

# 4-复杂度

C生万物 ● 大道至简 ● 鲍鱼科技+v(15339278619)

## 1、目标

- 📌 重点掌握时间复杂度的计算
- 掌握空间复杂度的计算

## 2、复杂度的概述

- 📌 **时间复杂度**和**空间复杂度**是程序性能的重要指标，衡量一个程序的好坏，就看这两个指标，时间越少，空间越小，是一个最理想的好程序，但往往时间和空间是一对矛盾，不可兼得，因此经常有牺牲空间换时间，或者牺牲时间换空间的说法，其实，程序要做的就是在具体的环境下，寻求一种时间和空间的平衡，最终让性能达到最优。

**复杂度的计算分为两种：**

- 1、事后统计的方法
- 2、事前的分析估算

**事后统计**需要每一个程序都上环境进行测试，况且不同的环境所运行的时间还会有所不同，不现实

**事前分析**是一种估算，只取对程序影响最大的维度来衡量程序的效率，具有参考价值，所以时间复杂度和空间复杂度都是事前分析估算的方法

## 3、时间复杂度

- 什么是时间复杂度

- 📌 **首先了解两个概念：时间频度，渐近时间复杂度**

- 1、时间复杂度：是某个算法的时间耗费，它是该算法所求解问题规模 $n$ 的函数

一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为 $T(n)$ 。

2、渐近时间复杂度：是指当问题规模趋向无穷大时，该算法时间复杂度的数量级在时间频度中， $n$ 称为问题的规模，当 $n$ 不断变化时，时间频度 $T(n)$ 也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。

一般情况下，算法中基本操作重复执行的次数是问题规模 $n$ 的某个函数，用 $T(n)$ 表示，若有某个辅助函数 $f(n)$ ，使得当 $n$ 趋近于无穷大时， $T(n)/f(n)$ 的极限值为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。



当我们评价一个算法的时间性能时，**主要标准就是算法的渐近时间复杂度**，因此，在算法分析时，往往对两者不予区分，经常是将渐近时间复杂度 $T(n)=O(f(n))$ 简称为时间复杂度，其中的 $f(n)$ 一般是算法中频度最大的语句频度。此外，算法中语句的频度不仅与问题规模有关，还与输入实例中各元素的取值相关。但是我们总是**考虑在最坏的情况下的时间复杂度**。以保证算法的运行时间不会比它更长。

## • 时间复杂度的排序



常见的时间复杂度，按数量级递增排列依次为：

常数阶 $O(1)$ 、对数阶 $O(\log_2 n)$ 、线性阶 $O(n)$ 、线性对数阶 $O(n \log_2 n)$ 、平方阶 $O(n^2)$ 、立方阶 $O(n^3)$ 、

$k$ 次方阶 $O(n^k)$ 、指数阶 $O(2^n)$ 。

按性能好坏排列：

$O(1) > O(\log_2 n) > O(n) > O(n \log_2 n) > O(n^2) > O(n^3) > O(n^k) > O(2^n)$

## • 时间复杂度的计算步骤



### 1、计算出基本操作的执行次数 $T(n)$

基本操作即算法中的每条语句（以;号作为分割），语句的执行次数也叫做语句的频度。在做算法分析时，一般默认为考虑最坏的情况。**这一步是时间复杂度最难的地方**

## 2. 计算出 $T(n)$ 的同等数量级 $f(n)$

求 $T(n)$ 的数量级，只要将 $T(n)$ 进行如下一些操作：

忽略常量、低次幂和最高次幂的系数令 $f(n)=T(n)$ 的数量级。

## 3. 用大O来表示时间复杂度

当 $n$ 趋近于无穷大时，如果 $\lim(T(n)/f(n))$ 的值为不等于0的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。

记作 $T(n)=O(f(n))$ 。

### 举例说明

```
1 void Sum(int n)
2 {
3     int num1, num2;
4     for (int i = 0; i < n; i++)
5     {
6         num1 += 1;
7         for (int j = 1; j <= n; j *= 2)
8         {
9             num2 += num1;
10        }
11    }
12 }
```



分析：

### 1. 计算 $T(n)$

语句`int num1, num2;`的频度为1；

语句`i=0;`的频度为1；

语句`i<n; i++; num1+=1; j=1;`的频度为 $n$ ；

语句`j<=n; j*=2; num2+=num1;`的频度为 $n*\log_2 n$ ；

$$T(n) = 2 + 4n + 3n*\log_2 n$$

### 2. 计算 $f(n)$

忽略掉 $T(n)$ 中的常量、低次幂和最高次幂的系数

$$f(n) = n*\log_2 n$$

### 3. 计算 $\lim(T(n)/f(n))$


$$\lim(T(n)/f(n)) = (2+4n+3n*\log_2 n) / (n*\log_2 n)$$

$$= 2 \cdot (1/n) \cdot (1/\log_2 n) + 4 \cdot (1/\log_2 n) + 3$$

当 $n$ 趋向于无穷大， $1/n$ 趋向于0， $1/\log_2 n$ 趋向于0

所以极限等于3。

$$T(n) = O(n \cdot \log_2 n)$$

 简化计算：

1、计算语句频度  $T(n)$

$$T(n) = 2 + 4n + 3n \cdot \log_2 n$$

2、决定算法复杂度的是执行次数最多的语句，所以去掉常数、低阶、以及最高阶的系数

$$T(n) = n \cdot \log_2 n$$

3、最后采用 $O$ 表示

$$T(n) = O(n \cdot \log_2 n)$$

## 4、常见时间复杂度

### • 常数阶 $O(1)$

**$O(1)$** ，表示该算法的执行时间总是为一个常量，不论输入的数据集是大是小，只要是没有循环等复杂结构，那这个代码的时间复杂度就都是 $O(1)$ ，如：

```
1 int i = 1;
2 int j = 2;
3 int k = i + j;
```

上述代码在执行的时候，它消耗的时间并不随着某个变量的增长而增长，那么无论这类代码有多长，即使有几万几十万行，都可以用 **$O(1)$** 来表示它的时间复杂度。

### • 对数阶 $O(\log n)$

```
1 int i = 1;
2 while(i < n)
3 {
4     i = i * 2;
5 }
```

上面的代码，在while循环里面，每次都将i乘以2，乘完之后，i距离n就越来越近了，直到i不小于n退出。我们试着求解一下，假设循环次数为x，也就是说2的x次方等于n，则由 $2^x=n$ 得出 $x=\log n$ 。因此这个代码的时间复杂度为 $O(\log n)$

- 线性阶 $O(n)$

$O(n)$ ，表示一个算法的性能会随着输入数据的大小变化而线性变化，如

```
1 for(int i=0; i<n; ++i)
2 {
3     sum += i;
4 }
```

这段代码，for循环里面的代码会执行n遍，因此它消耗的时间是随着n的变化而变化的，因此这类代码都可以用 $O(n)$ 来表示它的时间复杂度。

- 线性对数阶 $O(n\log n)$

线性对数阶 $O(n\log n)$ ，就是将时间复杂度为对数阶 $O(\log n)$ 的代码循环n遍，那么它的时间复杂度就是 $n \cdot O(\log n)$ ，也就是 $O(n\log n)$ ，如下，

```
1 for (int k=0; k<n; ++k)
2 {
3     int i = 1;
4     while(i < n)
5     {
6         i = i * 2;
7     }
8 }
```

- 平方阶 $O(n^2)$

$O(n)$  表示一个算法的性能将会随着输入数据的增长而呈现出二次增长。最常见的就是对输入数据进行嵌套循环。如果嵌套层级不断深入的话，算法的性能将会变为立方阶 $O(n^3)$ ， $O(n^4)$ ， $O(n^k)$ 以此类推

```
1 for(int i=0; i<n; ++i)
2 {
3     for(int j=0; j<n; ++j)
4     {
```


```
5     count++;
6     }
7 }
```


- 指数阶 $O(2^n)$


$O(2^n)$ ，表示一个算法的性能会随着输入数据的每次增加而增大两倍，典型的方法就是斐波那契数列的递归计算实现

```
1 int Fibonacci(int n)
2 {
3     if(n <= 1)
4         return n;
5     return Fibonacci(n-2) + Fibonacci(n-1);
6 }
```

## 5、空间复杂度

 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一个量度，同样反映的是一个趋势，一个算法所需的存储空间用 $f(n)$ 表示。 $S(n)=O(f(n))$ ，其中 $n$ 为问题的规模， $S(n)$ 表示空间复杂度。

 一个算法在计算机存储器上所占用的存储空间，包括**存储算法本身所占用的存储空间**，**算法的输入输出数据所占用的存储空间**和**算法在运行过程中临时占用的存储空间**这三个方面。

 一般情况下，一个程序在机器上执行时，除了需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。若输入数据所占空间只取决于问题本身，和算法无关，这样只需要分析该算法在实现时所需的辅助单元即可。若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为 $O(1)$ 。当一个算法的空间复杂度与 $n$ 成线性比例关系时，可表示为 $O(n)$ ，类比时间复杂度。

空间复杂度常用的有： $O(1)$ 、 $O(n)$

- 常数阶 $O(1)$

如果算法执行所需要的临时空间不随着某个变量n的大小而变化，即此算法空间复杂度为一个常量，可表示为  **$O(1)$**

```
1 void BubbleSort(int ar[], int n)
2 {
3     for(int i=0; i<n-1; ++i)
4     {
5         for(int j=0; j<n-1-i; ++j)
6         {
7             if(ar[j] > ar[j+1])
8             {
9                 int tmp = ar[j];
10                ar[j] = ar[j+1];
11                ar[j+1] = tmp;
12            }
13        }
14    }
15 }
```

- 线性阶 $O(n)$

```
1 void MergeSort(int *ar, int left, int right)
2 {
3     int n = right - left;
4     int *tmp = (int*)malloc(sizeof(int) * n);    //n个辅助空间
5     assert(tmp != NULL);
6
7     //归并
8     _MergeSort(ar, left, right-1, tmp);
9
10    free(tmp);
11 }
```