03. 复杂度分析（下）

前言

> - 最好情况时间复杂度（best case time complexity）
> - 最坏情况时间复杂度（worst case time complexity）
> - 平均情况时间复杂度(average case time complexity)
> - 均摊时间复杂度(amortized time complexity)

最好、最坏情况时间复杂度

试分析以下代码的时间复杂度：

```
// n表示数组array的长度
int find(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
    if (array[i] == x) pos = i;
  }
  return pos;
}
```

这段代码要实现的功能是，在一个无序的数组（array）中，查找变量 x 出现的位置；如果没有找到，就返回 -1；找到，就返回其位置。

按照之前讲的分析方法，这段代码的复杂度是 O(n)，其中，n 代表数组的长度。

在数组中查找一个数据，并不需要每次都把整个数组都遍历一遍，因为有可能中途找到就可以提前结束循环了。

但是，这段代码写的不够高效。可以这样优化一下这段查找代码。

```
// n表示数组array的长度
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;
}
```

优化完之后，就不能使用之前讲的分析方法。因为，要查找的变量 x 可能出现在数组的任意位置。如果数组中第一个元素正好是要查找的变量 x，那就不需要继续遍历剩下的 n - 1 个数据了，那时间复杂度是 O（1）.但如果数组中不存在变量 x，那就需要把整个数组都遍历一遍，那时间复杂度就成了 O（n）。所以，不同的情况下，这段代码的时间复杂度是不一样的。

为了表示代码在不同情况下的不同时间复杂度，需要引入三个概念：最好情况时间复杂度、最坏情况时间复杂度和平均情况时间复杂度。

> - **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就是，在最糟糕的情况下，执行这段代码的时间复杂度。**

就像刚刚讲的，在最理想的情况下，要查找的变量 x 正好是数组的第一个元素，这个时候对应的时间复杂度就是最好情况时间复杂度；如果数组中没有要查找的变量 x，就需要把整个数组都遍历一遍才行，所以这中最糟糕的情况下，对应的时间复杂度就是最坏情况时间复杂度。

平均情况时间复杂度

最好情况时间复杂度和最坏情况时间复杂度对应的都是极端情况下的代码复杂度，发生的概率其实并不大。

所以，**为了更好地表示平均情况下的复杂度，就需要引入另一个概念：平均情况时间复杂度，简称平均时间复杂度。**

要查找的变量 x 在数组中的位置，有 n + 1 种情况：**在数组的 0 ~ n - 1 位置中** 和 **不在数组中**。

把每种情况下，查找需要变量的元素个数累加起来，然后再除以 n + 1，就可以得到需要遍历元素个数的平均值，即：

$$\frac{1+2+3+...+n+n}{n+1} = \frac{1+2+3+..+n}{n+1} + \frac{n}{n+1} = \frac{n(n+1)}{2(n+1)} + \frac{2n}{2(n+1)} = \frac{n(n+3)}{2(n+1)}$$

时间复杂度的 大 O 标记法，可以省略掉系数、低阶、常量，所以，把刚刚这个公式简化之后，得到的平均复杂度就是 O（n）。

这个结论虽然是正确的，但是计算过程稍微有点儿问题。刚讲的这 n + 1种情况，出现的概率并不是一样的。

要查找的变量 x，要么在数组里，要不就不在数组里。这两种情况对应的概率统计起来很麻烦，为了方便理解，假设在数组中与不在数组中的概率都为 。

另外，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，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 中任意位置的概率是 1/(2n)。

因此，前面的推导过程中存在的最大问题就是，没有将各种情况发生的概率考虑进去。如果把每种情况发生的概率也考虑进去，那平均时间复杂度的计算过程就变成了这样：

$$1 \times \frac{1}{2n} + 2 \times \frac{1}{2n} + 3 \times \frac{1}{2n} + ... + n \times \frac{1}{2n} + n \times \frac{1}{2} = \frac{3n+1}{4}$$

这个值就是概率论中的 **加权平均值**，也叫作 **期望值**，所以平均时间复杂度的全称应该叫 **加权平均时间复杂度** 或者 **期望时间复杂度**。

引入概率之后，前面那段代码的加权平均值 .

用大 O表示法表示，去掉系数和常量，这段代码的加权平均时间复杂度仍然是 O（n）。

实际上，在大多数情况下，并不需要区分最好、最坏、平均情况时间复杂度三种情况。很多时候，使用一个复杂度就可以满足需求了。只有同一块代码在不同的情况下，时间复杂度有量级的差距，才会使用这三种复杂度表示法来区分。

均摊时间复杂度

下面这段代码实现了一个往数组中插入数据的功能。当数组满了之后，就是代码中的 count == array.length 时，用 for 循环遍历数组求和，并清空数组，将求和之后的 sum 值放到数组的第一个位置，然后再将新的数据插入。但如果数组一开始就有空闲的空间，则直接将数据插入数组。

```
// array表示一个长度为n的数组
// 代码中的array.length就等于n
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;
}
```

- 最理想的情况下，数组中有空闲空间，只需要将数组插入到数组下标为 count 的位置就可以了，所以最好情况时间复杂度为 O（1）。
- 最坏的情况下，数组中没有空闲了，需要先做一次数组的遍历求和，然后再将数据插入，所以最坏情况时间复杂度为 O（n）。
- 平均时间复杂度，将通过概率论的方法来分析：

假设数组的长度是 n，根据数据插入的位置不同，可以分为 n 中情况，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 种情况发生的概率都一样，都是1/(n+1)。

所以，根据加权平均的计算方法，求得平均时间复杂度就是：

$$1 \times \frac{1}{n+1} + 1 \times \frac{1}{n+1} + .. + 1 \times \frac{1}{n+1} + n \times \frac{1}{n+1} = O(1)$$ 。

至此为止，前面的最好、最坏、平均时间复杂度的计算，理解起来应该都没有问题。但是这个例子里的平均复杂度分析其实并不需要这么复杂，不需要引入概率论的知识。

这是为什么呢？

先来对比一下这个 insert() 的例子 和 前面那个 find() 的例子，就会发现这两者有很大差别。

首先，find() 函数在极端情况下，复杂度才为 O（1）。但 insert() 在大部分情况下，时间复杂度都为 O（1）。只有个别情况下，复杂度才比较高，为 O（n）。

其次，对于 insert() 函数来说，O（1）时间复杂度的插入和 O（n）时间复杂度的插入，出现的频率是非常有规律的，而且有一定的前后时序关系，一般都是一个 O（n）插入之后，紧跟着 n - 1 个 O（1）的插入操作，循环往复。

所以，针对这样一种特殊场景的复杂度分析，并不需要像之前讲平均复杂度分析法那样，找出所有的输入情况以及相应的发生概率，然后再计算加权平均值。

所以，引入了一种更加简单的分析方法：**摊还分析法**，通过摊还分析得到的时间复杂度，叫做 **均摊时间复杂度**。

> 均摊时间复杂度一般都等于最好情况时间复杂度。因为在大部分情况下，时间复杂度 O 都是 最好情况下的时间复杂度，只有在个别时刻才会退化为 最坏情况复杂度，所以把耗时多的操作的时间均摊到其他操作耗时少的操作上，平均情况下的耗时就接近 最好情况时间复杂度。

那究竟如何使用摊还分析法来分析算法的均摊时间复杂度呢？

还是继续看在数组中插入数据的例子。

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，均摊下来，这一组连续操作的均摊时间复杂度就是 O（1）

均摊时间复杂度、摊还分析应用场景

均摊时间复杂度和摊还分析应用场景比较特殊，所以并不会经常用到。为了方便理解记忆，简单总结一下它们的引用场景：

> 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，就可以将这一组操作放在一块儿分析，看是否能将较高的时间复杂度那次操作耗时，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。

尽管很多数据结构和算法书籍都花了很大力气来区分平均时间复杂度和均摊时间复杂度，但个人认为，均摊时间复杂度就是一种特殊的平均时间复杂度，没必要花太多精力去区分它们。最应该掌握的是它的分析方法，摊还分析。至于分析出来的结果是叫平均还是叫均摊，这只是个说法，并不重要。

练习

```
// 全局变量，大小为10的数组array，长度len，下标i。
int array[] = new int[10];
int len = 10;
int i = 0;

// 往数组中添加一个元素
void add(int element) {
   if (i >= len) { // 数组空间不够了
```

```
        // 重新申请一个2倍大小的数组空间
        int new_array[] = new int[len*2];
        // 把原来array数组中的数据依次copy到new_array
        for (int j = 0; j < len; ++j) {
          new_array[j] = array[j];
        }
        // new_array复制给array，array现在大小就是2倍len了
        array = new_array;
        len = 2 * len;
      }
      // 将element放到下标为i的位置，下标i加一
      array[i] = element;
      ++i;
    }
```

最好情况时间复杂度：O（1）

最坏情况时间复杂度：O（n）

平均时间复杂度：O(1)

用空间换时间

当内存空间充足的时候，如果更加追求代码的执行速度，就可以选择空间复杂度相对较高、但时间复杂度相对很低的算法或者数据结构。

相反，如果内存比较紧缺，比如代码跑在手机或者单片机上，这个时候，就要反过来用时间换空间的设计思路。

举缓存的例子，缓存实际上就是利用了空间换时间的设计思想。如果把数据存储在硬盘上，会比较节省内存，但每次查找数据都要询问一次硬盘，会比较慢。但如果通过缓存技术，事先将数据加载在内存中，虽然会比较耗费内存空间，但是每次数据查询的速度就大大提高了。

所以总结一下，对于执行较慢的程序，可以通过消耗更多的内存（空间换时间）来进行优化；而消耗过多内存的程序，可以通过消耗更多的时间（时间换空间）来降低内存的消耗。

注脚

笔记时间：2021-02-22 二次总结《算法与数据结构之美》王争专栏