DNA 序列的 k-mer index 问题

学　校　　华南理工大学

学生姓名　贺煜辉，沈跃佳，谢言

目录

DNA 序列的 k-mer index 问题

摘要

DNA 序列的 k-mer 的数据管理是生物信息管理中一个非常基础且重要的问题。解决这一问题，不仅对 DNA 片段的对比工作提供了很大的便利并且将极大的加快各类生物信息的处理速度。

针对 DNA 序列的 k-mer 索引问题，此文通过对问题的深入分析以及对数据的处理和理论推导，综合利用哈希表算法和 KMP 查找算法，建立起了一套算法模型。

模型简述如下：

采用哈希表对 k-mer 建立索引，当临界值$(10) \leq k \leq 100$时，用 KMP 算法查找确定具体行号和位置号。当$k \leq$ 临界值(10)时，用哈希查找确定行号，用 KMP 算法查找确定位置号。

通过对数据的深入分析，确定了临界值等于 10，因而高效地结合两种算法，模型的索引时间复杂度、索引空间复杂度为 O(n)，查找时间复杂度、空间复杂度为 O(1)。

最后，用 Visual Studio 2013 进行 c 语言编程，将算法模型进行了实际检验，并且与其他各种算法模型做比较，不仅验证了模型中理论推导的准确性，并且验证了算法模型的高效性。

此模型组合了哈希表与 KMP 算法，相比于字典树、普通哈希表、顺序查找等方法，兼顾了内存占用和查询速度等各个方面，具有查询速度极快，内存占用量较小，支持全部 k 值，建立索引迅速的优点，并在实际程序检验中有优秀的表现。

关键词：DNA 序列；k-mer 索引；哈希表；KMP 算法；C 语言

一、问题重述

随着生物科技的快速发展，对生物信息进行高效的管理成为了当务之急，DNA序列的k-mer便是一种在生物研究中广泛应用的生物信息。

现给出100万条DNA序列，每条序列由100个碱基（有四种类型：A、T、G、C）组合而成，。对这些DNA序列的k-mer制定一种数据索引方法，使得可以在后面的操作中快速查找出某个k-mer在数据中的具体位置（行号和位置号）。

问题及要求：

（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

依据题目要求，需要对给定的k建立索引，并且在建立之后能对目标k-mer快速查找并输出其具体位置。将查询速度做为第一考量，兼顾其他各个要求。如内存占用和建立索引时间，以及支持的k的范围。

为了满足查询速度快的要求，考虑建立哈希索引表，采用哈希查找与 KMP匹配算法相结合的方式使得算法可以兼顾内存占用小与查询速度快。随后以 k为变量，对于数据和算法进行分析，进一步优化、细化模型。并在建立模型后，通过理论推导算出算法复杂度，通过实际检验对模型做出评价。

三、模型假设

（1） 假设K的范围为2到100，K为1的时候没有意义。

（2） 每次建立索引，只需支持一个 k 值。

（3） 假设 8G 内存可以全部占用。

四、符号说明

k：k-mer 中的 k 值，即：k-mer 的长度

M：待查找的 k-mer 序列

M_1：待查找的 k-mer 序列的前十位（k>10 时）

X/X_1：k-mer 存在的行号

Y：k-mer 存在的位置号

(x，y):行号和位置号的二维数组，x 为行号，y 为位置号

f：哈希函数

S:建表时每次读取的 k-mer

L((k)：对 k 建立索引后，每个索引表头的链表的平均长度

T(n)：时间复杂度

S(n)：空间复杂度

五、模型建立

5.1、问题分析

由于要将查询速度作为第一考量，因此需要较为细致的索引表，采取建立哈希表索引算法模型。采取 KMP 匹配算法来寻找具体的位置。

问题的另一关键点是：针对一个 k 建立一次索引。

5.2、索引算法模型

5.2.1、给出哈希表相关定义

哈希表：

哈希表是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做哈希函数。在本题中，关键码就是每个 k-mer。

5.2.2、基本步骤

（A）. 寻找哈希函数

由于 DNA 序列均有 A、C、G、T 四种碱基排列组合得到的，所以我们令哈希

函数f为：

$$f(A) = 00，f(C) = 01，f(G) = 10，f(T) = 11 \quad (1)$$

这样对于每一个 k-mer,都有唯一的一组长度为 2k 位的二进制数与其对应。

例如：k=4 时，$f(ATCG) = 00110110$（A 对应前两个数字 00，T 对应第 3,、4 个数字 11，依次类推）；k=5 时，$f(ATCGC) = 0011011001$。

（B）. 建立寻址方法

直接寻址法：

由于当 k 固定时,每一种 k-mer 与每一个长度为 2k 的二进制数是双射的(即一一对应)，且均为连续的。因此采用直接寻址法。以 k=4 为例，建立4^k个表头，编号为从 00000000 到 11111111，每读取一个 k-mer 序列 S，将其行号存于表头编号为 f（S）后面的链表中。

数据结构如下：

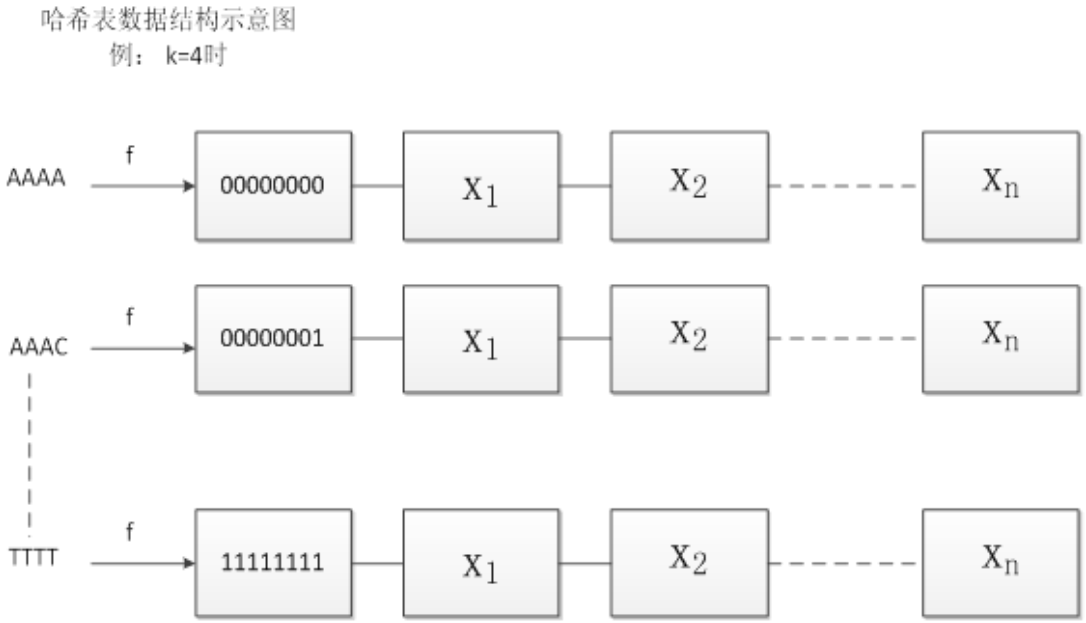

图 1 哈希表数据结构示意图

注：每个表头后面的链表长度不一样。

5.3、查找算法模型

为使查找速度最快，采用 KMP 算法对字符串进行匹配查找。

5.3.1、KMP 字符串匹配相关概念

KMP 算法是一种改进的字符串匹配算法，关键是利用已经得到的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。

5.3.2、主要步骤示例：

从 S 中查找 T（S 和 T 末尾的“\0”代表字符串的结束符），如图 2 所示：

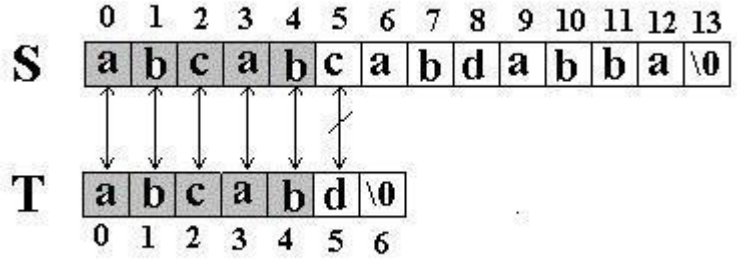

图 2 KMP 算法的第一趟匹配过程

先从S[0]与T[0]比较，S[0] = T[0]，则比较S[1]与T[1]…。当比较当发现 S[5]不等于 T[5]后，并不从头再寻找，由于在前面的比较中，已经保留了 S 与 T 匹配成功位置上的信息，发现T[0]T[1] = T[3]T[4] = S[3]S[4]，则将 T 后错三位，直接比较T[2]与S[5]，如图 3 所示。从而快速匹配成功。

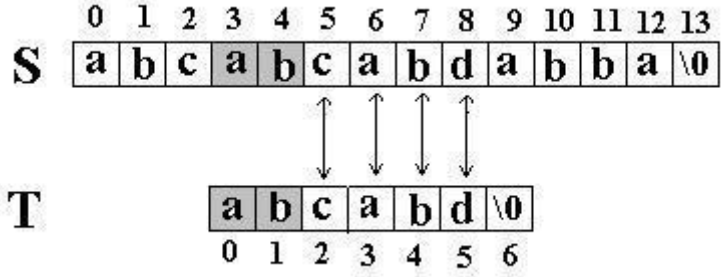

图 3 KMP 算法的第二趟匹配过程

而常规的匹配算法只会将 T 后错一位，将 T[0]与 S[1]进行比较。这使得以及读取的 T 的信息没有利用起来。而 KMP 弥补了这一点，高效地利用信息从而达到快速匹配。

5.4、数据分析与算法模型优化

5.4.1、每一种 K-mer 出现次数分析

对于给定的 k 值，理论上 k 个碱基能组成的 k-mer 种类为4^k。对于一百万条长度均为 100 的 DNA 来讲，假设每个 k-mer 不一样，实际上出现的种类最多为

$$100\ 0000 \times (101 - \mathrm{k}) \qquad (2)$$

设

$$f(k) = 4^k - 1000000 \times (101 - k) \quad (3)$$

f(k)与 k 的关系如图 4 所示（f(k)为纵坐标，k 为横坐标）：

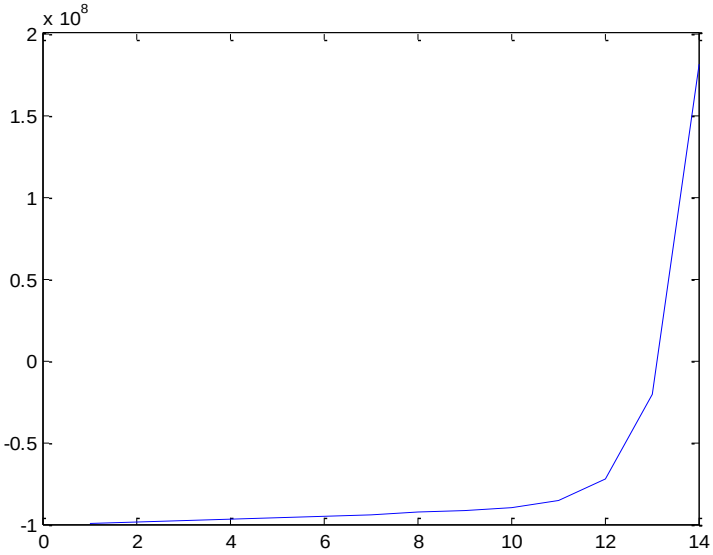

图 4 f(k)的函数图像

如图 3 所示，当 k 大于 13 以后，$4^k \gg 1000000 \times (101 - k)$，也就是说，此时实际上出现的 k-mer 序列的种类远远小于理论上的种类，如果对 k 大于 13 建立哈希表的话,将会出现有很多表头后面链表长度为 0 的情况(因为有些 k-mer 种类没有出现过)，会造成内存空间的极大浪费。

5.4.2、每类 K-mer 的平均数组长度分析。

（1） 当$k \leq 4$，由于 k-mer 种类过少，假设每条 DNA 中都含有所有种类的 k-mer。则每个 k-mer 表头对应的链表的长度为

$$L(k) = 1000000 \qquad (k \leq 4) \quad (4)$$

当$k > 4$时，假设每条 DNA 产生的 k-mer 无重复，则平均每种 k-mer 对应的表头后的链表的长度为

$$L(k) = \frac{1000000 \times (101 - k)}{4^k} \qquad (k > 4) \quad (5)$$

L(k)的函数图像如图 5 所示，具体数据如表 1 所示：

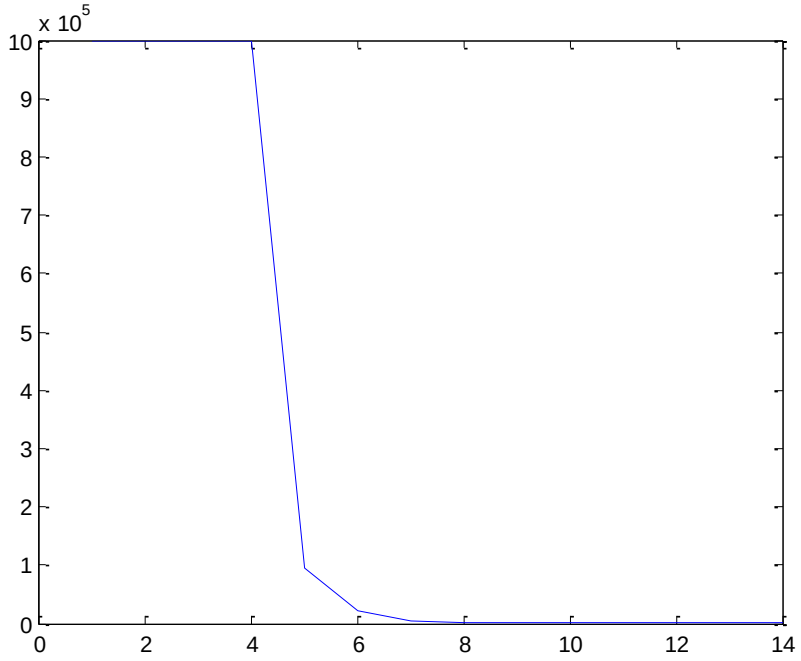

图 5　L(k)的函数图像

表 1　k 取不同值时 L(k)的大小

k	8	9	10	11	12	13
平均链表长度 L(k)	1419	351	86	21	5	1

通过对程序的实际测试，对于 30 万条 DNA 建表后 k=10 时测试的链表长度大约为 20-35，换算成 100 万条 DNA，链表长度大约为 90 左右

由此可见，当 k 过大时，链表长度过短，索引表过于稀疏，造成浪费，当 k 过小时，链表长度过长，不方便后续的具体位置的查找工作。

5.4.3、算法模型优化

为了避免索引表过于稀疏，造成内存浪费，以及索引表过于繁琐，减慢查询速度的情况，通过对表 1 的分析，发现 k 取 10 时数组长度较为合适，选取 10 作为临界值。

以下为优化后的模型：

①当 k<10 时，直接建立4^k个表头的哈希表。查找时，读取 M（待查找的 k-mer），到关键码为 f（M）的链表中提取行号，再用 KMP 算法到该行查找 M 出现的具体位置。

②当 k>=10 时，建立4^{10}个表头的哈希表。查找时，只对 M（待查找的 K-mer）

的前 10 位即 M1 进行读取，到关键码为 f（M_1）的数组中提取行号，再到每个可能的行里用 KMP 算法匹配查找 S 并输出具体位置。（由于 k 大于 10 后，每个链表长度大概为 80，查找并不会耗费太多时间）。

5.4.4、算法模型流程图

（1） 索引表建立的具体算法流程如图 6 所示：

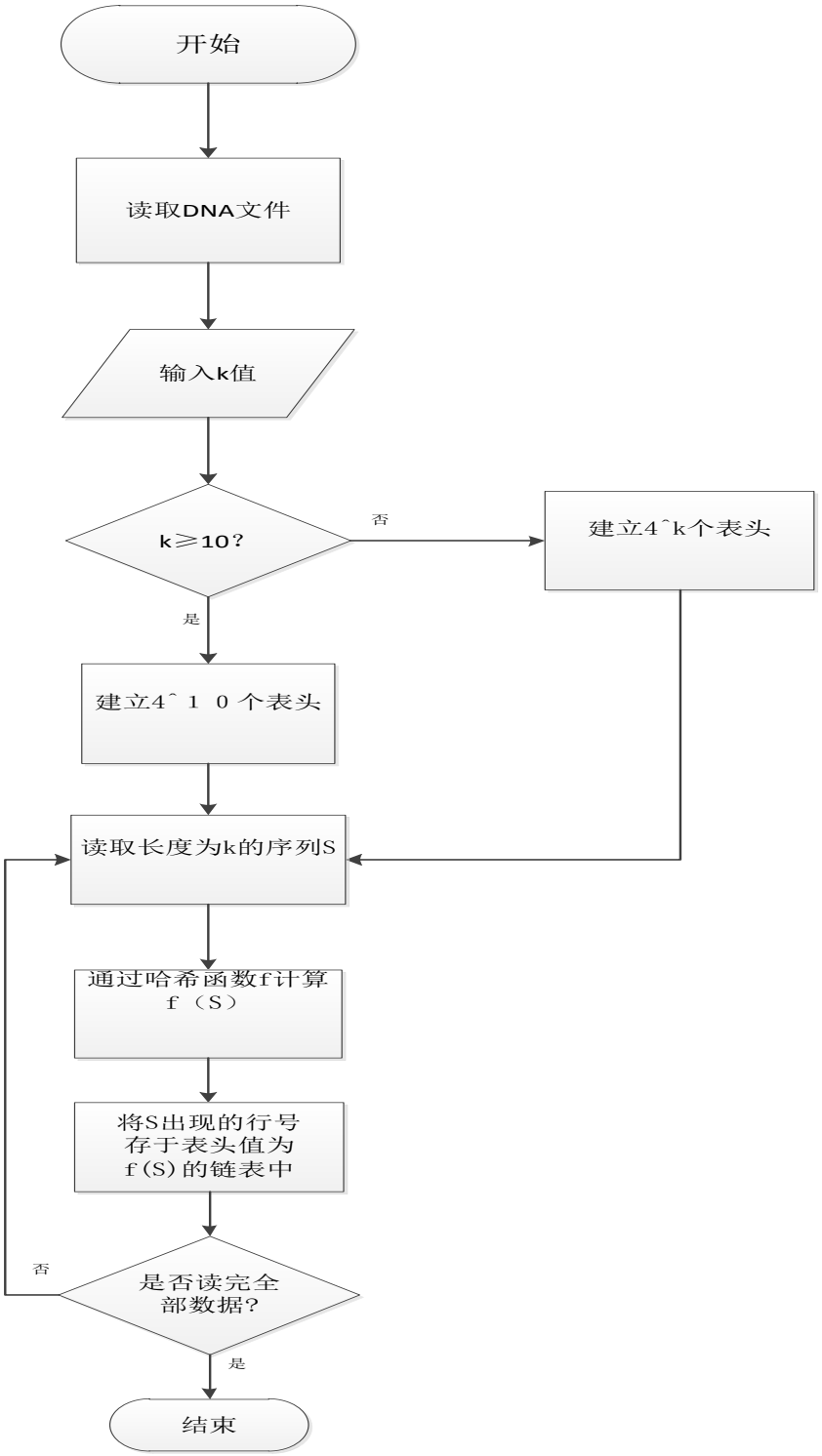

图 6 建立索引表的算法流程图

（2） 查找算法流程图如图 7 所示

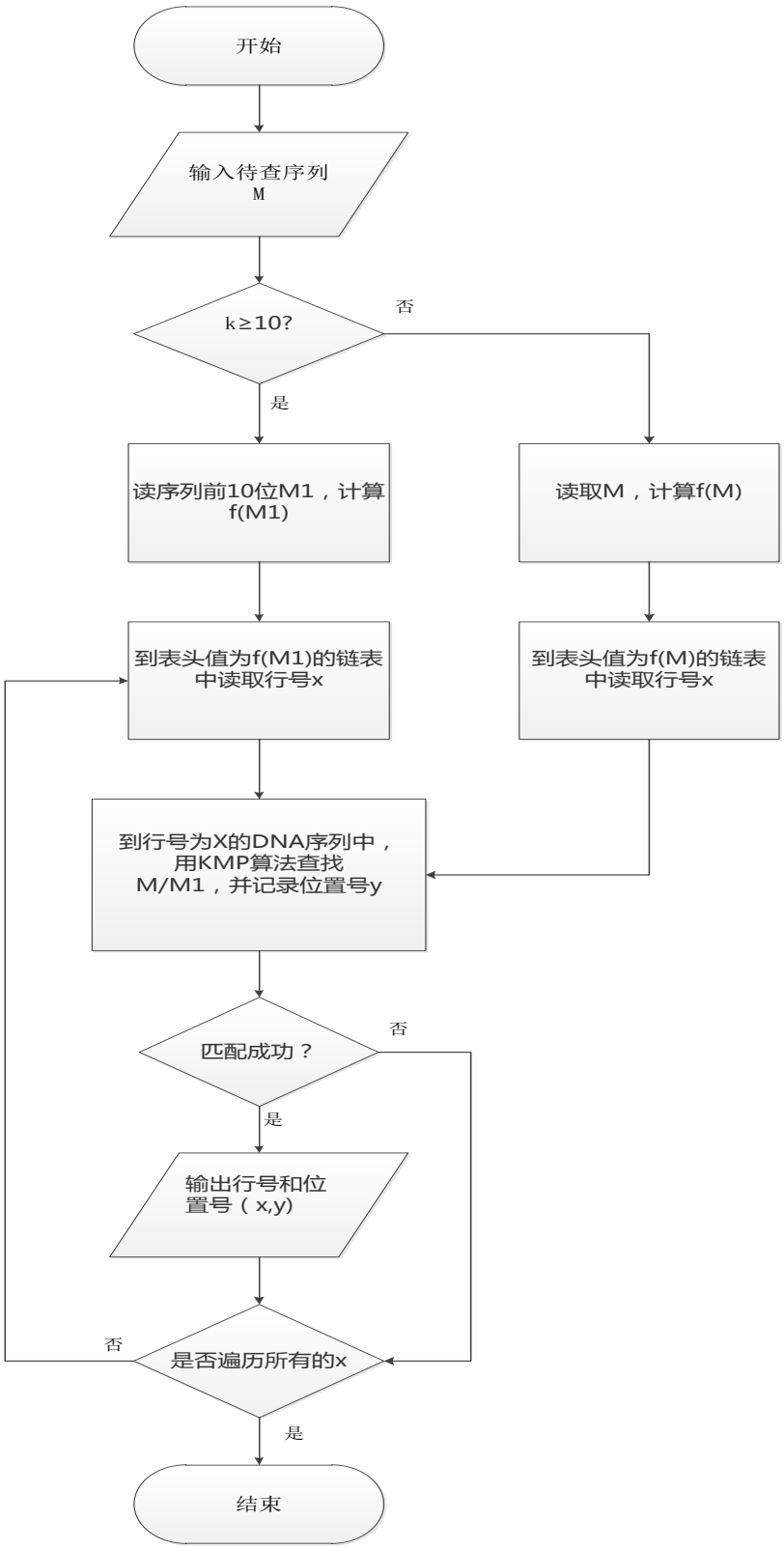

图 7 查找算法流程图

（3） 总流程图如图 8 所示

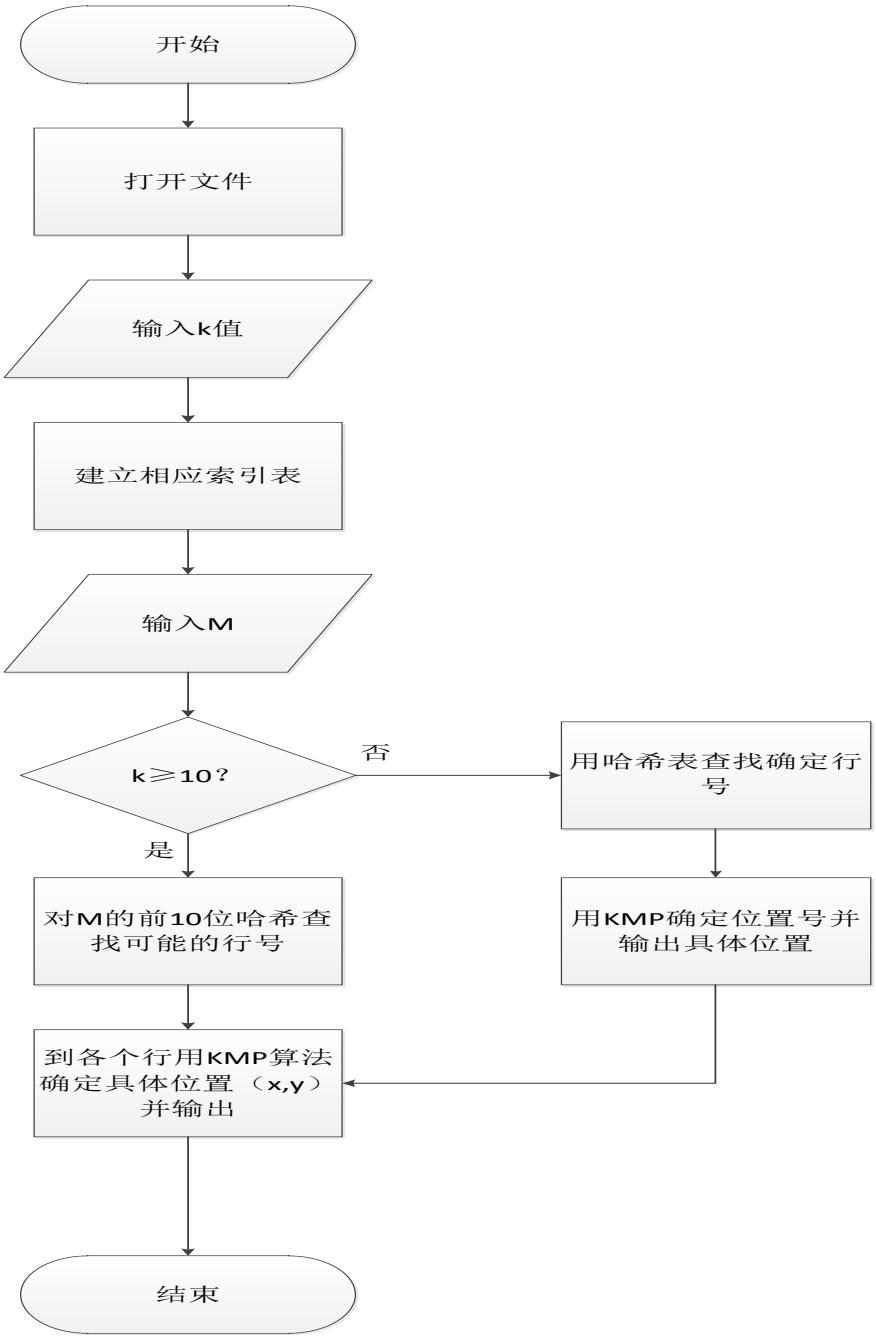

图 8 模型总流程图

六、模型求解

6.1、建立索引表的复杂度分析

6.1.1、时间复杂度

哈希表实际上是对所有数据进行遍历，所以其时间复杂度和数据量有关，所以时间复杂度为:

$$T(n) = O(n) \qquad (6)$$

6.1.2、空间复杂度

由于不论 k 的取值为多少，哈希表消耗的内存都是较为固定的常值（后面的内存分析会详细说明），和 k 没有直接关系，但是和总数据量成线性关系，所以空间复杂度为：

$$S(n) = O(n) \quad (7)$$

6.2、使用索引查询的复杂度分析

索引查询分为哈希表查询行号和 KMP 算法查询位置号两部分。

6.2.1、时间复杂度

A. KMP 查询位置号

已知主字符串为 S，待匹配的字符串串 T，由于 KMP 算法的思想是主串不回溯的简化算法，匹配的时候只会将 T 不断右移进行比较，所以字符比较最多的情况就是遍历 S 中所有的元素，则时间复杂度为 O（LenthS）,即 KMP 算法的时间复杂度为 O（n）。对于本模型来讲，查找时 DNA 长度固定为 100 则比较次数平均为 100*80 次，与 k 的选取以及总数据量无直接关系，所以查询位置号的时间复杂度为 O(1)。

B. 哈希表查询行号

哈希表是通过计算关键码值（待查找 k-mer 对应的哈希函数值）来定位元素位置，所以只需一次即可，复杂度为 O(1)。

C. 索引查询总时间复杂度:

$$T(n) = O(1) + O(1) = O(1)) \quad (8)$$

6.2.2、空间复杂度

A. 哈希查找

对于哈希查询部分，只需提取数据，空间复杂度为:

$$S_1(n) = O(1) \quad (9)$$

B. KMP 匹配查找

由于 KMP 算法在计算时，占用内存用来保存比较串的信息，所以其空间复杂度为 O（lenthT）.对于本模型来讲，长度为常值，所以空间复杂度为:

$$S_2(n) = O(1) \quad (10)$$

故索引查询空间复杂度为：

$$S(n) = O(1) + O(1) = O(1) \quad (11)$$

6.3、理论内存占用分析

由于本模型对于$k > 10$均采用 k=10 建立索引表，所以以 k=10 为例分析内存占用空间。

在索引表中，k=10 的情况下，一共有4^{10}个表头，平均每个表头后的链表长度为 90，链表的每一个节点占用 8B（即 8 字节）内存，。则：

理论总占用内存$= 4\text{^}10 \times 90 \times 8B = 0.703GB$

由于程序在运行时会占用一些内存空间，故本数据和实际运行有些许差别。

6.4、模型实际效果

注：由于电脑内存较小，以下只对前 30 万条 DNA 进行读取、建表和检索。

运行环境为 windows7 64 位操作系统下，CPU 为 Intel Core i3 处理器。

6.4.1、查询时间

k=7 的查询时间为 0.68s。

k=10 的查询时间为 0.000s

k=23 时的查询时间 0.00000s

基本上 k 大于 10 之后查询时间都为 0.00000

6.4.2、支持 k 值范围

k 支持 2-100。

6.4.3、建立索引表的时间

k=5 耗时 11.46s 建立索引。

k=7 耗时 13.17 秒建立索引。

k=21，耗时 17.96 秒建立索引。

k=80，耗时 18.07 秒建立索引

基本上当 k 大于 10 之后建表时间普遍在 18s 附近，而 k 较小时建表时间要小于 18s。

七、模型的比较与评价

7.1、模型的比较

7.1.1、字典树介绍

字典树是一种用于快速检索的多叉树结构。其基本思想就是把要查找的关键

字看作一个字符序列，并根据构成关键字的字符的先后顺序构造用于检索的树结构。

所以，类比于字典，字典树在检索时，其步骤如下：

(1) 从根结点开始一次搜索；

(2) 取得要查找关键字的第一个字符，并根据该字符选择对应的子树并转到该子树继续进行检索；

(3) 在相应的子树上，取得要查找关键字的第二个字符，并进一步选择对应的子树进行检索；

(4) 迭代过程；

(5) 在某个结点处，关键字的所有字符已被取出，则读取附在该结点上的信息，即完成查找。

图 9 为利用字典树进行关键字查询的一个流程图：

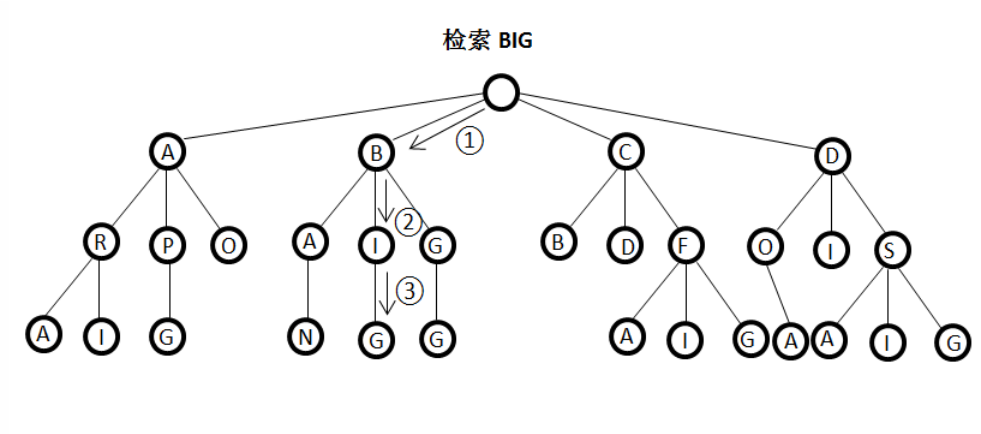

图 9 字典树查询关键字流程图

7.1.2、顺序查找

顺序查找法是一种很常见的程序设计查找方法，就是在一个已知无(或有序）序队列中找出与给定关键字相同的数的具体位置。

其基本原理就是：从表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字和给定值 K 相比较。若当前扫描到的结点关键字与 K 相等，则查找成功；若扫描结束后，仍未找到关键字等于 K 的结点，则查找失败

7.1.3、各种算法模型与本模型的比较

表 2 各种模型的比较结果

k	优化哈希表（本文模型）		字典树			顺序查找	
	建表时间/s	查询时间/ms	建表时间/s	查询时间/ms	消耗内存/GB	查询时间/ms	消耗内存/GB
3	33	0.6	20	0	0.5	1.4	较小，可忽略
7	39	0.68	36	0	0.9	1.5	
10	52	0	52	0	2.7	1.6	
20	53	0	83	0	3.6	1.6	
50	54	0	103	0	4.2	1.6	
70	53	0	溢出	0	5	1.6	
100	53	0	溢出	0	7.3	1.6	

7.2、模型的评价

7.2.1、本模型优点

本模型组合了哈希表与 KMP 算法，各取所长，和其他算法模型相比较，兼顾了内存占用和查询速度等各个方面，具有查询速度极快，内存占用量较小，支持全部 k 值，建立索引时间较小的优点。模型的索引时间复杂度、索引空间复杂度为 O（n）查找时间复杂度、查找空间复杂度均为 O(1)。并在实际检验中有很好的表现。

7.2.2、本模型缺点

在用程序实现算法时，由于代码语句的不规范等不完善的地方，导致程序的实际占用内存量虽然能满足要求，但是和理论内存量比较存在偏差。在内存使用量上还有提高的空间。

八、参考文献

[1] 姜启源 谢金星 叶俊.数学模型[M].4.北京：高等教育出版社，2011.

[2] 李春葆.数据结构教程[M].4.北京:清华大学出版社，2013.

附录

k-mer Index 4.cpp 源程序

需要环境：

VS2013

64 位操作系统（Linux 或其他可以满足单进程 8G 的系统）

支持单进程 8G 内存

```
#include "stdafx.h"
#include"stdlib.h"
#include"string.h"
#include"time.h"
#include"math.h"

#define FILEPATH "newfile.dat"
#define SEQLEN 100
#define SEQCOUNT 1000000
#define INDEXMAX_K 10
char SequenceString[SEQCOUNT][SEQLEN + 1];//存储所有的母序列
int ChildSeqCount;//根据k值记录子序列个数
char ChildString[SEQLEN + 1] = { '\0' };//存放子序列

struct Node//链表结点
{
    int sequence;//sequence记录子序列所在的母序列的序列号
    Node* next;
};

struct TableHead
{
    Node* head;//标志链表头部
    Node* end;//标志链表尾部
    int count;//测试用，记录此链表的节点数
};

void WriteFile();//将文件的DNA序列写入SequenceString中
unsigned int StringToNum(char string[], int k);//将子串转化为其对应的数字

void Get_next(char T[], int k, int next[]);
```

```
int Index_KMP(char S[], char T[], int k, int pos);//k为k-mer值

int InsertNode(TableHead* Head, int sequence,  unsigned int num);//往索引表中插入结点，Head为表头，sequence为母序列的序号值-1，position为子序列位置，num为子序列对应的数字，返回值为1则表明结点生成成功，返回0则表明没有生成结点
TableHead* BuildTable(int k);//建立索引表，返回值为表头，k为k-mer值
void DestroyLink(TableHead* Linkhead); //销毁链表

TableHead* Intersection(Node* LinkHead1, Node* LinkHead2);//求两个子序列都位于的母序列的交集，返回值为链表头
void Search(TableHead* Head, int k, char string[]);//查找子序列所在母序列的序号和位置,Head为索引表头，k为k-mer值，string为要查找的子序列
void SearchSolution_1(TableHead* Head, char string[],int k);//当k>=10时，选用查找方案1，string为待寻找的子串
//void SearchSolution_2(TableHead* Head, char string[], int k);//当k>=20时，选用查找方案2,string为待寻找的子串，k为k-mer值
void SearchSolution_2(TableHead* Head, char string[], int k);//当k小于10时

#include"Write.h"
#include"BuildTable.h"
#include"Matching.h"
#include"Search.h"

int main()
{
    clock_t start, finish;
    double  duration;

    start = clock();
    WriteFile();
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("DNA读取成功,用时%f seconds，请输入k值建立索引\n",duration);

    TableHead* Head;
    int k;
    scanf_s("%d",&k);
    start = clock();
    if (k >= 10)    //k>=10的时候按k=INDEXMAX_K=10建立索引
        Head = BuildTable(INDEXMAX_K);
    else
        Head = BuildTable(k);
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
```

```
    printf("索引表建立成功,用时%f seconds，请输入要查找的长度为k的子序列
\n",duration);


    printf("请输入长度为%d的子序列\n",k);
    char string[SEQLEN + 1];
    scanf_s("%s", string, _countof(string));
    while (strlen(string) != k)
    {
        printf("子序列长度不为k，请重新输入\n");
        scanf_s("%s", string, _countof(string));
    }
    start = clock();
    Search(Head, k, string);
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("查找用时%f seconds\n", duration);

    printf("请按任意键结束程序\n");
    system("pause");

    return 0;
}
```

Write.h

```
void WriteFile()
{
    FILE* fp;
    if (fopen_s(&fp, FILEPATH, "rb") != 0)
    {
        printf("can't open the file\n");
        exit(0);
    }
    else
        printf("文件打开成功\n");
    for (int i = 1; i <= 1000000; i++)
    {
        fread(SequenceString[i - 1], sizeof(char), SEQLEN, fp);
        SequenceString[i - 1][SEQLEN] = '\0';
        if (!feof(fp))
            fgetc(fp);
    }
```

```
    fclose(fp);

}
```

BuildTable.h

```
unsigned int StringToNum(char string[], int k)
{
    unsigned int num = 0;
    for (int i = 0; i<k; i++)
    {
        num = num << 2;
        switch (string[i])
        {
        case 'A':num += 0; break;
        case 'C':num += 1; break;
        case 'G':num += 2; break;
        case 'T':num += 3; break;
        default:break;
        }
    }

    return num;
}

int InsertNode(TableHead* Head, int sequence, unsigned int num)
{
    Node* node;// = (Node*)malloc(sizeof(Node));
    if (!(node = (Node*)malloc(sizeof(Node))))
    {
        printf("结点生成失败，此时母序列为%d\n",  sequence);
        //exit(0);
        getchar();
    }
    node->sequence = sequence + 1;//因为0<=sequen<SEQCOUNT，所以此处应该加上
1
    node->next = NULL;
    TableHead* p = Head + num;
    if (p->head == NULL)
    {
        p->head = p->end = node;
        p->count++;//测试用
        return 1;
    }
    else
    {
```

```
        if ((sequence+1) != p->end->sequence)//倘若该子序列所在的母序列已经记录，则不记录
        {
            p->end->next = node;
            p->end = node;
            p->count++;//测试用
            return 1;
        }
        else
        {
            free(node);
            return 0;//没有生成结点
        }

    }
}

TableHead* BuildTable(int k)
{
    TableHead* Head;
    ChildSeqCount = (int)pow(4.0, k);
    if (!(Head = (TableHead*)malloc(sizeof(TableHead)*ChildSeqCount)))
    {
        printf("索引表头建立失败\n");
        exit(0);
    }
    else
        printf("索引表头建立成功，正在建立索引表，请稍候\n");
    for (int i = 0; i < ChildSeqCount; i++)//将索引表头初始化
    {
        (Head + i)->head = (Head + i)->end = NULL;
        (Head + i)->count = 0;//测试语句
    }

    unsigned int num = 0;//将子串转化为相应的数字，00代表A，01代表C，10代表G，11代表T
     int position = 0;//记录子序列位置

    int count = 0;//测试用，测试完要删除
    FILE* fp;
    if (fopen_s(&fp, "D:\\Bakup\\桌面\\校内选拔赛题目\\B题附件\\myfile.txt", "w") != 0)//测试用
    {
        printf("文件建立失败\n");
```

```
        exit(0);
    }
    for (int i = 0; i < SEQCOUNT; i++)
    //上面那句才是真正要用的，下面这句是测试用的
    //for (int i = 0; i < 300000;i++)
    {
        count = 0;//测试用
        for (position = 1; position <= SEQLEN - k + 1; position++)//将每一个母序列中长度为k的子序列读取出来,生成链表结点，并将其位置记录到结点中
        {
            for (int j = 0; j < k; j++)
                ChildString[j] = SequenceString[i][position + j - 1];
            ChildString[k] = '\0';           //子序列读取完毕
            num = StringToNum(ChildString, k);
            count += InsertNode(Head, i, num);//count为测试用，调试完应予以修改

        }
        fprintf(fp,"第%d个母序列生成了%d个结点\n",i+1,count);//测试语句
    }

    fclose(fp);//测试用

    return Head;
}

void DestroyLink(TableHead* Linkhead)
{
    Node* p = Linkhead->head;
    Node* temp = NULL;
    Linkhead->head = Linkhead->end = NULL;
    while (p)
    {
        temp = p;
        p = p->next;
        free(temp);
    }
}
```

Matching.h

```
void Get_next(char T[], int k, int next[])//T的长度为k
{
    int i = 0; next[0] = -1;
    int j = -1;
    while (i < k - 1)
    {
        if (j == -1 || T[i] == T[j])
        {
            i++; j++;
            next[i] = j;
        }
        else
            j = next[j];
    }
}

int Index_KMP(char S[], char T[], int k, int pos)//k为k-mer值
{
    //int next[2 * INDEXMAX_K];
    int next[SEQLEN];
    Get_next(T, k, next);
    int i = pos - 1, j = 0;
    while (i <= SEQLEN - 1 && j <= k - 1)
    {
        if (j == -1 || S[i] == T[j])
        {
            i++; j++;
        }
        else
            j = next[j];
    }
    if (j >= k)
        return i + 1 - k;
    else
        return 0;//无子串T则返回0
}
```

Search.h

```
TableHead* Intersection(TableHead* LinkHead1, TableHead* LinkHead2)
{
    Node* p1 = LinkHead1->head; Node* p2 = LinkHead2->head;
    TableHead* intersection = (TableHead*)malloc(sizeof(TableHead));
    intersection->head = intersection->end = NULL;
    Node* node = NULL;
    while (p1&& p2)
    {
        if (p1->sequence == p2->sequence)
        {
            if (!(node = (Node*)malloc(sizeof(Node))))//分配失败则退出
            {
                printf("位置链表结点生成失败,请按任意键结束\n");
                getchar();
            }
            node->sequence = p1->sequence;
            node->next = NULL;
            if (intersection->head == NULL)
                intersection->head = intersection->end = node;
            else
            {
                intersection->end->next = node;
                intersection->end = node;
            }
        }
        else if (p1->sequence > p2->sequence)
            p2 = p2->next;
        else
            p1 = p1->next;
    }
    return intersection;
}

void SearchSolution_1(TableHead* Head, char string[],int k)
{
    //char temp[INDEXMAX_K + 1] = { '\0' };
    char temp[SEQLEN + 1] = { '\0' };
    for (int i = 0; i < INDEXMAX_K; i++)
        temp[i] = string[i];
    unsigned int num = StringToNum(temp, INDEXMAX_K);
```

```
    Node* p = (Head + num)->head;
    int sequence;
    while (p)//子序列string存在于母序列中
    {
        sequence = p->sequence;
        int pos = 1;
        while (pos >= 1 && pos <= SEQLEN - k + 1)
        {
            pos = Index_KMP(SequenceString[sequence - 1], string, k, pos);
            if (pos)//当pos不为0时，表示有子串
            {
                printf("存在于第%d个DNA序列，位置为%d\n ",sequence, pos);
                pos++;
            }
            else    //无子串则跳出循环
                break;
        }
        p = p->next;
    }
}


void SearchSolution_2(TableHead* Head, char string[], int k)
{
    unsigned int num = StringToNum(string, k);
    Node* p = (Head + num)->head;
    int sequence;
    int pos;
    while (p)
    {
        sequence = p->sequence;
        pos = 1;
        while (pos >= 1 && pos <= SEQLEN - k + 1)
        {
            pos = Index_KMP(SequenceString[sequence - 1], string, k, pos);
            if (pos)//当pos不为0时，表示有子串
            {
                printf("存在于第%d个DNA序列，位置为%d\n ", sequence, pos);
                pos++;
            }
            else    //无子串则跳出循环
                break;
        }
        p = p->next;
```

```
    }
}

void Search(TableHead* Head,int k,char string[])//支持的k值都大于等于10
{
    //if (k >= 10 && k < 20)
    if (k >= 10)
        SearchSolution_1(Head, string, k);
    //else
    //SearchSolution_2(Head, string, k);
    else
        SearchSolution_2(Head, string, k);
}
```

字典树源程序（Code Blocks 软件）

```
#include <bits/stdc++.h>
#include <windows.h>
using namespace std;

const int STR_LEN = 105;

typedef pair<int,int> position; // pii.first/SeqIndex 为序列编号,
pii.second/posInSeq 为相应序列中出现的位置

char CHAR_SET[] = "ATCG";
inline int idx(const char &c) {
    for(int i=0; CHAR_SET[i]; i++) if(c==CHAR_SET[i]) return i;
    return -1;
}

struct TrieNode {
    TrieNode *nxt[4];
    vector<unsigned> posList;
    TrieNode() {
        for(int i=0; i<4; i++) nxt[i] = NULL;
```

```
        }
        void add(const int &SeqIndex, const int &posInSeq) {
            posList.push_back(SeqIndex*1000 + posInSeq-1);
        }
        void show() {
            for(unsigned i=0; i<posList.size(); i++) {
                cout << posList[i]/1000 << ',' << posList[i]%1000+1 <<
' ';
            }
            cout << endl;
        }
    };
    struct KmerTrie {
        int k;
        TrieNode *root;
        KmerTrie() {
            root = new TrieNode();
        }
        void insert(const char *str, const int &SeqIndex) {
            for(int i=0; i+k<=100; i++) {
                insert(str+i, SeqIndex, i+1);
            }
        }
        void insert(const char *substr, const int &SeqIndex, const int
&posInSeq) {
            TrieNode *target = findNode(substr, true);
            target->add(SeqIndex, posInSeq);
        }
        TrieNode *query(const char *str) {
            return findNode(str, false);
        }
```

```
    TrieNode *findNode(const char *str, const bool &create) {
        TrieNode *now = root;
        for(int i=0; i<k; i++) {
            if(now->nxt[idx(str[i])] == NULL) {
                if(!create) return new TrieNode();
                now->nxt[idx(str[i])] = new TrieNode();
            }
            now = now->nxt[idx(str[i])];
        }
        return now;
    }
};

char buf[STR_LEN];
KmerTrie kmer;
void readAndInsert() {

    cout << "请输入 k 值：";
    cin >> kmer.k;

    DWORD t_begin, t_end;
    t_begin = GetTickCount();

//    普通文件的读取方法
    FILE* input = fopen("dna.txt", "r");
    for(int i=1; fscanf(input, "%s", buf)!=EOF; i++) {
        kmer.insert(buf, i);
        if(i%10000 == 0) printf("%d\n", i);
    }
    fclose(input);
```

```
    //    fa 文件读取方法
    //    FILE* input = fopen("solexa_100_170_1.fa", "r");
    //    for(int i=1; fgets(buf, STR_LEN, input)!=NULL; i++) {
    //        fgets(buf, STR_LEN, input);
    ////          cout << buf << endl;
    //        for(int j=0; j<100; j++) {
    //               if(buf[j]!='A' && buf[j]!='T' && buf[j]!='C' &&
buf[j]!='G') {
    //                cout << buf << endl;
    //                return;
    //            }
    //        }
    //        buf[100] = '\0';
    //        kmer.insert(buf, i);
    //        if(i%10000 == 0) printf("%d\n", i);
    //    }
    //    fclose(input);

        t_end = GetTickCount();
        cout << "Build Time: " << t_end-t_begin << "ms" << endl;
    }

    void query() {
        while(cin >> buf) {

            DWORD t_begin, t_end;
            t_begin = GetTickCount();

            TrieNode *result = kmer.query(buf);

            t_end = GetTickCount();
```

```
            cout << "Query(" << t_end-t_begin << "ms): ";
system("pause");
            result->show();

        }
    }

    int main() {
        readAndInsert();
        query();
        return 0;
    }
```

顺序查找（Code Blocks 软件）

```
    #include<iostream>
    #include<fstream>
    #include<cstring>
    #include <windows.h>
    using namespace std;

    const char*path = "dna.txt";
    char*dna[1000000] = { 0 };
    int main() {
        char text[110] = { 0 };
        char sequence[110] = { 0 };

        fstream R;
        while (cin.getline(text, 110)) {
            DWORD t_begin, t_end;
            t_begin = GetTickCount();
```

```
        int line = 0;
        int pos = 0;
        R.open(path, ios::in);
        R.getline(sequence, 110);
        line++;
        while (!R.eof()) {
            char*p = strstr(sequence, text);
            if (p != 0) {
                pos = p - sequence;
//                cout << "line: " << line << endl;
//                cout << "pos: " << pos << endl;
//                system("pause");
            }
            R.getline(sequence, 110);
            line++;
        }
        R.close();

        t_end = GetTickCount();
        cout << "Time cost " << t_end-t_begin << "ms " << endl;
    }
}
```