1--时间复杂度和空间复杂度计算

【本节目标】

- **1.什么是时间复杂度和空间复杂度?**
- **2.如何计算常见算法的时间复杂度和空间复杂度?**
- **3.有复杂度要求的算法题练习**

1.什么是时间复杂度和空间复杂度?

> ### 1.1算法效率
>
> **算法效率分析分为两种：第一种是时间效率，第二种是空间效率。** 时间效率被称为时间复杂度，而空间效率被称作空间复杂度。 时间复杂度主要衡量的是一个算法的运行速度，而空间复杂度主要衡量一个算法所需要的额外空间，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。
>
> ### 1.2 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，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。**
>
> ### 1.3 空间复杂度的概念
>
> 空间复杂度是对一个算法在运行过程中**临时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。**
>
> ### 1.4 复杂度计算在算法的意义
>
> 腾讯C++后台开发实习凉经
>
> 2天前 71
>
> 【一面】2小时
>
> 1. vector扩容机制
> 2. 红黑树了解吗，红黑树的插入、删除（直接放弃）
> 3. hash冲突是什么，如果单链长度太长怎么办
> 4. 快排堆排归并时间复杂度，快排最坏的情况，怎么推导的
> 5. 进程间通信的区别，消息队列，共享内存，管道
> 6. 三个编程题：strstr，堆排，实现hash的插入、查询、删除
> 7. select、epoll了解吗
> 8. 文件系统了解吗
> 9. erase删除会返回什么
> 10. const作用，const变量一定不能改嘛
> 11. exteren “c”作用
> 12. static关键字作用，修饰函数时作用
> 13. quic是什么？ 这个听都没听说过
> 14. 让我设计一个买票系统，不知道咋设计，gameover
> 15. TCP三次握手，滑动窗口，我给他画了图，他让我别画了
>
> 剑指 Offer 56 - I. 数组中数字出现的次数
>
> 难度 中等 361
>
> 一个整型数组 nums 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。
>
> 校园招聘的在笔试算法题和面试中都会考察对复杂度的计算和理解
>
> **示例 1:**
>
> ```
> 输入: nums = [4,1,4,6]
> 输出: [1,6] 或 [6,1]
> ```

2.1如何计算常见算法的时间复杂度?

2.2 大O的渐进表示法

```c
// 请计算一下Func1基本操作执行了多少次?
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 = 10 F(N) = 130
- N = 100 F(N) = 10210
- N = 1000 F(N) = 1002010

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。**

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。得到的结果就是大O阶。

使用大O的渐进表示法以后，Func1的时间复杂度为:

$$O(N^2)$$

- N = 10 F(N) = 100
- N = 100 F(N) = 10000
- N = 1000 F(N) = 1000000

通过上面我们会发现大O的渐进表示法**去掉了那些对结果影响不大的项**，简洁明了的表示出了执行次数。

另外有些算法的时间复杂度存在最好、平均和最坏情况：

最坏情况：任意输入规模的最大运行次数(上界)

平均情况：任意输入规模的期望运行次数

最好情况：任意输入规模的最小运行次数(下界)

例如：在一个长度为N数组中搜索一个数据x

最好情况：1次找到

最坏情况：N次找到

平均情况：N/2次找到

在实际中一般情况关注的是算法的最坏运行情况，所以数组中搜索数据时间复杂度为O(N)

2.3常见时间复杂度计算举例

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);
}
```

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);
}
```

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);
}
```

实例4:

```
// 计算strchr的时间复杂度?
const char * strchr ( const char * str, char character )
{
  while(*str != '\0')
  {
      if(*str == character)
          return str;

      ++str;
  }

  return NULL;
}
```

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;
    }
```

```
}
```

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;
}
```

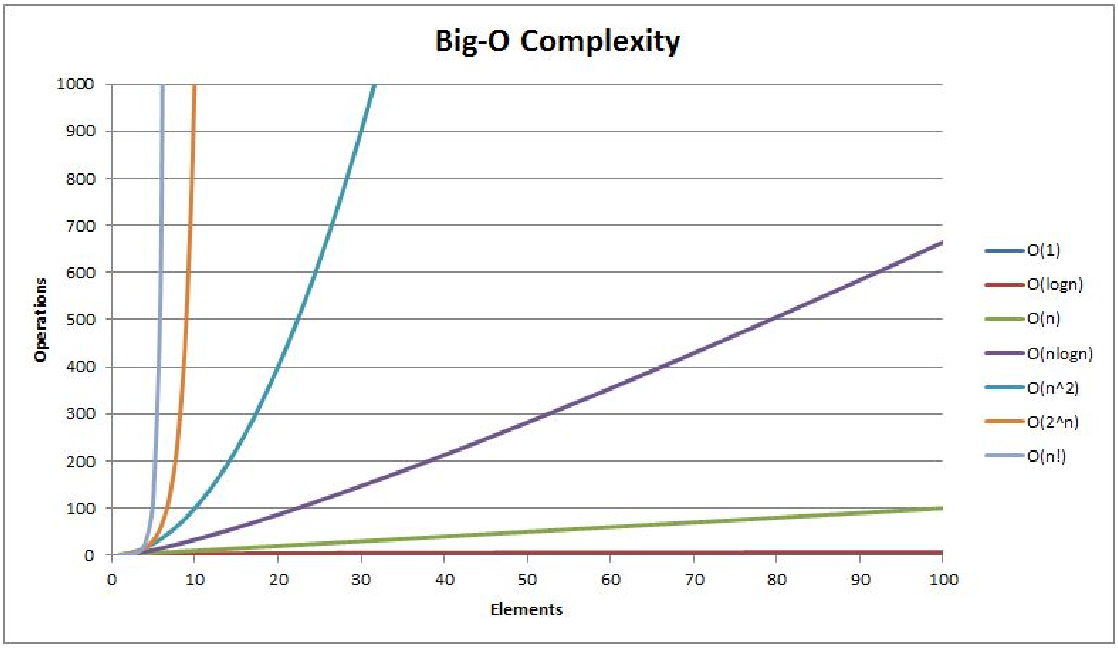
实例7：

```
// 计算阶乘递归Factorial的时间复杂度?
long long Factorial(size_t N)
{
    return N < 2 ? N : Factorial(N-1)*N;
}
```


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)。

复杂度对比：



2.2.常见空间复杂度的计算

空间复杂度是对一个算法在运行过程中**临时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**。

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)
    {
        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;
    }
}
```

实例2：

```
// 计算Fibonacci的空间复杂度？
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 ;
}
```

实例3：

```
// 计算阶乘递归Factorial的空间复杂度？
long long Factorial(size_t N)
{
    return N < 2 ? N : Factorial(N-1)*N;
}
```

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)
3. 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)

3.有复杂度要求的算法题练习

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

示例 1：

```
输入：[3,0,1]
输出：2
```

```
0 1 2 3
3 0 1
```

示例 2：

```
输入：[9,6,4,2,3,5,7,0,1]
输出：8
```

```
0 1 2 3 4 5 6 7 8 9
9 6 4 2 3 5 7 0 1
```

3.2 旋转数组OJ链接：https://leetcode-cn.com/problems/rotate-array/

思路一：

```
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]
```

思路二：

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

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