第5章

分治

在那些最强有力的问题求解技术中, 有一种方法是将问题拆成更小且更易解决的部分。较小的问题没有那么强势, 于是我们便有可能将注意力集中在那些细节问题上, 而这是从整体上研究原问题时所看不到的。要是能将问题拆成更小的同类子问题, 那么一个递归算法就会开始逐渐成形。如今每台计算机都有多核处理器, 而要想发挥并行处理的功用, 得将任务分解成至少与处理器个数相同的子任务。

算法设计中有两种重要的范式, 它们都基于将问题拆分至更小问题的这种思想。在第10章中我们会讲解动态规划, 它通常先从问题中移除一个元素, 再解决这个新形成的小问题, 然后在较小问题解的基础上以合适的方式再将移除元素重新加回来。而**分治**(divide-and-conquer)则是将问题划分成几部分, 比如说两半, 再分别求解这两部分, 最后将它们重新整合起来从而形成完整的解。

要将分治作为一种算法技术来使用, 我们必须将问题分成两个更小的子问题, 再对它们分别递归求解, 最后整合这两个子问题的解从而得到原问题的解。只要整合所用时间少于求解这两个子问题的时间, 便能获得一个高效的算法。归并排序(4.5节中已讨论)是分治算法的一个经典实例。对于两个具有$n/2$个元素的有序表, 我们需要花费$O(n \log n)$时间才能得到, 而归并这两个有序列表却只需线性时间, 因此归并排序很高效。

许多重要的算法, 包括归并排序、快速傅里叶变换和Strassen矩阵相乘算法, 其优异的性能都要归功于分治这项算法设计技术。然而我发现, 除了二分查找及其众多变种之外, 分治是一种在实际中很难用好的设计技术。此外, 递推关系(recurrence relation)决定了此类递归算法的性能开销, 而我们对分治算法的分析能力取决于能否求出递推关系的渐近记号, 因此我们还将介绍递推式(recurrence)的求解技术。

5.1 二分查找及相关算法

要在有序数组S中查找键值, 二分查找是一种很快的算法, 而它可称得上是所有分治算法之母。为了查找q这个键, 我们用q和位于中间的键$S[n/2]$比较。如果q出现在$S[n/2]$之前, 它必然属于S的前半部; 否则它必然属于S的后半部。不断在q每次所归属的那部分上递归重复此过程, 我们就能在$\lceil \log n \rceil$次比较之内找到该键所处位置——相比于使用顺序查找平均所需$n/2$次比较, 这是一个很大的超越。[1]

```
int binary_search(item_type s[], item_type key, int low, int high)
{
    int middle;                         /* 中间元素的下标 */
    if (low > high)
```

[1] 译者注: 我们对二分查找的代码实现略有更改, 原文中计算`middle`时有可能会产生溢出错误。

```
        return -1;                  /* 未找到此键 */
    middle = low + (high - low) / 2;

    if (s[middle] == key)
        return middle;

    if (s[middle] > key)
        return binary_search(s, key, low, middle - 1);
    else
        return binary_search(s, key, middle + 1, high);
}
```

上述这些你可能已经学过了。不过, 重要的其实是去体会二分查找到底有多快。**二十问题**(twenty questions)是一个流行的儿童游戏, 一名游戏者选一个单词, 而另一个不断问"真/假"问题[1]来试着去猜出那个单词。如果问过20个问题后单词仍不能确认, 选词的游戏者获胜; 否则, 提问的游戏者赢得奖金。事实上, 提问的游戏者始终有一种策略能赢得奖金, 也就是基于二分查找的方案。指定一部纸版字典, 提问的游戏者从中间翻开, 再选一个单词(比如说"move"), 然后问那个未知单词依字母序是否在"move"之前。由于普通字典收词数大约为50 000到200 000, 我们能确保上述过程会在20次提问内结束。

5.1.1 出现次数的计数

有不少巧妙的算法都是二分查找稍加改动而得。假定我们想对某个有序数组中给定键k(比如说Skiena)的出现次数进行计数。因为排序会将所有k的副本集中在一段紧密相连的区段中, 此问题即可简化为寻找符合要求的区段并度量其长度。

使用前面所给的二分查找子程序, 可让我们在$O(\log n)$时间内找到一个键值为k的元素(记为x)的下标, 显然x所在的区段符合要求。要定出区段边界很自然会想到从x向左挨个测试元素, 直到我们找到首个不同于待查键的元素为止, 然后再从x向右重复此查找过程。右边界和左边界的差再加1, 刚好就是k出现次数的计数值。

上述算法可在$O(\log n + s)$时间内运行完毕, 其中s是所给键的出现次数。如果整个数组由完全相同的键构成, 此时间可能会与直接查找的线性时间一样糟。若将二分查找修改成寻找包含k的那个区段的*边界*, 而不是找k本身, 我们可得到一个更快的算法。假定我们从上述实现中删去关于相等性的测试:

```
    if (s[middle] == key)
        return middle;
```

并在每个不成功的查找中返回下标`high`而不是`-1`。由于没有相等性测试, 所有查找现在都将是不成功的。每当待查键和一个具有相同键值的数组元素比较时, 查找便会在右半部继续执行, 最终停在该区段的右边界。改变二元比较算符`>`的方向后, 再重复此查找过程可将我们带至该区段的左边界。两次查找均用时为$O(\log n)$, 因此无论区段大小如何, 我们都能在对数时间内完成k所出现次数的计数工作。

[1] 译者注: 只能用"真/假"或"是/否"回答的问题。

若是将我们的二分查找函数修改为在搜索失败时返回`(low + high) / 2`,[1] 而非原有的`-1`, 便可获得一个位于两个特定数组元素之间的位置, 若是数组中能找到待查键k, 其实这两个元素所在的位置本应是成功返回的下标值。以上变更为我们提供了求解以上串结长度问题的另一种方案: 我们基于这种修订方案寻找$k-\epsilon$和$k+\epsilon$所在位置, 其中ϵ是一个足够小的常数, 能保证数组中没有在这两个值之间的键, 进而也意味着两次查找必然都会失败。显然, 这种基于两次二分查找的方案同样只需要$O(\log n)$时间。

5.1.2 单侧二分查找

现在假定我们有一个数组, 其构成是一连串0再接着一连串但个数未知的1, 我们想要找出它们之间的转折点。如果我们知道数组中元素个数的上界n, 在此数组上的二分查找将在$\lceil \log n \rceil$次测试内提供转折点的下标。

要是没有n这样的上界, 我们可在逐步增大的区间$(A[1], A[2], A[4], A[8], A[16], \cdots)$上不断测试, 直到找出首个非零值为止。于是我们拥有了一个目标数据滑动窗, 而且很适合二分查找去处理。无论数组实际长度为多少, 这种**单侧二分查找**(one-sided binary search)都能在至多$2\lceil \log p \rceil$次比较内找到转折点的下标p。只要我们想去寻找当前位置附近的某个键, 这时候最有用的就是单侧二分查找。[2]

5.1.3 平方根和其他方根

n的平方根是平方等于n的数r, 即$r^2 = n$。每台袖珍计算器都具备平方根计算的功能, 但是研制一个高效计算平方根的算法对我们来说依旧很有启发性。

首先, 注意到$n \geqslant 1$情况下n的平方根必然在1到n之间。令$l = 1$且$r = n$。考虑此区间的中点$m = (l + r)/2$。[3] m^2与n相比, 是大还是小? 如果$n > m^2$, 那么待求平方根肯定比m大, 于是令$l = m$再让算法重复上述步骤; 如果$n < m^2$, 那么待求平方根肯定小于m, 于是令$r = m$再让算法重复上述步骤。无论哪种情况下, 我们都只用一次比较便将区间减半。因此, $\lceil \log n \rceil$轮之后我们将能找到n的平方根, 且偏差在$\pm 1/2$以内。

这种**对分法**(数值分析中的叫法)还能用于更一般性的问题中, 也即寻找方程的根。若存在x使得$f(x) = 0$则称x是方程f的根。假定我们从满足$f(l) > 0$和$f(r) < 0$的初值l和r开始处理。如果f是连续函数, 在l和r之间肯定存在一个根。取$m = (l + r)/2$, 每次测试$f(m)$, 我们就能将这个包含方程根的窗口$[l, r]$截半, 而切掉哪半段则视$f(m)$的符号而定, 直到我们的估计值(即m)达到精度要求为止。

数值分析中专门有一类所谓的"求根算法", 它们比二分查找收敛更快, 事实上也是由于能快速求解前面这两个问题而得名的。二分查找总是去测试区间的中点, 而求根算法却不一定这样做, 它们以插值来找一个离实际的根值更近的测试点。不过, 二分查找的优点在于

[1] 译者注: 此处得小心处理溢出问题, 不过通常搜索失败时上下界差值为1, 可对原表达式化简处理。实际上, 这种方案不是很方便, 而且随后的ϵ选取若无线性扫描也未能保证稳妥。

[2] 译者注: 当$p \ll n$时, $2\lceil \log p \rceil$这个开销比$\lceil \log n \rceil$划算得多。更重要的是, 对于上界未知的单调型问题(不限于处理数组), 单侧二分查找更能发挥功用, 例如对某个单调递增函数找出满足特定约束的最大值。

[3] 译者注: 连续型问题取中点比起离散型的二分查找算法方便多了, 不妨仔细体会。

简单和稳健, 并且在不附加关于待计算函数特性的情况下, 它仍会尽自己所能去快速地[1]完成计算。

领悟要义: 二分查找及其变种是分治算法的完美典范。

5.2 算法征战逸事: 错中揪错

Yutong起身宣布了他几周辛勤工作的成果。“死了。” 他愤怒地说。随即, 房间中每个人都发出了哀叹。

我所在的团队正在研发一种制造疫苗的新方法: 合成减毒病毒工程(Synthetic Attenuated Virus Engineering)或者叫SAVE(刚好是“拯救”)。由于基因编码的运行机制, 对于给定任何长为n的蛋白质, 一般会有3^n种可用于编码的不同基因序列。打眼一看, 似乎它们都一样, 因为这些所描述的是完全相同的蛋白质。但是这3^n个同义基因序列中的每一个, 使用生物机器的方式都存在些许不同, 而翻译的速度也都有所差异。

通过将一种危险性较低的基因取代原有病毒基因, 我们有望造出某种弱毒疫苗: 一种能起到同样致病效果但毒性较弱的制剂。我们人类的身体可以在不得病的前提下打败稍弱的病毒, 并在这个过程中训练免疫系统将来去战胜更强的敌人。但是我们需要的是较弱的病毒而不是已死的病毒: 你基本不可能从打败那些已经死掉的东西这件事中学到任何技能。

“死亡意味着在这个包含1200个碱基的区域中肯定存在一个地方, 病毒在那里进化而给出一个特别的信号, 你可以将它理解为序列存活所必需的东西。” 团队里面的一位资深病毒学家说道。由于我们在该点位改变了序列, 导致我们杀死了病毒。“我们必须找到这个信号, 这样才能让病毒重获新生。”

“但是有1200个位置要去找! 我们怎么才能找到啊?” Yutong问道。

我思考了一会。我们必须排错, 这听起来好像和在程序中排错的问题一样。我想起多少个寂寞的夜晚, 我努力想弄清到底是哪一行让程序崩溃。我经常得去委委屈屈地注释掉大段的代码, 然后再次运行以测试它是否依然会崩溃。当我将注释过的区域缩减到足够小的一段后, 问题往往就很容易解决了。而查找这个区域的最好方式就是……

“二分查找!” 我宣布。在图5.1的第Ⅱ组设计中, 现在我们用已死的关键信号缺失毒株所编码的子序列(通常以深色标记)替换掉前半部分原先的活性编码子序列(通常以浅色标记)。如果这个杂交基因序列可以存活, 这意味着关键信号肯定发生在序列的后半部分, 而要是病毒死亡则意味着问题必然出现在前半部分。我们通过在一个长为n的区域上执行二分查找, 只需$\lceil \log n \rceil$轮测序实验便可将关键信号定位到所在区域。

“我们可以只做四轮实验就能把长为n的基因序列中包含信号的区域大小缩减到$n/16$,” 我告诉他们。那位资深病毒学家很激动, 但是Yutong的脸色却发白了。

“再做四轮实验!” 他抱怨起来。“我已经花了整整一个月合成、克隆, 本以为这是最后一次培养病毒了。现在你又让我整个重做, 再等着看结果确定信号究竟在哪半部分, 做完一遍还不够, 然后接着要再重复三遍? 你想都别想! ”

[1] 译者注: 即仍按每次减半的方式进行, 仅与区间大小有关(仍为对数量级), 而与函数特性无关。

图 5.1 设计四种合成基因序列来定位一个特定序列信号: 浅色部分从可存活序列中提取, 而深色部分则从存在致死缺陷的序列中提取。基因序列Ⅱ、Ⅲ和Ⅳ均可存活, 而基因序列I存在缺陷, 这个实验结果只能用位于右起第5个区域的致死信号来解释。

Yutong的这番话说明他已经意识到, 二分查找的力量来自于“信息交互”: 我们在第r轮中的检索方案取决于通过$r-1$在第1轮中的检索结果。二分查找生来就是串行算法, 因此, 如果单次比较操作是一个缓慢而费事的过程时, $\log n$次比较操作突然之间就显得不是很好了。不过, 我还藏着一个很炫酷的小戏法没展示呢。

“连着这样做四轮对你来说确实工作量太大了, Yutong。不过, 你能不能同时做四种不同的设计? 如果我们一次把它们都给你的话。” 我又问他。

“如果我同时对四组不同的序列做同样的工作, 这倒没什么难度,” 他说。“其实不比我只做其中一个麻烦多少。”

问题解决了。我提议让他们同时合成四种病毒设计, 也即图5.1中标为I、Ⅱ、Ⅲ和Ⅳ的序列。事实证明, 只要你能够检索任意子集而不是仅能处理顺序相连的两半, 你就可以并行化二分查找。请注意这四种设计所定义的每一列都是由深色(死)和浅色(活)所组成的独特模式。[1] 因此, 四种合成设计中的这种活/死模式就能在一轮实验唯一地给出其中关键信号的位置。在这个例子中刚好病毒I死亡而其他三个依然存活, 这样就准确地将致死信号定位于从右数第5个区域。

Yutong迎难而上, 付出一个月的辛劳(不是几个月)后发现了脊髓灰质炎病毒中的一个新信号[SLW+12]。他使用分治的观点在一堆错乱之中成功揪出错误, 而分治在每步将问题一分为二时会运行得最快。请注意, 我们每次用四种设计组成一套, 都得由一半深色和一半浅色组成, 这样的安排可使16个区域对应不同的颜色模式。传统的二分查找其实属于交互式操作, 反复处理到最后一次测试在最终所剩的两个区域间选出结果。而通过将测试扩展到可以一次性按对分方式处理完序列, 我们消除了对多次测序的需求, 从而使整个过程加快了很多。

5.3 递推关系

许多分治算法的时间复杂度可以很自然地通过递推关系建模来得到。要想理解什么样的分治算法能高效执行, 很重要的一点就是求出相应的递推关系量级, 此外递推关系还可以为一般情况下的递归算法分析提供重要工具。对于满是数学符号的算法分析有畏难心理的

[1] 译者注: 实为4位二元编码, 也可理解为长为4的二进制数(从0000到1111), 共16个。

那些读者完全可以跳过本节, 但算法设计中有许多很重要的洞见来自对递推关系特性的熟谙。[1]

那么, 什么是递推关系呢? 它是一种根据自身来定义的等式。Fibonacci数是以递推关系$F_n = F_{n-1} + F_{n-2}$来定义, 它将在10.1.1节中讨论。此外还有许多自然数函数(natural function)也可以很容易地表述为递推关系。任意多项式亦可用递推关系表示, 如线性函数:

$$a_n = a_{n-1} + 1, a_1 = 1 \quad \longrightarrow \quad a_n = n$$

任意指数函数也可用递推关系表示, 例如2的幂:

$$a_n = 2a_{n-1}, a_1 = 1 \quad \longrightarrow \quad a_n = 2^{n-1}$$

最后, 有很多古怪的函数不能以常见记号简单地描述, 它们也可用递推关系表示:

$$a_n = na_{n-1}, a_1 = 1 \quad \longrightarrow \quad a_n = n!$$

这意味着递推关系是一种强有力的函数表示方法。

这种自引用(self-reference)特性是递推关系和递归程序/算法所共有的, 而recurrence(递推)和recursive(递归)共用一个词根也说明了它们之间确有共通之处。从本质上看, 递推关系提供了一种分析递归结构(例如递归算法)的手段。

分治递推关系

分治算法往往会将所给问题拆成若干(比如说a个)较小的部分, 每部分规模量为n/b。[2] 分治算法还得再花$f(n)$时间将这些子问题的解组合成一个完整的解。令$T(n)$表示该算法在最坏情况下解决一个规模量为n的问题所用时间。那么$T(n)$便可由以下递推关系给出:

$$T(n) = aT(n/b) + f(n)$$

考虑下列实例, 都是我们所学过的算法:

- **归并排序**——归并排序的时间性能由递推关系$T(n) = 2T(n/2) + O(n)$决定, 因为该算法将数据分成大小相等的两半, 待这两半排完序再花费线性时间将它们归并。事实上, 此递推关系的解为$T(n) = O(n \log n)$, 正好和我们前面所分析的一样。
- **二分查找**——二分查找的时间性能由递推关系$T(n) = T(n/2) + O(1)$决定, 因为我们每步只花费了常数时间将原问题简化成子问题(规模量为原问题的一半)。事实上, 此递推关系的解为$T(n) = O(\log n)$, 正好和我们前面所分析的一样。
- **快速建堆**——`bubble_down`(详见4.3.4节)这种建堆方法可建立一个含有n个元素的堆, 其过程是: 先构建两个含有$n/2$个元素的堆, 随后在对数时间内将它们与根合并。上述过程可浓缩为递推关系$T(n) = 2T(n/2) + O(\log n)$。事实上, 此递推关系的解为$T(n) = O(n)$, 正好和我们前面所分析的一样。

求解递推关系意味着你得找出一个较好的闭形式函数/渐近记号来表述或界定。我们可用5.4节中将要讨论的**主定理**(master theorem)来处理分治算法中常见的递推关系。

[1] 译者注: 若某个递推关系其解的渐近记号较好, 比如为$O(\log \log n)$, 即可用此来设计算法。van Emde Boas树就是一例。

[2] 译者注: 这里的a指分治后真正需要去处理的子问题数, 而不是规模量变更系数b。比如二分查找将问题一分为二, 我们所要处理的子问题数为$a = 1$, 而$b = 2$。

5.4 求解分治递推关系

形如$T(n) = aT(n/b) + f(n)$一般称为分治递推关系, 事实上它很容易求解, 因为其解通常属于三种完全不同的情况之一:

(1)若对于某个常数$\epsilon > 0$有$f(n) = O(n^{\log_b a - \epsilon})$, 则$T(n) = n^{\log_b a}$。

(2)若$f(n) = \Theta(n^{\log_b a})$, 则$T(n) = \Theta(n^{\log_b a} \log n)$。

(3)若对于某个常数$\epsilon > 0$有$f(n) = \Omega(n^{\log_b a + \epsilon})$, 且对于某个常数$c < 1$有$af(n/b) \leqslant cf(n)$, 则$T(n) = \Theta(f(n))$。

尽管这看上去有些吓人, 但它实际上不难运用。问题在于从上述所谓的主定理中找出适用于你所要处理的递推关系的那种情况。情况(1)适用于建堆和矩阵相乘, 而情况(2)适用于归并排序。对于比较笨拙的算法, 通常会出现情况(3), 其中对子问题进行组合的开销压倒[1]了所有的其他开销。

图5.2展示了一个典型的$T(n) = aT(n/b) + f(n)$分治算法所对应的递归树。每个规模量为n的问题被分解成a个规模量为n/b的问题。每个规模量为k的子问题需要$O\big(f(k)\big)$时间来进行它自身的内部处理工作, 即划分和整合[2]这两步中间的那些工作。算法所需总时间是所有这些内部处理开销之和再加上建立递归树的附加费用。该树的高度为$h = \log_b n$, 且叶子结点的个数为$a^h = a^{\log_b n}$, 通过一些代数运算处理, 刚好可将$a^{\log_b n}$化为$n^{\log_b a}$。[3]

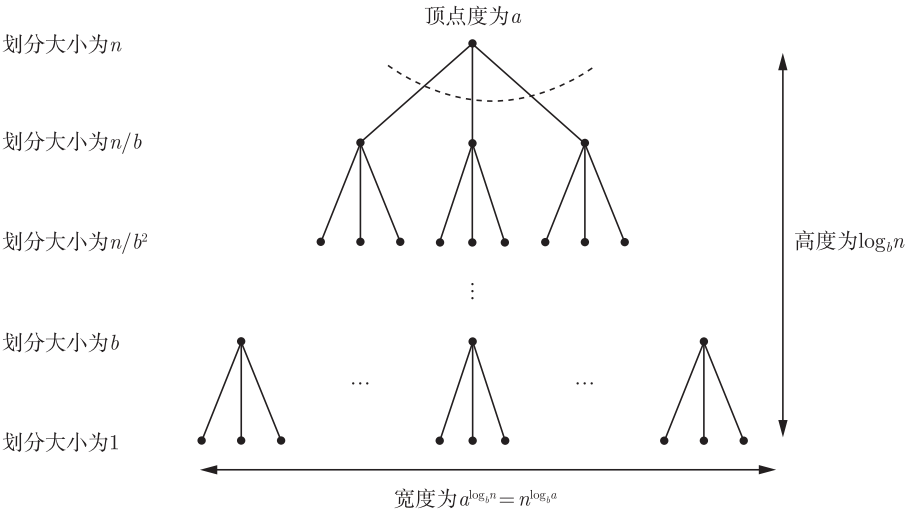

图 5.2 将每个规模量为n的问题分解成a个规模量为n/b的子问题所形成的递归树

主定理的三种情况对应三种不同的开销(它们都是关于a、b和$f(n)$的函数), 而这三种开销都有可能占据上风:

[1] 译者注: 注意$af(n/b) \leqslant cf(n)$, 这表明$f(n)$的增长不算“特别”快。

[2] 译者注: 这里的划分和整合是观念上的, 而在算法上只需常数时间。如归并排序中划分是将元素从中分成两块, 内部处理工作是归并有序表, 而整合则是归并后简单地从算法中返回一个有序表。

[3] 译者注: 附加费用是从$T(n)$变为$aT(n/b)$最后再变为$a^{\log_b n}T(1)$这部分, 可用叶子结点的个数$a^{\log_b n}$描述, 也就是说这部分的时间开销是$O(n^{\log_b a})$, 或可视作将附加费用分摊到叶子结点。而对应规模量为k的子问题的内部结点处的开销跟$f(k)$有关, 因此可称为内部处理工作。算法的总时间即为$n^{\log_b a}$再加上内部处理开销, 只需比较这两者之间的关系即可得到时间复杂度。

- 情况1: 叶子太多——若叶子结点的个数超过内部处理所需开销的总和, 则运行总时间为$O(n^{\log_b a})$。
- 情况2: 每层工作量相同——当我们沿树下移时, 每个问题都会变得更小, 但是会有更多的问题要去解决。若每层内部处理所需开销相同, 则运行总时间即为每层的$n^{\log_b a}$时间乘以层数$\log_b n$, 即运行总时间为$O(n^{\log_b a} \log n)$。
- 情况3: 根结点的处理过于费时——若内部处理所需开销随n增长足够快, 则处理根所需开销就会变成最强的量级。要是这样的话, 运行总时间便为$O(f(n))$。

5.5 快速乘法

你至少知道两种将整数A和B相乘而获得$A \times B$的方法。通常你首先学到的是$A \times B$意味着将B份A相加, 于是便可给出一种对两个n位十进制数相乘的$O(n \cdot 10^n)$时间算法。然后你学到的是对按位表示的较长数字逐项做乘法, 比如:

$$9256 \times 5367 = 9256 \times 7 + 9256 \times 60 + 9256 \times 300 + 9256 \times 5000 = 13\,787\,823$$

不妨仔细观察上式, 我们在每个数位之后所填补的那些零并没有真正作为乘数参与运算, 而我们只是通过将乘积直接移位(移动距离恰为零的个数)从而达到相同的功效。假设我们实际执行两个一位数字的相乘只需常数时间(可通过在乘法表中查找来实现), 那么以上算法可在$O(n^2)$时间内完成对两个n位十进制数的乘法运算。

本节我们将介绍一种更快的大数相乘算法, 它属于较为经典的分治算法。为简单起见, 不妨设乘数均为$n = 2m$位, 我们可以将每个数分成两段, 而这些数段均为m位, 这样一来, 原有数字的乘积便可很容易地基于各个数段的乘积而构建。设$w = 10^{m+1}$, 于是A和B可分别表示为$A = a_0 + a_1 w$且$B = b_0 + b_1 w$, 其中a_i和b_i分别对应每个数字各自的数段。有了这样的分段方法之后, 于是可知:

$$A \times B = (a_0 + a_1 w) \times (b_0 + b_1 w) = a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2$$

通过以上处理过程, 我们将两个n位数的乘法问题化为四个$n/2$位数的乘积。如果不明白, 可以回想一下前面我们所讨论的"乘以w不参与实际计算": 它只是在乘积后面补零而已。有了这四个乘积结果, 我们还必须将它们全部加起来, 当然这只需$O(n)$时间即可完成。

我们将两个n位数相乘所花费的总时间记为$T(n)$, 如果我们在每个较小的乘积上递归使用相同的算法, 那么这种相乘算法的运行时间可由以下递推式给出:

$$T(n) = 4T(n/2) + O(n)$$

使用主定理(情况1)易知该算法能在$O(n^2)$时间内运行完毕, 而它与逐位乘法基本上完全没有分别。这说明我们的方案只能叫分而未治, 根本算不上分而治之。

Karatsuba算法基于另一种分解给出乘法的递推式, 从而让运行时间得到了改善。假设我们先算出以下三个乘积:

$$
\begin{aligned}
q_0 &= a_0 b_0 \\
q_1 &= (a_0 + a_1)(b_0 + b_1) \\
q_2 &= a_1 b_1
\end{aligned}
$$

请注意我们接下来会使用一个巧妙的运算组合:

$$
\begin{aligned}
A \times B &= (a_0 + a_1 w) \times (b_0 + b_1 w) \\
&= a_0 b_0 + a_0 b_1 w + a_1 b_0 w + a_1 b_1 w^2 \\
&= q_0 + (q_1 - q_0 - q_2) w + q_2 w^2
\end{aligned}
$$

于是我们现在只需要三个“半长”(half-length)乘法以及若干加法便能算出原有数字的乘积。w的相关项依然不作为乘数参与计算, 不妨回想前文的讨论: 它们只是零的移位而已。因此Karatsuba算法的时间复杂性由下列递推式掌控:

$$T(n) = 3T(n/2) + O(n)$$

由于$n = O(n \log_2 3)$, 而这属于主定理的第1种情况, 因此$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$。这相比于大数相乘的平方算法是一个很大的改进, 而实际上对于500位左右的数字来说, 其执行速度胜过常规数乘算法不少。

这种使用较少的乘法但会用较多的加法的策略, 同样可为矩阵相乘快速算法助力, 不过其递推式定义略有不同。在2.5.4节中所讨论的矩阵嵌套循环乘法中, 两个$n \times n$矩阵相乘要花费$O(n^3)$时间, 原因是我们要想算出矩阵乘积中n^2个元素, 得逐个位置执行n维向量的点乘运算才能完成。然而, Strassen找到了一种分治算法[Str69], 该算法巧妙地处理7个$n/2 \times n/2$的矩阵积从而获得两个$n \times n$矩阵的乘积, 其运行时间的递推式为

$$T(n) = 7T(n/2) + O(n^2)$$

由于$\log_2 7 = 2.81$, 因此$O(n^{\log_2 7})$强于$O(n^2)$, 而这同样适用于主定理的第1种情况, 易知$T(n) = \Theta(n^{2.81})$。

研究人员通过越来越复杂的各种递推式不断地“改进”[1]Strassen算法, 目前的最好结果可达到$O(n^{2.3727})$, 不妨参阅16.3节以了解更多细节。

5.6 最大子范围与最近点对

假设你接到任务要为一个对冲基金撰写广告文案, 该基金本年度的每月业绩为

$$[-17, 5, 3, -10, 6, 1, 4, -3, 8, 1, -13, 4]$$

虽说今年你亏了钱, 但是从五月到十月这个时段你的净收益共计17个单位, 相比于这一年的其他连续时段而言算是最大收益。这给了你吹嘘的资本, 并以此入手来准备文稿。

[1] 译者注: 其实在算法实用性上没有太多的“改进”。

最大子范围问题的输入为一个含有n个数的数组A, 输出是一对下标i和j, 它们能够最大化$S = A[i] + A[i+1] + \cdots + A[j]$。由于有负数的存在, 将整个数组相加并不一定能最大化目标函数。显式地测试每种可能的区间(也即“开始—结束”端点对)需要$\Omega(n^2)$时间, 我们在这里给出一个可在$O(n \log n)$时间内运行完的分治算法。

假设我们将数组A分为左右两半。最大子范围会在哪里? 它要么位于左半边要么位于右半边, 也可能从中间横跨两边。一个在$A[l]$和$A[r]$之间寻找最大子范围的递归程序可以很容易地调用自身来处理左右两个子问题, 那么我们又该如何找到位于中点m向两边延伸的那种最大子范围(注意必然能覆盖m位置和$m+1$位置)?

解决问题的关键在于, 其实这种居中向两边延伸的最大子范围, 应该是位于左侧且结束于m的最大子范围与位于右侧且开始于$m+1$的最大子范围这两者的并集, 如图5.3所示。左侧的这种最大子范围其和值V_l可在线性时间内通过扫视找到(算法21), 当然右侧的相应和值也可以类似地找到。

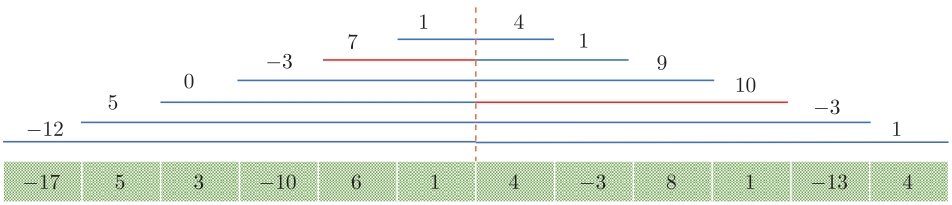

图 5.3 最大子范围之和: 要么整个在中轴左边, 要么整个在中轴右边, 或者(像此图实例这样)基于以中轴为界分别朝左右两边延伸的两个最大子范围之和(分别是7与10)来求和

算法 21 LeftMidMaxRange(A, l, m)

```
1 S ← M ← 0
2 for i from m downto l do
3 |   S ← S + A[i]
4 |   if (S > M) then
5 |   |   M ← S
6 |   end
7 end
8 return M
```

这种分治算法将数组分成两半(对应数组长度n除以2), 再以线性时间整合子问题。我们设该递归算法所花费的时间为$T(n)$, 易知:

$$T(n) = 2T(n/2) + \Theta(n)$$

根据主定理的情况2可推出$T(n) = \Theta(n \log n)$。

“分别在两侧找到最优再检查横跨中轴的情况”这个一般化方案也可适用于其他问题, 下面来考虑在一个由n个点构成的集合中寻找点对之间的最小距离的问题。

这个问题在一维中很简单, 4.1节中我们已经见过: 将点进行排序之后, 最近点对必然是相邻的; 排序后从左到右以线性时间扫视, 便可获得一个$\Theta(n \log n)$算法。但是我们可以用