《数据结构与算法实验》第 8 次实验

学院： 专业： 年级：
姓名 学号： 日期： 2022 年 5 月 2 日

第一部分 验证实验

一、 实验目的

1. 掌握串的顺序存储结构。
2. 验证顺序串及基本操作的实现，掌握字符串的操作特点。
3. 掌握对称矩阵的压缩存储方法。
4. 验证对称矩阵的压缩存储的寻址方法。

二、 实验内容

1. **串操作的实现：**
   - 定义一个包含串的长度、拼接、比较大小等基本操作的头文件函数原型。
   - 实现串的求长度、拼接、比较大小等基本操作。
2. **对称矩阵的压缩存储：** 建立一个 n×n 的对称矩阵 A。
   - 将对称矩阵用一维数组 SA 存储。
   - 在数组 SA 中实现对矩阵 A 的任意元素进行存取操作

三、 设计编码

1. 串操作的实现

(1)str.h

```h
#ifndef Str_H
#define Str_H
int strlen(char *s);
char *strcat(char *s1,char *s2);
int strcmp(char *s1,char *s2);
#endif
```

(2) str.cpp

```cpp
#include "str.h"

int strlen(char *s)
{
    char *p = s;
    int len = 0;
    while (*p != '\0')
    {
        len++;
        p++;
    }
    return len;
}

char *strcat(char *s1,char *s2)
{
    char *p = s1, *q = s2;
    while (*p != '\0')
        p++;
    while (*q != '\0')
    {
        *p = *q;
        p++;
        q++;
    }
    *p = '\0';
    return s1;
}

int strcmp(char *s1,char *s2)
{
    char *p = s1,*q = s2;
    while (*p != '\0' && *q != '\0')
    {
        if (*p > *q) return 1;
        else if (*p < *q) return -1;
        else {p++;q++;}
    }
    if (*p == '\0' && *q == '\0') return 0;
    if (*p != '\0') return 1;
    if (*q != '\0') return -1;
}
```

(3)str_main.cpp

```cpp
#include <iostream>
#include "str.h"
using namespace std;

int main()
{
    char ch[20]="I love",*str="China!";
    cout<<strlen(ch)<<endl;
    cout<<strlen(str)<<endl;
    cout<<strcmp(ch,str)<<endl;
    cout<<strcmp(str,ch)<<endl;
    strcat(ch,str);
    for (int i = 0; ch[i] != '\0'; i++)
        cout<<ch[i];
    cout<<endl;
    return 0;
}
```

2. 对称矩阵的压缩存储

```cpp
#include <iostream>
using namespace std;
const int N = 5;

int main( )
{
    int A[N][N], SA[N * (N + 1) / 2] = {0};
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j <= i; j++)
            A[i][j] = A[j][i] = i + j;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            cout<<A[i][j]<<"   ";
        cout<<endl;
    }
    for (i = 0; i < N; i++)
        for (j = 0; j <= i; j++)
            SA[i * (i - 1)/2 + j] = A[i][j];        //压缩存储
    cout<<endl<<"请输入行号和列号：";
    cin>>i>>j;
    cout<<endl<<i<<"行"<<j<<"列的元素值是：";
```

```
    i--;
    j--;
    if (i >= j)
        cout<<SA[i * (i - 1)/2 + j]<<endl;
    else
        cout<<SA[j * (j - 1)/2 + i]<<endl;
    return 0;
}
```

四、 运行与测试

1. 串操作的实现

```
7
6
1
-1
I love China!

--------------------------------
Process exited after 0.01165 seconds with return value 0
请按任意键继续. . .
```

2. 对称矩阵的压缩存储

```
0   1   2   3   4
1   2   3   4   5
2   3   4   5   6
3   4   5   6   7
4   5   6   7   8

请输入行号和列号：4 3

4行3列的元素值是：5

--------------------------------
Process exited after 10.93 seconds with return value 0
请按任意键继续. . .
```

第二部分 设计实验

一、实验目的

通过共计 6 个实验，实现对字符串、模式匹配相关算法的理解。

二、实验内容

1.凯撒密码：参考课本 p92

凯撒密码是一种简单的信息加密方法，通过将信息中每个字母在字母表中向后移动常量 k 以实现加密。例如，如果 k 等于 3，则对待加密的信息，每个字母都向后移动 3 个字符，以小写字母为例：a 替换为 d，b 替换为 e， 以此类推。字母表尾部的字母绕回到开头， 因此,x 替换为 a，y 替换为 b。具体步骤如下：

- 求待加密字符串 ch 在字母表中的位移量：ch-‘a’。
- 向后移动 k 个位移量：ch-‘a’+k。
- 如果超过字母表的尾部，则绕回开头：(ch-‘a’+k)%26。
- 求结果位移量对应的小写字母：‘a’+(ch-‘a’+k)%26。

2.BF 算法：参考课本 p81

- 给定两个字符串 S 和 T，在主串 S 中寻找子串 T 的过程称为模式匹配,T 称为模式。如果匹配成功，返回 T 在 S 中的位置；如果匹配失败，返回 0。
- 这是一种带回溯的匹配算法，简称 BF 算法，其基本思想是：从主串 S 的第一个字符开始和模式 T 的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，从主串 S 的第二个字符开始和模式 T 的第一个字符进行比较，重复上述过程，直至 S 或 T 中所有字符比较完。若 T 中的字符全部比较完毕，则匹配成功，返回本趟匹配的开始位置；否则匹配失败，返回 0。

3.KMP 算法：参考课本 p83

- BF 算法简单但效率较低，一种对 BF 算法做了很大改进的模式匹配算法是 KMP 算法，其基本思想是主串不进行回溯。

4. 求公共子串：参考课本 p96,5(1)

• 模式匹配是严格匹配，强调模式在主串中的连续性。例如，模式“bc” 是主串“abcd”的子串，而“ac”就不是主串“abcd”的子串。但在实际应用中有时不需要模式的连续性。例如，模式“哈工大”与主串“哈尔滨工业大学”是非连续匹配的，称模式“哈工大”是主串“哈尔滨工业大学”的子序列。

• 要求设计算法，判断给定的模式是否为两个主串的公共子序列。

5. 统计文本中的单词个数：参考实验书 p200

文本可以看成是一个字符序列，在这个序列中，有效字符被空格分隔为一个个单词，设计算法统计文本中单词的个数。

• 被处理的文本内容可由键盘键入。

• 可以读取任意文本内容，包括英文、汉字、数字等，单词的判定以空格为界。

• 设计算法统计文本中的单词个数。

• 分析算法的时间性能。

6. 幻方：参考实验书 p200

幻方在我国古代称为“纵横图”。它是在一个 n×n 的矩阵中填入 1~n^2 的数字（n 为奇数），使得每一行、每一列每条对角线的累加和都相等。

• 设计数据结构储存幻方。

• 设计算法完成任意 n 阶幻方的填数过程。

• 分析算法的时间性能。

三、 设计编码

1.凯撒密码-Caesar.cpp

```cpp
#include <iostream>
#include <cstring>
#include <iomanip>
using namespace std;

void Caesar(string &a,int k)
{
```

```
    int l = a.length();
    for (int i = 0; i < l; i++)
    {
        a[i] = 'a' + (a[i] - 'a' + k)%26;
    }
}

int main()
{
    string a;
    int k;
    cout<<"输入原始字符串：<<endl;
    cin>>a;
    cout<<"输入移动量 k：<<endl;
    cin>>k;
    Caesar(a,k);
    cout<<"加密后字符串为：<<endl;
    cout<<a;
}
```

2. BF 算法-BF.cpp

```
#include <iostream>
#include <cstring>
#include <iomanip>
using namespace std;

int BF(const char S[],const char T[])
{
    int i=0,j=0;
    while ((S[i] != '\0') && (T[j] != '\0'))
    {
        if (S[i] == T[j])
        {
            i++;
            j++;
        }
        else
        {
            i = i - j + 1;
            j = 0;
        }
    }
    if (T[j] == '\0') return i-j+1;
```

```
    else return 0;
}

int main()
{
    char S[20],T[20];
    cout<<"输入主串 S"<<endl;
    cin>>S;
    cout<<"输入子串 T"<<endl;
    cin>>T;
    cout<<BF(S,T)<<endl;
}
```

3.KMP 算法-KMP.cpp

```
#include <iostream>
#include <cstring>
#include <iomanip>
using namespace std;
const int N = 100;

void GetNext(const char T[],int next[])
{
    next[0] = -1;
    int j = 0,k = -1;
    while (T[j] != '\0')
    {
        if (k = -1||T[j] ==T[k])
        {
            j++;
            k++;
            next[j] = k;
        }
        else k = next[k];
    }
}

int KMP(char S[],char T[],int next[])
{
    int i=0,j=0;
    while (T[j] != '\0' && S[i] != '\0')
    {
        if (T[j] == S[i]||j == -1)
        {
```

```
            i++;
            j++;
        }
        else
        {
            j = next[j];
            if (j == -1)
            {
                i++;
                j++;
            }
        }
    }
    if (T[j] == '\0') return i-j+1;
    else return 0;
}

int main()
{
    char S[N],T[N];
    cout<<"S:"<<endl;
    cin>>S;
    cout<<"T:"<<endl;
    cin>>T;
    int next[N];
    GetNext(T,next);
    cout<<KMP(S,T,next)<<endl;
}
```

4. 求公共子串-commonstring.cpp

```
#include <iostream>
#include <iomanip>
#include <cstdio>
using namespace std;

int common(string a,string b)
{
    int l1=a.length(),l2=b.length();
    for (int i = 0; i < l1; i++)
    {
        int k = 0;
        for (int j = i; j < l1; j++)
        {
```

```
            if (a[j] == b[k])
            {
                k++;
                if (k == l2) return 1;
            }
        }
    }
    return 0;
}

int main()
{
    string a,b;
    cout<<"input S:"<<endl;
    cin>>a;
    cout<<"input T:"<<endl;
    cin>>b;
    cout<<common(a,b);
    return 0;
}
```

5. 统计单词数-count.cpp

```
#include <iostream>
#include <cstring>
#include <iomanip>
#include <stdio.h>
using namespace std;
const int N = 100;

int main()
{
    int count=0;
    char S[1024];
    gets(S);
    int i=0;
    while(1)
    {
        if(S[i]=='\0')
        {
            char a=S[i-1];
            if(a!=' ') count++;
            break;
        }
```

```
        if(S[i]==' '&&i>=1)
        {
            char a=S[i-1];
            if(a!=' ') count++;
        }
        i++;
    }
    cout<<count;
}
```

6. 幻方-Square.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
int const N = 7;
//修改 N 值可得不同大小的幻方
void Square(int a[N][N], int n)
{
    int i,j,k,iTemp,jTemp;
    i = 0; j = (n - 1) / 2;
    a[0][j] = 1;                        //在第 0 行的中间位置填 1
    for ( k = 2; k <= n*n; k++)
    {
        iTemp = i;
        jTemp = j;                      //暂存 i,j 的值
        i = (i - 1 + n) % n;            //即 i=i-1; if (i<0) i=n-1
        j = (j - 1 + n) % n;            //即 j=j-1; if (j<0) j=n-1
        if (a[i][j] > 0)
        {
            i = (iTemp + 1) % n;        //如果位置(i, j)已经有数，填入同一列下一行
            j = jTemp;
        }
        a[i][j] = k;
    }
}

void print_Square(int a[N][N],int n)
{
    int i = 0,j = 0;
    for ( i = 0; i < n; i++)
     {
        for ( j = 0; j < n; j++)
            cout<<setw(3)<<a[i][j]<<" ";
```

```
        cout<<endl;
    }
}

int main()
{
    int a[N][N];
    for( int i = 0; i < N; i++)
    {
        for( int j = 0; j < N; j++)
            a[i][j] = 0;
    }
    Square(a,N);
    print_Square(a,N);
    return 0;
}
```

四、运行与测试

1. 凯撒密码

```
输入原始字符串:
helloworld
输入移动量k:
5
加密后字符串为:
mjqqtbtwqi
--------------------------------
Process exited after 7.159 seconds with return value 0
请按任意键继续. . .
```

2. BF 算法

```
输入主串S
abababcdefg
输入子串T
abc
5

--------------------------------
Process exited after 11.97 seconds with return value 0
请按任意键继续. . .
```

3. KMP 算法

```
S:
helloworld
T:
ell
2


--------------------------------
Process exited after 7.969 seconds with return value 0
请按任意键继续. . .
```

4.求公共子串

```
input S:
helloworld
input T:
hld
1
--------------------------------
Process exited after 512.1 seconds with return value 0
请按任意键继续. . .
```

5.统计单词数

```
I am from China.
4
--------------------------------
Process exited after 22.67 seconds with return value 0
请按任意键继续. . .
```

6.幻方

```
 28  19  10   1  48  39  30
 29  27  18   9   7  47  38
 37  35  26  17   8   6  46
 45  36  34  25  16  14   5
  4  44  42  33  24  15  13
 12   3  43  41  32  23  21
 20  11   2  49  40  31  22

--------------------------------
Process exited after 0.2034 seconds with return value 0
请按任意键继续. . .
```

五、总结与心得

这几个验证和设计实验，难度最大的应该是 KMP 算法。KMP 算法首先难在它的原理不好

理解，为了高效匹配，让子串不回溯，而是直接到达下一个它应该到的地方。这就需要研究子串本身的对称性质，因此我们引入 next 数组作为衡量标准。由于 next 数组只与子串有关，与主串无关。在求这个数组的时候，我们多次使用了一个思想“利用前面已知的回文性质，求解新的回文性质”。

通过这几个实验的设计和运行，我更加深入了解了字符串和模式匹配相关的算法，尤其是在它们的实际运用方面有了更深刻的理解。

第三部分 综合实验 1：近似串匹配

一、问题描述

在一个文本中查找某个给定的单词，由于单词本身可能有文法上的变化，加上书写和印刷方面的错误，实际应用中往往需要进行近似匹配。这种近似串匹配与串匹配不同，实际问题中确定两个字符串是否近似匹配不是一个简单的问题，例如，可以说 pattern 与 patern 是近似的，但 pattern 与 patient 就不是近似的，这存在一个差别大小的问题。

设样本 P，文本 T，对于一个非负整数 K，样本 P 在文本 T 中的 K-近似匹配是指 P 在 T 中包含至多 K 个差别的匹配（一般情况下，假设样本是正确的）。这里的差别是指下列三种情况之一：

(1) 修改：P 与 T 中对应字符不同。

(2) 删去：T 中含有一个未出现在 P 中的字符。

(3) 插入：T 中不含有出现在 P 中的一个字符。

二、基本要求

- 设计算法判断样本 P 是否是文本 T 的 K-近似匹配；
- 设计程序实现算法；
- 设计有代表性的测试数据；
- 分析算法的时间复杂度。

三、算法设计-ASM.cpp

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int D[50][50];

int Min(int a,int b,int c)
{
    int min = a;
    if (b < a) min = b;
    if (c < min) min = c;
    return min;
}

int ASM(char P[], char T[], int m, int n, int k)
{
    for (int j = 0; j <= n; j++)
        D[0][j] = 0;
    for (int i = 0; i <= m; i++)
        D[i][0] = i;
    for (int j = 1; j <= n; j++)
    {
        for (int i = 1; i <= m; i++)
        {
            if (P[i-1] == T[j-1])
                D[i][j] = Min(D[i-1][j-1], D[i-1][j]+1, D[i][j-1]+1);
            else
                D[i][j] = Min(D[i-1][j-1]+1, D[i-1][j]+1, D[i][j-1]+1);
        }
    }
    int min=D[m][1];
    for(int j=1;j<=n;j++)
        if(min>D[m][j]) min=D[m][j];
    return min;
}

int length(char S[])
{
    int l=0;
    while(S[l]!='\0') {l++;}
    l++;
    return l;
}
```

```
int main()
{
    char P[100]="happy",T[100]="Have a hsppy day.";
    int k;
    cout<<"样本 P:  "<<P<<endl;
    cout<<"样本 T:  "<<T<<endl;
    cout<<"最小近似匹配为:  ";
    cout<<ASM(P,T,length(P)-1,length(T)-1,k);
    return 0;
}
```

四、运行测试

测试结果如图。

```
样本P:  happy
样本T:  Have a hsppy day.
最小近似匹配为:  1
--------------------------------

Process exited after 0.1945 seconds with return value 0
请按任意键继续. . .
```

五、总结与心得

近似匹配问题是一个动态规划问题，而动态规划问题的核心在于写出状态转移方程。

对于求解目标问题，当我们需要用到前面已知的求解结果时，可以考虑采取动态规划。每一步的最优解，都是在前面某一步的最优解的基础上，经过变化，继承过来的。算法中另一个重要的问题是如何把“修改、删去、插入”三个操作，在代价转移方程中体现出来。在算法描述中， 4 中可能的情况，第一种是完全匹配，第二种是修改，第三种是删除，第四种是插入。这个程序的算法复杂度为 O(n × m)。

第四部分 综合实验 2：数字旋转方阵

一、问题描述

编写程序输出一个逆时针旋转的数字矩阵

二、基本要求

- 设计数据结构存储数字旋转方阵；
- 设计算法完成任意 N(1≤N≤10)阶数字旋转方阵；
- 分析算法的时间复杂度。

三、算法设计

本题采用二维数组储存矩阵，同时采用递归方法进行求解。

用二维数组 dataIN]N]表示 N×N 的方阵，观察方阵中数字的规律，可以从外层向里层填数。在填数过程中，每一层的起始位置很重要。

设变量 siz 表示方阵的大小，则初始时 size=N，填完一层则 size=size-2，每次填数字分为四步骤，设变量 istart,jstart 表示每步的起始位置：

1.竖着向下填写 size 个。行为 istart 逐一累加，列为 jstart。

2.横着向右填写 size-1 个。行为 istart，列为 jstart 逐一累加。

3.竖着向上填写 size-1 个。行为 istart 逐一减少，列为 jstart。

4.横着向左填写 size-2 个。行为 istart，列为 jstart 逐一减少。

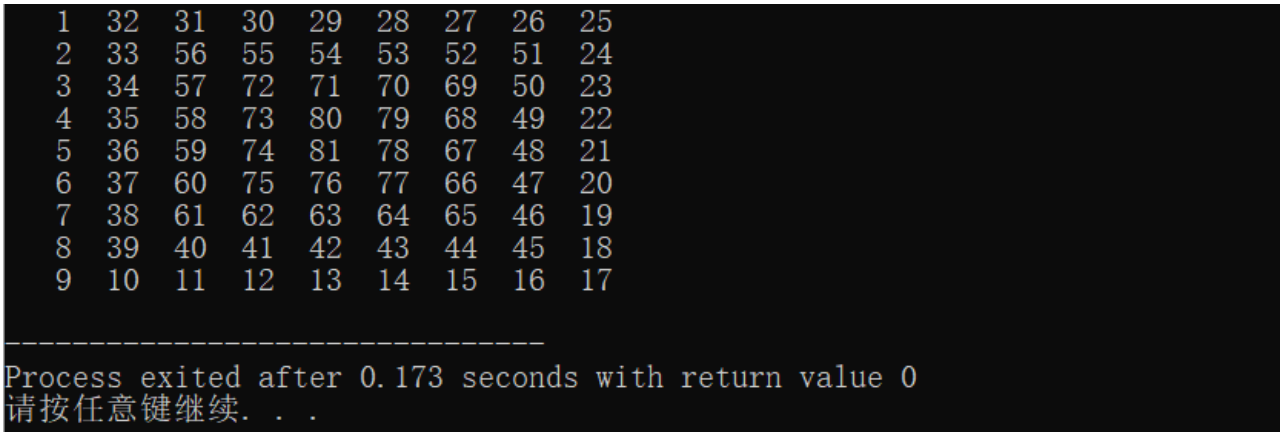
四、代码设计

```
#include <iostream>
#include <iomanip>
using namespace std;
int a[1000][1000];

void pin(int N, int n, int num)
{
```

```
    int istart = (N-n)/2+1;
    int jstart = (N-n)/2+1;
    for (int i = 0; i < n; i++)
    {
        num++;
        a[istart+i][jstart] = num;
    }
    istart = istart + n - 1;
    for (int j = 1; j < n; j++)
    {
        num++;
        a[istart][jstart+j] = num;
    }
    jstart = istart;
    for (int i = 1; i < n; i++)
    {
        num++;
        a[istart-i][jstart] = num;
    }
    istart = (N-n)/2+1;
    for (int j = 1; j < n-1; j++)
    {
        num++;
        a[istart][jstart-j] = num;
    }
    if (n-2 > 0) pin(N,n-2,num);
}

int main()
{
    int N=9;
    pin(N,N,0);
    for(int i=1;i<=N;i++)
    {
        for(int j=1;j<=N;j++)
            cout<<setw(4)<<a[i][j];
        cout<<endl;
    }
    return 0;
}
```

五、运行测试

测试结果如图。



六、总结与心得

数字旋转方阵是矩阵问题，自然而然可以想到用二维数组进行存储。算法的难点在于计算数目填写的起始位置 istart 和 jstart，因此使用了循环结构的连结和嵌套，算法的复杂度为 O(N×N)。