```
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

```c
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，所有的情况缩到最简
                                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

```c
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 - l <= 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[l]` 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）。