实验 7 人体快速排序计算机

B3 小组

人体计算机的组成（成员分工）

- 控制器：施家鑫
- 计数器：宋祎涵
- 监督器：秦硕
- 监控器：葛天行
- 数据组：薛宇，颜煜，郑嘉诚，徐睿，张璟浩，周星宇，臧文商，秦吉祥，秦宇彤，李怡乐，刘雨杭，郭一凡，李想，黄涵茜，宋若森，凌淞茗，周昊铖，赵莘萌，赵阳博卿，吴涵，赵承刚，张智博

指令集

指令	操作数	描述
PARTITION	start_index, end_index	**开始**对范围内元素进行排序操作
SELECT_PIVOT	index	**指定索引**为 index**的同学作**为本次分区的基准（Pivot）
COMPARE	index_A, pivot	**命令索引** A **和基准**值进行身高比较。
SWAP	index_A, index_B	**命令索引** A **和** B**的同学立刻交**换彼此的位置。

快速排序代码Ver 2.1

```
int COMPARE(index, pivot) {
        return s[index]-pivot;
}
func SELECT_PIVOT(index) {
        pivot=s[index];
}
func SWAP(index_A, index_B) {
        temp=s[index_A];
        s[index_A]=s[index_B];
        s[index_B]=temp;
}
```

```
void main() {
        PARTITION(0,21);
        // 共22名数据组成员
}
```

```
func PARTITION(start_index, end_index) {
        if((end_index-start_index)<=0) return;
        if((end_index-start_index)==1) {
                if(COMPARE(start_index,s[end_index])>0)
                        SWAP(start_index, end_index);
                return;
        }
        L=start_index, R=end_index;
        index=rand(start_index, end_index);
        // 在范围内随机选基准元素
        SELECT_PIVOT(index);
        while(L<=R) {
                for(;(L<=R)&&(COMPARE(L,pivot)<=0);L++)
                // 此时，满足L=R或者找到L位置上的数值>基准值
                for(;(L<=R)&&(COMPARE(R,pivot)>0;R--)
                // 此时，满足L=R或者找到R位置上的数值<=基准值
                SWAP(L,R); // 交换L，R坐标上的元素数据
        }
        PARTITION(start_index, R);      // 对左侧子数组排序
        PARTITION(L,end_index);         // 对右侧子数组排序
}
```

- 执行效果：总计196步

遇到问题1：

没有清楚理解**“控制器”**具体应该执行的职责，在执行排序操作时，将比较等**思考过程**（应当在大脑中执行的操作）讲出，导致**效率低下**，控制器执行了太多的**冗余**工作。

在王老师指出该问题之后，**“控制器”**理解清楚问题所在，并重新梳理清楚了自己应当执行的所有操作内容，在之后的排序操作中**避免**了**冗余**内容的出现，对**“控制器”**这一角色有了进一步加深且明确的理解。

遇到问题2：

按照最初版本的排序代码，排序时，若无法在“基准”两侧找到符合交换条件的两个数据，则将**单独**的**需要调整位置**的数据元素移至“基准”另一侧的**外侧新位置**上，导致整体数据空间占用严重增大（具体表现为：在操场上队伍断开且**延伸过长**，造成了空间的**浪费**）

解决：

在助教老师的提醒之下，组长对快速排序代码进行重新思考，改正了最初版本排序代码的错误/不恰当之处，重新书写了排序代码的逻辑，在使快速排序操作变得高效的同时，**避免**了**空间**的**浪费**。

快速排序代码Ver 2.2

```
int COMPARE(index, pivot) {
    return s[index]-pivot;
}
func SELECT_PIVOT(index) {
    pivot=s[index];
}
func SWAP(index_A, index_B) {
    temp=s[index_A];
    s[index_A]=s[index_B];
    s[index_B]=temp;
}
```

```
void main() {
    PARTITION(0,21);
    // 共22名数据组成员
}
```

```
func PARTITION(start_index, end_index) {
    if (start_index >= end_index) return;
    // 选择基准并放到开始位置
    index = rand(start_index, end_index);
    SWAP(start_index, index);
    SELECT_PIVOT(start_index);
    L = start_index + 1;
    R = end_index;
    while (L <= R) {
        while (L <= R && COMPARE(L, pivot) <= 0) L++;
        while (L <= R && COMPARE(R, pivot) > 0) R--;
        if (L < R) {
            SWAP(L, R);
            L++; R--;
        }
    }
    // 将基准放到正确位置
    SWAP(start_index, R);
    PARTITION(start_index, R-1);
    PARTITION(R+1, end_index);
}
```

快速排序代码Ver 3.1

```
void SWAP(index_A, index_B) {
    int t = s[index_A];
    s[index_A] = s[index_B];
    s[index_B] = t;
}
int PARTITION(int start, int end) {
    int pivot = s[start];
    int L = start, R = end;
    while (1) {
        while (s[L] < pivot) L++;
        while (s[R] > pivot) R--;
        if (L >= R) return R;
        SWAP(L, R);
        L++;
        R--;
    }
}
```

```
void QUICK_SORT(int start, int end) {
    if (start < end) {
        // 随机化：将随机元素与第一个交换，再分区
        int random_index = start + rand() % (end - start + 1);
        SWAP(random_index, start);
        int pivot_index = PARTITION(start, end);
        QUICK_SORT(start, pivot_index);
        QUICK_SORT(pivot_index + 1, end);
    }
}

void main() {
    QUICK_SORT(0, 21); // 对22个元素排序
}
```

快速排序代码Ver 3.1

```
函数 分割(起点, 终点):
    基准值 = 数组s[起点]
    左指针 L = 起点
    右指针 R = 终点
    循环执行（无限循环）：
        当 数组s[L] 小于 基准值 时：
            将 L 加 1
        当 数组s[R] 大于 基准值 时：
            将 R 减 1
        如果 L 大于或等于 R：
            返回 R 作为分界位置
        调用 交换(L, R)
        将 L 加 1
        将 R 减 1
结束函数
```

```
函数 交换(位置A, 位置B):
    临时变量 t = 数组s[位置A]
    将 数组s[位置A] 设为 数组s[位置B]
    将 数组s[位置B] 设为 t
结束函数

过程 快速排序(起点, 终点):
    如果 起点 小于 终点：
        随机选一个位置 随机下标
        调用 交换(随机下标, 起点)   // 随机化基准
        基准下标 = 分割(起点, 终点)
        调用 快速排序(起点, 基准下标)
        调用 快速排序(基准下标+ 1, 终点)
结束过程

主程序:
    调用 快速排序(0, 21)   // 对下标 0~21 共 22 个元素进行排序
结束程序
```

- 执行器1执行效果：总计197步

- 执行器2执行效果：总计131步

排序前

排序后

- 如何确保三个正确性？

- 结果正确性 最终验证: 排序完成后，“控制器”会从队首（身高最低）到队尾（身高最高）逐一进行相邻比较。
- 算法正确性 设置的“监督器”被授予最高权限，一旦发现指令与算法不符，可以立即喊“停”，保证执行过程的算法正确。使用**不同的执行器**仍能达到**正确**的结果，说明了算法的正确性。
- 系统正确性 唯一的指令源: “控制器”是**唯一**的指令发布者，“数据组”成员被严格要求只能被动地接收并执行来自“控制器”的指令，无法自行操作；所有的指令集按照**串行顺序**，“控制器”必须在一个指令被完全执行后，才能发布下一个指令，从而在宏观上保证了整个系统的串行执行。

- 实验过程出现了哪些意想不到的情况？如何应对？

- 在最初的实验中，我们为每个人按照初始顺序编号，为了方便，在两人交换时，同时交换彼此的序号，但是我们发现，这样的方式对于执行器与数据组来说都造成了不小的负担——需要从头数当前数据是第几位，地址编号难以记忆。
- 我提出了两大类方案：一类是每个人手持编号，在交换时同时互相交换手中的编号纸条；另一类是在从操场的地面上做标记，方便控制器清楚地知道每个数据所在的位置。经过讨论，我们一致认为第二类更加符合计算机内地址对应数据的逻辑，因此在第二节实验课时准备了几个水杯作为分区标记物，用水杯进行间隔，将数据区域分为5个区域（A～E），每个区域内有5个空位地址（1～5），由此，我们的数组地址明确且不可变，在程序执行过程中，变化的是地址对应的不同数据，方便了执行器的命令发送。

- 计算机为什么应该有寄存器？

- 在进行人体快速排序实验的过程中，按照我们的代码，需要记录L、R两个变量（左右指针）的值，对于单一的执行器来说，难以同时实现，因此，需要使用寄存器记录L、R的位置，方便执行器专注于比较身高，判断交换等命令。
- 另外，我们的快速排序代码在会循环调用，并且将整片数据一分为二地进一步操作，然而由于串行顺序，不能对两片数据同时进行更加细化的操作，只能先做完左侧，再对右侧进行操作，诸如此类的“深度查找”过程中，需要寄存器记下还在“队列”中等待的右侧数据范围。

- 为什么会出现执行次数不同？
  快速排序算法为什么要有随机选择？

```
// 随机化：将随机元素与第一个交换，再分区
int random_index = start + rand() % (end - start + 1);
SWAP(random_index, start);
```

- 在对每一片数据进行快速排序的开始时，为了提高排序算法的效率，加入了**随机**，每次随机选择一个元素作为基准，并且放到最左侧的位置。造成了同一算法，同一数据但是执行次数不同的情况。
- 快速排序的平均时间复杂度是优秀的 $O(n \log n)$，但它的最坏情况时间复杂度是 $O(n^2)$。这个最坏情况通常发生在分区极其不平衡的时候。通过随机选择一个元素作为基准，我们极大地降低了连续多次都选中最差基准（最大或最小值）的概率。我们用一个很小的**随机成本**，换取了算法整体的**稳定性**和**高效性**。