**18-压缩算法**

**背景**

对于「压缩」这个概念，我们大家都不陌生，无论是我们日常使用的WinRAR，7Zip等压缩软件，还是我们日常访问的视频、图片、音频数据，无一离不开数据压缩算法。但是，它在计算机领域里又是一个比较低调的存在，很少被各种专业书籍提及。今天，我们就来一起探索一下无损数据压缩算法，并且在压缩编码的领域内，入个门。

**信息的传递与如何度量**

今天我们以文字信息作为讨论的主体，将信息的传递，寄托在文字之上。

**信息熵**

我们的讨论都是以二进制编码为主，现在，我们先以一个常见的游戏来理解二进制能够表示的信息数量。

在N久以前，微软小冰有一个读心术的游戏。 这个游戏是让玩家随便在心里想一个东西，小冰则通过提问来猜这个东西是什么。玩家针对每次提问只需要回答“是”或者“否”，最终通过15个问题（甚至不需要15个问题）小冰就会猜出来你心里想的是什么东西。这个游戏表明，我们通过15个问题将「这个东西是什么」这个信息，传递给了小冰，确认这个东西（set）的信息量就是2^15。由于每个问题只需要回答是或否（也就是二进制），换句话说15个问题可以覆盖32768（2^15) 个东西。

我们看到似乎很多东西能够通过无数个问题的是或否来无限接近，那么换句话说，任何信息似乎都可以通过二进制来表示。

下面我们就来考虑怎么去用计算机中的二进制表示信息，以及我们需要多大的规模去表示一个信息。首先我们思考一下任意一个十进制的数字，在计算机中我们需要多少位二进制去表示呢？

答案是：b = Log2(x) + 1 备注：假设X就是这个十进制数的话，b是表示这个十进制数所需要的最小位数。

|  |
| --- |
| 这个公式是怎么得来的？  正整数 X 的二进制表示中共有 b 位，因此 X 的最大值就是所有的位全部取1，即  1111 = 15(10)  MaxValue = 2^0 + 2^1 + ... + 2^(b-1) = 2^b - 1。  最小值就是第 b 位取1，其余位全部取0，此时MinValue = 2^(b-1)。  1000 = 8(10)  因此，我们有 2^(b-1) ≤ x < 2^b。对等式两边以2 为底取对数，有b - 1 ≤ ⌊log(2)x⌋ < b，即：b = ⌊log(2)x⌋ + 1。 |

|  |
| --- |
| 为什么我们需要这个公式，是因为我们今天讨论的主题叫做压缩。大家都知道一个十进制数在计算机中需要用8bit （ASCII编码）来表示，如果用某种特殊的方式能够将其二进制表示位数缩短到低于8bit，那么我们就可以说实现了压缩。 |

上面的公式让我们可以很好的去衡量任意的十进制数在计算机中表达的复杂度，我们接下来的想法都要这个LOG2的角度去理解后续要讲的数据压缩效率。但是上面的公式仅仅是表示一个数的，如果我们想要表示一个数据集呢。也许这题你会，如果有n个数，那表示这些数据所需要的二进制数不就是n \* （⌊log(2)x⌋ + 1 ）嘛。

这个公式看起来很简洁明了，但是这样对吗？或者说，这是所需要的最小二进制位数吗？

让我们先看看下列例子：

G = [A,B,B,C,C,C,D,D,D,D]

如果我们只考虑总共出现的元素种类那么我们可以得到

S = set(G) = [A,B,C,D]

显然光靠种类还不足以表示G这个集合，还需要描述这些元素各自出现了几次。

G = [1,A,2,B,3,C,4,D]

这样就能恰当的描述这个集合了，我们所运用的所有字符数从10个降低到了8个，我们可以说，我们实现了压缩。

那么到这里，我们就可以判断n \* （⌊log(2)x⌋ + 1 ）这个公式是不恰当的，因为我们并不需要针对每个字符都计算一遍它所需要的二进制位数，而只需要一个 4D 就可以表示，D，D，D，D。

除此之外，显然这个压缩算法是有问题的，举个例子，如果有这样一个集合

G = [A,B,C,D,E,F,G,H,I]

那么用我们的频次表示法来表示

G = [1,A,1,B,1,C,1,D,1,E,1,F,1,G,1,H,1,I]

显然所需要的字符数比原来还要多，那么这就不是一个通用的压缩算法，只在某些特定数据集下有用。

从上面的两个例子来说，我们似乎有两个直观的感受

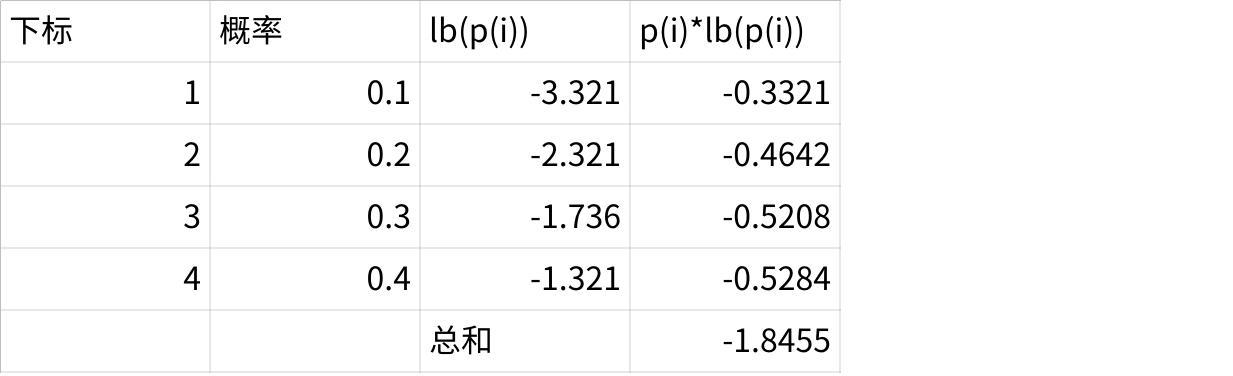
1. 数据越混乱，我们所需要描述它所用的信息量就越大
2. 似乎可以通过在数据出现的频率上面做文章，来实现数据的压缩。

**香农的公式**

《信息论》的作者香农博士，给出了描述信息熵的公式。

H(X)=−∑p(xi)lbp(xi)

我们来看一下G = [A,B,B,C,C,C,D,D,D,D] 数据集在这个公式上的具体体现。



**点击图片可查看完整电子表格**

最后一个求和取负号以后，我们能得到一个整数1.8455，这个就是表示这个数组每个符号平均需要1.8455个二进制。

让我们来回顾一下热力学熵的概念：一个热力学量，表示一个系统中无法转换为机械功的热能的量，也可以解释为系统的无序度或者随机度。我们直接搬来用吧，把这个东西叫做信息熵。用来表示一个一个数据集平均随机程度就行。越大表示越混乱，越随机，越需要更多的位数去表达。

结合我们上面的例子，总结就是表示某个数据集所需的二进制位数最少，集合中每个符号平均需要的最小二进制位数最少。这就是一个建立在每个符号出现概率上进行数据度量的指标。

**常见的压缩算法**

* [哈夫曼编码](https://www.baidu.com/s?wd=%E5%93%88%E5%A4%AB%E6%9B%BC%E7%BC%96%E7%A0%81&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=08a5AUx5Q7gnSub7W9yC9gUjxVjnPIcexWCu%2BGrN8DfqwXUDV9qkbLGeI08&sa=re_dqa_generate)：一种常用的无损压缩算法，通过构建最优前缀码来减少数据量。
* [LZ77算法](https://www.baidu.com/s?wd=LZ77%E7%AE%97%E6%B3%95&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=ec1c2T4HUtlhMaa13UzjBHY%2Be%2FOJSxolBBUFmlA3fXGJyvxbPVqzGt6ySbA&sa=re_dqa_generate)：基于滑动窗口的字典编码机制，适用于多种数据压缩场景。
* LZ78算法：通过编码连续字符序列来减少数据量。
* [LZW算法](https://www.baidu.com/s?wd=LZW%E7%AE%97%E6%B3%95&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=b2cdzf3ClZJ3okwtSCnxoaocQ89TKvetq4koJFhRrRLmmKp9OGfhiN0wI6I&sa=re_dqa_generate)：通过将字符串替换为更短的编码来减少数据量。
* [LZMA](https://www.baidu.com/s?wd=LZMA&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=b2cdzf3ClZJ3okwtSCnxoaocQ89TKvetq4koJFhRrRLmmKp9OGfhiN0wI6I&sa=re_dqa_generate)：‌[Deflate](https://www.baidu.com/s?wd=Deflate&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=b3c6Grh6MmRsWCA3utXTGxHW%2BOMsPKMxqv8h%2BwBxok4qc9VB9uOPLuo0hME&sa=re_dqa_generate)和LZ77算法的改良和优化版本，适用于高压缩率的需求。
* [LZO](https://www.baidu.com/s?wd=LZO&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=ff97dLm7ay%2Bp5k5DTccYHQKHA6tPepjOlWbTUHQq1wy1kEQjgUUVCRGG%2BM0&sa=re_dqa_generate)：专注于提高解压速度的无损压缩算法。
* Deflate：一种广泛使用的无损压缩算法，基于LZ77和哈夫曼编码。
* [GZIP](https://www.baidu.com/s?wd=GZIP&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=ff97dLm7ay%2Bp5k5DTccYHQKHA6tPepjOlWbTUHQq1wy1kEQjgUUVCRGG%2BM0&sa=re_dqa_generate)：基于Deflate算法的文件压缩格式，广泛用于网络传输和存储。
* [Snappy](https://www.baidu.com/s?wd=Snappy&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=3efdr9xK3P%2Bav7N14DmOZOmx9ecYrq8FDWTctag4GPDt%2BilHsLyDZLq9XfM&sa=re_dqa_generate)：快速的无损数据压缩算法，适用于大数据处理。
* [LZ4](https://www.baidu.com/s?wd=LZ4&usm=4&ie=utf-8&rsv_pq=a4fefa0e000b6020&oq=%E5%B8%B8%E8%A7%81%E5%8E%8B%E7%BC%A9%E7%AE%97%E6%B3%95&rsv_t=3efdr9xK3P%2Bav7N14DmOZOmx9ecYrq8FDWTctag4GPDt%2BilHsLyDZLq9XfM&sa=re_dqa_generate)：一种非常快速的压缩算法，适用于实时数据处理。

**压缩算法的特点和应用场景**

* 哈夫曼编码：适用于文本文件压缩，通过构建最优前缀码来减少数据量。
* LZ77算法：适用于多种数据压缩场景，包括文件传输和存储、多媒体数据压缩等。
* LZ8算法：通过编码连续字符序列来减少数据量，适用于文本和程序代码的压缩。
* LZW算法：通过将字符串替换为更短的编码来减少数据量，适用于字符串密集的数据。
* LZMA：适用于需要高压缩率的应用场景，如大型文件和数据库的压缩。
* LZO：适用于需要快速解压的应用场景，如实时数据处理和大型文件的快速解压。
* Deflate：适用于需要高效压缩的文件传输和存储场景，如网络传输和电子邮件附件。
* GZIP：适用于需要高效压缩的文件格式，如ZIP文件和网页内容压缩。
* Snappy：适用于大数据处理场景，如实时数据处理和大规模数据集的压缩。
* LZ4：适用于需要高速压缩的应用场景，如实时数据处理和大型文件的快速压缩。

**变长编码 VLC**

我们来看看，从这个符号出现次数，也就是频率上，能够做哪些文章来实现数据的压缩？

我们通过一个实验出发。假设一个实验集(1000长度)里面只有A和B两个元素，根据他们出现的概率来计算熵。



**点击图片可查看完整电子表格**

从上图不难看出当两个符号出现的概率一样的时候熵最大，冗余度最小。这样给了我们一些启示，如果我们给出现概率高的符号分配最小的位数，给出现概率低的符号分配更多的位数，这样是不是就能实现压缩。

有了上面的基础，我们就可以动手了，对数据进行VLC有3个步骤：

1. 遍历每个符号并计算概率
2. 根据概率分配码字，概率越大分配的码字越短
3. 再次遍历进行编码

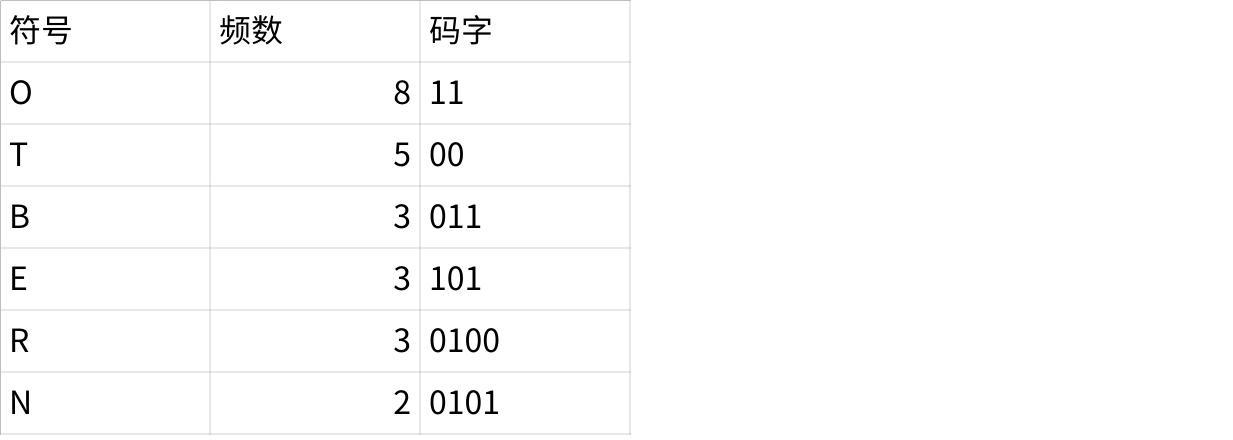
下面是一个例子：TOBEORNOTTOBEORTOBEORNOT

计算频率



**点击图片可查看完整电子表格**

分配码字 需要注意避免*前缀重复*就行，



**点击图片可查看完整电子表格**

下面我们对其中一个片段TOBEORNOT进行编码示意就能得到

0011011101110100010111000 一共 24个二进制位，对比每个字符8个位，总共72位。

然而，这种方法有很多缺陷，导致只是用来表示压缩数据流以外就没有其他应用。问题在于：

* 不按字节或者整型对齐
* 对于大数值，码字长度的增长速度一般会超过lb（n）个二进制位
* **解码速度很慢 （因为只能每次读一个二进制位）**

我们不难看出为了使用VLC，我们必须计算数据流中各个符号出现的概率，然后选择一个与数据模型概率分布接近的VLC方法。一旦选错了编码方法，压缩后的数据流可能还会更大。

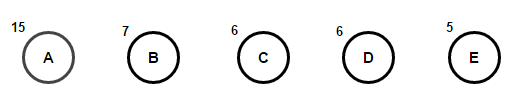
**统计编码**

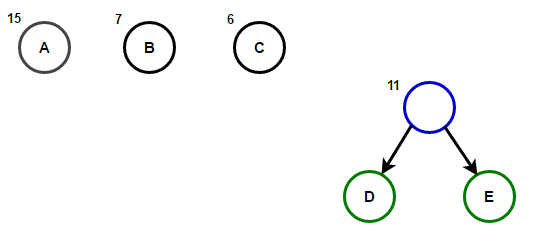
**Huffman Encode**

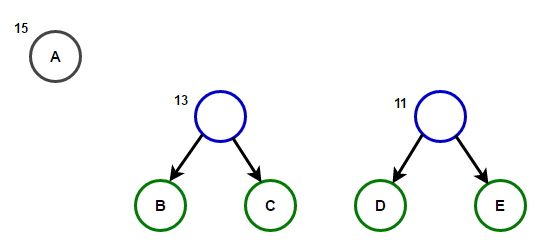
从上面的编码过程中我们可以发现，分配码字是一个很复杂的过程，而且码字选的不好，似乎会导致压缩失败。那么有没有一套成熟的理论、现成的方法可以供我们直接使用去分配码字呢？

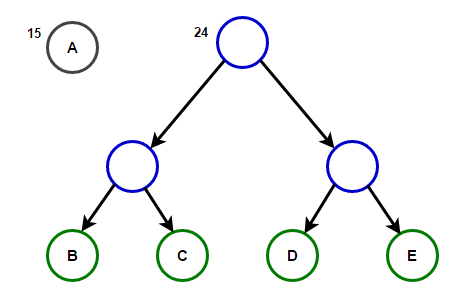
下面我们请出来Huffman编码。

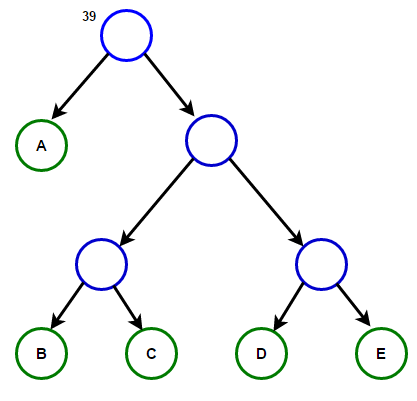
考虑一些仅由 'A', 'B', 'C', 'D'， 和 'E' 字符，它们的频率是 15, 7, 6, 6, 5。下图说明了算法所遵循的步骤：



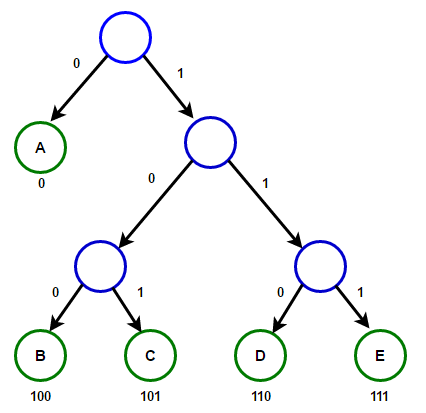








从根到任何叶节点的路径存储与该叶节点相关联的字符对应的最优前缀码(也称为霍夫曼码)。



但是在这个例子中，我们也看到，C和D的出现频次一样，那么这棵树构造出来的形式可能也就不一样。因此在Huffman解码的时候，就需要将Huffman打包到压缩后的文件中发送给解压方，才能够正常运行。

那么Huffman编码需要将字典打包到压缩文件中去的问题，有没有办法解决呢？也是有的，下面我们请出LZW编码。

**LZW编码**

LZW编码 (Encoding) 的用来压缩的核心思想比较简单，就是把出现过的字符串映射到记号上，这样就可能用较短的编码来表示长的字符串，实现压缩，例如对于字符串：

ABABAB

可以看到子串AB在后面重复出现了，这样我们可以用一个特殊记号表示AB，例如数字2，这样原来的字符串就可以表示为：

AB22

这里我们称2是字串AB的记号(Symbol)。那么A和B有没有记号来表示？当然有，例如我们规定数字0表示A，数字1表示B。实际上最后得到的压缩后的数据应该是一个记号流 (Symbol Stream) :

0122

这样我们就有一个记号和字符串的映射表，即字典 (Dictionary) ：



**点击图片可查看完整电子表格**

有了压缩后的编码0122，结合字典，就能够很轻松地解码 (Decoding) 原字符串ABABAB。刚才我们不是说，LZW解决了Huffman编码必须将字典文件编码到压缩文件中去的弊端吗？怎么还是需要结合字典才能解码呢？别着急，在上面的例子中，我们其实只需要将A和B各自的初始编码记录下来，这样就可以自行编码出AB的码字。而在LZW初始会有一个默认的字典，包含了所有256个8bit字符，单个字符的记号就是它自身，用数字表示就是ASCII值。在此基础上，编码过程中加入的新的记号的映射，从256开始，称为扩展表(Extended Dictionary)。在这个例子里是为了简单起见，只有两个基础字符，所以规定0表示A，1表示B，从记号2开始就是扩展项了。

**字典的生成**

这里有一个问题：为什么第一个AB不也用2表示？即表示为222，这样不又节省了一个记号？这个问题实际上引出的是LZW的一个核心思想，即压缩后的编码是**自解释** (self-explaining) 的。什么意思？即字典是**不会被写进压缩文件的**，在解压缩的时候，一开始字典里除了默认的0->A和1->B之外并没有其它映射，2->AB是在解压缩的过程中一边加入的。这就要求压缩后的数据自己能告诉解码器，完整的字典，例如2->AB是如何生成的，在解码的过程中**还原**出编码时用的字典。

用上面的例子来说明，我们可以想象ABABAB编码的过程：

1. 遇到A，用0表示，编码为0。
2. 遇到B，用1表示，编码为1。
3. 发现了一个子串AB，添加映射2->AB到字典里。
4. 后面又出现了AB子串，都用2来编码。

**LZW算法实例**

下面给出一个稍微复杂一点的例子，来说明LZW的原理，重点是理解解码中的每一步是如何对应和还原编码中的步骤，并恢复编码字典的。

**编码过程**

编码器从原字符串不断地读入新的字符，并试图将单个字符或字符串编码为记号 (Symbol)。这里我们维护两个变量，一个是**P** (Previous)，表示手头已有的，还没有被编码的**字符串**，一个是**C** (current)，表示当前新读进来的**字符**。

这里用一个例子来说明编码的过程，之所以用小写的字符串是为了和P，C区分。

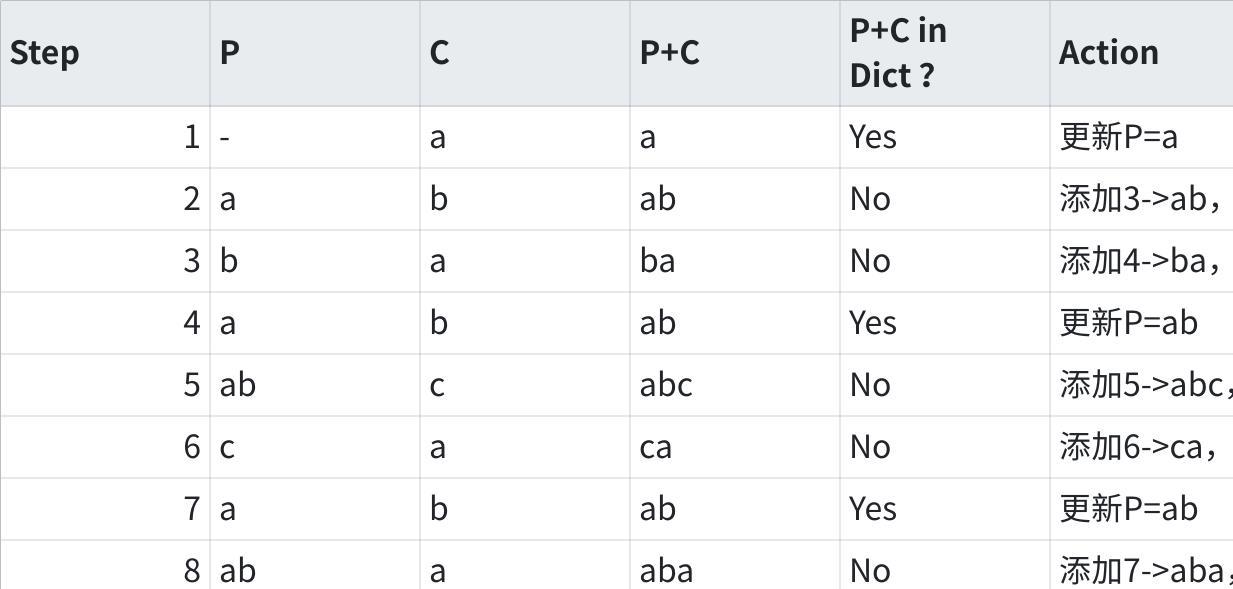
ababcababac

初始状态字典里有三个默认的映射：



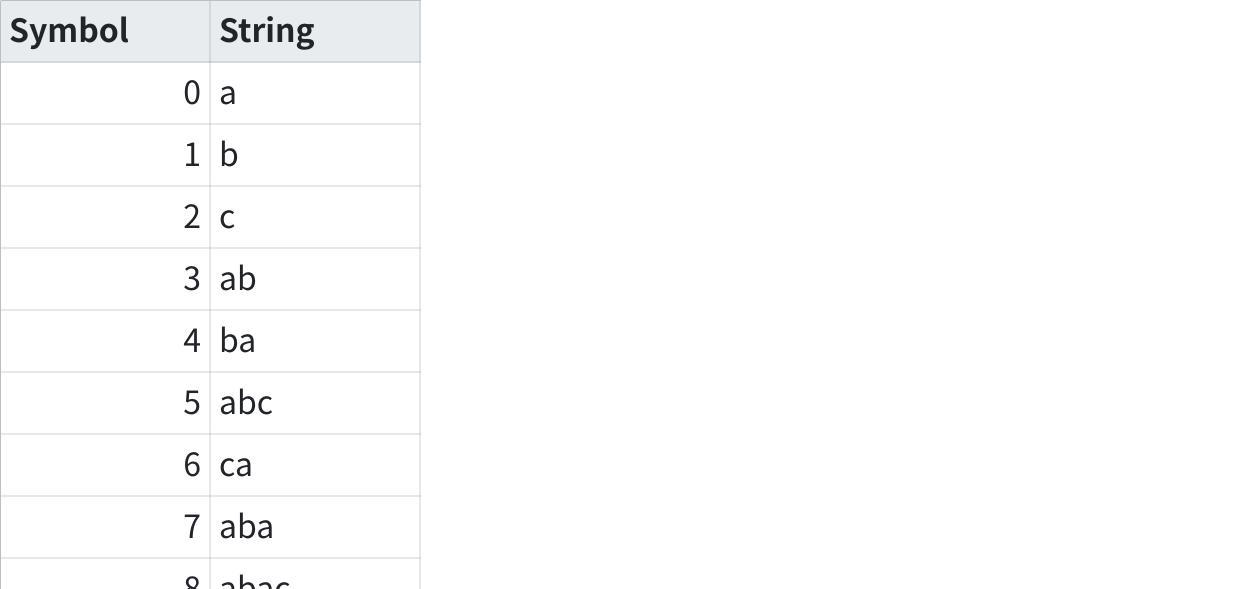
**点击图片可查看完整电子表格**

开始编码：



**点击图片可查看完整电子表格**

输出的结果为0132372，完整的字典为：



**点击图片可查看完整电子表格**

这里用一个图来展示原字符串是如何对应到压缩后的编码的:



**解码算法**

解码的过程比编码复杂，其核心思想在于解码需要还原出编码时的用的字典。因此要理解解码的原理，必须分析它是如何对应编码的过程的。下面首先给出算法：

解码器的输入是压缩后的数据，即记号流 (Symbol Stream)。类似于编码，我们仍然维护两个变量**pW** (previous word) 和**cW** (current word)，后缀W的含义是word，实际上就是记号 (Symbol)，一个记号就代表一个word，或者说子串。**pW**表示之前刚刚解码的记号；**cW**表示当前新读进来的记号。

注意cW和pW都是记号，我们用Str(cW)和Str(pW)表示它们解码出来的原字符串。

我们还是结合之前的例子来说明，我们需要从记号流

|  |
| --- |
| Plaintext 0 1 3 2 3 7 2 |

解码出：

|  |
| --- |
| Plaintext a b ab c ab aba c |

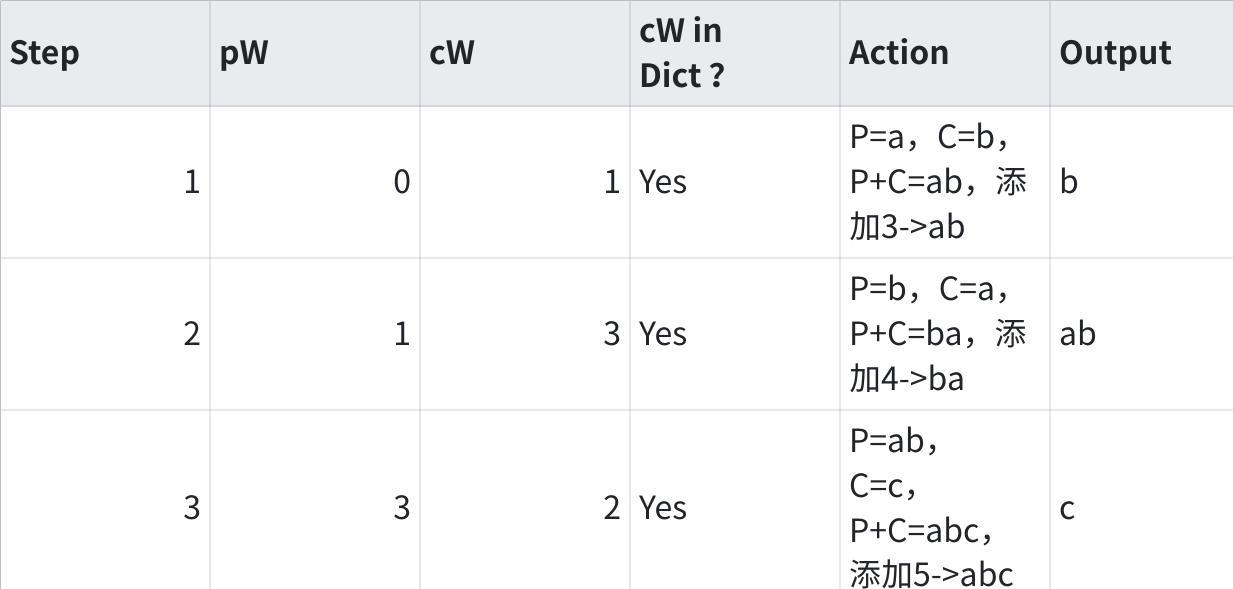
这里我用空格表示出了记号是如何依次对应解码出来的子串的，当然在解码开始时我们根本不知道这些，我们手里的字典只有默认项，即：



**点击图片可查看完整电子表格**

解码开始：

0 1 3 2 3 7 2  
首先读取第一个记号cW=0，解码为a，输出，赋值pW=cW=0。然后开始循环，依此读取后面的记号：



**点击图片可查看完整电子表格**

好，先解码到这里，我们已经解出了前5个字符 a b ab c。一步一步走下来我们可以看出解码的思想。首先直接解码最前面的a和b，然后生成了3->ab这一映射，也就是说解码器利用前面已经解出的字符，如实还原了编码过程中字典的生成。这也是为什么第一个a和b必须保留下来，而不能直接用3来编码，因为解码器一开始根本不知道3表示ab。而第二个以及以后的ab就可以用记号3破译出来，因为此时我们已经建立了3->ab的关系

**snappy压缩原理**

Snappy是一种高效的数据压缩算法，由Google开发。Snappy压缩算法的核心思想是利用哈希表来寻找重复的字节序列，并通过变长整数对这些序列进行编码。当数据中存在大量重复数据时，Snappy压缩效果尤为显著。

**LZ4压缩算法的基本概念和特点**

LZ4是一种基于LZ77算法的无损数据压缩算法，由[Yann Collet](https://www.baidu.com/s?wd=Yann%20Collet&usm=1&ie=utf-8&rsv_pq=ad2f6bf2000df853&oq=lz4%E5%8E%8B%E7%BC%A9%E5%8E%9F%E7%90%86&rsv_t=eb65ORksfElApMtExnpeACNfqAYZxyaNhpORCfE1JHPRJlX8Vr67lMc7YjU&sa=re_dqa_generate)在2011年设计实现。它侧重于压缩和解压缩的速度，而不是压缩比。LZ4的压缩和解压缩速度非常快，每核压缩速度大于500MB/s，多核CPU可叠加；解压缩速度每核可达GB/s量级。

LZ4压缩算法的工作原理

LZ4算法基于[字典](https://www.baidu.com/s?wd=%E5%AD%97%E5%85%B8&usm=1&ie=utf-8&rsv_pq=ad2f6bf2000df853&oq=lz4%E5%8E%8B%E7%BC%A9%E5%8E%9F%E7%90%86&rsv_t=40b1RLDIyg3ae10TDP68mGVkTq5Rdctn3PbxilxR0NGPUl54de2%2BNbX7Nvs&sa=re_dqa_generate)的压缩方法，通过查找数据中的重复字符串并用较小的标记来代替这些重复字符串，从而实现数据压缩。它使用滑动窗口和前向缓冲区来维护一个短语字典。滑动窗口在数据中滑动时，将数据载入前向缓冲区形成一批短语，这些短语成为字典的一部分。通过这种方式，LZ4能够快速找到并替换数据中的重复字符串。

LZ4压缩算法的实现细节

LZ4实现了两种格式：lz4\_block\_format和lz4\_frame\_format。lz4\_block\_format用于一般场景，而lz4\_frame\_format用于特殊场景如文件压缩、管道压缩和流式压缩。一个完整的lz4压缩块由多个序列组成，每个序列以token开始，包含字面量和匹配副本的长度和偏移量。

LZ4压缩算法与其他压缩算法的比较

与其他压缩算法相比，如gzip或zstd，LZ4的压缩比并不突出。然而，它的压缩和解压缩速度非常快，非常适合于需要快速处理大量数据的场景。相比之下，LZ4HC（高压缩率版本）提供了更高的压缩率但牺牲了速度。

**Zstd压缩算法**

它是一种快速的无损数据压缩算法，其实现逻辑大致如下：

1. 字典的构建‌：Zstd在压缩前会建立一个字典，用于存储之前的数据块。这个字典可以是静态的（预先构建好的）或者动态的（通过动态建模构建），并且可以根据压缩的数据类型进行自适应调整。
2. 分析数据‌：Zstd会对输入的数据进行分析，寻找其中的重复模式，并将其替换为一些较短的指针，指向之前已经压缩过的重复数据。
3. 构建[Huffman树](https://www.baidu.com/s?sa=re_dqa_generate&wd=Huffman%E6%A0%91&rsv_pq=bde55b050007483f&oq=zstd%E5%8E%8B%E7%BC%A9%E5%8E%9F%E7%90%86&rsv_t=e8c5kpP7FW2rdIouQp0+eP6XH8vyHzIFvqUd/NVcjevrW+oTalF/wdvcqtk&tn=baidu&ie=utf-8)‌：Zstd使用了一种叫做FSE（[Finite State Entropy](https://www.baidu.com/s?sa=re_dqa_generate&wd=Finite%20State%20Entropy&rsv_pq=bde55b050007483f&oq=zstd%E5%8E%8B%E7%BC%A9%E5%8E%9F%E7%90%86&rsv_t=e8c5kpP7FW2rdIouQp0+eP6XH8vyHzIFvqUd/NVcjevrW+oTalF/wdvcqtk&tn=baidu&ie=utf-8)）的算法对压缩后的数据进行编码。这种算法通过构建Huffman树来实现，使得高频词的编码长度短，低频词的编码长度长，从而达到更高的压缩率。
4. 压缩‌：根据分析和编码结果，Zstd将原始数据压缩成一段连续的二进制数据。Zstd可以使用多种压缩级别，不同级别对应不同的压缩速度和压缩比。
5. 解压‌：压缩后的数据可以通过相同的算法进行解压。解压的过程中需要使用之前压缩时建立的字典和Huffman树等数据结构。

总之，Zstd的实现逻辑是通过学习分析数据的重复模式，使用FSE算法和Huffman树对压缩后的数据进行编码，从而达到高压缩比和快速解压的目的1。