C++ 算法：贪心算法（下）讲义

1.1 问题引入：分卷子

现在有一大摞混在一起的试卷，已知 A, B, C, D 四个等级的卷子数量。智能分卷机每次只能执行一次二分操作，即将一摞卷子分成两堆，且同等级的卷子必须在同一堆。

每次操作的**成本**，等于当前被分的那一摞卷子的**总数量**。请设计一个最优的分类方案，使得总成本最小。

例如，A, B, C, D 的数量分别为 10, 20, 30, 40。

- **方案一 (1+3拆分):**
    1. `{A,B,C,D}` -> `{A}` + `{B,C,D}`，成本 = 10+20+30+40 = 100
    2. `{B,C,D}` -> `{B}` + `{C,D}`，成本 = 20+30+40 = 90
    3. `{C,D}` -> `{C}` + `{D}`，成本 = 30+40 = 70
    4. 总成本 = 100 + 90 + 70 = **260**
- **方案二 (2+2拆分):**
    1. `{A,B,C,D}` -> `{A,B}` + `{C,D}`，成本 = 10+20+30+40 = 100
    2. `{A,B}` -> `{A}` + `{B}`，成本 = 10+20 = 30
    3. `{C,D}` -> `{C}` + `{D}`，成本 = 30+40 = 70
    4. 总成本 = 100 + 30 + 70 = **200**

不同的拆分结构，导致了完全不同的成本。这引发了我们的核心问题：最优的拆分结构是什么？

1.2 学习目标

1. 掌握贪心模型——**霍夫曼模型。**
2. 使用**双队列法**高效实现霍夫曼模型。
3. 识别并将“合并果子”、“哈夫曼编码”等问题归纳到霍夫曼模型中。

---

2. 分卷子

2.1 正向思考

一个自然的猜想是：为了让总成本小，应该尽早地把数量多的卷子分离出去，这样它就不用参与后续的拆分，似乎能降低成本。
例如，对于 {10, 20, 30, 40}，我们每次都分离出当前最大的一堆：

1. `{A,B,C,D}` -> `{D}` + `{A,B,C}`，成本 100
2. `{A,B,C}` -> `{C}` + `{A,B}`，成本 60
3. `{A,B}` -> `{B}` + `{A}`，成本 30
4. 总成本 = 100 + 60 + 30 = **190。**这确实比之前的200和260都小！

但这个策略是普适的吗？考虑 {30, 30, 30, 30}：

- **按“最大”分离：** 成本 = 120 + 90 + 60 = **270。**
- **按“对半分”：** 成本 = 120 + 60 + 60 = **240。**

结论： “每次分离出最大的一堆”这个贪心策略是**错误**的。问题的关键不仅在于分离哪个，还在于**拆分的结构**。正向思考（从上往下拆分）让我们陷入了困境。

---

3. 逆向思维

当一条路走不通时，不妨倒过来走。**“从上往下”拆分的最优解，就是等价于“从下往上”合并的最小成本解！**

每一次拆分操作，成本是被拆分堆的总量。反过来看，每一次合并操作，成本是被合并堆的总量。这两个过程是等价的。

于是可以将“分卷子”问题，转化为一个更经典的逆向问题——**合并果子。**

> **问题描述：** 有 `n` 堆果子，每次可以合并任意两堆，合并的体力消耗等于两堆果子的重量之和。求将所有果子合并成一堆的最小总体力消耗。

3.1 贪心策略的确立

假设有三堆果子 {1, 2, 9}：

- 先合并 1 和 9 (成本10)，再与 2 合并 (成本12)，总成本 22。
- 先合并 2 和 9 (成本11)，再与 1 合并 (成本12)，总成本 23。
- 先合并 1 和 2 (成本3)，再与 9 合并 (成本12)，总成本 **15。**

我们发现，一个较早合并产生的大重量堆（如10或11），会像滚雪球一样，在后续的合并中**反复贡献**它的重量，从而累加出巨大的总成本。

为了让这个雪球滚得尽可能慢、尽可能小，我们的贪心策略必须是：**每一次合并，都选择当前果园里重量最小的两堆果子进行合并。**

---

4. 霍夫曼模型与实现

我们将“合并果子”这种问题抽象出来，称之为**霍夫曼模型。**

4.1 模型定义

当一个问题符合以下特征时，它就属于霍夫曼模型：

1. **初始元素：** 给定 `n` 个带权值的独立元素。
2. **操作规则：** 存在一种“二合一”的合并操作。
3. **成本计算：** 每次合并的成本是被合并两元素的权值之和，新产生的元素权值也等于这个和。
4. **目标：** 经过 `n-1` 次合并后，使总成本最小。

对于所有符合霍夫曼模型的问题，最优贪心策略都是：永远优先合并当前权值最小的两个元素。

4.2 高效实现：双队列法

如何高效地“永远找到并合并最小的两个”？
一个想法是：将所有数字放入一个数组，排序，取出最小的两个合并，把新数字放回数组，再排序... 这样做效率比较低。

这里我们介绍一种巧妙的**双队列法**：

核心思想：

1. 初始的 `n` 堆果子，我们先将其**从小到大排序**，放入第一个队列 `Q1`。
2. 所有通过合并**新产生**的果子堆，我们放入第二个队列 `Q2`。

关键洞察：

- `Q1` 内部本身是有序的。
- 我们总是合并最小的两个，所以后合并产生的新堆，重量一定不小于先合并产生的新堆。因此，我们每次将新堆放入 `Q2` 的**队尾**，`Q2` 内部也自然地**保持了有序**！
- 基于此，在任何时刻，要寻找全局的最小值，它只可能出现在哪里？`Q1` **的队头** 或 `Q2` **的队头**。

实现步骤：

1. 将所有 `n` 个初始值读入数组，并进行一次**升序排序**。
2. 创建两个队列（或用 `vector` 模拟），`Q1` 存储排序后的初始权值，`Q2` 为空。
3. 循环 `n-1` 次，执行合并操作：
   a. 定义一个 `get_min()` 函数，它只比较 `Q1` 和 `Q2` 的队头，返回并弹出较小的那个。（需处理其中一个队列为空的情况）。
   b. 调用两次 `get_min()`，得到全局最小的两个元素 `min1`, `min2`。
   c. 计算成本 `cost = min1 + min2`，并累加到总成本中。
   d. 将新产生的权值 `cost` **放入** `Q2` **的队尾**。
4. 循环结束后，累加的总成本即为最小总成本。

这个方法的总时间复杂度为 `O(N log N)`，主要来自最初的一次排序，非常高效。

关键代码（哈夫曼编码和合并果子都是本代码）：

```
// get_min 函数从两个有序队列的队头取出一个全局最小值
// 实现非常简单故省略，只需要比较两个队列的队头元素谁更小返回即可（注意判空！）
int get_min(queue<int>& q1, queue<int>& q2);

// 以下为 main 函数中内容
// 先对进行一次升序排序
sort(a.begin(), a.end());

// 从两个有序队列的队头取出一个全局最小值
for (int i = 0; i < n - 1; ++i) { // 进行 n-1 次合并
    // 取出全局最小的两个数
    int first_min = get_min(Q1, Q2);
    int second_min = get_min(Q1, Q2);
    int current_cost = first_min + second_min;
    total_cost += current_cost;
    Q2.push(current_cost); // 将新堆放入 Q2
}
// 最终输出 total_cost 即可
```

---

5. 哈夫曼编码

至此，我们已经掌握了通过逆向思维，用“合并”来解决“拆分”问题的思想。现在，我们将这个思想应用到一另一个问题上——**数据压缩**。而这个问题的最优解，正是以其发明者命名的**哈夫曼编码**。

问题： 为一组字符设计一种二进制**前缀码**（即任何字符的编码不能是另一字符编码的开头部分），使得出现**频率高的字符编码尽可能短**，**频率低的字符编码可以长一些**。最终目标是使编码后的信息流**总长度最小**。

5.1 核心目标

要解决这个问题，我们首先要理解一串信息的总长度是怎么计算的。
`总长度 = Σ（每个字符的频率 × 其编码的长度）`

这个公式有点抽象。让我们把它转化成一个更直观的猜谜问题：

- **编码：** 计算机用 `0` 和 `1` 进行编码。
- **猜谜：** 我们通过问一系列"是/否"问题来确定一个字符。

可以建立一个对应关系：**一个字符的"编码长度"，就等于在这场猜谜游戏中，唯一确定它所需要问的"问题个数"。**

例如，设计了这样一套问题流程，在集合 {A, B, C, D} 中选择一个字符让你猜测：

1. 第一个问题："它在A/B组里吗？" -> 是(0) / 否(1)
2. 第二个问题："它是A吗？" -> 是(0) / 否(1)
   如果一个字符的"答案记录"是 `00`，那么它的编码就是 `01`，编码长度为2，因为我们问了2个问题才找到它。
3. 因为这套答案记录具有唯一性，所以答案的记录就可以直接构成该字符的二进制编码(例如 00 唯一指代着A, 01 就唯一指代着 B)。

在这个模型下，**一个字符的"编码长度"，就严格等于找到它所需要问的"问题个数"**。所以，我们的目标可以转化为：
最小化总编码长度 = 最小化总提问次数
也就是：**总提问次数 = Σ (每个字符的频率 × 找到它所需的问题数)**

5.2 正向思考的难点：第一步应该问什么？

有了这个清晰的目标，我们该如何设计提问顺序呢，使得我们**能以更短的问题区分出频率最大的字符**？
比如，对于频率为 `{5, 10, 13, 14, 20}` 的五个字符，我们的第一个问题该怎么问？

- 是问："它在 `{5, 10}` 这组里吗？"
- 还是问："它在 `{20}` 这组里吗？"
- 或者问："它在 `{13, 14}` 这组里吗？"

这是一个非常难的开端。第一个问题有无数种问法，不同的问法会产生完全不同的后续问题。无法判断哪一个开局能导向全局最优。

5.3 逆向思维：最后一个问题问什么？

当正向思考走不通时，再次使用逆向思维。不去想**第一个**问题，我们去想**最后一个**问题。

在一个完整的提问流程中，"最后一个问题"是用来区分哪两个选项的？
根据我们的贪心原则"**频率低的，问的问题也可以越多**"，那些被藏在问题链条最深处、需要最多问题才能找到的，必然是**频率最低**的两个字符！

这给了我们一个出发点：

1. **第一步：** 在 `{5, 10, 13, 14, 20}` 中，找到两个频率最低的元素：**5** 和 **10**。

2. **构建过程：** 我们可以把这两个字符看作一个小团体。为了区分它俩，我们会设计一个专门的问题。这个专门的问题，在整个提问结构中，将处于最末端。
3. **逆向合并：** 我们把这个小团体“合并”成一个整体，并思考这个合并操作的“代价”是什么。

5.4 “合并代价”的含义

“合并 5 和 10” 产生的和 `15`，它的物理意义是什么？

让我们把那个专门用于区分“5”和“10”的问题称为 `Q_5_10`。

- `Q_5_10` **何时被问到？** 只有当我们已经确定目标字符就是“5”或“10”两者之一时，才会问这个问题。
- `Q_5_10` **总共会被问到多少次？** 在解码整个信息流的过程中，**每当出现一次频率为5的字符，或频率为10的字符**，我们最终都需要通过 `Q_5_10` 来做出最后区分。因此，这个问题总共会被问到 `5 + 10 = 15` 次。

结论：
每一次合并操作的代价（两个频率之和），就等于区分这两个子集的问题，在整个信息中被提出的总次数。

`Q_5_10` 这个问题一共会被询问15次，这个询问次数就是代价，也是**这次合并对“总提问次数”（也就是总编码长度）的贡献值**。

5.5 算法流程

现在，问题豁然开朗。因为前面已经论述了：在这个模型下，**一个字符的“编码长度”，就严格等于找到它所需要问的“问题个数”**。所以，我们的目标可以转化为：**要最小化总编码长度 就是要最小化总提问次数。**

而最小的总提问次数的求取就像是上面的 `Q_5_10` 一样，`Q_5_10` 这个问题一共会被询问15次，这个询问次数就是代价，也是**这次合并对总的提问次数的贡献值。**

故我们可以得到公式：**最小总编码长度 = Σ (每次合并操作的代价)**

这和“合并果子”问题中，“最小总体力消耗 = Σ (每次合并的体力)”是**完全等价**的！

5.6 举例实验

- **初始频率集合：{5, 10, 13, 14, 20}**
- **第一次合并：**
  - **操作：** 找到并合并两个频率最低的元素 `5` 和 `10`，形成一个代表 `{5, 10}` 子问题的新节点，其频率为 `5 + 10 = 15`。
  - **意义：** **此次合并的代价为 15。**这表示：在整个信息流中，那个用来区分“频率为5的字符”和“频率为10的字符”的最终问题，总共会被问到 **15** 次。
  - **剩余集合：** `{13, 14, 15, 20}`
- **第二次合并：**
  - **操作：** 在新集合中，合并最小的 `13` 和 `14`，新频率为 `13 + 14 = 27`。
  - **意义：** **此次合并的代价为 27。**这代表区分“频率为13的字符”和“频率为14的字符”的问题，总共会被问到 **27** 次。
  - **剩余集合：** `{15, 20, 27}`
- **第三次合并：**
  - **操作：** 合并最小的 `15` 和 `20`，新频率为 `15 + 20 = 35`。

- **意义：** **此次合并的代价为 35。**这里的 `15` 代表了 `{5, 10}` 组合。因此，这个代价的含义是：那个用来区分“`{5, 10}` 组”和“频率为20的字符”的问题，总共会被问到 **35** 次。
- **剩余集合：** `{27, 35}`

- **最后一次合并：**
  - **操作：** 合并最后的 `27` 和 `35`，得到根节点，频率为 `27 + 35 = 62`。
  - **意义：** **此次合并的代价为 62。**这代表整个提问流程中的第一个问题，即用来区分“`{13, 14}` 组”和“`{5, 10, 20}` 组”的顶层问题，总共会被问到 **62** 次（也就是每次都需要问）。
  - **剩余集合：** `{62}`，合并完成。

计算总成本：

我们将每一次合并产生的“代价”——即每个中间节点的频率——相加：

`总成本 = 15 (区分5和10) + 27 (区分13和14) + 35 (区分{5,10}和20) + 62 (区分{13,14}和{5,10,20}) = 139`

这就是最终的最小总编码长度。每一次合并的代价，都代表了一个“区分问题”在整个信息流中被提出的总次数。将所有这些“区分问题”的提问次数加起来，自然就等于所有字符在被唯一确定前，需要被提问的总次数。

通过这种方式，将一个看似复杂的编码问题，转化为了一个直观的、可累加的“合并代价”问题，并用贪心策略找到了最优解。

也可以通过类比的方式发现哈夫曼编码的构建过程，与合并果子是等价的！

- 字符频率 <==> 果子重量
- 合并频率 <==> 合并果子
- 最小总编码长度 <==> 最小总体力消耗

---

6. 总结与思考

6.1 核心要点

- **霍夫曼模型**：**目前可以暂时不理解哈夫曼编码**，但是要理解这个霍夫曼模型是“n元素带权，二合一合并求最小总成本”的问题。
- **核心策略**：并且也要知道**霍夫曼模型的解决策略永远优先合并当前权值最小的两个元素。**
- **逆向思维**：当正向的“拆分”或“构建”问题无从下手时，尝试逆向的合并过程。
- **高效实现**：**双队列法**是一种不依赖特殊数据结构，仅通过排序和队列就能高效解决霍夫曼模型的巧妙方法。

6.2 思考

1. **文件压缩**：思考一下，为什么哈夫曼编码能实现文件压缩？什么类型的文件压缩率会比较高？
2. **模型变种**：如果每次可以合并 `k` 堆果子，而不是2堆，贪心策略应该如何调整？