目录

DNA 位置检索

摘要

本文主要根据 k 的不同取值分别建立了三种索引方式的数学模型，从而实现对每一个给定的 *k-mer* 即键值能够快速找到其在 DNA 序列中的位置，即确立键值与 DNA 序列中位置一一对应的关系，并且尽可能的使“索引”所需的空间与时间最小。

首先我们对于碱基符号采用四进制编码，即每个碱基符号使用已规定的 2bit 数据表示。若不考虑任何限制因素，根据已有的数据结构相关知识，得到了初始模型。此时所有的键值是利用线性表进行存储，然后每一个键值分别对所有 DNA 序列中的每一个位置进行检索并输出。

在初始模型的基础上，我们根据 k 的不同取值得到了三种不同的优化模型。

当 k≤3 时，采用模型一。此时不对键值集进行存储，而是通过线性表中的序号实现与键值的配对；利用 13 个无符号字符型(unsigned char)对一条 DNA 序列的位置信息进行了存储。计算得到模型一的空间复杂度是13×4^k MB；时间复杂度$O(M\times(N-k+1))$。

当 k=3,…,11 时，采用模型二。在模型二中依旧不对键值进行存储；对于每条 DNA 序列的位置信息则通过 4Byte 的长整型(long int)空间进行存储。计算得到模型二的空间复杂度约为$O(4\times4^{k-10}+N-k+1)$MB；时间复杂度是$O(M(N-k+1)$。

当$k\geq8$时，采用模型三。模型三中给定一个 k 检索并利用四进制编码得到 M 条 DNA 中每一个位置所对应的 *k-mer* 即构成键值集 *H*; 并使用两个等长的数组分别存储检索得到的键值和相应位置，对于位置信息的存储采用模型二中的类似方法。计算得到模型三的空间复杂度是$(4+\left\lceil\frac{k}{4}\right\rceil)(N-k+1)$MB；时间复杂度是$O(n\times\left\lceil\frac{k}{4}\right\rceil+n)$。同时经过计算和实践运行程序得到，模型三适用于所有的 k 值，尤其在 k 值很大时首选模型三。

最后我们建立了基于空间复杂度与时间复杂度的评价函数，根据计算所得的理论数据与实际运行程序所得数据进行了比较分析。其结果与我们所建立的模型基本符合。

关键词：索引，四进制编码 ，折半查找法，空间复杂度，时间复杂度

一、问题重述

这个问题来自 DNA 序列的 *k-mer index* 问题。

给定一个 DNA 序列，这个序列只含有 4 个字母 *A*、*T*、*C*、*G*，给定一个整数值 *k*，从该 DNA 序列的第一个位置开始，取一连续 *k* 个字母的短串，称之为 *k-mer*， 然后从该 DNA 序列的第二个位置， 取另一 *k-mer*，这样直至该 DNA 序列的末端，就得一个集合，包含全部 *k-mer*。

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。也就是，对给定一个整数值 *k* 来说，当查询某一具体的 *k-mer*，通过这种数据索引方法，可返回其在 DNA 序列中的位置。

问题：

现在以文件形式给定 100 万个 DNA 序列，序列编号为 1-1000000，每个基因序列长度为 100 。

（1）要求对给定 *k*， 给出并实现一种数据索引方法，可返回任意一个 *k-mer* 所在的 DNA 序列编号和相应序列中出现的位置。每次建立索引，只需支持一个 *k* 值即可，不需要支持全部 *k* 值。

（2）要求索引一旦建立，查询速度尽量快，所用内存尽量小。

（3）给出建立索引所用的计算复杂度，和空间复杂度分析。

（4）给出使用索引查询的计算复杂度，和空间复杂度分析。

（5）假设内存限制为 8G，分析所设计索引方法所能支持的最大 *k* 值和相应数据查询效率。

（6）按重要性由高到低排列，将依据以下几点，来评价索引方法性能

- 索引查询速度
- 索引内存使用
- 8G 内存下，所能支持的 *k* 值范围
- 建立索引时间

二、问题分析

2.1 索引的理解

本题中的索引就是对每一个给定的 *k-mer* 即键值能够快速找到其在 DNA 序列中的位置，对于每一个 *k-mer* 都对应着一个位置集合，在数学上就是指键值与 DNA 序列中的位置集形成了一一对应的函数关系。

2.2 存储问题分析

存储方式的分类及优缺点

(1)顺序存储：用一组地址连续的存储单元依次存储线性表的数据元素。顺序存储的优点有存储密度大，存储空间利用率高；但缺点有存储空间需要预先设定，插入或删除数据元素都会引起大量的结点移动。

(2)链式存储（非顺序存储）：不需要用一组地址连续的存储单元来依次存储线性表的数据元素。非顺序存储的优点有存储空间不需预先设定，插入或删除数据元素时不会引起大量的结点移动。而缺点是指针域需要外加存储空间，对于任意结点的操作都要首先从开始指针顺链查找。

(3)定长存储：用一组地址连续的存储单元存储串值的字符序列，按照予定义的大小，为每个定义的串变量分配一个固定长度的存储区，串的实际长度可在这予定义长度的范围内随意，超过予定义长度的串值则被舍去。

(4)非定长存储：不同于定长存储方式的，不予先定义存储区的大小

因此可知顺序的定长存储可以实现快速检索和存储量大的功能，但本问题又有诸多限制：

Ⅰ.索引的建立过程是在线的（不可预知的）

Ⅱ.内容*c*的长度是不确定的，同时也是在线的。这说明，若采用定长存储，则需要预留足够的空间，不管今后需不需要，这些空间将长期占用。

2.3 检索问题分析

检索即查找是根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。而衡量某一检索算法的好坏通常以“其关键字和给定值进行过比较的记录个数的平均值”作为依据。

为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的**平均查找长度(*ASL*)**[1]

检索方式的分类及优缺点

（1）顺序查找：从表中最后一个记录开始，逐个进行记录的关键字和给定值的比较。顺序查找的优点是算法简单且适应面广，缺点是查找效率较低。

（2）折半查找：先确定待查记录所在的范围（区间），然后逐步缩小范围直到找到或找不到该记录为止。折半查找的优点是查找效率较高，缺点是只适用于有序表，且限于顺序存储结构（对线性链表无法进行折半查找）

三、假设与符号说明

3.1 假设

假设一： DNA 序列中的碱基标识只有 A、C、G、T 四种

假设二： 系统内存中运行的其他程序对本程序没有影响

3.2 符号说明

符号	符号说明
k	*k-mer* 的长度
h	键值即一个具体的 *k-mer*
c	每一个键值 *k-mer* 的所有位置构成的集合，称作内容 c
f	键值 h 与内容 c 形成的一一对应关系
H	所有出现的 *k-mer* 构成的集合
C	所有键值 h 对应的内容 c 构成的集合
M	DNA 序列的个数,即 1000000
N	每个 DNA 序列的长度，即 100
i	DNA 序列的编号，$1 \le i \le M$
j	每一条 DNA 序列中的位置，$1 \le j \le N$
p	键值集 H 中元素的总个数，即所有出现 *k-mer* 的总个数
t	键值 h 按照递增顺序排列后的序号，$1 \le t \le p$
x	在某一条 DNA 上出现某 *k-mer* 的次数
T	时间复杂度
S	空间复杂度
n	算法问题的规模大小，即$n = M(N - k + 1)$
Y	模型评价参数

四、索引的数学定义

本题目的要求是建立一个 DNA 序列中 *k-mer* 的索引，并能实现利用该索引的快速查找。所谓索引，在数学上可以理解为“键值-内容”构成的函数，即：

$$F^*: \{A, C, G, T\}^k \to 2^{\{1,\dots,M\}\times\{1,\dots,N-k+1\}}$$

$$h \longmapsto c = F^*(h)$$

其中集合$\{A,C,G,T\}^k$表示所有可能的 *k-mer* 构成的集合，并且称其中的元素 h 为键值。集合$\{1,\dots,M\}\times\{1,\dots,N-k+1\}$表示 *k-mer* 在 DNA 序列中可能出现的位置所构成的集合，具体的：

$$(i,j) \in \{1,\dots,M\}\times\{1,\dots,N-k+1\}$$

表示第 i 个 DNA 序列的第 j 个位置，并称(i,j)为一个“位置”。而幂集$2^{\{1,\dots,M\}\times\{1,\dots,N-k+1\}}$则为$\{1,\dots,M\}\times\{1,\dots,N-k+1\}$的所有子集构成的集合，也就是说，某一个键值 h 的像 c 是一个由若干个位置构成的集合。

可以证明，

$$F^*: \{A, C, G, T\}^k \to 2^{\{1,\dots,M\}\times\{1,\dots,N-k+1\}}$$

可能并不是一满射，集合$\{A,C,G,T\}^k$的一个子集与幂集$2^{\{1,\dots,M\}\times\{1,\dots,N-k+1\}}$的一个子集构成一一映射，因此对函数 *F**进行初步优化，得到：

$$F: \mathrm{H}_0 \to C_0$$

$$h \longmapsto c = F(h)$$

其中

$$H_0 = \{\text{所给 DNA 序列中包含的所有} k-mer\}$$

$$C_0 = \{1,\dots,M\}\times\{1,\dots,N-k+1\}$$

4.1 键值集 H 的表达

4.1.1 键值 h 的表达

对于每一个数据元素 h 即键值采用四进制编码[2]，每个碱基符号使用 2bit 数据表示：

$$A-00 \quad C-10$$

$$T-01 \quad G-11$$

同时，对键值 h 进行编码。

例如：*ATCGA* 编码为 00 11 10 01 00

而这种将 *k-mer* 序列映射为整数关键字[3]的算法也已经有人使用过。

4.1.2 键值集 H 的存储方式

键值集 H 是集合 $\{A,C,G,T\}^k$ 的子集,同时键值集 H 中的元素是本题给定的 M=1000000 个长度为 N=100 的 DNA 序列中所有出现的 *k-mer*，易得键值集 H 中的元素个数 p：

$$p \le M(N-k+1)$$

又根据碱基的种类只有 4 种 A、T、C、G，所以：

$$p \le 4^k$$

于是可以得到：

$$p \le min\{4^k, M(N-k+1)\}$$

通过计算解得，当 k 略大于 13 的时候两者持平($k \approx 13.19390884$)：

$$4^{13} = 67108864 < M(N-13+1) = 88000000$$

即

$$\begin{cases} 4^k < M(N-k+1) & k \le 13 \\ 4^k > M(N-k+1) & k > 13 \end{cases}$$

考虑程序实现各方面方便，我们可以令：

$$p=\begin{cases}4^k & k\le 13\\ M(N-k+1) & k>13\end{cases}$$

于是我们分别对键值集 H 中元素个数的不同进行探讨，从而采用不同的存储方式。

4.2 位置集 C 的表达

4.2.1 内容 c 的表达

方式一：集合形式$C=\{c|c\subset\{1,\dots,M\}\times\{1,\dots,N-k+1\}\}$

例如：$c_h=F(h)=\{(i_1,j_1),(i_x,j_x)\}$

例如：$c_h=F(h)=\{(i_1,j_1),(i_x,j_x)\}$，将$c_h$用如下等价$0-1$矩阵$(c_{ij})_{M\times(N-k+1)}$形式来表达，其中矩阵中未显示的元素均为 0，且$c_{ij}$满足：

$$c_{ij}=\begin{cases}1,(i,j)\in c_h\\ 0,(i,j)\notin c_h\end{cases}$$

$c_h=$

	j_1	…	j_x	..	j_{N-k+1}	
	1					i_1
						.
						.
						.
			1			i_x
						.
						.
						i_M

图 1

每一个键值 h 的位置集都可以用如上等价$0-1$矩矩阵来表示，这类的$0-1$矩阵分为三类，其中有两类比较特殊分别是[4]：

（1）非 **0** 元素占所有元素比例较大的矩阵称为稠密矩阵

（2）非 **0** 元素占所有元素比例较小的矩阵称为稀疏矩阵

根据简单的分析可得：

当$k \le 2$时，c是稠密矩阵；

当$3 \le k \le 11$时，c既不是稠密矩阵也不是稀疏矩阵；

当$k \ge 12$时，c是稀疏矩阵。

4.2.2 位置集 C 的存储方式

在模型一中 C 使用连续的$4^k \times M \times 13$Byte 空间来存储内容 c，在模型二中使用4^k个指针指向键值相应的内容 c，指向的内容 c 采用若干个长度为$\left\lceil \frac{M(N-k+1)}{4^k} \right\rceil$的线性表来存储。在模型三中使用连续的$M(N-k+1)$长整型空间来存储内容 c。前两者存储方式所需的空间是随着k呈指数级方式增长的，而模型三就其内容 c 来说是单调递减的。

4.3 函数 f 的表达

键值集 H 和位置集 C 的对应关系在模型一和模型二中我们借助线性表中的位置与键值的一一对应关系来实现的；在模型三中直接存储键值和相应的内容实现的。

五．模型的建立分析及优化

5.1 初始模型

若不考虑任何限制因素，则根据已有的数据结构相关知识，极易得到初始模型。

5.1.1 键值集 H 的构建

将 H 用线性表的形式存储，线性表的构建方法：

（1）采用顺序存储的方法

（2）对于每一个数据元素即键值采用四进制编码，并且线性表中的每个数据元素都有一个确定的位置。

（3）若某一个键值编码不存在于线性表中，我们采用折半插入的方法将键值的编码值插入到相应位置。折半插入是在一个有序表中通过折半查找的方法来查找再插入。

5.1.2 位置集 C 的构建

最简单也是最容易的方法，即对于出现的每一个键值 *k-mer*，我们都分别对 *M* 条 DNA 序列的 *N-k*+1 个位置进行检索，当检索到到该 *k-mer* 时，返回其在 DNA 序列中的位置 (i, j)。

初始模型的结构如图 2 所示：

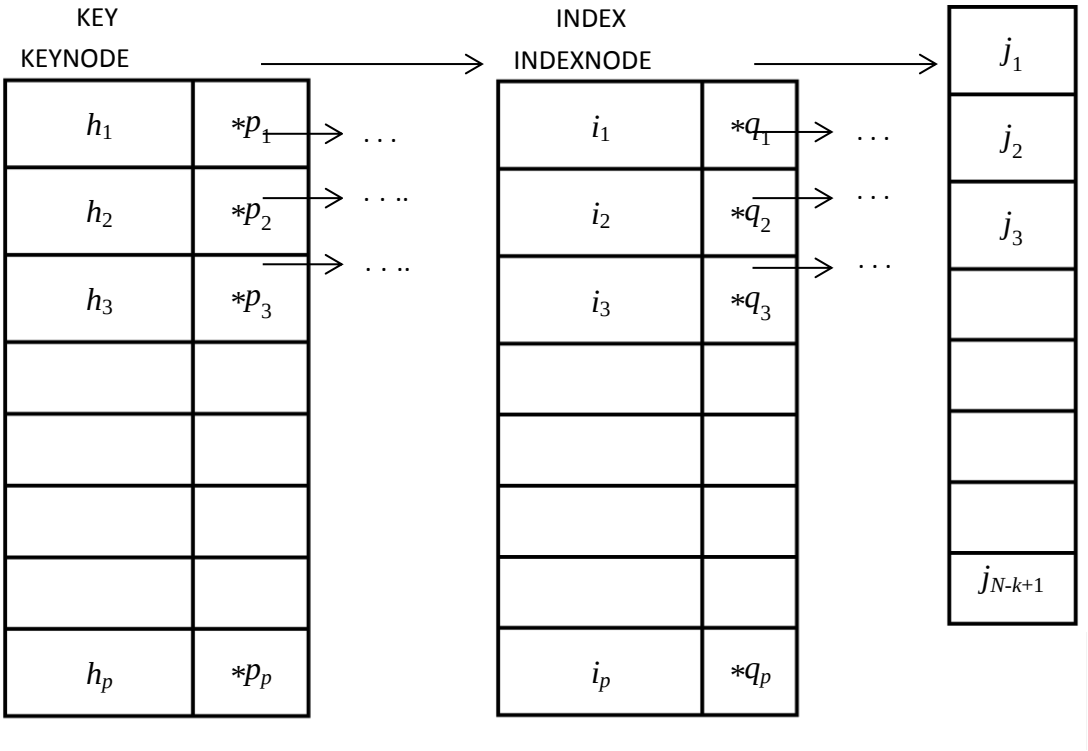


图 2　初始模型结构图

5.2 模型一：完全顺序存储/0-1 矩阵

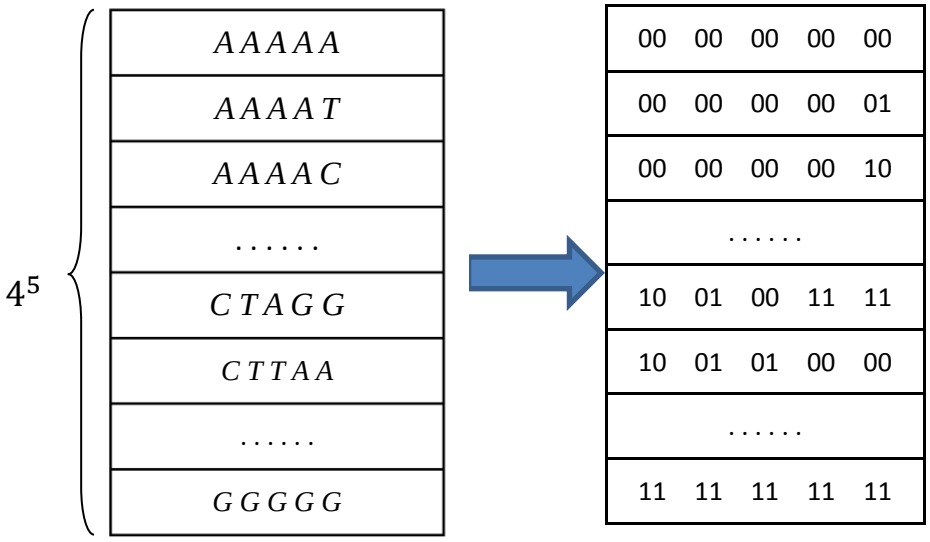
5.2.1 键值集 *H* 的构建

键值集 *H* 中的元素采用四进制编码，可以得到每一个 *k-mer* 所对应的整数，然后将这些整数按递增的顺序进行排列并标记序号 t $(1 \le t \le p)$，便实现序号 t

和键值 h 的配对,因此无需对键值集 H 进行存储，该方式适用于

$4^k \ll M(N-k+1)$，即$\|H\| \ll \|C\|$。

例如$k=5$:



5.2.2 位置集 *C* 的构建

为了避免位置集 C 存储空间的大量浪费和检索的繁琐，因此我们对初始模型进行优化得到模型一，模型一适用于同一 i 有多个 j 满足的情况。

位置集 C 中的每个数据元素即内容 c 中均包含 M 个元素，又

$$\left\lceil \frac{100}{8} \right\rceil = 13$$

于是每个元素可以用 13 个无符号字符型(unsigned char)进行存储， 即

$$13 \times 8\text{bit} = 104\text{bit}$$

因为对于每一个内容 c 中都顺序存储了 M 条 DNA 序列的信息,于是通过标号便可以知道所在的 DNA 序列数即 i。

每一个 DNA 序列的长度为 N，所以我们可以用 100bit 进行存储该条 DNA 序列的信息，并通过具体计算得到 *k-mer* 在该条 DNA 上的位置信息即 j。

例如：*k-mer*　*ATCGA*

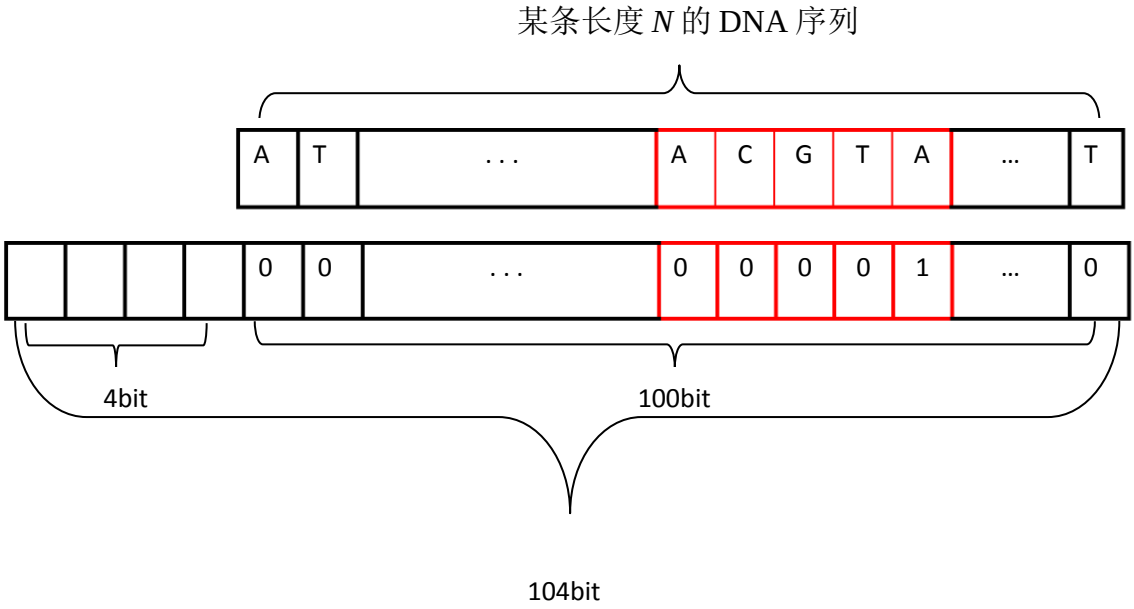


图 3　模型一位置集 C 的构建

模型一的结构如图 4 所示：

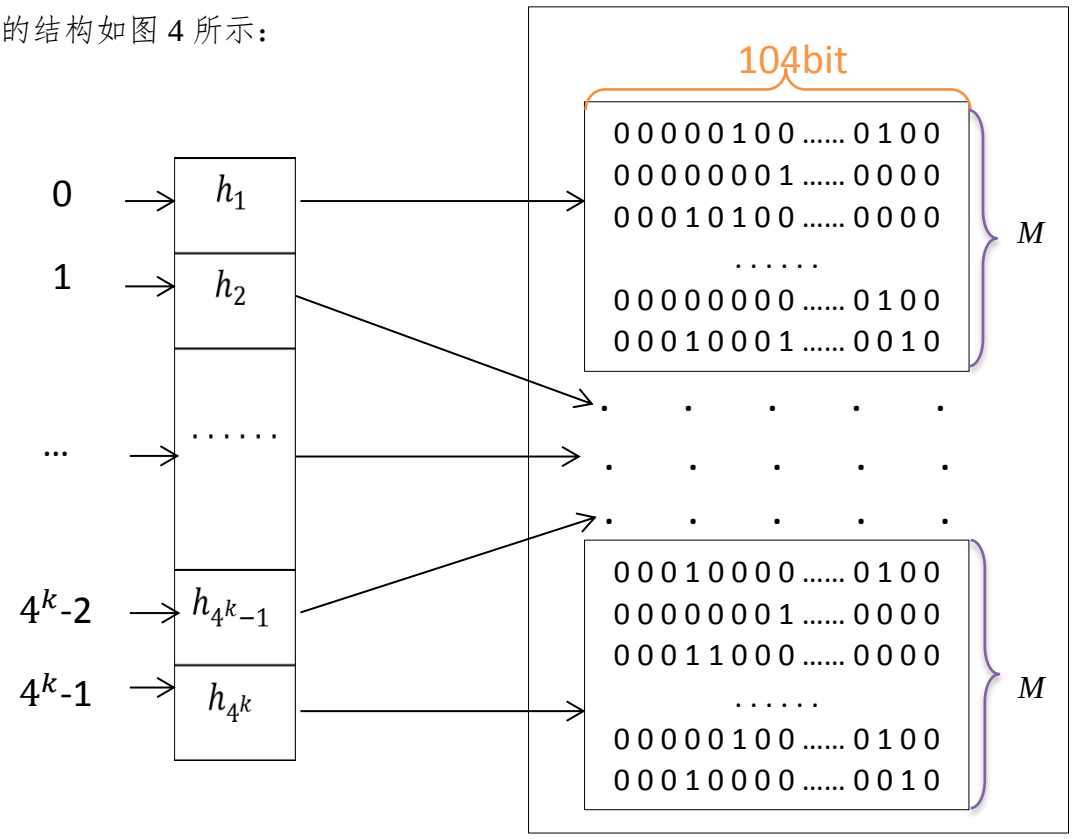


图 4　模型一结构图

5.2.3 事前分析

5.2.3.1 空间复杂度

算法的**空间复杂度**[1]即为算法所需存储空间的量度，记作：

$$S(n)=O(f(n))$$

由以上分析得到模型一适用于 k 很小的时候，所以此时$p=4^k$，因此模型一的空间复杂度：

$$S_1=4^k\times M\times 13\,\text{Byte}$$

$$=13\times 4^k\,\text{MB}$$

5.2.3.2 时间复杂度

一个算法是由控制结果（顺序、分支和循环三种）和原操作（指固有的数据类型的操作）构成的，则算法时间取决于两者的综合效果。为了便于比较同一问题的不同算法，通常的做法是，从算法中选取一种对于所研究问题（或算法类型）来说是基本操作的原操作，以该基本操作重复执行的次数作为算法的时间度量。

一般情况下，算法的基本操作重复执行的次数是模块 n 的一个函数 $f(n)$，因此，算法的时间复杂度记作：

$$T(n)=O(f(n))$$

它表示随着问题规模 n 的增大，算法执行时间的增长率和 $f(n)$的增长率相同，称作算法的渐进时间复杂度（*Asymptotic Time Complexity*），简称**时间复杂度**[1]。

时间复杂度是总运算次数表达式中受 n 的变化影响最大的那一项（不含系数）。本次的时间复杂度计算我们将考虑在最坏情况下的时间复杂度，即分析最坏情况以估计算法执行时间的一个上界。

因此我们分析得到模型一的时间复杂度，在 *main* 函数中，模型一的时间复杂度主要体现在函数 *InsertKEY* 中，详细程序见附录，其中循环次数有：

(*5)循环次数：　$n=4^k$

(*6)循环次数：$N+k-1$

模型一的建立比较直观，一共有 4^k 个 *k-mer*，我们需要从第一串 DNA 序列的第一个位置移动，然后在该 *k-mer* 的 0-1 矩阵中的相应位置标记为 1，直至第

M 串 DNA 序列的最后一个位置。根据具体的算法中，得到建立索引的时间复杂度为：

$$T_1(k)=O(M(N-k+1))$$

而检索的时间复杂度为：

$$O(1)$$

5.3 模型二：完全顺序存储/坐标集

5.3.1 键值集 *H* 的构建

对于键值集 H 我们采用与模型一相同的方法

5.3.2 位置集 *C* 的构建

我们考虑到极有可能会出现一个 *k-mer* 在一条 DNA 序列上出现次数极少的状况，于是我们又一次对模型优化。

位置集 C 中的每个数据元素即内容 c 中均包含 M 条 DNA 序列中的位置信息，其中每条 DNA 序列的位置信息通过 4Byte 空间进行存储， 即

$$4\times 8=32\,\text{bit}$$

因为共 M 条 DNA 序列，由

$$\lceil \log_2 M \rceil = 20$$

$$2^{20}=1048576>1000000$$

所以我们可以用 20bit 空间存储 DNA 的序列数；

又

$$\lceil \log_2 N \rceil = 7$$
$$2^7 = 128 > 100$$

可知 7bit 空间便足够存储该 DNA 序列上的一个位置信息，$(32-20)\div 7=1.....5\,\text{bit}$

此时该 4Byte 空间上最多可以存储一个位置信息，规定此种形式为结构一。

但当 k 小的时候一定会出现某键值 h 在同一条 DNA 序列上的位置超过一处的情况，于是根据不同 DNA 序列上位置信息的个数相应的增加以 4Byte 为一个单位的存储空间。此时增加的 4Byte 空间全部用来存储位置信息，$32\div7=4.....4$ bit 即一单位的 4Byte 空间最多可以存储 4 个位置信息，规定此种形式为结构二。

结构一和结构二还包括 1bit 的控制位和 3bit 计数位（记录结构二 4Byte 空间中已存储位置信息的个数）。

进一步说明：

（1） 1bit 控制位：每个 4Byte 空间的首位作为控制位。结构一的控制位存储 0；结构二的控制位存储 1。

（2） 3bit 计数位：利用二进制编码记录一单位 4Byte 空间中位置信息的个数（可知位置信息个数的可能性只有 1,2,3,4），于是规定：

1－000　　2－001

3－010　　4－011

注：结构一有且仅有一个位置信息，无需记录已存储位置信息的个数，所以 3bit 计数位为 000。

（3）结构一、结构二的关系：内容 c 中关于每条 DNA 序列位置信息的记录一定包含结构一，而包含结构二的个数则根据其包含的位置信息个数确定。并且结构二一定紧跟在结构一之下。

模型二中对 (i, j) 的压缩存储，节省了相同 i 的存储空间，又比“初始模型”节省了指针空间的开销。

上述 4Byte 空间的结构一与结构二如图 5, 图 6 所示：

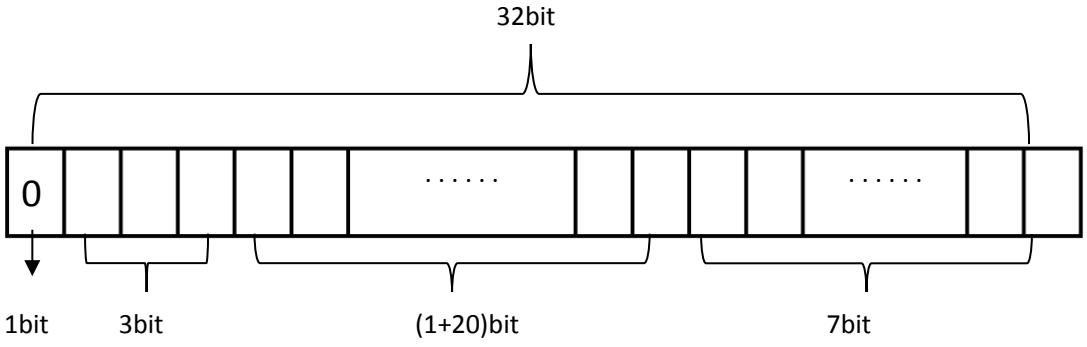


图 5　4Byte 空间结构一

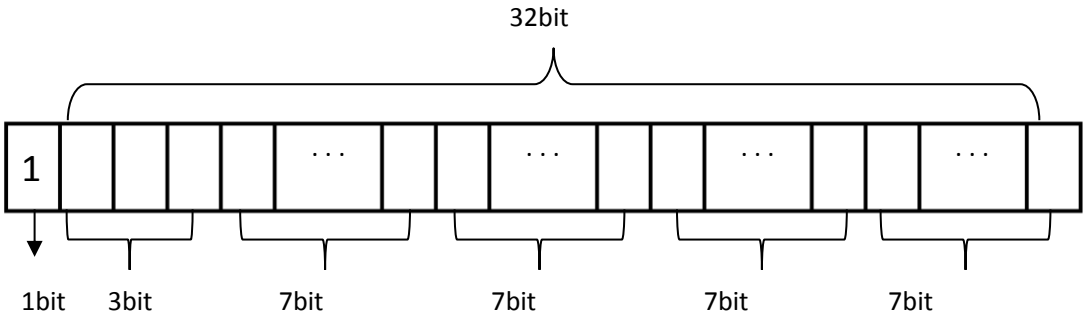


图 6　4Byte 空间结构二

模型二的结构如图 7 所示：

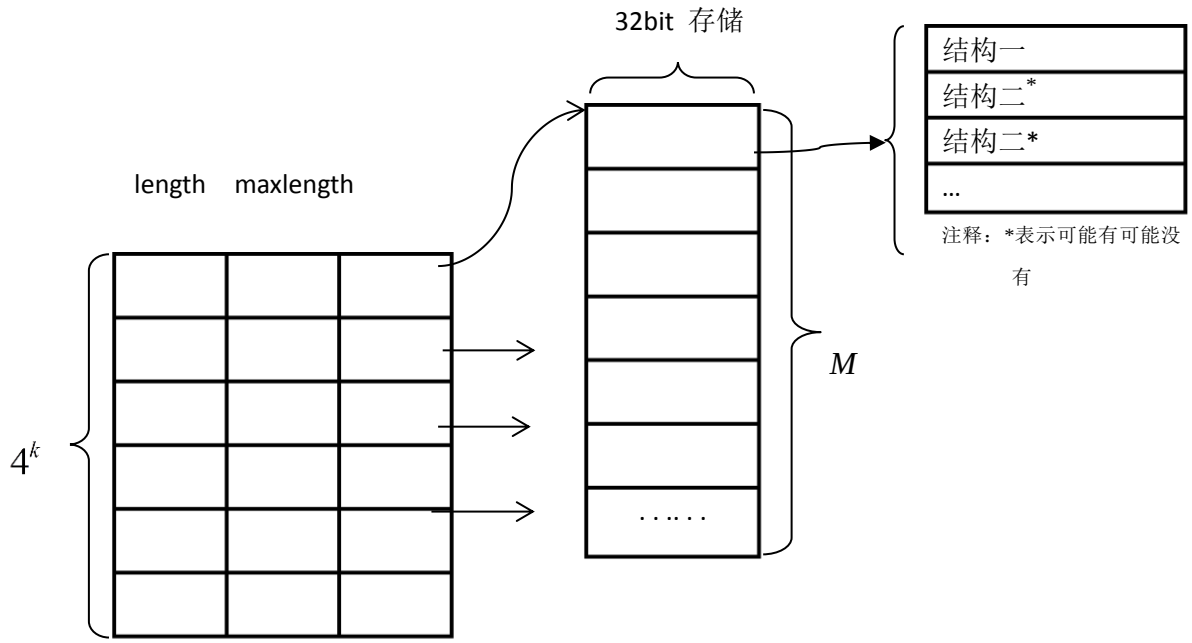


图 7　模型二结构图

5.3.3 事前分析

5.3.3.1 空间复杂度

根据之前对模型一的分析可得模型二的空间复杂度，其中模型二中$p = 4^k$：

$$S_2 = 4^k \times 4 + M(N-k+1) \times 4 \text{ Byte}$$

$$\approx 4 \times (4^{k-10} + N - k + 1) \text{ MB}$$

5.3.3.2 时间复杂度

模型二是在模型一的基础上对 *C* 的存储进行优化，继而对 *C* 的存储时间复杂度的变化体现在函数 *InsertKEY* 中，详细程序见附录，其中循环次数有：

(*7)的循环次数：*k*-1

(*8)的循环次数：4^k

(*9)的循环次数：*N*

因此模型二建立索引的时间复杂度为：

$$T_2(k) = O\big(M(N-k+1)\big)$$

而检索的时间复杂度为：

$$O(1)$$

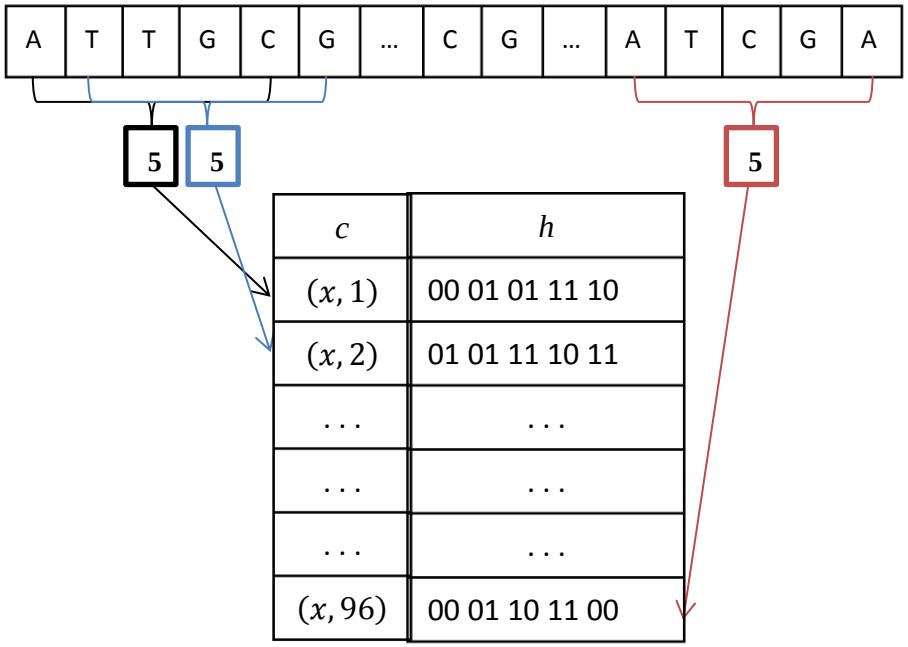
5.4 模型三：完备顺序存储/坐标集

运行模型一、二时我们发现键值在插入过程中因为使用了插入排序法导致在建立索引的时候会花费大量的时间，因此再一次优化得到了模型三。

5.4.1 键值集H_0的构建

给定一个 k 检索并利用四进制编码得到 M 条 DNA 中每一个位置所对应的 *k-mer* 即构成键值集H_0。

例如：第 x 条碱基序列，且$k=5$



5.4.2 位置集C_0的构建

根据键值集的构建方式，易知位置集C_0*映射*键值集H_0是一个满射，是一个单值函数；进行逆运算，由键值集H_0 →位置集C_0，可能是一个多值函数。

$$C_0 = \{1, \dots, M\} \times \{1, \dots, N-k+1\}$$

$$H_0 = \{\text{所给}DNA\text{序列中包含的所有}k-mer\}$$

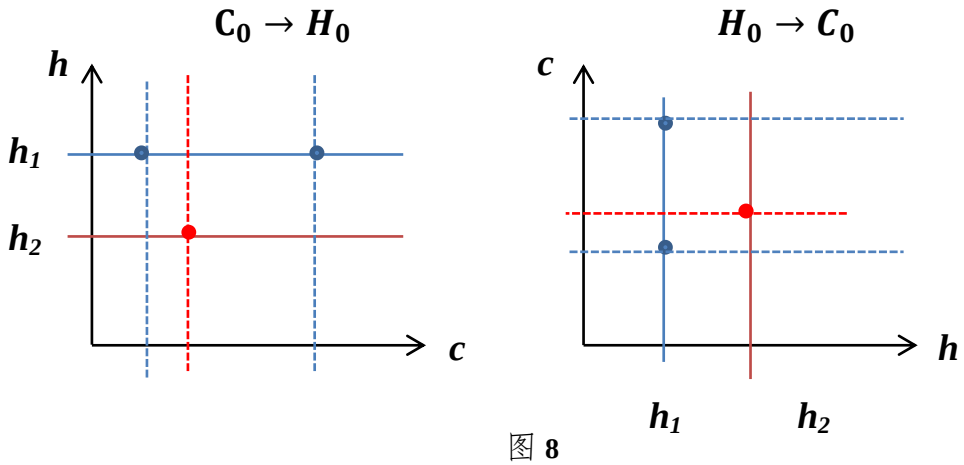


图 8

我们使用两个等长的数组分别存储检索得到的键值和相应位置，对于位置信息的存储采用模型二中的 32bit 空间结构一。相同键值重复计数则两个数组一一对应，再进行二路归并排序且保持之前对应关系不变。

模型三的结构如图 9 所示：

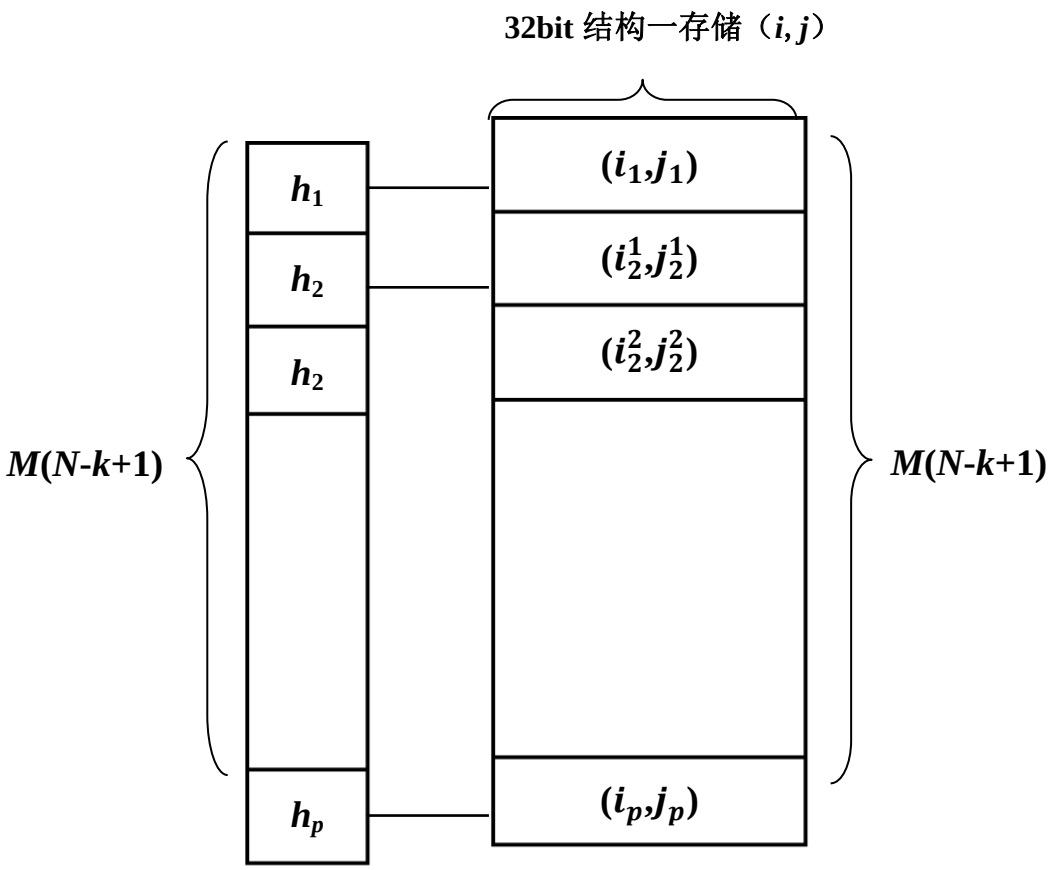


图 9　模型三结构图

5.4.3 程序设计思路

模型三是在 *k* 值增大时进行的改进，我们主要针对模型三的程序设计进行解释。

模型三的建立我们充分考虑了时间复杂度和空间复杂度，分别在编码、解码 *k-mer*，键值 *H* 的归并排序和查找 *k-mer* 所处区间段进行优化。

5.4.3.1 编码、解码 *k-mer*

在对一串 DNA 序列编码的过程中，核心操作是利用位运算将字符类型的 *A*、*C*、*G* 、*T* 分别换作四进制 00、01、10、11 进行记录，这样既节省了内存空间，又方便程序的编写。在申请空间时，我们申请了 *m* 个无符号字符型，其中$m = \left\lceil \frac{k}{4} \right\rceil$。我们采用滑动的解码方式，即利用已解得的上一个 *a.kmer*，然后向右滑动一个位置得到新的编码，也就是说可以将 *a.kmer* 位运算左移两个位置再与新字符的编码进行按位与运算。设编码函数 *f*(*X*)，则解码过程与编码过程互为逆运算，即：

$$g(r) = f^{-1}(X)$$

编码函数 *f*(*X*)如下：

$$f(X) = \begin{cases} 00, X = A \\ 01, X = C \\ 10, X = G \\ 11, X = T \end{cases}$$

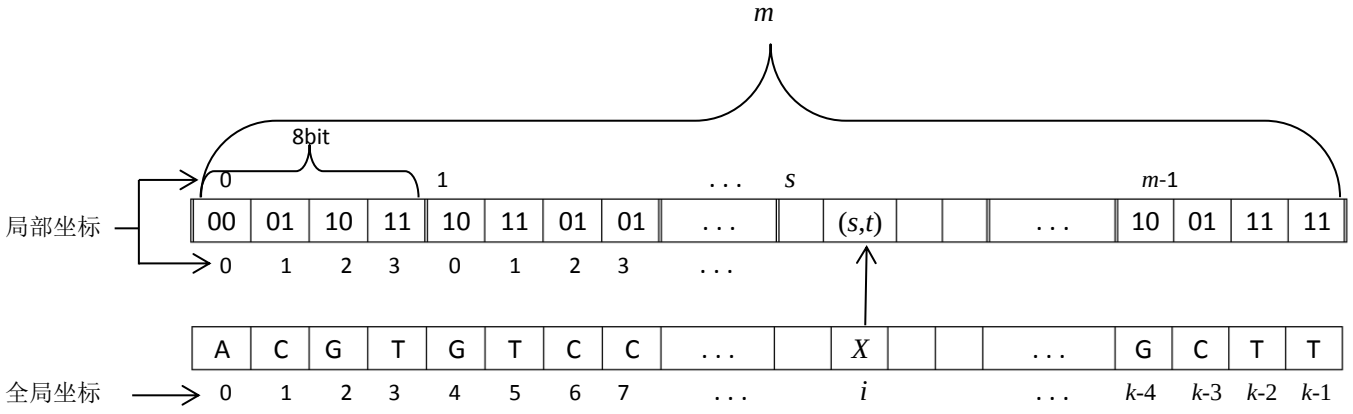
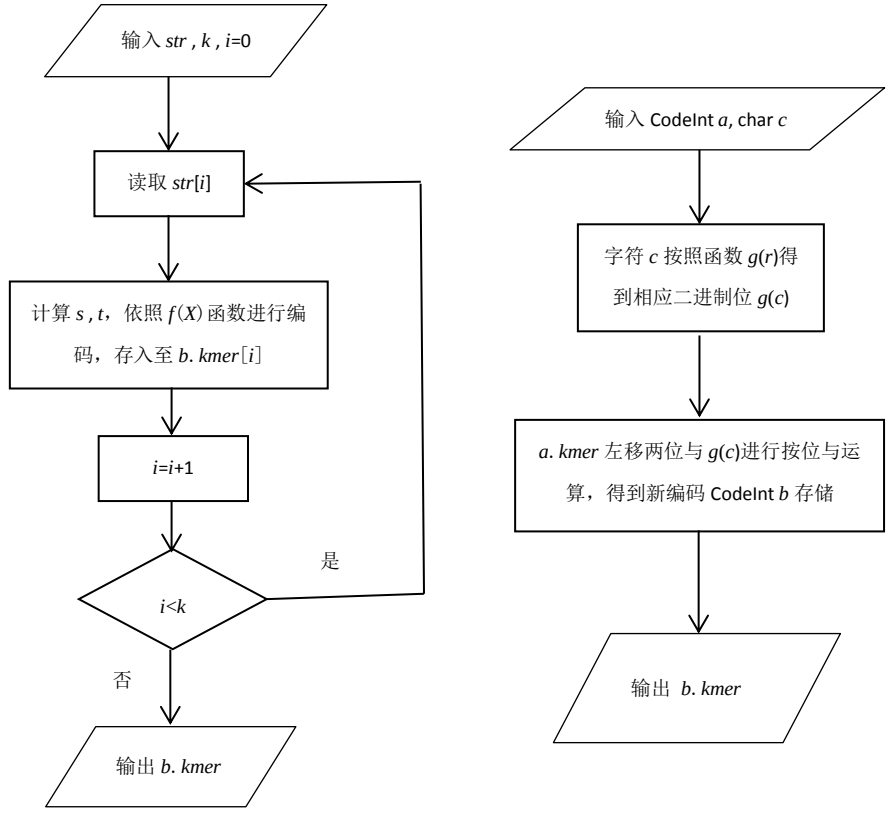


图 10　编码结构



(a) (b)

图 11　编码及窗口滑动流程图

设字符 X 的全局坐标为 i，局部坐标为 (s,t)，其中 $0 \le s < m, 0 \le t < 4, 0 \le i < k$。下面为其互换公式：

$i \to (s,t)$：

$$\begin{cases} s = \left\lfloor \frac{i}{4} \right\rfloor \\ t = i \bmod 4 \end{cases}$$

$(s,t) \to i$：

$$i = 4s + t$$

详细程序见附录三。编码及窗口滑动流程图如图 11，解码流程图如图 12。

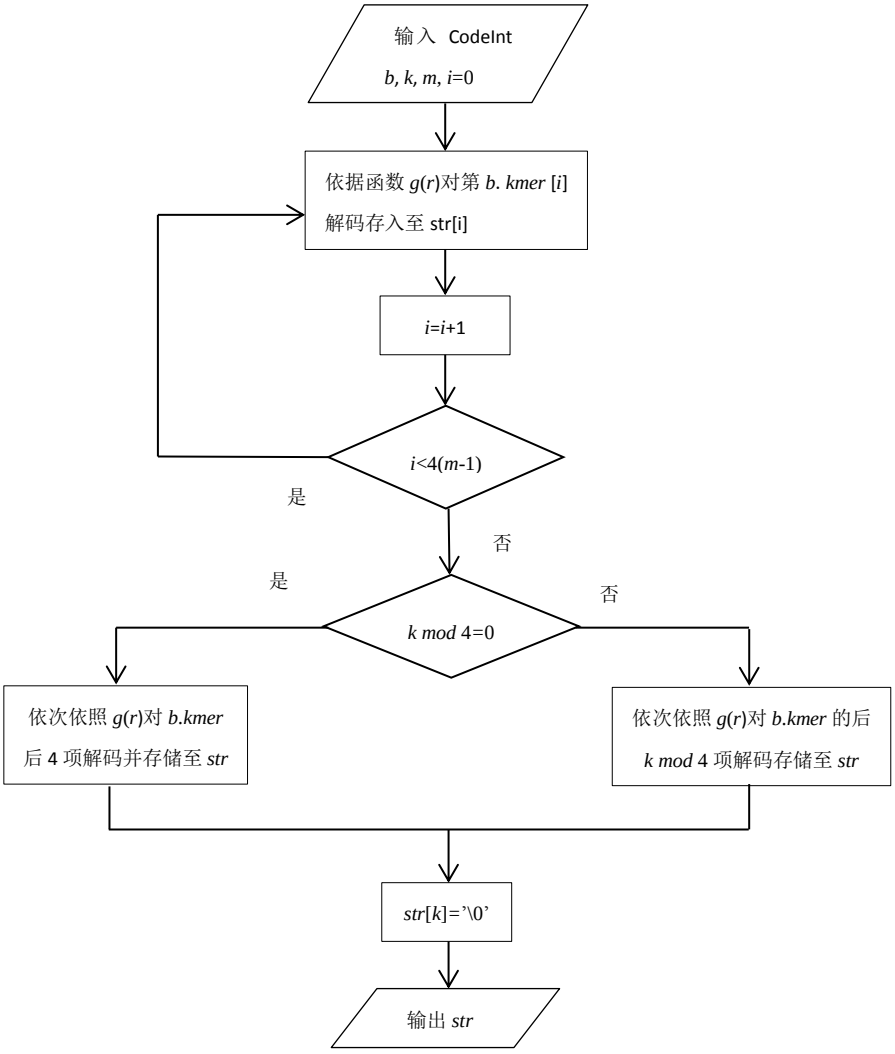


图 12　解码流程图

5.4.3.2 键值 *H* 的归并排序

归并排序是建立在归并操作上的一种有效的排序算法，将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为**二路归并**。假设初始序列含有 n 个记录，则可看成是 n 个有序的子序列，每个子序列的长度为 1，然后两两归并，得到$\left\lceil \frac{n}{2} \right\rceil$个长度为 2 或 1 的有序子序列；再两两归并，……，如此重复，直至得到一个长度为 n 的有序序列为止。假设两个有序表的长度分别为 m 和 n,无论是顺序存储结构还是链表存储结构，都可在$O(n\log_2 n)$的时间量级上实现。而在这次的操作中，我们同时对两个数组进行归并排序，其中数组 *sourceInd*[]的归并同步数组 *sourceArr*[]的归并。例如下图为 2-路归并排序的一个例子。

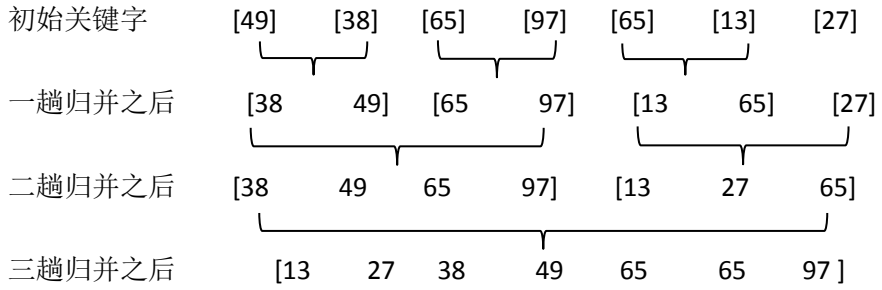


图 13　归并排序

归并排序详细程序可见附录三。

5.4.3.3 折半查找 *k-mer* 所处区间段

当数据量很大适宜采用折半查找，查找时，数据需是排好序的。主要思想是：先确定待查记录所在的范围（区间），然后逐步缩小范围直到找到或找不到该记录为止。假设指针 *low* 和 *high* 分别指示待查元素所在范围内的上界和下界，指针 *mid* 指示区间的中间位置，及$mid = \lfloor (low + high)/2 \rfloor$。设查找的数组区间为 *array*[*low*，*high*]，首先确定该期间的中间位置，其次将查找的值 *T* 与 *array*[*mid*] 比较。若相等，查找成功返回此位置；否则确定新的查找区域，继续折半查找。区域确定如下：如若 *array*[*k*]>*T*，则由数组的有序性可知 *array*[*k*,*k*+1,…,*high*]>*T*; 故新的区间为 *array*[*low*,…,*k*-1]。如若 *array*[*k*]<*T*，类似上面查找区间为 *array*[*k*+1,…,*high*]。每一次查找与中间值比较，可以确定是否查找成功，不成功当前查找区间缩小一半。递归查找即可，时间复杂度为: $O(\log_2 n)$。

其查找结构如下：

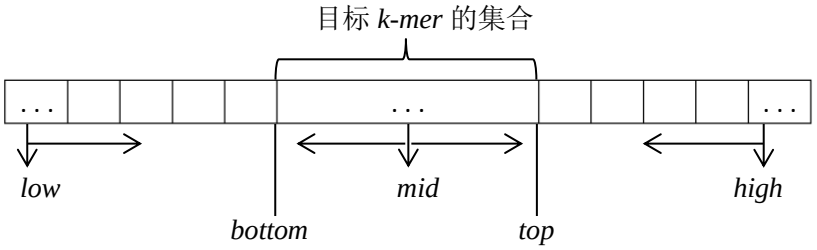


图 14　折半法查找结构

其中，*bottom* 为目标 *k-mer* 的集合的首位置，*top* 为目标 *k-mer* 的集合的末位置。详细程序见附录三。折半法流程图 15 如下：

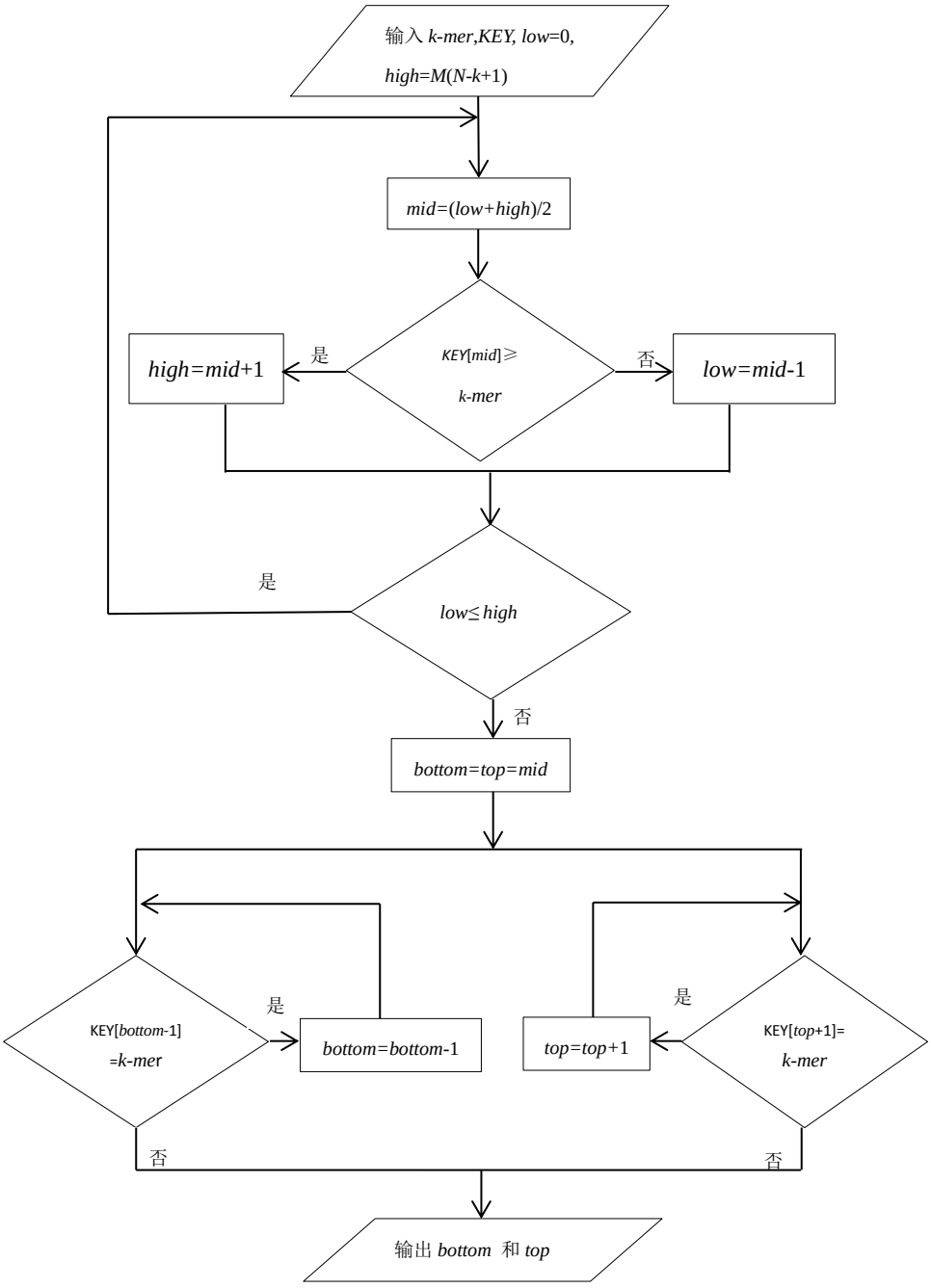

输入 k-mer,KEY, low=0,
high=M(N-k+1)
mid=(low+high)/2
KEY[mid]≥
k-mer
是
否
high=mid+1
low=mid-1
是
low≤high
否
bottom=top=mid
KEY[bottom-1]
=k-mer
是
bottom=bottom-1
top=top+1
是
KEY[top+1]=
k-mer
否
否
输出 bottom 和 top

图 15　　折半法流程图

5.4.4 事前分析

5.4.4.1 空间复杂度

同理根据之前分析可知，模型三适用于任意 k 值，而我们主要针对 $k>13$ (1) 的时候对模型优化得到的模型三，所以此时取 $p=M(N-k+1)$

$$S_3 = M(N-k+1)\times(4+\left\lceil \frac{k}{4} \right\rceil)\ \text{Byte}$$

$$=(4+\left\lceil \frac{k}{4} \right\rceil)(N-k+1)\ \text{MB}$$

分别计算出模型三中 $k=1,2,...,100$ 的空间复杂度，见附录一。

其图像如图 16:

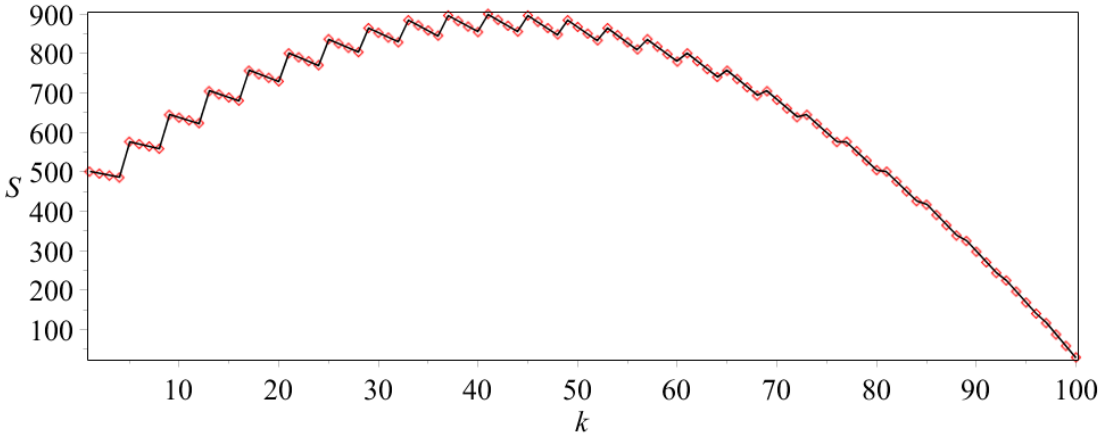


图 16　模型三的理论空间复杂度

由此可以看到，模型三适用于任意 k 值，尤其当 k 很大的时候首选模型三。

5.4.4.2 时间复杂度

模型三的建立初衷我们是为了检索 k 值比较大的键值，但在程序的设计中，

还是考虑了所有的 k 的选择。同时运用到了归并排序和递归算法，这样能有效增加检索速度，即减小时间复杂度。

其中，递归算法是把问题转化为规模缩小了的同类问题的子问题。然后递归调用函数来表示问题的解，一般通过函数或子过程来实现。递归方法一般为在函数或子过程的内部，直接或者间接地调用自己的算法。

归并排序是建立在归并操作上的一种有效的排序算法，将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并[1]。（详细算法可见附录）

依次考虑上述程序的循环次数，可得模型三的时间复杂度为：

$$T_3 = O\left(n \times \left\lceil \frac{k}{4} \right\rceil + n\right) + O(nlog_2 n)$$

其中，n=M(N-k+1)。

5.5 模型四：海绵模型（准哈希）

受“概率分析”的启发，我们进一步分析得到模型四—海绵模型（准哈希），其基本思想如下。

5.5.1 键值空间的预留

最初我们需要预留足够大小的键值空间，这里将其定为“最小”空间的两倍。

最小空间指$M(N-k+1)$，当$M = 1000000$，$N = 100$时，为简化，取$k = 1$，故此预留空间取为4^{14}：

$$1000000 * 100 \ = \ 100000000$$
$$4^{14} \ = \ 268435456$$

也就是说，现有数据最多能产生 1 亿个 *k-mer*，而我们提供了 2.6 亿个空间，这 1 亿个 *k-mer* 分布到 2.6 亿个空间里，必然会产生很多“孔”，这也是本模型命名为“海绵模型”的原因。

5.5.2 键值集的构建

预留空间按 14-*mer* 的“*M* 编码”编号，某个 *k-mer* 的哈希值取其前 14 个字符构成的“*M* 编码”，这样插入 *k-mer* 时按照“*M* 编码”向其对应编号的预留空间插入。但是，这样的哈希值必然会产生重复，对于哈希值相同而 *k-mer* 不同的情况，即在插入时发现该空间已被占用，则按大小顺序依次后移。

在四个碱基出现概率相同的情况下，这 1 亿个 *k-mer* 在 2.6 亿个空间的分布是均匀的，或者说孔的分布是均匀的，因此，哈希值重复的概率并不大，事实上几乎是零。即使出现了重复的现象，那么插入后移的规模也不大，稍作移动就可填入孔内，此时不需继续移动。

因此，这样的结构可以产生几乎$O(1)$的时间复杂度（无论是插入一个键值，还是检索一个键值）。
最终，海绵模型的具体结构如下图所示：

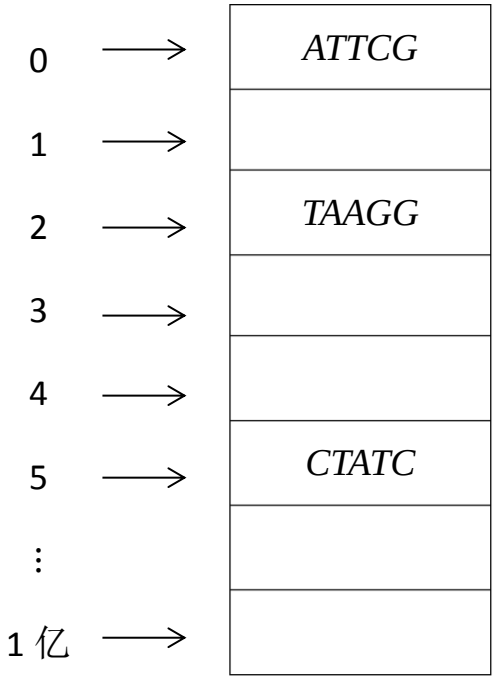


图 17　模型四结构

六、结果分析

6.1 基于概率的存储模型选择

假设 DNA 中 *A,T,G,C* 出现的概率是大致相同的。

（1）某个 *k-mer* 编码不出现的概率为：

$$p=\frac{1}{4^k}，\ q=1-p，\ n=M(N-k+1)$$

$$P_1 = q^n$$

计算得到对不同的 *k* 值，某个 *k-mer* 编码不出现的概率如表 1：

表 1　　*k-mer* 编码不出现的概率（$k=1,2,...,13$）

k	P_1	k	P_1
1	$2.1836\times10^{-12493874}$	8	5.0318×10^{-617}
2	$2.30986\times10^{-2774844}$	9	3.8284×10^{-153}
3	$2.48906\times10^{-670264}$	10	2.0370×10^{-38}
4	$7.35946\times10^{-164880}$	11	4.8059×10^{-10}
5	9.97576×10^{-40736}	12	0.0050
6	1.05506×10^{-10074}	13	0.2695
7	1.74746×10^{-2492}		

图像如图 18 所示：

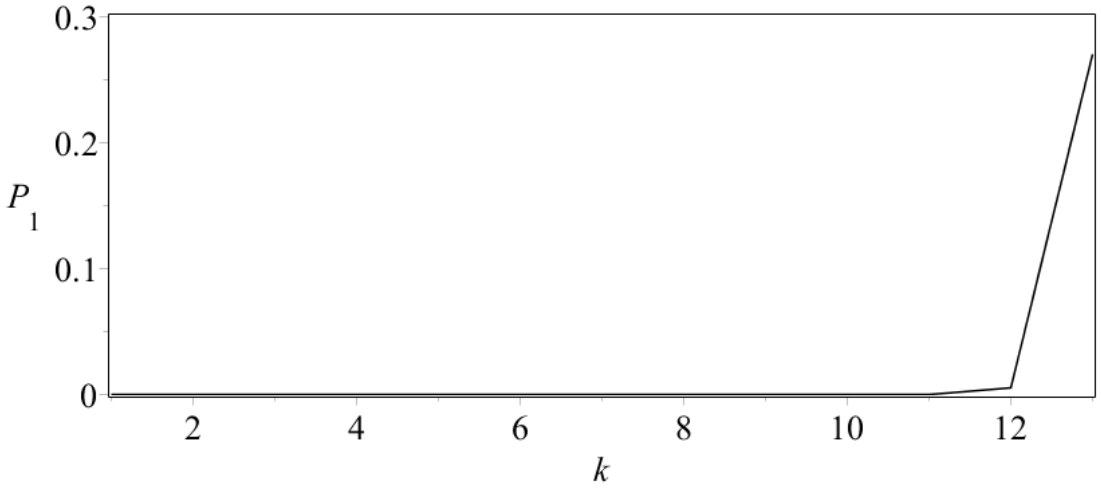


图 18　某 k-mer 编码不出现的概率密度图

显而易见，k=12 左右是一个分界点，进一步计算得到：

表 2　某 *k-mer* 编码不出现的概率($\boldsymbol{k=1,2,\ldots,13}$)

k	11	12	13
P_1	4.8060×10^{-10}	0.0050	0.2695
4^k	4.1943×10^{6}	1.6777×10^{7}	6.7109×10^{-7}
$P_1\times4^k$	2.0157×10^{-3}	8.3377×10^{4}	1.8086×10^{7}

得出：

当 k=11 时概率上缺少不到一个 *k-mer*；

当 *k*=12 时可能缺少 83377 个 *k-mer*；

当 *k*=13 时可能缺少的 *k-mer* 个数非常大。

于是考虑排序等开销建议$k \le 11$时采用模型二，而对于 *k*=12,*k*=13 采用模型三。

（2）对于某个 *k-mer*，某条 DNA 中不出现该 *k-mer* 的概率为：

$$P_2 = (1-\frac{1}{4^k})^{N-k+1} = q^{N-k+1}$$

计算得到对不同的 *k* 值，某条 DNA 中不出现该 *k-mer* 的概率如表 3：

表 3　某条 *DNA* 中不出现某一具体 *k-mer* 的概率（$k=1,2,...,13$）

k	P_2	*k*	P_2
1	3.2072×10^{-13}	8	9.9858×10^{-1}
2	1.6794×10^{-3}	9	9.9965×10^{-1}
3	2.1367×10^{-1}	10	9.9991×10^{-1}
4	6.8410×10^{-1}	11	9.9998×10^{-1}
5	9.1047×10^{-1}	12	9.9999×10^{-1}
6	9.7707×10^{-1}	13	1.0000
7	9.9428×10^{-1}		

图像如图 19 所示：

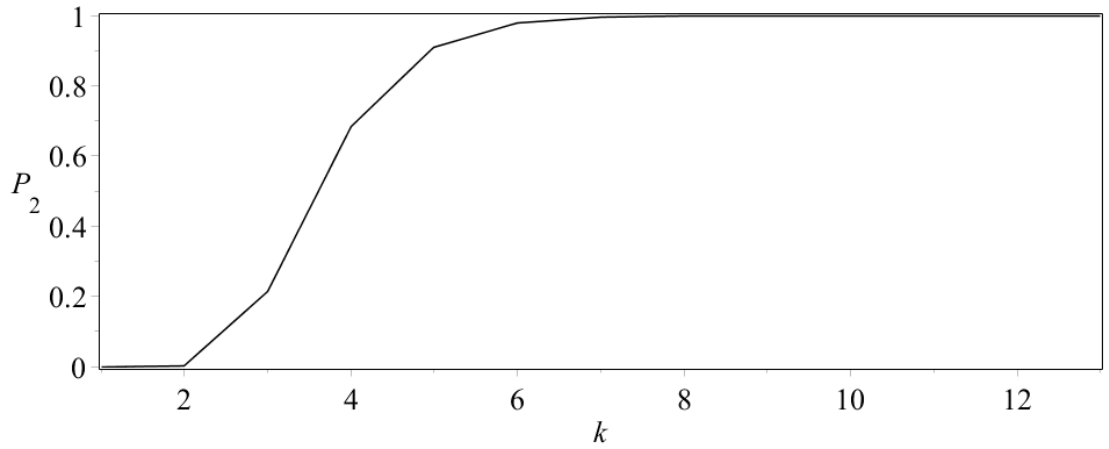


图 19　某条 DNA 中不出现某一 *k-mer* 的概率分布

（3）对于某个 *k-mer*，在某条 DNA 上出现 *t* 次的概率可以用下面多项式的系数来体现($\mathrm{coef}(f,x,t)$表示提取多项式f关于变量x的t次的系数)：

$$f = (px + q)^n$$

$$P_3(t) = coef(f, x, t) = \binom{t}{n} p^t q^{n-t}$$

得到对不同的 *k* 值，在某条 DNA 上出现 *t* 次的概率$(x > 0)$见下表 4：

表 4 某一 *k-mer* 在某条 DNA 上出现不超过 *x* 次的概率

k	*t*	ΣP_3
1	<14	2.4578×10^{-3}
2	<14	9.9657×10^{-1}
	<10	9.0942×10^{-1}
	<6	4.1008×10^{-1}
	<2	1.2764×10^{-2}
3	<14	1.0000
	<10	1.0000
	<6	9.9549×10^{-1}
	<2	5.4604×10^{-1}
4	<14	1.0000
	<10	1.0000
	<6	1.0000
	<2	9.4433×10^{-1}
5	<14	1.0000
	<10	1.0000
	<6	1.0000
	<2	9.9591×10^{-1}
12	<14	1.0000
	<10	1.0000
	<6	1.0000
	<2	1.0000

根据所得数据，我们可以得到：

当 *k*=1，2 时，

建议采用模型一，使用 104bit 存储位置信息；

当 k=3 时，

可以采用模型一，也可以考虑模型二进行存储；

当$k = 4, \ldots, 11$时，

考虑采用模型二；

当$k \geq 8$时，

建议采用模型三。

综上所述：

$$\begin{cases} k \leq 3 & \text{模型一} \\ 3 \leq k \leq 11 & \text{模型二} \\ k \geq 8 & \text{模型三} \end{cases}$$

6.2 空间复杂度

根据以上分析，我们对不同的 k 值采用相应的模型，运行程序，并记录相关空间复杂度，统计得到下表：

表 5　模型一、二不同 k 值的空间复杂度

k	1	2	3	4	5	6	7	8
模型一	50M	198M	793M	3.1G	12G			
模型二（32 位）	112M	163M	304M	363M	368M	370M	423M	612M
模型二（64 位）	404M	469M	642M	727M	733M	733M	782M	966M

根据所得数据作图如下图 20：

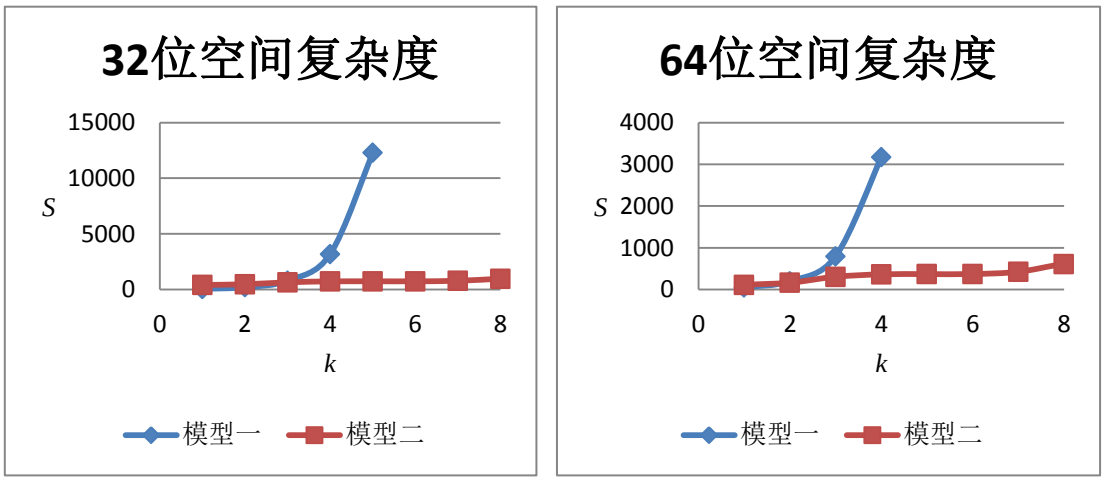


图 20 模型一模型二实际空间复杂度

运行模型三程序 $k=1,2,...,100$，分别记录相关空间复杂度，统计数据见附录一并绘制得到下散点图 21：

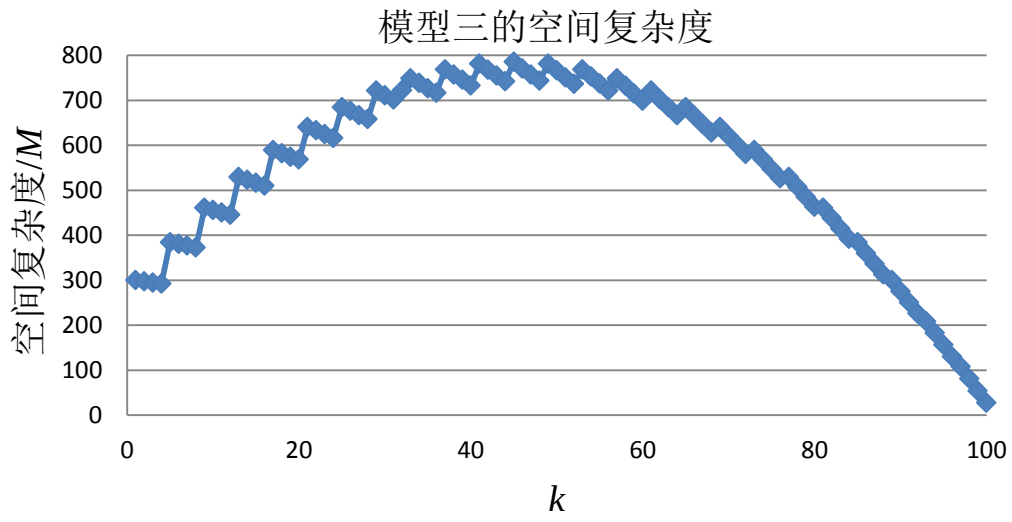


图 21 模型三实际空间复杂度

6.3 时间复杂度

运行模型一与模型二的程序，并记录相关时间复杂度，统计得到下表：

表 6 模型一、二不同 *k* 值的时间复杂度

k	1	2	3	4	5	6	7	8
模型一	5s	6s	7s	8s	12s			

模型二（32 位）	9s	9s	9s	10s	15s	34s	107s	378s
模型二（64 位）	11s	10s	9s	9s	16s	32s	100s	366s

根据所得数据作图如下图 22：

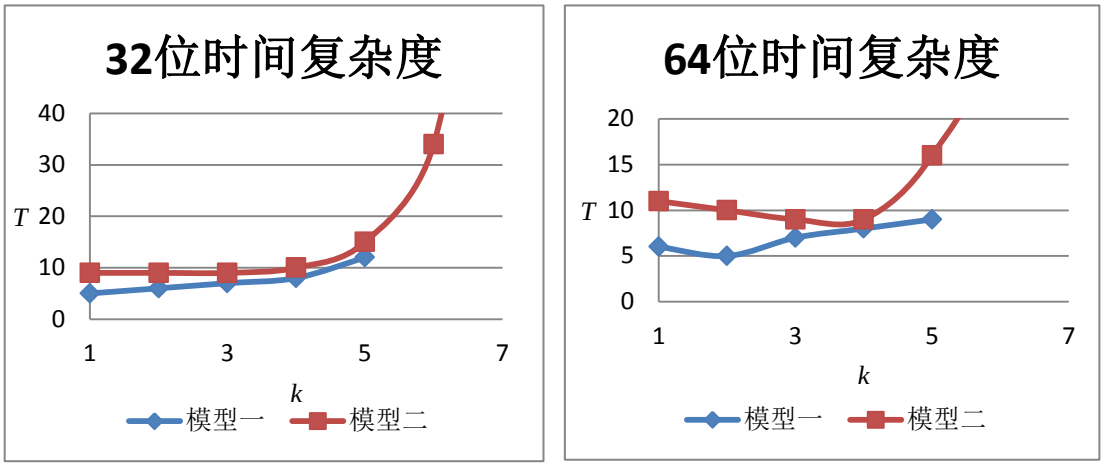


图 22　模型一模型二实际时间复杂度

运行模型三程序 $k=1,2,...,100$，分别记录相关时间复杂度，统计数据见附录一，并绘制得到下散点图 23：

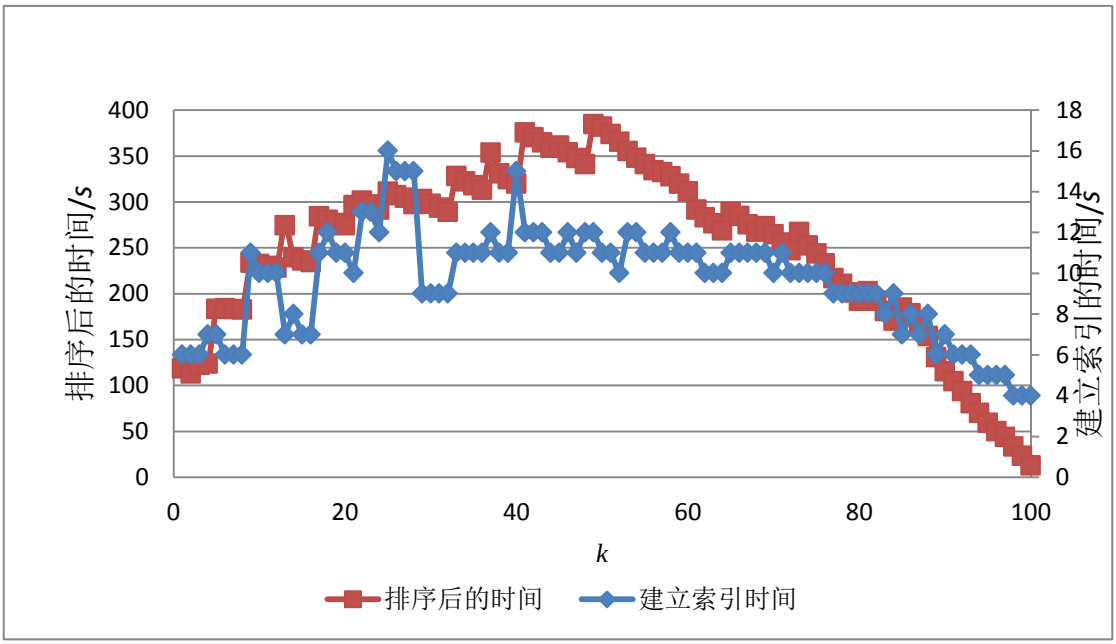


图 23　模型三实际时间复杂度

6.4 模型评价

针对本问题我们建立了三种模型来求解，为此我们建立一个评价函数对三个模型进行评价与分析。

6.4.1 模型的评价函数

因为数据库的检索与索引涉及到时间与空间复杂度，于是建立的评价函数为：

$$Y = S \times T$$

其中 S 的单位为 MByte，T 的单位为 s。

正如我们所知当我们减少内存消耗时我们所需要消耗时间便增加了,当我们减少时间消耗时我们需要消耗的内存便增加了,我们所建立的评价函数需要综合考虑两者,使两者达到最优化。我们考虑用 $Y=S\times T$ 来度量模型的好坏。便是考虑到无论是还是 S 还是 T 的增加都会带来 Y 的增加。我们之所以不选则加的原因是，防止其中一个指标过大减弱了另一个指标的影响。

根据之前空间复杂度与时间复杂度的计算分别得到模型一、二、三具体的评价函数如下：

模型一

$$S=13\times 4^{k}$$

$$T = O(M(N-k+1))$$

$$Y = S \times T = (13 \times 4^{k}) \times O(M(N-k+1))$$

模型二

$$S = O(4 \times 4^{k-10} + N - k + 1)$$

$$T = O(M(N-k+1))$$

$$Y = S \times T = O(4 \times 4^{k-10} + N - k + 1) \times O(M(N-k+1))$$

模型三

$$S = (4 + \left\lceil \frac{k}{4} \right\rceil)(N-k+1)$$

$$T = O(n \times \left\lceil \frac{k}{4} \right\rceil + n)$$

$$n = M \times (N-k+1)$$

$$Y = S \times T = [(4 + \left\lceil \frac{k}{4} \right\rceil)(N-k+1)] \times O(n \times \left\lceil \frac{k}{4} \right\rceil + n)$$

6.4.2 评价函数理论计算

我们根据所得到的评价函数分别计算了三种模型的评价参数，具体数据如下：

表 7 模型一

k	1	2	3	4	5
T	6	5	7	8	12
S	50	198	795	3.1×10^{3}	1.23×10^{4}
Y	300	990	5565	2.48×10^{4}	1.5×10^{5}

表 8 模型二

k	1	2	3	4	5	6	7	8	9
T	11	10	9	9	16	22	100	326	1420
S	112	163	304	363	368	370	423	612	705
Y	1232	1630	2736	3267	5688	8140	42300	1.99×10^{5}	1.0×10^{6}

表 9 模型三

k	1	2	3	4	5	6	7	8	9
T_1	6	6	6	7	7	6	6	6	11
T_2	118.46	112.84	122.41	123.56	183.42	184.09	182.84	182.72	233.41
S	286	283	280	278	366	362	359	355	439
Y	33879.56	31933.72	34274.8	34349.68	67131.72	66640.58	65639.56	64865.6	102467

6.5 模型分析

6.5.1 理论值与实际值分析

分别绘制不同模型的空间复杂度的和时间复杂度理论值与实际值的比较曲线。模型一

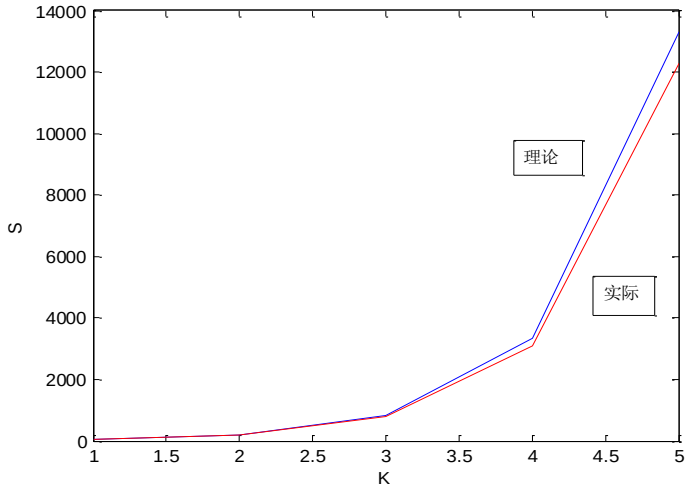


图 24　模型一空间复杂度理论与实际值的比较

拟合函数：$S=13\times4^{k}$

模型三

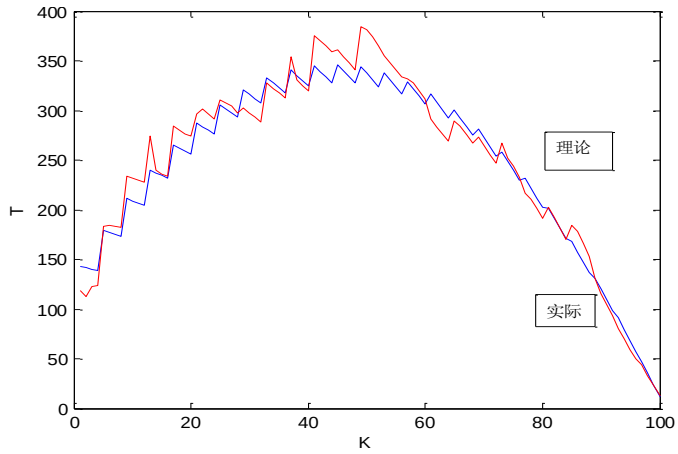


图 25　模型三时间复杂度理论与实际值的比较

拟合函数：$T=0.432\times(101-k)\times\left\lfloor\frac{k}{4}\right\rfloor+(101-k)$

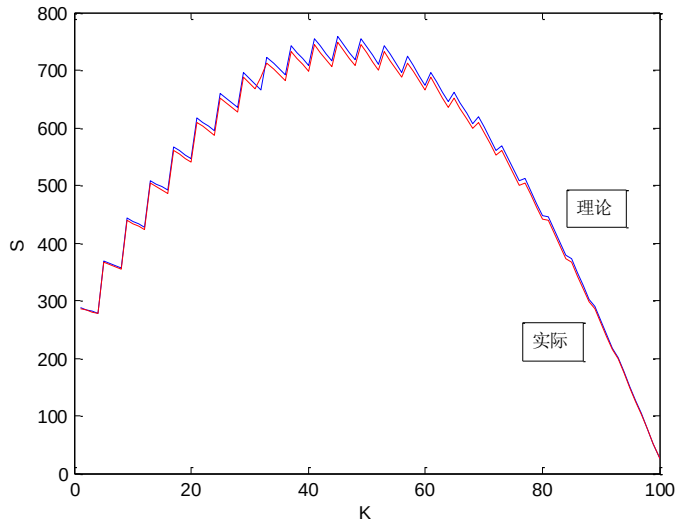


图 26　模型三空间复杂度理论与实际值的比较

拟合函数：$S = 0.97 \times \left\lceil \frac{k}{4} \right\rceil \times (101 - k) + 1.9 \times (101 - k)$

6.5.2 基于时间复杂度的分析 $T(k)$

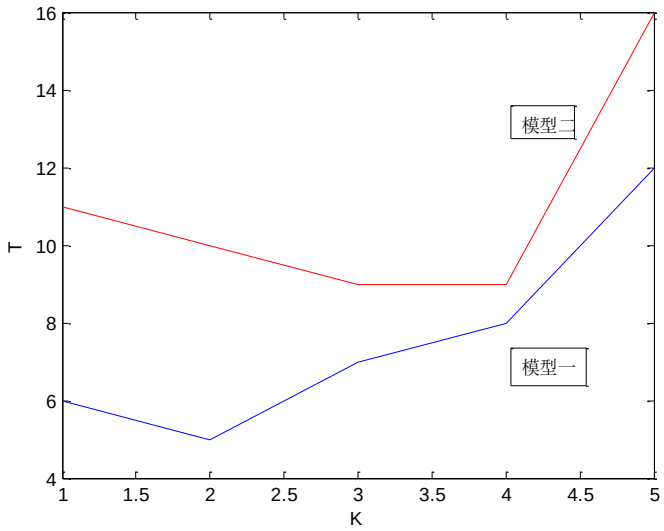


图 27 模型一与模型二的时间复杂度的比较

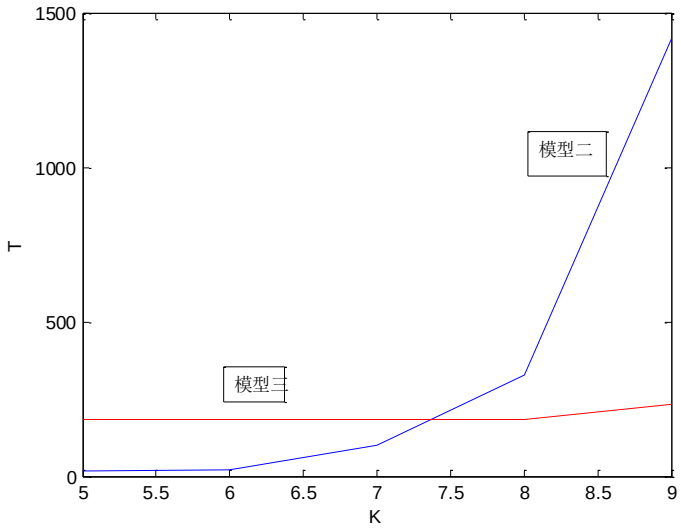


图 28 模型二与模型三的时间复杂度的比较

$k \leq 4$时，模型一比模型二耗时时间少，模型三建立索引时间与二者几乎相等，若加上排序则远大于二者。所以，$k < 4$时认为模型一最优，模型二优于模型三；对于$k > 5$时模型一所用内存已经超过了题目中的要求，我们对模型一不做考虑。

比较模型二与模型三：当$5 \leq k \leq 7$时，模型二所用时间比模型三要少；对于$k \geq 8$时模型三比模型二要优很多。

6.5.3 基于空间复杂度的分析 $S(k)$

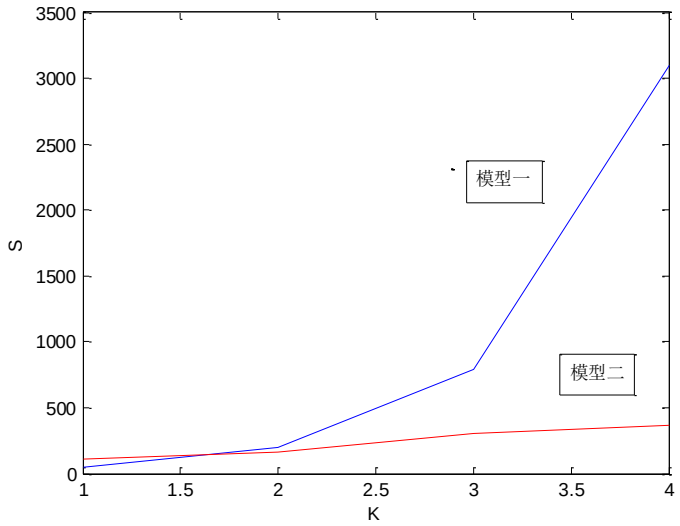


图 29 模型一与模型二空间复杂度的比较

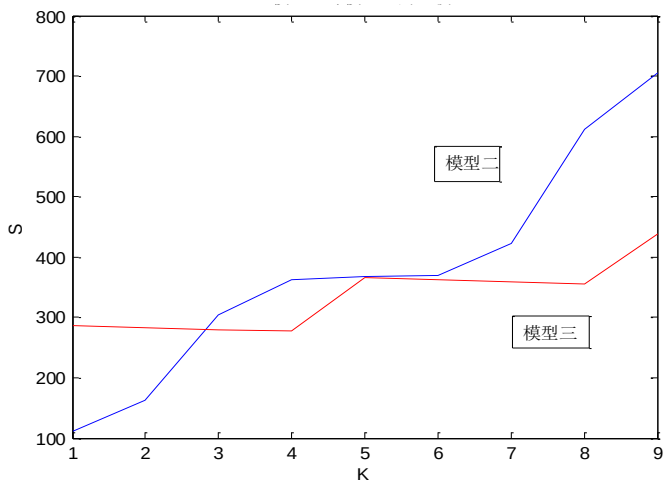


图 30 模型二与模型三的空间复杂度的比较

当$k=1$时，模型一要比模型二比模型三耗内存要少，模型二要比模型三耗内；
当$2 \le k \le 5$时，模型二比模型三所耗内存少，模型三比模型一所耗内存少；
当$k > 5$时，模型三所耗内存最少。

6.5.4 综合分析 $Y(k)$

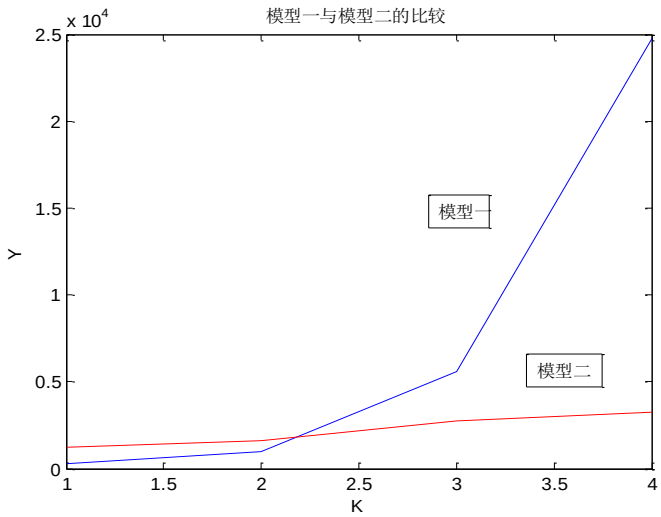


图 31 模型一与模型二的综合比较

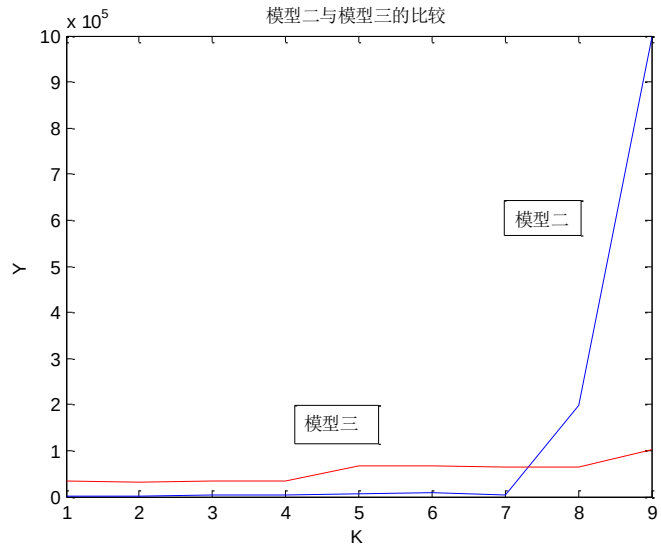


图 32 模型二与模型三的综合比较

基于评价函数对三者进行综合分析，得到：

当$k = 1,2$时，模型一优于模型二，模型二优于模型三；

当$3 \leq k \leq 7$时，模型二优于模型三；

当$k \geq 8$时，模型三最优。

6.5.5 关于 8G 内存限制的分析

模型一：当$k = 5$时所耗用的内存已达到 12G 超过了 8G。模型一所支持 k 值为 1

到 4。

模型二：模型二所耗内存数成指数型增长，我们对模型二根据理论分析计算当 $k = 21$时其所用内存达到 16G，超过了我们的限制。

模型三：我们根据实际数据得到模型三所耗内存在 k=46 时取得最大值，其最大值为 748M，远远小于 8G，所以模型三支持所有的 k。

6.6 程序实际运行结果

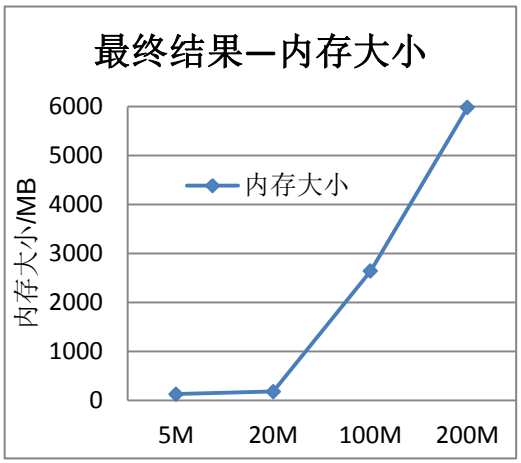
对于夏令营中华大基因所给出的题目：所有 DNA 序列中只包含 A、T、G、C 四种字母，给定一个长度为 100 的 DNA 序列，分别在大小为 5M、10M、100M 和 200M 的 DNA 序列文件中检索含有该已知 DNA 序列的个数。

针对华大基因的题目，我们对原有程序进行了相应的轻微变动，运行程序得到最终结果如下表所示：

表 10　程序实际运行结果

DNA 序列大小	5M	10M	100M	200M
建立索引时间（s）	2.015	9.542	79.01	219.085
seq1 检索时间（s）	2	2	3	3
seq2 检索时间(s)	5	5	7	8
内存大小(MB)	131.5	184.7	2643.9	5983.5
待检索 DNA 数量(条)	315416	84089	143344	456411

绘制相应图像如下：



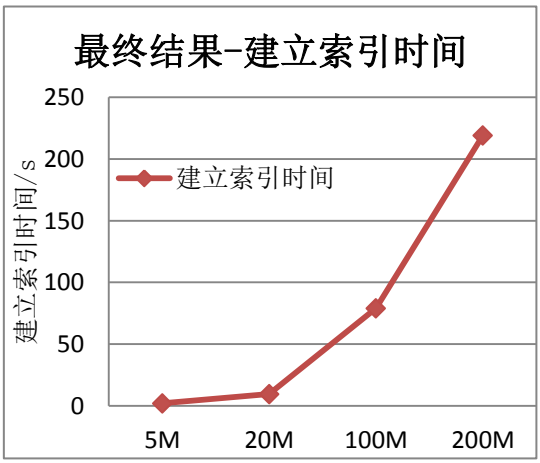


图 33　程序运行最终结果

参考文献

[1] 严蔚敏,吴伟民,数据结构[M],北京:清华大学出版社,1996.

[2] 纪震,周家锐,朱泽轩,基于生物信息学特征的 DNA 序列数据压缩算法[J],电子学报,2011.

[3] 王树林,王戟,陈火旺,张鼎兴, k-长 DNA 子序列计数算法研究[J],计算机工程,2007.5.

[4] 陈明，数据结构（C 语言版）[M]，北京：清华大学出版社，2005.9.

[5] 谭浩强，C 程序设计（第二版）[M]，北京：清华大学出版社，2002.

[6] 熊文萍，基于 k-mer 短序列的 DNA 数据压缩算法研究[D]，广州：华南理工大学，2014.4.

[7] 张鑫鑫，生物序列数据 K-mer 频次统计[D]，合肥：中国科学技术大学，2014.4.

附录

附录一 模型三的空间复杂度

模型三$k=1,2,...,100$的理论空间复杂度

k	1	2	3	4	5	6	7	8	9	10
$S_3/(M)$	500.0	495.0	490.0	485.0	576.0	570.0	564.0	558.0	644.0	637.0
k	11	12	13	14	15	16	17	18	19	20
$S_3/(M)$	630.0	623.0	704.0	696.0	688.0	680.0	756.0	747.0	738.0	729.0
k	21	22	23	24	25	26	27	28	29	30
$S_3/(M)$	800.0	790.0	780.0	770.0	836.0	825.0	814.0	803.0	864.0	852.0
k	31	32	33	34	35	36	37	38	39	40
$S_3/(M)$	840.0	828.0	884.0	871.0	858.0	845.0	896.0	882.0	868.0	854.0
k	41	42	43	44	45	46	47	48	49	50
$S_3/(M)$	900.0	885.0	870.0	855.0	896.0	880.0	864.0	848.0	884.0	867.0
k	51	52	53	54	55	56	57	58	59	60
$S_3/(M)$	850.0	833.0	864.0	846.0	828.0	810.0	836.0	817.0	798.0	779.0
k	61	62	63	64	65	66	67	68	69	70
$S_3/(M)$	800.0	780.0	760.0	740.0	756.0	735.0	714.0	693.0	704.0	682.0
k	71	72	73	74	75	76	77	78	79	80
$S_3/(M)$	660.0	638.0	644.0	621.0	598.0	575.0	576.0	552.0	528.0	504.0
k	81	82	83	84	85	86	87	88	89	90
$S_3/(M)$	500.0	475.0	450.0	425.0	416.0	390.0	364.0	338.0	324.0	297.0
k	91	92	93	94	95	96	97	98	99	100
$S_3/(M)$	270.0	243.0	224.0	196.0	168.0	140.0	116.0	87.0	58.0	29.0

模型三的相关实际数据

k	1	2	3	4	5	6	7	8
T_1	6	6	6	7	7	6	6	6
T_2	118.46	112.84	122.41	123.56	183.42	184.09	182.84	182.72
S	286	283	280	278	366	362	359	355
Y	33879.56	31933.72	34274.8	34349.68	67131.72	66640.58	65639.56	64865.6

k	9	10	11	12	13	14	15	16
T_1	11	10	10	10	7	8	7	7
T_2	233.41	232.17	230.11	228.27	274.3	239.49	235.9	234.13
S	439	434	429	424	504	498	492	486
Y	102467	100761.8	98717.19	96786.48	138247.2	119266	116062.8	113787.2

k	17	18	19	20	21	22	23	24
T_1	11	12	11	11	10	13	13	12
T_2	284.25	280.34	276.72	274.2	296.4	301.03	296.91	291.51
S	561	554	547	541	610	603	595	587
Y	159464.3	155308.4	151365.8	148342.2	180804	181521.1	176661.5	171116.4

k	25	26	27	28	29	30	31	32
T_1	16	15	15	15	9	9	9	9
T_2	311.06	307.15	304.45	297.33	302.98	297.57	293.47	288.85
S	652	644	635	627	687	677	668	688
Y	202811.1	197804.6	193325.8	186425.9	208147.3	201454.9	196038	198728.8

k	33	34	35	36	37	38	39	40
T_1	11	11	11	11	12	11	11	15
T_2	327.91	321.94	317.99	312.99	353.55	330.66	324.59	319.77
S	713	703	692	682	732	721	710	698
Y	233799.8	226323.8	220049.1	213459.2	258798.6	238405.9	230458.9	223199.5

k	41	42	43	44	45	46	47	48
T_1	12	12	12	11	11	12	11	12
T_2	375.54	370.33	364.65	358.76	361.07	353.93	347.64	341.09
S	744	731	719	707	748	734	721	708
Y	279401.8	270711.2	262183.4	253643.3	270080.4	259784.6	250648.4	241491.7

k	49	50	51	52	53	54	55	56
T_1	12	11	11	10	12	12	11	11
T_2	384.39	381.69	373.86	365.47	355.33	348.07	341.04	334.21
S	744	730	715	701	732	717	702	687
Y	285986.2	278633.7	267309.9	256194.5	260101.6	249566.2	239410.1	229602.3

k	57	58	59	60	61	62	63
T_1	11	12	11	11	11	10	10
T_2	332.2	327.72	319.77	311.41	291.33	283.2	276.3
S	713	697	681	665	687	669	652
Y	236858.6	228420.8	217763.4	207087.7	200143.7	189460.8	180147.6

k	64	65	66	67	68	69	70	71
T_1	10	11	11	11	11	11	10	11
T_2	268.83	289.03	284.3	275.36	267.22	273.4	264.46	255.41
S	635	652	634	616	598	610	591	572
Y	170707.1	188447.6	180246.2	169621.8	159797.6	166774	156295.9	146094.5

k	72	73	74	75	76	77	78	79
T_1	10	10	10	10	10	9	9	9
T_2	247.12	266.94	252.22	243.58	232.75	216.82	210.66	201.17
S	553	561	541	521	501	504	483	462
Y	136657.4	149753.3	136451	126905.2	116607.8	109277.3	101748.8	92940.54

k	80	81	82	83	84	85	86	87
T_1	9	9	9	8	9	7	8	7
T_2	191.86	202.41	191.96	180.93	170.34	184.63	178.35	165.8
S	441	439	417	395	373	366	343	320
Y	84610.26	88857.99	80047.32	71467.35	63536.82	67574.58	61174.05	53056

k	88	89	90	91	92	93	94	95
T_1	8	6	7	6	6	6	5	5
T_2	153.51	130.71	115.44	104.49	93.78	80.45	70.14	59.04
S	298	286	262	238	215	198	174	149
Y	45745.98	37383.06	30245.28	24868.62	20162.7	15929.1	12204.36	8796.96

k	96	97	98	99	100			
T_1	5	5	4	4	4			
T_2	49.94	43.75	33.28	23.36	12.75			
S	124	103	77	51	26			
Y	6192.56	4506.25	2562.56	1191.36	331.5			

附录二 统计文件中 *A*、*C*、*G*、*T* 出现的次数

统计 solexa_100_170_1.fa 文件中 *A*、*C*、*G*、*T* 出现的次数

（solexa_100_170_2.fa 文件读取程序类同）

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
int main()
{
    int l=0,m=0,n=0,o=0;
    int len = 100;
    int s=0,j;
    FILE *fp;
    char filename[]="solexa_100_170_1.fa";
    char   line[len+2];
    char *p;
    fp = fopen(filename, "r");
    if(fp == NULL){
        printf("error on open file : %s \n", filename);
        exit(EXIT_FAILURE);
    }
    while( fgets(line , len+2, fp) )
    {
        if (s)
        {
            p=line ;
            for(j=0;j<100;j++)
            {   if(*p=='A') l++;
                if(*p=='T') m++;
                if(*p=='C') n++;
                if(*p=='G') o++;
                p++;
            }
        }
             s=!s;
    }
     printf("the number of A:%d\n",l);
     printf("the number of T:%d\n",m);
     printf("the number of C:%d\n",n);
     printf("the number of G:%d\n",o);
}
```

附录三 模型三的程序

```
#include "string.h"
#include "time.h"
#include "stdio.h"
#include "stdlib.h"


#define SUCCESS 1
#define FAILED    0

#define M 1000000
#define N 100
#define K 5



typedef    unsigned short int    usint;
typedef    unsigned char    kint;
typedef    long    int    lint;
typedef    short int    sint;


typedef struct{
      kint kmer[(K+3)>>2];
}CodeInt;




/*--------Encoding-------------*/
/*    'A'      encode to 00 = 0          */
/*    'C'      encode to 01 = 1          */
/*    'G'      encode to 10 = 2          */
/*    'T'      encode to 11 = 3          */
/*    could optimize encoding        */
/*-----------------------------*/


/* forward direction order encoding */
/* ACGTC encode to kmer[0]=[ACGT] , kmer[1]=[C]    */
CodeInt encode(kint k, char* kmer)
{
```

```
	kint i;
	CodeInt r;
	for(i=0;i<(k+3)>>2;i++)r.kmer[i]=0; /* 必须初始化 */
	for(i=0;i<k;i++){
		//printf("%X\n",r.kmer[i/4]);
		if(kmer[i]=='A') continue;
		else if(kmer[i]=='C') r.kmer[i/4] |= ((kint)1 << 2*(3-i%4));
		else if(kmer[i]=='G') r.kmer[i/4] |= ((kint)2 << 2*(3-i%4));
		else if(kmer[i]=='T') r.kmer[i/4] |= ((kint)3 << 2*(3-i%4));
		else{
			printf("Expected letter A/C/G/T, But recieved %c\n",kmer[i]);
			exit(-1);
		}
	}
	//for(i=0;i<(k+3)>>2;i++) printf("%X",r.kmer[i]);printf("\n");
	return r;
}



/*  原先 k-mer 编码为 a, 后续增加字符 c, 生成新的编码 */
CodeInt encode0(kint k, CodeInt a, char c)
{
	kint i,m=(k+3)>>2;
	CodeInt b;
	for(i=0;i<m-1;i++) b.kmer[i] = ( a.kmer[i]<<2 ) | ( a.kmer[i+1] >> 6 );
	b.kmer[m-1] = a.kmer[m-1] << 2;

	if(c=='A') NULL;
	else if(c=='C')    b.kmer[m-1] |= (1 << 2*(3-(k-1)%4));
	else if(c=='G')    b.kmer[m-1] |= (2 << 2*(3-(k-1)%4));
	else if(c=='T')    b.kmer[m-1] |= (3 << 2*(3-(k-1)%4));
	else {
		printf("Expected letter A/C/G/T, But recieved %c\n",c);
		exit(-1);
	}
	return b;
}



/* 将 CodeInt 解码成字符串，存储到 str 中 */
void decode(CodeInt c, kint k, char *str)
{
```

```
	kint i,j,r,s,m=(k+3)>>2;
	for(i=0;i<m-1;i++){
		r=c.kmer[i];
		for(j=0;j<4;j++){
			s=r&3;
			if(s==0) str[i*4+3-j]='A';
			else if(s==1) str[i*4+3-j]='C';
			else if(s==2)str[i*4+3-j]='G';
			else str[i*4+3-j]='T';		// s==3
			r>>=2;
		}
	}
	if(k%4 !=0 ){
		str[k]='\0';
		r=c.kmer[m-1] >> 2*(4-k%4) ;
		for(j=0;j<k%4;j++){
			s=r&3;
			if(s==0) str[(m-1)*4+k%4-j-1]='A';
			else if(s==1) str[(m-1)*4+k%4-j-1]='C';
			else if(s==2)str[(m-1)*4+k%4-j-1]='G';
			else str[(m-1)*4+k%4-j-1]='T';		// s==3
			r>>=2;
		}
	}
	else{
		r=c.kmer[m-1] ;
		for(j=0;j<4;j++){
			s=r&3;
			if(s==0) str[m*4-j-1]='A';
			else if(s==1) str[m*4-j-1]='C';
			else if(s==2)str[m*4-j-1]='G';
			else str[m*4-j-1]='T';		// s==3
			r>>=2;
		}
	}
}




CodeInt* InitKEY(lint n)
{
	lint i;
```

```
	CodeInt *p;
	p=(CodeInt *)malloc(n*sizeof(CodeInt));
	if(!p){
		printf("could not malloc in InitKEY.\n");
		exit(-1);
	}
	return p;
}

lint* InitINDEX(lint n)
{
	lint i;
	lint *p;
	p=(lint *)malloc(n*sizeof(lint));
	if(!p){
		printf("could not malloc in InitKEY.\n");
		exit(-1);
	}
	return p;
}



void DestroyKEY(CodeInt* key, lint* index)
{
	free(key);
	free(index);
	key=0;
	index=0;
}







void TravelKEY(CodeInt* key, lint *index, lint n, kint k)
{
	lint i,j,t;
	lint s;
	char *str;
	str=(char*)malloc((k+1)*sizeof(char));
	str[k]='\0';
```

```
	for(s=0;s<n;s++){
		decode(key[s],k,str);
		t=index[s];
		j = 0x7F	&	t;		/* 低 7 位存放 j */	/* 能支持 128 位定长的 DNA 序列 */
		i = 0xFFFFFF & (t>>7);	/* 紧接的 24 位存放 i */	/* 能支持 2^24=16M 条 DNA 序列 */
		printf("n=0x%X\t%s, (i,j)=(%d,%d)\n",s,str,i,j);
	}
	free(str);
}
```

```
void InsertKEY(CodeInt *key, lint *index, lint n, char *str, lint ind, kint k)
{
	char* str1=0;
	CodeInt r;
	kint i,len;

	str1=(char *)malloc((k+1)*sizeof(char));
	str1[k]='\0';
	str1[0]='A'; /* 首位任意初始化 */
	for(i=0;i<k-1;i++) str1[i+1]=str[i];
	r=encode(k,str1);
	free(str1);
	str1=0;


	len=strlen(str);	// 字符串长度
	for(i=0;i<len-k+1;i++){ // size = m-K+1
		r=encode0(k,r,*(str+i+k-1));
		key[ind*(N-k+1)+i]=r;
		index[ind*(N-k+1)+i] = (ind << 7) | i ;
	}
	//TravelKEY(key,index,ind,k);
printf("=====================================\n\n");
}
```

```
int CompareCode(CodeInt a, CodeInt b,kint m)
{
	kint i;
	for(i=0;i<m;i++){
		if(a.kmer[i]<b.kmer[i]) return -1;		// a<b
		else if(a.kmer[i]>b.kmer[i]) return 1;	// a>b
		else continue;
	}
	return 0;	// a=b
}
```

```
void Merge(CodeInt sourceArr[],CodeInt tempArr[],lint startIndex, lint midIndex,lint endIndex,
lint sourceInd[], lint tempInd[],kint m)
{
	lint i = startIndex,j=midIndex+1,k = startIndex;
	kint s;
	while(i!=midIndex+1 && j!=endIndex+1){
		if( CompareCode(sourceArr[i],sourceArr[j],m)<=0){
			for(s=0;s<m;s++) tempArr[k].kmer[s] = sourceArr[i].kmer[s];
			tempInd[k++] = sourceInd[i++];
		}
		else{
			for(s=0;s<m;s++) tempArr[k].kmer[s] = sourceArr[j].kmer[s];
			tempInd[k++] = sourceInd[j++];
		}
	}
	while(i!=midIndex+1){
		for(s=0;s<m;s++) tempArr[k].kmer[s] = sourceArr[i].kmer[s];
		tempInd[k++]=sourceInd[i++];
	}
	while(j!=endIndex+1){
		for(s=0;s<m;s++) tempArr[k].kmer[s] = sourceArr[j].kmer[s];
		tempInd[k++]=sourceInd[j++];
	}
	for(i=startIndex;i<=endIndex;i++)
		for(s=0;s<m;s++) sourceArr[i].kmer[s] = tempArr[i].kmer[s];
```

```
    for(i=startIndex;i<=endIndex;i++)sourceInd[i] = tempInd[i];
}
//内部使用递归
void MergeSort1(CodeInt sourceArr[],CodeInt tempArr[],lint startIndex,lint endIndex, lint sourceInd[], lint tempInd[], kint m)
{
    int midIndex;
    if(startIndex<endIndex){
        midIndex=(startIndex+endIndex)/2;
        MergeSort1(sourceArr,tempArr,startIndex,midIndex,sourceInd,tempInd,m);
        MergeSort1(sourceArr,tempArr,midIndex+1,endIndex,sourceInd,tempInd,m);
        Merge(sourceArr,tempArr,startIndex,midIndex,endIndex,sourceInd,tempInd,m);
    }
}
//内部使用递归,并行版本
void MergeSort2(CodeInt sourceArr[],CodeInt tempArr[],lint startIndex,lint endIndex, lint sourceInd[], lint tempInd[], kint m)
{
    int midIndex;
    if(startIndex<endIndex){
        midIndex=(startIndex+endIndex)>>1;
        /*设置并行区域内由两个线程执行*/
        //omp_set_num_threads(2);
        #pragma omp parallel
        //sections private( tempArr, tempInd)
        {
            #pragma omp section nowait
            {
                #pragma omp section
                MergeSort2(sourceArr,tempArr,startIndex,midIndex,sourceInd,tempInd,m);
                #pragma omp section
                MergeSort2(sourceArr,tempArr,midIndex+1,endIndex,sourceInd,tempInd,m);
            }
        }
        Merge(sourceArr,tempArr,startIndex,midIndex,endIndex,sourceInd,tempInd,m);
    }
}

void SortKEY(CodeInt *key, lint *index, lint n,kint m)
{
    lint i;
    CodeInt *tempKey;
    lint *tempIndex;
    tempKey=InitKEY(n);
```

```
    tempIndex=InitINDEX(n);
    MergeSort2(key,tempKey,0,n-1,index,tempIndex,m);
    free(tempIndex);
    free(tempKey);
}


/* 二分法查找区间段 */
int SearchKEY(CodeInt *key, lint *index, lint n, CodeInt r, kint m, lint loc[])
{
    lint low=0,high=n-1,mid,bottom,top;
    int flag;
    while(low<=high){
        mid=(low+high)/2;
        flag=CompareCode(key[mid],r,m);
        if(flag==1)high=--mid;
        else if(flag==-1)low=++mid;
        else break;
    }
    if(low>high) return 0;

    bottom=top=mid;
    if(CompareCode(key[low],r,m)==0)bottom=low;
    if(mid>0 && CompareCode(key[mid-1],r,m)==-1) low=mid;
    while(low<bottom){
        mid=(low+bottom)/2;
        flag=CompareCode(key[mid],r,m);
        if(flag==-1) low=++mid;
        else bottom=mid;
    }
    if(CompareCode(key[high],r,m)==0)top=high;
    if(mid<n-1 && CompareCode(key[mid+1],r,m)==1) high=mid;
    while(high>top){
        mid=(high+top+1)/2;
        flag=CompareCode(key[mid],r,m);
        if(flag==1) high=--mid;
        else top=mid;
    }
    loc[0]=low;
    loc[1]=high;

    return 1;

}
```

```
int Searching(CodeInt *key, lint *index, lint n)
{
	kint k=K;
	CodeInt r;
	char str[N+1];
	char flag;
//	lint n = M*(N-K+1);
	kint m = (K+3)>>2;
	lint loc[2];


	printf("Please input k-mer for searching (k=%d, ONLY accept Capital Letters A/C/G/T):\n
",k);
	while(1){
		fflush(stdin);scanf("%[ACGT]",str);
		printf("k-mer is %s, length=%d\n",str,strlen(str));
		str[k]='\0';
		if(strlen(str)!=k){
			printf("Expected k-mer with k=%d, Please input again:\n",k);
			continue;
		}
		r=encode(k,str);		// printf("%s=%d\n",str,r);
		if(SearchKEY(key,index,n,r,m,loc)){
			printf("k-mer: %s is location in [%d, %d]\n",str,loc[0],loc[1]);
		}else{
			printf("NO such k-mer in all DNA Sequences.\n");
		}


		printf("search again?([Y]/N)");
		fflush(stdin);
		scanf("%[nN]",&flag);
		if(flag=='N' || flag=='n')break;
		printf("Please input k-mer for location: ");
	}
	return 1;
}


// small scale for test, need loc[][3] space
```

```
int SummaryKEYtest(CodeInt *key, lint *index, lint n, lint loc[][3])
{
	lint i,j;
	kint m=(K+3)>>2;
	CodeInt r;
	lint count=0;
	char str[K+1];

	loc[0][0]=0;
	loc[n][0]=n;
	r=key[0];

	for(i=1;i<n;i++){
		if(CompareCode(r,key[i],m)==0)continue;
		loc[count][1]=i-1;
		loc[count][2]=i-loc[count][0];
		loc[++count][0]=i;
		r=key[i];
	}
	// last k-mer
	loc[count][1]=n-1;
	loc[count][2]=n-1-loc[count-1][0];
	count++;

	// total info
	loc[count][0]=n;
	loc[count][1]=count;
	loc[n][1]=count;


	printf("\n\n\n\n");
	printf("k-mer\tBegin\tEnd\tSubtotal\n");
	printf("----------------------------------------\n");
	for(i=0;i<count;i++){
		decode(key[loc[i][0]],K,str);
		printf("%s\t%ld\t%ld\t%ld\n",str,loc[i][0],loc[i][1],loc[i][2]);
	}
	printf("----------------------------------------\n");
	printf("total:\t%ld\t%ld\n",n,count);

	return 1;
}
```

```
//
int SummaryKEY(CodeInt *key, lint *index, lint n, lint loc[])
{
	lint i,j;
	kint m=(K+3)>>2;
	CodeInt r;
	lint count=0;
	char str[K+1];

	loc[0]=0;
	loc[n]=n;
	r=key[0];

	for(i=1;i<n;i++){
		if(CompareCode(r,key[i],m)==0)continue;
		loc[++count]=i;
		r=key[i];
	}
	// last k-mer
	count++;

	// total info
	loc[count]=n;
	loc[n]=count;


	printf("\n\n\n\n");
	printf("k-mer\tBegin\tEnd\tSubtotal\n");
	printf("----------------------------------------\n");
	for(i=0;i<count;i++){
		decode(key[loc[i]],K,str);
		printf("%s\t%ld\t%ld\t%ld\n",str,loc[i],loc[i+1]-1,loc[i+1]-loc[i]);
	}
	printf("----------------------------------------\n");
	printf("total:\t%ld\t%ld\n",n,count);

	return 1;
}




int GetData(CodeInt *key, lint *index)
{
```

```
CodeInt r;
FILE *fp;
char filename[2][20]={"solexa_100_170_1.fa","solexa_100_170_2.fa"};
int      len = N;
char* read;
char    line[N+1];
time_t time1, time2;
kint k=K;    /* 1<=k<=100 */
lint count=0;
lint n = M*(N-K+1);
kint m = (K+3)>>2;

fp = fopen(filename[0], "r");
if(fp == NULL){
      printf("error on open file : %s \n", filename[0]);
      exit(FAILED);
}
time1=time(NULL);
/* key[strlen(key)-1]='\0' */
while( fgets(line, len+1, fp) != NULL )
{
      if(strlen(line)==100){
            //printf("%s", line);
            InsertKEY(key,index,n,line,count,k);
            count++;
            //if(count>=500000)break;
      }else if(strlen(line)<100 && strlen(line)>100){
            printf("%s",line);
      }
}
time2=time(NULL);
printf("The difftime is: %f seconds\n",difftime(time2,time1));
fclose(fp);

//getchar();



fp = fopen(filename[1], "r");
if(fp == NULL){
      printf("error on open file : %s \n", filename[1]);
      exit(FAILED);
}
time1=time(NULL);
```

```
        /* key[strlen(key)-1]='\0' */
        while( fgets(line, len+1, fp) != NULL )
        {
                if(strlen(line)==100){
                        //printf("%s", line);
                        InsertKEY(key,index,n,line,count,k);
                        count++;
                        //if(count>=1000000)break;
                }else if(strlen(line)<100 && strlen(line)>100){
                        printf("%s",line);
                }
        }
        time2=time(NULL);
        printf("The difftime is: %f seconds\n",difftime(time2,time1));
        fclose(fp);
}









int main()
{
        CodeInt *key,r;
        lint *index;
        kint k=K;    /* 1<=k<=100 */
        lint n = M*(N-K+1);
        kint m = (K+3)>>2;
        lint *loc;

        clock_t clock1,clock2,clock3;



        clock1=clock();

        key=InitKEY(n);
        index=InitINDEX(n);

        // read files to get data
        GetData(key,index);
```

```
    // sort data
    printf("\n\nsort key(k=%d,m=%d,n=%ld):\n",k,m,n);
    SortKEY(key,index,n,m);
    clock2=clock();
    printf("running time : %f seconds\n",(double)(clock2-clock1)/CLOCKS_PER_SEC);

    getchar();

    loc=InitINDEX(n+1);
    SummaryKEY(key,index,n,loc);
    free(loc);

    // search k-mer
    Searching(key,index,n);



    //TravelKEY(key,index,n,k);

    DestroyKEY(key,index);

    return SUCCESS;
}
```