排序算法实验报告

班级：2018211303 姓名：马嘉骥 学号：2018211149

算法目的

对于由n个整数组成的数组，将其按从小到大顺序排列。

算法分析

1. 插入排序InsertionSort

> 插入排序的灵感来源于扑克牌游戏，假设玩家当前手中的牌是按从小到大顺序排列的，当玩家抽到一张新牌时，会从牌组的一端开始逐个查找比较，直到找到某个位置的牌小于新牌，而下个位置的牌大于新牌，则将新牌插入这两个位置中间。
>
> 时间复杂度：维护两个旗标 i 和 j ，通过两层循环，每层对数组遍历一次。
>
> ```
> T(N)=O(N^2)
> ```
>
> 稳定性：稳定。

2. 归并排序MergeSort

> 归并排序利用了分治的思想，基本方法为：将两个长度为n的数组进行排序之后，将其合并排序成为一个长度为2n的有序的数组。
>
> 时间复杂度：
>
> ```
> T(N)=2T(N/2)+O(N)
> 递推得
> T(N)=(2^x)*T(N/2^x)+x*O(N) 因为递推最终规模为1，则有N=2^x，x=log(N)
> T(N)=N*T(1)+log(N)*O(N)
> T(N)=N+log(N)*O(N)
> T(N)=O(NlogN)
> ```
>
> 稳定性：稳定。

3. 快速排序QuickSort

> 快速排序也利用了分治的思想。取一个pivot元素，并且遍历一遍数组，将每个元素与pivot比较大小，以此将数组划分为pivot元素、元素都比pivot小的左数组、元素都比pivot大的右数组，三个部分。对左右两个数组递归地调用此方法，直到左右子数组长度为1，这样得到的序列就是从小到大有序的。
>
> 时间复杂度：

```
Worst-case:每次选择的pivot元素都是极值
T(N)=T(0)+T(N-1)+O(N)
T(N)=O(1)+T(N-1)+O(N)
T(N)=T(N-1)+O(N)
T(N)=O(N^2)

Almost-best-case:每次将数组划分为不平衡的两部分
T(N)=p*T(N)+(1-p)*T(N)+O(N)    (p>=1/2)
得c*N*log(1/p)(N)<=T(N)<=c*N*log(1/(1-p))(N)+O(N)
T(N)=O(NlogN)

Best-case:每次将数组均分为两部分
代入上式可得T(N)=O(NlogN)
```

稳定性：不稳定。

算法源码

1. Insertion sort

```
extern PerfCounter pc;

void InsertionSort(int *arr, long long SIZE) {
    for (long long i = 1; i < SIZE; i++) {
        int tmp = arr[i];
        long long j;
        for (j = i - 1; j >= 0 && arr[j] > tmp; j--) {
            arr[j + 1] = arr[j];
            pc.CountInc(1);
            pc.MoveInc(1);
        }
        arr[j + 1] = tmp;
    }
}
```

2. Merge sort

```
extern PerfCounter pc;

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void Merge(int *arr, long long left, long long mid, long long right) {
    long long i, j, k;
    long long n1 = mid - left + 1;
    long long n2 = right - mid;

    /* create temp arrays */
```

```
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    pc.MoveInc((long long) n1 + n2);

    /* Merge the temp arrays back into arr[left..right]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = left; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        pc.MoveInc(1);
        pc.CountInc(1);
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
        pc.MoveInc(1);
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
        pc.MoveInc(1);
    }
}

/* left is for left index and right is right index of the
   sub-array of arr to be sorted */
void MergePass(int *arr, long long left, long long right) {
    if (left < right) {
        // Same as (left+right)/2, but avoids overflow for
```

```
        // large left and h
        long long mid = left + (right - left) / 2;

        // Sort first and second halves
        MergePass(arr, left, mid);
        MergePass(arr, mid + 1, right);

        Merge(arr, left, mid, right);
    }
}

void MergeSort(int *arr, long long size) {
    MergePass(arr, 0, size - 1);
}
```

3. Quick sort

```
extern PerfCounter pc;

/*
 * util func, swap two elements
 */
void Swap(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
    pc.MoveInc(1); // move counter++
}

/*
 * This function takes last element as pivot, places the pivot element
 * at its correct position in sorted array, and places all smaller
 * (smaller than pivot) to left of pivot and all greater elements to right of
pivot
 */
long long Partition(int *arr, long long left, long long right) {
    // fixed pivot choice outperforms random pivot and avg pivot when the data
is completely random!
    /*
    default_random_engine randE;
    uniform_int_distribution<int> u(left, right);
    randE.seed(time(nullptr));
    int pivot = arr[u(randE)];
    */
    int pivot = arr[right];
    long long i = (left - 1); // Index of smaller element

    for (long long j = left; j <= right - 1; j++) {
```

```
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // increment index of smaller element
            Swap(arr[i], arr[j]);
            pc.CountInc(1); // compare counter++
        }
    }
    Swap(arr[i + 1], arr[right]);
    return (i + 1);
}

void QuickSort(int *arr, long long size, long long left, long long right) {
    if (left < right) {
        /* pi is partitioning index, arr[p] is now at right place */
        long long pi = Partition(arr, left, right);
        // Sort elements in the 2 partitions
        QuickSort(arr, size, left, pi - 1);
        QuickSort(arr, size, pi + 1, right);
    }
}
```

测试代码

1. 工具函数及对象

```
//生成一个随机数组，使用了C++ random库的 default_random_engine 和
uniform_int_distribution。
void RandListGen(int *arr, const long long size, int lower, int upper) {
    default_random_engine randE;
    uniform_int_distribution<int> u(lower, upper);
    randE.seed(time(nullptr));
    for (long long i = 0; i < size; i++) {
        arr[i] = u(randE);
    }
}

void PrintArray(int *arr, long long size) {
    for (long long i = 0; i < size; i++)
        cout << arr[i] << "\t";
    cout << endl;
}

//用于性能测试的计数器对象。
class PerfCounter {
private:
    long long Count;
    long long Move;
public:
```

```
    void CountInc(long long n) {
        Count += n;
    }

    void CountClear() {
        Count = 0;
    }

    long long GetCount() {
        return Count;
    }

    void MoveInc(long long n) {
        Move += n;
    }

    void MoveClear() {
        Move = 0;
    }

    long long GetMove() {
        return Move;
    }
};
```

2. 测试结果正确性

```
//正确性测试，调用算法进行排序之后，与C++STL的sort函数排序结果进行比较，验证正确性。
void CorrectnessTest(int *arr, const long long size, int method, int lower, int
upper) {
    RandListGen(arr, size, lower, upper);
    int *sortedArr = new int[size];
    for (long long i = 0; i < size; i++) {
        sortedArr[i] = arr[i];
    }
    cout << "Generated source array:" << endl;
    PrintArray(arr, size);
    switch (method) {
        case 0: {
            InsertionSort(sortedArr, size);
            cout << "Insertion sorted array:" << endl;
            break;
        }
        case 1: {
            MergeSort(sortedArr, size);
            cout << "Merge sorted array:" << endl;
            break;
        }
```

```
        case 2: {
            QuickSort(sortedArr, size, 0, size - 1);
            cout << "Quick sorted array:" << endl;
            break;
        }
    }
    PrintArray(sortedArr, size);

    //check if sort is correct
    int *correctArr = new int[size];
    for (long long i = 0; i < size; i++) {
        correctArr[i] = arr[i];
    }
    sort(correctArr,correctArr+size);
    int flag = 1;
    for (long long i = 0; i < size; i++) {
        if (sortedArr[i] != correctArr[i])
            flag = 0;
    }
    if (flag == 1)
        cout << "Sort OK." << endl << endl;
    else
        cout << "Sort failed." << endl << endl;
}
```

3. 测试性能

```
//测试性能，使用了C++的chrono库，获得最高精确到纳秒的计时。同时在排序算法每次进行比较或者移动元素时，会调用PerfCounter的CountInc()或MoveInc()方法，在排序结束时，输出该算法对数组元素比较和移动的次数。
void PerfTest(int *arr, long long size, int method, int lower, int upper) {
    RandListGen(arr, size, lower, upper);
    extern PerfCounter pc;
    int *sortedArr = new int[size];

    for (long long i = 0; i < size; i++) {
        sortedArr[i] = arr[i];
    }

    pc.CountClear();
    auto start = chrono::system_clock::now();
    switch (method) {
        case 0:
            InsertionSort(sortedArr, size);
            break;
        case 1:
            MergeSort(sortedArr, size);
            break;
```

```
        case 2:
            QuickSort(sortedArr, size, 0, size - 1);
            break;
    }
    auto end = chrono::system_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);

    switch (method) {
        case 0:
            cout << "Insertion sort finished" << endl;
            break;
        case 1:
            cout << "Merge sort finished" << endl;
            break;
        case 2:
            cout << "Quick sort finished" << endl;
            break;
    }
    cout << "Time elapsed: " << duration.count() << " us.\t" << endl;
    cout << "Compare times: " << pc.GetCount() << "\tMove times: " <<
pc.GetMove() << endl;
}
```

4. 主函数

```
int main() {

    long long size = 7;
    int *arr = new int[size];

    CorrectnessTest(arr, size, 0, -100, 100);
    CorrectnessTest(arr, size, 0, INT_MIN, INT_MIN);
    CorrectnessTest(arr, size, 0, INT_MAX, INT_MAX);
    CorrectnessTest(arr, size, 0, 0, 0);

    //sleep to refresh the random number generator engine's seed
    this_thread::sleep_for(chrono::milliseconds(1000));
    CorrectnessTest(arr, size, 1, -100, 100);
    CorrectnessTest(arr, size, 1, INT_MIN, INT_MIN);
    CorrectnessTest(arr, size, 1, INT_MAX, INT_MAX);
    CorrectnessTest(arr, size, 1, 0, 0);

    //sleep to refresh the random number generator engine's seed
    this_thread::sleep_for(chrono::milliseconds(1000));
    CorrectnessTest(arr, size, 2, -100, 100);
    CorrectnessTest(arr, size, 2, INT_MIN, INT_MIN);
    CorrectnessTest(arr, size, 2, INT_MAX, INT_MAX);
    CorrectnessTest(arr, size, 2, 0, 0);
```

```
    delete[] arr;
    long long size = 1000;
    int* arr = new int[size];
    cout << "Array size: " << size << endl;
    PerfTest(arr, size, 0, INT_MIN, INT_MAX);
    PerfTest(arr, size, 1, INT_MIN, INT_MAX);
    PerfTest(arr, size, 2, INT_MIN, INT_MAX);

    delete[] arr;
    size = 10000;
    arr = new int[size];
    cout << "Array size: " << size << endl;
    PerfTest(arr, size, 0, INT_MIN, INT_MAX);
    PerfTest(arr, size, 1, INT_MIN, INT_MAX);
    PerfTest(arr, size, 2, INT_MIN, INT_MAX);

    delete[] arr;
    size = 100000;
    arr = new int[size];
    cout << "Array size: " << size << endl;
    PerfTest(arr, size, 0, INT_MIN, INT_MAX);
    PerfTest(arr, size, 1, INT_MIN, INT_MAX);
    PerfTest(arr, size, 2, INT_MIN, INT_MAX);

    return 0;
}
```

运行结果

1. 正确性测试（黑白盒测试）

```
Generated source array:
-64 93  56  -35 19  -94 -66
Insertion sorted array:
-94 -66 -64 -35 19  56  93
Sort OK.

Generated source array:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Insertion sorted array:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Sort OK.

Generated source array:
```

```
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Insertion sorted array:
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Sort OK.

Generated source array:
0 0 0 0 0 0 0
Insertion sorted array:
0 0 0 0 0 0 0
Sort OK.

Generated source array:
79  -93 -74 -49 54  -70 -4
Merge sorted array:
-93 -74 -70 -49 -4  54  79
Sort OK.

Generated source array:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Merge sorted array:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Sort OK.

Generated source array:
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Merge sorted array:
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Sort OK.

Generated source array:
0 0 0 0 0 0 0
Merge sorted array:
0 0 0 0 0 0 0
Sort OK.

Generated source array:
-34 76  -22 52  -63 89  12
Quick sorted array:
-63 -34 -22 12  52  76  89
Sort OK.

Generated source array:
```

```
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Quick sorted array:
-2147483648 -2147483648 -2147483648 -2147483648 -2147483648 -2147483648
-2147483648
Sort OK.

Generated source array:
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Quick sorted array:
2147483647  2147483647  2147483647  2147483647  2147483647  2147483647
2147483647
Sort OK.

Generated source array:
0 0 0 0 0 0 0
Quick sorted array:
0 0 0 0 0 0 0
Sort OK.


Process finished with exit code 0
```

2. 性能测试

- 复杂度测试

```
Array size: 1000
Insertion sort finished
Time elapsed: 1448 us.
Compare times: 253291 Move times: 253291
Merge sort finished
Time elapsed: 189 us.
Compare times: 8716 Move times: 273243
Quick sort finished
Time elapsed: 111 us.
Compare times: 4811 Move times: 278723

Array size: 10000
Insertion sort finished
Time elapsed: 155269 us.
Compare times: 24826022 Move times: 25104745
Merge sort finished
Time elapsed: 2119 us.
Compare times: 120495 Move times: 25371977
Quick sort finished
Time elapsed: 1463 us.
Compare times: 79302  Move times: 25457948
```

```
Array size: 100000
Insertion sort finished
Time elapsed: 13204682 us.
Compare times: 2506426401 Move times: 2531884349
Merge sort finished
Time elapsed: 26092 us.
Compare times: 1536592  Move times: 2535222205
Quick sort finished
Time elapsed: 17288 us.
Compare times: 959320 Move times: 2536248271

Process finished with exit code 0
```

所得结果符合推导的时间复杂度。

- 大数据量测试

```
Array size: 100000000
Quick sort finished
Time elapsed: 28455195 us.
Compare times: 1747110384 Move times: 1813778369

STL sort finished
Time elapsed: 9416379 us.

Process finished with exit code 0
```

```
Array size: 1000000000
Quick sort finished
Time elapsed: 316450622 us.
Compare times: 18762093166  Move times: 19428758458

STL sort finished
Time elapsed: 106664898 us.

Process finished with exit code 0
```

macOS，Intel Core i5U@3.8GHz，快速排序完成1亿随机数据排序用时28.4秒，完成10亿随机数据排序用时316秒，用时约为STL sort函数的三倍，性能可以接受。

小结

本次实验学习了三种排序算法，其中最令我惊讶的是在快速排序的时间复杂度推导处，只需要把分组从1:0稍微改变为0.9:0.1，就能让时间复杂度从O(N^2)骤降为O(NlogN)，将原本对于大量数据没有使用价值的算法变为可以轻松处理上亿数据的算法。