CHAPTER 4

Algorithms

In addition to data structures, algorithms are also quite common topics in interviews. There are many interview questions about search and sort algorithms. Backtracking, dynamic programming, and greedy algorithms are useful tools to solve many problems posed in coding interviews. Additionally, bit operations can be viewed as special algorithms to manipulate binary integers. All of these algorithms will be discussed in this chapter.

Recursion and Iteration

Repeated operations are handled either by recursion or iteration with loops. Recursion is a function (method) that invokes itself, directly or indirectly. One of the fundamental structures of programming is loops, which are built-in structures of most languages. For example, there are two strategies to calculate 1+2+…+*n*. The first strategy is a utilization of recursion, as shown in Listing 4-1.

Listing 4-1. *Recursive C Code to Calculate 1+2+…+*n

```
int AddFrom1ToN_Recursive(int n) {
    return n <= 0 ? 0 : n + AddFrom1ToN_Recursive(n - 1);
}
```

The second strategy is based on iteration, as shown in Listing 4-2.

Listing 4-2. *Iterative C Code to Calculate 1+2+…+*n

```
int AddFrom1ToN_Iterative(int n) {
  int i;
  int result = 0;
    for(i = 1; i <= n; ++ i)
        result += i;

    return result;
}
```

Usually, the code with recursion is more concise than the code with iteration. There is only one statement inside the recursive function, and there are more inside the iterative one. Additionally, loops, especially complex ones that involve nesting, are difficult to read and understand. Recursion may be

clearer and simpler to divide a complex problem into manageable pieces. For instance, recursive implementations of pre-order, in-order, and post-order traversal algorithms on binary trees are much simpler than iterative implementations.

■ **Tip** In most cases, recursive solutions are more concise as well as easier to implement than iterative solutions. Candidates may prefer to use recursion during interviews if it is not explicitly required to utilize iteration.

Recursion has some disadvantages even though it looks more concise and simpler than iteration. It is recursion when a function invokes itself, and function invokes consume time and space: it has to allocate memory on the stack for arguments, return address, and local variables. It also costs time to push and pop data on the stack. Therefore, the recursive solution to calculate 1+2+…+*n* is not as efficient as the iterative one.

Recursion has more negative impacts on performance if there are duplicated calculations. Recursion is essentially a technique to divide a problem into two or more subproblems. There are duplicated calculations if there are overlaps among subproblems. More details on the performance issue in recursion are discussed in the following subsection on the Fibonacci Sequence.

A more serious problem with recursion other than inefficiency is that it causes errors due to call stack overflows. As mentioned earlier, it consumes some memory on the stack for each recursive call, so it may use up all memory on the stack if recursive levels are very deep, and cause call stack overflow. For instance, when the input *n* to calculate 1+2+…+*n* is a relatively small number, such as 10, both recursive and iterative solution can get the correct result 55. However, if the input is 5000, the recursive solution crashes, but the iterative solution still gets the correct result 12502500. Therefore, iteration is more robust for large input data.

Fibonacci Sequence

■ **Question 23** Given a number *n*, please find the n^{th} element in the Fibonacci Sequence, which is defined as the following equation:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

Recursive and Inefficient

In many textbooks, the Fibonacci Sequence is taken as an example to illustrate recursive functions, so many candidates are quite familiar with the recursive solution and can write code quickly, as shown in Listing 4-3.

Listing 4-3. *Recursive C Code for Fibonacci Sequence*

```
long long Fibonacci(unsigned int n) {
    if(n <= 0)
        return 0;

    if(n == 1)
        return 1;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

However, recursion is not the best solution for this problem since it has serious performance issues. Let's take *f*(10) as an example to analyze the recursive process. The element *f*(10) is calculated based on *f*(9) and *f*(8). Similarly, *f*(9) is based on *f*(8) and *f*(7), and so on. The dependency can be visualized as a tree (Figure 4-1).

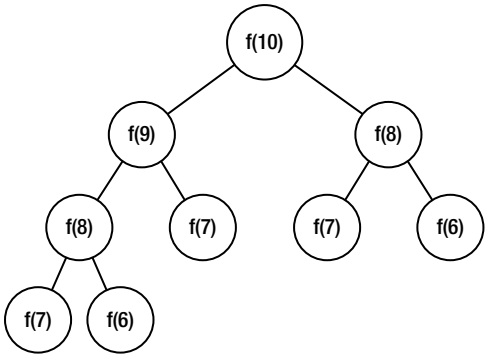

Figure 4-1. *The recursive process to calculate the* 10^{th} *element in the Fibonacci Sequence*

It is noticeable that there are many duplicated nodes in the tree. The number of duplication increases dramatically when *n* increases. It can be proven that its time complexity grows exponentially with *n*.

Iterative Solution with O(*n*) Time Complexity

It is not difficult to improve this performance. Since the slowness is caused by duplicated calculations, let's try to void the duplications. A solution is to cache the calculated elements. Before an element is calculated, we first check whether it is cached. It is not calculated again if an element is already in the cache.

A simpler solution is to calculate using a bottom-up process. The elements *f*(0) and *f*(1) are already known, and *f*(2) can be calculated based on them. Similarly, *f*(3) can be calculated based on *f*(2) and *f*(1), and so on. It iterates until *f*(n) is calculated. The code for this iterative solution is shown in Listing 4-4.

Listing 4-4. *Iterative C Code for Fibonacci Sequence*

```
long long Fibonacci_Solution2(unsigned int n) {
    int result[2] = {0, 1};
    long long  fibNMinusOne = 1;
    long long  fibNMinusTwo = 0;
    long long  fibN = 0;
    unsigned int i;

    if(n < 2)
        return result[n];

    for(i = 2; i <= n; ++ i) {
        fibN = fibNMinusOne + fibNMinusTwo;

        fibNMinusTwo = fibNMinusOne;
        fibNMinusOne = fibN;
    }

     return fibN;
}
```

Obviously, the time complexity for this iterative solution is O(*n*).

More Efficient in O(log*n*) Time

Usually, the solution costing O(*n*) time is the expected solution. However, interviewers may require a solution with O(log*n*) time complexity if they have higher expectation on performance.

There is an equation for the Fibonacci Sequence:

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

If you are interested, you can prove it with mathematical induction.

With the equation above, it finds f(*n*) if it finds $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$. Additionally, an element (including a number, a matrix, and so on) to the power of *n* can be calculated recursively based on the following equation:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & n \text{ is odd} \end{cases}$$

It can get a^n in O(log*n*) time with the equation above, so the n^{th} element in the Fibonacci Sequence can also be calculated in O(log*n*) time. The corresponding implementation is shown in Listing 4-5.

Listing 4-5. *C Code to Calculate Fibonacci Sequence in O(log*n*) Time*

```
struct Matrix2By2 {
    long long m_00;
    long long m_01;
```

```
    long long m_10;
    long long m_11;
};

struct Matrix2By2 MatrixMultiply(const struct Matrix2By2* matrix1, const struct Matrix2By2*
matrix2) {
    struct Matrix2By2 result;
    result.m_00 = matrix1->m_00 * matrix2->m_00 + matrix1->m_01 * matrix2->m_10;
    result.m_01 = matrix1->m_00 * matrix2->m_01 + matrix1->m_01 * matrix2->m_11;
    result.m_10 = matrix1->m_10 * matrix2->m_00 + matrix1->m_11 * matrix2->m_10;
    result.m_11 = matrix1->m_10 * matrix2->m_01 + matrix1->m_11 * matrix2->m_11;

    return result;
}

struct Matrix2By2 MatrixPower(unsigned int n) {
    struct Matrix2By2 result;
    struct Matrix2By2 unit = {1, 1, 1, 0};

    assert(n > 0);
    if(n == 1) {
        result = unit;
    }
    else if(n % 2 == 0) {
        result = MatrixPower(n / 2);
        result = MatrixMultiply(&result, &result);
    }
    else if(n % 2 == 1) {
        result = MatrixPower((n - 1) / 2);
        result = MatrixMultiply(&result, &result);
        result = MatrixMultiply(&result, &unit);
    }

    return result;
}

long long Fibonacci_Solution3(unsigned int n) {
    struct Matrix2By2 PowerNMinus2;
    int result[2] = {0, 1};

    if(n < 2)
        return result[n];

    PowerNMinus2 = MatrixPower(n - 1);
    return PowerNMinus2.m_00;
}
```

Source Code:

    023_Fibonacci.c

Test Cases:

- Normal case: 3, 5, 10
- Boundary case: 0, 1, 2
- Big numbers for performance tests, such as 40, 50, 100

■ **Question 24** There is a stair with *n* levels. A frog can jump up one level or two levels at one time on the stair. How many ways are there for the frog to jump from the bottom of the stairs to the top?
For example, there are three choices for the frog to jump up a stair with three levels: (1) it jumps in three steps, one level for each jump; (2) it jumps in two steps, one level for the first jump and two levels for the second jump; or (3) it jumps with two steps, two levels for the first step and one level for the last jump.

Let's define a function $f(n)$ for the number of choices available on a stair with n levels. There are two choices for the frog at the first step. One choice is to jump only one level, and it has $f(n\text{-}1)$ choices for the remaining n-1 levels. The other one is to jump two levels at the first step, and it has $f(n\text{-}2)$ choices for the remaining n-2 levels. Therefore, the total number of choices on a stair with n levels is $f(n) = f(n\text{-}1) + f(n\text{-}2)$, which is the Fibonacci Sequence.

■ **Question 25** There is a stair with *n* levels. A frog can jump up 1, 2, …, *n*-1, *n* levels at each step on the stair. How many approaches are there for this frog to jump from the bottom of the stair to the top?
For example, there are four approaches for the frog to jump on a stair with three levels: (1) it jumps with three steps, one level for each step; (2) it jumps with two steps, one level for the first step and two levels for the second step; (3) it jumps with two steps, two levels for the first step and one level for the second step; or (4) it jumps in only one step from the bottom to the top directly.

Similar to Question 24, a function $f(n)$ can be defined for the number of choices on a stair with n levels. Inspired by the solution of the previous problem, it is easy to get $f(n) = f(n\text{-}1)+f(n\text{-}2)+\ldots+f(1)+1$. Since $f(n\text{-}1)=f(n\text{-}2)+\ldots+f(1)+1$, $f(n)=2f(n\text{-}1)$. It is not difficult to get $f(n) = 2^{n-1}$.

What is the most efficient method to get $f(n)$? It costs $O(n)$ time if it is calculated sequentially with a loop, and it costs $O(\log n)$ time with recursion, as shown in the third solution for the Fibonacci Sequence.

Actually, there is an even faster solution available. Because $f(n)$ is 2 to the power of n-1, there is only one bit of 1 in the binary representation of the number. Therefore, $f(n)$ can be calculated with a left-shift operator, 1 << (n - 1), in O(1) time.

■ **Question 26** Rectangles with size 2×1 are utilized to cover other rectangles, horizontally or vertically. How many approaches are available to cover a 2×8 rectangle with eight 2×1 rectangles (Figure 4-2)?

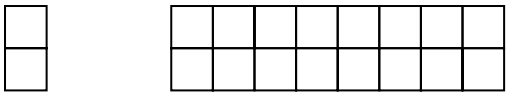

Figure 4-2. *A 2×1 rectangle and a 2×8 rectangle*

The number of choices to cover a 2×8 rectangle is denoted as $f(8)$. There are two choices to cover with the first 2×1 rectangle. One is to lay it vertically to the left side of the 2×8 rectangle, and the number of choices to cover the remaining 2×7 rectangle is $f(7)$. The other choice is to lay the first 2×1 at the top left corner horizontally. It has to lay another 2×1 rectangle at the lower left corner. The number of choices to cover the remaining 2×6 rectangle is $f(6)$.

We can see that $f(8)=f(7)+f(6)$, so it is also a Fibonacci Sequence.

Search and Sort

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

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

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

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

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

Listing 4-6. *Java Code to Partition an Array*

```
int partition(int[] numbers, int start, int end) {
    Random random = new Random();
    int pivot = random.nextInt(end - start + 1) + start;
    swap(numbers, pivot, end);

    int small = start - 1;
    for(int i = start; i <= end; ++i) {
        if(numbers[i] < numbers[end]) {
            ++small;
            if(i != small)
                swap(numbers, small, i);
        }
    }

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

    return small;
}
```

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

Listing 4-7. *Java Code for Quicksort*

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

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

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

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

The method `partition` is useful not only to sort, but also to find the arbitrary k^{th} number from an array. The interview question "Majorities in Arrays" (Question 29) and "Minimal k Numbers" (Question 70) are both solved with the `partition` method.

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

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

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

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

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

Interviewer: Yes.

Candidate: Is it allowed to use auxiliary memory?

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

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

Listing 4-8. *Java Code of Count Sort*

```
void countSort(int ages[]) {
    int oldestAge = 99;
    int timesOfAge[] = new int[oldestAge+1];
    for(int i = 0; i <= oldestAge; ++ i)
        timesOfAge[i] = 0;

    for(int i = 0; i < ages.length; ++ i) {
        int age = ages[i];
        if(age < 0 || age > oldestAge)
            throw new IllegalArgumentException("Out of range.");
        ++ timesOfAge[age];
    }

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

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

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

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

Binary Search in Partially Sorted Arrays

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

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

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

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

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

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

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

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

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

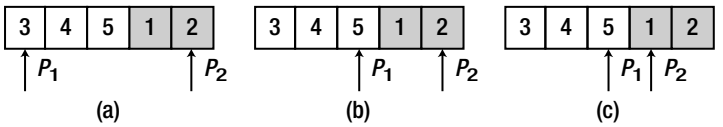

Figure 4-3. *Search the minimal element in an array {3, 4, 5, 1, 2}. Elements in the gray background are in the second increasingly sorted sub-array. (a)* P_1 *points to the first element, and* P_2 *points to the last element. Since the middle number 5 is greater than the number pointed to by* P_1*,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

P_2 *to the middle number for the next round of search. (c) Since* P_1 *and* P_2 *point to two adjacent numbers, the number pointed to by* P_2 *is the minimum.*

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

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

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

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

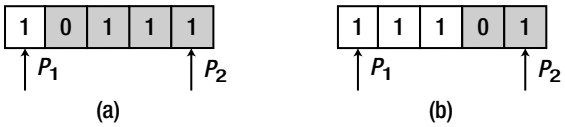

Figure 4-4. *Two rotations of an increasingly sorted array {0, 1, 1, 1, 1}: {1, 0, 1, 1, 1} and {1, 1, 1, 0, 1}. Elements with gray background are in the second increasingly sorted sub-array. In both cases, the first, last, and middle numbers are equal. (a) The middle element is in the second sub-array.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

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

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

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

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

        indexMid = (index1 + index2) / 2;

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

        if(numbers[indexMid] >= numbers[index1])
```

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

    }

    return numbers[indexMid];
}

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

    return result;
}
```

Source Code:

```
027_ArrayRotation.java
```

Test Cases:

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

---

■ **Question 28** A turning number is the maximum number in a unimodal array that increases and then decreases. Please write a function (or a method) that finds the index of the turning number in a unimodal array. For example, the turning number in the array {1, 2, 3, 4, 5, 10, 9, 8, 7, 6} is 10, so its index 5 is the expected output.

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

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

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

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

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

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

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

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

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

    return -1;
}
```

Source Code:

```
028_TurningNumber.java
```

Test Cases:

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

Majorities in Arrays

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

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

Based on the Partition Method

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

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

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

```
int getMajority_1(int[] numbers) {
    int length = numbers.length;
    int middle = length >> 1;
    int start = 0;
    int end = length - 1;

    int index = partition(numbers, start, end);
    while(index != middle) {
        if(index > middle) {
            end = index - 1;
            index = partition(numbers, start, end);
        }
        else {
            start = index + 1;
            index = partition(numbers, start, end);
        }
    }

    int result = numbers[middle];
    if(!checkMajorityExistence(numbers, result))
        throw new IllegalArgumentException("No majority exisits.");

    return result;
}
```

The method `partition` was discussed before for quicksort.

The majority is the element that occurs for more than half of the size of the array. How about the scenario where the element occurring most frequently does not meet the bar? That is why a method

checkMajorityExistence is defined. It is important to handle invalid inputs during interviews, as shown in Listing 4-12.

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

```
boolean checkMajorityExistence(int[] numbers, int number) {
    int times = 0;
    for(int i = 0; i < numbers.length; ++i) {
        if(numbers[i] == number)
            times++;
    }

    return (times * 2 > numbers.length);
}
```

Based on the Definition of Majority

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

This solution might be implemented as the code in Listing 4-13, where the method checkMajorityExistence is the same as in the preceding solution.

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

```
int getMajority_2(int[] numbers) {
    int result = numbers[0];
    int times = 1;
    for(int i = 1; i < numbers.length; ++i) {
        if(times == 0) {
            result = numbers[i];
            times = 1;
        }
        else if(numbers[i] == result)
            times++;
        else
            times--;
    }

    if(!checkMajorityExistence(numbers, result))
        throw new IllegalArgumentException("No majority exisits.");

    return result;
}
```

Comparison

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

Source Code:

`029_MajorityElement.java`

Test Cases:

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

Backtracking

Backtracking is a refinement of the brute-force approach, which systematically searches for a solution to a problem among all available options. It is suitable for scenarios where there is a set of options available at each step, and we must choose one from these. After a choice is made, there is a new set of options for the next step. This procedure is repeated over and over until we reach a final state.

Conceptually, all options compose a tree structure. Leaves in the tree correspond to solution states, some of which might be the final acceptable state, but others might not. The backtracking algorithm traverses this tree recursively, from the root down and in depth-first order.

When it reaches a leaf that corresponds to a non-acceptable state, it backtracks to continue the search for another leaf by revoking the most recent choice and tries out the next option. If it runs out of options, it revokes again and tries another choice at that node. If it ends up at the root with no options left, there are no acceptable states to be found.

As shown in Figure 4-5, there are two options available at each step. The backtracking algorithm starts at the root node A, and it selects the node B from the two options. It chooses D for the second step, and the state is not acceptable. Therefore, it backtracks to node B and then selects the next option and advances to node E. It reaches a non-acceptable state again, and it has to return back to node B again. Since it has tried all options on node B, it returns back to the parent node A. It selects the next option of node A, which is node C. It then selects node F. It stops traversal since it reaches an acceptable state.

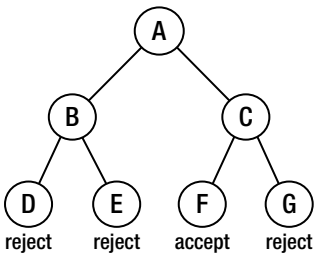

Figure 4-5. *Tree structure to visualize options for certain problems. There are two options available at each step.*

Usually the backtracking algorithm is implemented with recursion. When it reaches a node, it tries all options for the next step recursively if the state is not rejected.

String Path in Matrix

■ **Question 30** How do you implement a function to check whether there is a path for a string in a matrix of characters? It moves to left, right, up, and down in a matrix, and a cell for a step. The path can start from any cell in a matrix. If a cell is occupied by a character of a string on the path, it cannot be occupied by another character again.

For example, the matrix below with three rows and four columns has a path for the string "bcced" (as highlighted in the matrix). It does not have a path for the string "abcb", because the first "b" in the string occupies the "b" cell in the matrix, and the second "b" in the string cannot enter the same cell again.

a *b c* e
s f *c* s
a *d e* e

It is a typical problem that can be solved by the backtracking algorithm. First, we choose any entry in the matrix to start a path. Suppose the character value in an entry is *ch* and the entry is for the i^{th} node on the path. If the i^{th} character in the string is also *ch*, it tries all options available at that entry for the next step. All entries except those on boundaries have four options. It continues until all characters in the string have been found on a path.

The path of a string can be defined as a stack because of the recursive nature of the backtracking algorithm. When the algorithm enters a rejecting state where the character on the path is not the same as the corresponding character in the string, it returns back to the preceding state and pops a character off the stack.

Because a path can enter an entry in the matrix only once at most, a set of Boolean flags are defined to mark whether an entry has been occupied by a previous step on the path.

The solution can be implemented with recursion, as depicted in Listing 4-14.

Listing 4-14. *C++ Code for String Paths*

```
bool hasPath(char* matrix, int rows, int cols, char* str) {
    if(matrix == NULL || rows < 1 || cols < 1 || str == NULL)
        return false;

    bool *visited = new bool[rows * cols];
    memset(visited, 0, rows * cols);

    int pathLength = 0;
    for(int row = 0; row < rows; ++row) {
        for(int col = 0; col < cols; ++col) {
            if(hasPathCore(matrix, rows, cols, row, col, str, pathLength, visited))
```

```
                return true;
            }
        }

        delete[] visited;

        return false;
    }

    bool hasPathCore(char* matrix, int rows, int cols, int row, int col, char* str, int&
    pathLength, bool* visited) {
        if(str[pathLength] == '\0')
            return true;

        bool hasPath = false;
        if(row >= 0 && row < rows && col >= 0 && col < cols
                && matrix[row * cols + col] == str[pathLength]
                && !visited[row * cols + col]) {
            ++pathLength;
            visited[row * cols + col] = true;

            hasPath = hasPathCore(matrix, rows, cols, row, col - 1, str, pathLength, visited)
                    || hasPathCore(matrix, rows, cols, row - 1, col, str, pathLength, visited)
                    || hasPathCore(matrix, rows, cols, row, col + 1, str, pathLength, visited)
                    || hasPathCore(matrix, rows, cols, row + 1, col, str, pathLength, visited);

            if(!hasPath) {
                --pathLength;
                visited[row * cols + col] = false;
            }
        }

        return hasPath;
    }
```

When an entry with index (`row`, `col`) is not a rejecting state, the function `hasPathCore` has a try at the four neighbors with index (`row`, `col` - 1), (`row` - 1, `col`), (`row`, `col` + 1), and (`row` + 1, `col`). It continues until it reaches the final acceptable state when the statement `str[length] == '\0'` gets `true`, which indicates that all characters in the string have been matched in a path.

Source Code:

`030_StringPath.cpp`

Test Cases:

- Functional cases: A 2-D matrix has/does not have a path for a string
- Boundary cases: (1) A 2-D matrix has only a row, a column, or even an element. (2) A path for a string occupies all elements in a matrix.

Robot Move

■ **Question 31** A robot starts at cell (0, 0) of a grid with *m* rows and *n* columns. It can move to the left, right, up, and down, and moves one cell for a step. It cannot enter cells where the digit sum of the row index and column index are greater than a given *k*.

For example, when *k* is 18, the robot can reach cell (35, 37) because 3+5+3+7=18. However, it cannot reach cell (35, 38) because 3+5+3+8=19 and that is greater than *k*. How many cells can the robot reach?

Similar to the preceding problem, a grid can also be viewed as a 2-D matrix, where all cells except those on boundaries have four neighbors.

The robot starts to move from the cell (0,0). When it reaches a cell (*i*,*j*), we check whether it is a valid move according to the digits sum of the indexes. If the move is valid, we continue to move to the four neighboring cells (*i*,*j*-1), (*i*-1,*j*), (*i*,*j*+1), and (*i*+1,*j*). Therefore, the problem can be solved with the recursive code in Listing 4-15.

Listing 4-15. *C++ Code for Moving Robots*

```
int movingCount(int threshold, int rows, int cols){
    bool *visited = new bool[rows * cols];
    for(int i = 0; i < rows * cols; ++i)
        visited[i] = false;

    int count = movingCountCore(threshold, rows, cols, 0, 0, visited);

    delete[] visited;

    return count;
}

int movingCountCore(int threshold, int rows, int cols, int row, int col, bool* visited){
    int count = 0;
    if(check(threshold, rows, cols, row, col, visited)) {
        visited[row * cols + col] = true;

        count = 1 + movingCountCore(threshold, rows, cols, row - 1, col, visited)
                + movingCountCore(threshold, rows, cols, row, col - 1, visited)
                + movingCountCore(threshold, rows, cols, row + 1, col, visited)
                + movingCountCore(threshold, rows, cols, row, col + 1, visited);
    }

    return count;
}
```

Similar to the previous problem, a set of Boolean flags is defined in order to avoid duplicated moves.

The function `check` in Listing 4-16 checks whether a move to the cell (`row`, `col`) is valid, and the function `getDigitSum` finds the sum of all digits of a given number.

Listing 4-16. *C++ Code for Moving Robots*

```
bool check(int threshold, int rows, int cols, int row, int col, bool* visited){
    if(row >=0 && row < rows && col >= 0 && col < cols
        && getDigitSum(row) + getDigitSum(col) <= threshold
        && !visited[row* cols + col])
        return true;

    return false;
}

int getDigitSum(int number){
    int sum = 0;
    while(number > 0){
        sum += number % 10;
        number /= 10;
    }

    return sum;
}
```

Source Code:

`031_RobotMove.cpp`

Test Cases:

- Functional cases: Counting moves on a normal grid
- Boundary cases: Counting moves on a grid with only a row, a column, or even an element

Dynamic Programming and Greedy Algorithms

A popular topic during interviews is dynamic programming, which simplifies a complicated problem by breaking it down into simpler subproblems by means of recursion. If an interview problem has optimal substructure and overlapping subproblems, it might be solved by dynamic programming.

The ingredient optimal substructure means that the solution to a given optimization problem can be obtained by a combination of optimal solutions to its subproblems. Therefore, the first step to utilize dynamic programming is to check whether the problem exhibits such optimal substructures.

For example, the problem to find the minimal number of coins to make change for a value t exhibits optimal substructures. If a set of coins that has the minimal number of coins to make change for the value t contains a coin v_i, the set of coins excluding v_i should have the minimal number of coins to make change for value t-v_i.

The other ingredient, overlapping subproblems, means a recursive algorithm solves subproblems over and over, rather than always generating new subproblems. Dynamic programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be retrieved when necessary.

Let's take the problem of making change as an example again, assuming we are going to make change for value 15 with a set of coins with value 1, 3, 9, 10. $f(t)$ is defined as the optimal solution to get the minimum number of coins to make change for the value t. There are four subproblems for the overall problem $f(15)$: $f(14)$, $f(12)$, $f(6)$, and $f(5)$. It is noticeable that $f(11)$ is a subproblem of both $f(14)$ and $f(12)$, and $f(5)$ is a subproblem of both $f(14)$ and $f(6)$, so there are many overlapping subproblems. Solutions of subproblems are stored in a 2-D table to avoid duplicated calculations. More details about the problem to make change are discussed in the section *Minimal Number of Coins for Change*.

A choice is made at each step in dynamic programming, and the choice usually depends on the solutions to subproblems. Consequently, dynamic programming problems are typically solved in a bottom-up manner, progressing from smaller subproblems to larger subproblems.

Greedy algorithms are different from dynamic programming solutions, in which a greedy choice is made that seems best at the moment and then you solve the subproblem arising after the choice is made. Therefore, a greedy strategy usually progresses in a top-down order, making one greedy choice after another and reducing each given problem instance to a smaller one.

For example, in order to minimize the number of key presses on a cell-phone keyboard, we choose the most frequent character at the first step and locate it as the first character on a key, and then choose the next most frequent character and locate it as the first character on another key. If the first locations of all keys are already occupied, a character is located as the second one on a key, and so on. Actually, it is a greedy choice to select the most frequent character of the remaining characters and locate it in a key with the lowest index. More details about this problem are discussed in the section *Minimal Times of Presses on Keyboards*.

Edit Distance

■ **Question 32** Implement a function that finds the edit distance of two input strings. There are three types of edit operations: insertion, deletion, and substitution. Edit distance is the minimal number of edit operations to modify a string from one state to the other.

For example, the edit distance between "Saturday" and "Sunday" is 3 since the following three edit operations are required to modify one into another:

(1) S*a*turday → Sturday (deletion of 'a')

(2) S*t*urday → Surday (deletion of 't')

(3) Su*r*day → Su*n*day (substitution of 'n' for 'r')

There is no way to achieve it with fewer than three operations.

If a function f(i,j) is defined to indicate the edit distance between the substring of the first string ending with the j^{th} character and the substring of the second string ending with the i^{th} character, it is obvious that $f(i,0)=i$ because when we delete i characters from the substring of the first string ending with the i^{th} character, we get an empty string. (It is also the substring of the second string ending with the 0^{th} character.) Similarly, $f(0,j)=j$.

Let's analyze the cases when both i and j are greater than 0. If the i^{th} character of the second string is the same as the j^{th} character of the first string, no edit operations are necessary. Therefore, $f(i,j) = f(i\text{-}1,j\text{-}1)$ in such a case.

When the j^{th} character of the first string is different from the i^{th} character of the second string, there are three options available:

- Insert the i^{th} character of the second string into the first string. In this case, $f(i,j)=f(i-1,j)+1$.
- Delete the j^{th} character of the first string. In this case, $f(i,j)=f(i,j-1)+1$.
- Replace the j^{th} character of the first string with the i^{th} character of the second string. In this case, $f(i,j)=f(i-1,j-1)+1$.

What is the final value for $f(i,j)$? It should be the minimum value of the three cases. A formal equation can be defined for this problem:

$$f(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ f(i-1, j-1) & string1[j] = string2[i] \\ min\left(f(i-1,j), f(i,j-1), f(i-1,j-1)\right) + 1 & string1[j] \neq string2[i] \end{cases}$$

If we draw a table to show the edit distance values $f(i,j)$ between "Saturday" and "Sunday," it looks like Table 4-1. The edit distance of two strings is at the right-bottom corner of the table for edit distance values.

Table 4-1. *Edit Distance Value* f(i,j) *between "Saturday" and "Sunday"*

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

A table of edit distance values can be implemented as a 2-D array to simulate Table 4-1, which is the variable `distances` in the code found in Listing 4-17.

Listing 4-17. *C# Code for Moving Robots*

```
int GetEditDistance(String str1, String str2) {
    int len1 = str1.Length;;
    int len2 = str2.Length;
```

```
    int[,] distances = new int[len2 + 1, len1 + 1];

    int editDistance = GetEditDistance(str1, str2, distances, len1, len2);

    return editDistance;
}

int GetEditDistance(String str1, String str2, int[,] distances, int len1, int len2) {
    for (int i = 0; i < len2 + 1; ++i)
        distances[i, 0] = i;
    for (int j = 0; j < len1 + 1; ++j)
        distances[0, j] = j;

    for (int i = 1; i < len2 + 1; ++i) {
        for (int j = 1; j < len1 + 1; ++j) {
            if (str1[j - 1] == str2[i - 1])
                distances[i, j] = distances[i - 1, j - 1];
            else {
                int deletion = distances[i, j - 1] + 1;
                int insertion = distances[i - 1, j] + 1;
                int substitution = distances[i - 1, j - 1] + 1;
                distances[i, j] = Min(deletion, insertion, substitution);
            }
        }
    }

    return distances[len2, len1];
}

int Min(int num1, int num2, int num3) {
    int less = (num1 < num2) ? num1 : num2;
    return (less < num3) ? less : num3;
}
```

Source Code:

```
    032_EditDistance.cs
```

Test Cases:

- Functional cases: Finding edit distances of normal strings, including two identical strings, or a string is a substring of the other
- Boundary cases: One or two strings are empty

Minimal Number of Coins for Change

■ **Question 33** Please implement a function that gets the minimal number of coins with values v_1, v_2, …, v_n, to make change for an amount of money with value t. There are an infinite number of coins for each value v_i.
For example, the minimum number of coins to make change for 15 out of a set of coins with values 1, 3, 9, 10 is 3. We can choose two coins with value 3 and a coin with value 9. The number of coins for other choices should be greater than 3.

First, let's define a function $f(t)$, which is the minimum number of coins to make change for the total value t. If there are n different coins, we have n choices to make change for the value t. We can add a coin with value v_1 to a set of coins whose total value is t-v_1. The minimum number of coins to get a value t-v_1 is $f(t$-$v_1)$. Similarly, we can add a coin with value v_2 into a set of coins whose total value is t-v_2. The minimal number of coins to get value t-v_2 is f(t-v_2). It is similar for other coins.

Therefore, a problem to calculate $f(t)$ is divided into n sub-problems: $f(t$-$v_1)$, $f(t$-$v_2)$, …, $f(t$-$v_n)$. We can get a formal equation for $f(t)$ as the following accordingly:

$$f(t) = min(f(t - v_i)) + 1, \text{where} 0 < i \leq n$$

This equation can be implemented with recursion easily. However, the recursive solution may cause serious performance issues since there are overlaps when we divide this problem into n subproblems. A better solution is to utilize iteration and store the result of subproblems into a table (as in Table 4-2).

Table 4-2. *The Iterative Process to Calculate the Minimal Number of Coins to Make Changes for 15*

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
1	*1*	*2*	3	*2*	*3*	4	*3*	*4*	5	2	*2*	3	3	4	5
3	-	-	*1*	*2*	*3*	*2*	*3*	*4*	3	4	5	*2*	*2*	*3*	*3*
9	-	-	-	-	-	-	-	-	*1*	2	3	*2*	3	4	*3*
10	-	-	-	-	-	-	-	-	-	*1*	*2*	3	*2*	*3*	4

In Table 4-2, each column denotes the number of coins to make change for a specific value. We can calculate the numbers in Table 4-2 from left to right to simulate the iterative process to get the result of $f(15)$.

For instance, there are two numbers 4 and 2 under the column titled "6". We have two alternatives to make change for 6: the first one is to add a coin with value 1 to a set of coins whose total value is 5. Since the minimal number of coins to get value 5 is 3 (*highlighted* number under the column titled "5"), the number in the first cell under column titled "6" is 4 (4=3+1). The second choice is to add a coin with value 3 to a set of coins whose total value is 3. Since the minimal number of coins to get a value of 3 is 1

(the *highlighted* number under the column tile "3"), the number in the second cell under column title "6" is 2 (2=1+1). We highlight the number 2 in the column under title "6" because 2 is less than 4.

Even though we have a 2-D matrix to show the iterative process, it only requires a 1-D array for coding because it is only necessary to store the minimum number of coins to make change for each total value. The sample code is shown in Listing 4-18.

Listing 4-18. *C# Code for Coin Changes*

```
int GetMinCount(int total, int[] coins) {
    int[] counts = new int[total + 1];
    counts[0] = 0;

    const int MAX = Int32.MaxValue;

    for (int i = 1; i <= total; ++i) {
        int count = MAX;
        for (int j = 0; j < coins.Length; ++j) {
            if (i - coins[j] >= 0 && count > counts[i - coins[j]])
                count = counts[i - coins[j]];
        }

        if (count < MAX)
            counts[i] = count + 1;
        else
            counts[i] = MAX;
    }

    return counts[total];
}
```

Source Code:

```
033_CoinChanges.cs
```

Test Cases:

- There are multiple choices to make change for a total value
- There is only one choice to make change for a total value
- It is not possible to make change for a total value

Minimal Times of Presses on Keyboards

■ **Question 34** Please design an efficient algorithm to lay out cells on phone keyboards in order to minimize key presses.

The current phone keyboard looks like the following layout:

```
             key 2: abc   key 3: def
key 4: ghi   key 5: jkl   key 6: mno
key 7: pqrs  key 8: tuv   key 9: wxyz
```

The first press of a key types the first letter. Each subsequent press advances to the next letter. For example, to type the word "snow", we have to press the key 7 four times, followed by the 6 key twice, followed by the 6 key three times, followed by the 9 key once. The total number of key presses is 10.

This typical solution is not an efficient one. The letter 'i' is used much more frequently than the letter 'w' in English. However, we have to press the 4 key three times to get an 'i', and press the 9 key only once to get a 'w'. If these two letters are interchanged on the keyboard, it will reduce the overall press times.

Let's place letters on a keyboard one by one. It is a greedy choice at each step to select the most frequent letter from the remaining letters and locate it on a key with the lowest index. That is to say, we choose the most frequent letter at the first step and place it as the first letter on a key, and then choose the next most frequent letter and place it as the first letter on another key. If the first locations of all keys are already occupied, a letter is placed as the second one on a key, and so on.

When given the number of keys on a keyboard, as well the frequency of every letter in an alphabet (which is not necessarily English), we may implement the greedy algorithm with the C# code shown in Listing 4-19.

Listing 4-19. *C# Code to Get the Minimal Number Of Key Touches*

```
int MinKeyPress(int keys, int[] frequencies) {
    Array.Sort(frequencies);

    int letters = frequencies.Length;
    int presses = 0;

    // The last element has the highest frequency in
    // an increasingly sorted array
    for(int i = letters - 1; i >= 0; --i) {
        int j = letters - 1 - i;
        presses += frequencies[i] * (j / keys + 1);
    }

    return presses;
}
```

Letters are sorted based on their frequencies in this code. The most frequent letter is the last one in the sorted array, so we begin to select and place letters from the end of the array.

Source Code:

```
034_MinimalPresses.cs
```

Test Cases:

- Functional Tests: Given different sets of characters and various numbers of keys

Bit Operations

People get accustomed to the decimal system in daily life. However, a number is stored as a binary in computers, which is a sequence of 0s and 1s. For example, the decimal 2 is 10 in binary, and decimal 10 is 1010 in binary.

Besides 2 and 10, numbers can be converted with other bases. For instance, the time system has 60 as its base to record seconds, minutes, and hours. The following problem is an interesting interview question about numbers with uncommon bases: in Microsoft Excel, "A" stands for the first column, "B" for the second column, …, "Z" for the 26^{th} column, "AA" for the 27^{th} column, "AB" for the 28^{th} column, and so on. Please write a function to get the column index according to a given string.

A column index in Excel is converted with base 26, so it essentially requires converting a number with base 26 to a decimal number.

Each bit in a binary number, 0 or 1, can be manipulated. Bit operations can be divided into two categories. The first category contains bitwise operations, which include AND, OR, XOR, and NOT. The first three are binary operators taking two input bits, which are summarized in Table 4-3.

Table 4-3. *Bitwise AND, OR, and XOR*

Bitwise AND (&)	0 & 0 = 0	1 & 0 = 0	0 & 1 = 0	1 & 1 = 1
Bitwise OR (|)	0 | 0 = 0	1 | 0 = 1	0 | 1 = 1	1 | 1 = 1
Bitwise XOR(^)	0 ^ 0 = 0	1 ^ 0 = 1	0 ^ 1 = 1	1 ^ 1 = 0

The bitwise NOT is a unary operator, which has only one argument. It produces the opposite of the input bit: a 1 if the input is a 0, or a 0 if the input is a 1.

The second category of bit operations contains shift operations. The left-shift operator in $m << n$ shifts m to the left by n bits and inserts n 0s at the lower-order bits. For example:

```
00001010 << 2 = 00101000
10001010 << 3 = 01010000
```

The right-shift operator in $m >> n$ shifts m to the right by n bits. It inserts n 0s at the higher-order bits if m is positive and inserts n 1s if m is negative. In the following examples, it takes signed 8-bit numbers as the left operand for simplicity:

```
00001010 >> 2 = 00000010
10001010 >> 3 = 11110001
```

The right operator >> is called a signed right-shift in Java, and a new operator >>> named unsigned right-shift is introduced. The operator in $m >>> n$ also shifts m to the right by n bits, and it always inserts 0s at the higher-order bits no matter what the sign of m is. In the following two examples, it takes signed 8-bit numbers as the left operand for simplicity again:

```
00001010 >>> 2 = 00000010
10001010 >>> 3 = 00010001
```

The unsigned right-shift operator >>> is available only in Java, but unavailable in C/C++/C#.

Number of 1s in Binary

■ **Question 35** Please implement a function to get the number of 1s in the binary representation of an integer. For example, the integer 9 is 1001 in binary, so it returns 2 since there are two bits of 1.

It looks like a simple question about binary numbers, and we have many solutions for it. Unfortunately, the most intuitive solution for many candidates is incorrect. We should be careful.

Check the Rightmost Bit with Endless Loop

When candidates are given this problem during an interview, many of them find a solution in a short time: the solution checks whether the rightmost bit is 0 or 1, and then right-shifts the integer one bit and checks the rightmost bit again. It continues in a loop until the integer becomes 0.

How do you check whether the rightmost bit of an integer is 0 or 1? It is simple since we have the AND operation. There is only one bit of 1 in the binary representation of the integer 1, which is the rightmost bit. When we have the bitwise AND operation on an integer and 1, we can check whether the rightmost bit is 0 or 1. When the result of the AND operation is 1, it indicates the rightmost bit is 1; otherwise, it is 0. We can implement a function based on this solution quickly, as shown in Listing 4-20.

Listing 4-20. *C Code for Number of 1 in Binary (Version 1)*

```
int NumberOf1(int n) {
    int count = 0;
    while(n) {
        if(n & 1)
            count ++;

        n = n >> 1;
    }

    return count;
}
```

Interviewers may ask a question when they are told this solution: What is the result when the input integer is a negative number such as 0x80000000? When we right-shift the negative number 0x80000000 for a bit, it becomes 0xC0000000 rather than 0x40000000. The integer 0x8000000 is negative before the shift, so it is guaranteed to be negative after the shift. Therefore, when a negative integer is right-shifted, the first bit is set as 1 after the right-shift operation. If we continue to shift to the right side, a negative integer will be 0xFFFFFFFF eventually and it is trapped in an endless loop.

Left-Shift Operation on 1

If we are using Java, the unsigned right-shift operator >>> can be utilized to eliminate the problem of negative numbers.

We have to utilize other strategies if we are developing in C/C++. Instead of shifting the input integer *n* to right, we may shift the number 1 to left. We may check first the least important bit of the input number *n*, and then shift the number 1 to the left and continue to check the second least important bit of *n*. The code can be revised as shown in Listing 4-21.

Listing 4-21. *C Code for Number of 1 in Binary (Version 2)*

```
int NumberOf1(int n) {
    int count = 0;
    unsigned int flag = 1;
    while(flag) {
        if(n & flag)
            count ++;

        flag = flag << 1;
    }

    return count;
}
```

Minus One and Then Bitwise AND

Let's analyze what happens when you have a binary number minus 1. There is at least one bit 1 in a non-zero number. We first assume the rightmost bit is 1. It becomes 0 if you subtract 1 and other bits then keep unchanged.

Second, we assume the rightmost bit is 0. Since there is at least one 1 bit in a non-zero number, we suppose the m^{th} bit is the rightmost bit of 1. When it is minus 1, the m^{th} bit becomes 0, and all 0 bits behind the m^{th} bit become 1. For instance, the second bit of binary number 1100 is the rightmost 1 bit. When you take 1100 minus 1, the second bit becomes 0, and the third and fourth bits become 1, so the result is 1011.

In both situations above, the rightmost 1 bit becomes 0 when you subtract the 1. When there are some 0 bits on the right side, all of them become 1. The result of the bitwise AND operation on the original number and the minus 1 result is identical to the result gotten by modifying the rightmost 1 to 0. Take the binary number 1100 as an example again. Its result is 1011 when you apply minus 1. The result of the bitwise AND operation on 1100 and 1011 is 1000. If we change the rightmost 1 bit in the number 1100, it also becomes 1000 (Figure 4-6).

```
      1  1  0  0
(–)              1
-------------------
      1  0  1  1
(&)   1  1  0  0
-------------------
      1  0  0  0
```

Figure 4-6. *If we take 1100 minus 1 and then take the bitwise AND operation of the minus result 1011 and the original number 1100, the final result 1000 is the same as changing the rightmost 1 bit of the number 1100 to 0. (Numbers are in the binary representation.)*

The analysis can be summarized as follows: if we first apply minus 1 to a number and apply the bitwise AND operation to the original number and the minus result, the rightmost 1 bit becomes 0. We

repeat these operations until the number becomes 0. We can develop the code in Listing 4-22 accordingly.

Listing 4-22. *C Code for Number of 1 in Binary (Version 3)*

```
int NumberOf1(int n) {
    int count = 0;

    while (n) {
        ++ count;
        n = (n - 1) & n;
    }

    return count;
}
```

The number of times in the `while` loops equals to the number of 1 in the binary format of input `n`.

Source Code:

`035_NumberOf1.c`

Test Cases:

- Positive integers (including the maximum integer as a boundary test case)
- Zero
- Negative integers (including the minimum integer as a boundary test case)

■ **Question 36** Please check whether a number is a power product of 2 in only one C statement.

When a number is 2 to the power of k ($k \geq 0$), there is only one 1 bit in its binary representation. As discussed above, a statement (n-1)&n removes the only 1 bit in the number if n is 2^k. Therefore, the Boolean value of the statement `n != 0 && (n - 1) & n == 0` indicates whether n is 2^k.

■ **Question 37** Given two integers, *m* and *n*, please calculate the number of bits in binary that need to be modified to change *m to n.* For example, the number 10 is 1010 in binary, and 13 is 1101 in binary. We can modify 3 bits of 1010 to get 1101 in binary.

We can modify three bits of 1010 to get 1101 in binary because their last three bits are different. According to the definition of the bitwise XOR operation, it gets 1 when two input bits are different and gets 0 when they are same. Therefore, the result of XOR indicates the bit difference between *m* and *n*. For example, the XOR result of 1010 and 1101 is 0111. Three bits are different between these two numbers, and there are three 1 bits in their XOR result.

After we get the XOR result, the remaining task is to count 1s in a number that has been discussed already, as shown in Listing 4-23.

Listing 4-23. *C Code to Modify a Number to Another*

```
int bitsToModify(int number1, int number2) {
    int temp = number1 ^ number2;

    // the number of 1 bits in temp
    int bits = 0;
    while(temp != 0) {
        ++bits;
        temp = (temp - 1) & temp;
    }

    return bits;
}
```

Source Code:

```
037_ModifyANumberToAnother.c
```

Test Cases:

- One or two numbers are positive, zero, or negative

Numbers Occurring Only Once

■ **Question 38** Let's assume all numbers except two occur twice in an array. How do you get those two numbers to occur only once in O(*n*) time and O(1) space?

For example, only two numbers, 4 and 6, in the array {2, 4, 3, 6, 3, 2, 5, 5} occur once, and the others numbers occur twice. Therefore, the output should be 4 and 6.

This is a very difficult interview question. When an interviewer notices that a candidate does not have any ideas after a few minutes, it is possible for the interviewer to modify the question a little bit: If all numbers except one occur twice in the array, how do you find the only number occurring only once?

Why does the interviewer emphasize the times a number occurs (once or twice)? It reminds us of the characteristic of the bitwise XOR operation: it gets 1 when two input bits are different and gets 0 when they are same. Therefore, XOR gets 0 on a pair of duplicated numbers. If numbers in an array where all numbers except one occur twice are XORed, the result is exactly the number occurring only once.

With the simplified problem solved, let's return back to the original one. We know how to solve it if we could partition the array into two sub-arrays, where only one number appears once and the others appear twice. The only problem is how to partition the array into two.

Supposing the two numbers occurring once in the array are *num1* and *num2*. If we take XOR operations on all numbers in the array, the result should be the same as the result of *num1^num2*

because XOR gets 0 on pairs of duplicated numbers. The result of *num1^num2* is not 0 because *num1* and *num2* are two different numbers. There is a 1 bit at least in the XOR result.

Let's denote the first 1 bit in the XOR result as the i^{th} bit. Numbers in the array are partitioned according to their i^{th} bits. The i^{th} bits of all numbers in the first sub-array are 1s, and the i^{th} bits of all numbers in the second sub-array are 0s. Two duplicated numbers in a pair are in either the first or the second sub-array, but *num1* and *num2* cannot be in the same sub-array.

Take the sample array {2, 4, 3, 6, 3, 2, 5, 5} as an example. The XOR result of all numbers in the array is 0010 in binary, of which the second bit from the right is 1. Numbers in the array are partitioned into two sub-arrays: the first one is {2, 3, 6, 3, 2} where the numbers all have the property that the second bit from the right is 1, and the second one is {4, 5, 5}, which are numbers where the second bit from the right is 0. If we use XOR operations on these two sub-arrays, we can get the two numbers 6 and 4 occurring only once.

It is time to write code after we have confirmed our ideas about how to partition an array, as shown in 4-24.

Listing 4-24. *Java Code to Get Numbers Occurring Only Once*

```
void getOnce(int numbers[], NumbersOccurringOnce once){
    if (numbers.length < 2)
        return;

    int resultExclusiveOR = 0;
    for (int i = 0; i < numbers.length; ++ i)
        resultExclusiveOR ^= numbers[i];

    int indexOf1 = findFirstBitIs1(resultExclusiveOR);

    once.num1 = once.num2 = 0;
    for (int j = 0; j < numbers.length; ++ j) {
        if(isBit1(numbers[j], indexOf1))
            once.num1 ^= numbers[j];
        else
            once.num2 ^= numbers[j];
    }
}

// The first 1 bit from the rightmost
int findFirstBitIs1(int num){
    int indexBit = 0;
    while (((num & 1) == 0) && (indexBit < 32)) {
        num = num >> 1;
        ++ indexBit;
    }

    return indexBit;
}

// check whether the bit with index indexBit is 1
boolean isBit1(int num, int indexBit) {
    num = num >> indexBit;
    return (num & 1) == 1;
```

```
}

class NumbersOccurringOnce {
    public int num1;
    public int num2;
}
```

Source Code:

    038_NumbersOccuringOnce.java

Test Cases:

- Functional cases: Numbers in a array appear twice/four times except two numbers occurring once
- Boundary cases: An array only has two unique numbers

■ **Question 39** Two numbers out of *n* numbers from 1 to *n* are missing. The remaining *n*-2 numbers are restored in an array, not in any particular order. Please write a method (or a function) to find the missing two numbers.

Based on Arithmetic Calculation

Supposing the two missing numbers are *num1* and *num2*. It is easy to get the sum when adding all numbers in the array denoted as S_1 as well as the result of 1+2+…+*n* denoted as S_2. Let's denote S_2-S_1 as *s*. It is also easy to get the product when multiplying all numbers in the array denoted as P_1 as well as the result of 1×2×…×*n* denoted as P_2. Similarly, we denote P_2/P_1 as *p*.

Now we have the following pair of linear equations:

$$\begin{cases} num1 + num2 = s \\ num1 \times num2 = p \end{cases}$$

After solving this linear equations, we get *num1*= $\frac{s+\sqrt{s^2-4p}}{2}$ and *num2*= $\frac{s-\sqrt{s^2-4p}}{2}$. Therefore, this solution can be implemented with the code shown in Listing 4-25.

Listing 4-25. *Java Code to Get Two Missing Numbers (Version 1)*

```
void findMissing_solution1(int numbers[], NumbersOccurringOnce missing){
    int sum1 = 0;
    int product1 = 1;
    for(int i = 0; i < numbers.length; ++i){
        sum1 += numbers[i];
        product1 *= numbers[i];
    }

    int sum2 = 0;
```

```
    int product2 = 1;
    for(int i = 1; i <= numbers.length + 2; ++i){
        sum2 += i;
        product2 *= i;
    }

    int s = sum2 - sum1;
    int p = product2 / product1;

    missing.num1 = (s + (int)(Math.sqrt(s * s - 4 * p))) / 2;
    missing.num2 = s - missing.num1;
}
```

This solutions works when the range of numbers from 1 to n is relatively narrow. However, it causes overflow errors to calculate $1+2+\ldots+n$ and $1\times2\times\ldots\times n$ when n is large. Let's look for a more robust solution.

Based on Bit Operations

There are n-2 numbers in an array in the range from 1 to n missing two numbers *num1* and *num2*. We define another array with size 2n-2, of which the first n-2 numbers are copies of numbers in the original array and others are numbers from 1 to n. All numbers except two numbers, *num1* and *num2*, occur twice in the new array, and *num1* and *num2* occur only once. Therefore, this problem is equivalent to the preceding problem: find two numbers occurring once in an array where others occur twice. Let's solve it based on bit operations with the method `getOnce`, borrowed from the solution of the preceding problem, as shown in Listing 4-26.

Listing 4-26. *Java Code to Get Two Missing Numbers (Version 2)*

```
void findMissing_solution2(int numbers[], NumbersOccurringOnce missing){
    int originalLength = numbers.length;
    int extendedLength = originalLength * 2 + 2;
    int extention[] = new int[extendedLength];

    for(int i = 0; i < originalLength; ++i)
        extention[i] = numbers[i];
    for(int i = originalLength; i < extendedLength; ++i)
        extention[i] = i - originalLength;

    getOnce(extention, missing);
}
```

Source Code:

```
    039_FindTwoMissingNumbers.java
```

Test Cases:

- Functional tests: Two numbers out of n numbers from 1 to n are missing in an array
- Boundary tests: An array only has one or two numbers

- Robust tests: The maximum number n is very big

Summary

Similar to data structures, there are many coding interview questions about common algorithms. Many algorithms can be implemented based on both recursion and iteration. Usually, recursion looks more concise, but iteration is more efficient and robust with big data inputs.

When it is required to find or count a number in a sorted array, binary search might be the right choice. Merge sort and quicksort are the most important sort algorithms for interviews. The `Partition` method is useful not only in quicksort, but also to select the arbitrary k^{th} number out of an array.

The backtracking algorithm is used to search for a solution to a problem among all available options in depth-first order. A backtracking solution is implemented with recursion in most cases.

Two hot interview topics are dynamic programming and greedy algorithms, which are quite helpful tools to get an optimized solution (the minimum or maximum value) to a problem. If there is a greedy choice available at each step, the greedy algorithm works; otherwise, we may have a try dynamic programming.

Bit operations can be viewed as special algorithms on binary numbers. There are four bitwise operators, AND, OR, XOR, and NOT, as well as right-shift and left-shift to manipulate bits.