

**实验报告**



**题目： 实验一 三种排序算法的设计与分析**

**班 级： 2022211304**

**学 号： 2022211119**

**姓 名： 赵宇鹏**

**学 院： 计算机学院（国家示范性软件学院）**

**2024年 10 月 27 日**

## 实验目的

· 理解分治法的策略，掌握基于递归的分治算法的实现方法；

· 掌握基于分治法的合并排序、快速排序的实现方法；

· 理解并掌握在渐进意义下的算法复杂性的评价方法；

· 掌握算法测试的基本流程。

## 实验内容

#### 1. 算法的设计与实现

设计并实现堆排序、归并排序（合并排序）、快速排序算法，通过比较三种排序算法在不同数据量的情况下所需的移动次数、比较次数，分析算法在最差情况、平均情况下的算法复杂度。

#### 2. 测试要求

设计测试数据集，编写测试程序，用于测试：

a) 正确性：所实现的三种算法的正确性；

b) 算法复杂性：三种排序算法中，设计测试数据集，评价各个算法在算法复杂性上的表现；（最差情况、平均情况）

c) 效率：在三种排序算法中，设计测试数据集，评价各个算法中比较的频率，移动的频率。

#### **3. 撰写评价报告**

结合实验结果，在理论上给予总结和评价三种排序算法在算法复杂性和效率上的表现。形成电子版实验报告。

## 算法实现

本次实验总共分为三个文件：

SortAlgorithm.h：各排序算法的实现

SortData.h：测试数据集生成函数的实现

test.cpp：测试文件，负责调用各排序算法模块进行测试并输出结果

#### 堆排序

使用堆调整函数对大根堆进行调整，确保父节点的值一定大于子节点，每次将根节点取出放在数组的末尾，依次往前插入，并不断调整堆保持其为大根堆直至排序完成，堆排序的最坏、平均时间复杂度均为O(nlogn)，具体代码实现如下：

//堆调整

void heapify(int \*a, int n, int i, long long &comparions, long long &moves) {

    int largest = i;

    int l = 2 \* i + 1;

    int r = 2 \* i + 2;

    comparions++;

    if(l < n) {

        comparions++;

        if(a[l] > a[largest]) {

            largest = l;

        }

    }

    comparions++;

    if(r < n) {

        comparions++;

        if(a[r] > a[largest]) {

            largest = r;

        }

    }

    comparions++;

    if(largest != i) {

        swap(a[i], a[largest]);

        moves++;

        heapify(a, n, largest, comparions, moves);

    }

}

//堆排序

void heap\_sort(int \*a, int n, long long &comparisons, long long &moves) {

    for (int i = n / 2 - 1; i >= 0; i--) {

        comparisons++;

        heapify(a, n, i, comparisons, moves);

    }

    for (int i = n - 1; i > 0; i--) {

        comparisons++;

        swap(a[0], a[i]);

        moves++;

        heapify(a, i, 0, comparisons, moves);

    }

}

#### 归并排序

基于分治策略，将数组分成左右两部分，分别排序后进行合并，不断分治直至两个元素的比较，然后根据大小来进行合并，归并排序的时间复杂度在最坏和平均情况下为 O(nlogn)，适用于大规模数据排序，具体代码实现如下：

void merge(int \*a, int l, int m, int r, long long &comparisons, long long &moves) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for(int i = 0; i < n1; i++) {

        comparisons++;

        moves++;

        L[i] = a[l + i];

    }

    for(int i = 0; i < n2; i++) {

        comparisons++;

        moves++;

        R[i] = a[m + 1 + i];

    }

    int i = 0, j = 0, k = l;

    while(i < n1 && j < n2) {

        comparisons += 2;

        comparisons++;

        if(L[i] <= R[j]) {

            a[k] = L[i];

            i++;

        } else {

            a[k] = R[j];

            j++;

        }

        moves++;

        k++;

    }

    while(i < n1) {

        comparisons++;

        a[k] = L[i];

        moves++;

        i++;

        k++;

    }

    while(j < n2) {

        comparisons++;

        a[k] = R[j];

        moves++;

        j++;

        k++;

    }

}

void merge\_sort(int \*a, int l, int r, long long &comparisons, long long &moves) {

    if(l < r) {

        comparisons++;

        int m = l + (r - l) / 2;

        merge\_sort(a, l, m, comparisons, moves);

        merge\_sort(a, m + 1, r, comparisons, moves);

        merge(a, l, m, r, comparisons, moves);

    }

}

#### 快速排序

在本次实验中，为方便形成对比，我实现了三个快排程序，分别是选取数组随机元素作为基准点的快速排序，具有稳定性的三路快排以及通过选择数组首个元素，中间元素，末尾元素中的中位数来作为基准点的快速排序算法，通过分治策略将数组通过基准点被分割开，小的在基准点左侧，大的在右侧以实现排序，快排的平均时间复杂度为O(nlogn)，最差情况下为O(n2)，在我实现的这三类快排通过采取随机基准点的方法，三数取中位数的方法尽量避免了最坏情况的发生，

普通快排实现如下：

int partition(int \*a, int low, int high, long long &comparisons, long long &moves) {

    random\_device rd;

    mt19937 gen(rd());

    uniform\_int\_distribution<> dist(low, high);

    int randomIndex = dist(gen);

    swap(a[randomIndex], a[high]);

    moves++;

    int pivot = a[high];

    int i = low - 1;

    for(int j = low; j < high; j++) {

        comparisons += 2;

        if(a[j] < pivot) {

            i++;

            swap(a[i], a[j]);

            moves++;

        }

    }

    swap(a[i + 1], a[high]);

    moves++;

    return i + 1;

}

void quick\_sort(int \*a, int low, int high, long long &comparisons, long long &moves) {

    if(low < high) {

        comparisons++;

        int pi = partition(a, low, high, comparisons, moves);

        quick\_sort(a, low, pi - 1, comparisons, moves);

        quick\_sort(a, pi + 1, high, comparisons, moves);

    }

}

三路快排实现如下：

//三路快排

void quick\_sort\_3way(int \*a, int low, int high, long long &comparisons, long long &moves) {

    if(low >= high) {

        comparisons++;

        return;

    }

    int lt = low, gt = high;

    int i = low;

    random\_device rd;

    mt19937 gen(rd());

    uniform\_int\_distribution<> dist(low, high);

    int randomIndex = dist(gen);

    swap(a[randomIndex], a[high]);

    moves++;

    int pivot = a[high];

    while(i <= gt) {

        comparisons += 2;

        if(a[i] < pivot) {

            swap(a[lt], a[i]);

            moves++;

            lt++;

            i++;

        } else if(a[i] > pivot) {

            swap(a[i], a[gt]);

            moves++;

            gt--;

        } else {

            i++;

        }

    }

    quick\_sort\_3way(a, low, lt - 1, comparisons, moves);

    quick\_sort\_3way(a, gt + 1, high, comparisons, moves);

}

中位数快排实现如下：

// 首尾中数三数取中法找到 pivot

int medianOfThree(int \*a, int low, int high, long long &comparisons) {

    int mid = low + (high - low) / 2;

    comparisons += 3;

    if (a[high] < a[low]) swap(a[low], a[high]);

    if (a[mid] < a[low]) swap(a[low], a[mid]);

    if (a[high] < a[mid]) swap(a[mid], a[high]);

    swap(a[mid], a[high]);

    return a[high]; // 返回枢轴

}

// 分区函数

int partition\_of\_three(int \*a, int low, int high, long long &comparisons, long long &moves) {

    int pivot = a[high];

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        comparisons++;

        if (a[j] < pivot) {

            i++;

            swap(a[i], a[j]);

            moves++;

        }

    }

    swap(a[i + 1], a[high]);

    moves++;

    return i + 1;

}

// 三数取中法的快速排序

void quick\_sort\_medians(int \*a, int low, int high, long long &comparisons, long long &moves) {

    while (low < high) {

        comparisons++;

        int pivot = medianOfThree(a, low, high, comparisons);

        int pi = partition\_of\_three(a, low, high, comparisons, moves);

        // 选择较小部分递归，优化尾递归

        if (pi - low < high - pi) {

            quick\_sort\_medians(a, low, pi - 1, comparisons, moves);

            low = pi + 1;

        } else {

            quick\_sort\_medians(a, pi + 1, high, comparisons, moves);

            high = pi - 1;

        }

    }

}

#### 数据生成程序

为了全面评估算法，在验证算法正确性时，数据集应设计成不同规模，能够保证算法程序各个分支都被测试到，因此数据集规模可以通过sizes进行调整，目前程序中的数据规模设置为100，5000以及20000，每个生成的数据集包括：

·随机数据

·顺序数据

·逆序数据

·几乎有序的数据

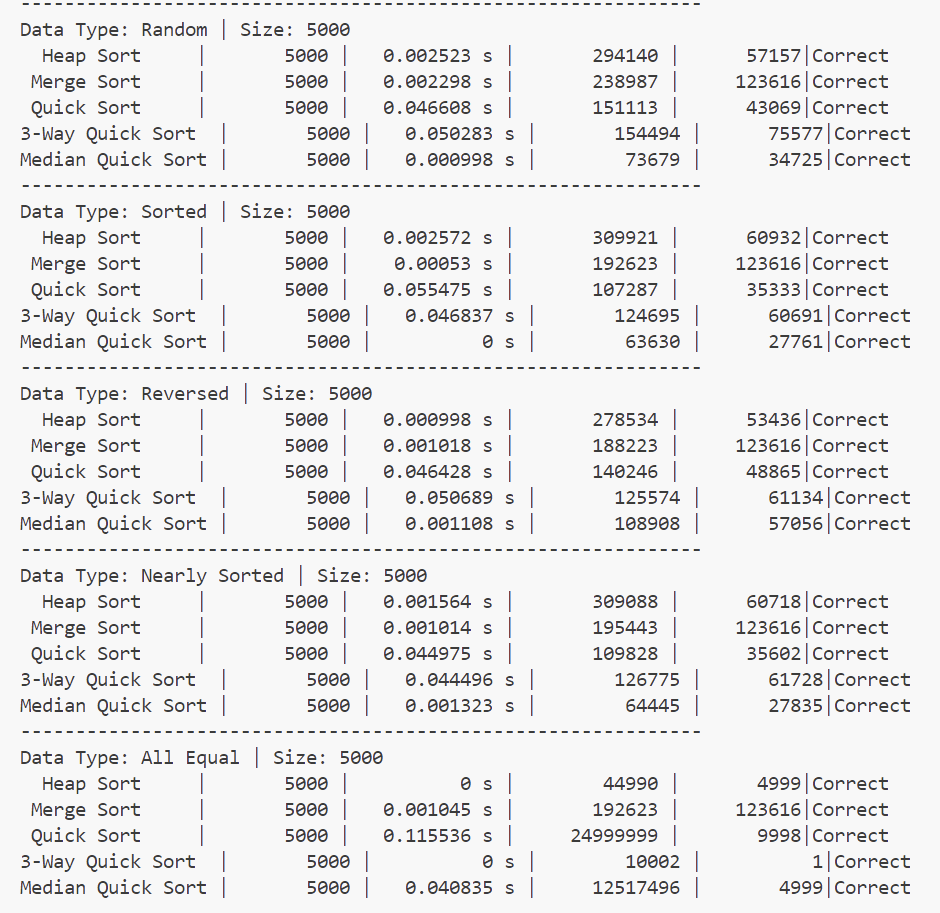
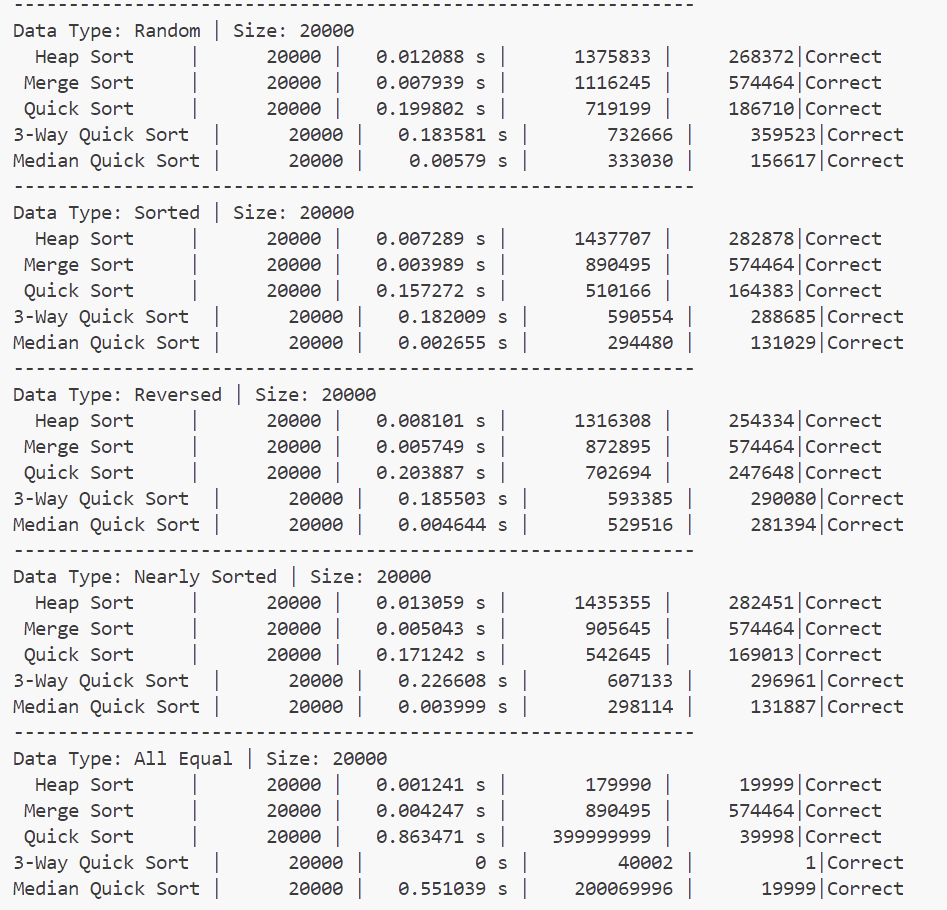
·全部相等的数据

这些数据集覆盖了不同的复杂性条件，以测试和验证算法在各种情况下的表现，具体代码可见程序的SortData.h头文件，在此不做展示。

#### 算法测试程序

分别调用不同的算法模块，以及相关的数据集输入，得到不同的测试结果，然后给出评估，通过调用C++库中的is\_sorted函数来对排序过后的数组进行检测，来验证算法的正确性，然后通过各排序算法模块被调用执行的结束时间减去调用时间得到程序的运行时间，并输出对不同规模不同种类的测试数据集程序的比较次数，数据的移动次数以及程序运行的时间，具体实现可看test.cpp文件

## 运行结果及分析



#### 正确性

可以看到在不同数据集（随机、顺序、逆序、近乎有序、全相等）下，三种排序算法均能得到正确的排序结果。

#### 算法复杂度

通过对不同规模数据集进行测试，记录比较次数和移动次数来评估三种算法的表现。结果表明：

·堆排序在不同数据集下表现稳定。

·归并排序的性能在数据规模较大时表现较好。

·快速排序在最坏情况下的复杂性较高，但三数取中法和三路快排优化了最坏情况，通过三路快排法实现了稳定的快排算法，同时三数取中尽量避免了程序效率的退化。

在非极端情况下，可以看到快速排序算法的比较次数和数据迁移次数最少，然后是归并排序的比较次数更少，然后是堆排序，因此理论上快排的速度最快，但是可能由于快排调用栈资源较多及基准点的选取并不合适，在输出结果中一般是三数取中快排最快，其次是归并排序，然后是堆排序，再然后才是快排和三路快排，而在极端情况（全部相等）下，可以看到快排和三数取中快排的比较次数会变得非常大，导致算法退化，速度减慢。

综上，堆排序在处理有序和无序数据时表现相对稳定，但比较次数较多；归并排序适用于大规模数据，具有良好的稳定性，但相对堆排序，移动次数略高。快速排序理论上在平均情况下效率最高，经过三数取中和三路分区优化后，能有效减少最坏情况的影响，三数取中

算法从输出结果上看比较稳定且效率最高。

## 实验总结

本次实验我实现了对堆排序，归并排序，快速排序及其优化：三路快排，三数取中快排的实现，在实验中我也遇到了不少问题，对三数取中快排，原本我的想法是实现课程PPT中的选取中位数的中位数作为基准点的方法，但是本人能力有限，实现的该算法在数据量稍微大一点后就会发生栈溢出，判断可能是集合划分的太小了，后续进行多次修改也没有修复，于是我便把该方法简化成了挑选数组首元素，中间元素以及末尾元素的中位数作为基准点的三数取中版；在对后续程序的测试中，我发现对完全相同的数组这种极端情况，如果数组内元素个数在五万及以上，快排的深层递归会递归太多层导致栈溢出，于是我便把数据集的规模设置在了两万以下，若需更大则需要把完全相等的数据集种类去除。

本次实验让我对这三种排序算法，尤其是快速排序有了更深刻的理解，并学会了如何去编写数据生成程序以及测试程序，这对之前只会手搓输入输出的我来说无疑是一个巨大的飞跃，与此次实验中我收获颇丰。