算法的时间复杂度

我们想要知道一个算法的「时间复杂度」，很多人首先想到的的方法就是把这个算法程序运行一遍，那么它所消耗的时间就自然而然知道了。

这种方式可以吗？当然可以，不过它也有很多弊端。
这种方式非常容易受运行环境的影响，在**性能高的机器**上跑出来的结果与在性能低的机器上跑的结果相差会很大。而且对测试时使用的**数据规模**也有很大关系。再者，并我们在写算法的时候，还没有办法完整的去运行呢。

因此，另一种更为通用的方法就出来了：「大O符号表示法」，即 T(n) = O(f(n))

我们先来看个例子：

```
for(i=1; i<=n; ++i)
{
    j = i;
    j++;
}
```

通过「大O符号表示法」，这段代码的时间复杂度为：O(n)，为什么呢?

在 大O符号表示法中，时间复杂度的公式是： **T(n) = O(f(n))**，其中f(n) 表示每行代码执行次数之和，而 O 表示正比例关系，这个公式的全称是：**算法的渐进时间复杂度**。

我们继续看上面的例子，假设每行代码的执行时间都是一样的，我们用 1颗粒时间 来表示，那么这个例子的第一行耗时是1个颗粒时间，第三行的执行时间是 n个颗粒时间，第四行的执行时间也是 n个颗粒时间（第二行和第五行是符号，暂时忽略），那么总时间就是 1颗粒时间 + n颗粒时间 + n颗粒时间 ，即 (1+2n)个颗粒时间，即： T(n) = (1+2n)*颗粒时间，**频度**从这个结果可以看出，这个算法的耗时是随着n的变化而变化，因此，我们可以简化的将这个算法的时间复杂度表示为：T(n) = O(n)

为什么可以这么去简化呢，**因为大O符号表示法并不是用于来真实代表算法的执行时间的，它是用来表示代码执行时间的增长变化趋势的。**

所以上面的例子中，如果n无限大的时候，**T(n) = time(1+2n) 中的常量1就没有意义了，倍数2也意义不大**。因此直接简化为 T(n) = O(n) 就可以了。

常见的时间复杂度量级有：

- 常数阶O(1)
- 对数阶O(logN)
- 线性阶O(n)

- 线性对数阶O(nlogN)
- 平方阶O(n^2)
- 立方阶O(n^3)
- K次方阶O(n^k)
- 指数阶(2^n)

上面从上至下依次的时间复杂度越来越大，执行的效率越来越低。

下面选取一些较为常用的来讲解一下（没有严格按照顺序）：

常数阶O(1)

无论代码执行了多少行，**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：

```
int i = 1;
int j = 2;
++i;
j++;
int m = i + j;
```

上述代码在执行的时候，它消耗的时候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。

线性阶O(n)

这个在最开始的代码示例中就讲解过了，如：

```
for(i=1; i<=n; ++i)
{
    j = i;
    j++;
}
```

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。

对数阶O(logN)

还是先来看代码：

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

从上面代码可以看到，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。我们试着求解一下，假设循环x次之后，i 就大于 2 了，此时这个循环就退出了，**也就是说 2 的 x 次方等于 n，那么 x = log2^n**
也就是说当循环 log2^n 次以后，这个代码就结束了。因此这

个代码的时间复杂度为：**O(logn)**

线性对数阶O(nlogN)

线性对数阶O(nlogN) 其实非常容易理解，将时间复杂度为O(logn)的代码循环N遍的话，那么它的时间复杂度就是 n * O(logN)，也就是了O(nlogN)。

就拿上面的代码加一点修改来举例：

```
for(m=1; m<n; m++)
{
     i = 1;
     while(i<n)
     {
          i = i * 2;
     }
}
```

平方阶O(n²)

平方阶O(n²) 就更容易理解了，如果把 O(n) 的代码再嵌套循环一遍，它的时间复杂度就是 O(n²) 了。
举例：

```
for(x=1; i<=n; x++)
{
   for(i=1; i<=n; i++)
    {
       j = i;
       j++;
    }
}
```

这段代码其实就是嵌套了2层n循环，它的时间复杂度就是O(n*n)，即 $O(n^2)$
如果将其中一层循环的n改成m，即：

```
for(x=1; i<=m; x++)
{
   for(i=1; i<=n; i++)
    {
       j = i;
       j++;
    }
}
```

那它的时间复杂度就变成了 O(m*n)

立方阶$O(n^3)$、K次方阶O(n^k)

参考上面的$O(n^2)$ 去理解就好了，$O(n^3)$相当于三层n循环，其它的类似。

除此之外，其实还有 平均时间复杂度、均摊时间复杂度、最坏时间复杂度、最好时间复杂度 的分析方法，有点复杂，这里就不展开了。