华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析　　**年级**：22 级　　**上机实践成绩**：

指导教师：金澈清　　**姓名**：郭夏辉

上机实践名称：贪心算法 2　　**学号**：10211900416　　**上机实践日期**：2023 年 5 月 18 日

上机实践编号：No.11　　**组号**：1-416

一、目的

1. 熟悉贪心算法设计的基本思想
2. 掌握计算哈夫曼编码的方法

二、内容与设计思想

1. 基于朴素固定长度编码编写字符串编码的代码。
2. 基于贪心算法编写计算哈夫曼编码的代码。
3. 请在网上随意找一些字符串（如文章报道等），对比两种实现方式编码结果长度，计算其压缩比。
4. 对比两种实现方式编码以及译码速度的差异，并简单描述你是如何实现编码和译码的。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

1. 写出计算两种编码方式的代码。

2. 用合适的统计图描述你的实验结果。

首先让我们先来回顾一下并对比一下两种算法，再来结合我实验过程中碰到的一些问题谈一下编码和译码

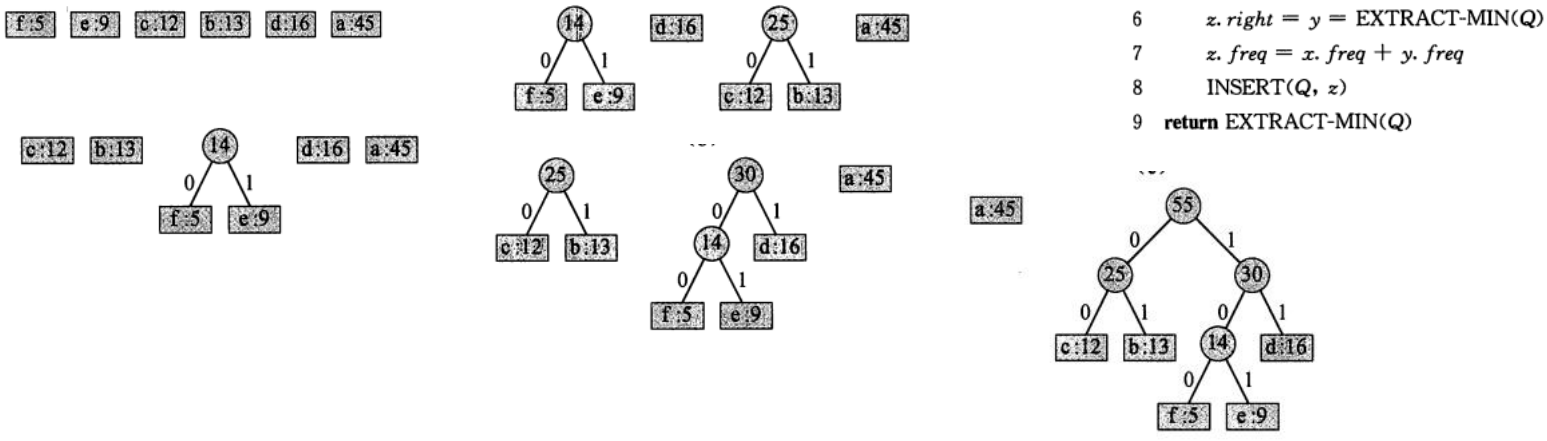
- Huffman 编码

从整体上来说，Huffman 编码是一种变长编码，赋予高频字符短码字(每个字符的底层肯定是一个 0/1 串，这便是该字符的码字)，而赋予低频字符长码字，然后利用贪心算法，较大程度地减少文件的总体编码长度，然后实现较好的数据压缩功能。

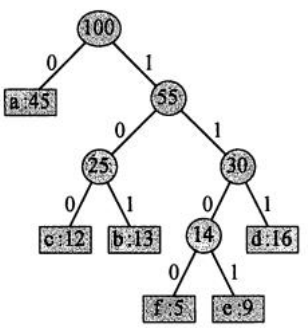
有关 Huffman 编码具体而严谨的证明此处就不展开了，我主要是围绕它的过程来谈一下吧：

在下面给出的伪代码中，我们假定 C 是一个 n 个字符的集合，而其中每个字符 $c \in C$ 都是一个对象，其属性 $c.freq$ 给出了字符的出现频率。算法自底向上地构造出对应最优编码的二叉树 T。它从 $|C|$ 个叶结点开始，执行 $|C|-1$ 个“合并”操作创建出最终的二叉树。算法使用一个以属性 $freq$ 为关键字最小优先队列 Q，以识别两个最低频率的对象将其合并。当合并两个对象时，得到的新对象的频率设置为原来两个对象的频率之和。

```
HUFFMAN(C)
1  n=|C|
2  Q=C
3  for i = 1 to n−1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)
```



通过细致的原理讲解和图示，相信我们对 Huffman 算法的编码流程有了一个更深刻的认识，但是面对一串已经编码好的字符串，我们应该如何去解码呢？其实也很简单，就是我们只要实现一个构建 Huffman 的逆操作就行，让编码字符串沿着已经架构好的 Huffman 树从上到下走一遍即可，走到头了说明某个字符的翻译工作完成了。



在具体的实践过程中，还有一个问题，就是如果我们碰到两个字符的频数相等，这个应

该如何处理？也算是结合 OJ 上相关测试样例的启发，我发现这时应该采用那个 ASCII 值更小的字符。

接着就是用代码实现了，本着利用 C++ STL 库的想法，我在这次的实验过程中大量地使用了相关的 STL 特性:

unordered_map

这是一个关联容器，它提供了基于键-值对的快速查找和插入操作。它可以确保 O(1)的时间复杂度实现相关的查找和插入操作，但是有一个不太好的性质就是无序的，不能直接进行排序这样的操作。然后 OJ 最开始的任务需要排序并输出相应的字符对应表，然后这里需要先转化为 vector 这样的有序数据结构才能进行 sort 操作。在本次实验中，我利用 unordered_map 来存储相应的字符表和对应的编码表。

pair

它用于存储两个不同类型的值（键值对），提供了一种简单的方式来将两个值组合在一起。在结合之后，我们也可以方便地去调用相关的键或者值，只用通过 first 或者 second 成员就能访问或者修改。

priority_queue

它是一个优先队列，其中的元素按照一定的优先级进行排序，具有最高优先级的元素位于队列的前面，然后我们可以通过自定义规则来改变优先级。在构造哈夫曼树的过程中使用它是因为构造哈夫曼树的过程中需要找到频数最小的两个元素进行合并。

- 朴素固定长度编码

固定长度编码的原理应该是比较容易的，我的想法是先预处理出文本中所有的不同字符，然后得到相应的最小二进制位数能够全部标识它们。（codeLength = ceil(log(n) / log(2));）然后呢，就是将这个可能出现的字符集合中的每一个字符都扫描一遍，然后根据第几次扫描得出一个相应的二进制定长编码。

至于如何解码，我的思路是这样的，就是根据编码时得到的表，根据那个固定的码的位数，制造一个反向的表格可以通过编码得到相应的字符。然后根据这个新表，一一对应从后往前地将编码翻译出来。

接着就是用代码实现了，这里我依旧大量地使用了相关的 STL 特性:

unordered_set

这个类似于 unordered_map，用于存储一组独特的元素，其中元素的顺序是不确定的。它也提供了高效的插入与查找功能，但是有一个比较大的特点就是唯一性，通过它的这个特性可以忽略相同且重复的插入操作。

bitset

bitset 是一个类模板，它用于表示固定大小的二进制序列。它提供了一组功能强大的操作和方法，用于对二进制数据进行处理和操作。我在这个实验中，利用 string code = bitset<32>(i).to_string().substr(32 - codeLength);

可以将 i 的 32 位二进制表示转化为字符串之后截取最后 codeLength 位，直到字符串的结束为止。

- 相关的性能测试

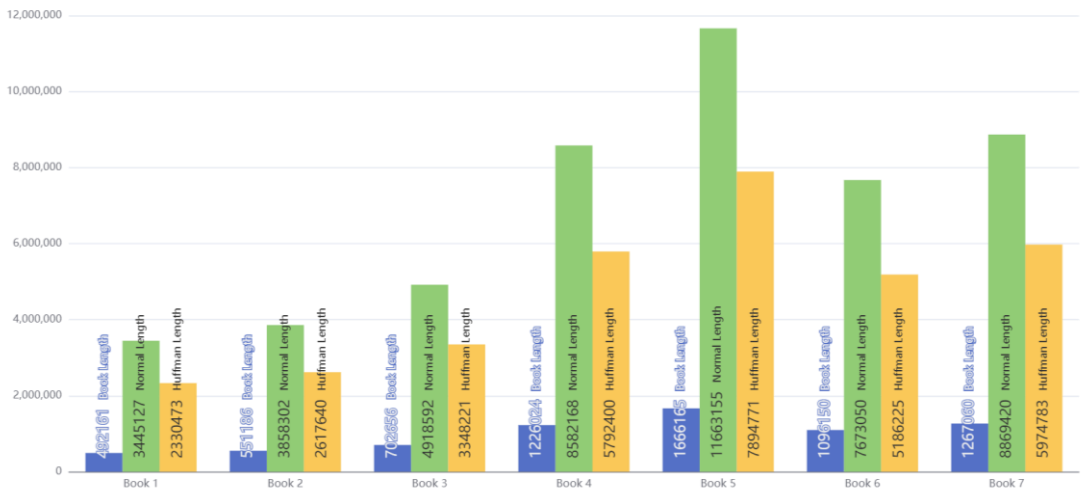
这次实验有一个比较棘手的问题就是在于面对随机的、无意义的文本，我们并不能很明确地看到 Huffman 算法的价值——因为 huffman 编码时基于文本字符的频数的，这又取决于我们日常使用的单词，如果单词就是混乱的，得到的结论也并不是那么靠谱。

虽然最后 Huffman 编码的效果和作者本人的风格是有关的，但是这个在英文单词基本固定的情况下应该差距不算大。为了较好地比较两种算法的编码和解码性能，我选择了自己比较喜爱的，也是较长篇的哈利波特七部曲英文原版作为输入数据。

五、总结

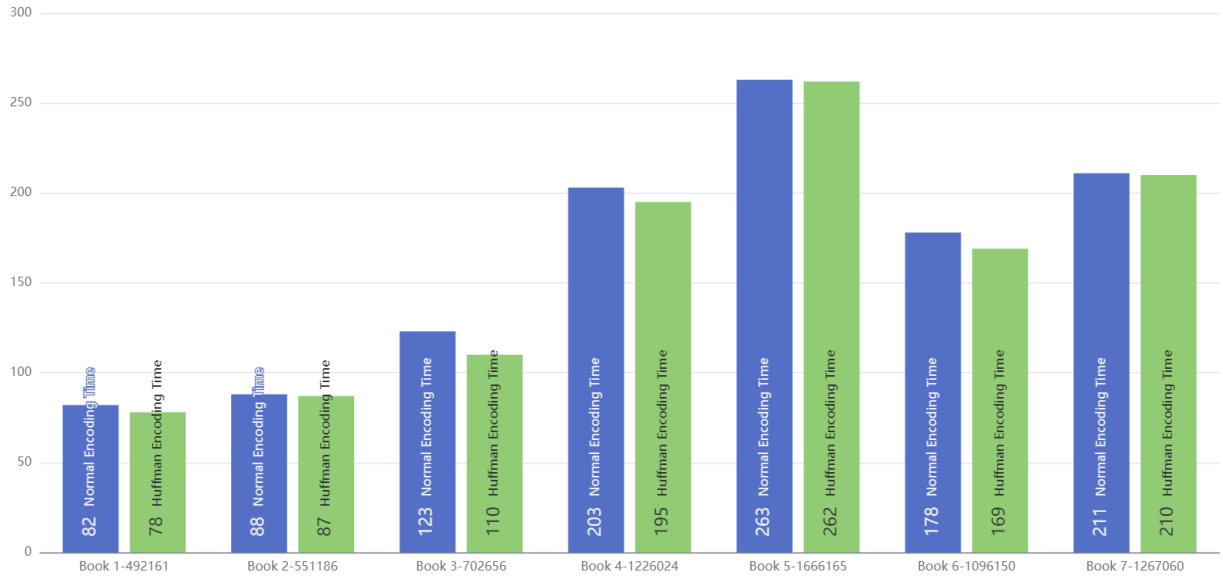
对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。

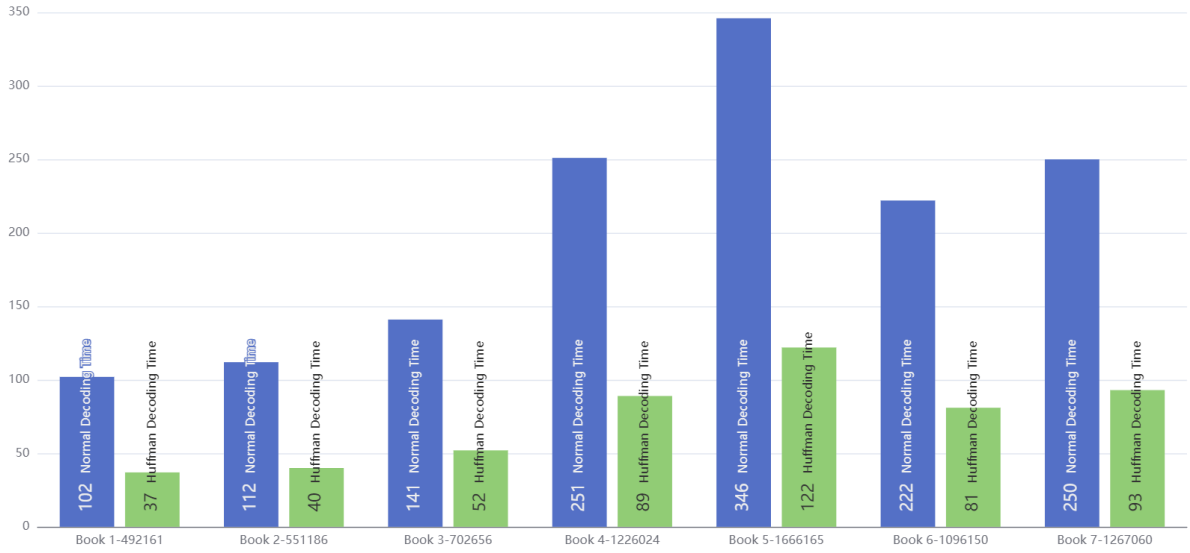
首先是针对编码长度的结果对比:



通过图示，可以很明确地看到使用 Huffman 的威力，相较于使用朴素的定长编码，7 本小说的优化程度分别是 32.4%，32.16%，31.93%，32.51%，32.31%，32.40%，32.64%

接下来就是两种算法的编码和译码时间对比，我先放的是编码时间，再放的是译码时间（最底下标注的有长度）：





我们先来讨论编码时间，可以看到在文件的长度一致时，Huffman 和朴素算法的编码时间相差不大。之所以会出现这样的情况，主要是因为两者构造对应表后一个字符一个字符地去对应到相应编码的时间差距不大（这里主要也是 unordered_map 和 unordered_set 的功能，两者都提供了常数级别的插入和查询功能，并且我的 Huffman 算法采用了优先队列来优化，显著地降低了构造表的时间）。综上所述，两者编码的时间应该都是 O(nlgn).

接着来讨论译码时间，可以很明显地看到在文件的长度一致时，Huffman 显著地比朴素算法的编码时间短。对于 huffman 算法而言，其对于每个字符的译码时间就是沿着 Huffman 树从根往下遍历的时间，这样运行时间就主要取决于字符串的长度和 huffman 树的平均深度。对于朴素编码而言，因为最开始已经生成了相应的编码译码对应表，这里只需要各个字符挨个对应就好了，其运行时间主要取决于字符串的长度。由于这里朴素编码面对的需要解码的字符串更长，而且涉及到了相关二进制位重复的转换和对应，因此运行时间要显著地高于 Huffman 编码，但是它们的渐进时间复杂度应该是近似的。

通过本次实验，在直观地见识到 Huffman 算法的强大压缩效果后，我对它有了一个更深刻的认识，最后想说在实验中一定要善于运用 C++ 的 STL 特性来简化问题，这样可以节省大量的时间。