算法设计与分析 LECTURE *

A Note for Recurrence

LECTURER: 杨启哲

LAST MODIFIED: 2023 年 10 月 18 日

这是一个关于递推式的一个简单 notes.

1 如何算出一个递推式

1.1 时间复杂性的回顾

我们首先要明确一点，就是一个算法它的运行时间是跟其输入有关的，也就是说算法的运行时间应该是关于输入的某种函数。这里关于输入的衡量严格来说指的是输入规模，但我们先不具体关注这一点，只要知道的是这是一个衡量输入的变量。比如输入是一个数组时，其实我们更为关注的时输入元素的个数。而输入时一个正整数的时候我们也可以视作是关于这个输入大小的一个函数。举个很简单的例子，大多数的排序算法所需要的时间都是一个依赖于元素个数的函数，比如快速排序，我们说其时间复杂性是 $\Theta(n\log n)$ 指的是对于 n 个元素而言进行排序所需要的时间是 $\Theta(n\log n)$。这里用 Θ 或者 O 符号的一个原因其实是我们想先忽略其系数，而只关心其增长的速度层次，因为哪怕是 $0.00000001n^2$ 最终也会比 $100000000n\log n$ 要增长的快。

Remark 1.1

事实上当输入是一个正整数 n 时，其输入规模实际上是 $\log n$，因为如果用二进制存储的话 n 需要 $\lceil\log n\rceil$ 位去存储；只要按一进制去存储时输入规模才会是 n。但我们依旧可以关心的是输入大小 n 与其运行时间 $T(n)$ 的关系。

我们讲过其实一个算法的运行时间可以根据这个算法里一个主要行为进行评判。有些时候我们可以直接通过算法的描述来求出相应的函数 $T(n)$，比如在第一次作业中的如下算法：

算法 1: COUNT

输入: 正整数 n
输出: 第 6 步的执行次数 $count$

```
1: count ← 0
2: for i ← 1 to ⌊log n⌋ do
3:     for j ← i to i + 5 do
4:         for k ← 1 to i^2 do
5:             count ← count + 1
6:         end for
7:     end for
8: end for
```

通过观察我们可以直接算出第五行 $count \leftarrow count + 1$ 的执行次数，这也是该算法的循环次数，或者说是这个算法的一个主要行为，从而其运行时间跟输入值 n 的关系为 $T(n) = \Theta(\log^3 n)$.

1.2 为什么是递推式？

上述说到，有的算法可以直接算出其主要行为的执行次数，但是当算法牵扯到一些递归调用的时候，直接计算是有困难的，比如考察下面这个算法：

算法 2: JustForRec

输入：正整数 $n = 2^k$
输出：一个正整数 sum

```
1: if n = 1 then return 1
2: end if
3: sum_1 ← JustForRec(n/2)
4: sum_2 ← JustForRec(n/2)
5: sum ← sum_1 + sum_2
6: return sum
```

这个算法其实就是计算了 n，主要操作行为其实就是第 5 行的加法，我们令 $T(n)$ 表示输入值为 n 时该算法的运行时间。由于其牵扯到了递归调用自身，似乎使得直接计算其次数变得困难。但我们再理一下这个算法究竟做了什么：

1. 规定了 $n = 1$ 这一特殊情况的返回值；
2. 调用了两次 JustForRec$(\frac{n}{2})$
3. 将上述的返回值加起来并返回。

再次强调，所谓这个算法的运行时间 $T(n)$ 是关于输入值 n 的一个函数，从而当 $n \geq 2$ 时这个算法实际运行时间如下：

1. 两次函数 JustForRec$(\frac{n}{2})$ 的运行时间；
2. 一次加法

显然一次 JustForRec$(\frac{n}{2})$ 的运行时间为 $T(\frac{n}{2})$，因此当 $n \geq 2$ 时整个算法的运行时间 $T(n)$ 的关系为：

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

而 $n = 1$ 时，该算法不做任何加法直接返回 1，因此 $T(1) = 0$。将两部分结合，我们就可以得到一个递推式：

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(\frac{n}{2}) + 1 & n \geq 2 \end{cases}$$

这就是递推式的得到方式。直白来说，当一个算法需要有其他的调用的时候，其运行时间就是其他调用过程的运行时间再加上调用完成后算法对这些结果处理时间之和。而对于分治算法而言，其递归调用很多都是同样但是规模更小的问题，从而我们就可能得到类似上述的递推式。此外当存在对自身的调用时，为了算法终止其往往会有初始条件，因此上述递推式会有一个 $n = 1$ 的特殊情况。

我们最后再看三个例子：

1. 考察如下的一个算法：

算法 3: FindMaxSub$(A, low, high)$

输入: 一个数组 A，数组的下界 low，数组的上界 $high$
输出: 一个数组 A 中的最大子数组

```
if low = high then
    return (low, high, A[low])
end if
mid ← ⌊(low+high)/2⌋
(left_low, left_high, left_sum) ← FindMaxSub(A, low, mid)
(right_low, right_high, right_sum) ← FindMaxSub(A, mid + 1, high)
(cross_low, cross_high, cross_sum) ← FindMaxCrossSub(A, low, mid, high)
if left_sum ≥ right_sum and left_sum ≥ cross_sum then
    return (left_low, left_high, left_sum)
else if right_sum ≥ left_sum and right_sum ≥ cross_sum then
    return (right_low, right_high, right_sum)
else
    return (cross_low, cross_high, cross_sum)
end if
```

首先我们要理解这个算法做了什么，它的输入有三个部分,A,low 和 $high$，但其实想表明的是该算法处理的就是 A 数组里下标从 low 到 $high$ 的子数组。这么写其实就是方便递归调用不产生新的空间开销。整个算法分为如下几部分：

(1) 当 $low = high$ 时，说明只有一个元素，直接返回这个元素, 这是特殊情况的处理。
(2) 调用 FindMaxSub(A, low, mid) 和 FindMaxCrossSub$(A, low, mid, high)$。注意到 mid 的计算，因此这两个子过程实际上数组规模小了一半。
(3) 调用 FindMaxCrossSub$(A, low, mid, high)$ 和后面的一些处理。我们这里没有列出这个调用，实际上它并不关键。大家可以认为这是一个线性时间的操作。也就是对于一个规模为 n 的数组，这个调用的时间复杂性为 $O(n)$，或者写为 cn，这里 c 是某个确定的常数。

从而我们令 $T(n)$ 表示该算法对于元素个数为 n 的数组所欲要的运行时间。从而当 $n \geq 2$ 时该算法的运行时间可以表示为：

$$T(n) = 2T(\frac{n}{2}) + cn$$

对于整个算法而言，其运行时间则为：

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(\frac{n}{2}) + cn & n \geq 2 \end{cases}$$

2. 我们再来看一个归纳的例子，即上课讲过的求排列个数：

算法 4: Perm2(n)

输入: 正整数 n
输出: $1, \ldots, n$ 的所有排列

```
for i ← 1 to n do
    P[i] ← 0
end for
perm2(n)
```

过程: $perm2(m)$
5: if $m = 0$ then output $P[1, \ldots n]$
6: else
7: for $j \leftarrow 1$ to n do
8: if $P[j] = 0$ then
9: $P[j] \leftarrow m$
10: $perm2(m-1)$
11: $P[j] \leftarrow 0$
12: end if
13: end for
14: end if

可以看到 Perm2(n) 实际由一个 n 次循环和调用 $perm2(n)$ 组成。因此关于其运行时间我们只需要关注 $perm2(m)$ 即可。但需要注意的是调用 $perm2$ 中 $perm2$ 中的 n 已经是个固定的数，要和其输入区分开来。令 $T(m)$ 表示输入值为 m 的时候 $perm2(m)$ 的运行时间，则我们有：

- $m = 0$ 时算法是直接输出当前数组排列的，因此 $T(m) = 0$
- $m \geq 1$ 时，算法将执行一个 n 次的循环，但由于此时至多有 m 个位置为 0，因此 $perm(m-1)$ 恰好被调用了 m 次，因此整个运行时间可以表示为：

$$T(m) = mT(m-1) + n$$

因此算法 $perm2(m)$ 的运行时间可以表示为：

$$T(m) = \begin{cases} 0 & m = 1 \\ mT(m-1) + n & m \geq 2 \end{cases}$$

当然可以说算法 Perm2(n) 的运行时间是 $T(n)$ 再加上 $O(n)$ 的初始化循环，但其实后者相比于前者来说实在太小可以忽略。

3. 最后则是一个简单的例子。

算法 5: Multi(n)

输入: 正整数 n
输出: 一个正整数 sum
1: if $n = 1$ then return 1
2: end if
3: $sum \leftarrow 0$
4: $sum_1 \leftarrow$ Multi($n-1$)
5: for $i \leftarrow 1$ to sum_1 do
6: $sum \leftarrow sum +$ Multi($n-1$)
7: end for
8: return sum

大家可以自己算一下，令 $T(n)$ 表示这个算法输入值为 n 的时候的运行时间，则有：

$$T(n) = \begin{cases} 0 & n = 1 \\ T^2(n-1) + 1 & n \geq 2 \end{cases}$$

2 如何求解一个递推式

上面我们介绍了如何求一个递推式，下面我们简单介绍一下如何求解递推式。首先求解递推式其实有点像我们之前高中学过的数列通项求解，因此很多技巧也是相同的。并且我们也必须强调不是所有的递推式都能推出相应 $T(n)$ 的准确表达，比如上面的最后一个例子. 因此我们这边主要介绍基于分治算法的递推式的求解，其特征一般就是所有系数都是常数。课上也讲到，我们有一个很好的手段-主定理 (Master Theorem)。

Theorem 2.1 (主定理 (Master Theorem))

假设递推式具有如下形式：

$$T(n) = \begin{cases} O(1) & n = 1 \\ aT(\frac{n}{b}) + O(n^d) & n > 1 \end{cases}$$

则 $T(n)$ 满足：

$$T(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d \log n) & a = b^d \\ O(n^{\log_b a}) & a > b^d \end{cases}$$

我们不过多的阐述主定理的运用和证明。我们主要来介绍一下如果一个分治算法的递推式不满足主定理的形式该如何解决，比如下列形式：

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\frac{2}{3}n) + T(\frac{1}{4}n) + 2n & n \geq 2 \end{cases}$$

其实解决这种式子的一个简单想法就是展开。比如在上述过程中：

$$\begin{aligned} T(n) &= T(\frac{2}{3}n) + T(\frac{1}{4}n) + 2n \\ &= (T(\frac{4}{9}n) + T(\frac{1}{6}n) + \frac{4}{3}n) + (T(\frac{1}{6}n) + T(\frac{1}{16}n) + \frac{1}{2}n) + 2n \\ &= (T(\frac{4}{9}n) + T(\frac{1}{6}n) + T(\frac{1}{6}n) + T(\frac{1}{16}n)) + (\frac{2}{3} + \frac{1}{4}) \cdot 2n + 2n \\ &= \cdots \end{aligned}$$

大家可以自己动手算一算，就会发现对于最后的 $2n$ 来说，每往下展开一层，其要加上一个 $(\frac{2}{3} + \frac{1}{4})$ 的倍数，即最后一部分的形式为:

$$2n + 2n \cdot (\frac{2}{3} + \frac{1}{4}) + 2n \cdot (\frac{2}{3} + \frac{1}{4})^2 + \cdots$$

而由于前面初始条件为 0，因此最终前面的所有全部为 0，于是不用考虑。注意到 $\frac{2}{3} + \frac{1}{4} < 1$，从而可以发现上式最后算出来还是 $O(n)$ 的。(这里也可以发现展开高度是 $o(\log n)$ 的，因为 T 里面的 n 的值之和每次展开都在以一个常数倍 $(\frac{2}{3} + \frac{1}{4})$ 缩小。) 一般来说，对于类似如下的递推式：

$$T(n) = \begin{cases} c & n = 1 \\ T(A_1 n) + T(A_2 n) + \cdots T(A_k n) + f(n) & n \geq 2 \end{cases}$$

如果 $A_1 + \cdots A_k < 1$，则该式算出来还是在 $O(f(n))$ 的，如果 ≥ 1 则情况会更加复杂需要进一步的讨论。我们这里就不多加阐述了。

3 总结

这是一个针对递推式的简单介绍，写的出发点是在于有同学在问卷里面反映不知道如何求递推式，因此写了这么一个 notss 希望对同学有所帮助。由于重点在于第一部分，第二部分写的较为简略。同时行文比较仓促可能会有不少错误的地方，欢迎指正。