

数据压缩实验报告

一、实验目的

本次实验旨在使用霍夫曼编码 (Huffman Code)、算术编码 (Arithmetic Code) 和 LZW 编码 (Lempel-Ziv Code) 三种无损压缩算法对英文文献进行压缩，对比各算法的编码效率、压缩比及编解码耗时，深入理解不同压缩算法的原理与应用场景。

二、算法原理

2.1 霍夫曼编码 (Huffman Code)

核心思想：基于贪心算法，为高频字符分配短码字，低频字符分配长码字，实现最优前缀编码。

步骤：

1. 统计字符频率，构建优先队列（最小堆）。

$$Q = \{(p_1, c_1), \dots, (p_n, c_n)\}$$

2. 迭代合并频率最小的两个节点，生成霍夫曼树。

$$p_{new} = p_i + p_j$$

3. 从根节点遍历树，左分支为 0，右分支为 1，生成字符编码，生成编码时满足 Kraft 不等式：

$$\sum_{i=1}^n 2^{-l_i} \leq 1$$

4. 将二进制码流填充为字节流存储。

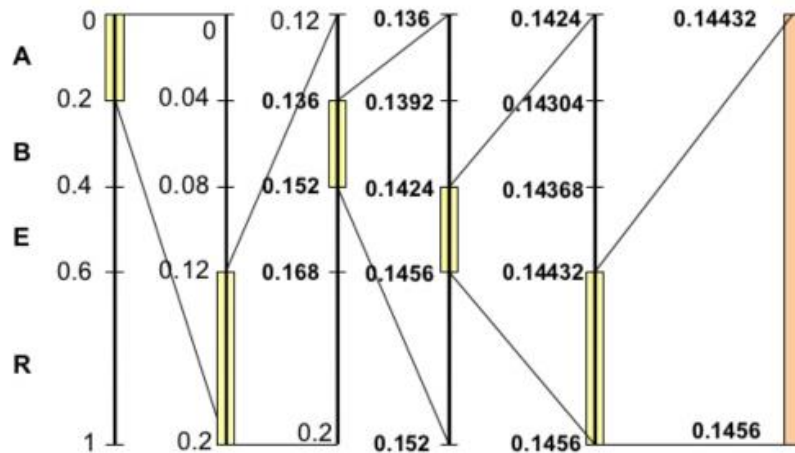
2.2 算术编码 (Arithmetic Code)

核心思想：将整个字符串映射为 $[0, 1)$ 区间内的小数，通过不断分割区间表示字符序列的概率分布。

步骤：

1. 统计字符频率，计算每个字符的概率区间。
2. 遍历字符，不断用当前字符的概率区间缩小全局区间，最终用区间

中点表示压缩结果。



初始区间 $[low_0, high_0) = [0, 1)$ 每次迭代更新区间：

$$\begin{cases} low_{k+1} = low_k + (high_k - low_k) \cdot F(c_{k+1}) \\ high_{k+1} = low_k + (high_k - low_k) \cdot F(c_{k+1} | c_{1:k}) \end{cases}$$

其中 $F(\cdot)$ 为累积分布函数

2.3 LZW 编码 (Lempel-Ziv-Welch Code)

核心思想：基于字典的无损压缩算法，将重复出现的字符串映射为字典索引，减少数据冗余。

步骤：

1. 初始化字典 (ASCII 字符集，大小 256)。

$$\mathcal{D}_0 = \{\text{ASCII}(0) \sim \text{ASCII}(255)\}$$

2. 遍历字符串，维护当前匹配序列，若当前序列 + 新字符不在字典中，则输出当前序列的索引，并将新序列加入字典。新短语添加规则：

$$\mathcal{D}_{k+1} = \mathcal{D}_k \cup \{w + c\}$$

其中 w 为当前匹配短语， c 为新字符 编码过程：每次输出索引满足：

$$\text{Index} = \underset{i}{\operatorname{argmax}} \{ \text{len}(\mathcal{D}_i[w]) | w < S \}$$

3. 压缩结果为字典索引的二进制流。

三、实验环境与数据准备

- 环境：

- Python 3.8
- 开发环境: vscode
- 依赖库: decimal (高精度计算)、collections.defaultdict (频率统计)、heapq (霍夫曼树构建)、json (元数据存储)
- **输入数据:** 英文文献《A public key infrastructure (PKI) ...》(内容见附件), 保存为 pki_text.txt, UTF-8 编码, 共 **6506 字节** (含终止符)。

四、实验步骤

4.1 霍夫曼编码实现流程

1. **统计频率:** 读取文本, 计算每个字符的出现次数。

```
freq = defaultdict(int)
for byte in text_bytes:
    freq[byte] += 1 # 统计每个字节出现的次数
```

2. **构建霍夫曼树:** 使用优先队列合并节点, 生成编码树。

节点依次入队

```
# 构建优先队列 (最小堆)
heap = []
for byte, count in freq.items():
    # 将每个字节转换为叶子节点加入堆
    heapq.heappush(heap, HuffmanNode(count, byte=byte))
```

合并最小两个节点

```
while len(heap) > 1:
    # 取出频率最小的两个节点
    left = heapq.heappop(heap)
    right = heapq.heappop(heap)
    # 合并为新的内部节点 (频率为两者之和)
    merged = HuffmanNode(left.freq + right.freq, left=left, right=right)
    heapq.heappush(heap, merged)
```

3. **生成编码表:** 遍历树生成字符到二进制码的映射。

递归遍历哈夫曼树，左分支编码为 0 右分支编码为 1

```
def build_code(node, current_code=""):
    if node is None:
        return
    if node.byte is not None:
        # 叶子节点: 记录字节对应的编码
        code_table[node.byte] = current_code
        return
    # 左子树编码加0, 右子树编码加1
    build_code(node.left, current_code + '0')
    build_code(node.right, current_code + '1')
```

4. **压缩**: 将文本转换为二进制码流, 填充为字节流, 保存为 compressed_huffman.bin, 并存储编码表为 huffman_codes.json。

4.2 算术编码实现流程

1. **统计频率**: 包含终止符\x00, 确保完整压缩。
2. **生成概率区间**: 将字符按频率排序, 分配连续概率区间 (最后一个字符包含 1.0)。

```
# 生成累积概率区间
for byte in sorted(freq.keys()): # 按字节值排序
    prob = Decimal(freq[byte]) / Decimal(total) # 计算概率
    cum_prob[byte] = (current, current + prob) # 记录区间
    current += prob # 累加概率
```

算术编码迭代公式实现

```
# 逐个字节更新区间
for byte in text_bytes:
    char_low, char_high = cum_prob[byte] # 获取当前字节的概率区间
    range_size = high - low # 当前区间长度
    # 缩小区间范围
    high = low + range_size * char_high
    low = low + range_size * char_low
```

3. **压缩**: 通过区间分割计算最终小数, 保存元数据 (字符列表、频率、结果小数)

4.3 LZW 编码实现流程

1. 初始化字典：包含 ASCII 字符（0-255）。

```
# 初始化字典：单字节到索引的映射（0-255）
dictionary = {bytes([i]): i for i in range(256)}
next_code = 256 # 下一个可用索引
s = bytes()      # 当前匹配字符串
encoded = []      # 编码结果列表
```

2. 压缩：遍历文本，动态扩展字典，记录索引序列，转换为 12 位二进制流，填充为字节流保存为 compressed_lzw.bin。

```
for byte in text_bytes:
    sc = s + bytes([byte]) # 尝试扩展当前字符串
    if sc in dictionary:
        s = sc # 存在则继续扩展
    else:
        # 输出当前字符串的索引
        encoded.append(dictionary[s])
        # 将新字符串加入字典
        dictionary[sc] = next_code
        next_code += 1
        s = bytes([byte]) # 重置当前字符串为当前字节
# 处理最后一个字符串
if s:
    encoded.append(dictionary[s])
```

五、实验结果

霍夫曼编码：
原始文件大小：6470 字节
压缩后大小：3593 字节
压缩比：55.53%
耗时：1.98ms

算术编码：
原始文件大小：6470 字节
压缩后大小：128 字节
压缩比：1.98%
耗时：36.27ms

LZW编码：
原始文件大小：6470 字节
压缩后大小：3548 字节
压缩比：54.84%
耗时：2.98ms

霍夫曼编码：
原始文件大小：6470 字节
压缩后大小：3593 字节
压缩比：55.53%
耗时：2.50ms

算术编码：
原始文件大小：6470 字节
压缩后大小：128 字节
压缩比：1.98%
耗时：36.13ms

LZW编码：
原始文件大小：6470 字节
压缩后大小：3548 字节
压缩比：54.84%
耗时：4.17ms

霍夫曼编码：
原始文件大小：6470 字节
压缩后大小：3593 字节
压缩比：55.53%
耗时：2.13ms

算术编码：
原始文件大小：6470 字节
压缩后大小：128 字节
压缩比：1.98%
耗时：38.19ms

LZW编码：
原始文件大小：6470 字节
压缩后大小：3548 字节
压缩比：54.84%
耗时：4.26ms

5.1 压缩效率对比（平均）

算法	原文件大小	压缩后大小	压缩比	耗时（ms）
霍夫曼编码	6470 B	3593 B	55.53%	2.20
算术编码	6470 B	128 B	1.98%	37.01
LZW 编码	6470 B	3548 B	54.84%	3.80

5.2 编码结果

1. 霍夫曼编码部分结果（huffman_compressed.bin）展示：

address	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Ascii	<input type="checkbox"/> unsigned
00000000	0d	a0	50	40	be	6a	83	0e	b1	e9	a5	fd	95	cf	53	1c	..P@.j.....S	
00000010	1c	d7	d2	d7	07	d9	d5	04	08	de	fb	d6	74	6d	fd	actm..	
00000020	25	7d	a0	d3	cc	68	27	27	ed	07	48	c7	50	2d	7d	5f	%}...h'...'H.P-}_	
00000030	27	ed	07	27	5a	05	04	0b	e6	a8	30	be	53	98	ee	91	'..'Z.....0.S...	
00000040	b4	0a	08	17	cd	50	61	c4	78	d0	c1	f6	83	4f	48	c7	...Pa.x....OH.	
00000050	11	90	80	5f	02	c6	a2	c7	b6	4e	b5	c4	8f	df	d9	7d	...N.....}	
00000060	46	fd	7a	05	04	0b	e6	a8	30	ee	16	fb	ea	f9	3e	5f	F.Z.....0...._	
00000070	bf	06	d5	f4	b5	d8	8c	84	02	f9	4e	59	f4	8c	73	c2NY..s.	
00000080	fe	6b	3e	f4	ce	78	a6	20	87	1a	93	8b	91	ec	74	67	.k>..x.tg	
00000090	e7	25	02	db	d6	75	8f	96	13	fa	46	3a	81	6b	ea	1d	..%...u....F:.k.	
000000a0	eb	38	07	b5	fd	d6	9e	43	93	ad	c2	df	7d	5f	27	c8	.8.....C....}_'	
000000b0	79	68	08	d8	f6	c4	7e	fe	cb	d3	ce	ae	2f	4b	ea	24	yh....~...../K.\$	
000000c0	27	2d	40	b3	d3	c9	eb	2d	a5	5e	b2	3f	7c	9f	31	ce	'-@.....-.^.? .1.	
000000d0	5a	81	67	d2	31	d6	31	ba	17	15	48	1f	bf	55	cb	3a	Z.g.1.1.H.U.:	
000000e0	be	96	bb	11	90	80	5f	29	cb	3e	f4	d2	af	59	1f	be_).>...Y.	
000000f0	4f	9d	1b	ed	7f	20	dc	2d	f7	d5	f2	7c	e5	a0	22	59	O.. .-... ..Y	
00000100	e9	e9	7d	44	80	1a	5f	7a	6a	c6	37	3e	77	ac	e8	19	..}D.._zj.7>w..	
00000110	38	5d	f2	75	b8	5b	ef	ab	e4	f9	cc	68	27	27	ed	06	8].u.[.....h'..'	
00000120	9e	53	d8	f6	dc	68	60	f4	ec	d3	02	ef	9a	41	d9	1a	.S...h`.....A.	
00000130	fd	75	9f	7a	67	3c	53	1d	eb	3a	06	4e	17	7e	fe	ee	.u.zg<S...N~..	
00000140	ea	fa	5a	ed	c2	df	7d	5f	27	cb	eb	1e	01	51	3a	74	..Z...}_'...Q:t	
00000150	0a	08	17	cd	50	61	ca	73	1d	d2	36	23	c6	86	0f	b4	...Pa.s.6#....	
00000160	1d	23	1d	c6	86	0f	4e	cd	30	2e	f9	a5	5e	b2	3f	7c	#...N.0...^.?	
00000170	9f	b4	1c	9d	68	cf	ce	72	3f	bd	32	36	e9	92	b6	f5	..h..r?.26....	
00000180	9c	47	ef	e1	de	b9	fd	46	5f	5b	ba	8c	77	ac	ee	16	.G.....F_[...w..	

验证：

第一行为：0d a0 50 40 be 6a 83 0e b1 e9 a5 fd 95 cf 53 1c

化为二进制：00001101 10100000 01010000 01000000 10111110 01101010
10000011 00001110 10110001 11101001 10100101 11111101 10010101
11001111 01010011 00011100

根据编码规则（见附录）：划分为 0000110 110 100000 01010 000010 00000
1011 11100 110 10101000 001 100001 110 1011 0001 111010 0110 1001
0111 1111 01010 0110 11100

映射回字符为：A public key infrastruc

可依发现与给定文本的前几个字符是相同的，说明 huffman 编码算法无误

2. 算术编码结果展示：

arithmetic_compressed.bin:

address	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Ascii	<input type="checkbox"/> unsigned	<input type="checkbox"/> bigendian
00000000	2e	49	5c	da	b7	26	2a	28	c2	6b	52	06	ba	77	a2	55	.I\..&*(.kR..w.U		
00000010	b7	14	0b	72	f1	22	a6	cb	4d	b7	a7	c5	9a	c3	c3	6c	.r."..M.....l		
00000020	4b	ff	e4	ed	ec	63	5a	75	df	56	0e	b2	38	dd	0a	59	K....cZu.V..8..Y		
00000030	45	d2	ad	f1	66	51	7c	5c	f7	dd	86	b3	d2	a1	54	fd	E...fQ \.....T.		
00000040	d6	16	0a	7a	84	df	ba	62	30	0c	df	a7	e6	af	5e	8e	.z...b0.....^.		
00000050	37	7b	df	f4	4e	5a	ae	5a	9e	99	43	4e	7f	5f	bf	d5	7{..NZ.Z..CN _..		
00000060	eb	2b	b4	e9	b1	6f	b2	12	4a	cf	f1	21	8f	20	1c	a9	+....o..J..!. .		
00000070	cb	b1	45	cb	2e	be	6f	98	6e	74	43	c7	48	aa	b0	25	..E...o.ntC.H..%		

最终压缩区间：[0.18080692615253526889, 0.18080692615253526889)

部分字符概率区间展示：

=== 字符概率区间表 ===

字节 10（字符： ）： 区间 = [0.0000000000, 0.0055641422)
字节 32（字符： ）： 区间 = [0.0055641422, 0.1567233385)
字节 34（字符： "）： 区间 = [0.1567233385, 0.1576506955)
字节 39（字符： '）： 区间 = [0.1576506955, 0.1582689335)
字节 40（字符： （）： 区间 = [0.1582689335, 0.1590417311)
字节 41（字符： ））： 区间 = [0.1590417311, 0.1598145286)
字节 44（字符： ，）： 区间 = [0.1598145286, 0.1676970634)
字节 45（字符： -）： 区间 = [0.1676970634, 0.1692426584)
字节 46（字符： .）： 区间 = [0.1692426584, 0.1752704791)
字节 47（字符： /）： 区间 = [0.1752704791, 0.1755795981)
字节 48（字符： 0）： 区间 = [0.1755795981, 0.1768160742)
字节 49（字符： 1）： 区间 = [0.1768160742, 0.1772797527)
字节 50（字符： 2）： 区间 = [0.1772797527, 0.1780525502)
字节 53（字符： 5）： 区间 = [0.1780525502, 0.1788253478)

经过反向推导，在精度允许的范围内验证成功

3. LZW 编码结果展示:

Lzw_compressed.bin:

address	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	Ascii	<input type="checkbox"/> unsigned
00000000	04	10	20	07	00	75	06	20	6c	06	90	63	02	00	6b	06	.. .u. l..c...k.	
00000010	50	79	02	00	69	06	e0	66	07	20	61	07	30	74	07	20	Py...i...f. a.0t.	
00000020	75	06	30	74	07	50	72	06	50	20	02	80	50	04	b0	49	u.0t.Pr.P ..P...I	
00000030	02	90	20	07	30	75	07	00	70	06	f0	72	07	40	73	02	.. .0u...p...r.@s.	
00000040	00	74	06	81	1a	06	40	69	11	20	72	06	90	62	07	50	.t...@i. r..b.P	
00000050	74	06	90	6f	06	e0	2c	02	01	19	07	60	6f	06	30	61	t...o...`o.0a	
00000060	13	51	37	02	00	61	06	e0	64	02	00	76	06	51	31	06	.Q7...a...d...v.Q1.	
00000070	61	06	13	f1	36	06	e0	20	06	f0	66	10	11	03	10	51	a...6... .f...Q	
00000080	07	10	90	79	12	90	75	07	30	65	14	50	66	12	61	52	...y...u.0ePf.aR	
00000090	10	41	06	10	81	0a	02	00	65	06	e0	63	07	20	79	07	.A.....e...c. y.	
000000a0	01	40	13	81	42	14	41	64	06	e0	61	10	40	65	12	91	..@..BAd...a.@e..	
000000b0	05	06	e0	6b	10	d0	67	14	f1	51	06	90	64	16	51	35	...k...g.Q...dQ5	
000000c0	13	51	73	02	00	77	06	91	2b	15	f1	54	16	21	0b	06	.Qs...w...+T!..	
000000d0	31	48	13	51	4a	13	e0	74	17	30	2e	02	01	00	11	d0	1H.QJ...t0.....	
000000e0	49	16	f1	71	06	c1	81	15	91	48	12	91	43	14	50	73	I.q...H..C0Ps	
000000f0	15	71	8f	06	d1	29	07	40	6f	12	10	65	06	31	18	06	q...).@o...e.1.	
00000100	50	6c	10	b0	65	07	80	63	06	81	43	06	71	2d	13	f0	Pl...e...c...C.q-..	
00000110	61	14	f1	47	07	21	2a	12	c1	0c	06	e1	8f	07	20	6e	a.G.!*..... n	
00000120	06	50	74	16	d1	45	1b	70	69	06	61	0b	12	b1	1a	19	.Pt.E0pi.a...0	
00000130	80	67	18	40	69	06	d0	61	06	31	0b	15	00	20	18	a1	.g@i...a.1.0.0.	
00000140	27	1c	51	4b	18	f0	2d	06	80	6f	06	c1	2e	06	e1	79	'0QK.-...o....y	
00000150	17	e1	cc	17	31	39	12	21	af	14	21	29	07	70	65	06	0...19...!0!).pe.	
00000160	21	5a	07	21	b7	07	31	39	06	f1	c8	1b	80	61	13	41	!Z...!19...0.a.A	
00000170	2c	1b	c1	d7	15	b1	a6	1e	c1	9b	1c	21	bb	12	e0	76	,0...0...0!...v	

验证:

第一行为: 04 10 20 07 00 75 06 20 6c 06 90 63 02 00 6b 06 5

转为二进制为: 00000100 00010000 00100000 00000111 00000000 01110101
00000110 00100000 01101100 00000110 10010000 01100011 00000010
00000000 01101011 00000110 0101

12 位为一组划分后转为十进制: 65 32 112 117 98 108 105 99 32
107 101

十进制就是对应字符的 ASCII 码: A public key (11 个字符)

可以看到验证正确

部分编码序列:

=== 编码序列 ===

65	32	112	117	98	108	105	99	32	107	101	121
114	117	99	116	117	114	101	32	40	80	75	73
116	115	32	116	104	282	100	105	274	114	105	98
118	111	99	97	309	311	32	97	110	100	32	118
111	102	257	259	261	263	265	121	297	117	115	101
32	101	110	99	114	121	112	320	312	322	324	356
269	103	335	337	105	100	357	309	309	371	32	119
309	330	318	116	371	46	32	256	285	73	367	369
343	399	109	297	116	111	289	101	99	280	101	108
319	97	335	327	114	298	300	268	110	399	114	110
299	282	408	103	388	105	109	97	99	267	336	32
108	302	110	377	382	460	371	313	290	431	322	297
111	456	440	97	308	300	444	471	347	422	492	411
115	401	84	442	404	406	370	344	491	419	421	499
467	395	470	530	32	474	476	411	313	273	386	426

编码序列第一行为：65 32 112 117 98 108 105 99 32 107
101 121 (12 个)

对应的字符为：A public key i (12 个)

验证成功

六、结果分析

6.1 压缩比对比

- **算术编码**压缩比最高（1.98%），因直接利用字符概率连续分割区间，避免了霍夫曼编码的“码字对齐”开销。
- **LZW 编码**压缩比次之（54.84%），因初始字典较小（仅 256 项），且英文文本中长重复序列较少，字典扩展效率有限。
- **霍夫曼编码**最低（55.53%），作为最优前缀编码，效率依赖字符频率分布（英文文本中高频字符如空格、字母 e 等提升压缩效果）。

6.2 耗时对比

- **霍夫曼编码**最快，构建优先队列和遍历树的时间复杂度为 $O(n \log n)$ 。

- LZW 编码次之，字典操作均为线性时间，编码逻辑简单。
- 算术编码最慢，高精度小数运算（如 Decimal 的乘法和区间分割）消耗大量计算资源。

6.3 优缺点总结




算法	优点	缺点	适用场景
霍夫曼编码	实现简单，稳定高效	需预统计频率，不适合实时压缩	文本、图像（JPEG）等
算术编码	压缩比高，无需存储编码表	实现复杂，浮点精度要求高	高压压缩比场景（如 PDF）
LZW 编码	无需预统计，适合实时压缩	压缩比依赖数据重复性	二进制文件、重复数据

七、结论




本次实验成功实现了三种无损压缩算法，验证了其在英文文本压缩中的有效性。算术编码在压缩比上表现最优，霍夫曼编码在通用性和实现复杂度间平衡，LZW 编码适合实时场景。实际应用中需根据数据特性（如频率分布、重复性）和性能需求选择算法。并且通过本次实验，我不仅掌握了无损压缩的核心算法原理，还知道了合理选择压缩算法的重要性，对基础的数据压缩思想有了较为深刻的感悟。

八、附录：源代码与压缩结果

8.1 压缩结果文件

- 霍夫曼编码: `compressed_huffman.bin`  `huffman_compressed.bin`
- 算术编码: `compressed.bin`  `arithmetic_compressed.bin`
- LZW 编码: `compressed_lzw.bin`  `lzw_compressed.bin`

8.2 编码中间结果

- 霍夫曼编码: `compressed_info.txt`  `huffman_info.txt`
- 算术编码: `compressed_info.txt`  `arithmetic_info.txt`
- LZW 编码: `compressed_info.txt`  `lzw_info.txt`

8.3 源代码

```
import heapq
import os
import time
from collections import defaultdict
from decimal import Decimal, getcontext # 用于高精度算术计算

# 霍夫曼树节点类
class HuffmanNode:
    def __init__(self, freq, byte=None, left=None, right=None):
        self.freq = freq # 节点频率
        self.byte = byte # 字节值（仅叶子节点有值）
        self.left = left # 左子树
        self.right = right # 右子树
```

```

# 用于堆排序的比较方法
def __lt__(self, other):
    return self.freq < other.freq

def huffman_compress(text_bytes):
    # 统计字节频率
    freq = defaultdict(int)
    for byte in text_bytes:
        freq[byte] += 1 # 统计每个字节出现的次数

    # 构建优先队列（最小堆）
    heap = []
    for byte, count in freq.items():
        # 将每个字节转换为叶子节点加入堆
        heapq.heappush(heap, HuffmanNode(count, byte=byte))

    # 构建霍夫曼树
    while len(heap) > 1:
        # 取出频率最小的两个节点
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        # 合并为新的内部节点（频率为两者之和）
        merged = HuffmanNode(left.freq + right.freq, left=left,
right=right)
        heapq.heappush(heap, merged)

    root = heapq.heappop(heap) if heap else None # 根节点
    code_table = {} # 编码表（字节->二进制字符串）

    # 递归构建编码表
    def build_code(node, current_code=""):
        if node is None:
            return
        if node.byte is not None:
            # 叶子节点：记录字节对应的编码
            code_table[node.byte] = current_code
            return
        # 左子树编码加 0，右子树编码加 1
        build_code(node.left, current_code + '0')

```

```

        build_code(node.right, current_code + '1')

    if root:
        build_code(root)

    # 生成编码位流
    encoded_bits = ''.join([code_table[byte] for byte in
text_bytes])
    # 计算填充位数（使总位数为 8 的倍数）
    padding = 8 - (len(encoded_bits) % 8)
    if padding != 8:
        encoded_bits += '0' * padding # 填充 0

    # 转换为字节列表
    bytes_list = [int(encoded_bits[i:i+8], 2) for i in range(0,
len(encoded_bits), 8)]

    # 保存压缩后的二进制文件
    with open('huffman_compressed.bin', 'wb') as f:
        f.write(bytes(bytes_list))

    # 保存中间信息（频率表和编码表）
    with open('huffman_info.txt', 'w', encoding='utf-8') as f:
        f.write("=== 字符频率表 ===\n")
        # 按频率从高到低排序
        for byte, count in sorted(freq.items(), key=lambda x: -
x[1]):
            # 处理不可打印字符（用空格表示）
            char = chr(byte) if 32 <= byte <= 126 else ' '
            f.write(f"字节 {byte:3d}（字符: {char}）: 频率 =
{count}\n")

        f.write("\n=== 霍夫曼编码表 ===\n")
        # 按编码长度排序
        for byte, code in sorted(code_table.items(), key=lambda
x: len(x[1])):
            char = chr(byte) if 32 <= byte <= 126 else ' '
            f.write(f"字节 {byte:3d}（字符: {char}）: 编码 =
{code}\n")

    return {

```

```

        'original_size': len(text_bytes),      # 原始大小
        'compressed_size': len(bytes_list),    # 压缩后大小
        'compression_ratio': len(bytes_list) / len(text_bytes) if
text_bytes else 0,
        'time': 0                             # 预留时间字段
    }

def arithmetic_compress(text_bytes):
    getcontext().prec = 1000 # 设置高精度十进制精度

    # 统计字节频率
    freq = defaultdict(int)
    for byte in text_bytes:
        freq[byte] += 1

    total = sum(freq.values()) # 总字符数
    cum_prob = {}              # 累积概率表 (字节->(下限, 上限))
    current = Decimal(0)       # 当前累积概率

    # 生成累积概率区间
    for byte in sorted(freq.keys()): # 按字节值排序
        prob = Decimal(freq[byte]) / Decimal(total) # 计算概率
        cum_prob[byte] = (current, current + prob) # 记录区间
        current += prob # 累加概率

    low = Decimal(0) # 区间下限
    high = Decimal(1) # 区间上限

    # 逐个字节更新区间
    for byte in text_bytes:
        char_low, char_high = cum_prob[byte] # 获取当前字节的概
率区间
        range_size = high - low # 当前区间长度
        # 缩小区间范围
        high = low + range_size * char_high
        low = low + range_size * char_low

    # 将最终区间转换为二进制字符串
    binary_str = []
    value = low # 取区间内任意值 (通常取下限)

```

```

for _ in range(1024): # 最多生成 1024 位二进制
    value *= 2        # 左移一位（相当于乘以 2）
    bit = int(value)   # 提取整数部分作为二进制位
    binary_str.append(str(bit))
    value -= bit       # 保留小数部分
    if value == 0:     # 提前结束条件
        break

binary_str = ''.join(binary_str)
# 填充到 8 的倍数
padding = 8 - (len(binary_str) % 8)
if padding != 8:
    binary_str += '0' * padding

# 转换为字节列表
bytes_list = [int(binary_str[i:i+8], 2) for i in range(0,
len(binary_str), 8)]
# 保存压缩后的二进制文件
with open('arithmetic_compressed.bin', 'wb') as f:
    f.write(bytes(bytes_list))

# 保存中间信息（概率区间和最终区间）
with open('arithmetic_info.txt', 'w', encoding='utf-8') as f:
    f.write("=== 字符概率区间表 ===\n")
    for byte in sorted(cum_prob.keys()):
        low_range, high_range = cum_prob[byte]
        # 转换为浮点数便于显示（牺牲精度）
        char = chr(byte) if 32 <= byte <= 126 else ' '
        f.write(f"字节 {byte:3d}（字符: {char}）; 区间 =
[{{float(low_range):.10f}}, {{float(high_range):.10f}}]\n")

    f.write(f"\n最终压缩区间: [{{float(low):.20f}},
{{float(high):.20f}}]\n")

    return {
        'original_size': len(text_bytes),
        'compressed_size': len(bytes_list),
        'compression_ratio': len(bytes_list) / len(text_bytes) if
text_bytes else 0,
        'time': 0
    }

```

```

    }

def lzw_compress(text_bytes):
    # 初始化字典：单字节到索引的映射（0-255）
    dictionary = {bytes([i]): i for i in range(256)}
    next_code = 256 # 下一个可用索引
    s = bytes()      # 当前匹配字符串
    encoded = []      # 编码结果列表

    for byte in text_bytes:
        sc = s + bytes([byte]) # 尝试扩展当前字符串
        if sc in dictionary:
            s = sc # 存在则继续扩展
        else:
            # 输出当前字符串的索引
            encoded.append(dictionary[s])
            # 将新字符串加入字典
            dictionary[sc] = next_code
            next_code += 1
            s = bytes([byte]) # 重置当前字符串为当前字节
    # 处理最后一个字符串
    if s:
        encoded.append(dictionary[s])

    # 将编码转换为 12 位二进制字符串（假设索引最大 4095）
    bits_str = ''.join([format(code, '012b') for code in
encoded])
    # 填充到 8 的倍数
    padding = 8 - (len(bits_str) % 8)
    if padding != 8:
        bits_str += '0' * padding

    # 转换为字节列表
    bytes_list = [int(bits_str[i:i+8], 2) for i in range(0,
len(bits_str), 8)]
    # 保存压缩后的二进制文件
    with open('lzw_compressed.bin', 'wb') as f:
        f.write(bytes(bytes_list))

    # 保存中间信息（字典大小和编码序列）

```



```

with open('lzw_info.txt', 'w', encoding='utf-8') as f:
    f.write(f"字典最大索引: {next_code - 1}\n")
    f.write("\n=== 编码序列 ===\n")
    for i, code in enumerate(encoded):
        if i % 20 == 0 and i != 0:
            f.write("\n") # 每 20 个编码换行
        f.write(f"{code:4d} ") # 固定宽度输出

    return {
        'original_size': len(text_bytes),
        'compressed_size': len(bytes_list),
        'compression_ratio': len(bytes_list) / len(text_bytes) if
text_bytes else 0,
        'time': 0
    }

def main():
    # 读取原始文件（自动处理 UTF-8 编码）
    with open('pki_text.txt', 'r', encoding='utf-8') as f:
        text = f.read()
    text_bytes = text.encode('utf-8') # 转换为字节流

    # 霍夫曼编码
    start = time.time()
    huffman_result = huffman_compress(text_bytes)
    huffman_time = (time.time() - start)*1000

    # 算术编码
    start = time.time()
    arithmetic_result = arithmetic_compress(text_bytes)
    arithmetic_time = (time.time() - start)*1000

    # LZW 编码
    start = time.time()
    lzw_result = lzw_compress(text_bytes)
    lzw_time = (time.time() - start)*1000

    # 输出霍夫曼编码结果
    print("霍夫曼编码:")

```

```

")    print(f"原始文件大小: {huffman_result['original_size']} 字节
")    print(f"压缩后大小: {huffman_result['compressed_size']} 字节
      print(f"压缩比: {huffman_result['compression_ratio']:.2%}")
      print(f"耗时: {huffman_time:.2f}ms\n")

      # 输出算术编码结果
      print("算术编码:")
      print(f"原始文件大小: {arithmetic_result['original_size']} 字
节")
      print(f"压缩后大小: {arithmetic_result['compressed_size']} 字
节")
      print(f"压缩比:
{arithmetic_result['compression_ratio']:.2%}")
      print(f"耗时: {arithmetic_time:.2f}ms\n")

      # 输出LZW 编码结果
      print("LZW 编码:")
      print(f"原始文件大小: {lzw_result['original_size']} 字节")
      print(f"压缩后大小: {lzw_result['compressed_size']} 字节")
      print(f"压缩比: {lzw_result['compression_ratio']:.2%}")
      print(f"耗时: {lzw_time:.2f}ms\n")

if __name__ == "__main__":
    main()

```