解决空间规模问题的几种常用的存储结构

广西柳州铁路第一中学 龙翀

【关键字】空间规模、存储结构

【摘要】空间规模型问题是近年来国际国内比赛的热点。这类问题对选手们掌握运用各种数据存储结构的能力提出了更高的要求，本文站在解决空间规模型问题的角度，深入分析了几种常用的数据存储结构，在这一类特殊问题当中表现出来的一些新的特点，总结了链式结构中的单链表、双链表和树型结构，矩阵结构各自的长处和短处，特别是通过一题多解比较说明了矩阵结构具有很大的潜力。论文对三道国际国内竞赛题作了详细的分析总结，这对于参加国际或国内比赛的选手来说具有很强的现实意义。

正文

一、引论

“规模”一词在《现代汉语词典》里的解释为“某一事物包括的范围”，而在计算机程序设计中，我们可以把它理解为程序运行时在空间和时间等方面的开销。它主要包括两个方面：空间规模和时间规模。它们都是在设计算法、编制程序时经常要考虑到的问题。存储结构，就是数据元素和元素之间的关系在计算机中的表示，它是为解决空间规模问题，或是通过解决空间规模问题间接地解决时间规模问题而总结的一些存储方法。

近年来，在信息学竞赛中出现了一类新型问题，这些问题的共同特点就是输入数据非常之大，可达到 1M 甚至几 M，我们不妨称这类问题为空间规模型问题。众所周知，用 Turbo Pascal 编程时仅使用最大为 640K 的常规内存，而仅凭这点有限的内存完全不可能把它们一一都存下来，这给我们对数据存储结构和以及设计算法提出了更高的要求。(见注 1)

程序是算法和数据结构的有机结合，而本文着重从数据结构的角度出发，针对数据规模型问题，重新研究一下几种常用的存储结构的运用情况和它们体现出来的新的特点。

二、几种常用的数据结构

通过对国际国内比赛中关于这类问题的存储结构模式进行仔细的研究，我发现解决这类问题常用的存储结构有三类：链式结构、树型结构和矩阵结构。下面我就这三种结构一一来分析它们在这类问题中的应用情况，并比较它们各自的优点和缺点。

1． 链式结构

在链式结构中常用的两种是单链表和双链表，我们先从一个简单的例子说起。

【问题 1】最佳游览路线(NOI'97 第二试第一题)

某游览区街道成网络状，东西向的街道是旅游街，只能由西向东走，并有一定的分值，南北向的街道是林荫道，既可从南向北走，也可从北向南走，没有分值，要求从某一路口开始游览，并在另一路口结束游览，使所走过的街道分值总和最大。其中 1≤旅游街道数目≤100，1≤林荫道数目≤20001，-100≤分值≤100。

【分析】初看起来，此题规模非常可怕(100*20001=2000100)，我们不可能存下如此庞大的规模。但细想起来，由于只能由西向东走，所以说每一纵行至多只能通过一次，而对于同一纵行的旅游街，我们可以通过林荫道自由到达任意一条旅游街，为了达到题目最佳的要求，我们只需走分值最大的街道就可以了，因此在存储时只需记录每一纵行中最大的分值即可，这样存储结构由二维变成一维，所用空间为 20001 个 shortint。再进一步分析，可以发现：若 A->B 是一条最佳浏览线路，C 是 A->B 中间一点，那么在子线路 A->C 上的分值一定非负，否则 C->B 的分值一定大于 A->B 的分值。

定义 Pi 为以 i 结尾的最优路径分值的总和，Ai 表示第 I 纵行的最大分值，我们可以得到一个状态转移方程：

$$Pi = \begin{cases} P_{i-1} + A_i (P_{i-1} + A_i \text{非负数}) \\ \text{¢} \quad P_{i-1} + A_i \text{非负数}) \end{cases}$$

边界条件：P0 = 0

目标：求 Max{Pi}

通过转化，我们就可以在读文件的同时将二维数据转化为一维，然后先从 P1 算起，每算一次 Pi 只需作一次判断，计算完 Pi 后又立即转入 P_{i+1} 的计算，直至处理完所有元素为止（见程序 1）。我们可以把它的结构表示：

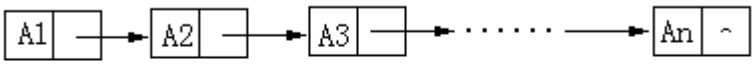


图1-1

这就是我们熟知的单链表，它在问题中表现为：对于处理当前的元素，无需回头查找在此之前的元素（因为前边的元素对它影响有限，可直接用几个变量表示出来）。对于这种表现，可以总结出单链表结构对于解决空间规模数据问题的特点：

1 效率高，用此结构编出来的程序正常情况下一般不会存在时间问题。道理很简单，因为对于每一个元素的处理只进行极有限的几次运算，复杂度较低（见注 2）；

2 由于单链表结构过于简单，在具体算法设计中对元素之间逻辑关系反映

不够（只有在相邻元素之间有一条单向边），所以说单链表结构在解题时有很大的局限性，对于难度较大的题目则很难用它来实现。但是作为如此高效的结构它仍是我们设计程序时努力的一个方向。

链式结构中的双向链表也是在程序设计中应用频繁的数据结构，它可以表示为：

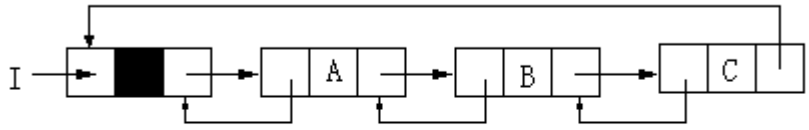


图1-2

其中每个单元除了存放元素外，还有一个前趋和后继指针，这在一定程序上解决了单链表表示元素之间的关系过于简单的缺陷，但使用双链表时应当注意尽量控制向前查找的深度，因为此类问题输入数据中包含的元素十分庞大，处理不当会大大增加程序时间规模。因此双向链表也有一定的局限性，为了进一步把它的特点分析透彻，我们将在下文继续论述。

2． 树型结构

树是我们熟悉的数据结构，其中最有代表性的是二叉树，这是一种特殊的非线性结构，在程序设计中有着广泛的应用。

【问题 2】Contact（IOI '98 第一试第一题）

从由 01 串构成的文件中，找出长度介于 A 和 B 之间出现次数最多的 N 个不同频率的子串，子串可以相互覆盖，输出结果必须按子串出现频率的降序排列，频率相同的子串按长度降序排列，频率和长度均相同的子串则按其对应数值降序排列。其中 $0 < A \le B \le 12$，$0 < N \le 20$，输入文件可达到 2MB。

【分析】这道题要求完成以下两步操作：

1 找出所有满足条件的子串，并统计各子串出现的频率；

2 把所有不同子串按要求排序输出。

其中第①步是关键。让我们先从具体实例开始研究，长度为 1 的串只有两个“0”和“1”。长度为 2 的串有 4 个：“00”、“01”、“10”、“11”，它们可以看成是在长度为 1 的子串后添加“0”或“1”得到。同样，长度为 3 的串又可以看成是在长度为 2 的子串后添加上“0”或“1”得到，例如“101”是在“10”之后添加“1”得到，以后也可依此类推。这样层层递进的关系使我们想到了树。以下是一棵二叉树，最上的为根结点，定义每个结点的左枝为 0 右枝为 1，这样一个子串可以与这棵二叉树中的一条路径一一对应，如图 2-1 所示：

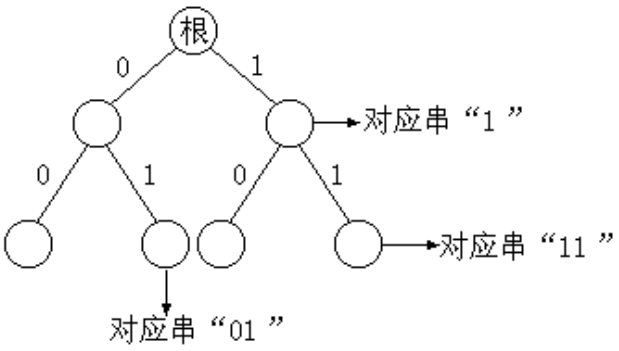


图2-1

树中的每个点赋予一定的权值，表示该点对应的子中在文件中出现的频率，那么我们可以得到一些累加规律，例如当 A=1，B=3 时，某个中间状态对应的二叉树为：

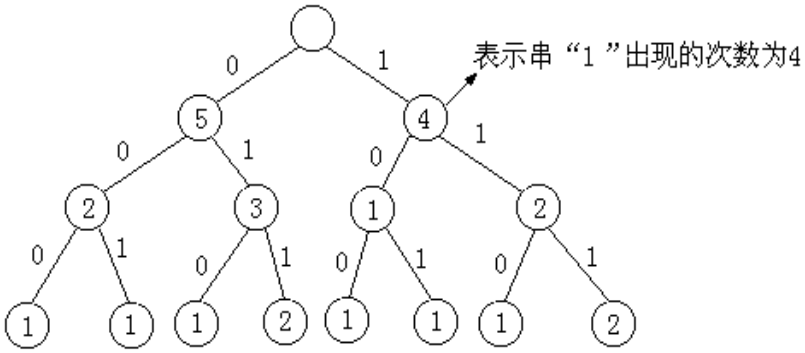


图2-2

假设现在读到一个串为“011”，其中以“0”开头且长度为 1 到 3 的子串有三个：“0”、“01”、“011”，统计时应将这三个子串的频率加 1，从图上可以直观地看到，这个操作相当于在对应的 01 路径上将各结点的频率加 1，如下图所示：

鉴于对应二叉树的结点很少（最大为 2^13-1 = 8191），完全可以多次遍历，

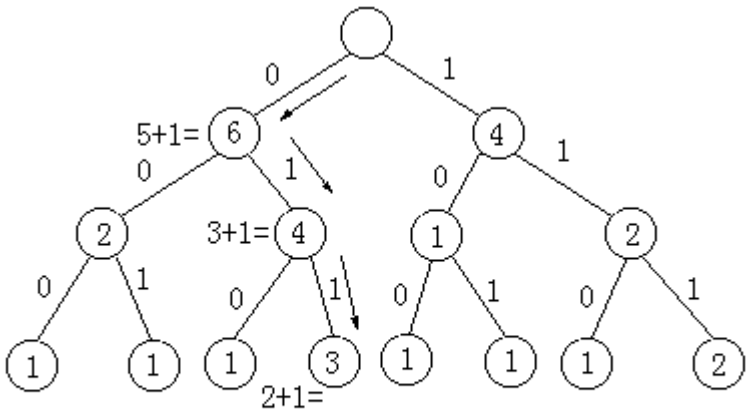


图2-3

不难从中找出前 N 个频率最大的子串，然后按从大到小的顺序输出。（见程序 2）

由这个问题我们可以看出，比起单链表来，树型结构中每一结点与其它结点的联系是一对多的，这样更有利于我们处理好数据。特别地它在处理空间规模问题表现出来的特点是：

① 规律性强。如果新代结点与子代结点之间的关系建立得好，那么就可以把庞大的元素安排得井井有条，就可以按照一定的顺序逐层深入，解决当前元素的处理。

② 可操作性强。我们对树的各种操作方法都较为熟练，在解题当中，只要我们根据题目特点，建立好树的各种关系，其它的操作（如查找、打印）就迎刃而解了。

3、矩阵结构

我们一般用一个二维数组来表示矩阵结构，它有 x、y 两个方向，我们在实际操作中常用的表示方法是：x 轴表示数据的元素，y 轴表示元素各种状态条件。

数组内具体的值表示元素在当前状态条件下的表现，一般用数值表示。如图所示

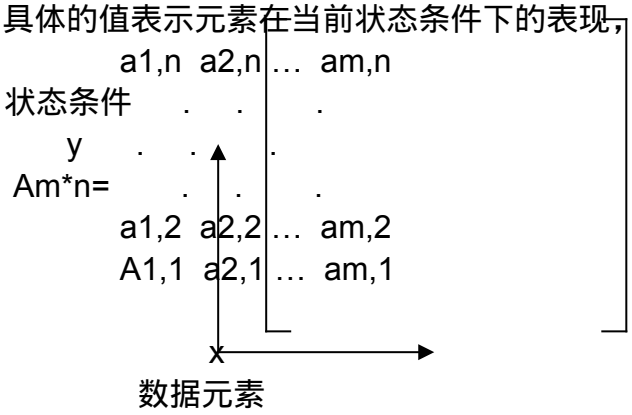


图 3-1

我们可以按照这种思想对问题 2 进行进一小的研究：

【问题 2 续】Contact（IOI'98 第一试第一题）

【分析 2 续】我们对此题继续分析还可以发现，每一个仅由 0 和 1 组成的子串也可以当作二进制数转化为十进制数来表示。然而每个十进制数与 01 串之间并不是一一对应的关系，如"01"、"010"和"0010"它们对应的十进制数都是 2，这是为什么呢？原来，在二进制中，10、010 和 0010 表示的是同一个数，但字串"10"、"010""0010"就不是同一字串了，因为这时"0"不是代表没有而是一个字符，它们之间存在着长度的差别，为了把长度不同而转化为十进制数却相同的字串区分开来，又设定了"长度"座标，这样，一维数组变成了二维短阵，简单表示为下图：

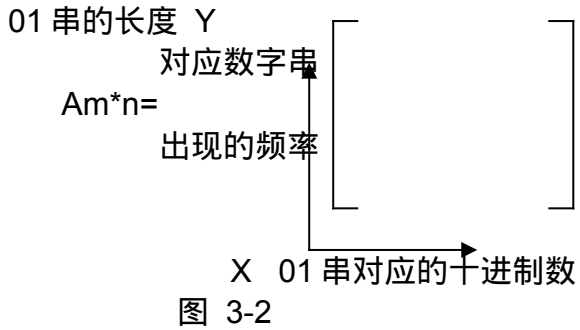


图 3-2

从表中我们可以查找出频率最大的前 N 个串，然后按要求输出（见程序 3）

相比较起来，由于程序 3 用数组操作，程序 2 要多次递归，后者比前者简洁，速度也快一些，请看下面列表分析(前 2 个为国际比赛中较大的 2 个，后两个自编，略大于 2MB)：(见注 3)

数据 程序	数据 1	数据 2	数据 3	数据 4
程序 2	1．4s	2．3s	5．8s	6．0s
程序 3	0．5s	1．2s	3．9s	3．5s

由于在矩阵结构中结合了数学的二进制计算，使程序效率进一步提高。是不是矩阵结构有着很大的潜力呢？其实，矩阵结构还能结合其它很多数学模型，

其中还包括动态规划，我们常常在使用它解决一些难题时收到了意象不到的效果。现在我们不妨再分析一个问题。

【问题 3】隐藏代码问题 codes（I0I'99 试题）

题目详见 IOI'99 试题。

【分析】本题主要解决两个重点

1 找最小右隐藏序列；

2 找到最小右隐藏序列后， 找出包含代码字最大长度和的互相不覆盖的隐藏代码。

后一个问题较为简单。定义序 Pj 为从文本开头到第 i 个字母止互相不覆盖的隐藏代码包含的代码长度和的最大值。

在没有找到以 j 结尾的隐藏代码时，Pj=0；

若当前找到一个以 i 开头，j 结尾的且匹配代码字长度为 L 的隐藏代码，即可列出如下方程：

Pi=max{Pj,Pk+L}1≤k<i P0=0

目标： max{Pj} 1≤j≤文本长度

可是，题目给出文本长度不是一百万的吗？其实，最小右隐藏序列不超过 10000， 所以不同的个数至多也只有一万个，所以我们在实际编程序时，不必以 j 作为下标就行了。

现在，我们集中精力解决最小右隐藏序列问题，实际上，我们还得找出最小右隐藏序列，例如，有两个码字 TUN、NIR 和一个文本 TUNNIR，其实 NNIR 也是最小右隐藏序列（根据定义）然而我们若按它计算，只能找到 TUN 或 NNIR，所以说， 我们在设计算法时必须要按最短的（也就是 NIR）计算。

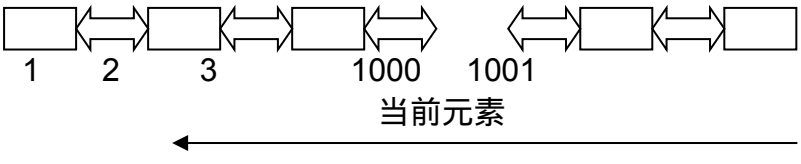
继续往下思考，我想出了两种方法，它们使用的是不同的存储结构，现供大家比较分析方法 A：我们在读到一个字符时，可以开利用一个数组 Pi，它表示是第 i 种代码字到当前字符时匹配到的最大长度（指还没有匹配完）例如：文本代码字为 ALL， AAALAL，我们可以如下实现

```
      AAALAL
P→111223
```

这样到最后一个字符时就有了一个隐藏代码了。但是我们还不能确定它是不是最短最小左隐藏序列，不能确定它的长度是否在 1000 以内，所以必须回归，也就是说从最后一个字符找起，按次序把从尾到头每一个字符找出来，直到找到第一个字符为止，（最多只回查 999 个字符，若还没有找完，就说明隐藏代码超过 1000）

```
AAALAL
  1233←
```

这样我们找到的就是最短最小右隐藏序列。经过这样的分析，我们可以知道方法 A 采用的是双向链表的存储结构，如下图：



最大回扫路径

图 3-3

但在此同时，我们也应注意到，在最坏的情况下，回扫深度为 1000。我们在前边已经分析过进行双向链表回扫时深度不能太大，然而这道题有它特殊的一面；最小右隐藏序列不超过 10000 个，所以说回归的次数不会很多，算法理论上来说是可行的，但是从精益求精的角度来说，我们还应继续探求更适合本题的一种结构。

方法 B：方法 A 的双链结构虽然还不是很理想的，但它给了我们不少启示。最重要的，它注意到了应该把目前匹配到的位置记下来，从而我们的查找就有一定的方向，这是必须予以肯定的。然而方法 A 却忽视了要记录隐藏序列的起始位置。为了找到当前隐藏序列的起始位置，进行了盲目的回扫，极大地浪费时间。我们自然地就想：能不能在匹配当中就把它的起点记下来呢？这也许行得通那么，记下了起始位置的同时，还要不要像方法 A 一样还把当前匹配到的代码字位置记下来？我们注意到了在介绍方法 A 时用的例子 AAALA，若在后边再加一个 L，变成 AAALALL，那么这出文本就有了两个最短最小右隐藏代码 AAALALL，其中第二个 L 就有双重身份：在 ALAL 中代表代码字的第三个字符，在 ALL 中代表代码字的第二个字符，这样，同一个文本中的同一个字符由对对应的隐藏代码起始位置不同，在同一个代码字中就有了不同的对应状态，这也是方法 A 没有处理好的。那么，该怎样处理了，在矩阵结构的分析当中我们曾经介绍过，矩阵中的 Y 轴方向一般是用来表示元素在不同的状态下的，我们也不妨这样定义：

X 轴为各代码字编号，Y 轴为各代码匹配到的位置（即第几个字母），Px,y 表示离当前读到的示符最近的会有第 1 种代码字前 1 个字母的最短最小隐藏方列在整个文述中的起始位置 。注意：现在的 X 轴已经依题意有所变动，表示的不再是当前读到的字符了。同时，我们也清楚,此时矩阵中同一纵行元素之间关系密切，而同一横行的元素之间没多大联系，我们就可以这样表示：

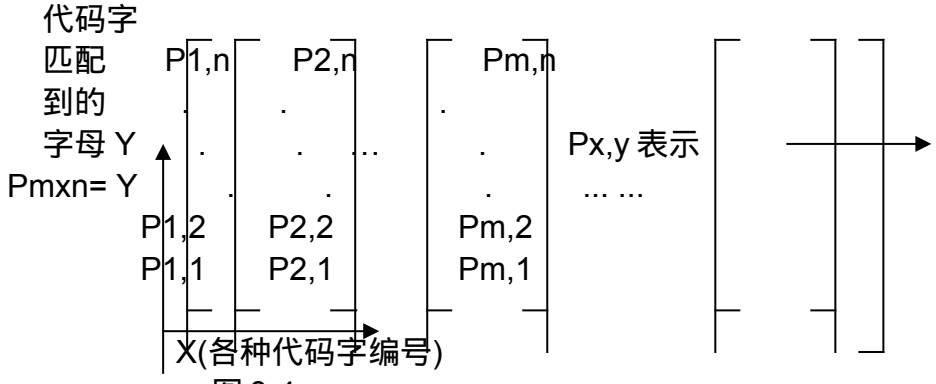


图 3-4

有了存储结构，我们就可以试着推导一下算法。

定义：$P_{i,j}$表示离当前读到的字符最近的含有第 i 种代码字前 j 个字母的最短最小右隐藏序列在整个文本中的起始位置，当还未找到时，$P_{i,j}=0$。其中 1<i<=N，1<=j<=Length(i)，Length(i)表示第 i 个代码字的长度。

根据定义，第 i 个代码字的 $P_{i,j}$ 的值可以前 j-1 个的值(即 $P_{i,j-1}$)决定，而条件是文本中读过的字母已经与该代码字的前 j-1 个字母匹配，同时当前读取的字母又与代码字的第 j 个字母相等，这样可以对 $P_{i,j}$ 的值进行修改。

假设：$A_{i,j}$ 表示第 i 种代码字的第 j 个字母(1<=i<=N,1<=j<=Length(i))。NowChar 表示当前文本中读到的字母，NowNum 表示当前读到的文本字母的序号。

若满足以下关系：

1 $A_{i,j}$=NowChar(即当前读取的字母又与第 i 个代码字的第 j 个字母相等)且

2 j=1(即是代码字的第一个字母)或

3 j>1(即不是代码字的第一个字母)且 $P_{i,j-1}$>0(即文本中读过的字母已经与该代码字的前 j-1 个字母匹配)且 NowNum+ $P_{i,j-1}$+1<=1000(即隐藏序列长度不超过 1000)。

则有下式成立：

1 当 j=1 时，$P_{i,j}$=NowNum；

2 当 j>1 时，$P_{i,j}$= $P_{i,j-1}$。

当 $P_{i,Length(i)}$<>0，即当 j 推到 Length(i)时，就找到了一个合乎要求的“最短最小右隐藏序列”，其编号为 i，起始位置为 $P_{i,Length(i)}$，终点位置为 NowNum。

同时，当我们从文本中读到一个字符时，要知道它在各个代码字中的位置，可以在所有的代码字中一个一个的查找，但这样效率不高，考虑到所有代码字的字符总数最多只有 100*100 个，因此可以按照字母的各类，把含有该字母的代码字的编号记录下来，这样查找可大大提高效率。

在推算过程中我们发现，在任一码匹配中，总是先从第一个开始匹配，第一个匹配到 了，再考虑第二个... 这样，我们容易看出，在每一纵行之间的关系实际上是单链表结构

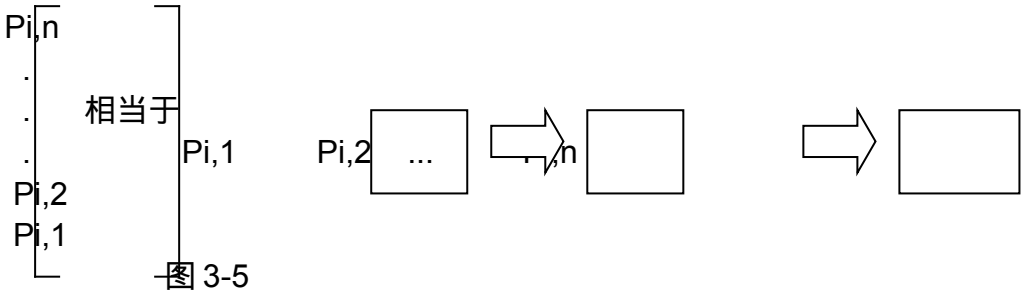


图 3-5

我们在前边还说过，使用单链表的效率一般是很高的,方法 B 应用了单链表组成的矩阵,推导出了动态规划递推公式，免去回扫的复杂度，值得我们再编一次程序（见程序 4）。运行结果也在我们的意料之中，方法 B 远优于方法 A。以下是种方法对应测试数据的运行时间(前 3 个为国际比赛中最后 3 个数据)。

	数据 1	数据 2	数据 3	数据 4
方法 A	3．4 s	2．8 s	4．0 s	9．9 s
方法 B	0．5 s	0．3 s	1．1 s	5．1 s

根据上面两个例子，我们对矩阵结构有了较深刻的理解。那么，为什么使用矩阵结构解决空间规模问题有着很大的潜力呢？我们在题目的分析中已经看出了不少，现在把它作一下归纳总结：

1 矩阵结构发扬了链式结构简单明了地表示元素状态的优点。它使用数组操作，而一般的数组操作速度与链表一样是很快的。

2 矩阵结构也很好地反映了树型结构清晰地反映元素之间的联系的思想，同时具有一些树型结构没有的优点。A、（X、Y）坐标的查找远比树的遍历方便 B、X、Y 轴的意义可以更好地结合数学方法定义。这样使用矩阵结构就具有更广

阔的思维空间。

3 矩阵结构能够通过综合的方法把链表等其它结构嵌套在内部，从而加强它解题的灵活性，而这一点是非常重要的。

所以说，我们在设计存储结构、算法时，矩型结构是不能忽视的，不过也不能说矩阵结构是万能的，不能忽视其它结构在解题当中的运用。

三、小结

要用一个小小的程序解决几兆的数据处理问题，确实不易，而要编程序解决国际国内这些带有隐蔽性条件约大数据处理问题，那就更不容易了。不过，通过上面的分析我们可以发现，这类问题还是有一定规律性的，我们应该在平时做到，对关于数据结构的各种算法精益求精，不能光拘泥于一般的处理算法，更要研究每种存储结构在某些特殊问题中的特殊处理方法，并且要多多总结每种结构在不同情况下表现出来的优缺点，培养自己判断分析的能力和权衡得失的能力。这样，我们的思路才会更加开阔，编程的效果才会更好。

最后，我们再把这几种解决空间规模问题常用的存储 结构的特点总结一下（均指在一般情况下）

结构类别		所需空间	操作速度	体现元素联系	应用范围	在空间规模型问题中对 应的处理手段
链式结构	单链表	少	快	少	窄	直接一步递推，不需回扫
	双向链表	少	较慢	多	较广	回头查看与当前读到元素有关信息
树型结构		多	一般	多	较广	把元素之间 联系具体化， 一般用 递归手段
矩阵结构		多	较快	很多	广	递推（包括动态规划）必要时进行一定的回查

附录

注 1、因为 N 值很大， 任何在处理一个个数据元素时的延时手段都能产生积累效应，而使程序异常地慢，所以我们应对平时一段的数据操作算法精益求精。

注 2、这里不用常用的 O(N^k)作算法复杂度的衡量，因为数据规模（即 N 值）很大，O(N ^ 2)的时间复杂度显然就不能承受了，更重要的，即使是 O(k*N)当 k 值很大时，程序也不能在规定时间内通过。

注 3、所有运行指标在 PIII450 上得到。

程序分析

以下 4 个程序的输入、输出文件名及格式均是按照原试题的要求设置的

程序 1：

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}

var m{旅行街},n{林荫道}:integer;
   a{当前游览分值(用于递推)},b{最大游览分值}:longint;
   t:array[1..20001] of integer;{存放各纵行最大值}
```

```
procedure input;{输入数据，存下各纵行最大值}
 var f:text;
     i,j,k:integer;
 begin
  assign(f,'input.txt'); reset(f);
  read(f,m,n); dec(n);
  for i:=1 to n do t[i]:=-maxint;{设置最小值，开始寻找}
  for i:=1 to m do
   for j:=1 to n do
    begin
     read(f,k);
     if k>t[j] then t[j]:=k;{各纵行只保留最大值}
    end;
  close(f);
 end;

procedure main;{递推求出结果}
 var i,j:integer;
 begin
  a:=0; b:=0;
  for i:=1 to n do
   begin
    inc(a,longint(t[i]));{累加 a}
    if a<0 then a:=0;{若 a 为负，舍去以前所有状态，清零}
    if a>b then b:=a;{若 a 优于 b,用 a 的值作为当前 b 的最优值}
   end;
 end;

procedure print;{打印结果}
 var f:text;
 begin
  assign(f,'output.txt');
  rewrite(f);
  writeln(f,b);
  close(f);
 end;

begin
 input;{输入数据，存下各纵行最大值}
 main;{递推求出结果}
 print;{打印结果}
end.
```

程序 2：

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}

const inputfile = 'contact.in';
      outfile   = 'contact.out';
      tc:array[1..13] of integer=(1,2,4,8,16,32,64,128,256,512,1024,2048,4096);
      {对应的二进制数分别为 1、10、100、1000.....便于取位查找}
      tz:array['0'..'1'] of byte=(0,1);{字符转数值}

type r=^rr;
     rr=record
         zero{当前字符是 0 时对应的轨迹方向},one{当前字符是 1 时对应的轨迹方向}:r;
         num{当前轨迹对应子串出现的频率数}:longint;
        end;
ar=string[12];{子串的具体方案}

var t:array[1..21] of longint;{最大的前 20 个频率数(为了排列方便设置第 21 个)}
    tch:array[1..1024] of char;{缓存设置}
    tree:r;{状态树的根结点}
    inf,outf:text;
    a,b,n{题目规定的输入变量},long{当前处理的有效子串长度},now{排列总数时记录当前已
有的不同的频率数}
    ,which{当前要查找输出的频率数所在位置},where{在第几层中查找}:byte;
    yu{截取有效长度的标准},m{有效子串的表达}:integer;

procedure maketree(floor:byte;{当前层数(根结点为第 0 层)} var p:r{当前指针});{生成空树}
 begin
  p^.num:=0;{总数初始值为零}

  if floor=b{已生成到叶子结点} then
   begin
    p^.zero:=nil;
    p^.one:=nil;{左右指针均为空}
   end
  else
   begin
    new(p^.zero);{生成它的孩子结点}
    maketree(floor+1,p^.zero);
    new(p^.one);
    maketree(floor+1,p^.one);
   end;
 end;

procedure init;{程序初始化}
```

```
 begin
  readln(inf,a,b,n);{输入 a、b、n 值，便于建树}
  new(tree);{设置树的根结点}
  maketree(0,tree);{生成空树}
 end;

procedure add(floor:byte;{层数} var p:r{当前指针所在处});{逐层累加}
 begin
  inc(p^.num);{总数累加}
  if floor<long then{判断是否已走到头}
   if ((m and tc[long-floor])=0){判断当前位置对应的值，决定下一步的指针轨迹}
    then add(floor+1,p^.zero)
     else add(floor+1,p^.one);
 end;

procedure main;{读入数据，累加子串出现的数目}
 var ch:char;{当前读到的字符}
     i:byte;{循环变量}
 begin
  long:=0;{尚未读到数据，有效长度为零}
  m:=0;{有效串初始值为空}
  yu:=tc[b+1];{例如，有效长度为 2，有效子串就是除以 4(二进制为 100)的余数所对应的二
进制串}
  repeat
   read(inf,ch);

   if ch='2' then{文件已读完}
    begin
     for i:=long downto 1 do{处理剩下的长度不足 b 的子串}
      begin
       add(0,tree);
       dec(long);
      end;
     exit;
    end;

   m:=(m*2+tz[ch]) mod yu;{生成当前有效子串}
   inc(long);
   if long=b then{若长度为 b 则进行逐层累加}
    begin
     add(0,tree);
     dec(long);{控制有效长度}
    end;
  until false;
```

```
 end;

procedure find(floor:byte; p:r);{排列出最大的 20 个频率数}
 var i,j:byte;{循环变量}
 begin
  if floor>=a{长度在要求范围之内} then
   begin
   {插入排序法}
    j:=now;
    while (j>0) and (t[j]<p^.num) do dec(j);

    if j<n then
     begin
      if (j=0) or (t[j]>p^.num) then
       begin
        if now<n then inc(now);
        for i:=now-1 downto j+1 do t[i+1]:=t[i];
        t[j+1]:=p^.num;
       end;

      if floor<b then{继续寻找}
       begin
        find(floor+1,p^.zero);
        find(floor+1,p^.one);
       end;
     end;
   end
  else
   begin{继续寻找}
    find(floor+1,p^.zero);
    find(floor+1,p^.one);
   end;
 end;

procedure look_up(floor:byte;{层数} p:r;{指针} ss:ar{当前子串状态});{按顺序输出符合要求
的子串}
 begin
  if (floor=where) and (p^.num=t[which]) then{满足层数、频率要求的就打印}
   write(outf,' ',ss);

  if (floor<where) and (p^.num>=t[which]) then{继续往下寻找}
   begin
    {相同频率、相同长度的子串按数值从大到小输出}
    look_up(floor+1,p^.one,ss+'1');
```

```
    look_up(floor+1,p^.zero,ss+'0');
   end;
 end;

procedure print;{输出具体方案}
 begin
  now:=0;
  fillchar(t,sizeof(t),0);
  find(0,tree);{排列出最大的 20 个频率数}

  assign(outf,outfile);
  rewrite(outf);
  settextbuf(outf,tch);
  for which:=1 to n do{按频率数从大到小的顺序输出}
   begin
    write(outf,t[which]);
    for where:=b downto a do{相同频率数的子串按长度从大到小输出}
     begin
      look_up(1,tree^.one,'1');
      look_up(1,tree^.zero,'0');
     end;
    writeln(outf);
   end;
  close(outf);
 end;

begin
 assign(inf,inputfile);
 reset(inf);
 settextbuf(inf,tch);
 init;{程序初始化}
 main;{读入数据，累加子串出现的数目}
 close(inf);
 print;{输出具体方案}
end.
```

程序 3：

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}
const inputfile = 'contact.in';
      outfile   = 'contact.out';

      tc :array[1..13] of integer=(1,2,4,8,16,32,64,128,256,512,1024,2048,4096);
      {对应的二进制数分别为 1、10、100、1000.....便于取位查找}
```

```
    put:array[1..12] of integer=(1,3,7,15,31,63,127,255,511,1023,2047,4095);
    {对应的二进制数分别为 1、11、111、1111.....便于取字串的后 1、2、3.....位}
    tz:array['0'..'1'] of byte=(0,1);{字符转数值}

type ar=array[0..4095] of longint;

var t  :array[1..21] of longint;{最大的 20 个频率数(为了排列方便设置第 21 个)}
   tk :array[1..12] of ^ar;{保存各种子串出现次数}
   tch:array[1..1024] of char;{缓存设置}
   inf,outf:text;
   a,b,n{题目规定的输入变量},long{当前处理的有效子串长度}
   ,kind{排列时记录当前已有的不同的频率数}:byte;
   yu:integer;{截取有效长度的标准}

procedure init;{程序初始化}
 var i:byte;{循环变量}
 begin
  readln(inf,a,b,n);{输入 a、b、n 值，便于初始化}
  for i:=1 to b do{初始清零}
   begin
    new(tk[i]);
    fillchar(tk[i]^,sizeof(tk[i]^),0);
   end;
 end;

procedure main;{读入数据，累加子串出现的数目}
 var i:byte;{循环变量}
     ch:char;{当前读到的字符}
     m:integer;{有效子串的表达}
 begin
  m:=0;{有效串初始值为空} long:=0;{尚未读到数据，有效长度为零}
  yu:=tc[b+1];{例如，有效长度为 2，有效子串就是除以 4(二进制为 100)的余数所对应的二
进制串}
  repeat
   read(inf,ch);
   if ch='2' then exit;{文件已读完，退出}
   m:=(m*2+tz[ch]) mod yu;{生成当前有效子串}
   if long<b then inc(long);{计算当前有效子串长度}

   for i:=1 to long do inc(tk[i]^[m and put[i]]);{累加以当前读到的字符结尾的有效子串长度}
  until false;
 end;

procedure sort;{排列出最大的 20 个频率数}
```

```
 var i,j，k，ii:integer;{循环变量}
 begin
  fillchar(t,sizeof(t),0);
  kind:=0;
  {将所有可能的子串一一寻找}
  for i:=a to b do
   for j:=0 to put[i] do
    begin
     k:=kind;
     {插入排序法}
     while (k>0) and (tk[i]^[j]>t[k]) do
      dec(k);

     if (k<n) and ((k=0) or (tk[i]^[j]<>t[k])) then
      begin
       if kind<n then inc(kind);

       for ii:=kind downto k+1 do
        t[ii+1]:=t[ii];
       t[k+1]:=tk[i]^[j];
      end;
    end;
 end;

procedure print;{按要求输出结果}
 var ii,i,j,l{循环变量},p{生成具体方案的过渡变量}:integer;
 begin
  assign(outf,outfile);
  rewrite(outf);
  settextbuf(outf,tch);
  for ii:=1 to n do{按频率数从大到小的顺序输出}
   begin
    write(outf,t[ii]);
    for i:=b downto a do{相同频率的子串按长度从大到小输出}
     for j:=put[i] downto 0 do{相同长度的子串按数值从大到小输出}
      if tk[i]^[j]=t[ii] then
       begin
        write(outf,' ');
        p:=j;
        for l:=1 to i do{打印具体子串}
         begin
          write(outf,p div tc[i+1-l]);
          p:=p mod tc[i+1-l];
         end;
```

```
      end;
    writeln(outf);
   end;
  close(outf);
 end;

begin
 assign(inf,inputfile);
 reset(inf);
 settextbuf(inf,tch);
 init;{程序初始化}
 main;{读入数据，累加子串出现的数目}
 close(inf);
 sort;{排列出最大的 20 个频率数}
 print;{按要求输出结果}
end.
```

程序 4：

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65520,0,655360}

const inputfile = 'words.inp';
      textfile  = 'text.inp';
      outfile   = 'codes.out';

type rrr={动态规划存储数据}
     record
      num:byte;{代码字编号}
      up{前一代码字}:integer;
      tot{当前最大代码字长度和},start,finish{起止位置}:longint;
     end;
    ar=array[1..100] of longint;{存放当前的代码字与文本的匹配情况}
    aaa=^aar;
    aar=record{字符在各代码字中的出现情况}
         num{代码字编号},ci{此字符在代码字中的出现位置}:byte;
         next:aaa;
        end;

var inf,outf:text;
    nn{隐藏代码数目},n{代码字数目},findwz{最大值出现的最末尾位置}:integer;
    m{当前读到的文本位置},findmax{最大代码字长度和}:longint;
    tn:array[1..10000] of ^rrr;
    start:array[1..100] of ^ar;
    tl:array[1..100] of byte;{各代码字长度}
```

```
   tv:array[1..1024] of char;
   t:array['A'..'z'] of aaa;
   pp:aaa;

procedure put_in(cch:char;{具体字符} nnum{代码字编号},nci:byte{出现位置});{记下字符所在位置}
 begin
  new(pp);
  pp^.next:=t[cch]^.next;
  t[cch]^.next:=pp;

  with pp^ do
   begin
    num:=nnum;
    ci:=nci;
   end;
 end;

procedure init;{读代码字文件，初始化}
 var i,j:byte;{循环变量}
     ii:char;{循环变量}
     s:string;{过渡变量}
 begin
  assign(inf,inputfile);
  reset(inf);
  settextbuf(inf,tv);
  readln(inf,n);
  for ii:='A' to 'z' do{t 数组初始化}
   if ii in ['A'..'Z','a'..'z'] then
    begin
     new(t[ii]);
     t[ii]^.next:=nil;
    end;

  for i:=1 to n do{初始化、存放数据}
   begin
    new(start[i]);
    fillchar(start[i]^,sizeof(start[i]^),0);
    readln(inf,s);
    tl[i]:=ord(s[0]);
    for j:=1 to tl[i] do{记下字符所在位置}
     put_in(s[j],i,j);
   end;
  close(inf);
```

```
 end;

procedure clear(snum:byte);{同值清零}
 var i:byte;
     now:longint{过渡变量};
 begin
  now:=start[snum]^[tl[snum]];
  start[snum]^[tl[snum]]:=0;
  for i:=tl[snum]-1 downto 1 do
   if start[snum]^[i]<>now then exit
    else
     start[snum]^[i]:=0;
 end;

procedure cl(sstart:longint; snum:byte);{用动态规划来计算当前匹配到的最大值}
 var i,j,k,maxwz{在当前隐藏代码之前的最大值所在位置}:integer;
     maxtot{在当前隐藏代码之前的最大值},pnow{当前目标最大值}:longint;
     b:boolean;{布尔标记}
 begin
  maxtot:=0; maxwz:=0;
  i:=1;
  while (i<=nn) and (tn[i]^.finish<sstart) do{计算在当前隐藏代码之前的最大值}
   begin
    if tn[i]^.tot>maxtot then{判断、替代当前最大值}
     begin
      maxtot:=tn[i]^.tot;
      maxwz:=i;
     end;
    inc(i);
   end;

  pnow:=maxtot+tl[snum];  {计算最大匹配值}
  b:=false;
  if (nn=0) or (tn[nn]^.finish<m) then
  {将当前隐藏代码放入数组中}
   begin
    inc(nn);
    new(tn[nn]);
    b:=true;
   end
  else
   if pnow>tn[nn]^.tot then b:=true;

  if b then
```

```
  begin
   if pnow>findmax then{记录当前目标最大值}
    begin
     findmax:=pnow;
     findwz:=nn;
    end;

   with tn[nn]^ do{当前以最近找到的代码字结尾的最大值}
    begin
     up:=maxwz; num:=snum;
     tot:=pnow;
     start:=sstart;
     finish:=m;
    end;
  end;

 clear(snum);
 {注意：在找到一个最小右隐藏数列后，应当看到数组里原有的数值会对后边的递推有不利的影响，如有一个文本为 TUNN，代码字为 TUN 的数据，当找到 TUN 后，当前数组为 1、1、1，若将它保留，又将会找到一个 TUNN 的隐藏代码。此时，应当将数组中还未找到新的(也就是开始位置更靠后的)隐藏代码的位置清空(可以把它叫作“同值清零”)}
 end;

procedure main;{读入文本、处理}
 var i:integer;{循环变量}
     nnum,nci{过渡变量}:byte;
     ch:char;{当前字符}
 begin
  assign(inf,textfile);
  settextbuf(inf,tv);
  reset(inf);
  m:=0;
  while not eof(inf) do
   begin
    read(inf,ch);
    pp:=t[ch];
    inc(m);
    while pp^.next<>nil do{依次寻找字符所在位置}
     begin
      pp:=pp^.next;
      nnum:=pp^.num; nci:=pp^.ci;
      if nci=1 then{根据长度、当前匹配到的位置来作出记录、存放的处理}
       if tl[nnum]=1 then cl(m,nnum)
        else
```

```
        start[nnum]^[1]:=m
      else
       if (start[nnum]^[nci-1]<>0) and (m-start[nnum]^[nci-1]+1<=1000){边界条件} then
        if nci=tl[nnum] then cl(start[nnum]^[nci-1],nnum)
         else
          start[nnum]^[nci]:=start[nnum]^[nci-1];
     end;
   end;
  close(inf);
 end;

procedure print;{输出结果}
 var k:integer;{逆推变量}
 begin
  assign(outf,outfile);
  rewrite(outf);
  settextbuf(outf,tv);
  writeln(outf,findmax);
  k:=findwz;

  while k<>0 do{逆推找具体方案}
   begin
    writeln(outf,tn[k]^.num,' ',tn[k]^.start,' ',tn[k]^.finish);
    k:=tn[k]^.up;
   end;
  close(outf);
 end;

begin
 nn:=0; findmax:=0;
 init;{读代码字文件，初始化}
 main;{读入文本、处理}
 print;{输出结果}
end.
```

参考书目

1．《数据结构(第二版)》(严蔚敏、吴伟民编著，清华大学出版社出版)

2．《IOI ’98 中国集训队优秀论文集》(IOI 中国队教练组编)

3．《信息学奥林匹克(季刊)》(王帆、倪兆中主编)

4．《青少年国际和全国信息学(计算机)奥林匹克竞赛指导--------图论的算法与程序设计》 (吴文虎、王建德编著，清华大学出版社出版)