1.进行文件压缩的必要性

像图片、声音、视频这些类型的多媒体数据要比文本数据占用多得多的内存空间，尤其是视频文件，文件传输时占用带宽大，存储又占用大量的硬盘空间。

举个例子：***一个1080p分辨率格式下90分钟的无压缩视频要多大?***

1帧大小 = 1920 x 1080 x 3 = 6220800 bytes (1920x1080是每一帧的像素数，3指的是每个像素红绿蓝三个通道各占一个字节0~256)

每秒大小 = 6220800 x 25 = 155.52MB!(假设帧率为每秒25帧，很小了！)

每分钟大小 = 155.52MB x 60 = 9.3312GB !

90分钟：大约839.81GB

存储高清视频的蓝光光碟容量不过只有大约50GB，所以视频如果不压缩根本没法存储，更不用说互相传送了。

2.简单黑白图像的压缩

假设黑白图像的数据如下图，黑色像素用1编码，白色图像用0编码：

记录连续出现相同数字的个数

黑白图像的尺寸为16x8 = 128,因此需要128bits来表示它。

如果我们从0开始算起，只保存一些列0和1的个数，那么上面的图像信息可以表示为：

21 6 9 1 5 2 8 1 3 2 1 1 8 1 2 2 2 1 8 6 21

该两种方法缺点，无法进一步压缩

其中最大的数字是21，所以可以统一使用5bits(2^5=32)来表示每一个数字,那么现在的存储空间变为5x21=105bits，节省了23bits。

3.字符串行程编码

和上面黑白图像压缩一样的道理，这里压缩一段字符串：**RRRRRGGGBBBBBRRRRGB**

压缩的结果为：**5R3G5B4R1G1B**

这种方法叫做**行程编码run-length encoding(RLE)**，应用于像BMP、TIFF格式的图像文件中

但这种压缩编码方式很有局限性，我们无法继续用相同的方法进一步压缩压缩后的数据，比如上面的压缩结果无法继续用这种方法压缩，这种压缩基于数据的重复性。

4.信息的可压缩性-信息量和信息熵

信息量

信息量是数据的可压缩水平

数据的压缩水平和数据的信息量有关，信息量越大自然数据量越大越难以压缩。数据的信息量如何来衡量呢?

例子：假设有一个64位的字符串，64个数字，其中63个0，另一个是1，1可能出现在任意位置，也就是在某个位置的概率为1/64。现在从左往右读取知道找到1。**（注意下表是找到1之前的概率，当找到1之后，之后的符号确定不可能再出现1了，因此P(bi=1)变为0，P(bi=0)变为1）**

对于在位置i的符号bi,bi为0或者为1的概率分布如下：

i	1	2	3	4	……	32	33	…	62	63	64
P(bi=0)	63/64	62/63	61/62	60/61		32/33	31/32		2/3	1/2	0
P(bi=1)	1/64	1/63	1/62	1/61		1/33	1/32		1/3	1/2	1

符号出现的概率越大信息量越小

事实上，一个符号的概率越大，那么它包含的信息就越小，也就是信息量和符号出现的概率成反比，信息量的定义为：

$$h(x) = log_2 \frac{1}{p(x)}$$

某种程度反比概率

计算对应的符号信息量表：

i	1	2	3	4	…	32	33	…	62	63	64
P(bi=0)	0.0227	0.023	0.0235	0.0238		0.444	0.458		0.585	1	0
P(bi=1)	6	5.977	5.954	5.9307		5	5.0666		1.585	1	0

信息量计算：如果1位于第4个位置：000100…0，则总的信息量为：

Sum = 0.0227 + 0.023 + 0.0235 + 5.9307 = 6

通用公式：

1到i有信息量　字符0的信息量　字符1的信息量

$$\sum_{n=1}^{i} h(b_n) = (\sum_{m=66-i}^{64} log_2 \frac{m}{m-1} + log_2 \frac{1}{1/(65-1)}) = log_2((\prod_{m=66-i}^{64} \frac{m}{m-1})(65-i)) = 6$$

每个字符信息量的和表示了整个字符串的信息量，这里至少需要6位来表示这个字符串，要保存的是公式中的索引（0~63）。

信息熵

字符串的信息量=字符信息的和

数据的压缩实际是用更短的数据来表示反复出现的数据实现压缩，因此数据重复率越高或者可预测性越强可压缩性就越高，不同数据可压缩的程度不一样，信息熵是用来衡量数据可压缩的程度的一个参数，计算信息最短的长度的期望，关于信息熵：http://www.ruanyifeng.com/blog/2014/09/information-entropy.html

上面能精确计算信息量是因为我们知道了字符串的具体结构，对于未知的信息我们只能根据概率计算其信息量的期望，计算公式为：

最差的情况是当所有符号的概率都相同，概率均匀分布，此时信息熵为最大值，因为对于一个子串，根本无法预测下一个符号。

5.对编码算法的要求

1. 通过编码后的编码必须可以准确解码，编码必须确定地对应一种原码；
2. 编码算法得到的编码要容易解码，可以很容易的找到信息的末尾，可以在线解码，可以直接对编码进行解码而不用知道完整的编码信息；
3. 编码必须是压缩的，否则失去了编码的意义；

6.哈夫曼编码

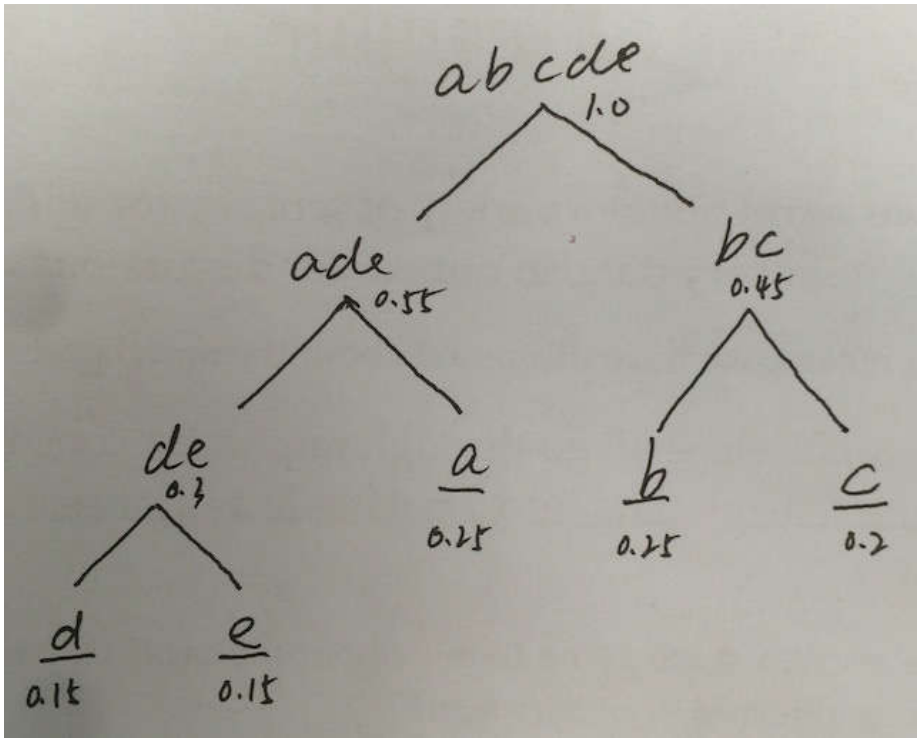
哈夫曼编码是David A. Huffman于1952年发明的一种满足上面对编码算法要求的一种编码算法。

举一个例子：知道一段字符串全部由a,b,c,d,e五个字母组成，已知了每个字母出现的频率：

a(0.25),b(0.25),c(0.2),d(0.15),e(0.15)

【如果不考虑编码算法，使用定长的编码来区别五个字母，不利用频率这些信息，那么五个字母每个字母需要用3bits表示(2^2=4,2^3=8).】

哈夫曼算法是利用频率信息构造一棵二叉树，频率高的离根节点近（编码长度短），频率低的离根节点远（编码长度长），手动构造方法是先将字母按照频率从小到大排序，然后不断选择当前还没有父节点的节点中权值最小的两个，构造新的父节点，父节点的值为这两个节点值的和，直到构造成一棵二叉树，上面的例子构造的Y一棵哈弗曼树如下（由于构造过程中叶子节点的值以及新节点的值可能会相同，所以哈弗曼树的结构不唯一）：



对构造的哈弗曼二叉树进行编码，左边为0，右边为1，就得到一个编码的哈弗曼树，从而对字符串进行编码。

哈夫曼算法C++实现，使用线性数组存储节点的方式实现，输入上面每个字母的权值可以得到哈弗曼树结构：

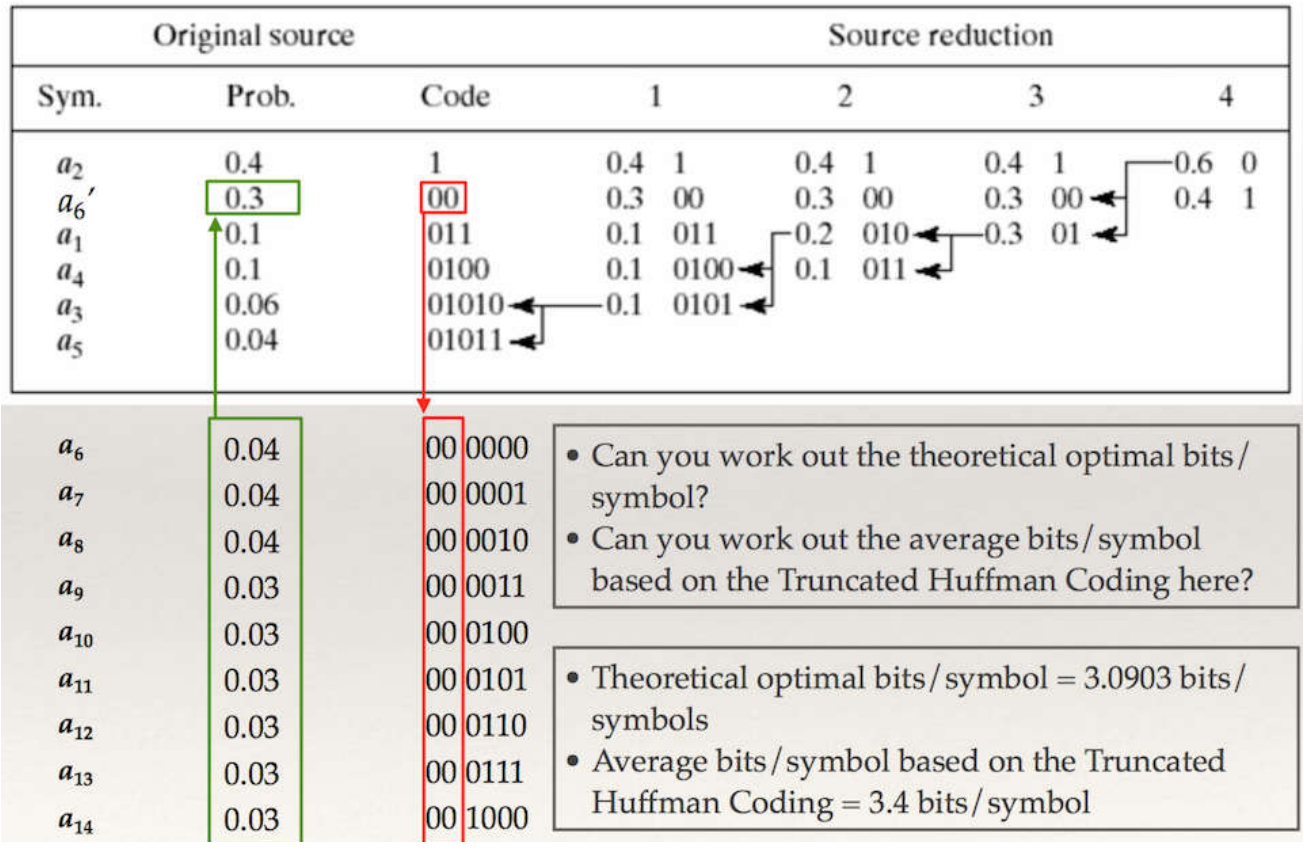
```
1  #include <iostream>
2  using namespace std;
```

```
6
7  typedef struct{
8      double weight;   // 节点权重
9      int lchild;      // 左子树
10     int rchild;      // 右子树
11     int parent;      // 父节点
12 }HTNODE;
13
14 typedef HTNODE HuffmanT[m];  // 一棵线性结构存储的哈弗曼树
15
16 /**
17  * 哈弗曼树初始化
18  */
19 void InitHT(HuffmanT T)
20 {
21     for(int i=0; i<m; i++)
22     {
23         T[i].lchild=-1;
24         T[i].rchild=-1;
25         T[i].parent=-1;
26     }
27     // 依次输入每个节点的权重
28     for(int i=0; i<n; i++)
29         std::cin>>T[i].weight;
30 }
31
32 /**
33  * 找出还没有父节点的节点中权值最小的两个,p1和p2是要选出的权值最小的两个节点的下标，n1是新父节点的下标
34  */
35 void SelectMin(HuffmanT T, int n1, int &p1, int &p2)
36 {
37     int i, j;
38     // 先任意找两个没有父节点的节点
39     for(i=0; i<n1; i++)
40         if(T[i].parent==-1)
41         {
42             p1=i;
43             break;
44         }
45     for(j=i+1; j<n1;j++)
46         if(T[j].parent==-1)
47         {
48             p2=j;
49             break;
50         }
51     // 搜索替换成权值最小的节点
52     for(i=0; i<n1; i++)
53         if((T[p1].weight>T[i].weight) && (T[i].parent==-1) && (p2!=i))
54             p1=i;
55     for(i=0; i<n1; i++)
56         if((T[p2].weight>T[i].weight) && (T[i].parent==-1) && (p1!=i))
57             p2=i;
58
59 }
60 /**
61  * 构造哈弗曼树
62  */
63 void CreatHT(HuffmanT T)
64 {
65     int i, p1, p2;
66     InitHT(T);
67     // 非叶子节点
68     for(i=n; i<m; i++)
69     {
```

```
72          T[p1].parent=T[p2].parent=i;
73          T[i].lchild=p1;
74          T[i].rchild=p2;
75          T[i].weight=T[p1].weight+T[p2].weight;
76      }
77  }
78
79  /**
80   * 打印哈弗曼树
81   */
82  void printHT(HuffmanT T)
83  {
84      for(int i=0; i<m; i++)
85      {
86          std::cout<<T[i].weight<<'\t'<<T[i].parent<<'\t'<<T[i].rchild<<'\t'<<T[i].lchild<<std::endl;
87      }
88  }
89
90  /**
91   * 前台测试
92   */
93  int main(){
94      HuffmanT T;
95      CreatHT(T);
96      printHT(T);
97      return 0;
98  }
```

缩减的哈夫曼算法

哈夫曼编码的缩减实质是对编码字符分组继续进行子分组内编码。先将所有字符按照出现的概率排序，将概率相近的字符作为一个整体参与上一级的编码，这样上一级的编码数量大大减少，分组内继续进行内部哈夫曼编码，同时在上一级中本组的编码作为组内编码的一个固定的前缀。



Shift Coding

和缩减的思想类似，将字符按照频率8个一组分块，每到下一块在前面加111进行区分后继续进行3bit的编码。

Source symbol	Probability	Binary Code	Huffman	Truncated Huffman	Binary Shift	Huffman Shift
Block 1						
a_1	0.2	00000	10	11	000	10
a_2	0.1	00001	110	011	001	111
a_3	0.1	00010	111	0000	010	110
a_4	0.06	00011	0101	0101	011	0111
a_5	0.05	00100	00000	00010	100	0110
a_6	0.05	00101	00001	00011	101	0101
a_7	0.05	00110	00010	00100	110	0100
Block 2						
a_8	0.04	00111	00011	00101	111 000	00 10
a_9	0.04	01000	00110	00110	111 001	00 111
a_{10}	0.04	01001	00111	00111	111 010	00 110
a_{11}	0.04	01010	00100	01000	111 011	00 0111
a_{12}	0.03	01011	01001	01001	111 100	00 0110
a_{13}	0.03	01100	01110	100000	111 101	00 0101
a_{14}	0.03	01101	01111	100001	111 110	00 0100
Block 3						
a_{15}	0.03	01110	01100	100010	111 111 000	00 00 10
a_{16}	0.02	01111	010000	100011	111 111 001	00 00 111
a_{17}	0.02	10000	010001	100100	111 111 010	00 00 110
a_{18}	0.02	10001	001010	100101	111 111 011	00 00 0111
a_{19}	0.02	10010	001011	100110	111 111 100	00 00 0110
a_{20}	0.02	10011	011010	100111	111 111 101	00 00 0101
a_{21}	0.01	10100	011011	101000	111 111 110	00 00 0100
Entropy	4.0					
Average length		5.0	4.05	4.24	4.59	4.13

7.Lempel-Ziv压缩算法

LZ算法及其衍生变形算法是压缩算法的一个系列。LZ77和LZ78算法分别在1977年和1978年被创造出来。虽然他们名字差不多，但是算法方法完全不同。这一系列算法主要适用于字母数量有限的信息，比如文字、源码等。流行的GIF和PNG格式的图像，使用颜色数量有限的颜色空间，其压缩就采用了两种算法的灵活变形应用。

LZ77：

推荐阅读文章：

LZ77压缩算法编码原理详解(结合图片和简单代码)

LZ77算法原理及实现

LZ77算法的思想是在编码解码过程中，使用之前刚结束编解码的部分数据的位置索引来代替当前要编解码的数据，压缩的实现靠的是之前编解码结束的部分数据和当前数据的重复性。

算法中几个重要的对象概念：

LAB(look-ahead-buffer):将要编码的固定长度的数据缓冲；

SB(search-buffer):刚过去的固定长度的数据缓冲，搜索缓冲区，也就是临时的数据字典，要从这里面搜索重复数据获得压缩索

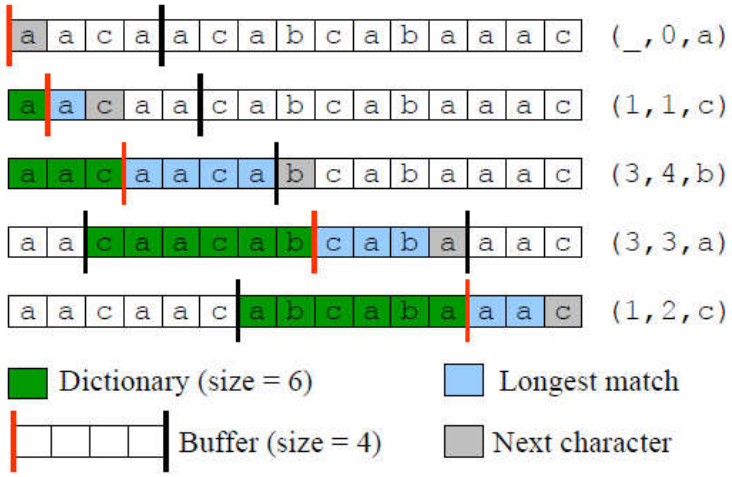
Token（p,l,c）编码的结果：

- p: 第一个数字指的是**SB**中开始匹配的位置，注意是**从Cursor往前倒着数，从1开始数**；
- l: 第二个数字指的是匹配成功的字符个数；
- c: 第三个指的是**LAB**中匹配结束的下一个字符；

注意：在匹配时是搜索字典中最长的匹配，且当前的LAB区域如果继续匹配也继续搜索（比如匹配到SB的最后一个字符后下一个到LAB的第一个字符仍然匹配则继续，直到不匹配或匹配数达到了LAB的长度限制则停止。如果在SB中一个都不匹配则不继续搜索，排除LAB第一个字符自身无尽循环匹配的情况）。

示例：

字符串“aacaacabcabaaac”的一个LZ77编码示例，其中缓冲长度分别为：LAB=4,SB=6



LZ77的变形

LZR：就是SB搜索缓冲的长度不固定了，算法输出的token位长度也是可变的。

LZH：算法的输出结果又进行了哈夫曼编压缩。

DEFLATE:当前最流行的基于LZ77的压缩算法，是很多通用的Unix压缩项目‘gzip’的一部分。

LZ78:

算法是将编码过程中之前编码过的所有字符作为了一个索引字典，之前的每一次编码是一个字典元素，之后的编码如果包含之前字典的元素则用该元素的索引代替实现压缩（注意是从之前的字典元素中找那个匹配最长的字典元素），同时记录不匹配的那个字符。可以想到，不断更新的字典中最长的字典元素很可能会越来越长且每次长一个字符。

示例： 对字符串“abacbabaccbabbaca“进行LZ78编码。

Step	Input	Token	New Dictionary Entry/Index
		Output = 0a0b1c2a4c0c4b5a	
1	a	0,a	a,1
2	b	0,b	b,2
3	ac	1,c	ac,3
4	ba	2,a	ba,4
5	bac	4,c	bac,5
6	c	0,c	c,6
7	bab	4,b	bab,7
8	Baca	5,a	Baca,8

字典和字典元素的索引

LZ78的变形算法

上面说到LZ78的最长的字典元素只会越来越长不受限制，那么就要使用变长的bit空间来保存字典索引，当前字典的需要保存索引的空间大小为$log2 (i)$ bits，i为目前字典中最长字典元素的长度。

LZC

算法给字典元素的长度设置一个最大值，如果匹配的结果超出最大值时就选择上一个相对较短的匹配的字典元素，防止字典元素变得太长。如果编码压缩率受限制变得太小，就清空之前的字典，比如重新开始压缩算法。

LZW

和LZ78不同的是，算法开始不是空的字典，一开始就把可能的所有单一字符作为最开始的字典，另外不是和LZ78那样记录字典索引和不匹配字符，而是只记录匹配的字典索引（不可能出现不匹配的情况，至少匹配一个字符了）。

示例： 对字符串“abacbabaccbabbaca”进行LZW编码：

Step	Input	Encoded Output	New Dictionary Entry/Index
1	a		
2	b	0 (a)	ab / 3
3	a	1 (b)	ba / 4
4	c	0 (a)	ac / 5
5	b	2 (c)	cb / 6
6	a		
7	b	4 (ba)	bab / 7
8	a		
9	c	4(ba)	bac / 8
10	c	2 (c)	cc / 9
11	b		
12	a	6 (cb)	cba / 10
13	b		
14	b	2 (ab)	abb / 11
15	a		
16	c		

Index	Dictionary Entry
0	a
1	b
2	c
3	ab
4	ba
5	ac
6	cb
7	bab
8	bac
9	cc
10	cba
11	abb
12	baca

算术编码

算数编码是考虑到解决哈夫曼编码的一个限制：对于信息的编码，要对每一个字符都要使用一个几个二进制的bit数区别表示，收到整体的影响，平均每个字符可能都要用不少的bit数空间来表示。
算术编码是将编码的消息表示成实数0和1之间的一个间隔，消息越长，编码表示它的间隔就越小，形成结合越来越紧密的编码，同时需要表示的二进制位数就越多，导致算数编码的最大问题就是计算机的精度问题，精度有限，正常情况下无法进行大量数据的编码，事实上只能编码很短的数据。后来有了其他的先进方法才使算术编码得到应用，具体参考算数编码文章链接。

推荐阅读：算术编码

文章标签： 压缩 多媒体 数据 哈夫曼算法 LZ77

个人分类： 图像处理与计算机视觉