导学周讲义

今日目标：

1：能够说出什么是数据结构，什么是算法

2：能说出大O时间复杂度是怎么得来的

3：能够说出时间复杂度的几个分析原则并加以实际应用

4：能够说出常见的几种时间复杂度O（1），O（n），O（log n），O（n * log n）

5：能理解空间复杂度的分析方式

1、概念

虽然概念很空洞，但是概念还是需要介绍的：

1. 数据结构是指一组数据的存储结构
2. 算法就是操作数据的方法

这只是抽象的定义，我们来举一个例子，你有一批货物需要运走，你是找小轿车来运还是找卡车来运？这就是数据结构的范畴，选取什么样的结构来存储；至于你货物装车的时候是把货物堆放在一起还是分开放这就是算法放到范畴了，如何放置货物更有效率更节省空间。

数据结构和算法看起来是两个东西，但是我们为什么要放在一起来说呢？那是因为数据结构和算法是相辅相成的，数据结构是为算法服务的，而算法要作用在特定的数据结构之上，因此，我们无法孤立数据结构来讲算法，也无法孤立算法来讲数据结构。

想要学习数据结构与算法，**首先要掌握一个数据结构与算法中最重要的概念：复杂度分析**

2、复杂度分析

数据结构和算法解决的是“快”和“省”的问题，即如何让代码运行的更快，如何让代码更省存储空间。因此执行效率是一个非常重要的考量指标，那如何来衡量代码的执行效率，就是我们所说的：时间，空间复杂度分析。

为什么要进行复杂度分析？

1：和性能测试相比，复杂度分析有不依赖执行环境、成本低、效率高、易操作、指导性强的特点。

2：掌握复杂度分析，将能编写出性能更优的代码，有利于降低系统开发和维护成本。

2.1、大O复杂度表示法

算法的执行效率，粗略地讲，就是算法代码执行的时间，那如何在不直接运行代码的前提下粗略的计算执行时间呢？

在IDEA中创建一个普通的java项目：algo-pro

在项目中创建一个类：`com.itheima.complexity.ComplexityAnalysis`

（1）先来看一段简短的代码，求：1，2，3，4.....n累加和

```
/**
  * 求1~n的累加和
  * @param n
  * @return
  */
public int sum(int n) {
    int sum = 0;
    for ( int i = 1; i <= n; i++) {
        sum = sum + i;
    }
    return sum;
}
```

假设每行代码执行时间都一样为：timer，那此代码的执行时间为多少呢：`(3n+3)timer`，由此可以看出来，所有代码的执行时间T(n)与代码的执行次数成正比。

（2）按照该思路我们接着看下面一段代码

```
public int sum2(int n) {
    int sum = 0;
    for (int i=1; i <= n; ++i) {
        for (int j=1; j <= n; ++j) {
            sum = sum +  i * j;
        }
    }
    return sum;
}
```

同理，此代码的执行时间为：`(3 n^2 + 3n + 3) * timer`

因此有一个重要结论：**代码的执行时间T(n)与总的执行次数成正相关**，我们可以把这个规律总结成一个公式。

```
T(n) = O(f(n))
```

解释一下：T(n)表示代码的执行时间，**n表示数据规模的大小**，f(n)表示了代码执行的总次数，它是一个公式因此用f(n)表示，O表示了代码执行时间与f(n)成正比

因此第一个例子中的T(n)=O(3n+3)，第二个例子中的T(n)=O(3 n^2 + 3n + 3)，这就是大**O时间复杂度表示法**

大 O 时间复杂度实际上并不具体表示代码真正的执行时间，而是**表示代码执行时间随数据规模增长的变化趋势**，所以，也叫作渐进时间复杂度，**简称时间复杂度**。

当n很大时，公式中的低阶，常量，系数三部分并不左右其增长趋势，因此可以忽略，我们只需要记录一个最大的量级就可以了，因此如果用大O表示刚刚的时间复杂度可以记录为：T(n)=O(n),T(n)=O(n^2)

2.2、复杂度分析方法

通用的复杂度分析法则如下：

（1）最大循环原则

大O复杂度表示法只代表一种变化趋势，公式中的低阶，常量，系数三部分并不左右其增长趋势，因此可以忽略，我们只需要记录一个最大的量级就可以了。因此*分析一个算法或者一个代码的时间复杂度时，只需关注循环执行次数最多的那一段代码即可*。

（2）加法原则

请分析一下如下代码的时间复杂度：

```
public int sum3(int n) {
    int sum_1 = 0;
    int p = 1;
    for (; p <= 100; ++p) {
        sum_1 = sum_1 + p;
    }

    int sum_2 = 0;
    int q = 1;
    for (; q < n; ++q) {
        sum_2 = sum_2 + q;
    }
    // O(n)
    int sum_3 = 0;
    int i = 1;
    int j = 1;
    for (; i <= n; ++i) {
        j = 1;
        for (; j <= n; ++j) {
            sum_3 = sum_3 +  i * j;
        }
    }
    // O(n^2)
    return sum_1 + sum_2 + sum_3;
}
```

其中两段最大量级的复杂度分别为O(n)和O(n^2)，其结果本应该是：T(n)=O(n)+O(n ^2)，我们取其中最大的量级，因此整段代码的复杂度为：O(n ^2)，其实也可以基于第一个最大循环原则得出复杂度为O(n ^2)

也就是说：*总的时间复杂度就等于量级最大的那段代码的时间复杂度*

另外再看一段代码：

```
public int sum4(int[] nArr, int[] mArr){
    int sum = 0;
    for(int i : nArr) {
        sum += i;
    }
    // O(M)
    for(int i : mArr) {
        sum += i;
    }
    //O(N)
    return sum;
}
```

在这个例子中，虽然没有明确说明数据规模n，但其实函数参数中两个数组的长度就是我们所理解的数据规模，而且这个地方出现了两个数据规模，我们分别叫做m，n

根据我们的分析原则，这段代码的时间复杂度为**O(m+n)**，它其实也是线性复杂度O(n)的一种。

（3）乘法原则

嵌套代码的复杂度等于嵌套内外代码复杂度的乘积，举个例子

```
public int sum5(int n) {
    int ret = 0;
    int i = 1;
    for (; i < n; ++i) {
        ret = ret + func(i);
    }
    // O（n）
    return ret;
}
public int func(int n) {
    int sum = 0;
    int i = 1;
    for (; i < n; ++i) {
        sum = sum + i;
    }
    return sum; //O(N)
}
```

单独看是:O(n),由于func(i)是O(n)因此整体是:O(n) * O(n) = O(n * n) = O(n ^2)

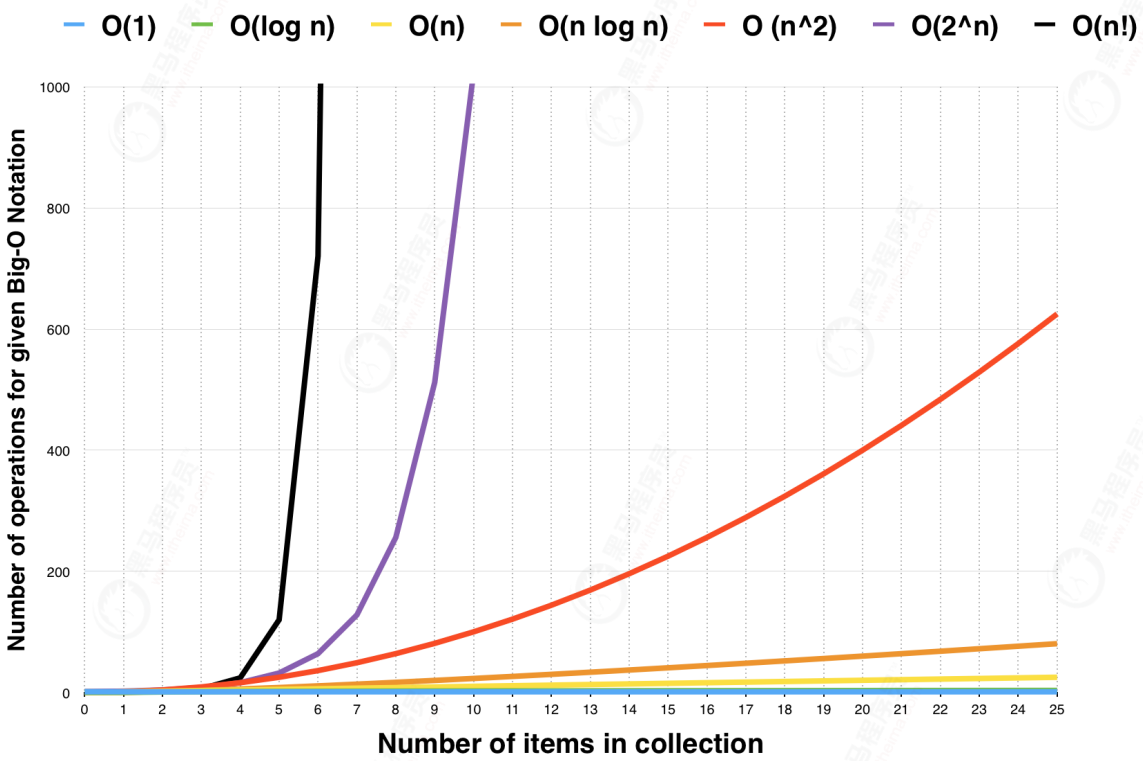
因此可以看出：*嵌套代码的复杂度等于嵌套内外代码复杂度的乘积*

2.3、常见的复杂度

虽然代码写法千差万别，但是我们平常所见的复杂度量级并不多，列举如下：

描述	表示形式
常数	O(1)
线性	O(n)
对数	O(log n)
线性对数	O(n * log n)
平方	O(n ^2)
立方	O(n ^3)
……	…….
k次方	O(n ^k)
指数	O(2 ^n)
阶乘	O(n !)

对于表格中所罗列的一些复杂度从上到下当n越大时算法执行时间会急剧增加，时间会无限增长，常见复杂度的增长曲线如下图



所以我们一般要关注前面的几种。且在实际工程应用中避免编写出复杂度超高的代码。

（1）O(1)

O(1)并不是指代码只有一行，它是一种常量级复杂度的表示方法，比如说有一段代码如下：

```
public void test01(int n){
    int i=0;
    int j = 1;
    return i+j;
}
```

代码只有三行，它的复杂度也是O(1)，而不是O(3)

再看如下代码：

```
public void test02(int n){
    int i=0;
    int sum=0;
    for(;i<100;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

整个代码中因为循环次数是固定的就是100次，这样的代码复杂度我们认为也是O(1),

因此总结下来就是：只要代码的执行时间不随着n的增大而增大，这样的代码复杂度都是O(1)，或者说：只要在算法中不存在递归语句，随n变化的循环语句等，即使有千万行代码，复杂度也是O(1)

（2）O(n)

这种复杂度量级随处可见，比如我们分析如下代码：

```
public void test03(int n){
    int i=0;
    int sum=0;
    for(;i<n;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

另外对于O(n^2)这种复杂度我们也就很好理解了

（3）O(log n)

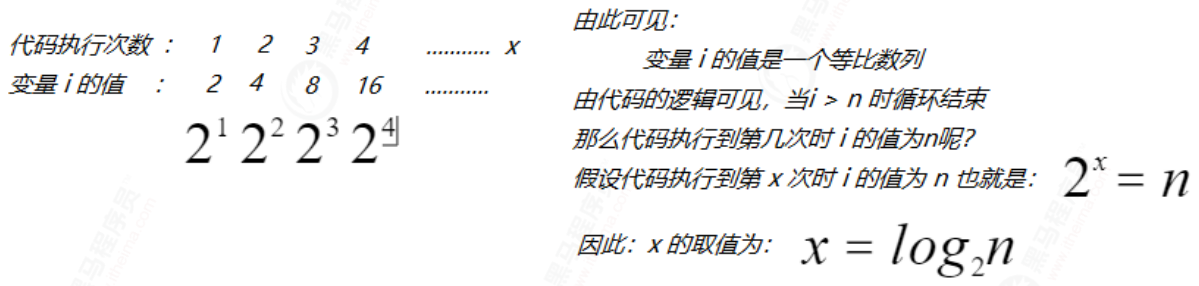
对数复杂度非常的常见，但相对比较难以分析，代码如下：

```
public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}
```

复杂度分析就是要弄清楚代码的执行次数和数据规模n之间的关系

根据经验：第四行代码：i = i*2 循环次数最多，因此我们要分析出第四行代码执行了多少次我们就得出了此代码的时间复杂度

那第四行代码执行了多少次呢？代码执行第一次i=2，执行第二次i=4，执行第三次i=8.....，i的取值其实是一个等比数列的形式，如图



由图中分析可知，代码的时间复杂度表示为 $O(log_2 n)$

那如果将代码进行修改为如下：

```
public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 3;
    }
}
```

通过刚刚的思路我们马上就能分析出来：这段代码的时间复杂度为：$O(log_3 n)$ 但是实际上不管是以2为底还是以3为底，或者以10为底，我们把所有的对数阶的时间复杂度都记为: $O(logn)$ 为什么呢？

我们知道对数有一个换底公式：$log_a b * log_b N = log_a N$，因此 $log_3 2 * log_2 n = log_3 n$,而以3为底，2的对数是一个常量系数，基于我们前面的讨论，使用大O标记时间复杂度时不考虑低阶，系数，常量，所以在对数阶时间复杂度中我们忽略对数的底统一表示为：$O(logn)$

（4）O(n * log n)

分析完O(log n)，那O(n * log n)就很容易理解了，比如下列代码：

```
public void test05(int n){
    int i=0;
    for(;i<=n;i++){
        test04(n);
    }
}
public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}
```

这是一种非常常见的算法时间复杂度，我们后续要学习的归并排序，快速排序的时间复杂度都是O(nlogn)

2.4、最好/最坏/平均复杂度

（1）最好/最坏复杂度

有一个需求：在数组array中查找变量x的位置，实现如下：

```
//其中n表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
        pos = i;
      }
  }
  return pos;
}
```

通过之前学习的复杂度分析方式，我们可以分析出来这段代码的复杂度为O(n)，但是这段代码实现的并不是特别好，我们稍作优化，因为在数组中查找某一元素并不需要把整个数组都遍历一遍，因为可能在遍历的途中就已经找到了，找到了就直接返回，提前结束循环，优化后的结果如下：

```
//其中n表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
        pos = i;
        break;
      }
  }
  return pos;
}
```

优化完成之后我们再看这段代码的复杂度还是O(n)吗？

要查找的变量x在可能在数组中的任意位置，如果数组中的第一个元素就是我们要找的变量x，那就不需要继续变量余下的n-1个元素了，那复杂度为O(1),如果数组中不存在变量x那需要完整的遍历一遍数组，那复杂度就是O(n)。所以：在不同的情况下，同一段代码的复杂度并不一样

因此我们需要引入三个概念：**最好情况复杂度，最坏情况复杂度，和平均情况复杂度**

最好情况复杂度：在最理想的情况下代码的时间复杂度

最坏情况复杂度：在最糟糕的情况下代码的时间复杂度

（2）平均情况复杂度

我们知道最好或者最坏情况复杂度分别对应了两种极端的情况，发生的概率并不大，为了更好的表示平均情况下的复杂度，我们引入平均情况复杂度。

那刚刚的代码如何分析评价情况复杂度呢？

```
//其中n表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
        pos = i;
        break;
      }
  }
  return pos;
}
```

查找变量x在数组中的位置有n+1中情况，在数组中0~n-1的位置上以及不在数组中，我们把每一种情况下代码需要遍历的次数加起来，然后除以n+1就得到了代码需要遍历执行的平均值。如图：

$$\frac{1+2+3+4+.....+n+n}{n+1}=\frac{\frac{n(n+1)}{2}+n}{n+1}=\frac{\frac{n^2+3n}{2}}{n+1}=\frac{n^2+3n}{2(n+1)}$$

我们知道，在大O复杂度标记法中，可以省略系数，低阶，常量，因此把该结果简化之后得到的复杂度仍然为O(n)

此时，结论虽然正确，但是计算过程稍微有点儿问题，问题出在哪里？就是我们刚刚讲到的这n+1种情况出现的概率并不是一样的，

通过之前的讲解我们知道查找变量x在数组中的位置要么在数组中，要么不在数组中，意味着在数组中0~n-1位置上出现的概率为1/2，不再数组中出现的概率为1/2,

此外在数组中0~n-1位置上每一种情况下的概率为1/n，因此根据概率法则，要查找的数据出现在0 ~ n-1中任意位置的概率为1/2n

所以前面的结论推导过程中我们需要将概率考虑进去，那计算平均时间复杂度的计算过程就变成了这样：

$$\frac{1}{2n}(1+2+3+...+n)+\frac{1}{2}n=\frac{\frac{n(n+1)}{2}}{2n}+\frac{n}{2}=\frac{n^2+n}{4n}+\frac{2n^2}{4n}=\frac{3n^2+n}{4n}=\frac{3n+1}{4}$$

这个值在概率论中叫**加权平均值，也叫做期望值，所以平均时间复杂度也叫期望时间复杂度**，同样的道理如果用大O表示法来表示，加权时间复杂度仍然为：O(n)

实际上：大多数情况下，我们不需要去区分最好/最坏/平均 时间复杂度，只有在同一段代码块在不同情况下的复杂度有量级的差距时才需要用这三种复杂度来加以区分

2.5、空间复杂度

时间复杂度表示了算法的执行时间与数据规模之间的增长关系，类比一下，**空间复杂度全称是渐进空间复杂度，表示算法占用的额外存储空间与数据规模之间的增长关系**

用一段小的代码来说明一下：

```
public void test(int n){
    int i=0;
    int sum=0;
    for(;i<n;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

代码执行并不需要占用额外的存储空间，只需要常量级的内存空间大小，因此空间复杂度是O(1)

```
void print(int n) {
  int i = 0;
  int[] a = new int[n];
  for (i; i <n; ++i) {
    a[i] = i * i;
  }
  for (i = n-1; i >= 0; --i) {
    System.out.println(a[i]);
  }
}
```

代码第二行，申请一个空间存储变量i，但是是常量阶的，跟数据规模n没有关系，第三行申请了一个大小为n的int数组，此外后面的代码几乎没有占用更多的空间，因此整段代码的空间复杂度就是O(n)

我们**常见的空间复杂度就是O(1),O(n),O(n ^2)**，其他像对数阶的复杂度几乎用不到，因此空间复杂度比时间复杂度分析要简单的多。因此掌握这些足够。