

```

void merge_sort(int* arr, int l, int r){
    if(r - l <= 1)return;
    int mid = (l + r) / 2;
    merge_sort(arr, l, mid);
    merge_sort(arr, mid, r);
    int* temp = (int *)malloc(sizeof(int) * (r - l));
    int k = 0, p1 = l, p2 = mid;
    while(p1 < mid || p2 < r){
        if((p1 < mid) && (p2 >= r || arr[p1] <= arr[p2])){
            temp[k++] = arr[p1++];
        }else{
            temp[k++] = arr[p2++];
        }
    }
    for(int i = l; i < r; i++){
        arr[i] = temp[i - l];
    }
    return;
}

int main(){
    int* arr_s = getRandData(SMALL_DATA_N);
    int* arr_b = getRandData(BIG_DATA_N);
    TEST(merge_sort, arr_s, SMALL_DATA_N);
    TEST(merge_sort, arr_b, BIG_DATA_N);
    free(arr_b);
    free(arr_s);
    return 0;
}

```

我写的这个是归并排序吗？

是的，你实现的确实是归并排序（Merge Sort）算法。我来分析一下你的代码：

归并排序的关键特征：

分治策略：将数组不断二分，直到子数组长度为1（你的条件是 `r - l <= 1`）

递归排序：分别对左右两部分递归调用 `merge_sort`

合并操作：将两个已排序的子数组合并成一个有序数组（你的 `while` 循环部分）

你的实现细节：

使用 `(l + r) / 2` 计算中点

使用临时数组 `temp` 进行合并

合并时正确处理了边界情况（`p2 >= r` 时自动选择左边元素）

最后将临时数组拷贝回原数组

需要注意的几点：

内存泄漏：你 `malloc` 了临时数组 `temp` 但没有 `free` 它，应该在用完后就释放

边界条件：你的实现是左闭右开区间（`[l, r)`），这与大多数C标准库的惯例一致

稳定性：由于你使用了 `arr[p1] <= arr[p2]`，这个排序是稳定的（相等时不交换）

建议的修改：

c

复制 下载

```
void merge_sort(int* arr, int l, int r){
    if(r - l <= 1) return;
    int mid = (l + r) / 2;
    merge_sort(arr, l, mid);
    merge_sort(arr, mid, r);
    int* temp = (int *)malloc(sizeof(int) * (r - l));
    int k = 0, p1 = l, p2 = mid;
    while(p1 < mid || p2 < r){
        if((p1 < mid) && (p2 >= r || arr[p1] <= arr[p2])){
            temp[k++] = arr[p1++];
        }else{
            temp[k++] = arr[p2++];
        }
    }
    for(int i = l; i < r; i++){
        arr[i] = temp[i - l];
    }
    free(temp); // 添加这行避免内存泄漏
    return;
}
```

你的实现是正确的归并排序，时间复杂度是 $O(n \log n)$ ，空间复杂度是 $O(n)$ 。

```

void quick_sort_my_hoare(int* arr, int l, int r){ //我的hoare分区版本
    if(r - l <= 2){
        if(r - l <= 1)return; /*边界情况只需要这一句就好了，所有情况缩到最简
                                就是这样一种情况，但是不可以 r - l <= 1, 因为挖坑法和
                                hoare分区的实现不同，hoare分区要求要交换，它不存在
                                r - 1, 不会缩到 r == l 这种情况，除非原数组开始就为空*/
        if(arr[l] > arr[l + 1])swap(arr[l], arr[l + 1]);
        return;
    }

    int pivot = arr[l], sp = l - 1, bp = r;
    while(true){
        do{++sp;}while(arr[sp] < pivot);
        do{--bp;}while(arr[bp] > pivot);
        if(sp > bp)break;
        swap(arr[sp], arr[bp]);
    }
    quick_sort_my_hoare(arr, l, sp);
    quick_sort_my_hoare(arr, sp, r);
    return;
}

```

那为什么我测试到我的归并排序运行时间比以上这个快排hoare分区的时间要长，一样的数据量

你的观察是正确的：**在大多数情况下，快速排序（Hoare 分区）会比归并排序更快**，即使它们的时间复杂度都是 $O(n \log n)$ 。原因主要有以下几点：

1. 常数因子差异（Cache 友好性）

快速排序 是 **原地排序** (in-place)，它只需要 **$O(1)$ 额外空间**（递归栈除外），并且它的访问模式对 **CPU 缓存更友好**（局部性原理）。

归并排序 需要 **$O(n)$ 额外空间**（临时数组 `temp`），并且每次合并都要 **拷贝数据**，导致更多的内存访问开销。

为什么影响大？

现代 CPU 的缓存（L1/L2/L3）速度比主存快很多，**快速排序**的交换操作更可能在缓存中完成，而 **归并排序**需要频繁访问主存（临时数组和原数组来回拷贝），导致速度变慢。

2. 递归调用开销

快速排序 的递归深度通常是 $O(\log n)$ ，并且 **分区后的子问题大小不一定严格平衡**（但平均情况下仍然很快）。

归并排序 的递归深度也是 $O(\log n)$ ，但 **每次递归都要进行合并操作**，额外增加了拷贝数据的开销。

3. 你的 Hoare 分区实现可能比标准 Lomuto 更快

你的 Hoare 分区版本：

c

📄 复制 ⬇️ 下载

```
int pivot = arr[l], sp = l - 1, bp = r;
while(true){
    do{++sp;} while(arr[sp] < pivot);
    do{--bp;} while(arr[bp] > pivot);
    if(sp > bp) break;
    swap(arr[sp], arr[bp]);
}
```

Hoare 分区 通常比 **Lomuto 分区**（常见于教科书版本）**交换次数更少**，因为它从两端向中间扫描，减少了不必要的交换。

你的实现还 **提前处理了小数组**（`r - 1 <= 2`），减少了递归调用次数。

4. 测试数据的影响

归并排序 的时间复杂度 **严格 $O(n \log n)$** ，无论输入数据是否有序，它都稳定执行。

快速排序 在 **最好/平均情况下**是 $O(n \log n)$ ，但在 **最坏情况下**（如已排序数组）可能退化到 $O(n^2)$ 。

但你的 Hoare 分区版本 **对已排序数据的处理较好**（因为 `pivot = arr[1]` 且两端扫描，不会像 Lomuto 那样退化严重）。

如果你的测试数据是 **随机数据**，快速排序通常会更快；如果是 **部分有序数据**，你的 Hoare 版本仍然可能比归并排序快。

如何验证？

你可以：

测试不同数据分布（随机、升序、降序、重复数据）。

统计比较和交换次数（归并排序拷贝次数多，快速排序交换次数少）。

使用性能分析工具（如 `perf`、`gprof`）查看 CPU 缓存命中率和函数调用开销。

结论

你的观察是合理的：在相同数据量下，**快速排序（Hoare 分区）通常比归并排序更快**，因为：

更少的额外内存访问（归并排序需要临时数组）。

更好的缓存局部性（快速排序是原地交换）。

Hoare 分区优化（减少交换次数）。

归并排序的优势：

稳定排序（保持相等元素的相对顺序）。

最坏情况 $O(n \log n)$ （快速排序可能退化）。

如果你的目标是 **追求最高速度**，快速排序（特别是 Hoare 分区）通常是更好的选择。

如果你的数据 **几乎有序或包含大量重复元素**，可以考虑 **三路快速排序** 或 **内省排序（Introsort）**（结合快速排序+堆排序）。

归并排序 (Merge Sort) 可以和许多现实世界或计算机科学中的**分治策略**、**协作流程**进行类比。以下是几个生动的比喻：

1. 图书馆整理书籍（分治+合并）

问题：图书馆新到一批乱序的书籍，需要按编号排序。

归并排序的类比：

分治：将书堆分成两半，分别交给两个管理员整理（递归拆分）。

合并：两个管理员各自整理好后，合并两个有序的书堆：

比较两堆最顶部的书编号，取较小的放入新堆，重复直到所有书归位。

关键点：依赖“有序子序列”的合并，而非原地交换。

2. 多人合作拼图（协作式工作）

问题：拼一幅巨大的拼图，时间紧迫。

归并排序的类比：

分治：将拼图分成几块，分给不同小组独立完成（递归排序子数组）。

合并：小组完成后，将拼好的区块边缘对齐合并（合并有序子数组）。

关键点：合并时需要协调边界匹配（类似合并时的指针比较）。

3. 流水线生产（稳定有序性）

问题：工厂需要组装一批手机，按订单顺序出厂。

归并排序的类比：

分治：将订单拆分成小批次，在不同车间并行组装（子数组排序）。

合并：各车间完成后，按订单号顺序合并成最终队列（稳定合并）。

关键点：保持原始顺序（稳定性），适合需要保序的场景。

4. DNA序列比对（分治+精确合并）

问题：比对两段长DNA序列的相似性。

归并排序的类比：

分治：将DNA序列分段处理（递归分解问题）。

合并：将分段比对结果合并成全局比对（合并有序数据）。

关键点：分治降低复杂度，合并保证全局一致性。

5. Git分支合并（版本控制）

问题：多个开发者并行开发后合并代码。

归并排序的类比：

分治：每个开发者在独立分支上工作（子数组排序）。

合并：通过比较时间戳或版本号合并提交（合并有序变更）。

关键点：冲突解决类似合并时的元素比较。

为什么这些类比合适？

分治思想：将大问题分解为小问题独立解决。

合并操作：需要协调子问题的解，而非直接交换（区别于快速排序的“竞争”式分区）。

稳定性：保持原始相对顺序（如拼图边缘对齐、订单顺序不变）。

对比快速排序的类比

快速排序更像体育比赛淘汰制：

选择一个“基准”（冠军候选），将其他人分成“更强”和“更弱”两组（分区），递归淘汰。

不稳定：比赛名次可能因分组方式变化（非保序）。

而归并排序像协作组装，强调**有序合并**而非竞争淘汰。

基数排序 (Radix Sort) 的时间复杂度取决于数据的位数 (d) 和每一位的基数范围 (k , 通常为10或256等), 其算法复杂度为:

时间复杂度

最优/平均/最坏情况:

$$O(d \cdot (n + k))$$

n : 待排序元素个数

d : 数字的最大位数 (或字符串的最大长度)

k : 每一位的可能取值数 (如十进制数字 $k = 10$, ASCII字符 $k = 256$)

空间复杂度

$$O(n + k)$$

需要额外的空间存储:

桶 (Buckets): 大小为 k 的计数数组或列表。

临时数组: 大小为 n 的中间结果 (某些实现中)。

关键细节

适用场景:

数据位数 d 较小且范围明确 (如整数、定长字符串)。

若 d 很大 (如超长数字), 效率会下降。

与比较排序的区别:

基数排序是**非比较排序**, 突破了比较排序的 $\Omega(n \log n)$ 下限。

实际效率可能优于 $O(n \log n)$ 的排序 (如快速排序、归并排序), 当 $d \ll \log n$ 时。

稳定性:

基数排序是**稳定排序** (保持相同键值的原始顺序)。

示例分析

排序 1,000,000 个 32 位整数：

$d = 32$ （假设按二进制位处理）， $k = 2$ （0和1）。

时间复杂度： $O(32 \cdot (10^6 + 2)) \approx O(32 \cdot 10^6)$ 。

实际中常按十进制位分组（ $d = 10$, $k = 10$ ），则 $O(10 \cdot (10^6 + 10))$ 。

对比快速排序：

快速排序平均 $O(10^6 \log 10^6) \approx O(20 \cdot 10^6)$ 。

基数排序可能更快（因常数因子较小）。

常见变种

LSD (Least Significant Digit) :

从最低位开始排序，适合整数和定长字符串。

MSD (Most Significant Digit) :

从最高位开始，类似字典序，适合变长字符串（可能需要递归）。

总结

基数排序在**数据范围有限且位数较少时**非常高效，但空间开销较大。其性能优势关键取决于 d 和 k 与 n 的关系。

① 本回答由 AI 生成，内容仅供参考，请仔细甄别。