

Username: Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

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 $1+2+\dots+n$. The first strategy is a utilization of recursion, as shown in [Listing 4-1](#).

Listing 4-1. Recursive C Code to Calculate $1+2+\dots+n$


```
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 $1+2+\dots+n$

```
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 $1+2+\dots+n$ 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 $1+2+\dots+n$ 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).

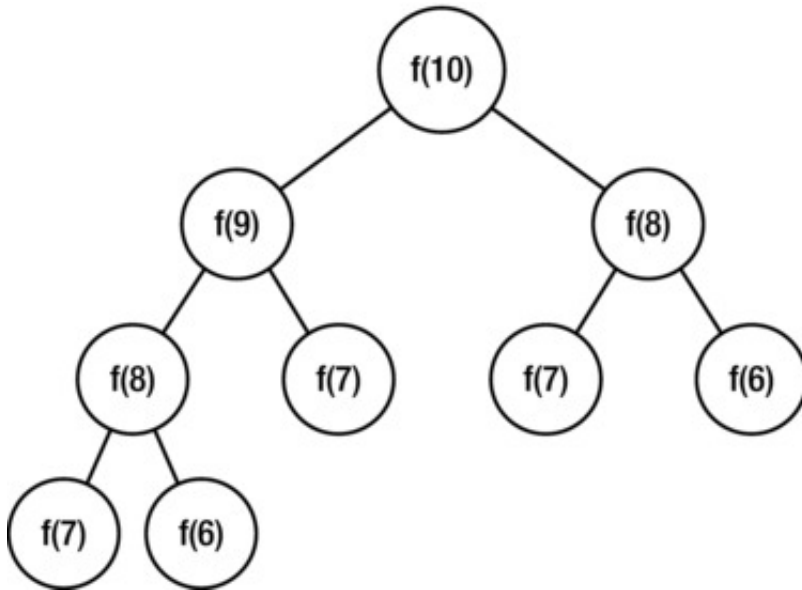


Figure 4-1. The recursive process to calculate the 10th element in the Fibonacci Sequence

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 fibMinusOne = 1;

    long long fibMinusTwo = 0;

    long long fibN = 0;

    unsigned int i;

    if(n < 2)

        return result[n];

    for(i = 2; i <= n; ++ i) {

        fibN = fibMinusOne + fibMinusTwo;

        fibMinusTwo = fibMinusOne;

        fibMinusOne = 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.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

With the equation above, it finds $f(n)$ if it finds $f(n-1)$ and $f(n-2)$. 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 $f(n-1)$ choices for the remaining $n-1$ levels. The other one is to jump two levels at the first step, and it has $f(n-2)$ choices for the remaining $n-2$ levels. Therefore, the total number of choices on a stair with n levels is $f(n) = f(n-1) + f(n-2)$, 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 $f(n) = f(n-1) + f(n-2) + \dots + f(1) + 1$. Since $f(n-1) = f(n-2) + \dots + f(1) + 1$, $f(n) = 2f(n-1)$. 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, $1 \ll (n-1)$, 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.