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**：21 级　　**上机实践成绩**：

指导教师：金澈清　　**姓名**：包亦晟

上机实践名称：贪心算法 2　　**学号**：10215501451　　**上机实践日期**：2023.5.18

上机实践编号：No.11　　**组号**：1-451

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

STL

本次实验中用到了许多 C++STL 里的东西，在此先作一个总体的介绍。

unordered_map

unordered_map 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。它是使用哈希表实现的，因此查找和插入操作的平均时间复杂度为 O(1)。它的作用类似于 python 中的字典。它的一个重要特性是**无序性**，所以我们是无法对其进行排序的。在本次实验中我使用 unordered_map 存储字符和其对应的编码。

vector

vector 是一个顺序容器，它提供了动态数组的功能。它可以在运行时动态调整大小，并支持快速的随机访问。

pair

pair 是一个模板类，它用于存储两个不同类型的值（键值对）。它提供了一种简单的方式来将两个值组合在一起，并允许以一对值的形式进行操作。我们可以通过 first 和 second`成员变量来访问这两个值。最重要的一点是，pair 是可以作为容器中的元素进行存储的，这给予了我们很大的操作空间。

priority_queue

priority_queue 是一个容器适配器，它提供了基于优先级的元素访问和插入操作。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。**我们可以通过自定义比较规则来改变排序方式**。我在构造哈夫曼树的过程中使用到了它，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。

unordered_set

unordered_set 是一个关联容器，它用于存储一组独特的元素，其中元素的顺序是不确定的。它基于哈希表实现，提供了高效的插入、查找和删除操作。它的一个重要特性是**唯一性**，当尝试插入重复元素时，插入操作会被忽略。因此，我们可以使用它来进行去重。

bitset

bieset 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。

```
bitset<8> bits(42);
```

这一行代码会生成整数 42 的二进制表示的后 8 位。我在朴素固定长度编码中用到了 bitset。

朴素固定长度

书中给出的定长编码是这样的：

	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长编码	000	001	010	011	100	101
变长编码	0	101	100	111	1101	1100

如何生成呈现这种规律的编码呢？我的最初思路是通过 dfs 穷举出所有的 01 序列可能性。但是一是时间复杂度很高，二是不能保证是按照 000,001,010…这样的顺序生成的。什么序列能够呈现这样一种规律呢？我想了很久终于想到了，答案是：1,2,3…的二进制表示。

我的基本思路如下：

1. 预处理出文本中所有的不同字符
2. 根据不同字符的数量估算需要的位数
   - `int codeLength = ceil(log(n) / log(2));`

     这里的 ceil 函数表示向上取整，括号中的内容是根据换底公式来估算需要的二进制位数，ceil 和 log 都在 cmath 库中
3. 令 n=不同字符数，让 i 从 0 迭代到 n-1，每一次迭代中都给予相应索引处的字符一个定长编码
   - `string code = bitset<32>(i).to_string().substr(32 - codeLength);`

     这一行代码的作用是将 i 的 32 位二进制表示转化为字符串之后截取最后 codeLength 位，substr 中传入的参数表明从 32-codeLength 处截取，直到字符串结束

哈夫曼编码

哈夫曼编码的思路其实是很清楚的，我在本次报告中就写一些 C++代码实现方面的问题。

在代码中，我定义了一个结构体，存放字符，频数，以及左右孩子。那么就出现一个问题，由于后面的过程中需要用到小根堆，而此时比较的元素的数据类型是一个复杂的结构体，这就意味着我们需要自己定义比大小的规则。

```
struct Compare {
    bool operator()(const T& lhs, const T& rhs) {
    }
};
```

返回 true 表示 lhs 应排在 rhs 前面，返回 false 表示 lhs 应排在 rhs 后面。

这里的一个注意点是：**建树时若概率（频数）相等，则 ASCII 码小的当做频数更小**。也就是说我们需要先比较频数，如果相等，再比较 ASCII 码值的大小。这一点我最初没注意到，导致 2 和 10 测试用例过不了，想必是出现了频数相等的情况。

创造最终的哈夫曼编码的时候，我们用到了久违的 dfs（当然我在之前一次的实验中已经用到过了）。这里是一个 void 型 dfs，因为我们的目标不是找到一个字符的编码就行了，而是要找到所有字符的编码。这里的 dfs 比较简单，相信好好学过上学期数据结构初步的同学都能写出来。

test1

在 test1 中，我进行了对于朴素固定长度编码译码速度的测试。

在进行测试之前，我通过 unorderedset 对于字符进行了去重操作，最终得到的都是不同的字符，而 unorderedset 的大小就是不同字符的个数。

这里使用文件流的时候我们要注意文件指针的问题。在我们统计不同字符的时候，需要过一遍文件。而在之后编码的时候，我们又需要过一遍文件。但是此时的文件指针已经到达了文件末尾，所以在编码之前，我们要注意把文件指针移动到文件头。可以通过以下代码实现：

```
f.clear();
f.seekg(0, ios::beg);
```

还有就是 unorderedset 是无法通过索引访问元素的，而在 generateCode 函数中是必须要通过索引访问的，所以要先将 unorderedset 转化为 vector。通过下面一行代码就可以实现：

```
charset.assign(t.begin(), t.end());
```

在译码的时候，因为要根据 01 序列得出字符了，所以需要构建另一个 unordered_map，将 01 序列作为键，字符作为值。

```
unordered_map<string, char> decodeTable;
for (const auto& pair : codeTable) {
        decodeTable.emplace(pair.second, pair.first);
}
```

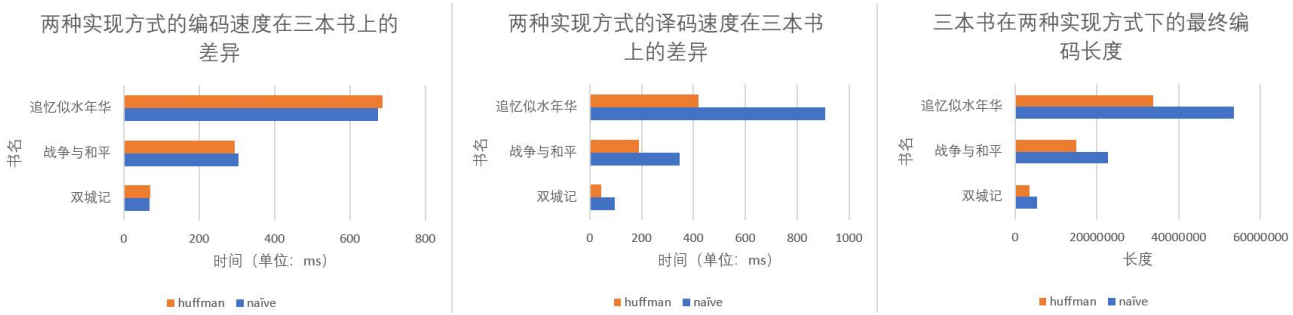
test2

测试哈夫曼编码的代码其实和上面的差不多。在译码的时候，我最初想要采取和 test1 中相同的方法，构建另一个 unordered_map，将 01 序列作为键，字符作为值。但是问题在于，这里是变长编码，你不能确定哪几个 01 确定字符，遂作罢。我最终采取的是遍历哈夫曼树的方式。跟着 01 序列走，如果走到叶子结点了，那就代表了一个字符，就可以记录下，然后从 root 重新开始。

main

我选择了双城记、战争与和平和追忆似水年华三本书作为测试材料。因为我发现如果文章字数太少，那么测试出来的时间就会很少，难以展现实验结果，所以选取了字比较多的名著。同时，这三本书的字数也有一定的差距，容易进行比较。双城记有 139933 字，战争与和平有 573652 字，追忆似水年华有 1374301 字。

折线图



五、总结

编码时间比较

从上面的两张图当中，我们可以看出当书的字数越多的时候，两种方式的编码/译码时间也会随之增加。在编码时间上，哈夫曼编码和朴素固定长度编码是差不多的。我测量的编码时间其实是**这两种方式最终得到从字符到 01 序列的 unordered_map 的时间**再加上**根据该 unordered_map 将文本完全编码的时间**。对于后者，两者都是遍历整个文本，然后根据 unorderedmap 进行编码，所以这段两者的时间是基本一样的。假设 n 是文本中不同字符的个数。哈夫曼编码需要先建堆，需要$O(n)$时间。然后再进入 while 循环，循环 n-1 次，而每个堆操作需要$O(\lg n)$，所以循环的总体时间贡献为$O(n\lg n)$。而朴素固定长度编码只需要$O(n)$。但是为什么最后的编码时间差不多呢？我的猜测是朴素固定长度的常数比较大。因为在 generateCode 中使用了 biteset 这一类模板还有 tostring 和 substr 两个函数，并且我指定了 bitset 大小为 32 位，to_string 自然也固定转换 32 位，substr 也是固定切割 codeLength 位。这些虽然在渐进时间复杂度中被忽略了，但对于实际运行还是有一定影响的。

译码时间比较

与编码时间不同，在译码时间上，哈夫曼编码相比起朴素长度编码明显具有优势。哈夫曼编码译码时间就是**跟着编码字符串沿着哈夫曼树寻找的时间**，它是与编码字符串长度和寻找时间成正比的。朴素固定长度编码译码时间就是**根据反向 unorderedmap 遍历编码字符串的时间**，我并**没有**将构造反向 unorderedmap 的时间囊括在内。因此朴素固定长度编码译码时间仅仅和编码字符串成正比的。而即便如此，哈夫曼编码的时间仍然是显著得少。可以看出两者所得到的编码字符串长度的差距有多大了。

编码结果长度比较

通过实验数据，在双城记、战争与和平和追忆似水年华三本书上哈夫曼编码得到的编码字符串长度与朴素长度编码得到的编码字符串长度之比分别为 64.45%，53.15%，65.47%。又一次显示出哈夫曼编码能给出的编码字符串的长度之短，这与哈夫曼编码能给出文本的最优字符编码的理论分析相符合。