# 常见的时间复杂度

1) 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)

```
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。

2) 对数阶 O(log2n)

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

**说明**：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。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 = $\log_2 n$ 也就是说当循环 $\log_2 n$ 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($\log_2 n$) 。 O($\log_2 n$) 的这个2时间上是根据代码变化的，i = i * 3， 则是 O($\log_3 n$) .

如果 $N = a^x (a > 0, a \neq 1)$，即a的x次方等于N（a>0，且a≠1），那么数x叫做以a为底N的对数（logarithm），记作 $x = \log_a N$。其中，a叫做对数的底数，N叫做真数，x叫做"以a为底N的对数"。

3) 线性阶 O(n)

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

**说明**：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

4) 线性对数阶 O(nlog2n)

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

**说明**：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)

5) 平方阶 O(n^2)

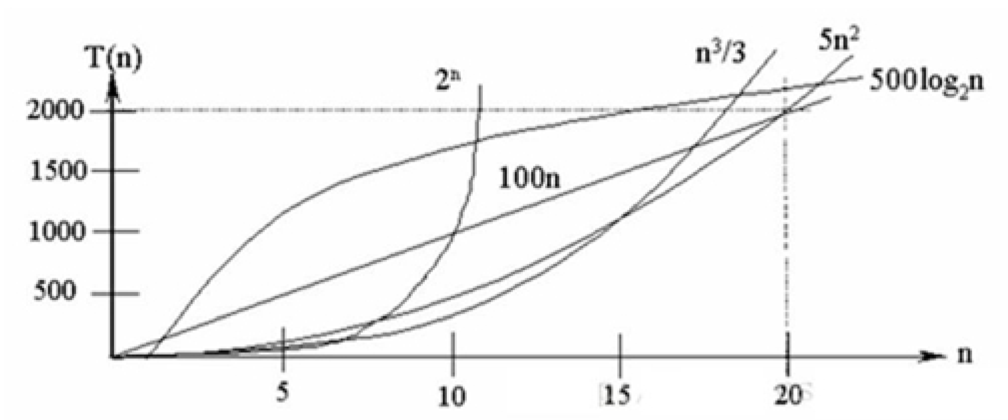
```
for(x=1; i<=n; x++)
{
   for(i=1; i<=n; i++)
    {
       j = i;
       j++;
    }
}
```

**说明**：平方阶O($n^2$) 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($n^2$)，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，那它的时间复杂度就变成了 O(m*n)

6) 立方阶 O(n^3)

7) k次方阶 O(n^k)

8) 指数阶 O(2^n)




说明：

1) 常见的算法时间复杂度由小到大依次为：Ο(1)＜Ο(log2n)＜Ο(n)＜Ο(nlog2n)＜Ο(n^2)＜Ο(n^3)＜ Ο(n^k) ＜Ο(2^n) ，随着问题规模 n的不断增大，上述时间复杂度不断增大，算法的执行效率越低

2) 从图中可见，我们应该尽可能避免使用指数阶的算法