# 大O复杂度表示法

前提：假设每行代码执行的时间都一样，表示为unit\_time

变量解释：

T(n) : 表示代码执行的时间

n: 表示数据规模的大小

f(n): 表示每行代码执行的次数的总和

O: 表示代码的执行时间T(n)与f(n)表达式成正比

## T(n) = O(n)

int cal(int n) {

int sum = 0; ---------------------------------------- 1

int i = 1; --------------------------------------------- 2

for (; i <= n; ++i) { --------------------------------- 3

sum = sum + i; ------------------------------ 4

}

return sum;

}

这里假设每行语句的执行时间为unit\_time，则1、2行分别需要1个单位的unit\_time，3、4行分别需要n个单位的unit\_time，那么这段代码需要的总时间就是(2n+2) \* unit\_time，也就是T(n) = O(2n + 2)，由于大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，而公式中的低阶、常量、系数并不影响增长趋势，所以我们只需要记录一个最大量级就可以了，记为：T(n) = O(n)

## T(n) = O(n²)

int cal(int n) {

int sum = 0; -----------------------------------------1

int i = 1; --------------------------------------------- 2

int j = 2; --------------------------------------------- 3

for (; i <= n; ++i) { --------------------------------- 4

j = 1; ------------------------------------------ 5

for (; j <= n; ++j) { -------------------------- 6

sum = sum + i \* j; ------------------- 7

}

}

return sum;

}

同理分析，1、2、3执行时间分别为1个单位的unit\_time，4、5需要n个单位的unit\_time，6、7需要n²个单位的unit\_time，也就是T(n) = O(2n²+2n+3)，同样去除公式中的低阶、常量、系数三大部分，取最大量级就得到了T(n) = O(n²)

# 时间复杂度分析

## 只关注循环执行次数最多的一段代码

int cal(int n) {

int sum = 0; ---------------------------------------- 1

int i = 1; --------------------------------------------- 2

for (; i <= n; ++i) { --------------------------------- 3

sum = sum + i; ------------------------------ 4

}

return sum;

}

使用之前的一段实例，其中1、2都是常量级的执行时间，与n的大小无关，所以对于复杂度并没有影响。循环执行次数最多的是3、4，所以这段要重点分析。前面已经说过，这两行代码被执行了n次，所以总的时间复杂度就是O(n)

## 加法法则：总复杂度等于量级最大的那段代码的复杂度