;
                }
                else {
                    high = mid;
                }
            }
```

```
            lookup[low] = seq[i];
        }
    }

    return longestLength;
}
```

As we know, it takes O(log$n$) time to search in a sorted array with n elements, so the overall time complexity is ($n$log$n$).

Source Code:

`097_MinimalMoves.java`

Test Cases:

- Normal Test Cases (An arbitrary array with some numbers)
- Boundary Test Cases (There is only one number in the array; the array is increasingly/decreasingly sorted)

Most Profit from Stock

■ **Question 98** Stock prices are stored in an array in the order of date. How do you get the most profit from a sequence of stock prices?

For example, the most profit to be gained from the sequence of ordered stock prices {9, 11, 5, 7, 16, 1, 4, 2} is 11, bought when the price was 5 and sold when the price was 16.

The stock profit is the difference in prices in buying and selling stock. Of course, we can sell stock only after we buy it. A pair is composed of a buying price and a selling price. Therefore, the most profit is the maximum difference of all pairs in a sequence of stock prices. This problem is essentially to get the maximum difference for all pairs in an array.

The naive brute-force solution is quite straightforward. We can get the result for each number minus every number on its left side, and then get the maximum difference after comparisons. Since O(n) minus operations are required for each number in an array with n numbers, the overall time complexity is O(n^2). A brute-force solution usually is not the best one. Let's try to reduce the times for these minus operations.

Based Divide and Conquer

An array is divided into two sub-arrays of the same size. The maximum difference of all pairs occurs in one of the three following situations: (1) two numbers in a pair are both in the first sub-array; (2) two numbers in a pair are both in the second sub-array; or (3) the minuend is the maximum number in the second sub-array, and the subtrahend is the minimum number in the first sub-array.

It is not difficult to get the minimum number in the first sub-array and the maximum number in the first sub-array. How about getting the maximum difference of all pairs inside two sub-arrays? They are actually subproblems of the original problem and we can solve them via recursion. Listing 8-24 contains the sample code of this solution.

Listing 8-24. *C++ Code to Get Pair with Maximum Difference (Version 1)*

```
int MaxDiff_Solution1(int numbers[], unsigned length) {
    if(numbers == NULL && length < 2)
        return 0;

    int max, min;
    return MaxDiffCore(numbers, numbers + length - 1, &max, &min);
}

int MaxDiffCore(int* start, int* end, int* max, int* min) {
    if(end == start) {
        *max = *min = *start;
        return 0x80000000;
    }

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

    int maxLeft, minLeft;
    int leftDiff = MaxDiffCore(start, middle, &maxLeft, &minLeft);

    int maxRight, minRight;
    int rightDiff = MaxDiffCore(middle + 1, end, &maxRight, &minRight);

    int crossDiff = maxRight - minLeft;

    *max = (maxLeft > maxRight) ? maxLeft : maxRight;
    *min = (minLeft < minRight) ? minLeft : minRight;

    int maxDiff = (leftDiff > rightDiff) ? leftDiff : rightDiff;
    maxDiff = (maxDiff > crossDiff) ? maxDiff : crossDiff;
    return maxDiff;
}
```

With the function `MaxDiffCore`, we get the maximum difference of pairs in the first sub-array (`leftDiff`), and then get the maximum difference of pairs in the second sub-array (`rightDiff`). We continue to calculate the difference between the maximum in the second sub-array and the minimum number in the first sub-array (`crossDiff`). The greatest value of the three differences is the maximum difference of the whole array.

We can get the minimum and maximum numbers, as well as their difference in O(1) time, based on the comparison of minimum and maximum numbers of sub-arrays, so the time complexity of the recursive solution is $T(n)=2T(n/2)+O(1)$. We can demonstrate that its time complexity is $O(n)$ with the Master Theory.

Storing the Minimum Numbers while Scanning

Let's define *diff*[*i*] for the difference of a pair whose minuend is the i^{th} number in an array, and the subtrahend corresponding to the maximum *diff*[*i*] should be the minimum of all numbers on the left side of the i^{th} number in an array. We can get the minimum numbers on the left side of each i^{th} number in an array while scanning the array once. The code in Listing 8-25 is based on this solution.

Listing 8-25. *C++ Code to Get Pair with Maximum Difference (Version 2)*

```
int MaxDiff_Solution2(int numbers[], unsigned length) {
    if(numbers == NULL && length < 2)
        return 0;

    int min = numbers[0];
    int maxDiff =  numbers[1] - min;

    for(int i = 2; i < length; ++i) {
        if(numbers[i - 1] < min)
            min = numbers[i - 1];

        int currentDiff = numbers[i] - min;
        if(currentDiff > maxDiff)
            maxDiff = currentDiff;
    }

    return maxDiff;
}
```

It is obvious that its time complexity is O(n) since it is only necessary to scan an array with length n once. It is more efficient than the first solution on memory consumption, which requires O(log$n$) memory for call stack due to recursion.

Source Code:

`098_MaximalProfitBuyingSellingStock.cpp`

Test Cases:

- Normal Test Cases (An arbitrary array with some numbers for stock prices)
- Boundary Test Cases (There is only one number in the array; the array is increasingly/decreasingly sorted)
- Robustness Test Cases (The pointer to the array is `NULL`)

Divergent Thinking Skills

Divergent thinking is a thought process to generate creative ideas in a short period of time by looking for new opportunities and ways to get things done. Instead of taking obvious steps and walking in a straight line from a problem to the solution, we try to see different aspects of the situation and use unusual points of view.

Interviewers pay a lot of attention to candidates' divergent thinking skills because divergent thinking skills demonstrate creativity and passion to explore new solutions. Sometimes interviewers intentionally disallow candidates from taking the traditional solutions. What they expect is for candidates to think from creative perspectives. For example, some interviewers ask candidates to add, subtract, multiply, and divide without using +, -, ×, and ÷ operations. They expect candidates to jump outside the boundary of arithmetic calculation and find solutions with bit operations.

Divergent thinking skills also demonstrate the breadth and depth of knowledge. Candidates are able to explore solutions from various points of view only when they have deep understanding of various domains. For instance, there is a popular interview problem that requires calculating 1+2+…+n without using loops, multiplication and division, keywords including `if`, `switch`, and `case`, as well as the conditional operator. If a candidate has a broad and deep understanding of C++, he or she could solve it with constructors, virtual functions, function pointers, and template specialization.

Calculating 1+2+…+*n*

■ **Question 99** How do you calculate 1+2+…+*n* without multiplication, division, key words including `for`, `while`, `if`, `else`, `switch`, and `case`, as well as a conditional operator (`A?B:C`)?

As we know, 1+2+…+*n*=*n*(*n*+1)/2. However, this equation cannot be utilized because multiplication and division are disallowed. 1+2+…+*n* is also usually calculated iteratively or recursively. Since `for` and `while` have been disallowed, loops cannot be utilized. If we are going to calculate recursively, we have to use `if` or the conditional operator to determine when to end the recursion. However, neither are they allowed.

Based on Constructors

The purpose of a loop is to repeat execution *n* times. Actually, we can repeat execution without `for` and `while` statements. For example, a type is defined first and its constructor will be invoked *n* times if *n* instances are created. If the code to accumulate is inserted into the constructor, 1+2+…+*n* is calculated, as shown in Listing 8-26.

Listing 8-26. *C++ Code to Calculate 1+2+…+n (Version 1)*

```
class Temp {
public:
    Temp() { ++ N; Sum += N; }

    static void Reset() { N = 0; Sum = 0; }
    static unsigned int GetSum() { return Sum; }

private:
    static unsigned int N;
    static unsigned int Sum;
};
```

```
unsigned int Temp::N = 0;
unsigned int Temp::Sum = 0;

unsigned int Sum_Solution1(unsigned int n) {
    Temp::Reset();

    Temp *a = new Temp[n];
    delete []a;
    a = NULL;

    return Temp::GetSum();
}
```

Based on Virtual Functions

The difficulty with utilizing recursion without `if` and the conditional operator is that we cannot determine when to stop. If it is difficult to end recursion inside a function, how about defining two functions? The first function takes the calculation, and the second one acts as a terminator. We have to choose one of them at every step during execution. That is to say, the terminator is selected when *n* becomes 0; otherwise, the first function is selected for all non-zero *n*. Therefore, we have another solution, as shown in Listing 8-27.

Listing 8-27. *C++ Code to Calculate 1+2+…+n (Version 2)*

```
class A;
A* Array[2];

class A {
public:
    virtual unsigned int Sum (unsigned int n) {
        return 0;
    }
};

class B: public A {
public:
    virtual unsigned int Sum (unsigned int n) {
        return Array[!!n]->Sum(n-1) + n;
    }
};

int Sum_Solution2(int n) {
    A a;
    B b;
    Array[0] = &a;
    Array[1] = &b;

    int value = Array[1]->Sum(n);

    return value;
}
```

It makes choices based on virtual functions `A::Sum` and `B::Sum` in the code above. When `n` is not zero, the result of `!!n` is 1 (`true`), so it invokes `B::Sum` to accumulate; otherwise, it tells `A::Sum` to stop when `n` is zero.

Based on Function Pointers

Virtual functions are not available in C, so the preceding solution does not work for C programmers. Fortunately, we could simulate virtual functions with function pointers, as shown in Listing 8-28.

Listing 8-28. *C Code to Calculate 1+2+…+n (Version 3)*

```
typedef unsigned int (*fun)(unsigned int);

unsigned int Solution3_Teminator(unsigned int n)  {
    return 0;
}

unsigned int Sum_Solution3(unsigned int n) {
    static fun f[2] = {Solution3_Teminator, Sum_Solution3};
    return n + f[!!n](n - 1);
}
```

The function `Sum_Solution3` calls itself recursively till `n` is decreased to 0 because `!!0` is 0 (`false`).

Based on Templates

We could also utilize compilers to calculate with the code in Listing 8-29.

Listing 8-29. *C++ Code to Calculate 1+2+…+n (Version 4)*

```
template <unsigned int n> struct Sum_Solution4 {
    enum Value { N = Sum_Solution4<n - 1>::N + n};
};

template <> struct Sum_Solution4<1> {
    enum Value { N = 1};
};
```

The value of `Sum_Solution4<100>::N` is the result of 1+2+…+100. When the C++ compiler sees `Sum_Solution4<100>`, it generates code for the class template `Sum_Solution4` with 100 as the template argument. Note that the type `Sum_Solution4<100>` depends on the type `Sum_Solution4<99>` because in the code above `Sum_Solution4<100>::N=Sum_Solution4<99>::N+100`. The compiler generates code for `Sum_Solution4<99>`, which depends on `Sum_Solution4<98>`. The recursive process stops at the type `Sum_Solution4<1>` because it is explicitly defined.

The process to calculate `Sum_Solution4<100>::N` for 1+2+…+100 is in the compiling time, so the input *n* should be a constant value. Additionally, C++ compilers have constraints on the depth of recursive compiling, so *n* cannot be a large value.

Source Code:

`099_Accumulate.cpp`

Test Cases:

- Normal Test Cases (Input 5 and 10 to calculate 1+2+…+5 and 1+2+…+10)
- Boundary Test Cases (Input 1)

Implementation of +, -, *, and /

■ **Question 100** How do you implement a function to add two integers without utilization of arithmetic +, -, *, and / operators?

First of all, let's analyze how we add decimals with three steps. Take 5+17=22 as an example. The first step is to add digits without carries, and we get 12. (The digits of the least significant bits are 5 and 7, and the sum of them is 2 without a carry. The tens digits are 0 and 1, and the sum of them 1.) The second step is for carries. There is a 1 carry for the least significant digits, and the actual value for the carry is 10 (1×10=10 in decimal). The last step is to add the sums of the two steps together, and we get 12+10=22.

What can we calculate without arithmetic operators? It seems that we do not have other choices except bit operations. Bit operations are based on binary numbers. Let's have a try at adding binary numbers with three steps similar to the process above.

The binary representations of 5 and 17 are 101 and 10001 respectively. We get 10100 without carries in the first step. (They are two 1 digits at the least significant digits. In binary 1+1=10, and the sum is 0 without carries.) Carries are recorded in the second step. There is a carry 1 at the least significant digits to add 101 and 10001, and the actual value of the carry is 10 (1×10=10 in binary). When 10100 and 10 are added, the sum is 10110, and it is 22 in decimal. Therefore, binary numbers can also be added in three steps.

Let's move on now to replace the steps with binary operations. The first step is to add digits without carries. The results are 0 when adding 0 and 0, as well as adding 1 and 1. The results are 1 when adding 0 and 1, as well as 1 and 0. Note that these results are the same as bitwise XOR operations. The bitwise XOR results of 0 and 0, as well as 1 and 1, are 0, while the XOR results of 0 and 1, as well as 1 and 0, are 1.

The second step is for carries. There are no carries while adding 0 and 0, 0 and 1, as well as 1 and 0. There is a carry only when 1 and 1 are added. These results are the same as bitwise AND operations. Additionally, we have to shift carries to the left for one bit to get the actual carry value.

The third step is to add results of the first two steps. The adding operations can be replaced with bit operations again. These two steps above are repeated until there are carries.

It is time to write code after we have clear picture of how to simulate addition with bit operation. The sample code is shown in Listing 8-30.

Listing 8-30. Java Code to Add

```
int add(int num1, int num2) {
    int sum, carry;
    do {
        sum = num1 ^ num2;
        carry = (num1 & num2) << 1;
        num1 = sum;
        num2 = carry;
    } while (num2 != 0);

    return num1;
}
```

Source Code:

```
100_103_ArithmeticOperations.java
```

Test Cases:

- Add positive numbers, negative numbers, and 0

■ **Question 101** How do you implement a function for the subtraction operation without utilization of arithmetic +, -, *, and / operators?

Subtraction can be implemented with addition because a-b=a+(-b). Additionally, -b can be gotten with bit operations because -b=~b+1. We have simulated addition with bit operations, so there are no problems in simulating subtraction with bit operations, as shown in Listing 8-31.

Listing 8-31. *Java Code to Subtract*

```
int subtract(int num1, int num2) {
    num2 = add(~num2, 1);
    return add(num1, num2);
}
```

Source Code:

```
100_103_ArithmeticOperations.java
```

Test Cases:

- Two numbers for subtraction are positive numbers, negative numbers, or 0

■ **Question 102** How do you implement a function for the multiplication operation without utilization of arithmetic +, -, *, and / operators?

As we know, $a \times n$ ($n \geq 0$) is the same as $a+a+\ldots+a$ (for n times). Therefore, we could invoke the `add` method above in a loop to implement multiplication. If we are going to multiply a number with n, we have to invoke `add` for n times. Is it possible to reduce the times that are necessary to be invoked?

Let's take $a \times 6$ as an example. The number 6 is 110 in binary, which is 2 (10 in binary) plus 4 (100 in binary), so $a \times 6 = a \times 2 + a \times 4$. Note that $a \times 2$ and $a \times 4$ can be calculated based on left-shift operations, which are $a << 1$ and $a << 2$, respectively.

Therefore, $a \times n$ can be implemented with a left-shift and additions. Since there are $O(\log n)$ 1 bits in the binary representation of n, the new solution invokes the `add` method for $O(\log n)$ times, as implemented in Listing 8-32.

Listing 8-32. *Java Code to Multiply*

```
int multiply(int num1, int num2) {
    boolean minus = false;
    if ((num1 < 0 && num2 > 0) || (num1 > 0 && num2 < 0))
        minus = true;

    if (num1 < 0)
        num1 = add(~num1, 1);
    if (num2 < 0)
        num2 = add(~num2, 1);

    int result = 0;
    while (num1 > 0) {
        if ((num1 & 0x1) != 0) {
            result = add(result, num2);
        }

        num2 = num2 << 1;
        num1 = num1 >> 1;
    }

    if (minus)
        result = add(~result, 1);

    return result;
}
```

Source Code:

`100_103_ArithmeticOperations.java`

Test Cases:

- The two numbers for multiplication are positive numbers, negative numbers, or 0

■ **Question 103** How do you implement a function to divide an integer by another without utilization of arithmetic +, -, *, and / operators?

For two positive integers, if $a/b=n$, $a-b\times n\geq 0$ and $a-b\times(n+1)<0$. Therefore, the division can be implemented with the subtract method in a loop. It invokes the method `subtract` for O(n) times when the result is n.

There is a more efficient solution available. If the result of a/b is n, and m bits are 1 in the binary representation of n (the n_1, n_2, …, n_m bit counting from the right end),

$$n = \sum_{i=1}^{m} 2^{n_t-1},$$

and

$$a - b \times \sum_{i=1}^{m} 2^{n_t-1} \geq 0.$$

Similar to before,

$$b \times 2^{n_t-1}$$

can be implemented with left-shift operation. Therefore, the division can be implemented with the code shown in Listing 8-33.

Listing 8-33. *Java Code to Divide*

```
int divide(int num1, int num2) {
    if (num2 == 0)
        throw new ArithmeticException("num2 is zero.");

    boolean minus = false;
    if ((num1 < 0 && num2 > 0) || (num1 > 0 && num2 < 0))
        minus = true;

    if (num1 < 0)
        num1 = add(~num1, 1);
    if (num2 < 0)
        num2 = add(~num2, 1);

    int result = 0;
    for (int i = 0; i < 32; i=add(i, 1)) {
        result = result << 1;
        if ((num1 >> (31 - i)) >= num2) {
            num1 = subtract(num1, num2 << (31 - i));
            result = add(result, 1);
        }
    }

    if (minus)
        result = add(~result, 1);

    return result;
}
```

Source Code:

`100_103_ArithmeticOperations.java`

Test Cases:

- The two numbers for multiplication are positive numbers, negative numbers, or 0

Final/Sealed Classes in C++

■ **Question 104** Please design a class in C++ that cannot be inherited.

In C# there is a keyword `sealed`, indicating a class that cannot be a parent class of other classes. In Java, there is a similar keyword `final`. There are no such keywords in C++, so we have to implement a mechanism for `sealed` or `final` classes.

Based on Private Constructors

As we know, a constructor of a C++ class invokes the constructor of its parent class, and a destructor invokes its parent class's destructor. If a constructor or destructor is declared as a private function, it cannot be invoked outside the class where it is defined. Therefore, a class whose constructors and destructor are private cannot have children classes. If we are going inherit it, a compiling error is raised.

How do you create and release instances of a class whose constructors and destructor are private? We may utilize public static member functions, as shown in Listing 8-34.

Listing 8-34. *C++ Code for Sealed/Final Classes (Version 1)*

```
class SealedClass1 {
public:
    static SealedClass1* GetInstance() {
        return new SealedClass1();
    }

    static void DeleteInstance( SealedClass1* pInstance) {
        delete pInstance;
    }

private:
    SealedClass1() {}
    ~SealedClass1() {}
};
```

It is a bit inconvenient to use these classes because there are some differences from normal classes. For example, we can only create instances of a `SealedClass1` on the heap, but we cannot create instances on the stack.

Based on Private Constructors

It is a little bit tricky to define a `sealed/final` class whose instances can be created on the stack. Listing 8-35 contains the sample code.

Listing 8-35. *C++ Code for Sealed/Final Classes (Version 2)*

```
template <typename T> class MakeSealed {
    friend T;

private:
    MakeSealed() {}
    ~MakeSealed() {}
};

class SealedClass2 : virtual public MakeSealed<SealedClass2> {
public:
    SealedClass2() {}
    ~SealedClass2() {}
};
```

We can create instances of the class `SealedClass2` on both the heap and stack space, so it is more convenient to use than `SealedClass1`.

Even though the constructor and destructor of `MakeSealed<SealedClass2>` are defined as private functions, they can be invoked by the class `SealedClass2` because `SealedClass2` is a friend class of `MakeSealed<SealedClass2>`.

The compiler raises an error when we are trying to derive a new class from `SealedClass2`, such as the class `Try` in Listing 8-36.

Listing 8-36. *C++ Code to Derive a Class from SealedClass2*

```
class Try2 : public SealedClass2 {
public:
    Try2() {}
    ~Try2() {}
};
```

Since the class `SealedClass2` is virtually inherited from `MakeSealed<SealedClass2>`, the constructor of `Try` skips `SealedClass2` and invokes the constructor of `MakeSealed<SealedClass2>`. The class `Try` is not a friend of `MakeSealed<SealedClass2>`, so it raises a compiling error to invoke the constructor of `MakeSealed<SealedClass2>` from `Try`.

Therefore, we cannot inherit children classes from `SealedClass2`, and it is a `sealed/final` class.

The source code for `SealedClass2` can be compiled smoothly in Visual Studio, but it cannot be compiled in GCC. Currently, the template argument cannot be a friend type in GCC. Therefore, the second type has problems from the perspective of portability.

Source Code:

`104_SealedClass.cpp`

Array Construction

■ **Question 105** Given an array *A*[0, 1, …, *n*-1], please construct an array *B*[0, 1, …, *n*-1] in which *B*[*i*]=*A*[0]×*A*[1]×... ×*A*[*i*-1]×*A*[*i*+1]×...×A[*n*-1]. No division should be involved to solve this problem.

If there are no limitations to utilize the division operation, *B*[*i*] can be calculated by $\prod_{j=0}^{n-1} A[j] / A[i]$. Be careful when *A*[*i*] is zero.

It is not allowed to use division here, so we have to explore alternatives. An intuitive solution to calculate *B*[*i*] is to multiply *n*-1 numbers, so the time complexity to construct the array *B* is O(n^2). There are more efficient solutions available.

B[*i*] is *A*[0]×*A*[1]×…×*A*[*i*-1]×*A*[*i*+1]×...×*A*[*n*-1], which is the multiplication result of two sequences *A*[0]×*A*[1]×...×*A*[*i*-1] and *A*[*i*+1]×...×*A*[*n*-2]×*A*[*n*-1]. Therefore, the array *B* can be visualized as a matrix, as shown in Figure 8-4. The multiplication result of numbers in the i^{th} row is *B*[*i*].

Let's define *C*[*i*]=*A*[0]×*A*[1]×…×*A*[*i*-1] and *D*[*i*]= *A*[*i*+1]×...×*A*[*n*-2]×*A*[*n*-1], so *B*[*i*]=*C*[*i*]×*D*[*i*]. Note that *C*[*i*]=*C*[*i*-1]×*A*[*i*-1], and *D*[*i*]=*D*[*i*+1]×*A*[*i*+1]. Therefore, *C*[*i*] can be calculated in a top down order, and *D*[*i*] can be calculated in bottom up order. After *C*[*i*] and *D*[*i*] are calculated, they are multiplied for *B*[*i*].

B_0	1	A_1	A_2	...	A_{n-2}	A_{n-1}
B_1	A_0	1	A_2	...	A_{n-2}	A_{n-1}
B_2	A_0	A_1	1	...	A_{n-2}	A_{n-1}
...	A_0	A_1	...	1	A_{n-2}	A_{n-1}
B_{n-2}	A_0	A_1	...	A_{n-3}	1	A_{n-1}
B_{n-1}	A_0	A_1	...	A_{n-3}	A_{n-2}	1

Figure 8-4. *The constructed array B is visualized as a matrix.*

This solution can be implemented in Java with the code in Listing 8-37.

Listing 8-37. *Java Code to Construct an Array*

```
void multiply(double array1[], double array2[]){
    if(array1.length == array2.length && array1.length > 0){
        array2[0] = 1;
        for(int i = 1; i < array1.length; ++i){
            array2[i] = array2[i - 1] * array1[i - 1];
```

```
        }

        int temp = 1;
        for(int i = array1.length - 2; i >= 0; --i){
            temp *= array1[i + 1];
            array2[i] *= temp;
        }
    }
}
```

The time complexity of this solution is O(n) obviously, and it is more efficient than the intuitive solution above.

Source Code:

`105_ConstuctArray.java`

Test Cases:

- Normal Test Cases (There are positive numbers, negative numbers, and 0 in the array A)
- Special Test Cases (There are no, only one, or more 0s in the array A)

Summary

Besides programming abilities, it is important for candidates to demonstrate other skills, such as communication skills, knowledge migration skills, mathematical modeling skills, and divergent thinking skills.

Candidates show their communication skills when they describe project experiences and solutions to coding interview problems clearly. Additionally, it is an indication of good communication skills if they ask pertinent questions proactively.

Knowledge migration skills are helpful in solving new problems. When an interview problem is similar to a classic algorithm or other popular problems, candidates may focus on the similarities and try to reapply the classic algorithm or known solutions of other problems.

There are many interesting interview problems abstracted from daily life that require mathematical modeling skills. In order to solve such problems, candidates have to choose appropriate data structures to algorithms according to the hidden model.

Sometimes interviewers disallow candidates from using traditional solutions. In such cases, candidates have to show their divergent thinking skills by solving problems with creative ideas from unusual perspectives.