

1, 定义

时间复杂度一般采用大O标记法, 即 $T(n) = O(f(n))$, 其中 $T(n)$ 表示代码运行时间; n 表示数据规模大小; $f(n)$ 表示每行代码执行次数总和, O 表示 $T(n)$ 与 $f(n)$ 的正比关系。大 O 时间复杂度实际上并不具体表示代码的真正运行时间, 而是表示代码执行时间随数据规模增长的变化趋势。

在大 O 表示分析中, 低阶项和常数项都可以省略, 只保留最高阶项即可; 如 $f(n) = 2n + 2$ 在大 O 标记法中记为 $T(n) = O(n)$, 而对于形如 $f(n) = 2n^2 + 2n + 3$ 表示为 $T(n) = O(n^2)$ 。

2, 时间复杂度分析

- 关注循环次数多的代码

```
1 public int accumulate(int n) {
2     int sum = 0;
3     int i = 1;
4     for (; i <= n; i++) {
5         sum += i;
6     }
7     return sum;
8 }
```

其中for循环内的代码执行 n 次, 而其余代码执行1次, 与 n 的大小无关, 忽略常数项, 该段代码的时间复杂度为 $O(n)$ 。

- 加法法则

总复杂度为量级最大的那段代码的复杂度, 抽象为公式为:

若 $T_1(N) = O(f(n))$, $T_2(N) = O(g(n))$, 那么 $T(N) = T_1(N) + T_2(N) = O(f(n)) + O(g(n))$
 $= \max(O(f(n)), O(g(n)))$

```
1 public int accumulate(int n) {
2     int sum1 = 0;
3     for (int i = 1; i <= 100; i++) {
4         sum1 += i;
5     }
6
7     int sum2 = 0;
8     for (int i = 1; i <= n; i++) {
9         sum2 += i;
10    }
11
12    int sum3 = 0;
13    for (int i = 1; i <= n; i++) {
14        for (int j = 1; j <= n; j++) {
15            sum3 += i * j;
16        }
17    }
18 }
```

```

17     }
18     return sum1 + sum2 + sum3;
19 }

```

其中sum1段的代码循环执行了100次，与n无关。sum2段代码的复杂度为 $O(N)$ ，sum3段的代码复杂度为 $O(N^2)$ ；根据加法法则，我们只去其中最大量级的复杂度，所以该段代码的时间复杂度为 $O(N^2)$ 。

- **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，抽象为公式为：

若 $T_1(N) = O(f(n))$, $T_2(N) = O(g(n))$, 那么 $T(N) = T_1(N) * T_2(N) = O(f(n) * g(n))$

```

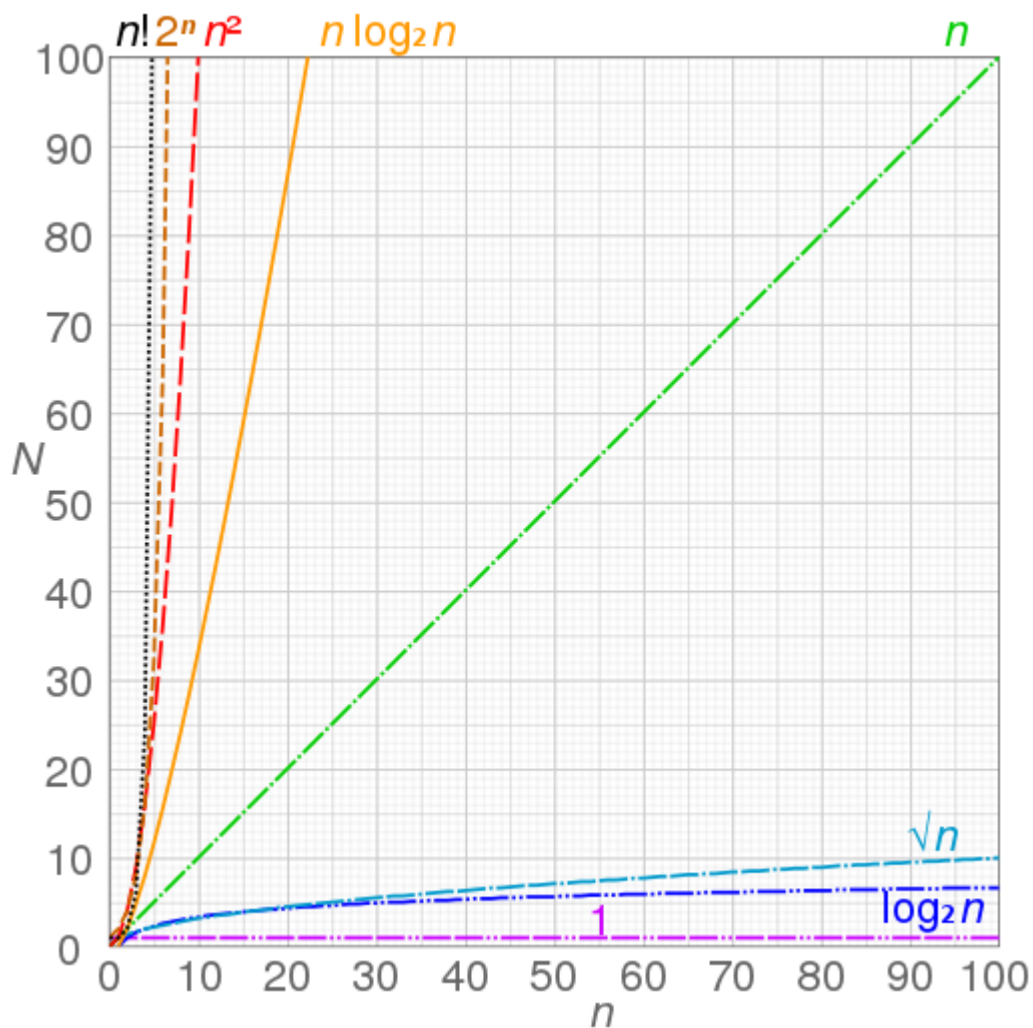
1  public int accumulate(int n) {
2      int result = 0;
3      for (int i = 1; i <= n; i++) {
4          result += f(i);
5      }
6      return result;
7  }
8
9  private int f(int n) {
10     int sum = 0;
11     for (int i = 1; i <= n; i++) {
12         sum += i;
13     }
14     return sum;
15 }

```

其中accumulate方法与f方法的时间复杂度都为 $O(N)$ ，但是f嵌套在accumulate中，所以整段代码的复杂度就为： $T(N) = T_1(N) * T_2(N) = O(N * N) = O(N^2)$

3，常见时间复杂度量级

度量级	大 O 表示
常量阶	$O(1)$
对数阶	$O(\log N)$
线性阶	$O(N)$
线性对数阶	$O(N \log N)$
平方阶	$O(N^2)$
立方阶	$O(N^3)$
指数阶	$O(2^n)$
阶乘阶	$O(N!)$



- 常见的时间复杂度有常量阶、对数阶、线性阶、线性对数阶以及平方阶，常量阶、线性阶与平方阶在第二节中已经分析，不再赘述；而一些高效的排序算法的时间复杂度就是线性对数阶，如快速排序，归并排序以及堆排序等。

• 对数阶

我们所熟知的二分查找的复杂度就是 $O(\log N)$ ，以下通过一段代码来分析对数阶复杂度：

```

1 public int test(int n) {
2     int res = 1;
3     while (res <= n) {
4         res *= 2;
5     }
6     return res;
7 }

```

该段代码是求 $2^x = n$ 的解，更确切的说，是找出 2^x 在小于或等于 n 的范围内最接近 n 的 x 的值；其中 $x = \log_2 n$ ，即while循环体内代码要执行 $\log_2 n$ 次，即其时间复杂度为 $O(\log_2 n)$ 。

若把循环体内代码 `res *= 2` 改为 `res *= 3`，不难分析出其时间复杂度就变为 $O(\log_3 n)$ ；但是为什么所有对数阶的时间复杂度都统一表示为 $O(\log N)$ ？

首先我们先复习对数换底公式: $\log_A B = \frac{\log_C B}{\log_C A}$ 则 $\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{\log_2 2}{\log_2 3} \cdot \log_2 n = \log_3 2 \cdot \log_2 n$ 所以 $O(\log_3 n) = O(\log_3 2 \cdot \log_2 n)$, 因为 $\log_3 2$ 为常数项, 所以该项可以忽略, 因此 $O(\log_3 n) = O(\log_2 n)$; 所以无论对数以哪个数为底, 最后都可以转化为一个常数项与以2为底的对数相乘, 因此在对数阶时间复杂度的表示方法里, 就忽略对数的底, 统一表示为 $O(\log N)$

- $O(m + n)$ 与 $O(m * n)$

此种表示形式的时间复杂度是由两个数据规模来决定的

```
1 public int accumulate(int m, int n) {
2     int sum1 = 0;
3     for (int i = 1; i <= m; i++) {
4         sum1 += i;
5     }
6
7     int sum2 = 0;
8     for (int i = 1; i <= n; i++) {
9         sum2 += i;
10    }
11
12    return sum1 + sum2;
13 }
```

由于我们不能事先知晓m与n哪个量级大, 所以就不能简单的利用加法规则取其最大量级, 那么像这种代码的时间复杂度就为 $O(m + n)$ 。

```
1 public int accumulate(int m, int n) {
2     int sum = 0;
3     for (int i = 1; i <= m; i++) {
4         for (int j = 1; j <= n; j++) {
5             sum += i * j;
6         }
7     }
8     return sum;
9 }
```

而类似上述代码依然可以使用乘法法则, 其时间复杂度为 $O(m * n)$

4, 最好、最坏、平均和均摊时间复杂度

以下将通过一段代码来讲述这几个时间复杂度:

```
1 public class Test {
2     private int[] array = new int[5];
3     private int N = 0;
4
5     public void push(int item) {
6         if (N == array.length) {
7             resize(2 * array.length);
8         }
9         array[N++] = item;
```

```

10     }
11
12     private void resize(int size) {
13         int[] temp = new int[size];
14         for (int i = 0; i < N; i++) {
15             temp[i] = array[i];
16         }
17         array = temp;
18     }
19 }

```

上述代码是用数组模拟一个栈的部分代码，其中 `push` 表示压栈操作，`resize` 表示对数组进行扩容的操作；当压入栈中的元素数量达到数组的容量时，就定义一个容量为之前两倍的新数组 `temp`，将旧数组 `array` 中的元素复制到新数组中，然后将 `array` 指向 `temp`。

- **最好时间复杂度：**最理想的情况下，当前栈中元素数量比数组的容量小，此时就直接执行代码块 `array[N++] = item;`，即此时的时间复杂度为 $O(1)$ 。
- **最坏时间复杂度：**最糟糕的情况下，当前栈中元素数量与数组的容量相等，此时就要执行 `resize` 方法进行扩容了，进入循环体，执行 N 次复制操作，此时的时间复杂度为 $O(N)$ 。
- **平均时间复杂度：**
 - 当栈中元素小于数组容量时，此时进行压栈就有 N 中情况，且每种情况的时间复杂度为 $O(1)$ ；当栈中元素与数组容量相等时，此时进行压栈就只有一种情况了，要进行扩容操作，这种情况的时间复杂度为 $O(N)$ ；则总共有 $N+1$ 中情况，对其取平均值：

$$\frac{1 + 1 + 1 + \dots + 1 + N}{N + 1} = \frac{2N}{N + 1}$$
在大 O 标记法中，可以省略系数与低阶项，所以其平均时间复杂度为 $O(1)$
 - 下面使用概率来分析，由于有 $N+1$ 中情况，每种情况的发生概率为 $\frac{1}{N+1}$ ，则其平均时间复杂度为：

$$1 \times \frac{1}{N+1} + 1 \times \frac{1}{N+1} + \dots + 1 \times \frac{1}{N+1} + N \times \frac{1}{N+1} = O(1)$$
- **均摊时间复杂度：**是一种特殊的平均时间复杂度，根据上述代码，每出现一次扩容操作时，即此时压栈的时间复杂度为 $O(N)$ ，那么后面的 N 次压栈操作的时间复杂度均为 $O(1)$ ，前后是连贯的，因此将 $O(N)$ 平摊到前 N 次上，得出均摊时间复杂度为 $O(1)$ 。