2.8 效率

::: details INFO
译者：laziyu

来源：2.8 Efficiency

对应：无
:::

决定如何表示和处理数据通常受到替代方案效率的影响。效率指的是表示或处理所使用的计算资源，例如计算函数结果或表示对象所需的时间和内存量。这些数量可以根据实现的细节而大大不同。

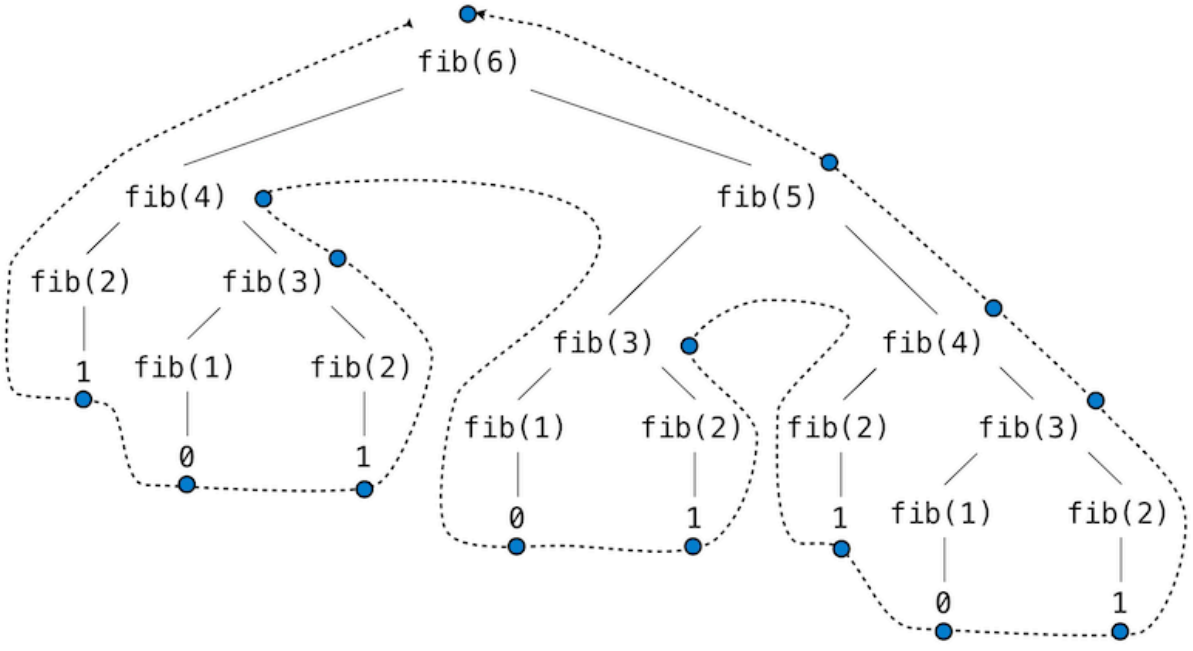
2.8.1 测量效率

测量程序运行所需的时间或消耗的内存确切值是具有挑战性的，因为结果取决于计算机配置的许多细节。更可靠地表征程序的效率的方法是测量某些事件发生的次数，例如函数调用次数。

让我们回到我们第一个树递归函数，即用于计算斐波那契数列中的数字的 `fib` 函数。

```
>>> def fib(n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return fib(n - 2) + fib(n - 1)
>>> fib(5)
5
```

考虑计算 `fib(6)` 时的计算模式，如下所示，为了计算 `fib(5)`，我们计算 `fib(3)` 和 `fib(4)`。而要计算 `fib(3)`，我们需要计算 `fib(1)` 和 `fib(2)`。总体而言，这个演化过程看起来像一棵树。每个蓝色圆点表示在遍历这棵树时计算出的一个斐波那契数的完成计算。



这个函数作为一个典型的树递归函数具有教学意义，但它是计算斐波那契数的一种极其低效的方式，因为它进行了大量的冗余计算。计算 `fib(3)` 的整个过程被重复执行。

我们可以测量这种低效性。这个高阶的 `count` 函数返回一个与其参数等效的函数，并且还维护一个 `call_count` 属性。通过这种方式，我们可以检查 `fib` 函数被调用的次数。

```
>>> def count(f):
        def counted(n):
            counted.call_count += 1
            return f(n)
        counted.call_count = 0
        return counted
```

通过计算对 `fib` 函数的调用次数，我们可以发现所需的调用次数增长速度比斐波那契数列本身还要快。这种调用的快速增长是树递归函数的特征。

```
>>> fib = count(fib)
>>> fib(19)
4181
>>> fib.call_count
10946
```

空间。要了解函数的空间需求，我们通常必须指定在计算的环境模型中如何使用、保留和回收内存。在计算表达式时，解释器会保存所有活动的环境，以及这些环境引用的所有值和帧。我们称一个环境是活动的，如果它为正在计算的某个表达式提供了评估上下文。每当为其创建第一个帧的函数调用最终返回时，环境将变为非活动状态。

例如，在计算 `fib` 时，解释器按照之前显示的顺序计算每个值，遍历树的结构。为此，它只需要跟踪在计算的任何时刻位于当前节点上方的那些节点。用于计算其他分支的内存可以被回收，因为它不会影响未来的计算。总的来说，树递归函数所需的空间将与树的最大深度成比例。

以下的图表描述了计算 `fib(3)` 时所创建的环境。在计算初始应用 `fib` 的返回表达式时，表达式 `fib(n-2)` 被计算，得到一个值为 0。一旦这个值被计算出来，相应的环境帧（被标记为灰色）就不再需要：它不是一个活动环境的一部分。因此，一个良好设计的解释器可以回收用于存储该帧的内存。另一方面，如果解释器当前正在计算 `fib(n-1)` ，则通过这个 `fib` 应用（其中 n 为 2）所创建的环境是活动的。反过来，最初用于将 `fib` 应用于 3 的环境是活动的，因为其返回值尚未被计算出来。

高阶函数 `count_frames` 用于跟踪尚未返回的函数调用次数 `open_count` 。它通过在计算过程中记录当前活动的函数调用次数来实现。 `max_count` 属性是 `open_count` 曾经达到的最大值，它对应于在计算过程中同时处于活动状态的最大帧数。

```
>>> def count_frames(f):
        def counted(n):
            counted.open_count += 1
            counted.max_count = max(counted.max_count, counted.open_count)
            result = f(n)
            counted.open_count -= 1
            return result
        counted.open_count = 0
        counted.max_count = 0
        return counted
>>> fib = count_frames(fib)
>>> fib(19)
4181
>>> fib.open_count
0
```

```
>>> fib.max_count
19
>>> fib(24)
46368
>>> fib.max_count
24
```

总结一下，`fib` 函数的空间要求（以活动帧数衡量）比输入小一个单位，这往往是较小的。而以递归调用次数衡量的时间要求比输出大，这往往是巨大的。

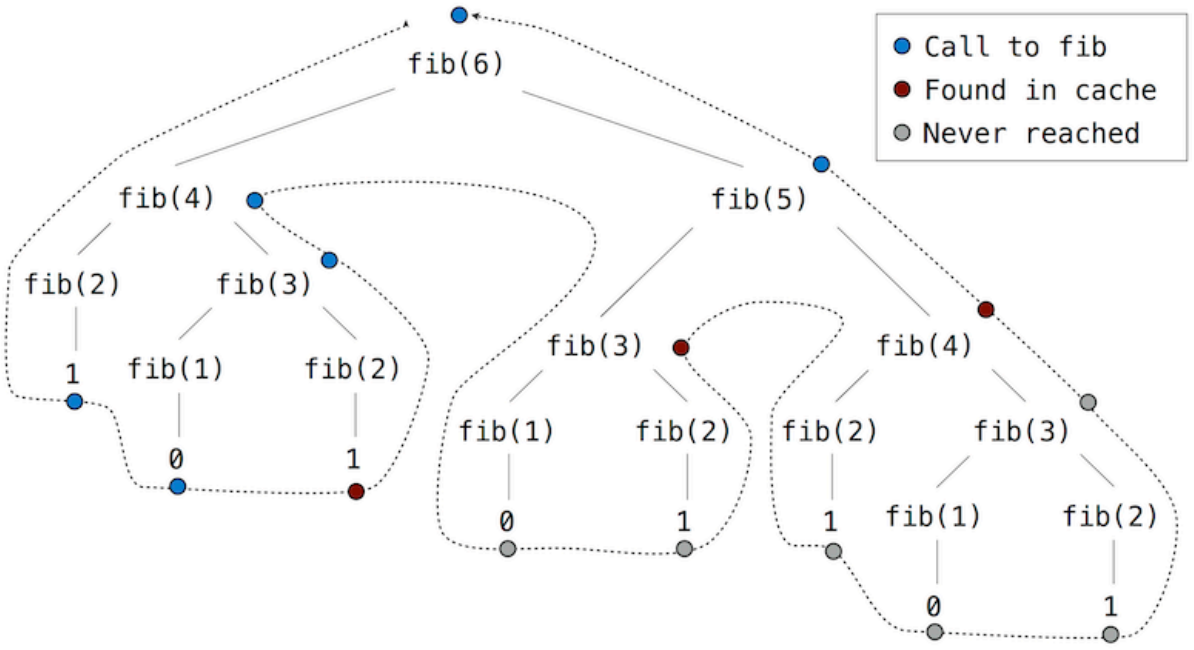
2.8.2 记忆化

树递归的计算过程通常可以通过记忆化（Memoization）来提高效率，这是一种增加递归函数效率的强大技术。记忆化函数会保存之前接收到的参数的返回值。如果第二次调用 `fib(25)`，它不会再通过递归重新计算返回值，而是直接返回已经计算好的结果。

记忆化可以自然地表达为一个高阶函数，也可以用作装饰器。下面的定义创建了一个缓存（cache），用于存储先前计算过的结果，索引是它们计算所用的参数。使用字典作为缓存的数据结构要求被记忆化的函数的参数是不可变的。

```
>>> def memo(f):
        cache = {}
        def memorized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memorized
```

如果我们将 `memo` 应用于斐波那契数列的递归计算，将会产生一个新的计算模式，如下所示。



在计算 `fib(5)` 时，在计算右侧分支的 `fib(4)` 时，已经计算过的 `fib(2)` 和 `fib(3)` 的结果被重复使用。因此，许多树递归计算根本不需要进行。

通过使用 `count` 函数，我们可以看到对于每个唯一的输入，`fib` 函数实际上只被调用一次。

```
>>> counted_fib = count(fib)
>>> fib =memo(count_fib)
>>> fib(19)
4181
>>> counted_fib.call_count
20
>>> fib(34)
5702887
>>> counted_fib.call_count
35
```

2.8.3 增长阶数

计算过程在消耗计算资源（时间和空间）的速率上可能有很大的差异，正如之前的例子所展示的那样。然而，准确地确定调用函数时将使用多少空间或时间是一项非常困难的任务，这取决于许多因素。分析一个计算过程的有用方法是将其与一组具有类似需求的过程分类。一种有用的分类是该过程的增长阶（Orders of Growth），它以简单的术语表达了计算过程的资源需求随输入的函数增长。

为了介绍增长阶的概念，我们将分析下面的函数 `count_factors`，它用于计算能够整除输入 n 的整数的数量。该函数尝试将 n 除以小于等于其平方根的每个整数。这个实现利用了以下事实：如果 k 能够整除 n，且 $k < \sqrt{n}$，那么必然存在另一个因子 $j = n / k$，使得 $j > \sqrt{n}$。

计算 `count_factors` 所需的时间是多少？准确的答案会因不同的计算机而异，但我们可以对涉及的计算量做出一些有用的一般观察。这个过程执行 `while` 语句的次数是小于等于 $\sqrt{n}$ 的最大整数。在 `while` 语句之前和之后的语句分别执行一次。因此，总共执行的语句数为 $w*\sqrt{n}+v$，其中 w 是 `while` 循环体中的语句数， v 是 `while` 循环外的语句数。虽然这不是一个精确的公式，但它通常能够描述作为输入 n 的函数而要计算的时间量。

要获得更精确的描述是很困难的。常量 w 和 v 实际上并不是常数，因为对因子的赋值语句有时会被执行，有时不会。增长阶分析使我们能够忽略这些细节，而是着重于增长的一般趋势。特别是，`count_factors` 的增长阶表达了以 $\sqrt{n}$ 速度计算 `count_factors(n)` 所需的时间，其中可能存在一些常量因子的误差范围。

Theta 符号（Theta Notation） 是一种用于表示算法的渐进性能的数学符号。在 Theta 符号中，我们考虑一个参数 n，它衡量了某个计算过程的输入规模大小，并且用 $R(n)$ 表示该过程对于输入规模 n 所需的某种资源量。在我们之前的例子中，我们将 n 视为要计算给定函数的数值，但还有其他可能性。例如，如果我们的目标是计算一个数的平方根的近似值，我们可以将 n 视为所需的有效位数。

$R(n)$ 可以表示所使用的内存量、执行的基本机器步骤数量等等。在每次执行固定数量步骤的计算机中，计算表达式所需的时间将与在计算过程中执行的基本步骤数量成正比。

我们可以说如果存在正常数 k_1 和 k_2 （与 n 无关），使得对于任何大于某个最小值 m 的 n，成立如下不等式：

$$k1·f(n)≤R(n)≤k2·f(n)$$

我们称 $R(n)$ 的增长阶为 $Θ(f(n))$，用 $R(n)=Θ(f(n))$ 表示（读作“*theta of f(n)*”）。换句话说，对于大的 n， $R(n)$ 的值总是夹在两个与 $f(n)$ 成比例的值之间：

- 一个下界 $k1 · f(n)$ 和
- 一个上界 $k2 · f(n)$

通过检查函数体，我们可以应用这个定义来展示计算 `count_factors(n)` 所需的步骤数量随着 $Θ(\sqrt{n})$ 增长

首先，我们选择 $k_1=1$ 和 $m=0$ ，以便下界表明对于任何 $n>0$ ， `count_factors(n)` 至少需要 $1·\sqrt{n}$ 步。在 `while` 循环之外至少有 4 行被执行，每行至少需要 1 步来执行。在 `while` 循环体内至少有 2 行被执行，还有 `while` 头本身。所有这些都需要至少 1 步。 `while` 循环体至少被执行 $\sqrt{n}-1$ 次。组合这些下界，我们可以看到该过程至少需要 $4+3·(\sqrt{n}−1)$ 步，这总是大于 $k_1·\sqrt{n}$ 。

其次，我们可以验证上界。我们假设 `count_factors` 函数体内的任何一行最多需要 p 步。尽管这个假设对于 Python 中的每一行代码都不成立，但在这种情况下是成立的。然后，计算 `count_factors(n)` 最多可能需要 $p·(5+4\sqrt{n})$ 步，因为在 `while` 循环之外有 5 行代码，在循环内有 4 行代码（包括 `while` 头）。即使每个 `if` 头都评估为 true，这个上界仍然成立。最后，如果我们选择 $k_2=5p$ ，那么所需的步骤总是小于 $k_2·\sqrt{n}$ 。我们的论证到这里完成了。

2.8.4 示例：指数运算

考虑计算给定数的指数问题。我们希望有一个函数，它以底数 b 和正整数指数 n 作为参数，并计算 b_n 。一种实现方法是通过递归定义：

$$b^n = b·b^{(n-1)}$$
$$b^0 = 1$$

这个递归定义可以很容易的转化为递归函数：

```
>>> def exp(b, n):
        if n == 0:
            return 1
        return b * exp(b, n - 1)
```

这是一个线性递归过程，它需要 $Θ(n)$ 步骤和 $Θ(n)$ 空间。就像阶乘一样，我们可以很容易地设计一个等效的线性迭代版本，它需要相似数量的步骤，但只需要常量空间。

这意味着递归的指数计算是一个线性过程，它的时间和空间复杂度都随着指数 n 的增长而线性增加。与此相反，通过线性迭代，我们可以将计算复杂度优化为仅需常数级的空间，但步骤数量仍然是与指数 n 成线性关系的。

```
>>> def exp_iter(b, n):
        result = 1
        for _ in range(n):
            result = result * b
        return result
```

我们可以通过使用连续平方法来以更少的步骤计算指数。例如，计算 b 的 8 次幂可以如下进行：

$$b^8 = b * (b * (b * (b * (b * (b * (b * b))))))$$

而我们可以使用三次乘法来计算它：

$$b^2 = b * b $$
$$b^4= b^2 _ b^2$$
$$b^8= b^4 _ b^4$$

这种方法对于指数是 2 的幂次的情况非常有效。我们也可以利用连续平方法来计算一般情况下的指数，如果我们使用如下递归规则：

$$b^n = \begin{cases} \left(b^{\frac{n}{2}}\right)^2 & \text{如果 } n \text{ 是偶数} \\ b \cdot b^{n-1} & \text{如果 } n \text{ 是奇数} \end{cases}$$

我们也可以用一个递归函数的方式来表达这个方法：

```
>>> def square(x):
        return x * x

>>> def fast_exp(b, n):
        if n == 0:
            return 1
        if n % 2 == 0:
            return square(fast_exp(b, n // 2))
        else:
            return b * fast_exp(b, n - 1)

>>> fast_exp(2, 100)
1267650600228229401496703205376
```

通过快速指数运算（`fast_exp`），计算过程在时间和空间上都以对数级别随 n 增长。要理解这一点，观察一下使用 `fast_exp` 计算 b 的 $2n$ 次幂仅仅比计算 b 的 n 次幂多进行一次乘法。因此，我们可以看到每一次允许的新乘法使得我们能够计算的指数大小翻倍（粗略估计）。因此，对于指数 n，所需的乘法次数大致以 2 为底数的 n 的对数增长。这个过程具有 $Θ(log_{n})$ 的增长阶。

当 n 变得很大时， $Θ(log_{n})$ 增长阶与 $Θ(n)$ 增长阶之间的差异会变得非常明显。例如，对于 n 为 1000，使用 `fast_exp` 只需要 14 次乘法，而不是 1000 次。这显示了使用快速指数运算相对于普通指数运算在效率上的巨大优势。

2.8.5 增长类别

增长阶是为了简化计算过程的分析和比较而设计的。许多不同的计算过程可以具有等效的增长阶，这表示它们的增长方式相似。对于计算机科学家来说，了解和识别常见的增长阶并确定具有相同增长阶的过程是一项重要的技能。

常数项：常数项不影响计算过程的增长阶。因此，例如， $Θ(n)$ 和 $Θ(500·n)$ 具有相同的增长阶。这个性质来自于 $Θ$ 符号的定义，它允许我们选择任意的常数 k_1 和 k_2（比如 $\frac{1}{500}$ ）作为上界和下界。为了简洁起见，增长阶中常数通常被忽略。

对数：对数的底数不影响计算过程的增长阶。例如，$log2(n)$ 和 $log10(n)$ 具有相同的增长阶。改变对数的底数等价于乘以一个常数因子。

嵌套：当内部的计算过程在外部过程的每一步中重复执行时，整个过程的增长阶是外部和内部过程的步骤数的乘积。

例如，下面的函数 `overlap` 计算列表 `a` 中与列表 `b` 中出现的元素数量。

```
>>> def overlap(a, b):
        count = 0
        for item in a:
            if item in b:
                count += 1
        return count

>>> overlap([1, 3, 2, 2, 5, 1], [5, 4 ,2])
3
```

对于列表的 `in` 运算符，其时间复杂度为 $Θ(n)$ ，其中 n 是列表 `b` 的长度。它被应用 $Θ(m)$ 次，其中 m 是列表 `a` 的长度。表达式 `item in b` 是内部过程，而 `for item in a` 循环是外部过程。该函数的总增长阶是 $Θ(m·n)$ 。

低阶项。随着计算过程的输入增长，计算中增长最快的部分将主导总的资源使用。$Θ$ 符号捕捉了这种直觉。总的来说，除了增长最快的项外，其他项都可以忽略而不影响增长阶。

例如，考虑函数 `one_more`，它返回列表 `a` 中有多少个元素是另一个元素的值加 1。也就是说，在列表 `[3, 14, 15, 9]` 中，元素 15 比 14 大 1，所以 `one_more` 将返回 1。

```
>>> def one_more(a):
        return overlap(a, [x + 1 for x in a])

>>> one_more([3, 14, 15, 9])
1
```

这个计算分为两个部分：列表推导和对 `overlap` 的调用。对于长度为 n 的列表 a，列表推导需要 $Θ(n)$ 步，而对 `overlap` 的调用需要 $Θ(n^2)$ 步。这两部分的总步数是 $Θ(n + n^2)$ ，但这并不是表达增长阶最简单的方式。

$Θ(n^2 + k*n)$ 和 $Θ(n^2)$ 对于任意常数 k 来说是等价的，因为对于任何 k，n^2 项在足够大的 n 下将主导总和。这里的边界要求仅对于大于某个最小值 m 的 n 成立，从而确立了这种等价性。为简洁起见，增长阶中的低阶项通常被忽略，所以我们不会在 $theta$ 表达式中看到求和。

常见的类别。 通过这些等价性，我们可以得到一小组常见的类别来描述大多数计算过程。最常见的类别如下，按从最慢到最快的增长顺序列出，并描述了随着输入增加而增长的情况。接下来将给出每个类别的例子。

类别	$Θ$ 表示	增长阶描述	例子
常数	$Θ(1)$	增长与输入无关	`abs`
对数	$Θ(log_{n})$	增长与输入的大小成正比	`fast_exp`
线性	$Θ(n)$	随着输入的递增，计算过程所需的资源会增加 n 个单位	`exp`
乘方	$Θ(n^2)$	随着输入的递增，计算过程所需的资源会增加 n 个单位	`one_more`
指数	$Θ(b^n)$	随着输入的递增，计算过程所需的资源会成倍增加	`fib`

除了这些之外，还有其他的增长阶类别，例如 `count_factors` 的 $Θ(\sqrt{n})$ 增长。然而，上述列举的类别是特别常见的。

指数增长描述了许多不同的增长阶，因为改变底数 b 会影响增长阶。例如，我们的树递归斐波那契计算 `fib` 的步数随输入 n 呈指数增长。特别地，我们可以证明第 n 个斐波那契数是最接近于

$$\frac{\phi^{n-2}}{\sqrt{5}}$$

的整数，其中 $ϕ$ 是黄金比例：

$$\phi=\frac{1+\sqrt{5}}{2}≈1.6180$$

我们还提到步数与结果值成正比，所以树递归过程需要 $Θ(ϕ^n)$ 步，这是一个随着 n 以指数级别增长的函数。