## 1. 题目分析

比较十种不同的内排序算法

## 2. 程序设计

十种内排序算法

## 3. 测试结果

| | 第一次（10） | | 第二次（50） | | 第一次（250） | | 第一次（1250） | | 第一次（6250） | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 比较次数 | 移动次数 | 比较次数 | 移动次数 | 比较次数 | 移动次数 | 比较次数 | 移动次数 | 比较次数 | 移动次数 |
| 冒泡排序 | 45 | 22 | 1225 | 217 | 31125 | 1651 | 780625 | 11064 | 19528125 | 69762 |
| 简单选择排序 | 72 | 114 | 1923 | 2121 | 44802 | 44364 | 1179612 | 1098558 | 28776471 | 27299598 |
| 快速排序 | 55 | 24 | 1275 | 641 | 31375 | 14934 | 781875 | 393204 | 19534375 | 192157 |
| 归并排序 | 55 | 42 | 332 | 739 | 1972 | 15432 | 11831 | 395702 | 70065 | 104655 |
| 简单插入排序 | 26 | 10 | 285 | 50 | 2003 | 33250 | 1214506 | 123250 | 9238022 | 12136250 |
| 折半插入排序 | 66 | 300 | 791 | 1500 | 5673 | 7500 | 41842 | 37500 | 285714 | 187500 |
| 二路插入排序 | 10 | 20 | 59 | 145 | 314 | 1332 | 1568 | 11908 | 117615 | 1289281 |
| 基数排序 | 68 | 50 | 572 | 429 | 3988 | 3474 | 25904 | 26136 | 158616 | 179449 |
| 希尔排序 | 30 | 64 | 685 | 542 | 15178 | 3828 | 394445 | 24824 | 9598397 | 153302 |
| 堆排序 | 72 | 81 | 1923 | 738 | 44802 | 5367 | 1179612 | 35361 | 28776471 | 220578 |

## 4. 用户使用说明

无需任何操作，自动测试

## 5. 附录

```
#include <algorithm>

#include <iostream>

#include <random>

#include <vector>

using namespace std;



// 关键字参加的交换次数

long long comparison_times = 0;



// 关键字参加的移动次数

long long move_times = 0;
```

```
// 随机数生成器，生成 int 范围内的整数
std::random_device seed;
std::default_random_engine engine{seed()};
std::uniform_int_distribution<int> dis(INT_MIN, INT_MAX);

// 排序算法 - 冒泡排序,
// 通过相邻元素的比较和交换,在每一轮使一个元素移动到正确的位置。
// 时间复杂度 O(n^2)
// 总体来说,选择排序每次交换都是使得最小值移至最前,效率略高一点。冒泡排序每次比较都可能发生交换,效率略低。
void bubbling_sort(vector<int> &arr) {
    for (int i = 0; i < arr.size(); ++i) {
        for (int j = 1; j < arr.size() - i; ++j) {
            ++comparison_times;
            if (arr[j] < arr[j - 1]) {
                swap(arr[j], arr[j - 1]);
                move_times += 3;
            }
        }
    }
}
```

```
// 排序算法 - 简单选择排序
// 每一轮从未排序的元素中选出最小的元素,使其移动到已排序的序列的末尾。
// 时间复杂度 O(n^2)
void simple_selection_sort(vector<int> &arr) {
    for (int i = 0; i < arr.size(); ++i) {
        ++move_times;
        int min_index = i;
        for (int j = i; j < arr.size(); ++j) {
            ++comparison_times;
            if (arr[min_index] > arr[j]) {
                min_index = j;
                ++move_times;
            }
        }
        swap(arr[min_index], arr[i]);
        move_times += 3;
    }
}

// 排序算法 - 快速排序算法
void quick_sort(vector<int> &arr, int left, int right) {
```

```
    // 递归终止条件:区间只有 0 或 1 个元素,已然有序,无需继续划分
    if (right - left < 1) {
        return;
    }

    // 随机选取一个数作为基准数 Pivot
    int index = left + rand() % (right - left + 1);
    int pivot = arr[index];
    ++move_times;

    // 初始化左右指针 lt 和 gt,cnt 用于遍历,lt 表示小于 pivot 的最后一个元素,gt 表示大于 pivot 的第一个元素
    int lt = left;
    int gt = right;
    int cnt = left;

    // 遍历数组,进行三向切分
    while (cnt <= gt) {
        // 当前元素小于 pivot,则交换至左指针 lt 处,lt 和 cnt 同时右移
        ++comparison_times;
        if (arr[cnt] < pivot) {
            swap(arr[cnt++], arr[lt++]);
```

```
            move_times += 3;
        }
        // 当前元素大于 pivot,则交换至右指针 gt 处,gt 左移
        else if (arr[cnt] > pivot) {
            swap(arr[gt--], arr[cnt]);
            move_times += 3;
        }
        // 等于 pivot,直接跳过
        else {
            ++cnt;
        }
    }

    // 递归调用,继续对左右两部分进行快速排序
    quick_sort(arr, left, lt - 1);
    quick_sort(arr, gt + 1, right);
}

// 排序算法 - 归并排序：通过递归将数组划分为两部分,排序后合并得到最终结果。
void merge_sort(vector<int> &tmp, vector<int> &arr, int left, int right) {
    if (right - left < 1) {  // 递归终止条件,子数组长度为 1
        return;
```

```
}

int mid = left + ((right - left) >> 1);  // 取中间索引

merge_sort(tmp, arr, left, mid);          // 对左半部分排序

merge_sort(tmp, arr, mid + 1, right);     // 对右半部分排序


int l = left, r = mid + 1, k = 0;  // 初始化变量

while (l <= left && r <= right) {  // 双指针,取较小者

    ++comparison_times;

    ++move_times;

    if (arr[l] < arr[r]) {

        tmp[k++] = arr[l++];  // 将较小值放入 tmp,指针后移

    } else {

        tmp[k++] = arr[r++];

    }

}

while (l <= mid) {  // 将左半部分剩余元素放入 tmp

    ++move_times;

    tmp[k++] = arr[l++];

}

while (r <= right) {  // 将右半部分剩余元素放入 tmp

    ++move_times;

    tmp[k++] = arr[r++];
```

```
    }
    copy(tmp.begin(), tmp.begin() + (right - left + 1),
         arr.begin() + left);  // 将 tmp 拷贝回 arr
    move_times += right - left + 1;
}

// 排序算法 - 简单插入排序
void simple_insertion_sort(vector<int> &arr) {
    for (int i = 1; i < arr.size(); ++i) {
        for (int j = i - 1; j >= 0; --j) {
            ++comparison_times;
            if (arr[j + 1] < arr[j]) {
                move_times += 3;
                swap(arr[j + 1], arr[j]);
            } else {
                break;
            }
        }
    }
}

// 排序算法 - 折半插入排序
```

```
void half_insert_sort(vector<int> &arr) {
    for (int i = 1; i < arr.size(); ++i) {
        int left = 0;
        int right = i - 1;
        while (left <= right) {
            int mid = left + ((right - left) >> 1);
            ++comparison_times;
            if (arr[mid] > arr[i]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        for (int k = i; k >= left; --k) {
            swap(arr[left], arr[k]);
            move_times += 3;
        }
    }
}

// 排序算法 - 二路插入排序
```

```
// 将数组分成已排序和未排序两部分,每次从未排序的部分找一个最小的元素插入到已排序的部分中。
// 时间复杂度 O(n^2)
void two_way_insertion_sort(vector<int> &arr) {
    for (int i = 1; i < arr.size(); ++i) {
        ++move_times;
        int key = arr[i];
        int j = i - 1;
        // 找到已排序部分第一个大于等于 key 的元素,并记录其索引
        while (j >= 0 && arr[j] > key && (++comparison_times)) {
            ++move_times;
            arr[j + 1] = arr[j];
            j--;
        }
        // 在已排序部分的正确位置插入 key
        arr[j + 1] = key;
        ++move_times;
    }
}

// 排序算法 - 基数排序
// 针对每一位数进行排序,从低位到高位逐渐排序,实现整体有序。要求元素的表示形式从低位到高位是有意义的。
```

```
// 时间复杂度 O(n*k),其中 k 是排序位数。
void radix_sort(vector<int> &arr) {
    // 获取最大数,确定排序位数
    int max_num = *max_element(arr.begin(), arr.end());
    comparison_times += arr.size();

    int num_digits = 0;
    while (max_num > 0) {
        max_num /= 10;
        num_digits++;
    }

    // 设置 10 个桶
    vector<vector<int>> buckets(10);
    // 按位排序,从个位开始
    for (int pos = 0; pos < num_digits; pos++) {
        // 将所有整数按指定位数放入桶中
        for (int num : arr) {
            int digit = 0;
            buckets[digit].push_back(num);
            ++move_times;
        }
```

```
            // 按桶顺序输出
            move_times += arr.size();
            arr.clear();

            for (auto &bucket : buckets) {
                for (int num : bucket) {
                    ++move_times;
                    arr.push_back(num);
                }
                bucket.clear();
            }
        }
    }

// 排序算法 – 希尔排序
// 是插入排序的一种优化版本。它通过间隔为 h 的增量来比较并交换相隔 h 个元素,采用递减的 h 值,最终当 h=1 时,变成普通的插入排序。
// 时间复杂度 O(nlogn)
void shell_sort(vector<int> &arr) {
    int h = 1;
    while (h < arr.size() / 3) {
```

```
        h = 3 * h + 1;  // 确定初始步长 h
    }

    while (h >= 1) {
        for (int i = h; i < arr.size(); i++) {
            int j = i;
            int temp = arr[i];
            ++move_times;
            while (j >= h && arr[j - h] > temp && ++comparison_times) {
                ++move_times;
                arr[j] = arr[j - h];
                j -= h;
            }
            ++move_times;
            arr[j] = temp;
        }
        h /= 3;  // 步长缩小
    }
}

// 排序算法 - 堆排序,利用堆结构(可看成完全二叉树)的特点实现排序。
// 时间复杂度 O(nlogn)
```

```
void heapify(vector<int> &arr, int n, int i) {
    int largest = i;     // 目前最大值的索引
    int l = 2 * i + 1;  // 左子节点索引
    int r = 2 * i + 2;  // 右子节点索引

    comparison_times += 2;
    if (l < n && arr[l] > arr[largest]) largest = l;
    if (r < n && arr[r] > arr[largest]) largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        move_times += 3;
        heapify(arr, n, largest);
    }
}

void heap_sort(vector<int> &arr) {
    // 建立最大堆,将数组转换成最大堆
    for (int i = arr.size() / 2 - 1; i >= 0; i--) heapify(arr, arr.size(), i);

    // 交换根节点和最后一个节点,调整最大堆,重复此操作
    for (int i = arr.size() - 1; i >= 0; i--) {
        move_times += 3;
```

```
            swap(arr[0], arr[i]);
            heapify(arr, i, 0);
        }
    }


void restore_status(vector<int> &tmp, vector<int> &arr,
                    long long &comparison_times, long long &move_times) {
    tmp = arr;
    comparison_times = 0;
    move_times = 0;
    return;
}


void print_statistical_results(const long long &comparison_times,
                               const long long &move_times) {
    cout << comparison_times << endl << move_times << '\n';
}


int main() {
    // ios::sync_with_stdio(false);
    int step = 10;
    for (int i = 0; i < 5; ++i, step *= 5) {
```

```
vector<int> arr, tmp;
for (int j = 0; j < step; ++j) {
    arr.push_back(dis(engine));
}

// 冒泡排序
restore_status(tmp, arr, comparison_times, move_times);
bubbling_sort(tmp);
print_statistical_results(comparison_times, move_times);

// 简单选择排序
restore_status(tmp, arr, comparison_times, move_times);
simple_selection_sort(tmp);
print_statistical_results(comparison_times, move_times);

// 快速排序
restore_status(tmp, arr, comparison_times, move_times);
quick_sort(tmp, 0, tmp.size() - 1);
print_statistical_results(comparison_times, move_times);

// 归并排序
restore_status(tmp, arr, comparison_times, move_times);
```

```
vector<int> temporary(arr.size());
merge_sort(temporary, tmp, 0, arr.size() - 1);
print_statistical_results(comparison_times, move_times);

// 简单插入排序
restore_status(tmp, arr, comparison_times, move_times);
simple_insertion_sort(tmp);
print_statistical_results(comparison_times, move_times);

// 折半插入排序
restore_status(tmp, arr, comparison_times, move_times);
half_insert_sort(tmp);
print_statistical_results(comparison_times, move_times);

// 二路插入排序
restore_status(tmp, arr, comparison_times, move_times);
two_way_insertion_sort(tmp);
print_statistical_results(comparison_times, move_times);

// 基数排序
restore_status(tmp, arr, comparison_times, move_times);
radix_sort(tmp);
```

```
        print_statistical_results(comparison_times, move_times);


        // 希尔排序
        restore_status(tmp, arr, comparison_times, move_times);
        shell_sort(tmp);
        print_statistical_results(comparison_times, move_times);


        // 堆排序
        restore_status(tmp, arr, comparison_times, move_times);
        heap_sort(tmp);
        print_statistical_results(comparison_times, move_times);


        cout << endl;
    }
    cout << "end";
    return 0;
}
```