算法的时间复杂度和空间复杂度

1.算法效率

2.时间复杂度

3.空间复杂度

1.算法效率

1.1 算法的复杂度

算法在编写成可执行程序后，运行时需要耗费时间资源和空间(内存)资源 。因此**衡量一个算法的好坏，一般**

是从时间和空间两个维度来衡量的，即时间复杂度和空间复杂度。

时间复杂度主要衡量一个算法的运行快慢，而空间复杂度主要衡量一个算法运行所需要的额外空间。在计算

机发展的早期，计算机的存储容量很小。所以对空间复杂度很是在乎。但是经过计算机行业的迅速发展，计

算机的存储容量已经达到了很高的程度。所以我们如今已经不需要再特别关注一个算法的空间复杂度。

摩尔定律是英特尔创始人之一戈登·摩尔的经验之谈，其核心内容为：集成电路上可以容纳的晶体管数目在大约每经过18个月便会增加一倍。

2.时间复杂度

2.1 时间复杂度的概念

时间复杂度的定义：在计算机科学中，**算法的时间复杂度是一个函数**，(数学里面带有未知数的函数表达式)它定量描述了该算法的运行时间。一个算法执行所耗费的时间，从理论上说，是不能算出来的，只有你把你的程序放在机器上跑起来，才能知道。但是我们需要每个算法都上机测试吗？是可以都上机测试，但是这很麻烦，所以才有了时间复杂度这个分析方式。一个算法所花费的时间与其中语句的执行次数成正比例，**算法中的基本操作的执行次数，为算法的时间复杂度。**

冒泡排序，对100w个数进行排序
10年前2核cpu、2g内存的机器
今天8核cpu、8g内存的机器
时间一样吗?
环境不同，具体运行时间就不同

```
// 请计算一下Func1中++count语句总共执行了多少次？
void Func1(int N)
{
    int count = 0;
    for (int i = 0; i < N ; ++ i)
    {
        for (int j = 0; j < N ; ++ j)
        {
            ++count;
```

```
        }
    }

    for (int k = 0; k < 2 * N ; ++ k)
    {
        ++count;
    }
    int M = 10;
    while (M--)
    {
        ++count;
    }
    printf("%d\n", count);
}
```

Func1 执行的基本操作次数：

$$F(N) = N^2 + 2 * N + 10$$

N越大后两项对结果的影响越小

N = 10 F(N) = 130

N = 100 F(N) = 10210

N = 1000 F(N) = 1002010

时间复杂度 O(N^2)

实际中我们计算时间复杂度时，我们其实并不一定要计算精确的执行次数，而只需要**大概执行次数，那么这**

里我们使用大O的渐进表示法。(估算)

2.2 大O的渐进表示法

大O符号（Big O notation）：是用于描述函数渐进行为的数学符号。

推导大O阶方法:

1、用常数1取代运行时间中的所有加法常数。

2、在修改后的运行次数函数中，只保留最高阶项。

3、如果最高阶项存在且不是1，则去除与这个项目相乘的常数。(系数）得到的结果就是大O阶

实例1:

```
// 计算Func2的时间复杂度?
void Func2(int N) {
 int count = 0;
 for (int k = 0; k < 2 * N ; ++ k)
 {
 ++count;
 }
 int M = 10;
 while (M--)
 {
 ++count;
```

```
}
printf("%d\n", count);
}
```

2N+10　O(N)

实例2:

```
// 计算Func3的时间复杂度?
void Func3(int N, int M) {
 int count = 0;
 for (int k = 0; k < M; ++ k)
 {
 ++count;
 }
 for (int k = 0; k < N ; ++ k)
 {
 ++count;
 }
 printf("%d\n", count);
}
```

没有说明M和N的大小关系

O(M+N)

一般情况下时间复杂度计算时未知数都是用的N

但是也可以时M、K等等其他的
M远大于N-> O(M)
N远大于M -> O (N)
M和N差不多大 ->O(M)或者O(N)

实例3:

```
// 计算Func4的时间复杂度?
void Func4(int N) {
 int count = 0;
 for (int k = 0; k < 100; ++ k)
 {
 ++count;
 }
 printf("%d\n", count);
}
```

O(1) 不是代表算法运行一次，而是常数次

实例4:

```
// 计算strchr的时间复杂度?
const char * strchr ( const char * str, int character );
```

```
while(*str)
{
        if(*str == character)
             return str;
        else
           ++str;
}
```

O (N)

hello world

假设查找的是h　1　　最好情况：任意输入规模的最小运行次数(下界)

假设查找的是w　N/2　　平均情况：任意输入规模的期望运行次数

假设查找的是d　N　　最坏情况：任意输入规模的最大运行次数(上界)

当一个算法随着输入不同，时间复杂度不同，时间复杂度做悲观预期，看最坏的情况

实例5：

```
// 计算BubbleSort的时间复杂度？
void BubbleSort(int* a, int n) {
 assert(a);
 for (size_t end = n; end > 0; --end)
 {
 int exchange = 0;
 for (size_t i = 1; i < end; ++i)
 {
 if (a[i-1] > a[i])
 {
 Swap(&a[i-1], &a[i]);
 exchange = 1;
 }
 }
 if (exchange == 0)
 break;
 }
}
```

```
// 计算BubbleSort的时间复杂度？
void BubbleSort(int* a, int n)
{
    assert(a);
    for (size_t end = n; end > 0; --end)
    {
        int exchange = 0;
        for (size_t i = 1; i < end; ++i)
        {
            if (a[i-1] > a[i])
            {
                Swap(&a[i-1], &a[i]);
                exchange = 1;
            }
        }

        if (exchange == 0)
            break;
    }
}
```

精确：F(N) = N*(N-1)/2

时间复杂度： O(N^2)

N-1
N-2
N-3
..
1

等差数列首项加尾项乘以项数除以2

实例6：

```
// 计算BinarySearch的时间复杂度？
int BinarySearch(int* a, int n, int x) {
 assert(a);
 int begin = 0;
 int end = n-1;
 while (begin < end)
 {
 int mid = begin + ((end-begin)>>1);
 if (a[mid] < x)
```

```
    begin = mid+1;
    else if (a[mid] > x)
    end = mid;
    else
    return mid;
    }
    return -1; }
```

实例6:

```
// 计算BinarySearch的时间复杂度?
int BinarySearch(int* a, int n, int x)
{
    assert(a);

    int begin = 0;
    int end = n;
    while (begin < end)
    {
        int mid = begin + ((end-begin)>>1);
        if (a[mid] < x)
            begin = mid+1;
        else if (a[mid] > x)
            end = mid;
        else
            return mid;
    }

    return -1;
}
```

算时间复杂度不能只去看是几层循环，而要去看他的思想

时间复杂度： O(N)　　$O(\log_2 N)$

X

怎么算的呢?
大家可以先自己思考一下
我们休息一下，一会再看

最好情况：O(1)
最坏情况：O()

假设查找X次

二分查找的思想是N每找一次就除以2，最后除到1，除了多少次程序就执行了多少次

N/2/2/2... = 1
2^X = N

$X = \log_2 N$

1 * 2 * 2* 2 ..... = N
2^X = N

$X = \log_2 N$

二分查找算法是一个非常牛逼的算法

N个数中查找	大概查找次数
1000	10
100W	20
10亿	30

$2^{10} = N$
$2^{20} = N$
$2^{30} = N$

树->二叉树->搜索二叉树->平衡搜索二叉树->AVLTree RBTree

$\log_2 N$

哈希表

B树系列

14亿
查找一个人，最多多少次
31

2^31 约等于 20亿

实例7:

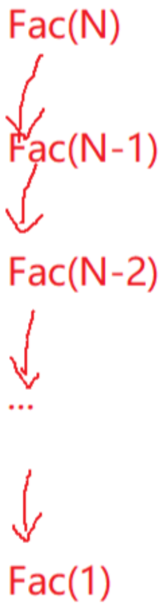
```
// 计算阶乘递归Fac的时间复杂度?
long long Fac(size_t N) {
 if(0 == N)
 return 1;

 return Fac(N-1)*N; }
```

实例7:

递归算法：递归次数*每次递归调用的次数

```
// 计算阶乘递归Fac的时间复杂度？
long long Fac(size_t N)
{
 if(0 == N)
     return 1;

 return Fac(N-1)*N;
}
```

O(N)



1

实例8：

```
// 计算斐波那契递归Fib的时间复杂度？
long long Fib(size_t N) {
 if(N < 3)
 return 1;

 return Fib(N-1) + Fib(N-2);
}
```

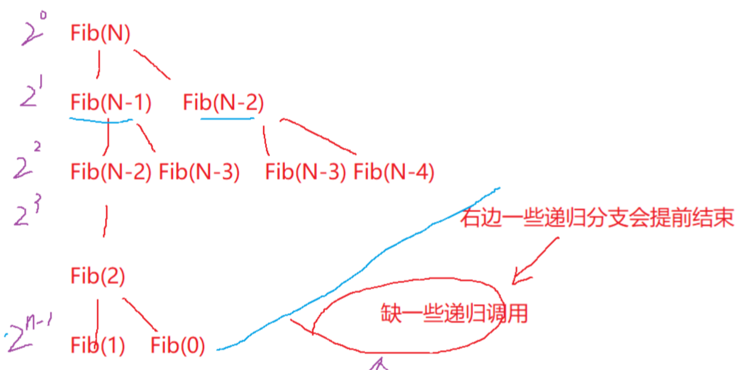
无须关心具体调用次数，每个函数它每次递归的次数都是O(1)，只需要计算它递归总共调用函数的次数

Fac(1)

1

实例8: 递归次数*每次递归调用的次数

```
// 计算斐波那契递归Fib的时间复杂度？
long long Fib(size_t N)
{
 if(N < 3)
     return 1;

 return Fib(N-1) + Fib(N-2);
}
```



$$Fib(N) = \underbrace{2^0 + 2^1 + 2^2 + ... + 2^{(N-1)}}_{2^N - 1} - X$$

X远小于2^N

时间复杂度：O(2^N)

(5) 等比求和：$S_n = a_1 + a_2 + a_3 + \ldots + a_n$

①当q≠1时，$S_n = \frac{a_1\left(1-q^n\right)}{1-q}$ 或 $S_n = \frac{\left(a_1 - a_n q\right)}{1-q}$

②当q=1时，$S_n = n \times a_1$

记 $\pi_n = a_1 \cdot a_2 \cdots a_n$ ，则有

在这个意义下，我们说：一个正项等比数列与等差数列是“同构”的。

```c
#include <stdio.h>
// O(2^N)
// 斐波那契额数列的递归写法完全一个实际没用的算法，因为太慢了
// 计算阶乘递归Fac的时间复杂度？
long long Fac(size_t N)
{
    if (0 == N)
        return 1;

    return Fac(N - 1) * N;
}
int main()
{
    printf("%lld\n", Fac(10));
    return 0;
}
```

实例答案及分析：

1. 实例1基本操作执行了2N+10次，通过推导大O阶方法知道，时间复杂度为 O(N)
2. 实例2基本操作执行了M+N次，有两个未知数M和N，时间复杂度为 O(N+M)
3. 实例3基本操作执行了10次，通过推导大O阶方法，时间复杂度为 O(1)
4. 实例4基本操作执行最好1次，最坏N次，时间复杂度一般看最坏，时间复杂度为 O(N)
5. 实例5基本操作执行最好N次，最坏执行了(N*(N+1)/2次，通过推导大O阶方法+时间复杂度一般看最

   坏，时间复杂度为 O(N^2)
6. 实例6基本操作执行最好1次，最坏O(logN)次，时间复杂度为 O(logN) ps：logN在算法分析中表示是底

   数为2，对数为N。有些地方会写成lgN。（建议通过折纸查找的方式讲解logN是怎么计算出来的）
7. 实例7通过计算分析发现基本操作递归了N次，时间复杂度为O(N)。
8. 实例8通过计算分析发现基本操作递归了2^ N次，时间复杂度为O(2^N)。（建议画图递归栈帧的二叉树讲解）

3.空间复杂度

空间复杂度也是一个数学表达式，是对一个算法在运行过程中**临时额外占用存储空间大小的量度** 。

空间复杂度**不是程序占用了多少bytes的空间**，因为这个也没太大意义，所以空间复杂度算的是**变量的个数**。

空间复杂度计算规则基本跟实践复杂度类似，也使用**大O渐进表示法**。

注意：**函数运行时所需要的栈空间*(存储参数、局部变量、一些寄存器信息等)在编译期间已经确定好了，因**

此空间复杂度主要通过函数在运行时候显式申请的额外空间来确定。

实例1、：

```
// 计算BubbleSort的空间复杂度?
void BubbleSort(int* a, int n) {
 assert(a);
 for (size_t end = n; end > 0; --end) 1234567812345678 12345
 {
    int exchange = 0;
    for (size_t i = 1; i < end; ++i)
    {
        if (a[i-1] > a[i])
        {
            Swap(&a[i-1], &a[i]);
            exchange = 1;
        }
    }
        if (exchange == 0)
        break;
 }
}
```

int* a int n 算法本身需要的空间 计算额外需要的空间大小，额外需要的是计算算法时定义的变量

空间复杂度O(1) **先定义end，进行这一次迭代，在定义 i,循环结束后 i就销毁了，**

在循环上来下来的时侯，在定义 i 和之前的 i 用的同一块空间， 再加上exchange 额外使用三个空间(常数个)

实例2：

```
// 计算Fibonacci的空间复杂度?
// 返回斐波那契数列的前n项   ==n个数的数组==   如果求第N个时间复杂度还可以优化到O(N)
long long* Fibonacci(size_t n) {
    if(n==0)
    return NULL;

    long long * fibArray = (long long *)malloc((n+1) * sizeof(long long));
    fibArray[0] = 0;
    fibArray[1] = 1;
    for (int i = 2; i <= n ; ++i)
    {
        fibArray[i] = fibArray[i - 1] + fibArray [i - 2];
    }
     return fibArray;
```

```
}
```

空间复杂度O(N) 额外malloc n+1 个空间（数组的空间） + i 影响最大的是N,其他项忽略掉

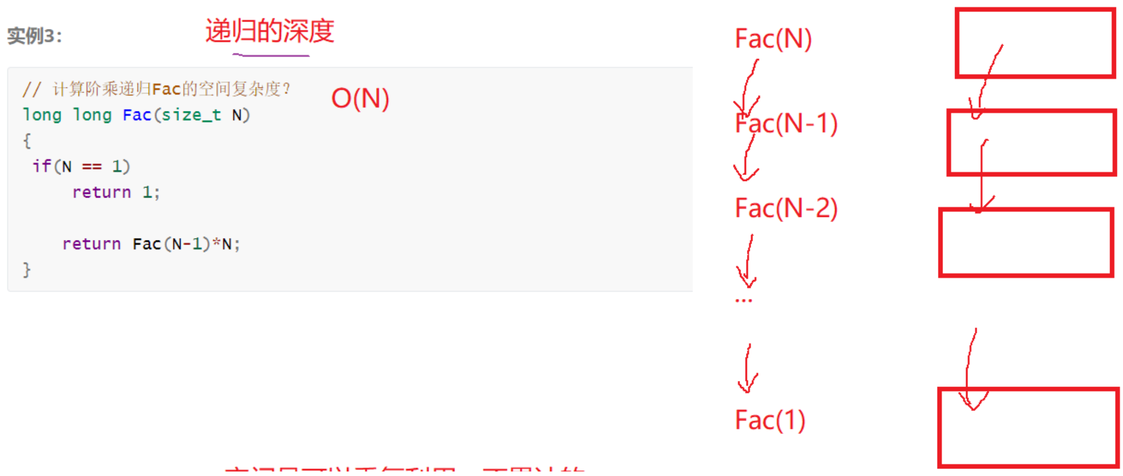
时间复杂度O(N) 2到N次 执行N-2次

实例答案及分析：

1. 实例1使用了常数个额外空间，所以空间复杂度为 O(1)
2. 实例2动态开辟了N个空间，空间复杂度为 O(N)

实例3：

```
// 计算阶乘递归Fac的空间复杂度？
long long Fac(size_t N) {
 if(N == 0)
 return 1;

 return Fac(N-1)*N; }
```



实例3递归调用了N次，开辟了N个栈帧，每个栈帧使用了常数个空间。空间复杂度为O(N)

实例4：

```
//计算斐波那契数的空间复杂度
long long Fac(size_t N)
{
    if (0 == N)
        return 1;

    return Fac(N - 1) * N;
}
```

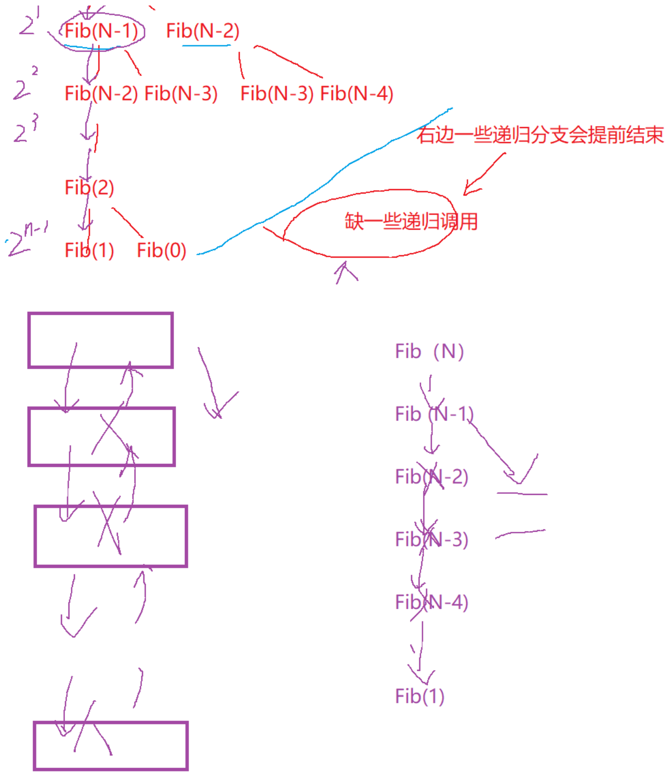
栈空间是不大的，linux系统下栈的空间只有8M，所以空间复杂度不可能是2^ N，2^N很快就崩了，

```c
long long Fac(size_t N) {
	if (0 == N)
		return 1;

	return Fac(N - 1) * N;
}
int main()
{
	printf("%lld\n", Fac(100000));
	return 0;
}
```

- 右边递归和左边递归用的是同一块空间，空间使用完可以还给操作系统，时间不可以
- **递归是一条路计算完后退回来在计算下一条路**，先计算N-1这一串递归，计算完递归退回来后在计算f(N-2),使用的是同一块空间

```c
// 计算斐波那契递归Fib的复杂度?
long long Fib(size_t N)
{
 if(N < 3)
     return 1;

 return Fib(N-1) + Fib(N-2);
}
```

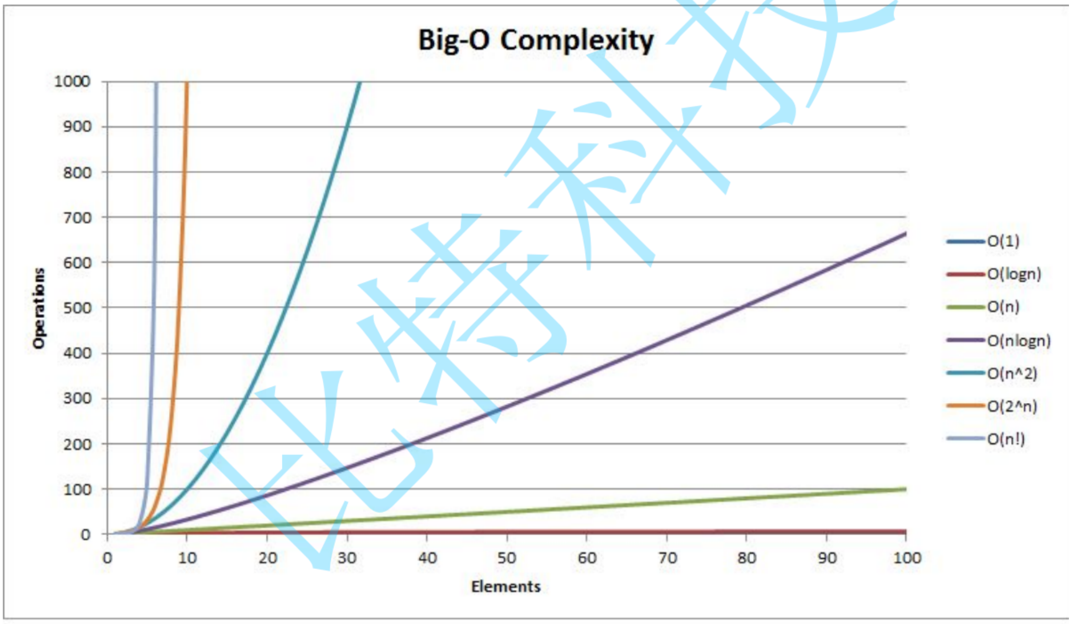
O(N)



11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111

一般算法常见的复杂度如下：

5201314	O(1)	常数阶
3n+4	O(n)	线性阶
3n^2+4n+5	O(n^2)	平方阶
3log(2)n+4	O(logn)	对数阶
2n+3nlog(2)n+14	O(nlogn)	nlogn阶
n^3+2n^2+4n+6	O(n^3)	立方阶
2^n	O(2^n)	指数阶



5. 复杂度的oj练习

5. 复杂度的oj练习

3.1消失的数字OJ链接：https://leetcode-cn.com/problems/missing-number-lcci/

面试题 17.04. 消失的数字

难度 简单 51

数组 nums 包含从 0 到 n 的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在O(n)时间内完成吗？

注意： 本题相对书上原题稍作改动

示例 1：

```
输入：[3,0,1]
输出：2
```

示例 2：

```
输入：[9,6,4,2,3,5,7,0,1]
输出：8
```

1 2 1 2 3

1 3 2 2 1

x = 0

思路1：排序 -》 qsort 快排->时间复杂度$O(n*\log_2 N)$

思路2：(0+1+2+3...+n) - (a[0]+a[1]+[2]+a[n-1])

时间复杂度O(N)　空间复杂度：O(1)

思路3：数组中值是几就在第几个位置写一下这个值

0 1 2 n

空间复杂度：O（N）　时间复杂度：O(N)

思路4：给一个值x = 0

x先跟[0,n]的所有值异或　时间复杂度O(N)

x在跟数组中每个值异或　最后x就是缺的那个数字

一道题有多种方法，那么我们不用实现，只需要分析出每种方法的复杂度

选择复杂度优的方式即可，这样就是复杂度在实际中意义

[0, 9]

```
输入：[9,6,4,2,3,5,7,0,1]
输出：8
```

x = 0

[0, 9]

```
输入：[9,6,4,2,3,5,7,0,1]
输出：8
```


```c
int missingNumber(int* nums, int numsSize){
    int x = 0;
    // 跟[0, n]异或
    for(int i= 0; i <= numsSize; ++i)
    {
        x ^= i;
    }

    // 再跟数组中值异或
    for(int i= 0; i < numsSize; ++i)
    {
        x ^= nums[i];
    }

    return x;
}
```

189. 旋转数组

难度 中等 1153

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

进阶：

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 你可以使用空间复杂度为 O(1) 的 **原地** 算法解决这个问题吗？

示例 1:

```
输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]
解释:
向右旋转 1 步: [7,1,2,3,4,5,6]
向右旋转 2 步: [6,7,1,2,3,4,5]
向右旋转 3 步: [5,6,7,1,2,3,4]
```

旋转1次

tmp 7

[1,2,3,4,5,6,7],

tmp 7

[7 1,2,3,4,5,6],

思路1：暴力求解，旋转K次

时间复杂度：O（N*K） 空间复杂度：O(1)

思路2：开辟额外空间 以空间换时间

```
输入: nums = [1,2,3,4,5,6,7], k = 3
```

tmp $\boxed{5,6,7\ \ 1,2,3,4}$

```
输出: [5,6,7,1,2,3,4]
```

时间复杂度：O（N） 空间复杂度： O(N)

7

10

思路3：

k == n时，不旋转就是要的结果

```
输入: nums = [1,2,3,4,5,6,7], k = 3
输出: [5,6,7,1,2,3,4]
```

0 3 4 6

4 3 2 1 5 6 7 前n-k个逆置

4 3 2 1 7 6 5 后k个逆置

5 6 7 1 2 3 4 整体逆置

时间复杂度：O(N) 空间复杂度：O（1）

```c
void rotate(int* nums, int numsSize, int k)
{
    if(k >= numsSize)
        k %= numsSize;

    // 前n-k个数逆置
    Reverse(nums, 0, numsSize-k-1);
    // 后k个逆置
    Reverse(nums, numsSize-k, numsSize-1);
    // 整体逆置
    Reverse(nums, 0, numsSize-1);
}
```

n = 7， k = 3