**数据结构课程设计**

**-修理牧场**

# 题目描述

项目说明：

农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要N块木头，每块木头长度为整数*Li*个长度单位，于是他购买了一个很长的，能锯成N块的木头，即该木头的长度是*Li*的总和。

但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨就设酬金等于所锯木头的长度。例如，要将长度为20的木头锯成长度为8，7和5的三段，第一次锯木头将木头锯成12和8，花费20；第二次锯木头将长度为12的木头锯成7和5花费12，总花费32元。如果第一次将木头锯成15和5，则第二次将木头锯成7和8，那么总的花费是35（大于32）。

项目功能要求：

输入格式：输入第一行给出正整数N（N<104），表示要将木头锯成N块。第二行给出N个正整数，表示每块木头的长度。

输出格式：输出一个整数，即将木头锯成N块的最小花费。

# 总体思路

1. Huffman树问题：

一根木头锯成N块有多种锯法，且部分锯面切割的时间先后可以颠倒，导致计算花费时更加复杂。但反过来，由N块木头“合并”成一块木头过程唯一。根据贪心思想，每次合并长度最小的两截木头，最终的花费一定是最小的。有两种方法：

1. 数据结构类似STL库中优先队列，每次输入向二叉树加入元素，构成小根堆存储木头长度，每次取两次根节点并删除，相加后添加到堆中，再进行调整，重复此操作到最后堆中只剩一个元素，即为结果。其过程实际上是求带权路径长度最小的二叉树，即Huffman树问题。
2. 使用一个数组用来存储已经按升序排好序的木头长度，记作oriArr；再用一个数组记录合并后的木头长度，记作joinArr。循环N-1次，每次找到这两个数组中最小的未进行整体合并的两个数，相加加入到joinArr中。

后者虽然空间上存储翻倍，但时间上效率更为高效，具体分析放在总结部分，因此我使用了第二种方法。

1. 用数学归纳法证明贪心策略的正确性

假设n截木头：。设花费为*W*。

1. 最开始，我们必定要合并两堆最小的，因为此时可以使花费最小（）。
2. 假设我们只剩下m堆了（m<n），且到此时花费仍为最小。那么我们仍需要合并两堆最小的，设新增花费为，则最小。

因此当合并成一堆时最小。即：最小。

# 代码结构设计

主要功能由以下两个函数实现：

void init(const int n, int\*& oriArr, int\*& joinArr)

int solve(const int num,int\*& oriArr, int\*& joinArr)

init函数，先用于对木头长度输入到oriArr数组中，并同时对joinArr数组全部初始化为INT\_MAX。然后对oriArr中木头长度进行升序排序，排序算法用的是三路划分快排。

solve函数用于不断合并木头，计算总费用。

# 具体实现

## 输入处理

由于是对数字进行逐一输入，且预先规定了木头总数量，直接循环cin即可，若输入非法字符或数字超出指定范围，则清空缓冲区重新输入。

## oriArr数组升序排序

由于初始数据中有较多的重复元素，而使用基本的快速排序效率十分低下。例如极端情况下，对于一个全部等值序列，基本的快速排序算法仍然要对序列进行划分直到得到最小序列。而三路划分快排可以满足对大量重复元素的排序，图1是排序过程中数组结构。



图 1 三路划分快排过程数组

三路划分快排的主体思路如下:

1. 分别从数组给定的区间左右两边向中间遍历，将左指针遍历到的，等于基准元素的划分到左等于区间，小于的就往后推；右指针遍历到的，等于基准元素的划分到右等于区间，大于的就往前推
2. 做完过程1，左右指针分别指向大于基准元素的和小于基准元素的数，交换左右指针即可再次循环上述过程，直到左右指针相遇。
3. 然后对调左等于区间和小于区间，对调右等于区间和大于区间。最后递归小于区间和大于区间。

三路划分快排不仅有效地处理了待排序元素中的重复值问题，而且在没有重复值时也能保持原有的性能。

具体实现代码如下：

1. **void** quickSort(**int**\* arr, **const** **int** left, **const** **int** right) {
2. **if** (right <= left) { //三路快排终止条件
3. **return**;
4. }
5. **int** p, q, i, j, pivot;
6. i = p = left;
7. j = q = right - 1;
8. pivot = arr[right];
9. **while** (**true**) {
10. //从左遍历，找到<=pivot的，将等于的归入左侧等于区间
11. **while** (i < right && arr[i] <= pivot) {
12. **if** (arr[i] == pivot) {
13. swap(arr, i, p++);
14. }
15. i++;
16. }
17. //从右遍历，找到>=pivot的，将等于的归入右侧等于区间
18. **while** (left <= j && arr[j] >= pivot) {
19. **if** (arr[j] == pivot) {
20. swap(arr, j, q--);
21. }
22. j--;
23. }
24. //这一趟遍历结束
25. **if** (i >= j)
26. **break**;
27. //此时，左边i指向的是大于pivot的项
28. //j指向的是小于pivot的项（亦即两while终止条件）
29. //现交换arr[i]和arr[j]，以便可再次遍历下去
30. swap(arr, i++, j--);
31. }
32. //调换左等于区间与小于区间
33. i--;
34. p--;
35. **while** (p >= left)
36. swap(arr, i--, p--);
37. //调换右等于区间与大于区间
38. j++;
39. q++;
40. **while** (q <= right)
41. swap(arr, j++, q++);
42. //递归小于区间和大于区间
43. quickSort(arr, left, i);
44. quickSort(arr, j, right);
45. }

## 合并木头

一共三个指针：指针i指向原始数据数组oriArr当前未合并的木头；指针j，对joinArr数组从下标0开始；指针k随for循环的进行而变动，表示合并一次的费用在joinArr中应存储的位置。用临时变量w记录每次合并花费的费用，对指针i和指针j指向的数进行两次比大小，小的加给w，相应指针自增，然后将w加给总费用变量，并在joinArr中指针k指向的位置赋值w。

以下表1和表2是输入样例的8根木头，长度分别为4 5 1 2 1 3 1 1的合并过程

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| oriArr（已排序） | | | | | | | |
| 0 | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| 1 | 1 | 1 | 1 | 2 | 3 | 4 | 5 |

表 1 原始数组

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| joinArr | | | | | | | cost |
| 0 | **1** | **2** | **3** | **4** | **5** | **6** |  |
| / | / | / | / | / | / | / | **0** |
| 2 | / | / | / | / | / | / | **2** |
| 2 | 2 | / | / | / | / | / | **4** |
| 2 | 2 | 4 | / | / | / | / | **8** |
| 2 | 2 | 4 | 5 | / | / | / | **13** |
| 2 | 2 | 4 | 5 | 8 | / | / | **21** |
| 2 | 2 | 4 | 5 | 8 | 10 | / | **31** |
| 2 | 2 | 4 | 5 | 8 | 10 | 18 | **49** |

表 2 合并数组变化过程

合并的函数代码如下：

1. **long** **long** **int** solve(**const** **int** num,**int**\*& oriArr, **int**\*& joinArr) {
2. **long** **long** **int**  sum = 0, w;
3. **int** i = 0, j = 0;
4. **for**(**int** k=1;k<num;k++){
5. **if** (oriArr[i] < joinArr[j])
6. w = oriArr[i++];
7. **else**
8. w = joinArr[j++];
9. **if** (oriArr[i] < joinArr[j])
10. w += oriArr[i++];
11. **else**
12. w += joinArr[j++];
13. joinArr[k-1] = w;
14. sum += w;
15. }
16. **return** sum;
17. }

# 项目小结

1. 性能分析

本题所用方法主要开销在对原数据进行排序和合并木头两个部分。内存开销上，总共用到了oriArr和joinArr两个数组，空间复杂度为；时间开销上，三路划分快排平均时间复杂度为，且此排序算法对含有大量重复值的序列排序效率会更高，而合并木头的过程仅对数组进行了一次遍历，因此时间复杂度为, 这种方法时间复杂度为，即。

而运用类似STL库中优先队列的做法，内存开销仅为初始数组，空间复杂度为，而建立大根堆的时间复杂度为，每次取根节点再调整和插入操作均为，总时间复杂度亦为，即。

1. 改进

两种方法没有明显区别，但如果进一步牺牲空间复杂度，将快速排序替换成时间复杂度为的桶排序，时间复杂度会降为，效率将大大提升，但随之而来的代价就是空间复杂度的大量增加。

若输入的数据有一个较为合理的区间，比如0~20000，和最大输入数量N的数量级一致，在本机上第一种方法耗时在区间3~6ms，而第二种将会在区间3~17ms，而二者内存消耗差别不大。图2和图3是10个测试点运行时间和内存使用情况。

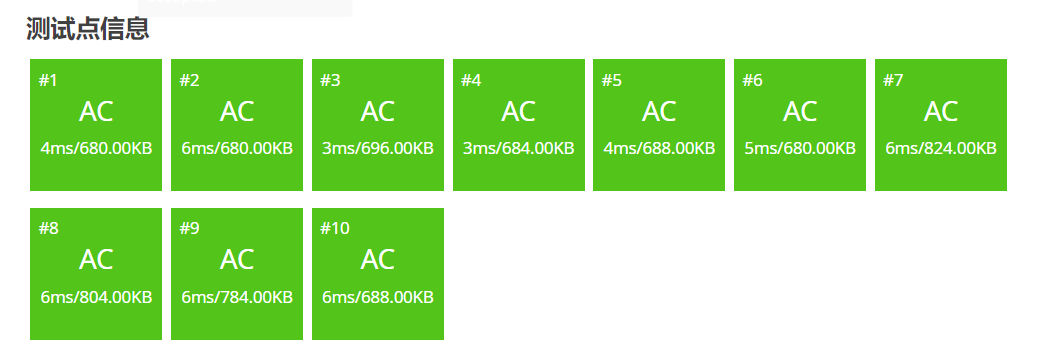


图 2 先排序后合并

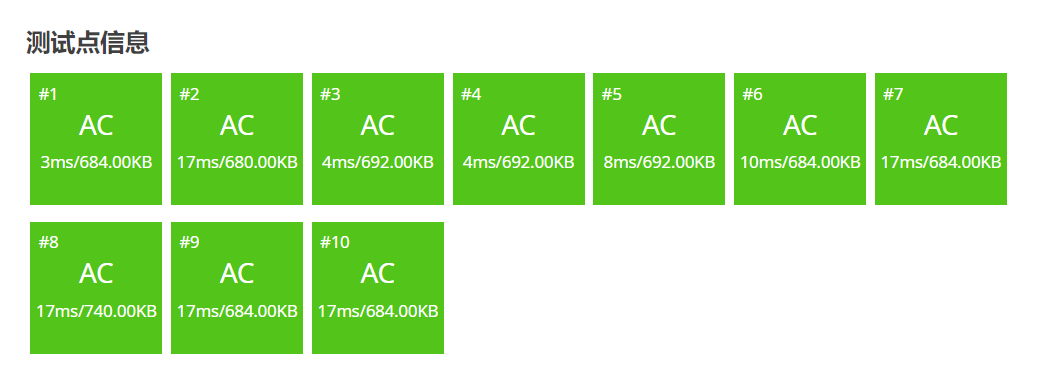
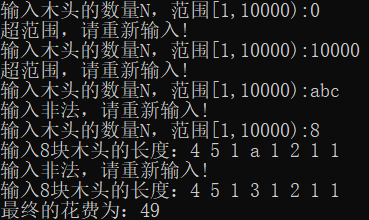


图 3 优先队列法

# 程序功能测试

错误输入测试：



将输入cin替换成rand随机生成，测试9999个数据

