广州大学学生实验报告

开课实验室： 电子楼 416B　　2018 年 5 月 20 日

学院	计算机科学与网络工程学院	**年级、专业、班**	软件 171	**姓名**	谢金宏	**学号**	1706300001
实验课程名称	数据结构					**成绩**	
实验项目名称	实验四 排序算法					**指导老师**	杜娇

一、实验目的

加强对多种排序算法的理解。

二、使用仪器、器材

1. 使用 Windows 操作系统的微型计算机一台。
2. 符合 C++11 标准的 C++编程软件。

三、实验内容及原理

编程实现**插入排序**、**选择排序**、希尔排序、堆排序、**冒泡**、双向冒泡、**快速排序**、归并排序、基数排序。

1. 被排序的对象由计算机随机生成，长度分别取 20、100 和 500。
2. 算法中增加比较次数和移动次数的统计功能。
3. 对实验结果做比较分析。

插入排序 $O(n^2)$

将待排序的数列分为已有序和无序两个部分，重复地将无序数列的第一个元素与有序数列的元素从后往前逐个进行比较，找出插入位置，将该元素插入到有序数列的合适位置中，直到整个数列有序。

选择排序 $O(n^2)$

将待排序的元素分为已有序和无序两个部分，重复地在无序数列中选取最小的元素插入有序数列的最后一个元素之后，直到整个数组有序。

希尔排序（分组插入排序）$O(n^{1.3\sim2})$

将待排序的序列按步长进行分组，分别对每组进行插入排序。初始步长取数组长度的一半，重复分组排序，每次排序的步长取上次排序步长的一半，直到步长为 0。

堆排序 $O(n\log_n)$

利用最大堆的性质进行排序。
1. 建堆 从后往前遍历待排序的数列，将元素入堆。设法维护最大堆的性质。
2. 出堆 将堆顶元素出堆，插入出堆序列的第一个元素之前。设法维护最大堆的性质。重复出堆过程直至堆为空。

冒泡排序 $O(n^2)$

从前往后遍历数列，若序列的下一个元素小于前一个元素，则将这两个元素交换位置。重复此过程直到整个序列有序。

双向冒泡排序 $O(n^2)$

交替进行从前往后的冒泡排序和从后往前的冒泡排序。

快速排序 $O(n\times\log_n)$

在待排序的数列中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。

归并排序 $O(n\times\log_n)$

将待排序的数列分为两段，从两段中逐个选最小的元素移入新数据段的末尾。

基数排序 $O(数值位数\ k\times n)$

将整数按位数切割成不同的数字，然后按每个位数分别比较。

四、实验过程原始数据记录

实验使用的源代码和二进制文件如下：


Sorting.cpp


Sorting.exe

辅助定义

```
using SecondsElapsedSinceFire = double;  // 算法运行时间
using CompareCount            = double;  // 算法的比较次数
using MoveCount               = double;  // 算法的交换次数
using AlgorithmName           = string;

using AlgorithmResult = tuple<SecondsElapsedSinceFire, CompareCount, MoveCount>;
using Algorithm = AlgorithmResult (vector<int>&);
```

辅助函数

```
// 生成长度为 length 的随机数据
vector<int> GenerateNumbers(int length)
{
    vector<int> random_numbers;
    while (length--) {
        random_numbers.push_back(rand());  // 产生一个随机数
    }  // 产生 length 个随机数
    return random_numbers;
}

// 打印数组中的元素
void ShowNumbers(vector<int>& numbers)
{
    for (int i = 0; i < numbers.size(); ++i)
    {
        cout << numbers[i] << ' ';
    }
    cout << endl;
}
```

```
// 判断两个数组是否相同
bool VectorIsSame(vector<int>& v1, vector<int>& v2)
{
    if (v1.size() != v2.size()) return false;  // 如果数组的大小不同则返回
false
    bool is_same = true;
    for (unsigned i = 0; i < v1.size(); ++i) {
        if (v1[i] != v2[i]) {  // 依次比较数组中的每一个元素
            is_same = false; break;
        }
    }
    return is_same;
}
```

实现的算法列表

```
// 实现的算法的列表
vector<pair<AlgorithmName, Algorithm *>> algorithms {
    {"插入排序", InsertSort},
    {"选择排序", SelectionSort},
    {"希尔排序", ShellSort},
    {"大顶堆排序", HeapSort},
    {"冒泡排序", BubbleSort},
    {"双向冒泡排序", TwoWayBubbleSort},
    {"快速排序", QuickSort},
    {"归并排序", MergeSort},
    {"基数排序", RadixSort}
};
```

程序入口 main 函数

```
int main()
{
    RunTest(20);
    RunTest(100);
    RunTest(500);
}
```

测试函数

```cpp
// 用 array_length 长度的数组对排序算法进行测试
void RunTest(int test_array_length)
{
    // 打印测试结果的表头
    cout << "测试数组长度：" << test_array_length;
    cout << ' ' << "（每个算法运行 10 次取平均值）" << endl;
    cout << "排序算法\t 运行时间（秒）\t 比较次数\t 移动次数" << endl;
    cout << "==========================================================" <<
endl;

    // 取随机数种子
    long rand_seed = time(nullptr);

    // 逐个执行算法
    for (unsigned i = 0; i < algorithms.size(); ++i) {
        srand(rand_seed);
        AlgorithmResult algorithm_result = RunAlgorithm(algorithms[i].second,
test_array_length);
        PrintAlgorithmResult(algorithms[i].first, algorithm_result);  // 打印算法运行
结果
    }

    cout << endl;
}
```

```cpp
// 对排序函数进行测试
AlgorithmResult RunAlgorithm(Algorithm algorithm, int test_array_length)
{
    double total_seconds_elapesed_since_fire = 0,
           total_compare_count = 0,
           total_swap_count = 0;

    for (int i = 0; i < 10; ++i) {  // 测试 10 次取平均值
        auto random_numbers(GenerateNumbers(test_array_length));  // 生成随机数
        auto std_result(random_numbers);   // 保存标准排序结果的数组
        auto func_result(random_numbers);  // 保存函数排序结果的数组

        sort(std_result.begin(), std_result.end());  // 使用标准算法进行排序
        AlgorithmResult result = algorithm(func_result);  // 进行自定义算法排序

        assert(VectorIsSame(std_result, func_result));  // 比较自定义算法的结果与标准结果

        total_seconds_elapesed_since_fire += get<0>(result),
        total_compare_count += get<1>(result),
        total_swap_count += get<2>(result);
    }

    AlgorithmResult algorithm_result = make_tuple(
        total_seconds_elapesed_since_fire / 10,
        total_compare_count / 10,
        total_swap_count / 10
    );

    return algorithm_result;
}
```

```cpp
// 打印算法执行的结果
void PrintAlgorithmResult(AlgorithmName algorithm_name, AlgorithmResult
algorithm_result)
{
    cout << algorithm_name << '\t';
    cout << get<0>(algorithm_result) << "\t\t"
         << get<1>(algorithm_result) << "\t\t"
         << get<2>(algorithm_result);
    cout << endl;
}
```

插入排序

```cpp
AlgorithmResult InsertSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    int sorted_length = 1;  // 已经排序的数组的长度
    while (sorted_length < numbers.size())
    {
        ++compare_count;

        int next_number = numbers[sorted_length];  // 取出下一个待排序的数字
        int insert_location = sorted_length;  // 下一个待排序的数字应当插入的位置
        while (insert_location > 0 && next_number < numbers[insert_location - 1]) {
            swap(numbers[insert_location], numbers[insert_location-1]);  // 将插入位置
及之后的元素后移
            --insert_location;

            compare_count += 2, move_count += 1;
        }
        ++sorted_length;  // 已排序的长度增加 1
    }
    ++compare_count;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

选择排序

```
AlgorithmResult SelectionSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    unsigned sorted_length = 0;  // 已经排序的数组的长度
    while (sorted_length < numbers.size())
    {
        // 在待排序的数组中找到最小的元素的位置
        auto min_element_location = min_element(numbers.begin() + sorted_length,
numbers.end());
        swap(numbers[sorted_length], *min_element_location);  // 将最小的元素插入
        ++sorted_length;

        compare_count += numbers.end() - numbers.begin() + sorted_length - 1;
        move_count += 1;
    }
    ++compare_count;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

希尔排序（改进的插入排序，缩小增量排序 Diminishing Increment Sort）

```
AlgorithmResult ShellSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    int d = round(numbers.size() / 2);  // 排序
    while (d) {  // 当排序步长不为 0
        ++compare_count;

        for (int i = 0; i < d; ++i) {  // 分组进行插入排序
            ++compare_count;

            for (int j = d + i; j < numbers.size(); j += d)
            {
                int insert_location = j;
                int next_number = numbers[j];
                while (insert_location >= d && next_number < numbers[insert_location - d])
{
                    // 将插入位置及之后的元素后移
                    swap(numbers[insert_location], numbers[insert_location - d]);
                    insert_location -= d;
                    compare_count += 2, move_count += 1;
                }
                ++compare_count;
            }
            ++compare_count;
        }
        ++compare_count;

        d /= 2;  // 缩小步长
    }
    ++compare_count;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

堆排序（大顶堆排序）

```
AlgorithmResult HeapSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    // 算法开始时，将 numbers 视为二叉树的顺序储存数组（第一个结点为 numbers[0]）。

    auto get_left_child = [](int x){ return 2*x+1; };    // 获取 index 节点的左子节点索引
    auto get_right_child = [](int x){ return 2*x+2; };


    // 算法在堆顶添加节点，并使用 perc_down 将新加入的节点下沉，以维持大顶堆的性质。
    // 将 numbers 数组[top, tail)左闭右开区间内的节点视为堆的节点
    auto perc_down = [&](int top, int tail) {  // 将 index 位置的结点下沉，以维持大顶堆的
性质
        while (true) {  // 当左子树不为空
            int left_child = get_left_child(top);   // 左子树的索引
            int right_child = get_right_child(top);

            if (left_child >= tail) {  // 若左子树为空则退出循环
                break;
            }

            int bigger_child;  // 左右子树中更大的那一个的索引
            if (right_child < tail) {  // 当右子树不为空
                bigger_child = (numbers[left_child] > numbers[right_child] ? left_child :
right_child);

                compare_count += 2;
            }
            else {
                bigger_child = left_child;

                compare_count += 1;
            }
            if (numbers[top] < numbers[bigger_child]) {  // 当 head 节点值小于其的一个孩子
节点时
                swap(numbers[top], numbers[bigger_child]);  // 将 head 节点下沉
                top = bigger_child;
                ++move_count;
            }
            else break;
        }
        ++compare_count;
    };


    // 向堆顶添加节点
    for (int i = numbers.size() - 1; i >= 0; --i)
        perc_down(i, numbers.size());
    compare_count += numbers.size();

    // 将堆顶节点出堆
    for (int i = numbers.size() - 1; i > 0; --i)
    {
        swap(numbers[0], numbers[i]);
        perc_down(0, i);
    }
    compare_count += numbers.size() - 1;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

冒泡排序

```
AlgorithmResult BubbleSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    for (int bound = numbers.size(); bound > 1; --bound)
    {
        ++compare_count;

        bool hasMove = false;  // 在一次遍历过程中是否进行了移动
        for (int i = 0; i < bound - 1; ++i) {  // 从左向右冒泡
            ++compare_count;
            if (numbers[i] > numbers[i+1]) {
                swap(numbers[i], numbers[i+1]);  // 将较大元素后移
                hasMove = true, ++move_count;
            }
        }
        ++compare_count;

        if (!hasMove) {
            ++compare_count;
            break;
        }
        ++compare_count;
    }
    ++compare_count;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

双向冒泡排序

```
// 双向冒泡排序
AlgorithmResult TwoWayBubbleSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    int lbound = 0, rbound = numbers.size();
    while (lbound < rbound)  // 在[lbound, rbound)索引范围内进行排序
    {
        ++compare_count;

        bool hasMove = false;
        for (int i = 0; i < rbound - 1; ++i) {  // 从左向右冒泡
            ++compare_count;
            if (numbers[i] > numbers[i+1]) {
                swap(numbers[i], numbers[i+1]);  // 将较大元素后移
                hasMove = true, ++move_count;
            }
        }
        --rbound, ++compare_count;
```

```
    for (int i = rbound - 1; i > lbound; --i)  // 从右向左冒泡
    {
        ++compare_count;
        if (numbers[i-1] > numbers[i])  // 将较小元素前移
        {
            swap(numbers[i-1], numbers[i]);
            hasMove = true, ++move_count;
        }
        ++compare_count;
    }
    ++lbound, ++compare_count;

    if (!hasMove) {
        ++compare_count;
        break;
    }
    ++compare_count;
}
++compare_count;

clock_t finish_clock = clock();  // 结束计时
SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock -
begin_clock) / CLOCKS_PER_SEC;

return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

快速排序

```
AlgorithmResult QuickSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    // 在[left_bound, right_bound)进行快速排序
    function<void(int, int)> quick_sort = [&](int lbound, int rbound){
        if (lbound + 1 >= rbound) return;
        ++compare_count;

        int privot = numbers[lbound];  // 哨兵
        int i = lbound + 1,  // 第一个大于 privot 的数
            j = rbound - 1;  // 最后一个小于或等于 privot 的数
```

```
        while (i <= j)  // 将 numbers 划分不大于 privot 的左侧和大于的 privot 右侧
        {
            ++compare_count;

            while (i < rbound && numbers[i] <= privot) {  // 确保下标有效的方法最好是跟固定的值比较
                compare_count += 2;
                ++i;
            }  // 定位到左侧第一个大于 privot 的数
            while (j >= lbound && numbers[j] > privot) {
                compare_count += 2;
                --j;
            }  // 定位到右侧第一个小于或等于 priovt 的数
            if (i < j) {
                swap(numbers[i], numbers[j]);  // 交换这两个数
                ++compare_count, ++move_count;
            }
        }  // 循环退出后，numbers[lbound ... i-1(j)]必然不大于 privot，numbers[j+1(i) ... rbound)必然大于 privot。
        ++compare_count;

        swap(numbers[lbound], numbers[j]), move_count += 1;  // 交换 privot 与最后一个不大于 privot 的数

        quick_sort(lbound, j);  // 对左部进行快速排序
        quick_sort(i, rbound);  // 对右部进行快速排序
    };


    quick_sort(0, numbers.size());

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock - begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

归并排序

```
AlgorithmResult MergeSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    // 对[lbound, rbound)进行归并排序
    function<void(int, int)> merge_sort = [&](int lbound, int rbound) {
        if (lbound + 1 >= rbound) return;
        ++compare_count;

        int middle = lbound + (rbound - lbound) / 2;
        merge_sort(lbound, middle);
        merge_sort(middle, rbound);

        int i = lbound, j = middle;
        vector<int> middle_result(rbound - lbound);  // 用于存放中间结果的临时数组
        auto middle_result_iterator = middle_result.begin();
```

```
        while (i < middle && j < rbound)  // 在左右两侧数组尚未取出的元素中较小的那个复制到临时数组中
        {
            *(middle_result_iterator++) = numbers[i] <= numbers[j] ? numbers[i++] : numbers[j++];

            compare_count += 3, move_count += 1;
        }
        while (i < middle) {
            *(middle_result_iterator++) = numbers[i++];  // 复制左侧数组剩余的元素到临时数组中

            ++compare_count, move_count += 1;
        }
        while (j < rbound) {
            *(middle_result_iterator++) = numbers[j++];

            ++compare_count, move_count += 1;
        }

        copy(middle_result.begin(), middle_result.end(), numbers.begin() + lbound);
    };

    merge_sort(0, numbers.size());

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock - begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

基数排序（以 2 为基数）

```
AlgorithmResult RadixSort(vector<int> &numbers)
{
    CompareCount compare_count = 0;
    MoveCount move_count = 0;
    clock_t begin_clock = clock(); // 开始计时

    vector<int> buckets[2];
    for (int i = 0; i < 30; ++i)
    {
        buckets[0].clear(), buckets[1].clear();  // 清空桶

        int mask = 1<<i;  // 用于取第 i 位数字的掩码
        for (int j = 0; j < numbers.size(); ++j)
        {
            buckets[bool(numbers[j] & mask)].push_back(numbers[j]);  // 根据第 i 位数字为 0 或 1 来放入对应的桶中
        }
        compare_count += numbers.size();

        copy(buckets[0].begin(), buckets[0].end(), numbers.begin());
        copy(buckets[1].begin(), buckets[1].end(), numbers.begin() + buckets[0].size());
        move_count += 2 * numbers.size();
    }
    compare_count += 31;

    clock_t finish_clock = clock();  // 结束计时
    SecondsElapsedSinceFire secnods_elapsed_since_fire = double(finish_clock - begin_clock) / CLOCKS_PER_SEC;

    return AlgorithmResult{secnods_elapsed_since_fire, compare_count, move_count};
}
```

五、实验结果及分析

完成了本次实验的全部要求。

测试数组长度为 20，进行 10 次测试取平均值，各个算法的表现如下：

排序算法	运行时间（秒）	比较次数	移动次数
插入排序	0	222.6	101.3
选择排序	0	591	20

希尔排序	0	182.8	37.9
堆排序	0	191.5	51.9
冒泡排序	0	312.3	101.3
双向冒泡排序	0	312.3	101.3
快速排序	0	206.5	21.8
归并排序	0	234.8	88
基数排序	0	631	1200

测试数组长度为 100，进行 10 次测试取平均值，各个算法的表现如下：

排序算法	**运行时间（秒）**	**比较次数**	**移动次数**
插入排序	0	5117.4	2508.7
选择排序	0	14951	100
希尔排序	0	1541.6	415.8
堆排序	0	1422.1	481.8
冒泡排序	0	5191.4	2508.7
双向冒泡排序	0.0012	6316	2508.7
快速排序	0	1735	162
归并排序	0.0005	1851.8	672
基数排序	0.0002	3031	6000

测试数组长度为 500，进行 10 次测试取平均值，各个算法的表现如下：

排序算法	**运行时间（秒）**	**比较次数**	**移动次数**
插入排序	0.0021	124557	63038.7
选择排序	0.0037	374751	500
希尔排序	0.0004	10529.6	3009.3
堆排序	0.0004	9429.4	3533.7
冒泡排序	0.0031	125896	62028.7
双向冒泡排序	0.0037	151727	62028.7
快速排序	0.0002	12167	1068.3
归并排序	0.0037	12706.8	4488
基数排序	0.0010	15031	30000