**Lab Report**

**on the Project of Data Compression**

16307130194 陈中钰

16级 计算机科学技术学院

**Contents**

1. General 2
2. Input 1: Pascal 3
3. Input 2: English 7
4. Input 3: Chinese 8
5. Input 4: Random Numbers 10
6. Input 5: Photo 10
7. Input 6: Gray Code 12
8. Input 7: π 15
9. Input 8: Ten Queens 16
10. Input 9: Random Numbers 17
11. Input 10: Permutation 19
12. Reflections 21
13. **General**
    1. 完成结果

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Input | Original Size | Program | | Auxiliary File | | Total Size | Compression Rate |
| Name | Size | Name | Size |
| Input 1 | 135,160B | 1.cpp | 355B | 1 | 4,784B | 5,139B | 3.54% |
| Input 2 | 264,360B | 2.cpp | 357B | 2 | 15,617B | 15,974B | 5.91% |
| Input 3 | 358,296B | 3.cpp | 357B | 3 | 30,210B | 30,567B | 8.43% |
| Input 4 | 320,000B | 4.cpp | 101B |  |  | 101B |  |
| Input 5 | 202,920B | 5.cpp | 116B | 5 | 25,365B | 25,481B | 12.5% |
| Input 6 | 49,152B | 6.cpp | 76B |  |  | 76B |  |
| Input 7 | 128,000B | 7.cpp | 224B |  |  | 224B |  |
| Input 8 | 79,640B | 8.cpp | 194B |  |  | 194B |  |
| Input 9 | 320,000B | 9.cpp | 109B |  |  | 109B |  |
| Input 10 | 1,179,646B | 10.cpp | 394B | 0 | 8,192B | 8,586B | 0.69% |

* 1. 编码方式

ASCII码使用指定的8位二进制数组合，来表示128种（标准ASCII码，仅低7位有效，最高位为0）或256种（扩展ASCII码，全8位均有效，用于中文编码）可能的字符。那么在没有任何压缩思绪的情况下，至少可以把每8位二进制数转换成1个字符，压缩率能达到12.5%。对于压缩后所需的附加文本，也可以采取这种方式进一步压缩。而且，如果编码转换后的文本内容是有意义的（如input1编码转换后为Pascal代码），那么还能便于寻找文本规律、寻找合适的压缩方法。

此外，还有把32位二进制数转换为int型数字、16位二进制数转换为short（int）型数字的整数编码方式，在某些情况下，把文本转换成一组数字，也可能便于寻找文本规律（如input10的第三部分转换成一组short数字后，为0~65535的排列）。

* 1. 寻找规律方法

对于ASCII编码或整数编码有意义的文本，可以对文本内容直接寻找规律（如input2编码转换后为英文短文）；而对于ASCII编码“乱码”、整数编码无明显意义的文本，则只能在二进制数的层次上寻找规律。通常，可以把二进制串分割成等长的短串（如10位二进制数一串或32位二进制数一串），通过观察不同短串之间的异同和关系，寻找一些特别的短串（如全为1的短串），来寻找文本整体的规律（如input8为十皇后的棋盘描述）。

* 1. 分析文本大小

对于二进制文本，首先可以分析文本大小，对文本大小进行质因数分解，获得文本大小的因数，进而可以猜测文本可以划分为多少位的短串。据此可以判断要用ASCII编码转换（如input3的大小是8的倍数，可以用ASCII编码转换），还是用整数编码转换（如input4的大小是32的倍数，可以用int型整数编码）。在二进制数层次上分析规律时，还能判断应该分割的位数（如input8是10的倍数，可以分割为每个短串10位）。进行上述编码转换或分割后，也许能便于寻找规律。

* 1. 代码压缩

由于评分准则主要看重的是代码长度，因此压缩文本后还需要压缩代码。而压缩代码主要有以下几种方法：

* 1. 牺牲时间/空间复杂度。由于评分准则主要看重的是代码长度，而对于解压的时间效率、空间限制要求低，那么在某些情况下，可以修改算法，适当牺牲时间/空间复杂度，减少解压代码量。
  2. 函数。如：bitset函数。
  3. 运算符优先级。如：+和|运算其实是一样的，但是运算级别不同，用上适当的符号可以省略掉不必要的括号。
  4. ASCII码。如：‘0’可以替换为48，‘1’可以替换为49，可以节省1B
  5. 循环合并。在某些情况下，结合%运算，两层循环可以合并成一层。
  6. 头文件。当需要用到bitset或者其他函数时，用#include<bits/stdc++.h>；此外，用#include<ios>就可以了。
  7. using namespace std；。用了这个之后就不用std::，但是这一段是20B，当std::超过4个时才需要用using namespace std；。
  8. 自增、自减。如i++可以在statement中最后出现的i结合，可以省1B。
  9. for(;;)。如果可以把循环体的statement全部放在update部分，那么可以省去括号。
  10. 宏定义。宏定义重复出现的部分，以宏定义代替。
  11. 变量命名。采用ABCD命名法。
  12. 除了#开头的必须单独一行以外，其他的代码都可以缩为1行，把所有的\t，\n能省就省。
  13. ……

以上的压缩代码方式就粗略地讲了一下，另外实在还有很多很多的压缩代码细节，能一点一点地压缩代码，在此就不一一叙述了。之后呈现的解压代码都是压缩好的，就不再展示压缩前的版本以及压缩过程了。

* 1. 验证步骤

1. 假设有解压代码code.cpp，编译生成code.exe可执行文件
2. 在当前文件夹打开cmd，运行以下命令，输出结果到output.txt

code.exe>output.txt

1. 把原文件input.txt放到同一个文件夹，运行以下命令进行比较

fc output.txt input.txt

1. 若输出“无差异”，则成功生成。
2. **Input 1: Pascal**
   1. 总体思想

ASCII编码转换

LZ77压缩

LZ77优化

* 1. ASCII编码转换

文本大小是8的倍数，那么首先可以把input1用ASCII编码方式转换。转换后发现是一篇有16,895个字符的Pascal代码。可以对代码内容直接进行压缩。

* 1. 初始想法

Pascal代码，和一般的代码一样，最大的特点是有很多重复的字段，如各种关键字、变量名。那么，首先想到的一种可行的压缩方法是，给每个重复字段指定打印的位置，如inline出现在了1位置、10位置，tx出现在了53位置、60位置，可以用map<string,vector<int>>来进行位置的记录，压缩过程是O(n)的。但是解压时，需要遍历全部，直到找到当前位置要输出的字符串，才能输出，所以是O(n2)的。

但是这种压缩方式，有很多的弊端。位置范围大，16895个字符需要用15位的地址表示，而对于大量的单独出现的字符、同一字符连续出现等情况，由于要记录很多位置信息，在重复串不多、重复长度短的情况下压缩效果很差，还很可能不减反增。因此这种方式不可取。

* 1. LZ77压缩

虽然不能记录每个重复串的位置，但是在输出过程中是可以利用前面出现过的字符的。如果记录好当前串与上一个重复串的距离d、重复长度l，那么在解压时，可以往前d个字符，取出长度为l的串，并打印出来，那么将能达到不错的压缩效果。根据这个思想，我找到了LZ77算法。

* 1. 定义
     1. lookahead buffer：等待编码的区域
     2. search buffer：已经编码的区域，搜索缓冲区
     3. 滑动窗口：指定大小的窗，包含“搜索缓冲区”（左） + “待编码区”（右）
  2. 编码思想

为了编码待编码区，编码器在滑动窗口的搜索缓冲区查找直到找到匹配的字符串。匹配字符串的开始字符串与待编码缓冲区的距离称为“偏移值”，匹配字符串的长度称为“匹配长度”。编码器在编码时，会一直在搜索区中搜索，直到找到最大匹配字符串，并输出(d, l )，其中d是偏移值，l是匹配长度。然后窗口滑动l，继续开始编码。如果没有找到匹配字符串，则输出(0, 0, c)，c为待编码区下一个等待编码的字符，窗口滑动1。以上所有信息都采用二进制编码，编码完成后，还能转换为ASCII编码，压缩为原来的1/8。

* 1. 编码过程
     1. 设置编码位置为输入流的开始
     2. 在滑窗的待编码区查找搜索区中的最大匹配字符串
     3. 如果找到字符串，输出(偏移值d， 匹配长度l)，窗口向后滑动l
     4. 如果没有找到，输出(0, 0, 待编码区的第一个字符c)，窗口向后滑动一个单位
     5. 如果待编码区不为空，回到步骤2；否则结束
  2. 伪代码

while( lookAheadBuffer not empty )

{

get a pointer (position, match) to the longest match

in the window for the lookAheadBuffer;

output a (position, length, char());

shift the window length+1 characters along;

}

* 1. 解压过程
     1. 从辅助文件中读取信息
     2. 如果为(0,0,c)，输出c；否则为(d,l)，那么输出d距离前长度为l的串
     3. 如果辅助文件读取结束，则结束；否则转(1)
  2. LZ77优化

但是LZ77有几个问题需要优化/细化：

1. (0,0,c)中使用了2个0，浪费储存空间。
2. 二元组和三元组的结构定义不方便区分。

其实只需要在每组信息开头用1或0进行标识，如果是1，那么接下来要读取d、l，如果是0，那么接下来要读取c，然后接着输出即可。而且，对于标准ASCII码，本来字符开头的MSD就是0，故对于单独的c的编码，0标识可以省略——当读取到0标识时，接下来要读取7个bit，并组成一个c输出。

1. d的编码长度、单独编码的界限

对于上述的压缩方法，重复串、单独c的编码长度是：

1. lrepeat = 1 + d的编码长度 + l的编码长度
2. lsingle = 8

设重复串的长度为x，那么当x \* lsingle < lrepeat时，就应当单独输出这x个字符，而不应该编码为重复串（当 = 成立时两种编码长度一样，随便一种就可以）。而我所预估的 lsingle > 16，故x<=2时将单独编码，当x>=3时编码为重复串。

由于所有重复串至少有3个字符，那么编码时重复长度l可以统一减去3，在解压的时候统一加上3，那么对于n位编码长度的l，原来可以编码的长度是0~2n-1，优化后的编码范围为3~2n-1+3，重复长度的范围从2n-1增大为2n-1+3，在编码长度不变的情况下提高了重复长度上限，进而提高压缩率。

* 1. 调整参数

以上的压缩、解压方法都是基于事先固定好d、l、c的编码长度，才可以实现，其中d、l的长度是需要设定的。

|  |  |  |
| --- | --- | --- |
| 项目 | 编码长度为n | 如果编码长度n变大，虽然编码变长，但是可以 |
| d | 搜索缓冲区长度2n | 提高重复的可能性，增大重复长度 |
| l | 重复长度2n-1+3 | 提高重复长度上限 |

因此针对不同的文本d、l是存在一个最优的平衡的。

定义结构：d编码长度Cd - l编码长度Cl – l重复串长度下限Cleast

1. 其中，如上文中提到的，Cleast设为x \* lsingle < lrepeat 中x的最大解+1。
2. 另外，Cd < log2(文本长度)+1，因为Cd = log2(文本长度)+1时已经能覆盖整篇文章了，增加编码长度也没有用了。
3. 如，14-6-3表示d编码长度为14，l编码长度为6，重复串长度下限为3。

最开始预设时为14-6-3，然后不断测试临近范围的版本，试图寻找最优的编码长度设置。

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 大小 | 版本 | 大小 |
| 12-4-2 | 40,231 B | 14-5-3 | 38,272 B |
| 13-4-2 | 39,798 B | 14-6-3 | 38,376 B |
| 13-5-3 | 38,385 B | 15-4-3 | 41,672 B |
| 13-6-3 | 38,636 B | 15-5-3 | 39,786 B |
| 14-4-3 | 39,988 B | 15-6-3 | 39,824 B |

从中14-5-3有最短的编码长度，比邻近的14-3、14-6、15-5、13-5、15-6、13-4都要短，因此这个编码长度的设置是最优的。

最后再对编码文件进行ASCII编码，在解压的时候先把辅助文件中的所有字符都变成二进制数并储存，然后再进行解压缩即可。

* 1. 最终实现（包括复杂度分析）

1. 压缩过程

|  |  |
| --- | --- |
| #include <iostream>  #include <algorithm>  #include <string>  #include <bitset>  #define BUFFER 16384  #define LOOK 34  #define DISTANCE 14  #define LENGTH 5  using namespace std;  string content;  int main()  {  freopen("pascal.txt", "rb", stdin);  freopen("feature14-5.txt", "wb", stdout);  unsigned char c;  string s;  while (~scanf("%c",&c))  {s = c;content += s;}  int ci, bi;int tci, tbi;int ml, md;int l;  int i, j;int eight;int length;  length = content.size();  cout << bitset<8>(content[0]);  for (ci = 1; ci < length; ci += ml)  {  md = ml = 0;  for (bi = ci - 1; bi >=0 && bi >= ci - BUFFER; bi--)  {l = 0;  for (tbi = bi, tci = ci; ; )  {  if (l >= LOOK || tci >= length)  {ml = l;md = ci - bi;l = 0;break;}  if (content[tci] == content[tbi])  {l++;tci++;tbi++;}  else  {  if (l > ml)  {ml = l;md = ci - bi;}  break;  }  }  }  if (ml - 3< 0)  {ml=1;cout<<bitset<8>(content[ci]);}  else  {cout << 1;  cout << bitset<DISTANCE>(md);  cout << bitset<LENGTH>(ml - 3);  }  }  } |  |
| 读入Pascal代码，输出到辅助文件 |
|  |
| 把整段Pascal代码存在string里方便遍历。复杂度为O(n)。 |
|  |
| 编码第一个字符 |
| 遍历string，编码剩下的字符 |
|  |
| Brute Force遍历搜索缓冲区，以寻找最长重复串的长度l和距离d。复杂度为O(16384\*34)。加上外层循环后为O(n\*16384\*34)。 |
|  |
| 如果最长重复串长度l小于下限长度，则单独编码当前字符 |
| 否则，编码为重复串 |

总体的复杂度大约为O(n\*16384\*34)，由于n=16,895，34>log2n，则复杂度级别约为O(n2 log2n)。

1. 解压过程

|  |  |
| --- | --- |
| #include<bits/stdc++.h>  #define G(x,i,d) for(x=0,j=d;j>0;j--)x|=s[i+j]-48<<d-j;  using namespace std;  string s,a;  int p,i,j,k,x,y;  main()  {  freopen("1","rb",stdin);  while(~scanf("%c",&x))  s+=bitset<8>(x).to\_string();  while(i<s.size())  if(s[i]-48)  {G(x,i,14)G(y,i+14,5)while(y--+3)a+=a[p++-x];i+=20;}  else  {G(x,i,7)a+=x;p++;i+=8;}  while(k<p)cout<<bitset<8>(a[k++]);  } |  |
| 输出先输出字符到string，最后再遍历转换为二进制数输出 |
| 读入文件，转换为二进制数，存到string中。复杂度O(n)。 |
| 读入编码，输出。由于重复串回溯长度最大为34，比log2n大，复杂度约为O(n\*log2n)。 |
| 重复串，读入d、l，在输出string中对应位置找到重复串，接到string后面 |
| 单独字符，接到string后面 |
| 遍历输出string，把字符转换为二进制码输出。复杂度为O(n)。 |

总体的复杂度约为O(n\*log2n)。

1. **Input 2: English**
   1. 总体思想

ASCII编码转换

复用Input 1代码

调整参数

* 1. ASCII编码转换

文本大小是8的倍数，那么首先可以把input2用ASCII编码方式转换。转换后发现是一篇有33,045个字符的English文章。可以对代码内容直接进行压缩。

* 1. 复用Input 1代码

这次的文本是一篇English文章，本质上与Pascal代码也是一样的，也有不少的重复单词/句型出现，那么可以使用相同的方法来进行压缩和解压。因此可以复用Input 1的代码。

* 1. 调整参数

但是针对不同的文本，d、l存在一个不同的最优平衡，因此还需要重新设置参数。

最开始预设时为14-5-3，然后不断测试临近范围的版本，试图寻找最优的编码长度设置。

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 大小 | 版本 | 大小 |
| 13-2-2 | 131,048 B | 14-4-3 | 127,344 B |
| 13-3-2 | 125,276 B | 14-5-3 | 132,020 B |
| 13-4-2 | 128,008 B | 14-6-3 | 137,621 B |
| 13-5-3 | 132,918 B | 15-3-3 | 125,302 B |
| 13-6-3 | 138,652 B | 15-4-3 | 130,520 B |
| 14-2-3 | 132,335 B | 15-5-3 | 135,150 B |
| 14-3-3 | 124,976 B | 15-6-3 | 140,688 B |

从中14-3-3有最短的编码长度，比邻近的14-2、14-4、13-3、15-3、15-4、13-2都要短，因此这个编码长度的设置是最优的。

最后再对编码文件进行ASCII编码，由于编码文件不是8的倍数，因此在文件结尾补上了6个0，再进行ASCII编码。而解压的时候先把辅助文件中的所有字符都变成二进制数并储存，然后再进行解压缩，且无视掉最后的6个0即可。

* 1. 最终实现（包括复杂度分析）

1. 压缩、解压过程与input 1的一致，只是参数不一样而已，故不再叙述。
2. 压缩的复杂度约为O(n\*16384\*10)，又n=33,045，故为O(n2 log2n)。对于解压过程，由于重复串回溯长度最大为10，略小于log2n，那么复杂度约为O(n\*log2n)。
3. 由于文本重复单词少且短，导致了压缩率相比于input 1有所下降。
4. **Input 3: Chinese**
   1. 总体思想

ASCII编码转换

复用Input 1代码

调整编码

调整参数

* 1. ASCII编码转换

文本大小是8的倍数，那么首先可以把input3用ASCII编码方式转换。转换后发现是一篇有44,787个字符的中文文章。可以对代码内容直接进行压缩。

* 1. 复用Input 1代码

这次的文本是一篇中文文章，本质上与English也是一样的，也有不少的重复单词/句型出现，那么可以使用相同的方法来进行压缩和解压。因此可以复用Input 1的代码。

* 1. 调整编码

中文与英文不同的是，中文的ASCII编码用的是扩展的ASCII编码，因此单独输出的c的MSD是可能为1的。那么之前的“通过检查到0，就判断为单独输出字符，读取7bit，并输出字符”的方式不再适用。因此编码时还必须要在单独输出的字符前加上标识0，那么在编码单独输出字符时，在输出字符编码之前先输出一个0即可。此时lsingle = 9，对应的重复串长度下限Cleast也需要按照上文的公式重新计算，在此不再叙述。

* 1. 调整参数

针对不同的文本，d、l存在一个不同的最优平衡，因此还需要重新设置参数。最开始预设时为14-4-2，然后不断测试临近范围的版本，试图寻找最优的编码长度设置。

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 大小 | 版本 | 大小 |
| 14-2-2 | 131,048 B | 14-4-2 | 127,344 B |
| 13-3-2 | 125,276 B | 14-5-3 | 132,020 B |
| 13-4-2 | 128,008 B | 14-6-3 | 137,621 B |
| 13-5-2 | 132,918 B | 15-3-2 | 125,302 B |
| 13-6-3 | 138,652 B | 15-4-3 | 130,520 B |
| 14-2-2 | 132,335 B | 15-5-3 | 135,150 B |

从中15-3-2有最短的编码长度，比邻近的15-2、15-4、14-3、14-2都要短，因此这个编码长度的设置是最优的。

最后再对编码文件进行ASCII编码，由于编码文件不是8的倍数，因此在文件结尾补上了4个0，再进行ASCII编码。而解压的时候先把辅助文件中的所有字符都变成二进制数并储存，然后再进行解压缩，且无视掉最后的4个0即可。

* 1. 最终实现（包括复杂度分析）

1. 压缩、解压过程与input 1的一致，只是编码稍有不同、参数不一样而已，故不再叙述。
2. 压缩的复杂度约为O(n\*32768\*10)，又n=44,787，故约为O(n2 log2n)。对于解压过程，由于重复串回溯长度最大为10，小于log2n，那么复杂度约为O(n\*log2n)。
3. 由于单独输出字符前要加上标识0，而且文本重复单词少且短，导致了压缩率相比input 1、2大幅下降。
4. **Input 4: Random Numbers**
   1. 总体思想

寻找规律：随机数

寻找种子

组合生成

* 1. 寻找规律

1. 文件大小为320,000，是8的倍数。但用ASCII编码为乱码，没用。
2. 文件也是32的倍数。尝试用int型数字编码，输出10,000个数字。
3. 分析数字。首先分析到数字的范围为-2147432634~2147238465，且没有重复。因此这是一个大范围的、无重复的、大量的一组int数，猜测可能与随机数有关。
4. 直接循环输出一组rand()的十进制及二进制表示（一般默认种子为1），观察该组随机数与文本中的数是否有算术运算或逻辑运算上的关系。算数上，十进制rand()与十进制的文本没有明显的和差关系；但是逻辑运算上，发现二进制rand()与二进制文本有关系，文本的第一个数与rand()的第一个数、第二个数有如下重叠的关系：

|  |  |
| --- | --- |
| -------------------------------00000000000101001100100000100011  ------00000000000000000000000000101001  -------------------------------00000000000000000100100000100011 | input 4第一个数 |
| rand()第一个数 |
| rand()第二个数 |

1. 经过对input 4的前10个数进行检验，发现：

input 4中第x个数=rand()第2\*x个数<<15|rand()第2\*x+1个数

那么猜测整个文本都符合这个规律。但是这一规律是否正确还需要验证。

* 1. 生成

在程序中，第n次调用rand()就会输出第n个随机数。那么根据上面这个规律，只要不断地输出rand()<<15|rand()的二进制表示即可。

* 1. 实现并验证（包括复杂度分析）

|  |  |
| --- | --- |
| #include<bits/stdc++.h>  int i;  main()  {  while(i++<1e4)  std::cout<<std::bitset<32>(rand()<<15|rand());  } |  |
| 循环输出逻辑运算结果，复杂度为O(n)。 |

经过文件对比，发现没有不同，说明这个猜想是正确的。复杂度为O(n)。

1. **Input 5: Photo**
   1. 总体思路

寻找规律：照片

寻找压缩方法

失败

* 1. 寻找规律

1. 用ASCII编码转换，乱码，没意义。
2. 尝试在二进制文本下寻找规律。发现文本中很多1111 1111，也就是255，于是针对这一特点查找资料，发现这一特点与jpeg图片的编码很类似，因此猜测这可能是一张图片的二进制编码。
3. 于是，把文本的ASCII编码转换后的.txt文件后缀改为.jpeg，打开后发现是一张“愤怒的丘吉尔”的图片。
   1. LZ77压缩
4. 尝试：复用input 1的代码来压缩input 5.txt。结果只短了一点点，压缩效果极差。
5. 原因分析：input 5用ASCII编码后的文本为乱码，文本重复度极小，使得重复的字符串中长度超过下限Cleast的极少，极大部分字符都是单独输出的，故压缩效果很差。
   1. Huffman压缩
6. 尝试：用Huffman编码对ASCII编码后的文本进行压缩。结果反而变长了，不可行。
7. 原因分析：input 5的文本很可能是已经经过Huffman编码压缩的了，故再次压缩会反而变长。
   1. 最终实现（包括复杂度分析）：ASCII编码

最终都没有找到很好的压缩方式，因此最终采取ASCII编码进行压缩。

1. 压缩过程

|  |  |
| --- | --- |
| #include <bits/stdc++.h>  using namespace std;  int main()  {  int i, k;char c;  freopen("input5.txt", "rb", stdin);  freopen("photo.txt", "wb", stdout);  do  {  for (i = 0, k = 0; i < 8 && (c = getchar()); i++)  k = k \* 2 + c - '0';  } while (c != EOF && cout << (char)k);  } |  |
| 读入input 5文本，输出到辅助文件 |
| 把8个bit转换为1个char，并输出到文件。复杂度为O(n)。 |

总的复杂度是O(n)的。

1. 解压过程

|  |  |
| --- | --- |
| #include<bits/stdc++.h>  int c;  main()  {  freopen("5","rb",stdin);  while(~scanf("%c",&c))  std::cout<<std::bitset<8>(c);  } |  |
| 读入辅助文件 |
| 输出每个字符的二进制表示。复杂度为O(n)。 |

总的复杂度为O(n)。

1. **Input 6: Gray Code**
   1. 总体思路

寻找规律：格雷码

生成：递归法

生成：异或法

* 1. 寻找规律

1. 尝试用ASCII编码转换，结果乱码无意义。
2. 由于文件还是32的倍数，尝试把每32 Byte转换为一个对应的int，获得1,536个数字，显然没有单调性。对这些数字用map统计，发现数据范围大，较分散，每个数也只出现一次，没有获得什么特殊规律。
3. 转向在二进制层面寻找规律。初步观察，发现文件开头和结尾0比较多，中间1比较多，有一种递增再递减的效果。仔细观察开头，由于0较多，1较少，故注意到1上面，大概看出来在差不多长的部分中都会有少量1出现。
4. 尝试把文件中的01串分成等长的部分，观察部分与部分之间有没有特别的关系。由于第一行中的1分布似乎较均匀，故尝试在第一个1的前后断开，尝试以这个长度来分割整个文件，也就是12 B一部分或者13 B一部分。由于13并不是49,152的因数，而12是，故把文件分成12 B的一段段。

000000000000

100000000000

110000000000

010000000000

011000000000

111000000000

101000000000

……

观察了前几段，每部分与上下部分都只有一位不一样，故猜测文件是12 bit格雷码，但仍需要进一步生成并验证。

* 1. 递归法生成（线性递归）

1. 基础：1位格雷码

0

1

1. 递归：n-1位格雷码生成n位格雷码（以n=4为例）

|  |  |  |  |
| --- | --- | --- | --- |
| n-1=3位格雷码 | 正序 | 右补0 | n=4位格雷码 |
| 000  100  110  010  011  111  101  001 | 000  100  110  010  011  111  101  001 | 0000  1000  1100  0100  0110  1110  1010  0010 | 0000  1000  1100  0100  0110  1110  1010  0010  0011  1011  1111  0111  0101  1101  1001  0001 |
| 反序 | 右补1 |
| 001  101  111  011  010  110  100  000 | 0011  1011  1111  0111  0101  1101  1001  0001 |

1. 退出：不断递归直至生成12位格雷码。
2. 实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<iostream>  #include<vector>  using namespace std;  vector<vector<int>> gray\_code(int bit)  {  vector<int> vi;  vector<vector<int>> vvi;  if (bit == 1)  {  vi.push\_back(0);  vvi.push\_back(vi);  vi[0] = 1;  vvi.push\_back(vi);  }  else  {  vvi = gray\_code(bit - 1);  int length = 1 << (bit - 1);  int sub\_length = bit - 1;  int i;  for (i = 0; i < length; i++)  vvi[i].push\_back(0);  for (i--; i >=0 ; i--)  {  vi = vvi[i];  vi[sub\_length] = 1;  vvi.push\_back(vi);  }  }  return vvi;  }  int main()  {  freopen("graycode.txt", "wb", stdout);  vector<vector<int>> vvi = gray\_code(12);  for (int i = 0; i < 4096; i++)  for (int j = 0; j < 12; j++)  cout << vvi[i][j];  } |  |
| 递归函数 |
|  |
| 当位数为1时，生成1位格雷码。 |
| 通过n-1位格雷码生成n位格雷码。 |
| 给正序的n-1位格雷码的后面补上0。复杂度为O(n)。 |
| 反序复制n-1位格雷码，并把最后一位改为1。复杂度约为O(n\*2^n)。 |
|  |
| 递归下去 |
| 递归生成并输出 |

总复杂度约是O(n\*2^n)。

1. 验证

结果验证为正确，说明input 6的确为12位格雷码（4096个）的全体。

1. 优化
2. 上述算法采用的是线性递归，因此可以转换为非递归实现（类似于Fibonacci的递归转非递归），减少堆栈的生成和消除，能提高效率、减少代码量。
3. 另外，可以把vector实现改为数组实现，还能较大减少代码量。
4. 但是即使优化过了，这种方法的复杂度还是太高了，代码量也仍然太大了。
   1. 异或生成
5. 格雷码生成公式

grey(x)=x^(x>>1); 其中，x>=0

1. 调整

按照上述公式可以按顺序生成格雷码，但是与input 6中的格雷码顺序刚好相反了

|  |  |  |
| --- | --- | --- |
| x | grey(x) | grey(x) in input 6 |
| 000000000000 | 000000000000 | 000000000000 |
| 000000000001 | 000000000001 | 100000000000 |
| 000000000010 | 000000000011 | 110000000000 |
| …… | …… | …… |

可以更改为以下公式，逐位生成grey(x)，首先生成LSD，最后生成MSD。那么就能生成倒叙的格雷码。

for (i = 0; i< 12; i++)

cout << (((x >> i) ^ (x >> (i + 1))) & 1);

1. 实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<ios>  int x;  main()  {  for(;x<49152;putchar((x/12^x/24)>>x++%12&1|48));  } |  |
| 此处把两层循环合并成一个循环。其中，外层循环为4096个自然数，内层循环为12，以生成倒叙的格雷码，共为4096\*12=49152。 |

总的复杂度为4096\*12=49152。

1. 结果

这个方式的复杂度低很多，代码量也极简，效果很好。

1. **Input 7: π**
   1. 总体思路

寻找规律：π的小数部分

生成：基于Spigot公式

* 1. 寻找规律

文本大小是8的倍数，那么首先可以把input7用ASCII编码方式转换。转换后发现是π的小数点后16,000位。那么可以运用π的高精度计算公式直接生成。

* 1. Spigot公式

计算π的高精度公式有很多，但是由于需要用程序实现，因此选取的公式最好仅限于整数的加减乘除运算，而这公式恰好只对整数做加、乘运算，很理想。

* 1. 实现思想

1. 对公式整体\*2，以计算出π。
2. 公式中每计算14项，便能获得4位精确的小数。故迭代14次后输出4位并储存到string中。
3. 由于公式计算的是31415926……而文本中的是1415926……故要生成16,000位，那么第一位要去掉，最后还要加上一位，由于计算过程以4位为一个单位，故一共生成16,004位，并输出1~16,000位（以第0位开始）。
4. 生成16,004位，一共要迭代16,004/4\*14=56014次。
5. 模拟长除法运算过程，用一个数组储存余数，下一步运算时先把余数扩大，做除法运算得到商，再做%运算获得余数并储存到数组中，在下一次运算的时候使用。以此生成高精度结果。
   1. 最终实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<ios>  int a=1e4,b,c=7e4,p,d,e,f[70001],g;  main()  {  char x[c];  for(;d=0,g=c\*2;c-=14, sprintf(x+4\*p++,"%.4d",e+d/a),e=d%a)  for(b=c;d+=(p?f[b]:2e3)\*a,f[b]=d%--g, d/=g--,--b; d\*=b);  for(;b<128e3;  putchar(x[b/8+1]>>7-b++%8&1|48));  } |  |
| 外层循环：4位4位地输出精确的小数到string中，复杂度约为O(n)。 |
| 内层循环：按公式计算14项，以生成精确的4位小数，复杂度为常数级别。 |
| 输出循环：输出1~16,000位的二进制表示，复杂度约为O(n)。 |

总的复杂度约为O(n)。

1. **Input 8: Ten Queens**
   1. 总体思路

寻找规律：十皇后

生成：递归法

生成：异或法

* 1. 寻找规律

1. ASCII编码为乱码，无意义。
2. 转向在二进制层次上寻找规律。发现文本中隔一定的距离，就会出现约10个连续的1，因此把文本划分为10位一段。发现每10段后都会出现1段全1，而每10段中，每一段都只有1个1。那么可以猜想，每10段为一部分，随后全1的一段起到了分割的作用。仔细观察开头的第一部分，发现不仅每行只有一个1，而且每列、每条斜线上都只有一个1。因此判断这是10\*10版本的八皇后问题，那么称之为十皇后。

1000000000

0010000000

0000010000

0000000100

0000000001

0000100000

0000000010

0100000000

0001000000

0000001000

11111111111

1. 猜想是否正确还需要进一步的生成并验证。
   1. 递归法
2. 十皇后规则：在10\*10的棋盘上放置10个棋子，每行、每列、每个斜线上只能有一个棋子。
3. 递归（深度遍历）所有可能的情况：一行行地放棋子，每一行中，逐个位置放置棋子，如果违反十皇后规则，则放下一个位置，再次判断是否符合规则，直至放置好这一行的棋子，接着递归到下一行棋子的放置。如果10行都放置好了，接着输出棋盘和最后的一行1，并返回，继续处理下一种可能的棋盘。
4. 用int c[10]记录每一行放置的棋子位置，放置好最后一行后，根据该数组输出二进制表示的棋盘。如c[5]=7，表示第5行（从第0行开始算）的棋子放在第7列（从第0列开始算），也就应该输出0000001000。
   1. 最终实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<ios>  int c[10],g=1,x,i;  q(int a)  {  if(a>9)  for(i=0;i<110;)  putchar(i%10==c[i/10]|i++/100|48);  for(int b=0;b<10;c[a]=b++,g?q(a+1):g++)  for(i=0;i<a;i++)  x=b-c[i],g\*=a-i^x&&x&&i-a^x;  }  main(){q(0);} |  |
| 递归（深度遍历）函数 |
| 如果棋盘放置完毕，输出棋盘，并返回。输出的两层循环合成了一层，因此复杂度为O(n^2)。 |
| 按顺序遍历当前行，直至找到可以放置的位置，记录位置，递归到下一行。每行每列都遍历，以及每个位置都要遍历判断是否合乎规则，故深度遍历的过程复杂度约为O(n^3) |
| 主函数设置递归起点。 |

最终输出结果与文本一致，说明猜想是正确的。

总的复杂度约为O(n^3)。

1. **Input 9: Random Numbers**
   1. 总体思路

寻找规律：随机数

生成：线性同余法

* 1. 寻找规律

1. ASCII码转换为乱码，无意义。
2. 把文本转换为int型数字。这10,000个数全都是正数，最大数为2147460346，所有数字没有重复，和input 4的感觉很像，像是随机数。于是尝试同样用rand()函数（默认种子为1）来生成这一组数，但是尝试过很多的算数运算、逻辑运算组合后，依然找不到合适的方式。
3. 于是转向寻找其他的生成随机数的方式。
4. 首先找到的是迭代取中法。但是这个方法是产生小数随机数的，而且很容易退化成0，跟文本中大量不重复的数矛盾。因此应该不是这个方法。
5. 第二个找到的方法是线性同余法。
   1. 线性同余法
6. 作用

用于产生整数随机数，在计算机中生成随机数多用这种方法，而且rand()函数本身也是基于这种方法实现的，能生成大量不重复数，效果很好。

1. 公式

rand[n+1]=(rand[n]\*a+b)%c

事先给出初始种子rand[0]，跟据上述公式可以迭代生成大量随机数。

1. 关键是要求出公式中的参数a、b、c以及初始的种子rand[0]
2. c
3. 如果c越大，那么余数则可能越大；
4. 如果c的因数越少，那么余数重复的可能越大

由于文本中的数最大有2147460346，而且不重复数量大，因此c是一个很符合上面性质的一个数，而2^31-1就是这样的一个数。因此猜测c=2^31-1。

1. 由于随机数很大，c也很大，很容易会溢出，而且本身所有随机数都为正数，故设置随机数的类型为unsigned int。
2. a和b
3. 联立方程

rand[1]=(rand[0]\*a+b)%c

rand[2]=(rand[1]\*a+b)%c

其中：c=2147460347, rand[0]=1162414137, rand[1]=1035654704, rand[2]=893643089

1. 利用公式

(x+y)%c=(x%c+y%c)%c

通过这个公式可以消去b，先求出a

1. 消去b

(rand[1]-rand[2])%c=(rand[0]-rand[1])\*a%c

从0开始遍历所有的a，求得a=16807或991511918是成立上述条件的

1. 先取a=16807，代入rand[1]=(rand[0]\*a+b)%c，从最小的数开始暴力遍历b，求得b=0是符合条件的。
2. 获得一个可能的公式rand[n+1]=(rand[n]\*16807)%((1<<31)-1)
3. 在实现的过程中，发现从rand[-1]开始计算会减少代码量，因此计算rand[0]=(rand[-1]\*16807)%((1<<31)-1)

求得rand[-1]= 1508303647

1. 验证证明公式正确
   1. 最终实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<ios>  unsigned a=1508303647,i=32e4;  main()  {  while(i)  putchar((i--%32? a:a=a\*16807%~(1<<31))>>i%32&1|48);  } |  |
| 设置初始种子及范围 |
|  |
| 迭代生成。这里把迭代的循环和输出随机数的二进制表示的循环合并了。复杂的约为O(n)。 |

总复杂度约为O(n)。

1. **Input 10: Permutation**
   1. 总体思路

寻找规律：

三部分

第一部分决定第三部分

生成

修正

* 1. 寻找规律

1. ASCII编码为乱码，无结果。
2. 二进制层面上找规律。发现中间有65535个连续的1（第二部分），猜想这个和input 8中连续的10个1一样起到分割的作用，而这一部分前面有65535（第一部分），后面有1048576（第三部分）。
3. 单独对第一、第三部分进行截断，尝试在二进制层面上寻找规律，但是最终没有结果。另外还尝试转换为char、int等格式，也没有找到特别的规律。
4. 最终，猜想第一部分可能和第三部分之间有关系（如果没有关系的话不就会分成两个input了吗？）。而第一部分差1才是65536=2^16，第三部分是2^20=2^16\*16。于是对第三部分尝试用16位转换为一个数字，发现第三部分是0~65535共65536个数字的排列。
5. 于是进而寻找该排列的规律。最终没找到什么规律。这是想起来，第一部分可能与第三部分有关系，那么有可能是第一部分的信息指定了第三部分中数字排列的顺序，那么转向寻找这种指定关系。
6. 经过了不断的折腾之后，最后发现：如果把第一部分的01串倒过来，发现0和1恰好对应着第三部分排列的增减性。但是这样的性质不足以生成排列，还需要更强的条件。
7. 既然01对应着增减性，那么01的个数可能与数字的出现位置有关。最后统计第一部分中每个数字附近的0、1的个数，发现第一部分倒过来，把第一个数编号为65535，并以此递减编号，发现除了编号1、2之外，如果编号后边连续的0越多，在第三部分出现的越靠前。
8. 根据这一性质，给第一部分中每个数字都编上号，并按照后面的跟着的连续0的个数多少作为关键字来排序，然后按连续0个数从多到少顺序输出对应的编号，结果失败。仔细看，发现基本顺序是对的，但是对于后继连续0的个数相同的几个数的顺序不对。
9. 于是忽略掉第一层连续0后面的连续1，继续统计第二层连续0的个数，把这个个数作为第二关键字进行排序。然而还是有的编号具有相同的第一层、第二层连续0的个数，于是继续求第三层、第四层、……的连续0的个数，直至求出第10层连续0的个数后，发现不同的编号所有的关键字不完全相同。
10. 检验的时候出现一个小问题，从15到3，第一优先级的连续0结束之后，第二优先级的连续0就到了文末，使得第二优先级特别小，但是正确的结果表明15到3的第二优先级应该是最高的，所以在文末又加了45个0。
11. 最后编码1和2特判，然后0当做65536。
12. 最后按照这个方法生成，结果正确！（代码就不再给出了）求连续0的个数的复杂度约O(n\*log2n)，而排序的复杂度约为O(n\*log2n)，故总的复杂度约为O(n\*log2n)。
13. 优化：上述方法主要是根据连续0的个数来进行排序的，然而如果直接按照每个数字后面的一定长度的字符串（称为特征串）来进行排序，也同样反映了连续0的个数，故效果也是一样的。但是这样修改这之后，原来的求1~10层连续0的个数的循环，可以简单地改为用s.substr(position,length)来获取s在position位置的长度为length的后缀子串，而且最后按照string这一个关键字进行排序就行了，在struct函数中定义按照string进行排序的bool函数，比定义按照储存在数组中的10层连续0的个数来进行排序的代码少得多。综上，复杂度与原方法量级相同，为O(n\*log2n)，但是常数会稍大一点（因为之前的求完第10层就停了，但是这个length至少要取最短的长度，故常数会稍大些），尽管如此，代码量将大大减少，更符合评比标准。
14. 还有一些需要修正的细节在下文中叙述。
15. 最后按照这个方法实现，生成的顺序与第三部分也是一致的。
16. 最后还要对作为生成第三部分的依据信息的第一部分进行ASCII编码，作为辅助文件。在最终生成函数中，首先输出辅助文件的二进制表示，再输出分割的1，再根据第一部分生成第三部分。
    1. 最终实现（含复杂度分析）

|  |  |
| --- | --- |
| #include<bits/stdc++.h>  using namespace std;  int i,q=65536,p;  struct e  {  int p;  string s;  bool operator<(e&n)  {return s<n.s;}  }a[65536];  main()  {  freopen("0","rb",stdin);  string s;char c;  while(~(c=getchar()))  s+=bitset<8>(c).to\_string();  a[0].s=s;  cout<<s.substr(++i,q);  for(;i<q;i++)  cout<<1;  for(;--i;)  {  a[i].s=s.substr(++p,99);  a[i].s[0]=49;  a[i].p=i;  }  a[1].s=48;  sort(a,a+q);  for(;i<q;)  cout<<bitset<16>(a[i++].p);  } |  |
| 定义结构体：编号、作为比较关键字的string、按照string进行排序的比较函数 |
|  |
| 定义为一个65536的数组，以对全部数进行排序。 |
| 读入第一部分的文件 |
|  |
| 读入第一部分信息，转换为二进制并储存。 |
| 特殊处理编号为0的特征串 |
| 输出第一部分 |
| 输出第二部分 |
| 复制剩下的子串。总复杂度约为O(n) |
| 特殊处理编号1的子串 |
| 按照字符串进行排序。复杂度约为O(n\*log2n)。 |
| 按照获得的顺序输出编号 |

总的复杂度为O(n\*log2n)。

1. **Reflections**

通过这次Project，第一次接触到压缩方面的知识，学习到了压缩算法、代码优化、代码压缩这些以前从来没有尝试、了解过的东西，也锻炼了自己寻找文本规律、辨析数据特征的能力，虽然总是会被一些input卡了好久好久，但是总的来说收获还是满满的！