# 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

int cal(int n) {

// part 1

int sum\_1 = 0;

int p = 1;

for (; p < 100; ++p) {

sum\_1 = sum\_1 + p;

}

// part 2

int sum\_2 = 0;

int q = 1;

for (; q < n; ++q) {

sum\_2 = sum\_2 + q;

}

// part 3

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;

}

}

return sum\_1 + sum\_2 + sum\_3;

}

这块代码分为3个部分，第一部分循环了100次，但是因为与n无关，所以是常量级的执行时间；第二部份及第三部分代码的时间复杂度很容易算出来是O(n)和O(n²)。综合这三段代码的时间复杂度，我们取其中最大的量级。所以，整段代码的时间复杂度就为 O(n²)。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。

## 乘法法则：嵌套代码的复杂度等于嵌套内外代码复杂度的乘积

int cal(int n) {

int ret = 0; ---------------------------------------- 1

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

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

ret = ret + f(i); ----------------------------- 4

} ----------------------------------------------------- 5

}

int f(int n) {

int sum = 0; -------------------------------------- 6

int i = 1; ------------------------------------------- 7

for (; i < n; ++i) { ---------------------------------8

sum = sum + i; -----------------------------9

} ---------------------------------------------------- 10

return sum; ------------------------------------ 11

}

我们单独看cal()函数。假设 f() 只是一个普通的操作，那第 1～5 行的时间复杂度就是T1(n) = O(n)。但f() 函数本身不是一个简单的操作，它的时间复杂度是 T2(n) = O(n)，所以，整个 cal() 函数的时间复杂度就是，T(n) = T1(n) \* T2(n) = O(n\*n) = O(n²)。

# 几种常见时间复杂度实例分析

## O(1)

int i = 8; ---------------- 1

int j = 6; ---------------- 2

int sum = i + j; -------- 3

O(1) 只是常量级时间复杂度的一种表示方法，并不是指只执行了一行代码。比如这段代码，即便有3行，它的时间复杂度也是O(1)。一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是O(1)。

## O(㏒n)、O(n㏒n)

i=1; --------------------- 1

while (i <= n) { --------- 2

i = i \* 2; ----------- 3

} --------------------------- 4

从代码中可以看出，变量i的值从1开始取，每循环一次就乘以2。当大于n时，循环结束。这样变量i的取值就是一个等比数列。如果我把它一个一个列出来，就应该是这个样子的：2º，2¹，2²……2^x，循环次数x使用㏒2(x) = n可以计算出，所以，这段代码的时间复杂度就是O(log2(n))。

i=1; --------------------- 1

while (i <= n) { --------- 2

i = i \* 3; ----------- 3

} --------------------------- 4

同理可以看出这段代码的时间复杂度就是O(㏒(3)n)。但是实际上，不管是以2为底、以3为底，还是以10为底，我们可以把所有对数阶的时间复杂度都记为O(㏒n)，原因是根据换底公式IMG_256，可以得出㏒(3)n就等于㏒(3)2 \* ㏒(2)n，所以O(㏒(3)n) = O(C \* log(2)n)，其中C=log(3)2是一个常量。基于我们前面的一个理论：在采用大O标记复杂度的时候，可以忽略系数，即 O(Cf(n)) = O(f(n))。所以，O(㏒(2)n) 就等于 O(㏒(3)n)。因此，在对数阶时间复杂度的表示方法里，我们忽略对数的“底”，统一表示为 O(㏒n)。对应的O(n㏒n) 就是当一段代码的时间复杂度是 O(㏒n)时，我们循环执行 n 遍，时间复杂度就是O(n㏒n)了。

## O(m+n)、O(m\*n)

int cal(int m, int n) {

int sum\_1 = 0;

int i = 1;

for (; i < m; ++i) {

sum\_1 = sum\_1 + i;

}

int sum\_2 = 0;

int j = 1;

for (; j < n; ++j) {

sum\_2 = sum\_2 + j;

}

return sum\_1 + sum\_2;

}

从代码中可以看出，m 和 n 是表示两个数据规模。我们无法事先评估 m 和 n 谁的量级大，所以我们在表示复杂度的时候，就不能简单地利用加法法则，省略掉其中一个。所以，上面代码的时间复杂度就是 O(m+n)。

int cal(int m, int n) {

int sum = 0;

int i = 1;

for (; i < m; ++i) {

int j = 1;

for (; j < n; ++j) {

sum = sum + 1;

}

}

return sum;

}

但是乘法法则继续有效：T1(m)\*T2(n) = O(f(m) \* f(n)) = O(m\*n)

# 针对在不同情况下时间复杂度不同的分析方法

int find(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;

}

## 最好情况时间复杂度

在最好情况下，数组中的第一个元素就是要查找的变量x，那就不需要遍历剩下的n-1个数据了，那么时间复杂度就是O(1)。所以最好情况时间复杂度就是在最理想的情况下，执行这段代码的时间复杂度。

## 最坏情况时间复杂度

在最坏情况下，数组中不存在变量x，那我们就需要将整个数组都遍历一遍，时间复杂度就是O(n)。所以最坏情况时间复杂度就是在最糟糕的情况下，执行这段代码的时间复杂度。

## 平均情况时间复杂度

我们都知道，最好和最坏情况时间复杂度都是在极端情况下的代码复杂度，发生的概率并不大，于是我们引入了另一个概念：平均情况时间复杂度，全程叫做加权平均时间复杂度或者期望时间复杂度。

首先假设x出现在数组内的概率为1/2，要查找的数据出现在0 ~ n-1这n个位置的概率也是一样的，记为1/n，所以根据概率乘法法则，要查找的数据出现在0 ~ n-1中任意位置的概率就是1/(2n)。这样循环次数的概率就是1 \* (1 / 2n) + 2 \* (1 / 2n) + 3 \* (1 / 2n) + ... + n \* (1 / 2n) + n \* (1 / 2) = (3n + 1) / 4。用大O表示法，去掉系数和常量，这段代码的加权平均时间复杂度为O(n)。

## 均摊时间复杂度(摊还分析)

int[] array = new int[n];

int count = 0;

void insert(int val) {

if (count == array.length) {

int sum = 0;

for (int i =0; i < array.length; ++i) {

sum = sum + array[i];

}

array[0] = sum;

count = 1;

}

array[count] = val;

++count;

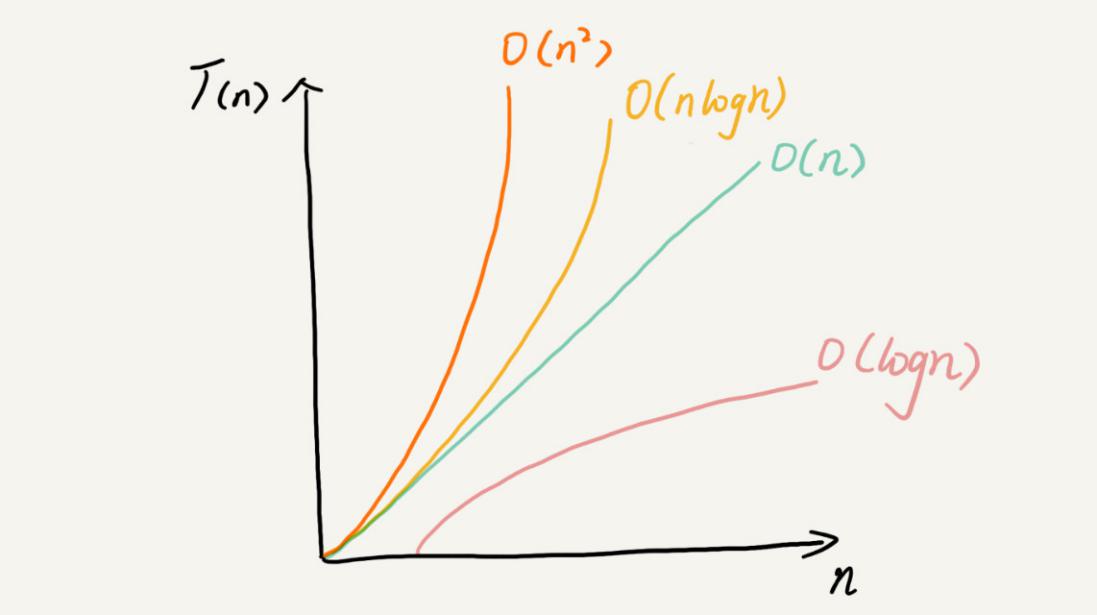
}

首先使用之前的平均情况时间复杂度来分析，假设数组的长度是n，根据数据插入的位置不同，我们可以分为n种情况。每种情况的时间复杂度是O(1)，除此之外，还有一种额外的情况，就是在数组没有空闲空间时插入一个数据，这个时候的时间复杂度是O(n)。而且这n+1种情况发生的概率一样，都是1/(n + 1)，所以根据加权平均的计算方法，平均时间复杂度为：1 \* (1 / (n + 1)) + 1 \* (1 / (n + 1)) + ... + n \* (1 / (n + 1)) = n / (n + 1) + (n / (n + 1)) = 2(n / n + 1) = O(1)。

重新分析场景，我们可以看到，每一次O(n)的插入操作，都会跟着n-1次O(1)的插入操作，所以把耗时多的那次操作均摊到接下来的n-1次耗时少的操作上，实质为(n + n - 1) / n = 2，这样均摊下来，这一组连续的操作的均摊时间复杂度就是O(1)。

总结一下，对一个数据结构进行一组连续操作中，大部分情况下时间复杂度都很低，只有个别情况下时间复杂度比较高，而且这些操作之间存在前后连贯的时序关系，这个时候，我们就可以将这一组操作放在一块分析，看是否能将较高时间复杂度那次操作的耗时，平摊到其他那些时间复杂度比较低的操作上。而且，在能够应用均摊时间复杂度分析的场合，一般均摊时间复杂度就等于最好情况时间复杂度。

# 常见复杂度执行效率对比



越高阶复杂度的算法，执行效率越低。常见的复杂度并不多，从低阶到高阶有：O(1)、O(㏒n)、O(n)、O(n㏒n)、O(n²)

# 空间复杂度分析

空间复杂度全称是渐进空间复杂度（asymptotic space complexity），表示算法的存储空间与数据规模之间的增长关系。

void print(int n) {

int i = 0; ----------------------------------- 1

int[] a = new int[n]; -------------------- 2

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

a[i] = i \* i; ---------------------------4

} --------------------------------------------- 5

for (i = n-1; i >= 0; --i) { ---------------- 6

print out a[i] -----------------------7

} -------------------------------------------- 8

}

跟时间复杂度分析一样，我们可以看到，第1行代码中，我们申请了一个空间存储变量 i，但是它是常量阶的，跟数据规模 n 没有关系，所以我们可以忽略。第2行申请了一个大小为 n 的 int 类型数组，除此之外，剩下的代码都没有占用更多的空间，所以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(n²)