关于 DNA 序列 *k*-mer index 问题的改进 Hash 函数多线程算法

黄恒意[1]　许仁杰[1]　张艺辉[1]

指导教师：数模教练组[1,2]

[1] 复旦大学数学科学学院，上海 200433

[2] 上海市现代应用数学重点实验室，上海 200433

摘　要

本文通过字符串匹配的形式解决了 DNA 序列的 *k*-mer index 问题。

最简单的方法是字典树，然而这种用空间换时间的算法只能适用于较小的 *k*。

我们尝试了 KMP 算法，这是一种利用需要匹配的 DNA 序列自身的特点进行查找的算法，其占用内存大约为 100MB，单次查找的时间复杂度为 O(*NL*) (其中 *L* 为 DNA 序列长度，*N* 为数据规模)，但是这种算法在用于多次查找时，时间复杂度过高。

所以我们采取了 Hash 算法，利用 4 进制数，对固定的 *k*，将长度为 *k* 的子序列进行分类，以此建立索引。这种方法对较小的 *k* 是适用的，但是对较大的 *k*，由于数值太大导致存储量过大、计算速度太慢，此时可采用按大素数取模的方法来分类。而这又有可能使许多不同的数放在同一类中，所以我们还需要利用另一个大素数将同一类中的数作进一步分离。经过对给定数据的测试，最多只需要用 3 个大素数就可以对这组 DNA 序列进行完全分离。经过改进的 Hash 算法查询搜索的计算复杂度为 O(1)，查询搜索的时间基本上远小于 1 ms，建立索引的计算复杂度为 O(*NL*)，建立索引的时间平均为 8 s。空间复杂度为 O(*NL*)，实际占用内存 1GB 左右。然后我们分别采用随机生成的 DNA 模拟数据、将原始 DNA 序列错位排列两种方法来检验 Hash 算法的实用性，发现建立索引的时间最大仍不超过 12 s，所以改进后的 Hash 算法是一种高效实用的算法。

鉴于 Hash 算法对原始 DNA 序列使用的内存远低于 8GB，采取多线程的方法可以加快建立索引的速度，建立索引的时间降低到原来的 2/3 左右，占用的内存则增加到原来的 2 倍左右。同样采用随机生成的 DNA 模拟数据和将原始 DNA 序列错位排列两种方法检验多线程 Hash 算法，结果表明多线程 Hash 算法在存储空间不超过 8GB 的条件下可以有效地减少建立索引的时间。

最后，以 DNA 序列长度和数据规模作为控制变量进行测试，发现对更大规模的 DNA 序列，改进后的 Hash 算法仍然适用。

关键词：*k*-mer index；Hash 函数；素数；分类；多线程

本文得到国家基础科学人才培养基金的资助，项目号 J1103105

一、 问题的重述与分析

这个问题来自 DNA 序列的 *k*-mer index 问题。

给定一个 DNA 序列，这个系列只含有 4 个字母 ATCG，如 *S* = “CTGTACTGTAT”。给定一个整数值 *k*，从 *S* 的第一个位置开始，取一连续 *k* 个字母的短串，称之为 *k*-mer（如 *k*= 5，则此短串为 CTGTA）， 然后从 *S* 的第二个位置，取另一 *k*-mer（如 *k*= 5，则此短串为 TGTAC），这样直至 *S* 的末端，就得一个集合，包含全部 *k*-mer。

对上述序列 *S* 来说，所有 5-mer 为

{CTGTA，TGTAC，GTACT，TACTG，ACTGT，TGTAT}

通常这些 *k*-mer 需一种数据索引方法，可被后面的操作快速访问。例如，对 5-mer 来说，当查询 CTGTA，通过这种数据索引方法，可返回其在 DNA 序列 *S* 中的位置为{1，6}。

k-mer index 问题本质上就是一个字符串匹配问题，对于字符串匹配问题一般采用 KMP 算法[1]和 Hash 函数[4]来解决。

二、 符号定义与说明

B[i]，B(i)： DNA 片段 B 的第 i 个字母

B[i..j]，B(i..j)：DNA 片段 B 的第 i 个字母到第 j 个字母这 j-i+1 个字母组成的子串

Num(B[i])： 用数字表示 B 串的第 i 个字母，A 代表 0，T 代表 1，C 代表 2，G 代表 3。

三、 KMP 算法

假设给定的 DNA 片段为 B（B 是一个长度为 *k* 的片段），将 B 放在每个 DNA 的每一位上进行比较，这样一定能找出所有完全匹配上的位置，但是这样做的查询速度明显非常慢。

判断片段 B 是否可以在 DNA 序列 A 的第 i 个位置完全匹配上需要先比较 B 的第 1 个字母 B(1)与 A 的第 i 个字母 A(i)是否相同，若相同则比较 B(2)与 B(i+1)，……，一旦不相同可直接判定为匹配失败，如果可以完全匹配上，则会一直比较到 B(k)与 A(i+k-1)，即需要进行 *k* 次计算。

给定了 100 万个 DNA 序列，且每个序列长度为 100，则每个 DNA 序列有 $100 - k + 1 = 101 - k$ 个位置可以进行匹配，最坏情况下每个位置需要进行 *k* 次计算，则对于 1 个 DNA 序列就要进行 $k\ (101 - k)$ 次计算，100 万个 DNA 序列一共进行了 $10^6\ k\ (101 - k)$ 次计算，当 $k = 50$ 的时候大约要进行 25×10^8 次计算，按照普通计算机大约 1 秒 1 亿次的运算能力，一次查询就需要 25 s 左右的时间，这是不能接受的。

当然这是最坏情况下的时间，比如 A = "AAAAAAAAAT"，B = "AAAAT"。给定的数据远好于最坏情况，所以时间不会达到 25 s，但是我们可以试图优化这个方法。

这个方法效率低下的一个原因是检测完一个 DNA 序列，最坏情况下需要 *k*

(101 – *k*) 次计算，使用 KMP 算法来匹配可以使这个计算次数降低为与 100 + *k* 同数量级，即时间复杂度为线性。下面介绍 KMP 算法。

KMP 算法是一种改进的字符串匹配算法，由 D. E. Knuth 与 V. R. Pratt 和 J. H. Morris 同时发现，因此人们称它为克努特-莫里斯-普拉特操作（简称 KMP 算法）。KMP 算法的关键是利用匹配失败后的信息，尽量减少两个字符串的匹配次数以达到快速匹配的目的。

例如，A = "TGTGTGTTGTGTCG"，B = "TGTGTCG"，我们来看看 KMP 是怎么工作的。我们用两个指针 i 和 j 分别表示，A[i–j+1..i]与 B[1..j]完全相等。i 不断增加，随着 i 的增加 j 相应地变化，且 j 满足以 A[i]结尾的长度为 j 的字符串正好匹配 B 串的前 j 个字符（j 当然越大越好）。现在需要检验 A[i+1]和 B[j+1]的关系。当 A[i+1] = B[j+1]时，i 和 j 各加 1；什么时候 j = m 了，我们就说 B 是 A 的子串（B 串已经完全匹配上了），并且可以根据这时的 i 值算出匹配的位置。当 A[i+1] ≠ B[j+1]，KMP 的策略是调整 j 的位置（减小 j 值），使得 A[i–j+1..i]与 B[1..j]保持匹配且新的 B[j+1]恰好与 A[i+1]匹配（从而使得 i 和 j 能继续增加）。我们看一看当 i = j = 5 时的情况：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B = T G T G T C G
j = 1 2 3 4 5 6 7
```

此时，A[6] ≠ B[6]。这表明，此时 j 不能等于 5 了，我们要把 j 改成比它小的值 j'。j'可能是多少呢？仔细想一下，我们发现，j'必须要使得 B[1..j]中的前 j'个字母和末 j'个字母完全相等（这样 j 变成了 j'后才能继续保持 i 和 j 的性质）。这个 j'当然要越大越好。在这里，B [1..5] = "TGTGT"，前 3 个字母和末 3 个字母都是 TGT。而当新的 j 为 3 时，A[6]恰好和 B[4]相等。于是，i 变成了 6，而 j 则变成了 4：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B =     T G T G T C G
j =     1 2 3 4 5 6 7
```

从上面的这个例子，我们可以看到，新的 j 可以取多少与 i 无关，只与 B 串有关。我们完全可以预处理出这样一个数组 P[j]，表示当匹配到 B 串的第 j 个字母而第 j+1 个字母不能匹配了时，新的 j 最大是多少。P[j]应该是所有满足 B[1..P[j]] = B[j-P[j]+1..j]的最大值。

再后来，A[7]=B[5]，i 和 j 又各增加 1。这时，又出现了 A[i+1]≠B[j+1]的情况：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B =   T G T G T C G
j =   1 2 3 4 5 6 7
```

由于 P[5]=3，因此新的 j=3：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B =     T G T G T C G
j =     1 2 3 4 5 6 7
```

这时，新的 j=3 仍然不能满足 A[i+1]=B[j+1]，此时我们再次减小 j 值，将 j 再次更新为 P[3]：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B =     T G T G T C G
j =     1 2 3 4 5 6 7
```

现在，i 还是 7，j 已经变成 1 了。而此时 A[8]仍然不等于 B[j+1]。这样，j 必须减小到 P[1]，即 0：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = T G T G T G T T G T G ……
B =           T G T G T C G
j =     0 1 2 3 4 5 6 7
```

终于，A[8] = B[1]，i 变为 8，j 为 1。事实上，有可能 j 到了 0 仍然不能满足 A[i+1] = B[j+1]（比如 A[8] = "d"时）。因此，准确的说法是，当 j = 0 了时，我们增加 i 值但忽略 j 直到出现 A[i] = B[1]为止。

这个过程可以理解为扫描字符串 A，并更新可以匹配到 B 的什么位置。算法是线性的。现在分析这个计算的时间复杂度，我们将用到时间复杂度的摊还分析中的主要策略，简单地说就是通过观察某一个变量或函数值的变化来对零散的、杂乱的、不规则的执行次数进行累计。KMP 的时间复杂度分析可谓摊还分析的典型。我们从上述的 j 值入手。每一次执行 while 循环都会使 j 减小(但不能减成负的)，而另外的改变 j 值的地方只有 j = j + 1。每次执行了这一行，j 都只能加 1；因此，整个过程中 j 最多加了 N 个 1。于是，j 最多只有 N 次减小的机会（j 值减小的次数当然不能超过 N，因为 j 永远是非负整数）。这告诉我们，while 循环总共最多执行了 N 次。按照摊还分析的说法，平摊到每次 for 循环中后，一次 for 循环的复杂度为 O(1)。整个过程显然是 O(A_length)的。这样的分析对于后面 P 数组预处理的过程同样有效，同样可以得到预处理过程的复杂度为 O(k)。

我们可以通过 P[1]，P[2]，…，P[j-1]的值来获得 P[j]的值。对于刚才的 B = "TGTGTCG"，假如我们已经求出了 P[1]，P[2]，P[3]和 P[4]，看看我们应该怎么求出 P[5]和 P[6]。P[4] = 2，那么 P [5]显然等于 P[4]+1，因为由 P[4]可以知道，B[1,2]已经和 B[3,4]相等了，现在又有 B[3] = B[5]，所以 P[5]可以由 P[4] 后面加一个字符得到。P[6]也等于 P[5]+1 吗？显然不是，因为 B[P[5]+1] ≠ B[6]。那么，我们要考虑“退一步”了。我们考虑 P[6]是否有可能由 P[5]的情况所包含的子串得到，即是否 P[6] = P[P[5]] + 1：

```
      1 2 3 4 5 6 7
B = T G T G T C G
P = 0 0 1 2 3 ?
```

P[5] = 3 是因为 B[1..3]和 B[3..5]都是 TGT；而 P[3]=1 则告诉我们，B[1]和 B[5]都是 T。既然 P[6]不能由 P [5]得到，或许可以由 P[3]得到（如果 B[2]恰好和 B[6]相等的话，P[6]就等于 P[3] + 1 了）。显然，P[6]也不能通过 P[3]得到，因为 B[2] ≠ B[6]。事实上，这样一直推到 P[1]也不行，最后得到 P[6] = 0。

至此，KMP 算法已全部介绍完毕。对于每个查询，给定了一个 B 串，首先预处理 P 数组，进行 k 次计算，时间复杂度 O(k)，DNA 序列 A 串长度 A_length = 100，检测 A 串的时间复杂度为 O(A_length)，由于 $k \le 100$，所以总时间复杂度 O(100 + k)为线性。而 100 万个 DNA 序列的总时间复杂度为 $O(10^6(100 + k))$。

KMP 算法只需一个长度为 k 的数组 p 和 B 本身，不需要其他空间。

总之，对 KMP 算法而言：

建立索引：只读入数据，不做任何额外计算。（读入数据的复杂度是 O(NL)，其中 L 为总数据长度，即 $N = 10^6$，$L = 10^8$。）

1. 计算复杂度：0。
2. 空间复杂度：0。

查询：

1. 计算复杂度：O(NL)。
2. 空间复杂度：O(k)。

性能评价：

1. 查询速度：复杂度 O(NL)，$k = 100$ 时大约进行 2 亿次计算，每次查询大约需要 1 到 2 s。
2. 索引内存使用：O(NL)，只用了不到 100MB 的内存。
3. 可支持的 k：可支持一切 k，即 1 到 100，并且 k 可以不事先给定。
4. 建立索引时间：0。没有建立索引。

KMP 算法的算法步骤如下：

1. 读入 k，B
2. 令 j = 0，i = 1
3. 如果 j > 0 且 B[j+1] ≠ A[i]不成立，转第 5 步
4. 令 j = p[j]
5. 如果 B[j+1]不等于 A[i]，转第 9 步
6. j = j+1
7. 如果 j 不等于 m，转第 9 步
8. j = p[j]
9. i = i + 1
10. 如果 i > n，结束；否则转第 3 步。

KMP 算法的流程图如图 1 所示。

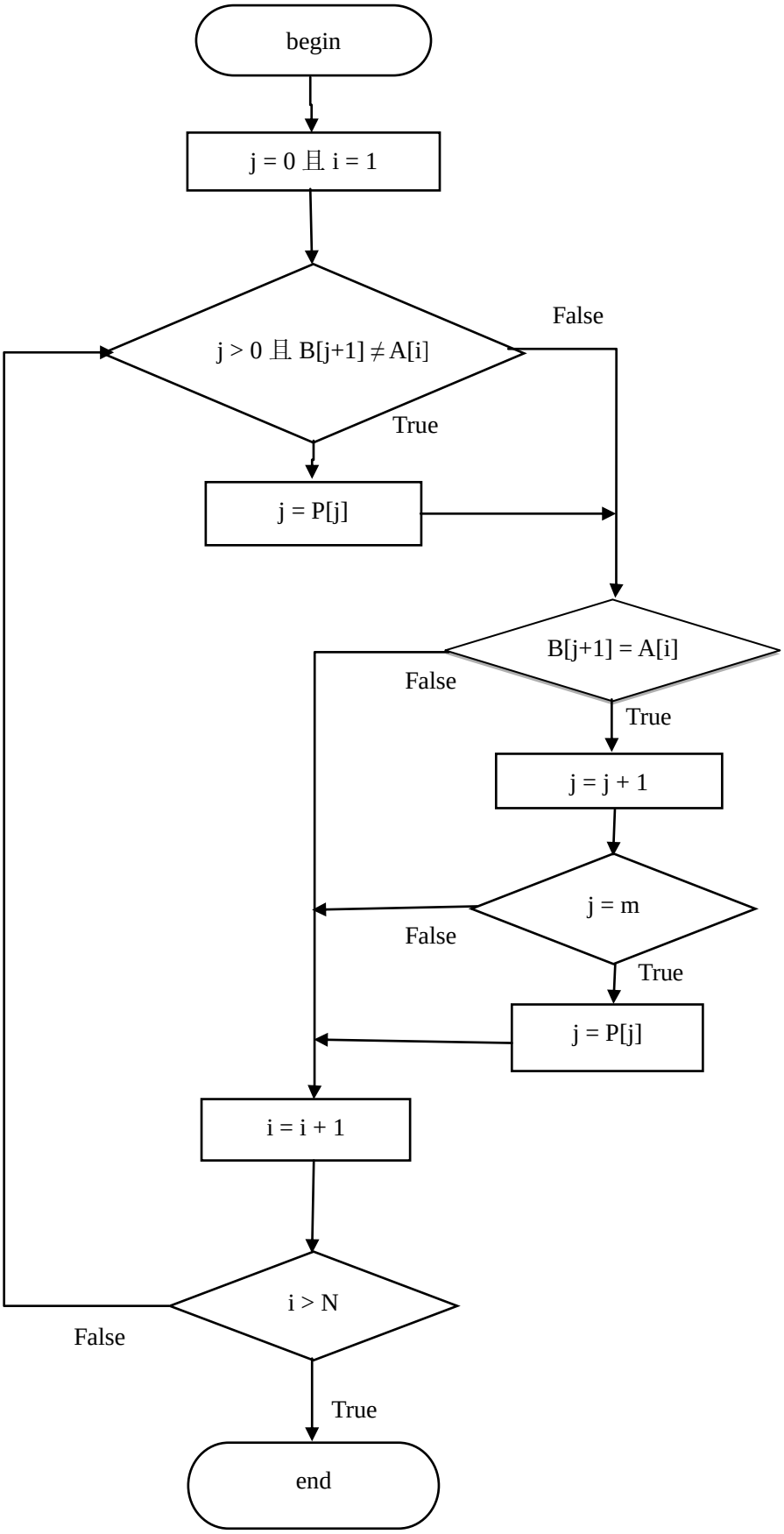
begin
j = 0 且 i = 1
j > 0 且 B[j+1] ≠ A[i]
False
True
j = P[j]
B[j+1] = A[i]
False
True
j = j + 1
j = m
False
True
j = P[j]
i = i + 1
i > N
False
True
end

图 1

四、 字典树方法

由于 KMP 算法没有建立 *k*-mer 问题的 index，所以我们引入字典树方法。

字典树[2]，又称单词查找树或 Trie 树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串，例如用在 DNA 序列中），所以经常被用于文本词频统计。

字典树利用字符串的公共前缀来节约存储空间，最大限度的减少无谓的字符串比较，查询效率非常高。然而它是一种典型的用空间换时间的结构。内存需求巨大。

下图是一个将字典树应用在 DNA 序列中的例子。假设现在有一个 DNA 序列是 ACTAC，k = 3。我们需要建立 ACT、CTA、TAC 这三个片段的索引。先取出 ACT，将其每个字母依次接在前一个字母下面，第一个字母 A 接在一个空结点下面。

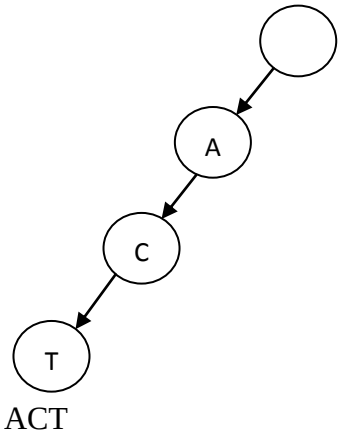

图 2

图 2 中从根结点这个空结点开始往下走到 T，形成了 ACT 这个 DNA 片段，则表示原 DNA 序列中存在 ACT 这个片段，只需把 ACT 出现的位置保存在最下层表示 T 的这个结点上，就建立了索引。

接着插入 CAT 和 TAC 这两个片段（见图 3）。

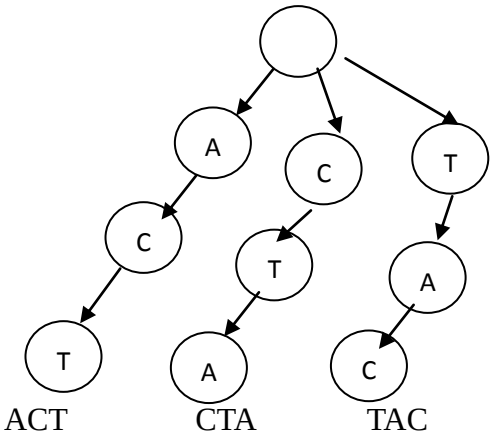

图 3

这就为 ACTAC 这个 DNA 序列的 3-mer 建好了索引。

现在查询 TTT 这个 DNA 片段的位置（见图 4）。

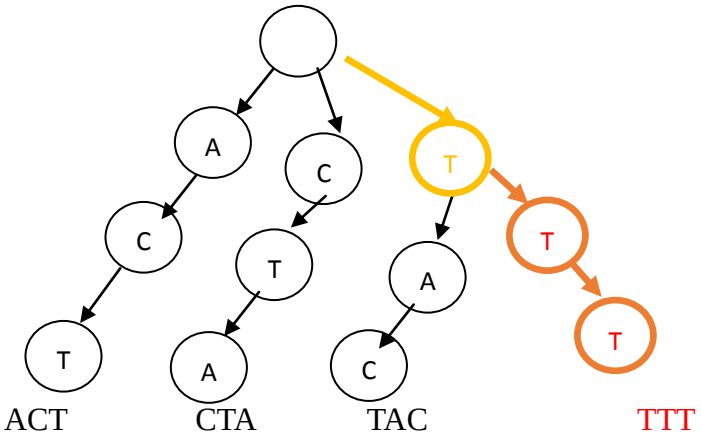

图 4

可以看到，首先从根结点开始向下查找第一位的 T，找到的这个 T 标上了黄色，然后查找第二位 T，然而在这棵树中黄色结点下面并没有直接连着一个为 T 的结点，所以这时候可以直接判定 TTT 这个 DNA 片段不存在与给定的 DNA 序列中。红色部分则是若能查找到 TTT，这棵树应该是怎样的。

再增加一个给定的 DNA 序列来 TACTG 说明这棵树的特性。取出长度为 3 的三个 DNA 片段 TAC、ACT、CTG。插入字典树，如图 5。

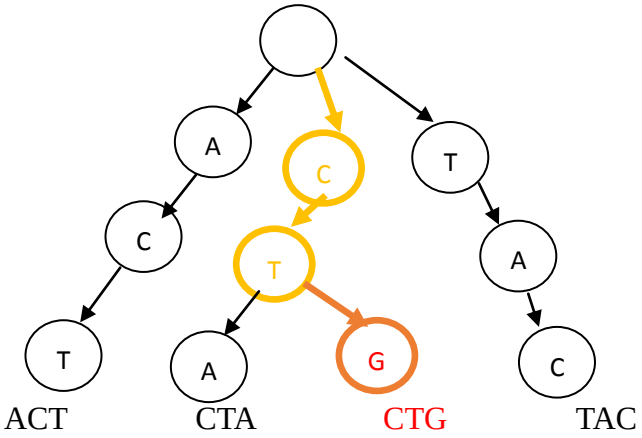

图 5

可以看到，原树中已存在 TAC 和 ACT，所以对树不用做任何修改。插入 CTG 的时候按照黄色路径一路向下走，发现第二位 T 再往下没有 G 这个点，则将 G 这个点插入树中如图 5 的位置。

上面这个例子说明这棵树是可以让各 DNA 片段的前缀共享的，例如 CTA 和 CTG 的公共前缀是 CT，则在树中他们共用了 CT 这两个点，节省了空间。

本题中每个 DNA 序列将要被取出$(100 - k + 1 = 101 - k)$个 DNA 片段，共有 100 万个 DNA 序列，就会有 $10^6 \times (101 - k)$个 DNA 片段，每个片段长度是 k，所有总共的脱氧核糖核苷酸数量是 $k \times 10^6 \times (101 - k)$，当 $k = 50$ 的时候这个数字将会达

到 2.55×10^9。

按照字典树中的存储结构，每个结点需要 4 个 int 型的附加信息，用来存储它向下的 ATCG 四个点的结点编号。如果这些 DNA 片段没有任何前缀的重合，需要的内存是 $2.55 \times 10^9 \times 4 / 1024^3 = 9.49$G。但是实际上这些 DNA 片段肯定互相之间是有重合的，然而当 k=50 的时候，有部分重合的片段将会非常少，重合的部分也可能非常少，所以在内存使用上，仅仅是存下这棵字典树，刚处理完十万个 DNA 序列，就超过了 3G 内存，存下所有数据将会超过 8G 内存，所以字典树这个数据结构不能完美解决 k 为 1 到 100 的所有情况。当 k＝15 的时候所占用的内存就会超过 8G。

还有一个值得注意的地方是，内存使用过大本身也会拖慢程序的运行速度，就像一个背着重物跑步的人。所以用了 6G 这么大的内存，虽然在允许范围内，但是速度会很慢。

k =5 时，建立索引时间为 3290 ms，查询时间为 0 ms。

该算法建立索引的算法步骤如下：

1. 读入 A
2. 令 k = 0，i = 0
3. i = i + 1
4. j = i – 1，令 state 为 0，即根节点
5. j = j + 1
6. 如果 state 这个点下面有 A[j]这个字母的点，转第 8 步
7. 建立一个点挂在 state 下面，字母为 A[j]
8. state = 新建节点的编号
9. 如果 j – i + 1 < k，转第 5 步
10. 如果 i + k – 1 < n，转第 3 步；否则结束

该算法建立索引的流程图如图 6 所示。

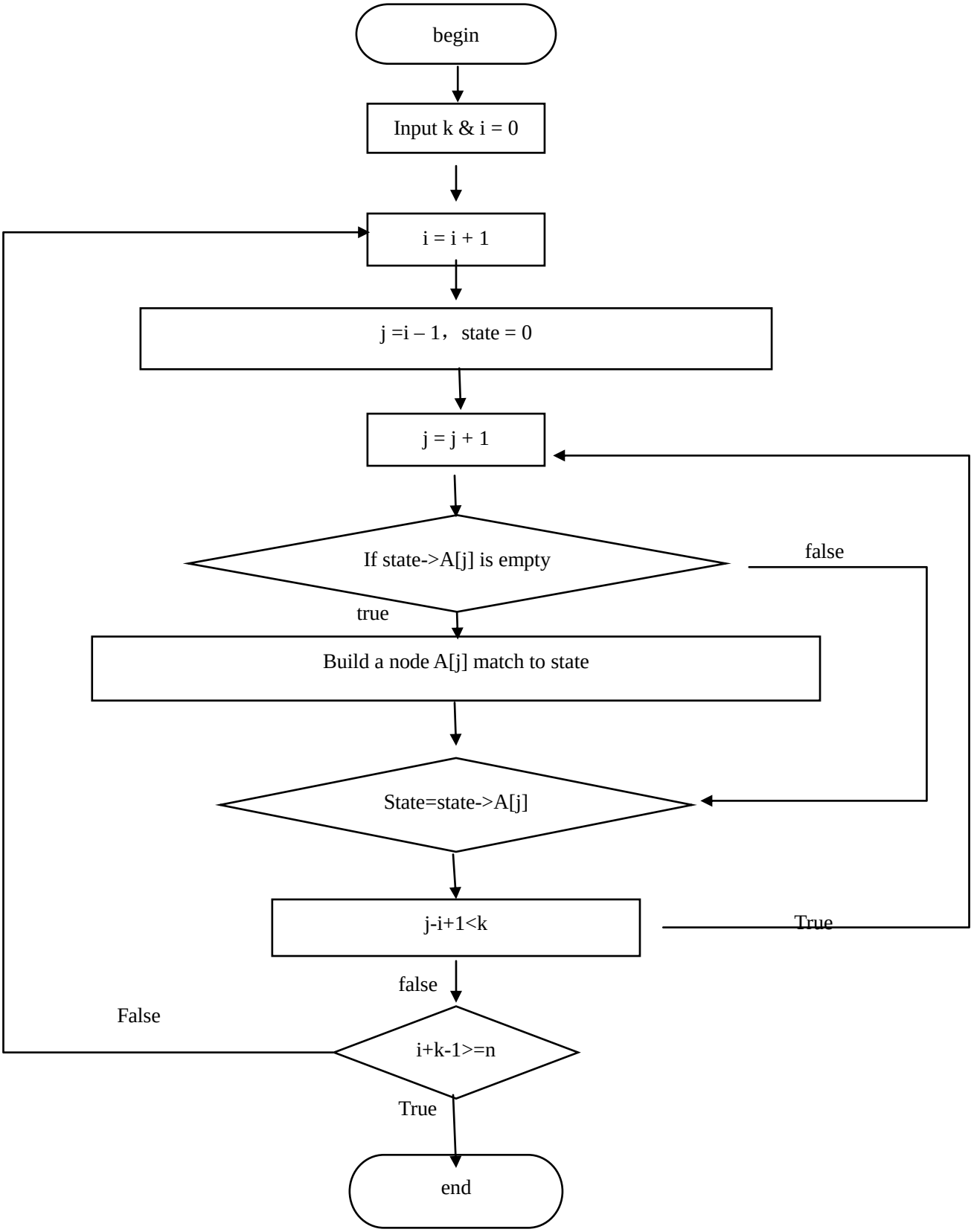
begin
Input k & i = 0
i = i + 1
j =i – 1，state = 0
j = j + 1
If state->A[j] is empty
false
true
Build a node A[j] match to state
State=state->A[j]
j-i+1<k
True
false
False
i+k-1>=n
True
end

图 6

该算法查找的流程图如图 7 所示。

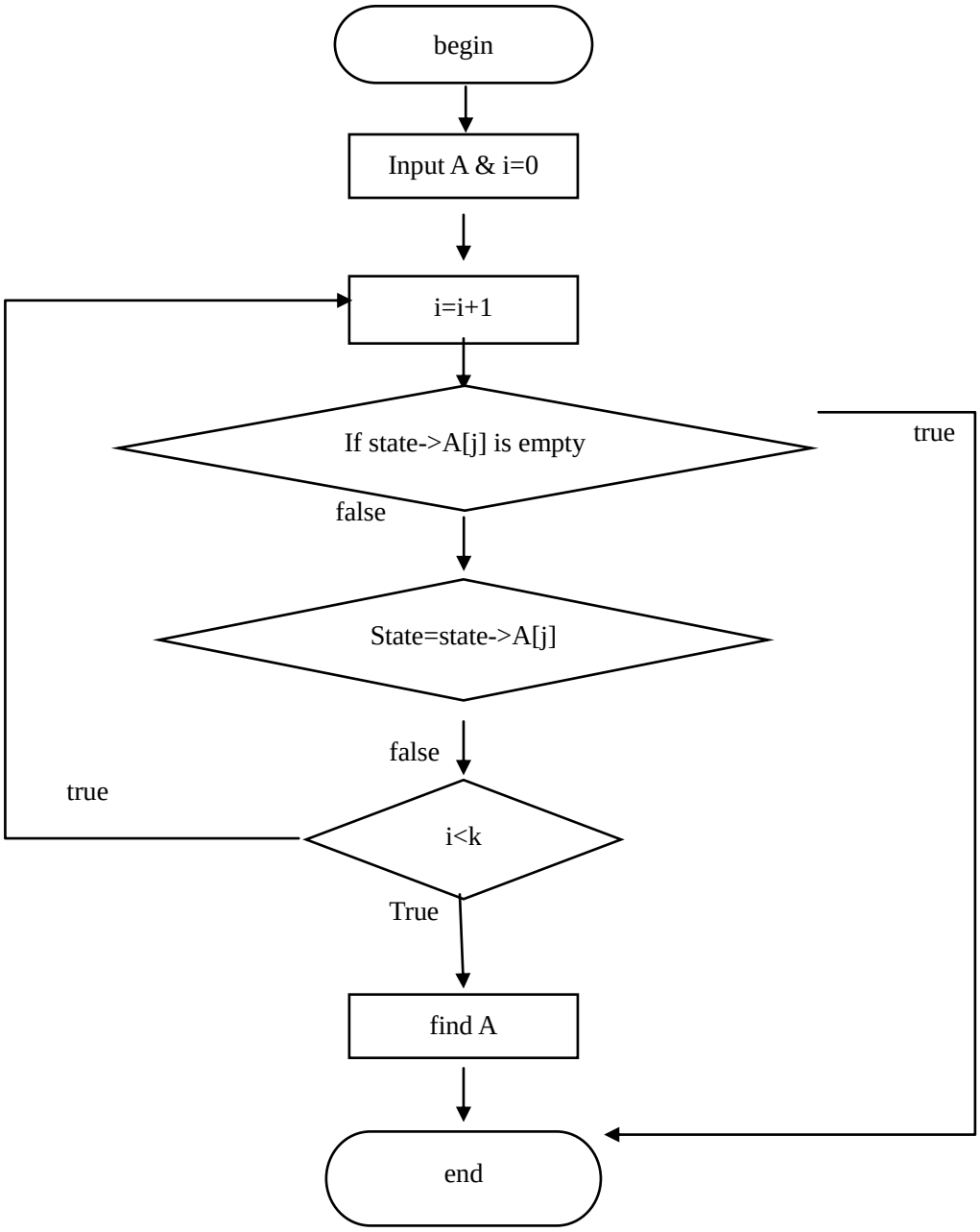

图 7

五、 后缀数组

字典树的储存空间超过 8GB 的要求，此时考虑后缀数组[3]这种强有力的处理后缀问题的数据结构。

后缀数组的倍增算法的主要思路是对每个字符开始的长度为 2k 的子字符串进行排序，求出排名，即这个字符串在所有长度为 2k 的子字符串中按字典序排序，排名第几。k 从 0 开始，每次加 1，当 2k 大于 n 以后，每个字符开始的长度

为 2k 的子字符串便相当于所有的后缀。每一次排序都利用上次长度为 2k-1 的字符串的排名，那么长度为 2k 的字符串就可以用两个长度为 2k-1 的字符串的排名作为关键字表示，然后进行基数排序，便得出了长度为 2k 的字符串的排名。

后缀数组的方法在罗穗骞的《后缀数组——处理字符串的有力工具》一文中已经有非常详细的介绍，这里对算法具体的实现细节不再赘述。

倍增算法的时间复杂度比较容易分析。每次基数排序的时间复杂度为 O(n)，排序的次数决定于最长公共子串的长度，最坏情况下，排序次数为 logn 次，所以总的时间复杂度为 O(nlogn)。

二分查找

让用户输入查询的 k-mer 长度 k，及查询的 k-mer，利用二分搜索法，将 k-mer 与 dnaList 各后缀的 k 前缀比较大小，找出一个起始位置为 sa[i]的后缀，它的 k 前缀与 k-mer 匹配。若不存在，则查找结束。

寻找剩余后缀

当我们用二分搜索找到匹配的项时,我们仍需要往前往后寻找其他的匹配项,由于当 k 的长度较小时，询问的 k-mer 出现的次数可能很多，如果我们仍将 k-mer 与后缀项进行匹配效率会变低下。

为了充分利用我们在二分搜索时所作的匹配工作,我们在构建后缀数组后建立了一个 height 数组,其中 height[i]表示 sa[i]代表的后缀与 sa[i-1]代表的后缀的最大前缀长度，这样我们在查找剩余后缀时只需要比较最大前缀长度是否大于等于 k-mer 的长度,一旦我们发现最大前缀长度小于 k-mer 的长度那我们就可以断定后续的后缀项中不是以 k-mer 为前缀的，这样省去了在查找剩余匹配后缀时所需要的再次匹配工作，这在匹配 k-mer 的后缀项多时效果尤为明显。

共需要大约 80s 的时间。显然这个时间太慢，不可接受。

后缀数组的流程图 8 如下：

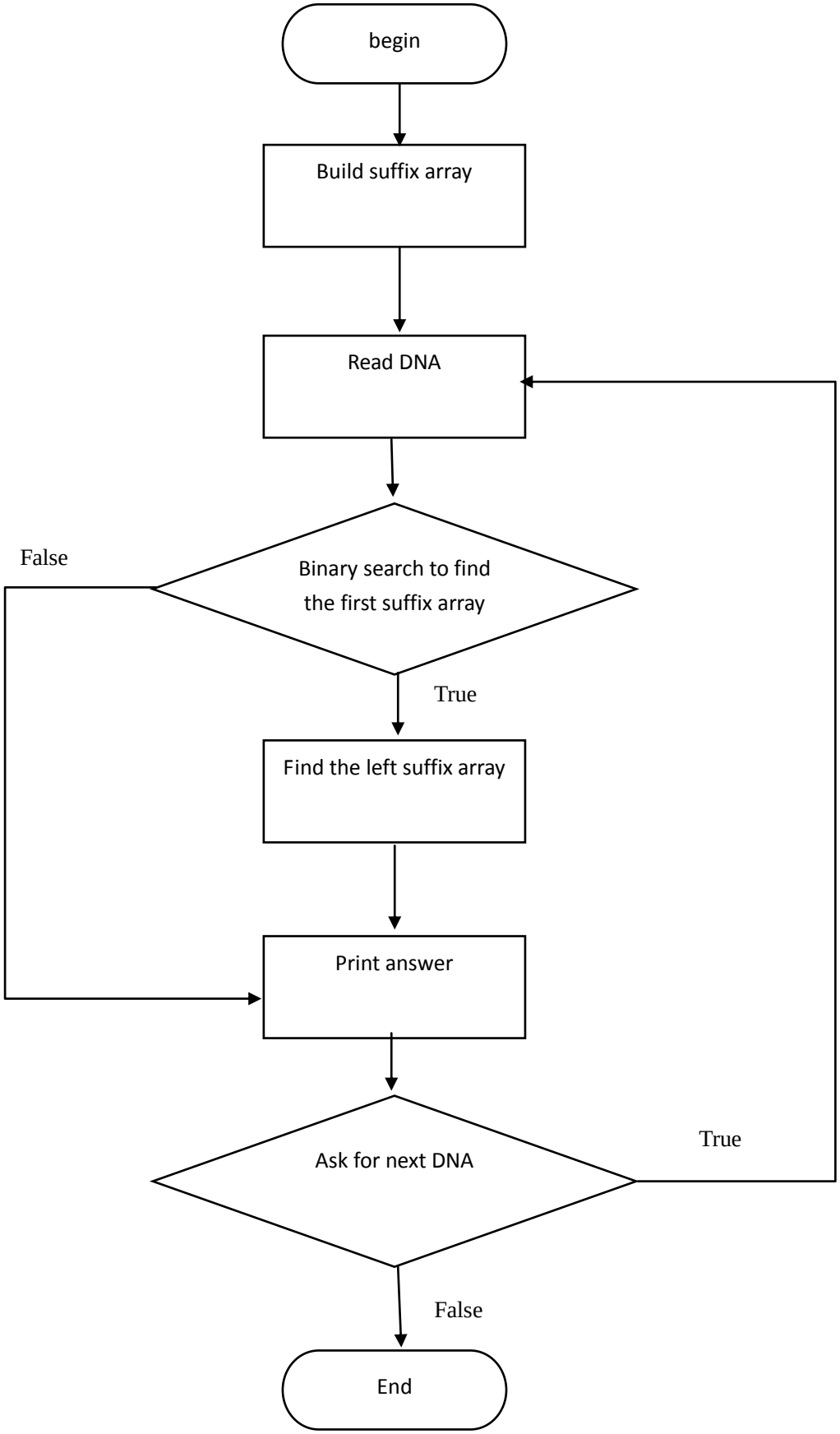
begin
Build suffix array
Read DNA
Binary search to find the first suffix array
False
True
Find the left suffix array
Print answer
Ask for next DNA
True
False
End

图 8

六、 Hash 函数算法

我们知道了单次匹配时间复杂度为 O(k)，这是因为要把长度为 k 的每个字母串进行比较，现在尝试在这一点上进行优化。

两个字母串进行比较要花费大量时间，而计算机中两个整数进行比较则只要花费 1 次运算的代价。我们考虑能否将 DNA 片段变换为一个整数。

构造一个函数 F1，假设一个 DNA 片段 S="ATCG"，可以把 S 看成 4 进制数，A 代表 0，T 代表 1，C 代表 2，G 代表 3，则 F1(S)=((1*4+2)*4)+3=27，但是计算机可以存储的单个整数是有大小限制的，若子母串长度为 100，这样转换后的整数不能直接存下来。令 F1(S)=((1*4+2)*4)+3 mod p=27 mod p(p 为一个不超过存储上限的数)，这样就能存下来。

F1 满足一个性质，对于两个字母串 S1 和 S2，如果 F1(S1)≠F1(S2)，则 S1≠S2，如果 F1(S1)=F1(S2)，则不能判断 S1 是否等于 S2。所以我们不能将判断字母串相同直接转换为判断数字相等，但是如果 F1(S1)≠F1(S2)，我们不用逐字母比较 S1 与 S2 就知其不相同，这大大提高了比较效率。

基于这一点我们建立了这样一个模型：构造 $F1(s)=\sum_1^k Num(A[i])*4^{k-i} modP1$，其中 P1= 100000007，Num(A[1])表示将 A 的第 1 个字母变为 0 到 3 的数字。这样所有 DNA 片段都会被映射到一个[0,P1-1]的整数上。为了更形象地说明，假设我们有 p1 个桶，每个桶上写着它们的号码，分别是 0 到 P1-1。现在取出第一个 DNA 序列做如下操作：

1)设此 DNA 序列为 A，取出长度为 k 的子串 A[1..k]，计算 V=F1(A[1..k])。

2)找到号码是 V 的桶。

1.如果此桶为空，则拿出一张白纸写上 A[1..k]这个串本身，再写上它属于第 1 个 DNA 序列的第 1 个位置。将这张纸放入桶内，白纸叠着放好。如下图 9：

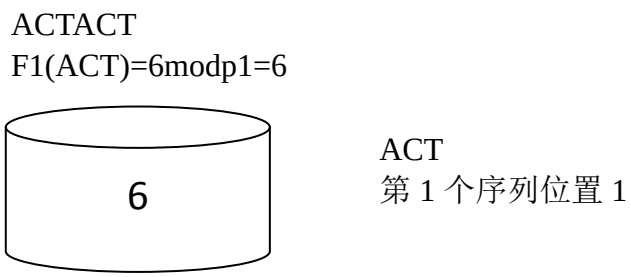

图 9

2.如果里面已经有其它白纸，则从上到下依次翻看白纸，比较纸上写的子串与 A[1..k]是否相同，若相同则在这张白纸上接在之前的字后面写上"第 1 个 DNA 序列的第 1 个位置"。然后放入桶内结束这一步的操作。如果翻看完所有白纸都没有找到相同的字母串，则做 1.中的操作。如下图 10，插入一个 XXX：

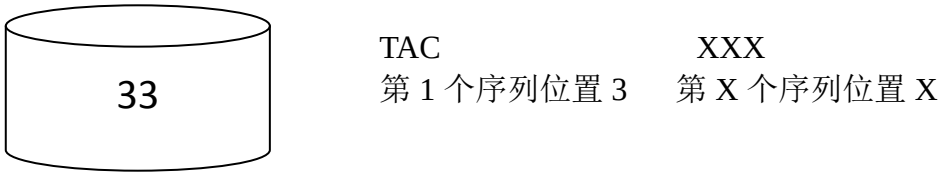

图 10

3)取出长度为 k 的子串 A[2..k+1]，计算 V=F1(A[2..k+1])。做 2)的操作。不同的仅是写的字。这里一个小技巧。F1(A[2..k+1])不需要用 k 次运算重新计算，因为 $F1(A[2\ldots k+1]) \equiv ((F1(A[1\ldots k])-Num(A[1]\times 4^{k-1})*4+Num(A[k+1]))\bmod p$。

4)依次取出长度为 k 的子串 A[3..k+2], A[4..k+3]……A[100-k+1..100]，做 3)的操作。

5)依次对第 2 个 DNA 序列，第 3 个 DNA 序列……第 100 万个 DNA 序列做上面 1)到 4)的操作。

这样我们就建立好了这个索引。对于每个询问 B，我们只需要计算 V=F1(B)，然后在 V 号桶里面逐张翻看白纸，若找到了写着 B 串的白纸，则上面写着所有关于 B 在哪个 DNA 序列的哪个位置出现的信息。若翻看完所有白纸都没有找到，则 B 没有出现过。如图 11：

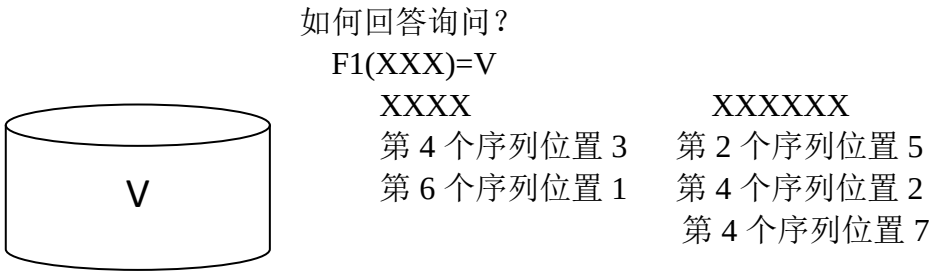

图 11

这个模型大大减少了用 k 次计算比较两个串是否相同的操作，只需在同一个桶内比较。而这个算法的效率很大程序上取决于所有的白纸是否较为平均的放在各个桶内。这与 F1 中对 P1 取模有关。P1=100000007 是一个质数，计算机可存储的一般整数上限为 2^{31}，数据中出现的子串所代表的 4 进制数的取值大致出现在$[4^{k-1},4^{k}-1]$之间，取足够大的质数 P1 保证了每个值较为平均地分布在每个桶上。这是计算机中高效的索引方法。我们可以继续优化这个模型。

假设有大量的 S1≠S2 满足 F1(S1)=F1(S2)，考虑增加一个函数 F2，如果 F2 这个函数构造的好的话，同时满足 F1(S1)=F1(S2)且 F2(S1)=F2(S2)的 S1≠S2 将会大大减少。如果再增加一个函数 F3，那么满足 3 个函数映射下都相等而本身不相同的 S1 与 S2 又会大大减少。

构造 $F2(S)=\sum_{1}^{k}Num(S[i])*4^{k-i}\bmod 79555771$

构造 $F3(S)=\sum_{1}^{k}Num(S[i])*4^{k-i}\bmod 80000023$

在 1)到 5)操作中，我们额外增加计算 F2 与 F3，将其写在白纸上的 DNA 片段本身的旁边。每次需要比较两个字母串前先比较两个串的 F2，F3 是否都相同，都相同才比较串本身。

这些桶和纸形成的结构如下图 12：

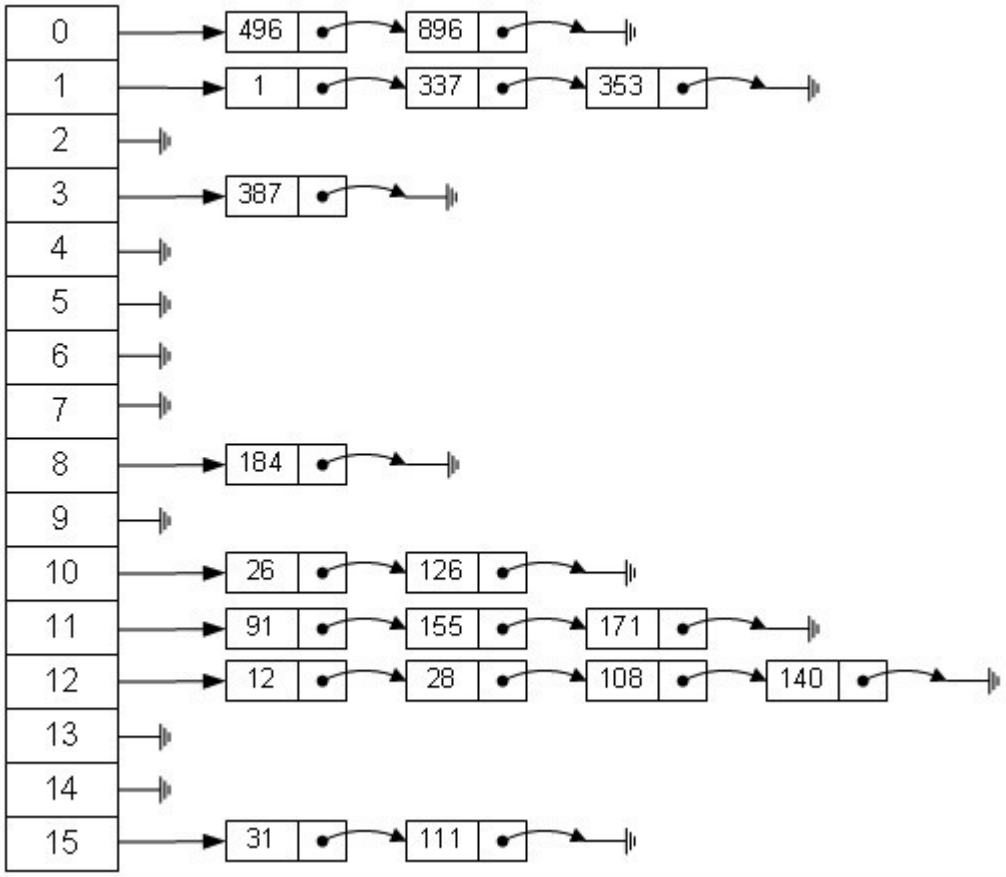

图 12

经写程序实践发现，在给定的 DNA 序列中，在各给定的 k 中，不存在两个不同的子串 S1≠S2，满足 F1(S1)=F1(S2)且 F2(S1)=F2(S2)且 F3(S1)=F3(S2)。也就是说在此题中比较两个 DNA 片段是否相同可以完全等价于判断三次整数是否相等。我们在程序中保留了满足 F1(S1)=F1(S2)且 F2(S1)=F2(S2)且 F3(S1)=F3(S2)情况下逐字母比较 S1 与 S2 的过程，但由于出错概率为 0，这几乎不影响查询速度。

这里简单说明一下为什么取一个 9 位数，因为一般的计算机整数类型是 32 位二进制，即 10 位十进制，取 10 位数的话计算过程中可能超出范围了，但这个数我们肯定取得越大越好，所以取了 9 位数。

鉴于用了 3 个函数之后还是要在很少的时候使用 DNA 本身检验是否相同，我们考虑了可否完全用若干个函数，即用若干个整数比较代替字符串比较。

一个长度为 100 的 DNA 片转换为 4 进制数大约是 4^{100}，大约是 60 位数，我们使用一个模 8 位数的函数 F1 和 3 个模 18 位质数 F2、F3、F4 的函数来计算，只要保证模的数 p1,p2,p3,p4 都是质数，则这个 4 进制数至少是 p1*p2*p3*p4，由于 p1*p2*p3*p4 为 8+18*3=62 位数，所以不存在两个不同 S1，S2，满足 F1(S1)=F1(S2)且 F2(S1)=F2(S2)且 F3(S1)=F3(S2) F4(S1)=F4(S2)，我们就做到了用比较整数完全代替比较字符串。但是这里的 18 位整数使用了计算机的特殊整数类型，最大可存储 64 位 2 进制数，计算机计算这种数据类型比一般的 32 位整数要慢，实际检验中我们发现这样做的运行速度比之前慢了不少，我们也尝试了用 6 个 9 位数代替 3 个 18 位数的做法，依然比之前慢，所以我们放弃了这个方法，使用之前的 3 个函数的做法。

我们来看一下建立索引空间复杂度。空间复杂度取决于 $10^6 \times (101-k)$个子串的重复程度，最坏情况下与 Len 同数量级，因此空间复杂度 O(Len)，实践中使用了 1G 内存就可满足一切 k 的查询。

我们接下来计算该算法的时间复杂度。

首先回忆一下，我们的 Hash 算法利用数论中模函数构造了若干同余类，在每个同余类中，采用链表把元素逐个链接起来，新进元素放在链接开头。

接下来我们引入两个基本假设

1. 简单一致分布规则：所有的元素分布在不同的同余类中的概率相等，并且相互独立。这类似于概率论中的独立同分布原则。

2. 元素总数与同余类个数成正比。这等价于我们所取的素数与元素总数成正比，在这个假设下，结合上条，我们就有结论：每个同余类中的元素个数一定。

第一步是对新的元素计算函数值 F1，然后把它插入到对应的同余类中，这个步骤需要的时间是 O(1)，如果再插入新元素前执行逐个检验，以确定有无已有该元素的话，在最坏的情况下，所需时间与表的长度成正比，我们来具体估计这个时间的范围。

假设所有的元素个数为 n，共有 m 个同余类，也即我们取的素数为 m，定义装载因子 alpha 为 n/m，即一个同余类中平均储存的元素数。在最坏的情况下，所有的元素都指向同一个同余类，这产生长度为 n 的链表，此时查找时间为 O(n)，再加上计算函数的时间，这就和用一个链表链接所有元素的情况差不多，幸好在我们的假设下不会出现这种情况。

在简单一致分布的假设下，对于 j=0,1, ... ,m-1,同余类 T[j]的长度用 nj 表示。那么有

Nj 的平均值为 E[nj]=alpha=n/m.

假定可以在 O(1)时间内计算出函数值 f1，那么查找同余类为 k 的元素的时间线性地依赖于表 T[k]的长度 nk。先不考虑计算以及寻找同余类的时间 O(1)，我们来看看查找算法期望查找的元素数，即为比较元素的同余类中有无该元素而在表 T[k]中检查的元素数。分为两种情况：

1.查找不成功，该同余类中没有一个元素与之相同。

2.查找成功

定理 1

对一个用链表存储的同余类，在简单一致假设下，一次不成功的查找所需要查找的元素个数的数学期望为 O(1+alpha)

证明：

在简单一致假设下，任何尚未被存储在表中的元素都是等可能的被分配到 m 个同余类中的任意一个之中。因为，当查找一个元素也就是字符串时，在不成功的情况下，检查的期望就是一直检查至链尾的数学期望。这一时间的期望长度就是该链表的长度，那么 E[nk]=alpha。于是，一次不成功的查找平均要检查 alpha 个元素，所需的总时间(包括计算函数的时间)为 O(1+alpha)。

对于成功的查找来说，情况略有不同，这是因为每个链表并不是等可能的被查找到的，因为每个链表的包含的元素个数可能并不相同。然而。下面的定理表明期望的查找时间仍然是 O(1+alpha)。

定理 2

在简单一致假设下，对于用链表技术存储元素的同余类，平均情况下的成功

查找时间为 O(1+alpha)

证明：

假定某次成功查找的元素与链表中任何一个元素相同的概率相等，那么在对元素 x 的一次成功的查找中，所检察的元素数比 x 所在的链表中，出现在 x 前面的元素数多 1。在该链表中，出现在 x 之前的元素都是在 x 之后插入的，这是因为所有新的元素都是在表头插入的。为了确定所查找元素的期望数目，对 x 所在的链表中，在 x 之后插入到链表中的期望元素数加 1，再对表中的 n 个元素 x 取平均，即可得到期望值。设 xi 是插入到所有表中的第 i 个元素，i=1,2,... ,n ，并以 ki 表示 xi 的同余类的值。那么利用概率论中示性函数 I，对于同余类 ki 和 kj，东一指示器随机变量 Xij=I{ki=kj}，在简单一致假设的条件下，对于固定的 i，P{ki=kj}=1/m，从而根据概率论的基本知识可得 E[Xij]=1/m，对于任意的 i=/j 都成立。于是，在一次成功的对元素 xi 的查找中，所检查元素的数目为$(1+\sum^{n}_{j=i+1}X_{ij})$。

其中 1 表示计算时间，Xij 表示在 xi 之后被插入到与 xi 同一个链表中的元素个数，也就是该链表中位于 xi 前面的元素个数，代表了需要查找的次数。那么对于所有的 n 个元素，对上述公式求和再取平均就得到平均查找时间：$1/n*\sum^{n}_{i=1}(1+\sum^{n}_{j=i+1}X_{ij})$.

再求期望，并利用期望性质以及上述若干结论，有

$E[1/n*\sum^{n}_{i=1}(1+\sum^{n}_{j=i+1}X_{ij})]= 1/n*\sum^{n}_{i=1}(1+\sum^{n}_{j=i+1}E[X_{ij}])$(根据期望线性)

$=1/n*\sum^{n}_{i=1}(1+\sum^{n}_{j=i+1}1/m)=1+1/nm\sum^{n}_{i=1}(n-i)$

$=1+1/nm(\sum^{n}_{i=1}n-\sum^{n}_{i=1}i)=1+1/nm(n^2-n(n+1)/2)$

$=1+(n-1)/2m=1+alpha/2-alpha/2n$

于是，一次成功的查找所需的全部时间(包括计算函数所需的时间)为 O(2+alpha/2-alpha/2n)=O(1+alpha)　定理证毕

有了这一结论再加上前面我们的第二条假设：链表数与元素总数成正比，即 n=O(m)，那么 alpha=n/m=O(m)=O(m)/m=O(1)。

这说明平均来说，查找操作需要常数量的时间，然而我们又知道插入操作在最坏的情况下需要 O(1)的时间，因而，全部的操作在平均情况下都可以在 O(1)时间内完成。

注意在本模型中，由于所取函数的性质，导致操作时间比理论值有更大的优化。

这是因为我们取了 9 位数的素数作为模函数，然而在较小的 k 值下(14 之下)，DNA 序列四进制转化之后的数比 9 位数还要小，所以在每个链表中基本不会有多于两个的链表长度，这将大大减小查询时间。即使在 k 值取值较大的情况，由于模函数的平均分布特性，产生了较为合理的 alpha 值，我们可以看到模型仍然取得了非常不错的表现。

哈希算法时间复杂度分析

我们来具体估计哈希算法中时间复杂度的系数。

先考虑最坏情况，所有的子列都被我们的函数映射到同一个哈希散列中，这时又有两种极端情况。

一、所有的子列实际上都相同。

这时不需要建立新的链表，对每个子列只需要计算三个模函数即可，时间复杂度为 3，假设计算所需时间为单位时间。我们可以看出这个同简单一致分布的情况并没有本质不同。

二、所有的子列互不相同。

这时对每个子列要建立新的链表，在此之前还要首先进行逐个比对。期望的比对个数为 n/2，在做每一次比对时，我们的模型并没有进行真正的字符串逐个审查，而是计算模函数 p2 和 p3，在实际运用中，由上述讨论知，如果有两个字符串的三个函数值完全一致，那么有充分信心认为它们是同一串字符串。这样的话需要进行的比对为：对在新字符串前面的每一个字符串计算两者的三个函数值并比对，至多需要进行 6 次运算，逐次计算六个函数值并比较。这样平均下来的时间复杂度为 6*n/2。这时最坏的一种情况，可以看出 6 是最坏的估计，因为有可能计算了两次之后就发现不同，不必再进行第三次比对。另一方面，这种情况在现实中其实不可能出现，一是因为所给 DNA 串有很强的随机性，而是因为我们所选的质数是与字符串总数成正比的，如果上述情况出现，那么所有的字符串总数为 n*m，这得出 m 为 1，与所取质数矛盾。

在简单一致分布的假设下

正如前面推导，有两类情况

一、一次不成功的搜索后插入链表

所需比对的字符串个数为 α，然而对每次字符串匹配，如前所述，最多需要进行 6 次运算，这样时间复杂度为 6*α。注意这时的 6 是最坏的情况，在实际中，根据数据的结构，计算次数也会有不同。

二、一次不成功的搜索后插入链表

如前所证，这时需要比对的字符串数量平均为　　1+α/2-α/(2n) ，

由于 α=n/m，我们可以把最后一项直接忽略掉，因为 m 是很大的质数，或者可以理解为我们做出最大的估计。这样平均每次需要的时间为 6*α/2，可以看出，上面的第二种情况正是这里的一种特殊形式，此时 α 即为 n。这从另一方面验证了我们模型的合理性。

总结

1. 在简单一致假设下，由于所有字符串平均分布，那么直观上，在字符串长度为 k 时，转换成的四进制数的量级为 4^(k+1)，那么平均每个同余类中有 4^(k+1)/m 个可能的不同的字符串列，这样需要新建链表的字符串数至多为 4^(k+1)，剩下的都会成功搜索到已有字符串。当注意 n 的值为 1000000*(101-k)，在实际中，我们只需要比较 4^(k+1)与 1000000*(101-k)的大小，再应用上述推导，即可得出理想情况下较为精确的时间复杂度。

2. 如前所述，上述推导中的系数 6 可以再做斟酌。实际情况中，绝大部分比较都至多用到两步函数比较，那么 6 可以减小为 4。

关于质数

我们之所以取质数作为模函数，有一个最大的好处，两个不同的数对两个质数同余时，即有性质：

a=b(mod p1)，a=b(mod p2)

当且仅当 a-b=0(mod p1*p2)

这个性质很好证明，笔者不再赘述。有了这个性质之后，当两个字符串对两个模函数的函数值分别相等时，他们完全相同的概率是很大的，因为它们的差能被 p1*p2 整除，而它们的差的量级很可能与 p1*p2 类似，这样只有差为零时才能满足要求。

上述性质对于三个质数同样成立，那么对于三个质数而言，三个函数值分别相等的话我们就有充分理由判定两个字符串一模一样，这样省去了逐个字符进行比较的麻烦，使程序速度大幅提升。

由前所述，程序的表现与质数的具体值并无直接关系，这是由简单一致分布决定的，而与质数的大小有很强的关联，具体来说，当质数越大时，计算机做计算的速度并没有显著下降，而建立链表需要执行的搜索次数却显著降低，这样的话我们倾向于取更大的质数。另一方面，质数具有稠密性，我们可以很容易的取到与 n 成正比的质数。

我们对计算机的存储位数进行估算后，取出较接近上限的 9 位的质数最为第一个质数。对接下来的两个质数，我们取 79555771 和 80000023。那么由上述素数的性质，我们知道在 k 值小于 40.1132(取对数得到)时是不会出现不同字符串有完全相同的模函数的，也就是这时的出错率为 0。而当 k 值比 40.1132 大时，由于相应的字符串的总数减少，经过我们的实际检验发现模型仍然保持了 0 出错率的优良性质。而计算机在对素数取模时，速度与质数的大小没有直接关系，这又保证了我们程序的高效。也就是我们尽量取很大的素数，不仅可以极大地增加准确率(100%)，又不会影响程序运行的速度。

进一步，当所取质数很大时，我们对程序进行优化，对于较小的 k 值，比如说 15 以下，由于转换而来的字符串四进制数没有超过所取质数，那么对于这部分 k 值不执行取模的操作，而是直接分配到相应的同余类，本质上来说这样做并没有影响，但是由于省略了一部分计算，对 k 值较小的情况大大提高了程序运行速度。

实践中不管 k 是多少每次查询费时都在 0.1 秒以下，反应出来的情况就是点下查询没有任何延迟的返回结果。而查询不需要任何额外空间。

由于 $4^{31}-1<2^{31}$,当 k<14 时，DNA 片段数值化后计算机整数类型可以直接存储下来，所以对于 k<14，我们做了一个特殊的处理，就是直接将 DNA 片段转化为 4 进制数进行比较，不需要对任何数取模，这可以减少建立索引的时间。

由于内存使用较少，我们简单的使用了并行计算来优化，使用了 2 个线程分别来建立 50 万条 DNA 序列的索引。查询时分别查询再合并结果。这样减少了计算时间，同时增加了内存使用。

我们测试了各个 *k* 建立索引的速度，所需要的时间如表 1 所示。

表 1

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1228	26	11215	51	9395	76	6823
2	1258	27	10755	52	9211	77	6849
3	1271	28	11141	53	8494	78	6819
4	1303	29	11055	54	8457	79	6687
5	1281	30	10613	55	9209	80	6561
6	1313	31	11641	56	8667	81	6547
7	1326	32	11613	57	8571	82	6255
8	1616	33	11913	58	8380	83	6099
9	1721	34	11476	59	8001	84	6200
10	5327	35	12055	60	8386	85	6188
11	10706	36	10626	61	7904	86	6005
12	9770	37	9502	62	8324	87	5869

13	8731	38	10511	63	8385	88	5853
14	11676	39	9829	64	7823	89	5762
15	11761	40	10005	65	7870	90	5637
16	11260	41	9737	66	7652	91	5632
17	11179	42	10577	67	7507	92	5560
18	11057	43	10900	68	7791	93	5411
19	12622	44	10240	69	7652	94	5370
20	11930	45	8700	70	7575	95	5604
21	11157	46	8608	71	7835	96	5301
22	11160	47	9548	72	7473	97	5279
23	11635	48	9965	73	7142	98	5158
24	10881	49	9301	74	7465	99	5057
25	11715	50	8320	75	7056	100	4966

可以看到 k=9 到 k=10 的时候时间上突然增加了不少，k=10 到 k=11 时间上又增加了不少，此后时间的变化趋于平缓。

经过我们的测试分析，k=9 的时候不同的 DNA 片段共有约 26 万种，k=10 的时候不同的 DNA 片段共有约 100 万种，k=11 的时候不同的 DNA 片段共有约 26 万种，k=10 的时候不同的 DNA 片段共有约 400 万种。

程序的运行时间由建立索引的主要操作与附加的一些操作组成，主要操作的时间与不同的 DNA 片段数量有关以及总 DNA 片段数，当 k<=9 时，主要操作的时间小等于附加操作的量级，所以时间较少。当 k=10 时，主要操作的时间超过了附加操作的量级，运行时间开始由主要操作主导。当 k=11 时，主要操作时间的量级继续上升,时间继续增大。之后不同的 DNA 片段数量的增加速度逐渐减小，所以时间上没有了突变。

为了测试是否在比较普遍的数据下这个程序的运行时间都差不多，我们生存了一组跟原数据大小相同的随机数据，即每个位置随机生成 ACTG 这 4 个字母中的一个。此外还生成了一组错位数据，即把原数据每个 DNA 序列的第 1 到 50 个字母截出，放到序列的最后，这样原来的前 50 个字母变成了后 50 个字母。时间如下

表 2

Random

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1231	26	16113	51	12543	76	7642
2	1271	27	16032	52	9769	77	7590
3	1382	28	15809	53	9820	78	7214
4	1406	29	15947	54	10152	79	7313
5	1443	30	15852	55	10169	80	7145
6	1467	31	15473	56	9993	81	7104
7	1333	32	11042	57	9437	82	6853

8	1566	33	11062	58	9170	83	6805
9	1627	34	11171	59	8842	84	6699
10	2322	35	11005	60	9165	85	6607
11	6275	36	10038	61	8968	86	6439
12	7828	37	11161	62	8805	87	6369
13	8791	38	11529	63	9247	88	6322
14	11612	39	11730	64	7977	89	6182
15	12214	40	10680	65	7807	90	6045
16	12786	41	10894	66	8372	91	5956
17	13353	42	11288	67	8255	92	5873
18	13045	43	11212	68	7794	93	5824
19	12917	44	11334	69	8020	94	5667
20	13842	45	11723	70	7930	95	5604
21	14754	46	13021	71	7793	96	5467
22	14755	47	12779	72	7774	97	5477
23	14732	48	12809	73	7810	98	5195
24	14120	49	12797	74	7718	99	5113
25	12070	50	12541	75	7530	100	5013

表 3

Reverse

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1173	26	10769	51	7955	76	6662
2	1252	27	9203	52	7800	77	6778
3	1284	28	9408	53	6488	78	6824
4	1288	29	7843	54	5533	79	6518
5	1275	30	7694	55	6204	80	5820
6	1300	31	7377	56	6594	81	6418
7	1327	32	8499	57	5172	82	6340
8	1578	33	9672	58	6081	83	6235
9	1762	34	8700	59	5932	84	5970
10	5749	35	9575	60	5632	85	5960
11	11179	36	8519	61	5485	86	6134
12	10440	37	8137	62	6083	87	5941
13	8816	38	7443	63	6954	88	5701
14	11677	39	7785	64	6589	89	5432
15	11823	40	7384	65	6420	90	5632
16	11437	41	7614	66	6627	91	5783
17	11144	42	6652	67	6343	92	5673
18	10192	43	7810	68	6184	93	5597
19	9409	44	7581	69	6285	94	5943
20	8863	45	6826	70	6457	95	5640

21	9688	46	6617	71	6266	96	5431
22	10266	47	6567	72	6510	97	5196
23	9929	48	6644	73	6908	98	5106
24	10647	49	6419	74	7226	99	5271
25	9979	50	7528	75	6819	100	5124

总结：

建立索引：

DNA 序列数 N=1000000,长度 L=100。

1.计算复杂度：O(NL)，其中 Len 为总数据长度 100*10^6=10^8。

2.空间复杂度：O(NL)

查询：

1.计算复杂度：O(k)。

2.空间复杂度：O(1)

性能评价：

表 4

查询速度	**复杂度 O(k)，由于 k 最大只有 100，可以看做 O(1),无论 k 是几每次查询都远小于 0.1 秒。**
索引内存使用	**O(NL)，只用了 1G 左右的内存。**
可支持的 k	**可支持一切 k。即 1 到 100。**
建立索引时间	**O(NL)。具体的时间已经列出。**

这个算法建立索引的算法流程如下：

1. 取出第一个 dna 序列 A，
2. k=0,i=0
3. 计算 A［i，i+k-1］的 F1，F2，F3 值
4. i＝i+1
5. 令 v=F1，取出第 v 个桶
6. 如果第 v 个桶中没有没比较过的元素，转第 9 步
7. 与桶中没比较过的一个元素比较 F2，F3，和字符串本身
8. 如果比较出不同，转第 6 步
9. 将 A［i，i+k-1］的位置等信息记录在第 v 个桶
10. 如果 i+k-1<=N，转第 4 步，否则结束。

这个算法查询的算法流程如下：

1. 读入 S
2. 计算 S 的 F1，F2，F3 值
3. 令 v=F1，取出第 v 个桶
4. 如果第 v 个桶中没有没比较过的元素，转第 7 步
5. 与桶中没比较过的一个元素比较 F2，F3，和字符串本身
6. 如果比较出不同，转第 8 步
7. 找到 S
8. 没找到 S，即不存在

这个算法建立索引的流程图 13 如下：
建立索引，对每个 DNA 序列 A：

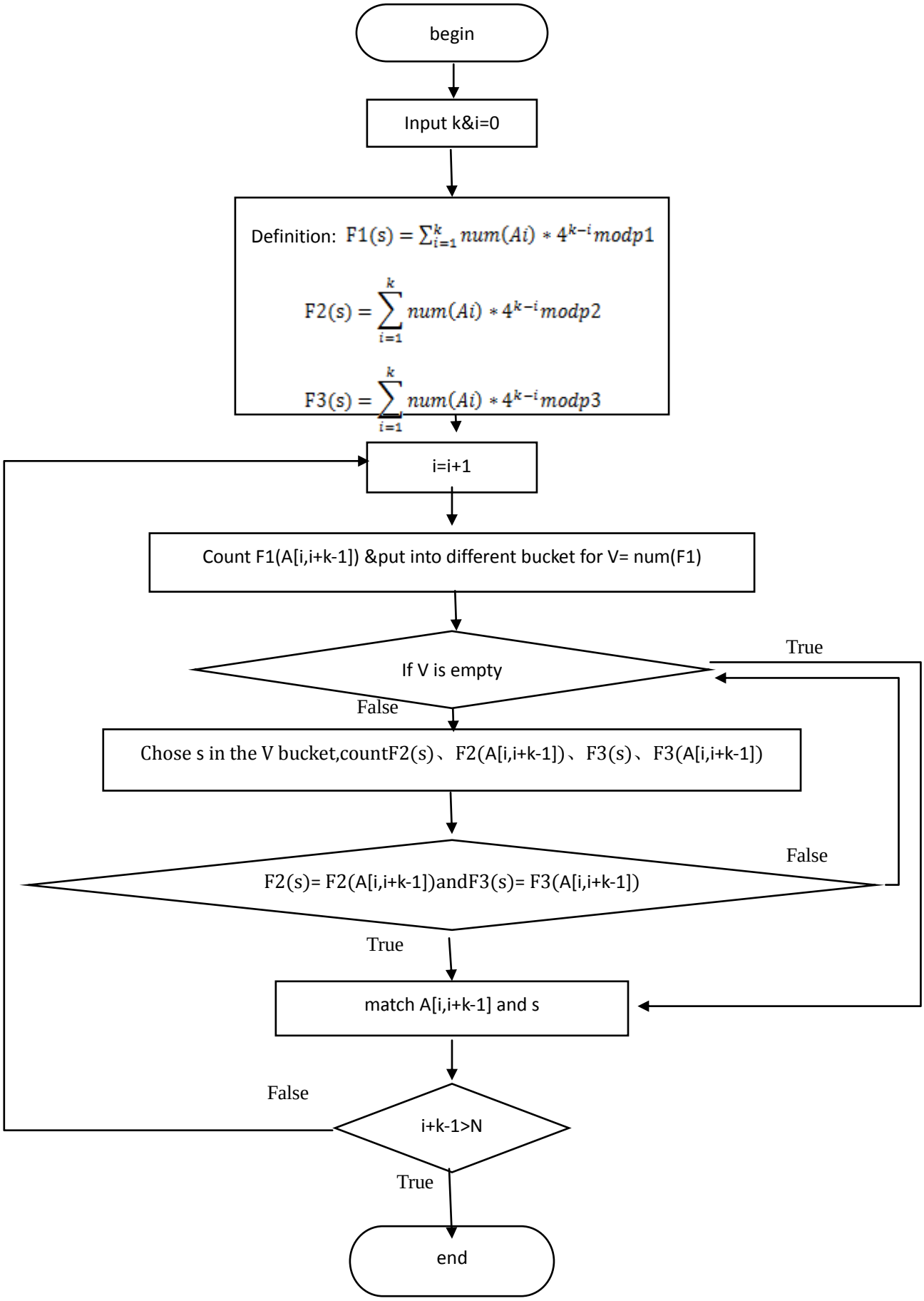
begin
Input k&i=0
Definition: $F1(s) = \sum_{i=1}^{k} num(Ai) * 4^{k-i} modp1$
$F2(s) = \sum_{i=1}^{k} num(Ai) * 4^{k-i} modp2$
$F3(s) = \sum_{i=1}^{k} num(Ai) * 4^{k-i} modp3$
i=i+1
Count F1(A[i,i+k-1]) &put into different bucket for V= num(F1)
If V is empty
True
False
Chose s in the V bucket,countF2(s)、F2(A[i,i+k-1])、F3(s)、F3(A[i,i+k-1])
F2(s)= F2(A[i,i+k-1])andF3(s)= F3(A[i,i+k-1])
False
True
match A[i,i+k-1] and s
i+k-1>N
False
True
end

图 13

需要对 100 万个 DNA 序列按照此流程图运作。
该算法查找的流程图 14 如下：

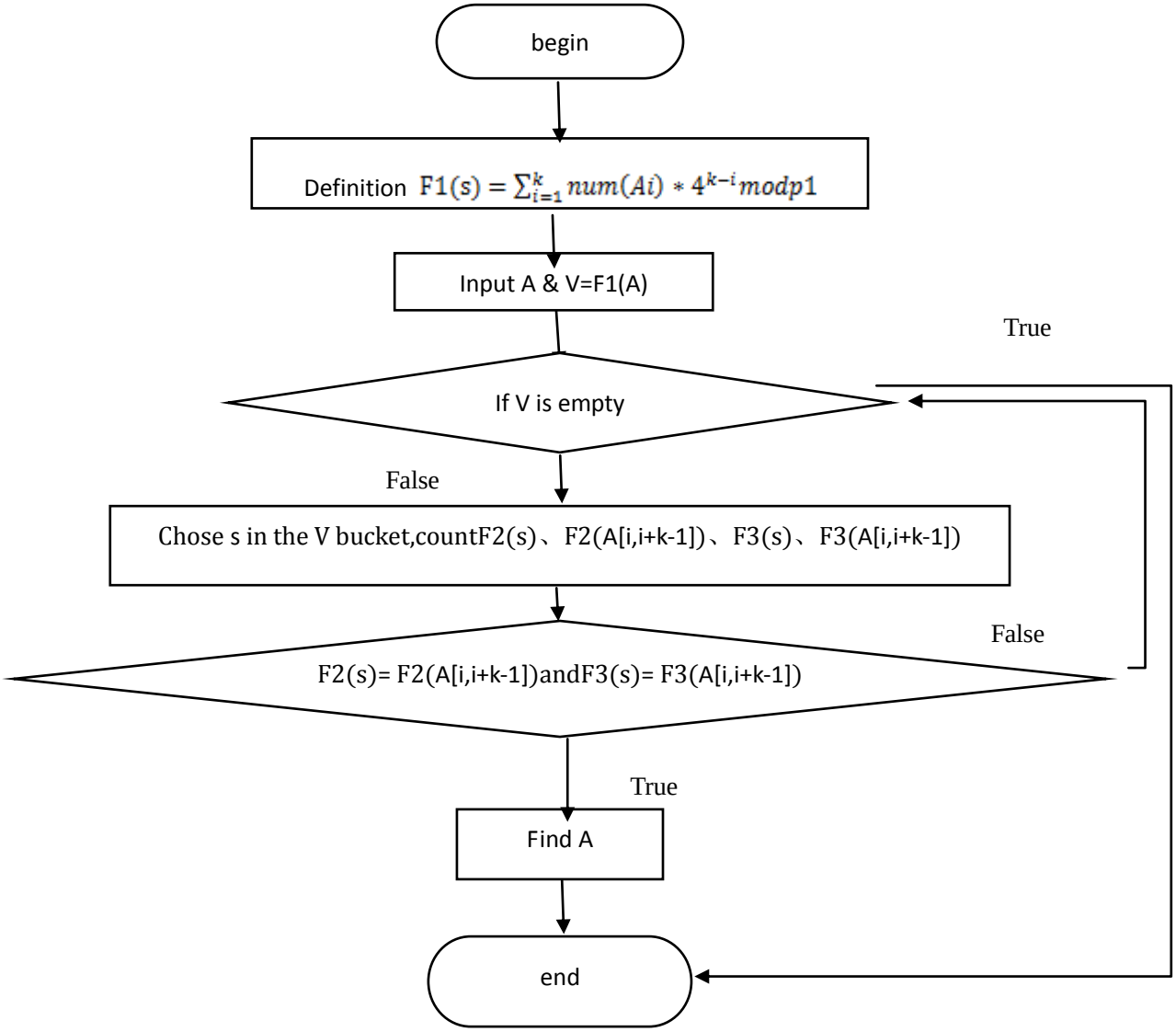

图 14

七、 多线程 Hash 函数算法

介绍：利用上述的 Hash 函数算法，并结合计算机程序多线程处理技术，对模型进行优化。

我们对正常数据，随机数据以及反转数据分别进行了测试，结果如下所示：

Usual:

表 5

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1069	26	7650	51	6568	76	4576
2	1052	27	7526	52	6244	77	4475
3	1075	28	7326	53	6127	78	4337
4	1136	29	7225	54	6100	79	4091
5	1043	30	7150	55	6324	80	4121
6	1163	31	7116	56	6004	81	4064
7	1069	32	7120	57	5898	82	4002
8	1253	33	6976	58	5925	83	3938
9	1440	34	6609	59	5640	84	3775
10	4174	35	6792	60	5660	85	3818
11	6311	36	6622	61	5774	86	3591
12	5296	37	6578	62	5392	87	3552
13	4562	38	6734	63	5547	88	3462
14	8273	39	6627	64	5264	89	3436
15	7845	40	7350	65	5287	90	3379
16	7764	41	7241	66	5368	91	3267
17	7860	42	7161	67	5204	92	3119
18	7638	43	7029	68	5073	93	3060
19	7862	44	7162	69	5104	94	2972
20	7753	45	6977	70	4988	95	2856
21	7786	46	6858	71	4802	96	2800
22	7665	47	6658	72	4901	97	2818
23	7598	48	6691	73	4621	98	2657
24	7547	49	6270	74	4602	99	2603
25	7315	50	6460	75	4685	100	2622

Random:

表 6

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1170	26	6156	51	6592	76	4801
2	1097	27	6154	52	6597	77	4742
3	1138	28	5896	53	6152	78	4461
4	1133	29	6039	54	6476	79	4505
5	1065	30	5975	55	6397	80	4256
6	1217	31	5951	56	6216	81	4238
7	1239	32	6081	57	6022	82	4216
8	1322	33	5803	58	6068	83	4059
9	1251	34	5960	59	6043	84	4060
10	1923	35	5475	60	6056	85	4010
11	3909	36	5639	61	5760	86	3887

12	4496	37	5380	62	5792	87	3699
13	5087	38	5236	63	5477	88	3750
14	7081	39	5562	64	5443	89	3677
15	6973	40	7010	65	5491	90	3645
16	6270	41	7136	66	5341	91	3570
17	6239	42	6754	67	5226	92	3412
18	6359	43	6826	68	5212	93	3305
19	6595	44	6790	69	5196	94	3045
20	6054	45	6819	70	4921	95	3102
21	6463	46	6640	71	5073	96	2987
22	5721	47	6618	72	4972	97	2869
23	6406	48	6715	73	4814	98	2689
24	6063	49	6472	74	4764	99	2555
25	6328	50	6420	75	4665	100	2569

Reverse:

表 7

k	时间/ms	k	时间/ms	k	时间/ms	k	时间/ms
1	1401	26	8125	51	6250	76	4361
2	1395	27	8054	52	6887	77	4116
3	1364	28	7952	53	6160	78	4176
4	1340	29	7773	54	6097	79	4009
5	1374	30	7721	55	5882	80	3977
6	1337	31	7870	56	5683	81	3868
7	1468	32	7773	57	5719	82	3821
8	1631	33	7564	58	5652	83	3656
9	1471	34	7498	59	5557	84	3626
10	4363	35	7466	60	5443	85	3565
11	6186	36	7370	61	5388	86	3494
12	5654	37	7130	62	5155	87	3575
13	4597	38	7185	63	5114	88	3361
14	7984	39	7003	64	5048	89	3301
15	8707	40	7290	65	4925	90	3239
16	8101	41	7460	66	4912	91	3191
17	8171	42	7543	67	4937	92	3205
18	8446	43	7557	68	4892	93	3041
19	8528	44	6944	69	4699	94	2968
20	8095	45	6740	70	4792	95	2897
21	9176	46	6759	71	4560	96	2820
22	8056	47	6287	72	4569	97	2757
23	7990	48	6342	73	4356	98	2745
24	8187	49	6423	74	4266	99	2658
25	8164	50	6225	75	4171	100	2657

八、 模型评估

1、 模型对比

表 8

	KMP 算法	字典树	后缀数组	Hash 算法	多线程 Hash
查询时间	1s 左右	1ms	1ms	1ms	1ms
查询空间复杂度	100MB	0	0	0	0
支持的 k	1-100	1-14	1-100	1-100	1-100
建立索引时间(k=5)	没有建立	3290ms	80s 左右	1043ms	1043ms
建立索引时间(k=10)	没有建立	20s 左右	80s 左右	5327ms	4174ms
建立索引时间(k=15)	没有建立	内存不足	80s 左右	11761ms	7845ms
建立索引时间(k=20)	没有建立	内存不足	80s 左右	11930ms	7753ms
建立索引时间(k=50)	没有建立	内存不足	80s 左右	8320ms	6460ms
建立索引时间(k=80)	没有建立	内存不足	80s 左右	6561ms	4121ms
建立索引时间(k=90)	没有建立	内存不足	80s 左右	5637ms	3379ms
建立索引时间(k=100)	没有建立	内存不足	80s 左右	4966ms	2622ms
建立索引所占空间	没有建立	当 k=5 时 4G 左右 当 k=10 时 6G 左右	2G 左右	1G 左右	2.5G 左右

2、 模型优点

(1) KMP 算法：空间使用非常小，不需要建立索引。并且不需要给定 k，可以随时查询不同长度的 DNA 序列。

(2) 字典树：查询速度快。充分利用空间换时间。

(3) 后缀数组：查询速度快。并且不需要事先给定 k。可以随时查询不同长度的 DNA 片段。

(4)Hash 函数算法：查询速度非常快，可以用瞬间回答来形容。空间使用虽然大于 KMP 算法，但只用了 1G 左右的内存，完全可以接受。完全可以适应几十万个几百万个查询，并实时返回结果。

3、 模型缺点

(1) KMP 算法：单次查询速度慢，需要 1 到 2 秒，从单次上看并不慢，但如果是大量的查询，如几十万个几百万个查询，这个速度是不可接受的。

(2) 字典树：空间开销巨大，不能支持所有的 k。

(3) 后缀数组：建立索引较慢。

(4) Hash 函数算法：需要事先给定 k。

4、 模型改进

我们对四种不同情况下的三个算法分别进行测试，增加 DNA 总数以及长度，分别对应为 100W-100,300W-100,100W-300 和 200W-200。

Hash 算法大数据运行时间

Hash N=3000000,L=100

表 9

k	Time/ms	k	Time/ms	K	Time/ms	k	Time/ms
1	3135	26	39096	51	26553	76	16107
2	3117	27	37410	52	26568	77	15653
3	3128	28	37321	53	23484	78	15740
4	3123	29	35686	54	23041	79	15485
5	3116	30	35018	55	22341	80	14900
6	3150	31	37648	56	22292	81	14753
7	3281	32	35760	57	23230	82	14431
8	3879	33	33785	58	25904	83	14283
9	4165	34	33238	59	21375	84	13906
10	16828	35	31893	60	21900	85	13565
11	35736	36	31013	61	21648	86	13442
12	40470	37	31416	62	20260	87	13160
13	36166	38	30923	63	20128	88	12956
14	52366	39	29963	64	19918	89	12791
15	45347	40	31516	65	18710	90	12730
16	44849	41	30245	66	18951	91	12509
17	45373	42	28489	67	18658	92	12396
18	46021	43	27835	68	18274	93	12308
19	42392	44	28639	69	17888	94	11908
20	43001	45	26859	70	17497	95	11738
21	42575	46	26972	71	17666	96	11627
22	41157	47	25924	72	16952	97	11421
23	40627	48	26791	73	16445	98	11152
24	40449	49	26586	74	16633	99	11094
25	41250	50	27955	75	15703	100	10945

Hash N=1000000,L=300

表 10

k	Time/ms	k	Time/ms	k	Time/ms
10	16911	110	48447	210	27701
20	70761	120	45035	220	27244
30	88362	130	42688	230	25343
40	72981	140	42876	240	24959
50	69474	150	39835	250	23738
60	72857	160	36207	260	22817
70	59326	170	34243	270	22167
80	62757	180	33108	280	21433
90	54347	190	32083	290	20648
100	48117	200	29998	300	19789

关于更大数据下 k 值可行性的展望

对于更大的数据量，多线程 Hash 算法不能达到问题要求。而 Hash 算法在 DNA 长度或者 DNA 数量分别变大时有不同的表现。当 DNA 数量扩大时，该算法在 k 很小(<10)以及很大(>90)时有较少的时间，但在中间部分有很大的突变，并且在 k=14 时达到 45 秒，在 k=14 两侧递减。这告诉我们当 DNA 数量扩张时，有少数几个极端的 k 值可以有较好表现。而在 DNA 长度增大时，我们可以看出各个 k 值对应的时间都有显著增加，说明数据增大对算法运行时间的影响很明显。

考虑各个算法在不同情况下的时间空间表现，其中未填的表格表示这个算法在该情形下无法满足空间要求。我们对三个算法分别进行排序。

表 11

	时间				空间
	查询	最大建立	最小建立	平均建立	
N=100 万，k=1~100					
KMP	0.5ms	无需建立索引			100M
Hash	<<1ms	12622ms	1271ms	7944ms	1G
多线程 Hash	<<1ms	8273ms	1043ms	5219ms	2.5G
N=300 万，k=1~100					
KMP	2s	无需建立索引			300M
Hash	1ms	52336ms	3116ms	23157ms	6.64G
多线程 Hash	内存不够，大于 8GB				
N=100 万，k=1~300					
KMP	2s	无需建立索引			300M
Hash	1ms	88362ms	16911ms	41208ms	6.68GB
多线程 Hash	内存不够，大于 8GB				
N=200 万，k=1~200					
KMP	2s	无需建立索引			300M
Hash	内存不够，大于 8GB				
多线程 Hash	内存不够，大于 8GB				

参考文献

[1]Cormen,Thomas;Lesiserson,Charles E.; Rivest,Ronald L.;Stein,Clifford (2001). "Section 32.4: The Knuth-Morris-Pratt algorithm". Introduction to Algorithms(Second ed.). MIT Press and McGraw-Hill. pp. 923–931.

[2] Black, Paul E. (2009-11-16). "trie". *Dictionary of Algorithms and Data Structures*.National Institute of Standards and Technology. Archived from the original on 2010-05-19.

[3]Manber,Udi;Mvers,Gene (1990). Suffix arrays:a new method for on-line string searches. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327.

[4]Knuth,Karl(1990).'The Art of Computer Programming'.3:Sorting and Searching(2nd ed).Addison-Wesley.pp.513-558.ISBN 978-0-262-53196-2

[5]罗穗骞 《后缀数组——处理字符串的有力工具》 IOI2009 国家集训队论文 2009 年