前言：

我的上机作业都是在自己的电脑上做，当时我也在学习linux系统，所以所有作业都在linux上完成。作业使用vim编辑器+gcc编译，所以操作起来没有vs上那么方便，代码调试方法也较为原始。当时系统没有装中文输入法，代码注释我用的是自己的蹩脚英语，望助教见谅。

## （一）线性表的应用：稀疏一元多项式运算器

1. 问题描述：利用链表的相关知识制造一个多项式运算器，并支持多种操作与运算
2. 算法描述：

数据结构：

在代码中多项式以链表形式存储，最主要的数据结构如下

typedef struct LNode{

double coef;

Int expn;

Struct LNode \*next;

}LNode,\*Linklist;

其中coef为系数，expn为指数。

在程序中可以存储10个多项式，在代码中使用头指针数组存储，即为Linklist arr[10]。（实际上使用了arr[11]，第11项作为临时存储，方便多项式在函数间传递，因为当时忘记有全局变量这个方法了）。

程序结构：

其中polynomial.c为main函数，对多项式的操作写在polyn\_func.c中。

double poww(double x，int n)：与math.h中的pow()作用相似，只是因为gcc编译math.h不通过，g++编译又会有指针方面的兼容问题，所以写了这个代替pow()。

Int order(Linklist Q)：该函数寻找并合并多项式Q的同类项，之后检查是否有系数为0的项，若有则删除该项。最后对多项式排序，使其保持降幂排列。在加减乘除等操作后都会使用该函数使多项式有序存储。

int RandCreatPolyn(Linklist \*p,int m)

与int CreatPolyn(Linklist \*p,int m)：前者生成共有m项的，系数为随机小数，指数为随机整数(1~20)的多项式，后者需要手动输入系数与指数。main函数如接受到生成的项数m为负值，自动以参数|m|调用前者，否则调用后者。

int AddPolyn(Linklist \*m,Linklist \*n)：传递多项式头指针的地址或者说其可变引用，

将第二个多项式简单地连接在第一个后面，然后调用order()合并同类项并重新排序。

int SubtractPolyn(Linklist \*A,Linklist \*B)：与上一函数相同，只不过连接时把第二个多项式的系数取负值。

int MultPolyn(Linklist \*P,Linklist \*Q)：多项式之间相乘，首先创建一个零多项式Z，然后把P的第i项与Q相乘，然后把结果累加至Z中，最后把P的头指针指向Z。

int DivPolyn(Linklist \*P,Linklist \*Q)：解释起来较为复杂，但是操作不难，每次把Q乘以Cix^Ei使得P与Q最高次相同然后调用减法函数计算二者差值。∑Cix^Ei即为商多项式，最后当剩余多项式最高次比Q小的时候，剩余项即为余数多项式。

int Least\_Greatest(Linklist \*P,Linklist \*Q)：与整数时相同，只不过是利用DivPolyn()函数来进行辗转相除。

其余函数的功能与函数名称一致，不一一赘述。

1. 调试分析

每次写好某函数后，我会在函数中添加printf()来判断函数的运行状态以及出错代码位置，程序中遇到的问题，我已记不太清楚了，主要问题出在运行时指针越界报错与循环无法终止上，主要查错方法为在函数的各个位置添加printf(“%d\n”,i++)；语句，运行程序后观察哪处输出异常，根据最后一次输出的数字来找到异常代码。

在编译时出现的其中一个问题是函数参数不统一导致的。（原因：写代码时忘记使用全局变量，所以我写的函数大多都需要一些参数）由于有些函数只需要不可变引用，所以只传递多项式的头指针即可，而另一些函数则修改了多项式本身，需要传递头指针的指针来应对需要修改头指针指向的情形。这种情况下我的函数参数定义时有的使用\* Linklisk，有的使用Linklist，这使我自己在写函数时产生了混淆。最后解决方法就是根据编译时的报错信息，找到对应的函数，重新写代码，并且在之后的函数中，尽量统一使用\* Linklist。

1. 算法的时空分析

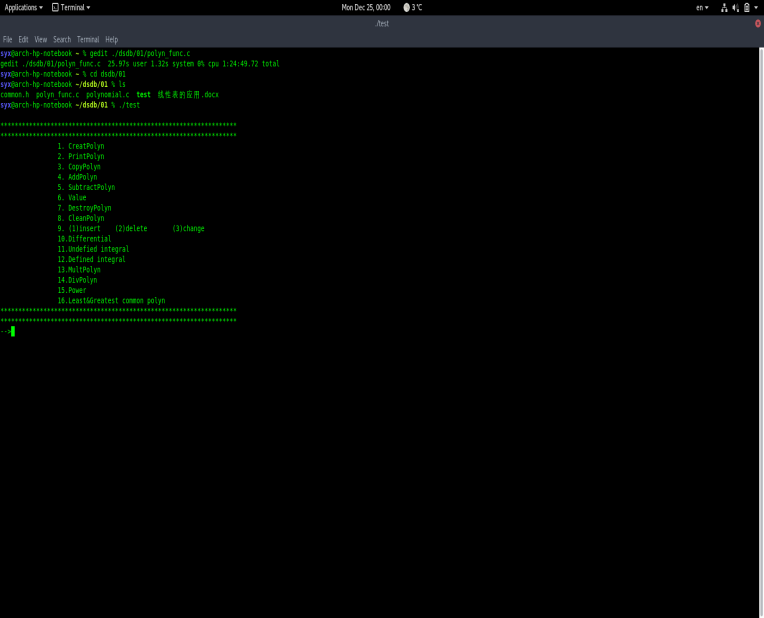
程序中的函数主要有两种，一种是单遍操作，所以是O(n)的，这些函数有poww(),ListArr(),RandCreatPolyn(),CreatPolyn(),PrintPolyn(),CopyPolyn(),Value(),DestoryPolyn(),CleanPolyn(),Insert()(这里的n为函数参数n),Change(),DiffPolyn(),UDFIntegral(),DEFIntegral()。

另一种为需要调用order()函数的函数，order()函数本身由O(n)的合并操作与O(n²)的冒泡排序构成，所以是O(n²)的。 而加法与减法调用了它，易知是O((m+n)²)的，乘法与除法由于反复调用了加法与减法，所以是O((m+n)³)的。最后的求最大公因式，辗转相除的次数是O(logn)的，每次除法调用了一次加减函数，所以总共是O(n²logn)的。

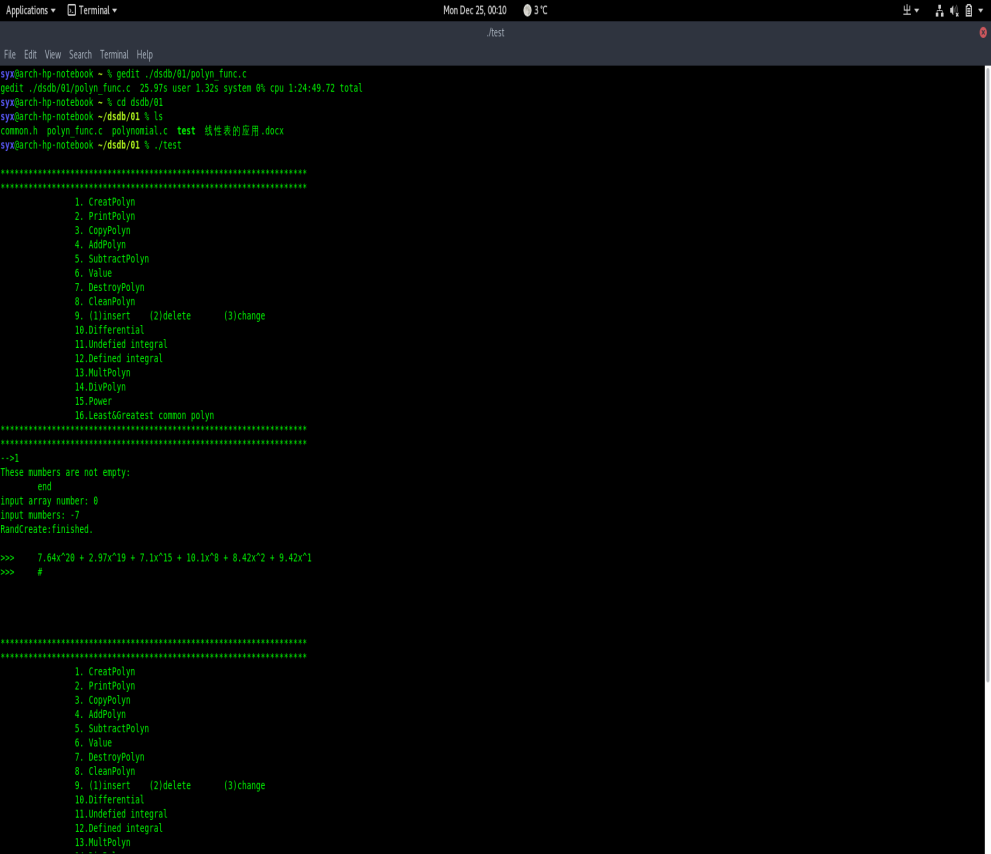
这种使用order()函数来简化加减乘除函数的写法只是在n较小时的方便之举，当n较大时应该重新写加法与减法，可以达到O(m+n)的时间，这样乘法与除法即为O((m+n)²)的了，或者修改order()，使用其他排序法来使其达到O(nlogn)的复杂度。

1. 测试结果及分析

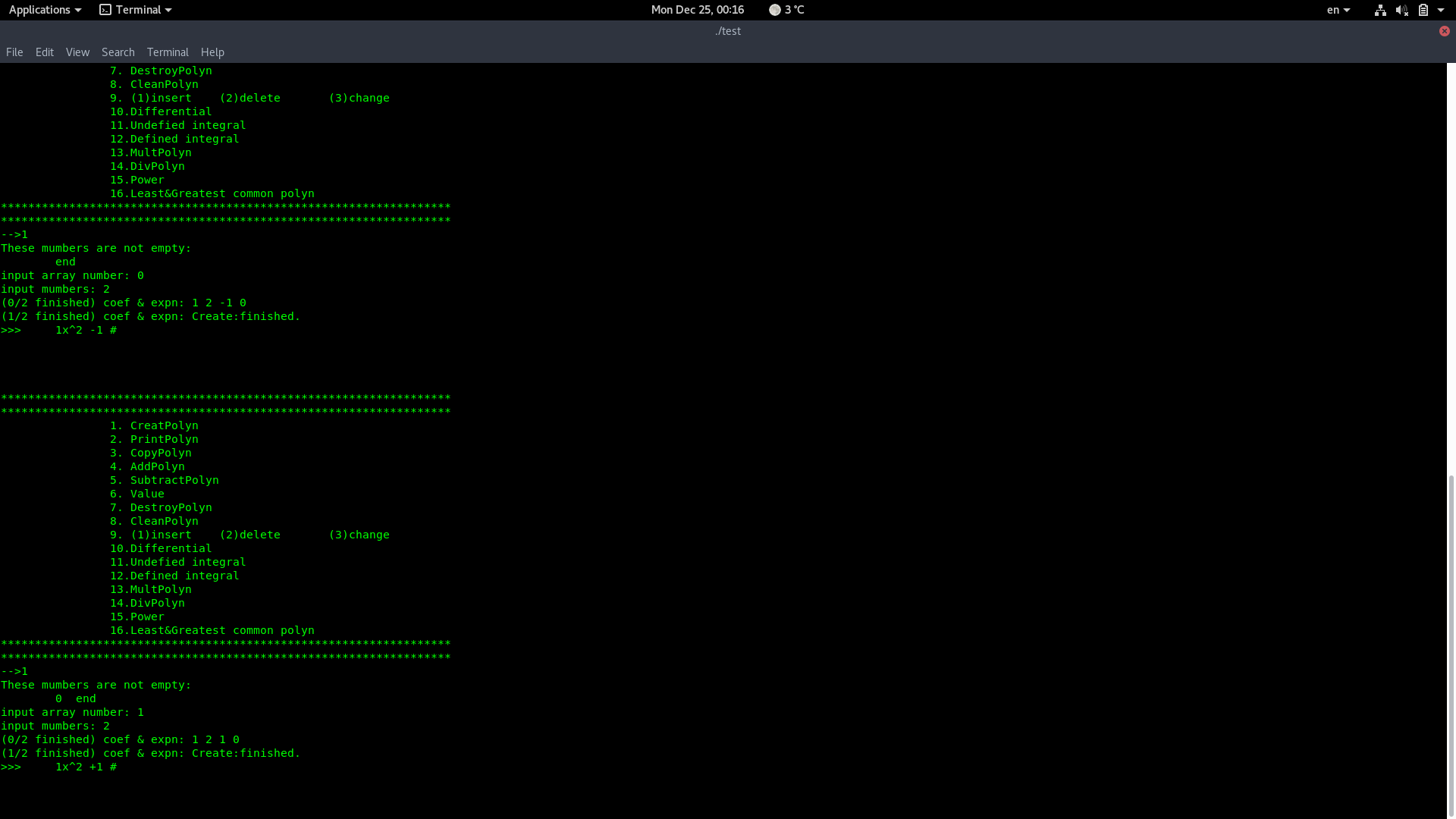
初始界面：

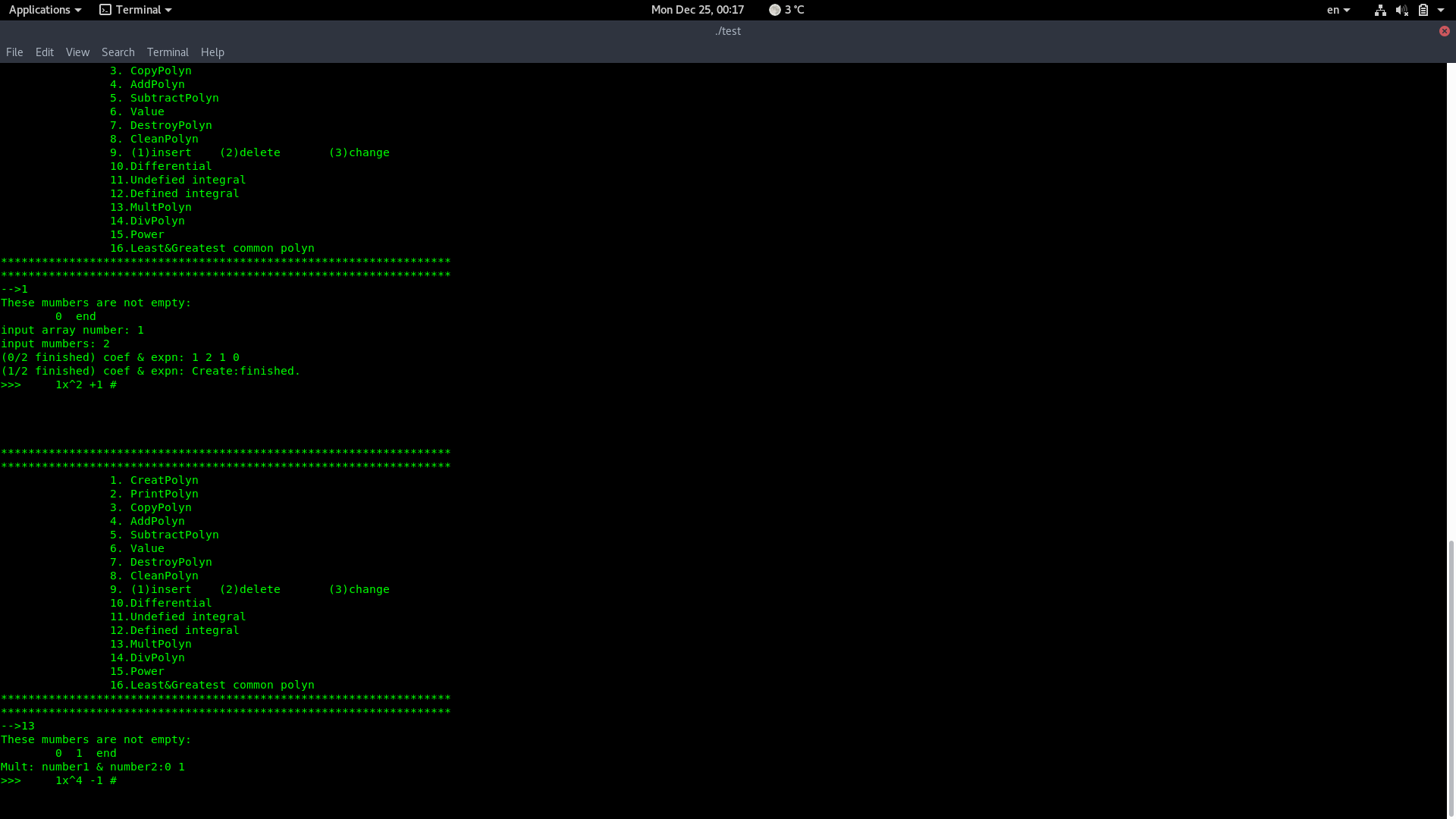


随机生成多项式：



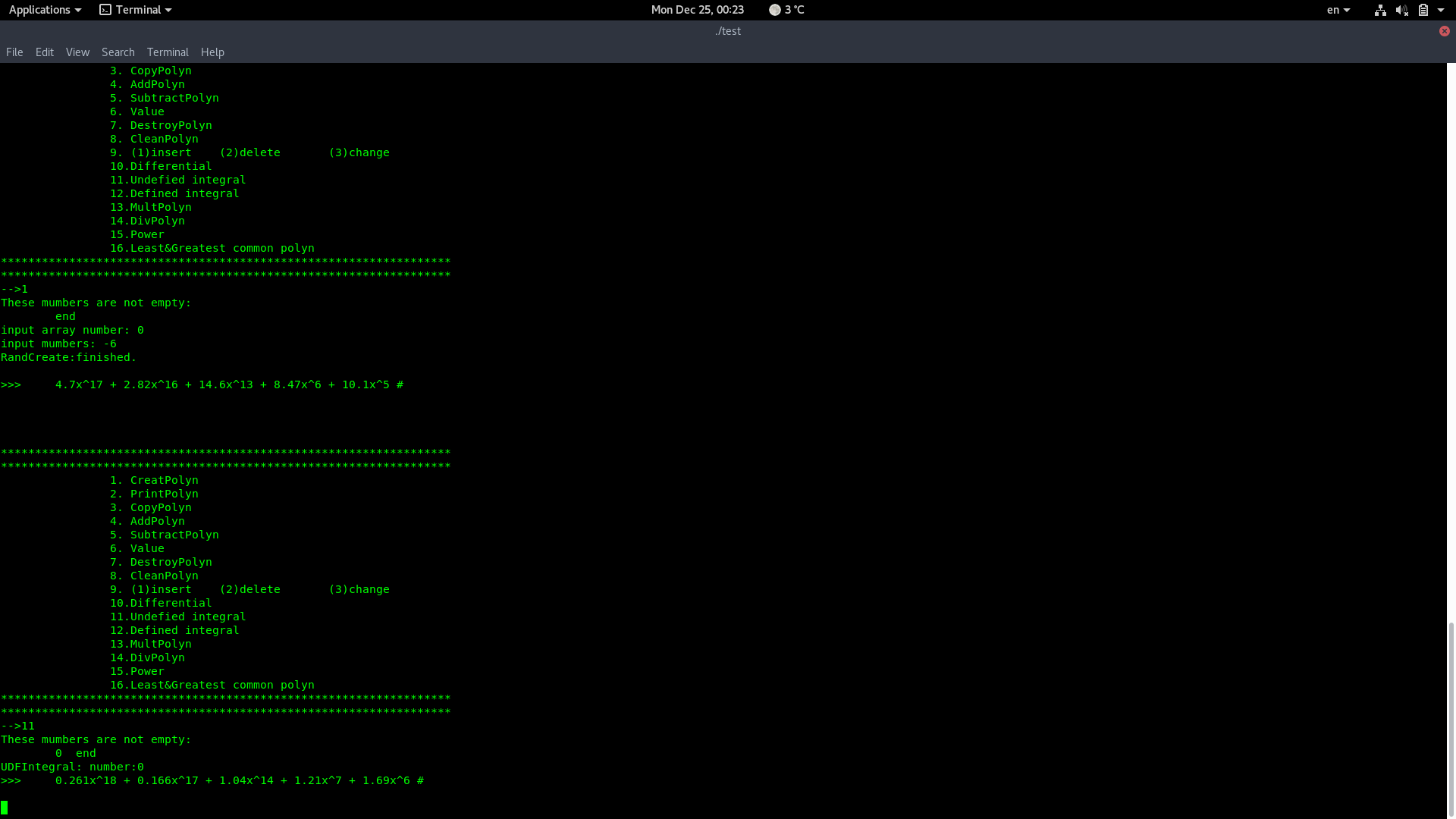
乘法演示：





演示了(x²+1)\*(x²-1)=x^4-1.

不定积分演示：



函数过多，不一一演示。

1. 实验体会

在写一个较大的程序前，应该首先构思好程序的基本骨架与结构，还有各个函数需要实现的功能，寻找重复代码并提炼为函数。正所谓磨刀不误砍柴工，做好准备工作看似麻烦，实际上减少了之后的无用功，精简代码，也更不容易犯错误。这次程序作业涉及的功能较多，我在写的时候没有事先规划，导致经常因为后面的内容修改前面的代码，这是不好的习惯，应该改正。

## 栈的应用

1. 问题描述

利用栈的相关知识，制作中缀式计算器，并实现对浮点数的操作。

1. 算法描述

数据结构：该程序总共使用了两个栈，一个存储char类型的运算符，另一个存储double类型的数值。对于前者，在stack\_func.c中定义了initstack(),push(),pop()等函数，与书中例题保持一致，对于后者，则把进栈出栈简化为arr[top++]=x与x=arr[--top].

程序结构：

以下为Main函数中用到的函数：

Int isOP(char s):判断字符s是否为运算符，如果是则返回1，不是则返回0.

Char precede(char a, char b):判断两个运算符的优先级并相应的返回‘<’,‘>’与‘=’.

Int trans(char s): 返回整型的(‘s’-0)，即把一个字符型的数字转化为整型数字。

Double operate(double a, char theta, double b):对二元运算符与其两个数值进行求值。对于sin(x)这类一元运算，为了保持形式统一，仍然使用该函数，只对a与theta进行操作，b返还至栈中。

1. 调试分析

在最初的版本中，读取浮点数只读取满足isdigital()的字符，所以只支持正数间的运算，当存在负数时，程序将执行错误操作，无法输出正确结果。之后增加了支持负数的代码，但在区分加减号与正负号时又出现了bug。最后解决方法：当读取到‘+‘，’-’时，读取前一个字符，如果为‘(’或其他运算符，则认为是正负号，如果为数字或者‘)’，则认为是加减号。

在最终程序中，如果规范地输入表达式，比如5\*(-2)-(+5),则肯定不会出错，但是如果输入5\*-2-+5这种不正确运用括号的式子，则可能出错(但对于这个例子仍然能够正确输出结果）。错误的条件为“存在至少3个连续的不是括号的运算符，且第二个为‘+’或‘-’号”，程序会把第二个判断为正负号，之后把后面的字符当作数字读入，从而引发错误。

在上一个错误修复之后，有一个小问题就是当表达式以‘+’或‘-’开头时，用上面的方法程序无法判断其为正负号或是运算符(不存在前一个字符)。解决的办法为，事先在存放数值的栈中加入一个0，这样修改少量代码后可以把表达式+S，-S等效为0+S或0-S，从而避免这个问题。

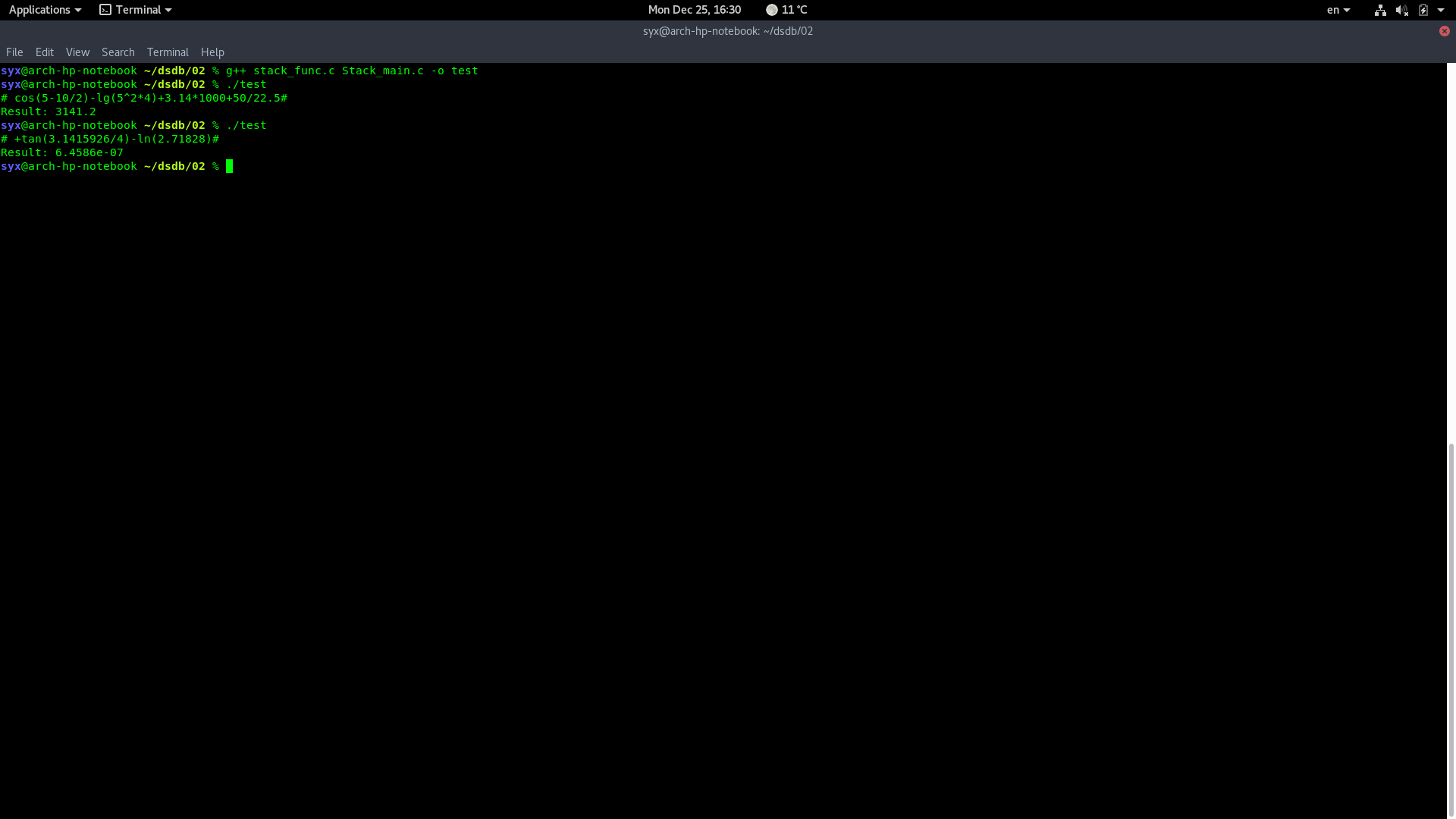
1. 算法的时空分析

Main()中的最大循环每执行一次，由getchar()读入一个字符并进行相应的操作。每个循环都对字符进行一次判断区分操作符与数字，之后进行相应的操作。相应的操作都是确定的，为O(1)的时间，所以总共的时间为O(n)的。n为输入的总字符数，浮点数中的每一位算一个字符。

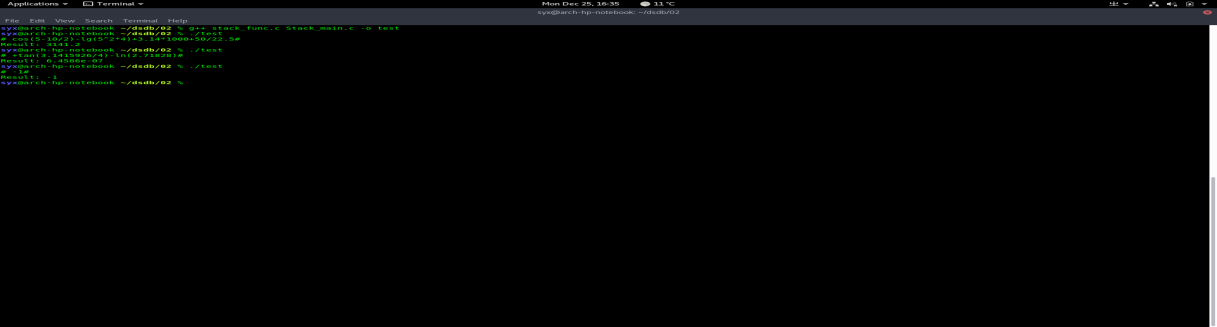
空间复杂度即为两个栈的大小再加上O(1)的程序中定义的变量，所以为O(M+N),M,N为在程序中设定的两个栈的大小。

1. 测试结果及分析

两次输入测试：

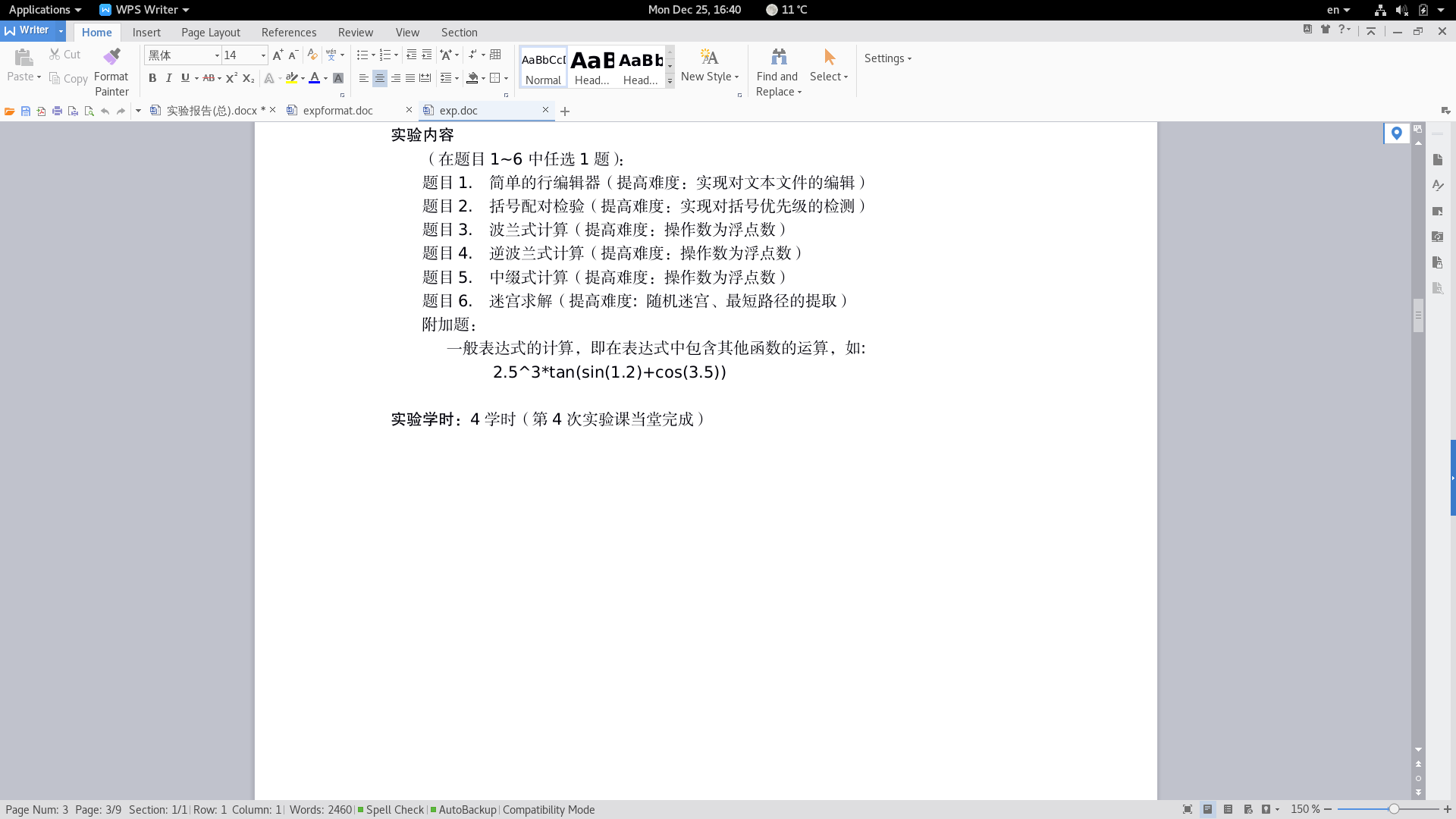


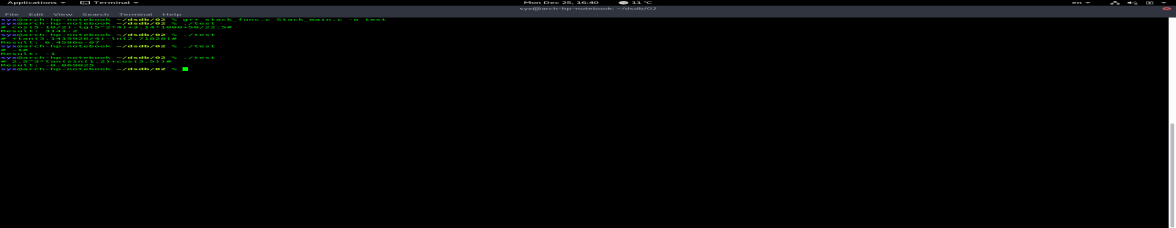
第一个展示了支持浮点数的加减乘除乘方与特殊函数的运算，第二个说明我们的运算是可以精确到小数点后多位的，并且输出格式合理，开头的正号也能正确识别了。



第三个则是对特殊情况下程序健壮性的检测↑.

最后一个是实验要求上的举例：





1. 实验体会和收获

考虑问题要考虑全面，把所有可能的情况都考虑进去，不然程序就容易出现bug。

## 二叉树的运算

1. 问题描述

利用二叉树对电文串进行Huffman编码，并且利用Huffman算法，编写一个文本压缩程序。

1. 算法描述

数据结构：HTNode为带有双亲指针的二叉树节点结构体，采用线性存储，使用int类型存储双亲与孩子节点的数组下标。HC为char\*\*类型，作为存放huffman编码的字符串数组名。已经完成huffman编码的字符，存入哈希表Hashlist中，这样能大大减少压缩文件所用的时间。

程序结构：首先是使用stat()函数统计文本信息，存入表D中，然后使用Huffmancoding()函数(与书中一致)对D中信息进行编码，保存在D中的Huffmancode HC中。

Void move(Dict \*D, Hashlist H, Huffmantree HT)：把D中的字符与对应编码放入HT中，然后再存入哈希表H中。

Int hash(char c){ return (int)c&255; }是上面用到的哈希函数，即把c在二进制下的值作为哈希表的地址，这样得到某字符后通过随机读取就能立即知道其huffman编码。

Output()：函数参数较多，就不写出来了。这个函数接收之前的所有信息，并以次把目标文件的每个字符转换为对应的huffman编码，达到压缩的目的。文件以‘rb’方式读取。

Char trans(char \*arr)：接上文，读取目标文件时每次读取一字节，并把对应字节的huffman编码转换为整型的‘0’与‘1’存入一个（int arr[8]）数组，每当数组八项存满后，该函数把这八项按二进制转换为1字节，这个字节中每一位的值都与数组中对应项相同。

压缩完成后，所得压缩文件的头部为压缩时使用的huffmantree，中部为huffman码，尾部有三个整型数，分别记录了编码次数，编码长度模8的余数，还有文中的字符种类。

Int decompress(char \*p)：接收文件名，并对文件进行解压。先读取文件尾部三个数据，然后读取头部huffman树，之后根据这些信息解压缩文件中部的编码。

1. 调试分析

测试时最好使用文件大小稍大一点的文件进行压缩，否则可能压缩文件节省的体积还没有在文件中写入的huffman树的体积大，这样也就看不出压缩的效果了。

调试中出现的问题主要出现在压缩文件的output()函数中，具体错误也记不清了，检查方法一是在函数中设置多个printf(“%d\n”，i++)函数，确定出问题的地方；二是用printf函数输出函数执行中的某些变量的值，与理论上的结果比较，从而确定自己设计的流程有没有出现问题。

曾经出现的一个问题是我设定的储存huffmancode的数组大小不够，数组不能存储完整的编码。这样有些数组结尾没有‘\0’标志，这使得某些过程不能正确停止，所以压缩后的文件大小比预想的要大很多，我因此发现了这个问题，然后改正。

1. 算法的时空分析

时间：

首先是统计词频，时间显然是O(n)的。

之后是构造huffman树，这需要k-1次循环，第i次循环需要在k+1-i个数中找最小的两个，所以是O(k²)的。对于n来说，他是O(1)的。

构建哈希表，由于字符的总数不超过256种，所以构建哈希表的循环次数不会超过256次，每次循环clone长度为设定值的str放入哈希表。这需要的总操作数的基数较大，但有上界，所以是O(1)的。

压缩，共n次操作，每次读取一个字符，在哈希表中找到他的编码，把编码写入临时数组，每当数组存满时写入文件，操作数都有上界，所以每次都是O(1)的。（实际上编码的最大长度正比于ln(k)，k为字符种类，但是k<=256，且对于文本文件来说，k与n关系不明确，在该情形下可当作常数，上文的构造操作同理）。综上，压缩操作是O(n)的。

解压缩与压缩相同，所以也是O(n)的。

综上，总的时间复杂度是O(n)的。

空间：

空间复杂度与时间刚好相对，主要与k有关，即字符的种类越多，所占用的空间越多。

在本程序中，因为已经知道了空间不可能超过256，所以数组长度都为定值256，存储huffmancode的数组也是设置的最大长度，所以在本程序中是O(1)的。

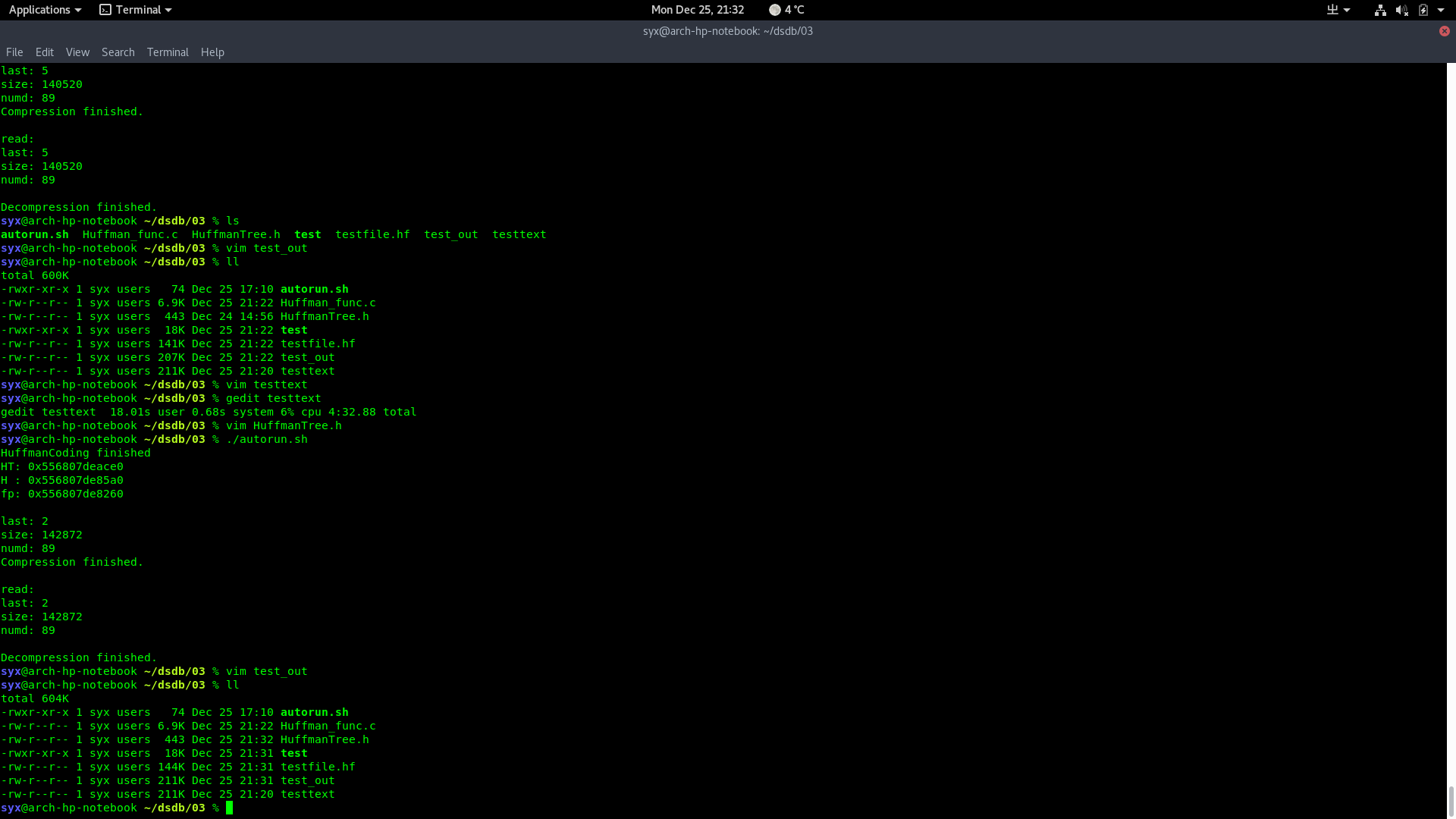
1. 测试结果及分析

压缩、解压后的文件与原文件相比，有时会差一个空格字符，但这个空格是怎么产生的，目前不太清楚。由于是对256种单字节进行压缩，所以对于非文本文件中每个字节完全随机的情况，压缩效果不理想，我对10M左右的mp3文件进行压缩，体积基本不变。但对于文本文件，很多时候字符种类不超过100种，这时压缩效果非常好。

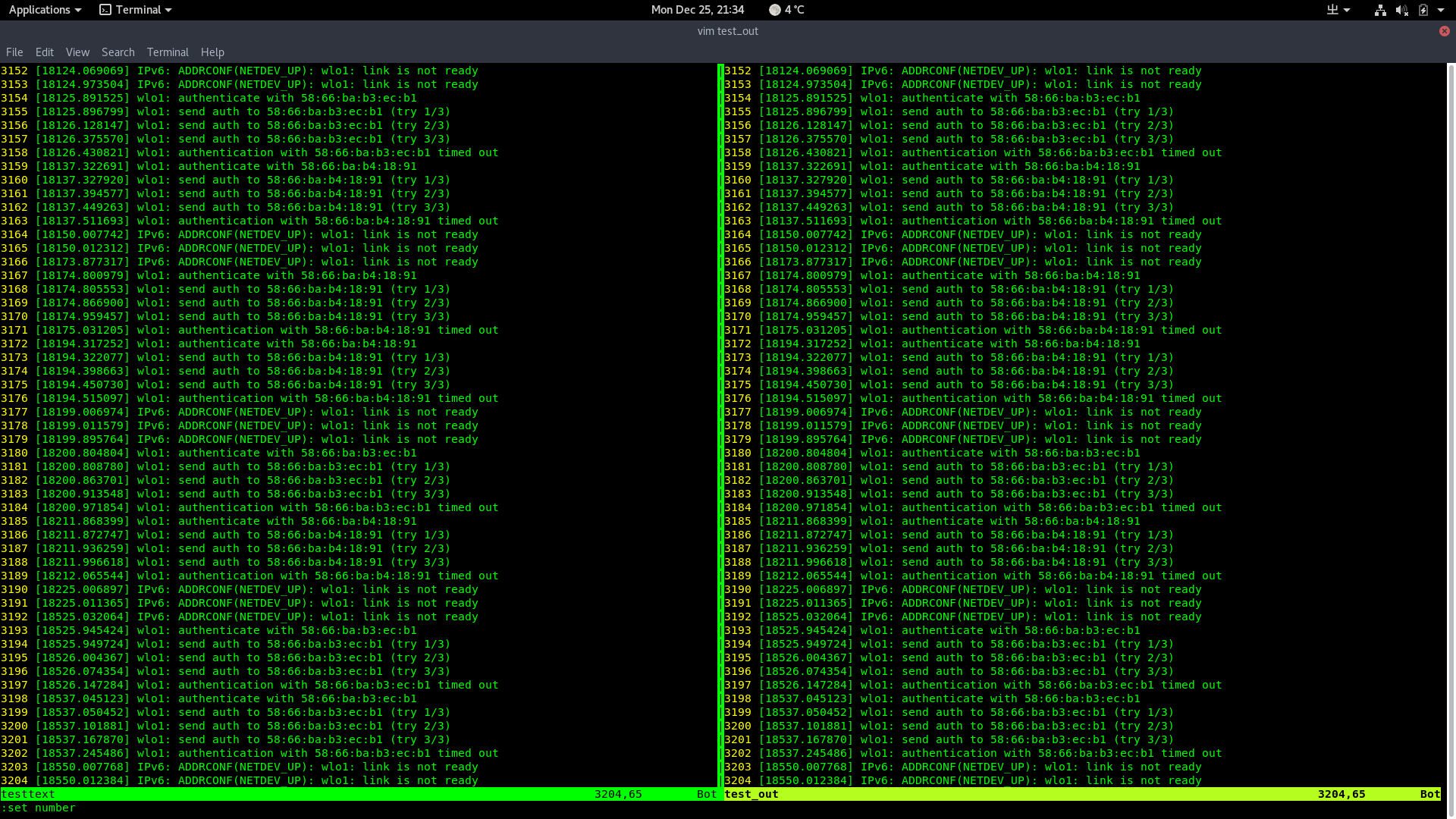
下图中testtext是随便选的文本，利用的是dmesg >> ./testtext，

test\_out是解压后的文件，

testfile.hf为压缩文件。



下图为两文件对比，二者完全相同。



6. 实验体会和收获

这个实验的收获很多，却不知如何向之前那样表达。心得很简单：写这种程序，每个细节都一定要搞清楚，不然找bug会累死人的......

## 图

1. 问题描述

输入铁路交通网G，利用Dijkstra算法求始发站与终点站间的最短路径。

1. 算法描述

数据结构：利用邻接矩阵存放铁路交通网G,利用二维数组记录两点之间的路径以及地点名称与其在程序内部的编号的对应关系。

程序结构：主要结构就是在main函数中获得地图，创建变量，调用ShortestPath\_DIJ()函数得到最小路径并在main函数中调用display()输出。

ShortestPath\_DIJ(int v0):生成从v0出发的所有最短路径，保存在全局变量中供main()操作。

display(int P[]，char (\*N)[4])：以某种格式输出P中第j行表示的路径。

1. 调试分析

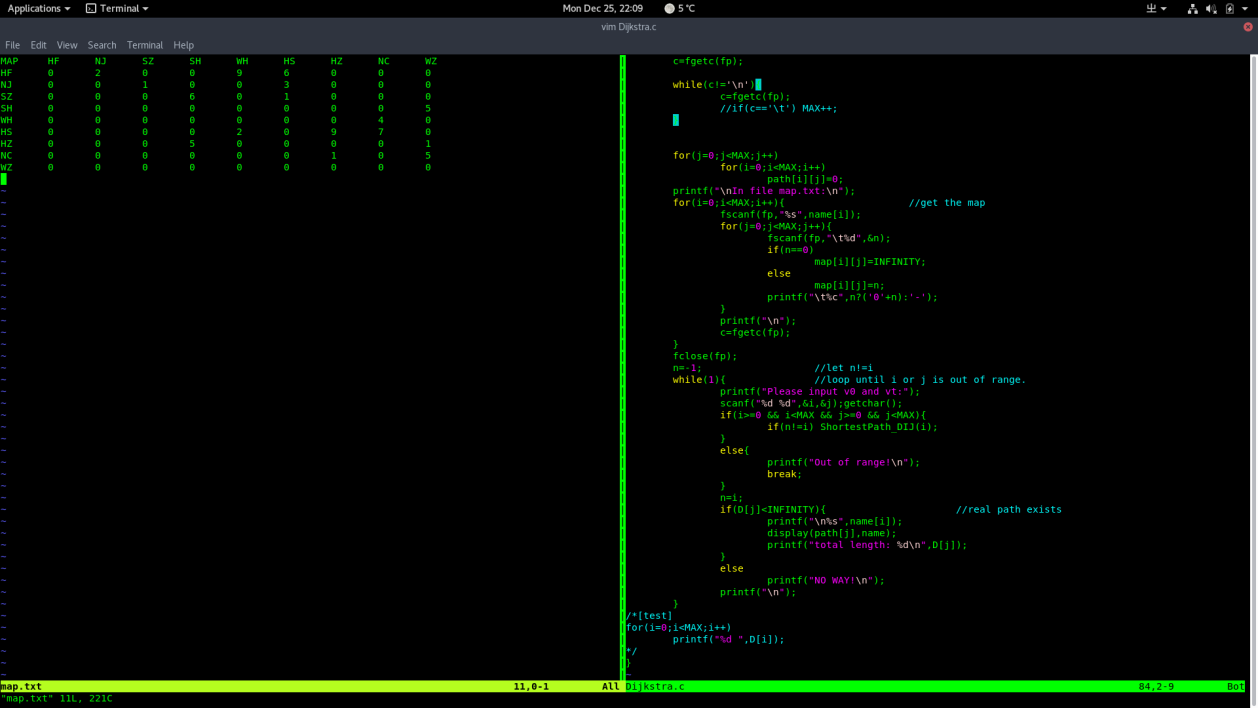
在Path[][]中，如果按书上的做法只用0或1记录是否经过某点，则无法区分经过点的先后。在编程时，我将其修改为用i++记录，这样在输出时，按其存储的数字由小到大输出，就能同时输出路径的顺序。程序较简单，没有太大的问题。

1. 算法的时空分析

与书上一致，本例Dijkstra算法的时间复杂度是O(n²)的，空间复杂度也是O(n²)的。

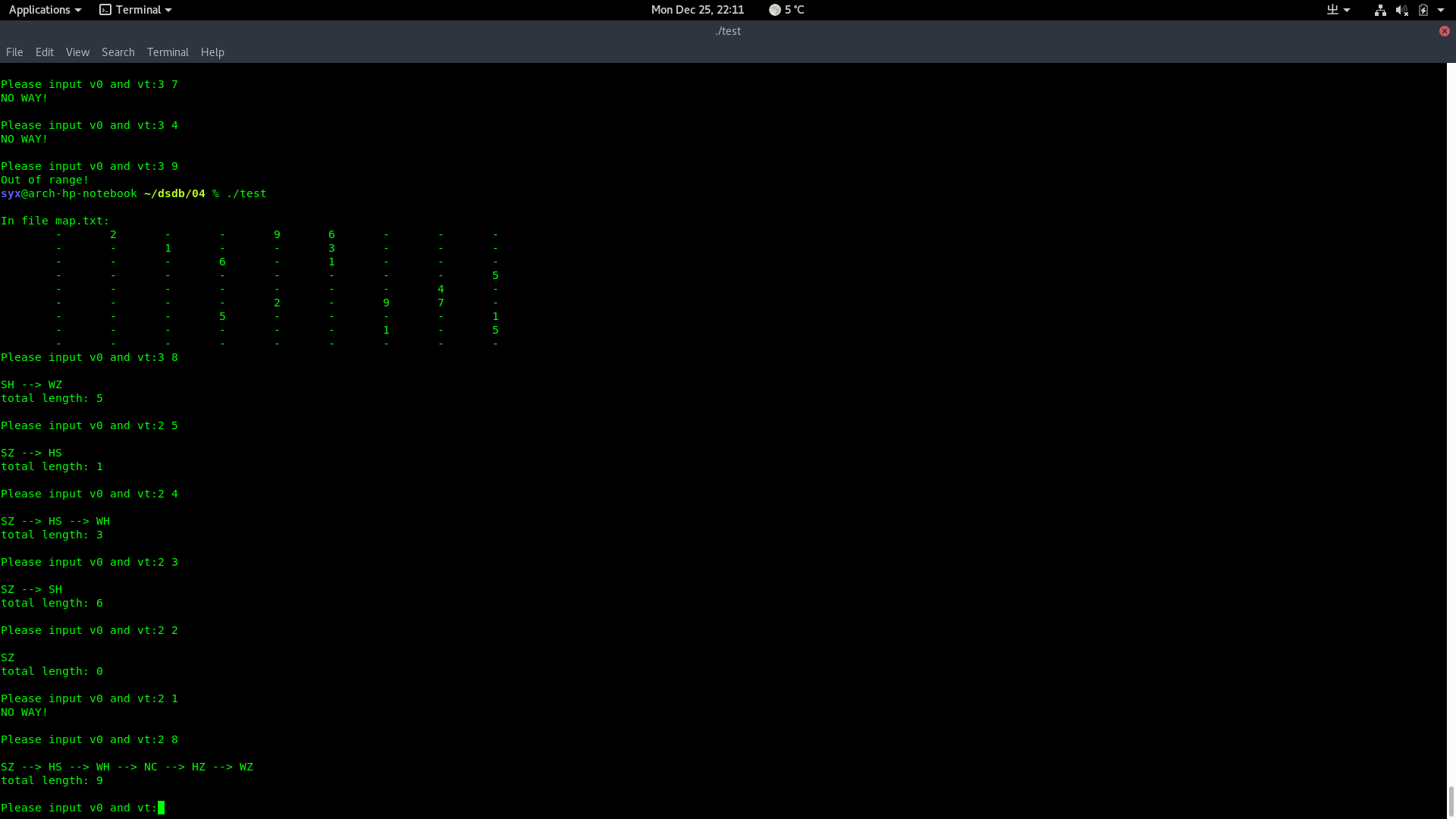
1. 测试结果及分析

下图为地图以及源码：



下图为测试结果：

黄色方框中测试了特殊情况下程序的结果



可以看到输出的效果还是很不错的。

1. 实验体会和收获

这次的实验相对简单一些，收获并没有写之前那种略复杂的程序时那么多，但是实践操作来实现一个算法，对于理解算法的工作原理来说是十分重要的。

## 哈希表

1. 问题描述

掌握哈希表的定义、特点，操作方法以及ASL的计算方法

1. 算法描述

数据结构：拉链法哈希表定义方式如下：

typedef struct LNode{

int data;

struct LNode \*next;

}Lnode,\*Linklist;

static Linklist HT2[13];

线性散列法的定义就是static int HT1[13]；

程序结构：

对于HT1与HT2，都与对应的下列函数：(‘%’表示可取1或2)

Int insert\_HT%(int a):接收一个整数，插入哈希表中。

Int make\_HT%(void):从键盘上输入的数组保存在某数组int D[13]中，该函数把D中的数存入哈希表中。记录总共的查找操作数，该操作数就是成功时的ASL\*n.

Int search\_HT%(int a)：查找给定的数字是否在表中。

FASL\_HT%(void)：对于表长为13的表，若存储的最大数字为X，那么对Y>X，查找Y~Y+12的每个数，正好遍历所有失败情况，这时的总查找操作数就是失败时的ASL\*13.这个函数就是这样来得到失败时的ASL的。

Print\_HT%(void)：输出对应哈希表。

Delete\_HT%(int a)：从哈希表中删除给定的数，并保持哈希表正确的结构。

1. 调试分析

在第一次给助教检查时，程序在删除操作上出现了错误的结果，这时因为我删除了某个节点后没有考虑一些复杂的情况，只是简单地判断被删除点后面的数字是否经过再散列，然后向前移动。实际上可能出现一些复杂的情况，这些情况下我的调整方法并不适用，不如重新创建哈希表。后来我找到了不用重新创建哈希表，就能将删除过某节点的表修改正确的方法。

1. 算法的时空分析

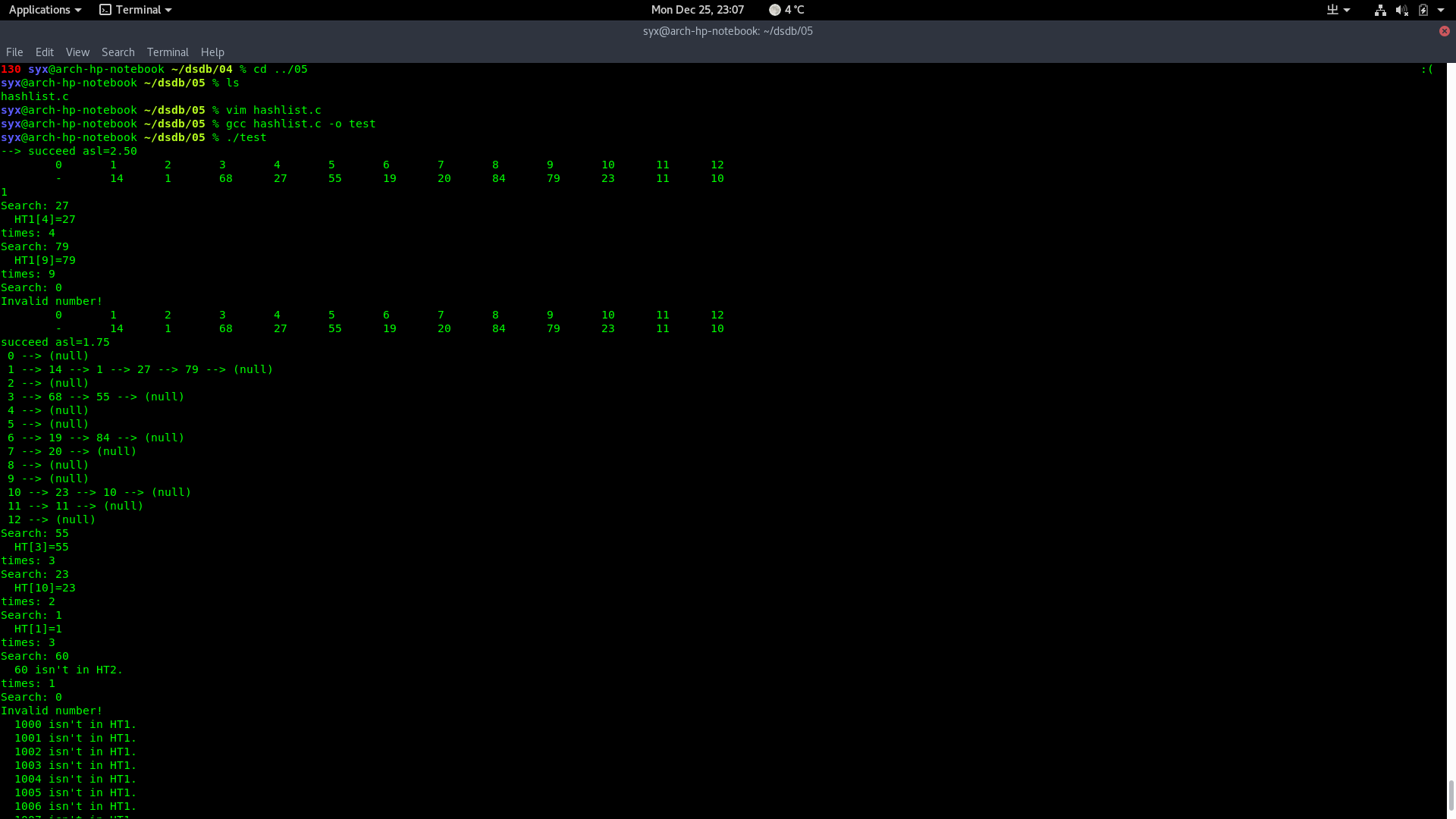
不同的解决冲突方法的时间复杂度公式在课本上已经给出，是与数据密度有关的函数，不再列出。

空间复杂度是O(1)的，因为没有使用多余的与数据大小有关的空间。

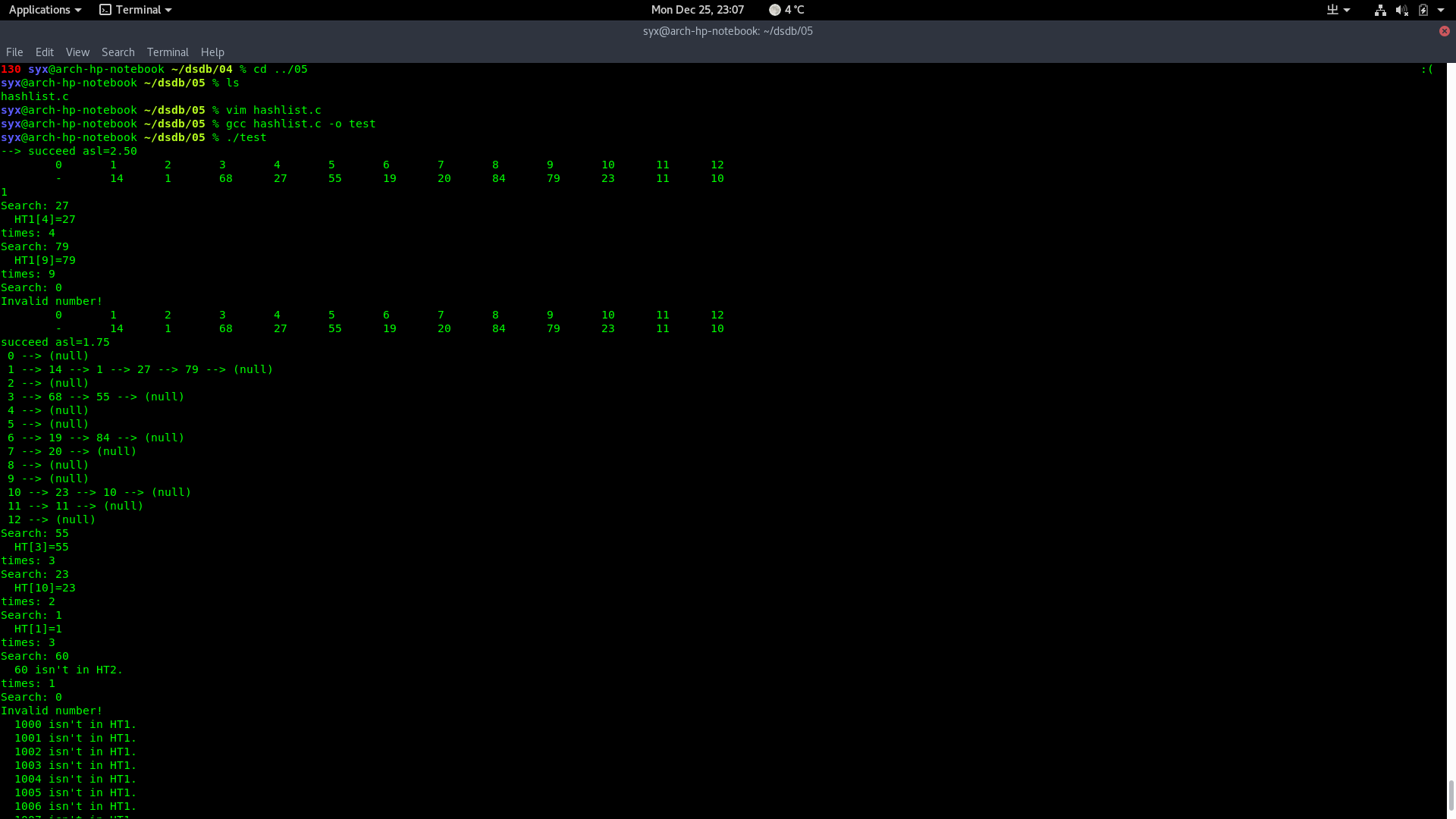
1. 测试结果及分析

(已略去手动输入步骤)

查找HT1中的数↓



查找HT2中的数↓



遍历错误数据来计算失败时的ASL↓



删除数据：



↓↓↓



可以看到删除27后，55与79也都向前移动了，说明这是正确的算法。

1. 实验体会和收获

最后一次的实验并没有之前那么复杂，但是有很多细节需要注意，比如没有考虑到删除节点时可能出现的问题，最后到验收的时候才发现（尴尬—\_—|||）。总的来说，还是要细心一点，既然是写代码就要把所有的特殊情况都考虑进去。