数据结构课程设计

项目说明文档

十一种排序算法的比较案例

作者姓名: 汪明杰

学　　号: 1851055

指导教师: 张　颖

学院专业: 软件学院 软件工程

同济大学

Tongji University

目录

1 项目分析

1.1 项目背景

所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。排序算法，就是如何使得记录按照要求排列的方法。排序算法在很多领域得到相当地重视，尤其是在大量数据的处理方面。一个优秀的算法可以节省大量的资源。在各个领域中考虑到数据的各种限制和规范，要得到一个符合实际的优秀算法，得经过大量的推理和分析。

本项目通过对 11 种排序算法性能的探究，能够为未来各个算法的使用途径提供一些帮助。

1.2 项目需求分析

针对于 11 种排序算法的比较这一系统，本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ **代码可读性强**

  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

- ✓ **健壮性**

  当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

- ✓ **可视化**

  该系统通过输出当前正在执行的操作内容，使得用户可以直观的了解到当前操作内容。

- ✓ **对比性**

  为了让数据之间更加具有对比性，本项目考虑了 `10-1000000` 个随机数的情况，同时分为了**随机序列**、**升序序列**和**降序序列**，还和**标准 stl 库**中的排序算法做了对比。

1.3 项目要求

1.3.1 功能要求

随机函数产生一百，一千，一万和十万个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的比较次数。

请在文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

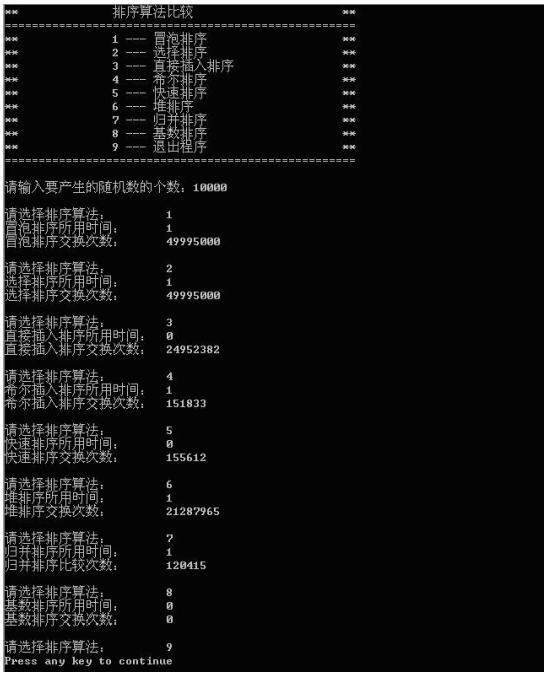
1.3.2 输入格式

生成随机数的大小。

1.3.3 输出格式

打印随机序列、升序序列、降序序列下各排序算法的排序时间和交换次数等。

1.3.4 项目示例



2 项目实现

本项目共实现了以下 11 种排序算法：

- ✓ 冒泡排序
- ✓ 选择排序
- ✓ 直接插入排序
- ✓ 折半插入排序
- ✓ 希尔排序
- ✓ 堆排序
- ✓ 快速排序
- ✓ 归并排序
- ✓ 桶排序
- ✓ LSD 基数排序
- ✓ MSD 基数排序

下面将依次对本项目实现的这 11 种排序算法进行介绍。

2.1 冒泡排序

2.1.1 算法逻辑

冒泡排序算法是把较小的元素往前调或者把较大的元素往后调。这种方法主要是通过对相邻两个元素进行大小的比较，根据比较结果和算法规则对该二元素的位置进行交换，这样逐个依次进行比较和交换，就能达到排序目的。冒泡排序的基本思想是，首先将第 1 个和第 2 个记录的关键字比较大小，如果是逆序的，就将这两个记录进行交换，再对第 2 个和第 3 个记录的关键字进行比较，依次类推，重复进行上述计算，直至完成第(n-1)个和第 n 个记录的关键字之间的比较，此后，再按照上述过程进行第 2 次、第 3 次排序，直至整个序列有序为止。排序过程中要特别注意的是，当相邻两个元素大小一致时，这一步操作就不需要交换位置，因此也说明冒泡排序是一种严格的稳定排序算法，它不改变序列中相同元素之间的相对位置关系。

2.1.2 算法代码

```
void bubbleSort(int *p, int n, sortData& sort)
{
    bool disordered = true;
    for (int i = n - 1; i >= 0 && disordered; --i)
    {
        disordered = false;
        for (int j = 0; j < i; ++j)
        {
            if (p[j] > p[j + 1])
            {
                mySTL::swap(p[j], p[j + 1]);
                sort.move += 3;
                disordered = true;
            }
            ++sort.compare;
        }
    }
}
```

2.1.3 算法分析

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

2.2 选择排序

2.2.1 算法逻辑

选择排序算法的基本思路是为每一个位置选择当前最小的元素。选择排序的基本思想是，基于直接选择排序和堆排序这两种基本的简单排序方法。首先从第 1 个位置开始对全部元素进行选择，选出全部元素中最小的给该位置，再对第 2 个位置进行选择，在剩余元素中选择最小的给该位置即可；以此类推，重复进行

“最小元素”的选择，直至完成第(n-1)个位置的元素选择，则第 n 个位置就只剩唯一的最大元素，此时不需再进行选择。

2.2.2 算法代码

```
void selectSort(int *p, int n, sortData& sort)
{
    for (int i = 0; i < n; ++i)
    {
        int min = i;
        for (int j = i + 1; j < n; ++j)
        {
            if (p[j] < p[min])
                min = j;
            ++sort.compare;
        }
        if (min != i)
        {
            mySTL::swap(p[i], p[min]);sort.move += 3;
        }
    }
}
```

2.2.3 算法分析

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第 n-1 个元素，第 n 个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果一个元素比当前元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那么交换后稳定性就被破坏了。因此，选择排序是一个不稳定的排序算法。

2.3 直接插入排序

2.3.1 算法逻辑

直接插入排序通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入，如此重复，直至完成序列排序。

2.3.2 算法代码

```
void straightInsertSort(int *p, int n, sortData& sort)
{
    for (int i = 1; i < n; i++)
    {
        int temp = p[i];
        ++sort.move;
        int j = i;
        while (j > 0 && (++sort.compare, temp < p[j - 1])) {  //
            p[j] = p[j - 1];
            ++sort.move;
            j--;
        }
        p[j] = temp;
        ++sort.move;
    }
}
```

2.3.3 算法分析

直接插入排序是稳定的算法，它满足稳定算法的定义，即：假设在数列中存在 a[i]=a[j]，若在排序之前，a[i]在 a[j]前面；并且排序之后，a[i]仍然在 a[j]前面

2.4 折半插入排序

2.4.1 算法逻辑

二分查找插入排序是直接插入排序的一个变种，区别是在有序区中查找新元素插入位置时，为了减少元素比较次数提高效率，采用二分查找算法进行插入位置的确定。

2.4.2 算法代码

```
void binaryInsertSort(int *p, int n, sortData &sort)
{
    for (int i = 1; i < n; ++i)
    {
```

```
        int begin = 0, end = i;
        int temp = p[i];
        ++sort.move;
        while (begin < end)
        {
            int mid = (begin + end - 1) / 2;
            if (temp < p[mid])  // a[mid]较大，在左侧找
                end = mid;
            else begin = mid + 1;
            ++sort.compare;
        }
        for (int k = i - 1; k >= begin; k--)
        {
            p[k + 1] = p[k];
            ++sort.move;
        }
        p[begin] = temp;
        ++sort.move;
    }
}
```

2.4.3 算法分析

折半插入排序算法是一种稳定的排序算法，比直接插入算法明显减少了关键字之间比较的次数，因此速度比直接插入排序算法快，但记录移动的次数没有变，所以折半插入排序算法的时间复杂度仍然为 $O(n^2)$，与直接插入排序算法相同。附加空间 O(1)。

2.5 希尔排序

2.5.1 算法逻辑

希尔排序实质上是一种分组插入方法。它的基本思想是：对于 n 个待排序的数列，取一个小于 n 的整数 gap 将待排序元素分成若干个组子序列，所有距离为 gap 的倍数的记录放在同一个组中；然后，对各组内的元素进行直接插入排序。这一趟排序完成之后，每一个组的元素都是有序的。然后减小 gap 的值，并重复执行上述的分组和排序。重复这样的操作，当 gap=1 时，整个数列就是有序的。

2.5.2 算法代码

```
void ShellSort(int *p, int n, sortData &sort)
{
    int gap = 1;
    while (gap < n / 3)
        gap = 3 * gap + 1;  // h 依次取1,4,13,40,121
    while (gap)
    {
        for (int i = gap; i < n; i++)
        {
            int temp = p[i];
            ++sort.move;
            int j = i;
            while (j >= gap && (++sort.compare, temp < p[j - gap]))
            {
                p[j] = p[j - gap];
                ++sort.move;
                j -= gap;
            }
            p[j] = temp;
            ++sort.move;
        }
        gap /= 3;
    }
}
```

2.5.3 算法分析

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。希尔排序的时间复杂度会比 $O(n^2)$好一些。同时，希尔排序是不稳定的排序算法。

2.6 堆排序

2.6.1 算法逻辑

堆排序是指利用堆（Heap）这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

堆是一类特殊数据结构的简称，通常是一个可以被看做一棵树的的数组对象。堆满足下述两个性质：

（1） 堆中某个节点的值总是不大于或者不小于其父节点的值；

（2） 堆总是一棵完全二叉树。

将根节点最大的堆叫做最大堆或者大根堆，根节点最小的堆叫做最小堆小根堆。常见的堆有二叉堆、斐波拉契堆等。主要函数如下所示：

◆ 建堆函数

将[start,end)内的元素调整为堆

```
template <class Iter>
void makeHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    ptrdiff_t len = end - start;
    if (len < 2) return;
    ptrdiff_t hole = (len - 2) / 2;
    while (hole >= 0) {
        //调整 hole 节点
        sinkFix(start, len, hole, *(start + hole), Less<value_type>());
        --hole;
    }
}
```

◆ 向上调整

向上调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void siftFix(Iterator first,
    ptrdiff_t top,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
```

```
    ptrdiff_t parent = (hole - 1) / 2;
    while (parent >= top && comp(*(first + parent), value)) {
        *(first + hole) = *(first + parent);
        hole = parent;
        parent = (hole + 1) / 2 - 1;
    }
    *(first + hole) = value;
}
```

◆ 向下调整

向下调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void sinkFix(Iterator first,
    ptrdiff_t len,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
    /* 1.将hole 下移到叶子处 */
    ptrdiff_t bigChild = 2 * hole + 2;  //从右孩子开始考虑
    while (bigChild < len) {
        if (comp(*(first + bigChild), *(first + bigChild - 1)))
            bigChild--;  //左孩子大于右孩子
        *(first + hole) = *(first + bigChild);
        hole = bigChild;
        bigChild = 2 * hole + 2;
    }
    if (len % 2 == 0 &&
        bigChild == len) {  //特殊情况：最后一个节点为左节点，没有兄弟节点
        *(first + hole) = *(first + bigChild - 1);
        hole = bigChild - 1;
    }
    /* 2.转化为向上调整问题 */
    siftFix(first, 0, hole, value, comp);
}
```

◆ 入堆

将*(end-1)调整到合适的位置，使得[start,end)变成堆

```
template <class Iter>
void pushHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    siftFix(start, 0, end - start - 1, *(end - 1), Less<value_type>());
```

```
}
```

◆ **出堆**

将*(start)交换到*(end-1)处(以便删去), [start,end-1)变成堆

```
template <class Iter>
void popHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    value_type value = *(end - 1);  //将原来的末尾值保存起来
    *(end - 1) = *(start);  //将*(start)移动到*(end-1),*(start)处出现
hole
    sinkFix(start, end - start - 1, 0, value, Less<value_type>());
}
```

2.6.2 算法代码

```
void HeapSort(int *p, int n, sortData &sort)
{
    for (int i = (n - 2) / 2; i >= 0; i--)  //堆化,从倒数第二层(向上)开始
修复
        fixHeap(p, n, i, sort);

    for (int i = n - 1; i >= 1; i--)
    {
        mySTL::swap(p[i], p[0]);        //将最大的元素放在最后
        sort.move += 3;          //
        fixHeap(p, i, 0, sort);  //调整剩下的堆
    }
}
```

2.6.3 算法逻辑

堆排序是一种选择排序，整体主要由构建初始堆+交换堆顶元素和末尾元素并重建堆两部分组成。其中构建初始堆经推导复杂度为 O(n)，在交换并重建堆的过程中，需交换 n-1 次，而重建堆的过程中，根据完全二叉树的性质，[log2(n-1),log2(n-2)...1]逐步递减，近似为 nlogn。所以堆排序时间复杂度一般认为就是 O(nlogn)级。堆排序是一种不稳定的排序算法。

2.7 快速排序

2.7.1 算法逻辑

快速排序的基本思想是:通过一趟排序算法把所需要排序的序列的元素分割成两大块，其中，一部分的元素都要小于或等于另外一部分的序列元素，然后仍根据该种方法对划分后的这两块序列的元素分别再次实行快速排序算法，排序实现的整个过程可以是递归的来进行调用，最终能够实现将所需排序的无序序列元素变为一个有序的序列。

2.7.2 算法代码

```
void _quickSort(int *a, int begin, int end, sortData &data)
{
    if (end - begin < 2)
        return;
    int i = begin, j = end - 1;
    //三路取中优化
    int mid = (i + j) / 2, k = i;
    if (a[mid] < a[k])
        k = mid;
    if (a[j] < a[k])
        k = j;
    if (k != j)
        mySTL::swap(a[k], a[j]);//三者最小值放到最右端
    if (a[mid] < a[i])
        mySTL::swap(a[mid], a[i]);//次小值放最左端

    //哨兵
    int pivot = a[i];
    while (i != j)
    {
        while (i < j && (++data.compare, a[j] >= pivot)) j--;
        a[i] = a[j];
        ++data.move;
        while (i < j && (++data.compare, a[i] <= pivot)) i++;
        a[j] = a[i];
        ++data.move;
    }
    a[i] = pivot;
```

```
    ++data.move;
    _quickSort(a, begin, i, data);
    _quickSort(a, i + 1, end, data);
}
```

2.7.3 算法分析

快速排序的平均时间复杂度也是 O(nlog2n)。因此，该排序方法被认为是目前最好的一种内部排序方法。

从空间性能上看，尽管快速排序只需要一个元素的辅助空间，但快速排序需要一个栈空间来实现递归。最好的情况下，即快速排序的每一趟排序都将元素序列均匀地分割成长度相近的两个子表，所需栈的最大深度为 log2(n+1)；但最坏的情况下，栈的最大深度为 n。这样，快速排序的空间复杂度为 O(log2n)。

同时，快速排序是一个不稳定的排序算法。

2.8 归并排序

2.8.1 算法逻辑

归并排序算法就是把序列递归划分成为一个个短序列，以其中只有 1 个元素的直接序列或者只有 2 个元素的序列作为短序列的递归出口，再将全部有序的短序列按照一定的规则进行排序为长序列。归并排序融合了分治策略，即将含有 n 个记录的初始序列中的每个记录均视为长度为 1 的子序列，再将这 n 个子序列两两合并得到 n/2 个长度为 2(当凡为奇数时会出现长度为 1 的情况)的有序子序列；将上述步骤重复操作，直至得到 1 个长度为 n 的有序长序列。需要注意的是，在进行元素比较和交换时，若两个元素大小相等则不必刻意交换位置，因此该算法不会破坏序列的稳定性，即归并排序也是稳定的排序算法。

2.8.2 算法代码

```
void _merge(int *a, int begin, int mid, int end, int *arr, sortData &da
ta)
{
    for (int k = begin; k < end; k++)
    {
        arr[k] = a[k];
        ++data.move;
```

```
    }
    int i = begin, j = mid;
    for (int k = begin; k < end; k++)
    {
        if (i >= mid)
            a[k] = arr[j++];
        else if (j >= end)
            a[k] = arr[i++];
        else
        {
            ++data.compare;
            if (arr[j] < arr[i])
                a[k] = arr[j++];
            else
                a[k] = arr[i++];
        }
        ++data.move;
    }
}

void _mergeSort(int *a, int begin, int end, int *arr, sortData &data)
{
    if (end - begin <= 1)
        return;
    int mid = (begin + end) / 2;
    _mergeSort(a, begin, mid, arr, data);
    _mergeSort(a, mid, end, arr, data);
    _merge(a, begin, mid, end, arr, data);
}
```

2.8.3 算法分析

不管元素在什么情况下都要做这些步骤，所以花销的时间是不变的，所以该算法的最优时间复杂度和最差时间复杂度及平均时间复杂度都是一样的为：O(nlogn)

归并的空间复杂度就是那个临时的数组和递归时压入栈的数据占用的空间：n + logn；所以空间复杂度为: O(n)。

归并排序算法中，归并最后到底都是相邻元素之间的比较交换，并不会发生相同元素的相对位置发生变化，故是稳定性算法。

2.9 桶排序

2.9.1 算法逻辑

将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）。桶排序是鸽巢排序的一种归纳结果。

2.9.2 算法代码

```
void bucketSort(int arr[], int n, sortData& data)
{
    int i, j;
    int *buckets = new int[n + 1];

    for (i = 0; i < n + 1; i++) // 清零
        buckets[i] = 0;
    // 1.计数,将数组 arr 中的元素放到桶中
    for (i = 0; i < n; i++)
        buckets[arr[i]]++; // 将arr[i]的值对应buckets 数组的下标，每有一个
就加1
    // 2.排序
    for (i = 0, j = 0; i < n + 1; i++) {
        while (buckets[i] > 0) { // 说明存有元素,相同的整数，要重复输出
            arr[j] = i;
            buckets[i]--;
            j++;
        }
    }
    delete[] buckets;
}
```

2.9.3 算法分析

桶排序的平均时间复杂度为线性的 O(N+C)，其中 C=N*(logN-logM)。如果相对于同样的 N，桶数量 M 越大，其效率越高，最好的时间复杂度达到 O(N)。当然桶排序的空间复杂度为 O(N+M)，如果输入数据非常庞大，而桶的数量也非常多，则空间代价无疑是昂贵的。此外，桶排序是稳定的。

2.10 LSD 基数排序

2.10.1 算法逻辑

将所有待比较数值（这里以正整数为例）统一为同样的数位长度，数位较短的数前面补零。设一整数的十进制位数表示为 k1,k2...kd，其中 k1 为高位，kd 为低位。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。即先从 kd 开始排序，再对 kd-1 进行排序，依次重复，直到对 k1 排序后便得到一个有序序列。

2.10.2 算法代码

```
void LSDRadixSort(int *a, int n, sortData &time)
{
    int max = -1000000000, maxd = -1;
    for (int i = 0; i < n; i++)
        if (a[i] > max) max = a[i];
    while (max)
    {
        max >>= 4;
        maxd++;
    }
    int *aux = new int[n];
    int *count = new int[radix];
    for (int d = 0; d <= maxd; d++)
    {
        memset(count, 0, radix * sizeof(int));
        for (int i = 0; i < n; i++) count[getDigitBase16(a[i], d)]++;
        for (int i = 1; i < radix; i++) count[i] += count[i - 1];  //前
缀和
        for (int i = n - 1; i >= 0; i--)
            aux[--count[getDigitBase16(a[i], d)]] = a[i];
        memcpy(a, aux, n * sizeof(int));
        time.move += 2 * n;  //
    }
    delete[] count;
    delete[] aux;
}
```

2.11 MSD 基数排序

2.11.1 算法逻辑

MSD 的方式与 LSD 相反，是由高位数为基底开始进行分配，但在分配之后并不马上合并回一个数组中，而是在每个组中建立“子组”，将每个桶子中的数值按照下一数位的值分配到“子组”中。在进行完最低位数的分配后再合并回单一的数组中。即先按 k1 排序分组，同一组中记录，关键码 k1 相等，再对各组按 k2 排序分成子组，之后，对后面的关键码继续这样的排序分组，直到按最次位关键码 kd 对各子组排序后。再将各组连接起来，便得到一个有序序列。所以，MSD 方法用递归的思想实现最为直接。

2.11.2 算法代码

```
void MSDRadixSort(int *a, int n, sortData &time)
{
    int max = -1000000000, maxd = -1;
    for (int i = 0; i < n; i++)
        if (a[i] > max) max = a[i];
    while (max)
    {
        max >>= 4;
        maxd++;
    }
    _MSDRadixSort(a, 0, n, maxd, time);
}

static void _MSDRadixSort(int *a, int begin, int end, int d, sortData &
time)
{
    const int n = end - begin;
    if (n <= 1 || d < 0) return;
    int *aux = new int[n];
    int *count = new int[radix];
    memset(count, 0, radix * sizeof(int));
    for (int i = begin; i < end; i++)
        count[getDigitBase16(a[i], d)]++;
    for (int i = 1; i < radix; i++)
        count[i] += count[i - 1];  //前缀和
```

```
    for (int i = begin; i < end; i++)
        aux[--count[getDigitBase16(a[i], d)]] = a[i];
    memcpy(a + begin, aux, n * sizeof(int));
    time.move += 2 * n;  //
    for (int i = 0; i < radix; i++)
    {
        _MSDRadixSort(a, begin + count[i],
            begin + (i == radix - 1 ? n : count[i + 1]), d - 1,
            time);  //递归
    }
    delete[] count;
    delete[] aux;
}
```

3 各排序算法对比

通过对各种排序算法实现方法的过程分析，可以得出它们的时间复杂度、空间复杂度和稳定性，如下所示：

算法	时间复杂度	空间复杂度	稳定性
冒泡排序	O(N²)	O(1)	是
选择排序	O(N²)	O(1)	否
直接插入排序	O(N²)	O(1)	是
折半插入排序	O(N²)	O(1)	是
希尔排序	$O(N^{1.3})$	O(1)	否
堆排序	O(NlogN)	O(1)	否
快速排序	O(NlogN)	O(logN)	否
归并排序	O(NlogN)	O(N)	是
桶排序	O(N)	O(N+M)	是
LSD 基数排序	O(NlogM)	O(N)	是
MSD 基数排序	O(NlogM)	O(N)	是

4 项目测试

4.1 10 个随机数测试

4.1.1 随机序列

**	随机序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	45	111
选择排序	0ms	45	15
直接插入排序	0ms	43	55
折半插入排序	0ms	22	55
希尔排序	0ms	18	39
堆排序	0ms	38	72
快速排序	0ms	33	29
归并排序	0ms	23	68
桶排序	0ms	0	0
MSD基数排序	0ms	0	20
LSD基数排序	0ms	0	20
stl库排序	0ms		

4.1.2 升序序列

**	升序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	9	0
选择排序	0ms	45	0
直接插入排序	0ms	9	18
折半插入排序	0ms	25	18
希尔排序	0ms	15	30
堆排序	0ms	41	90
快速排序	0ms	28	24
归并排序	0ms	15	68
桶排序	0ms	0	0
MSD基数排序	0ms	0	20
LSD基数排序	0ms	0	20
stl库排序	0ms		

4.1.3 降序序列

**	降序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	45	135
选择排序	0ms	45	15
直接插入排序	0ms	45	63
折半插入排序	0ms	19	63
希尔排序	0ms	21	43
堆排序	0ms	35	63
快速排序	0ms	36	30
归并排序	0ms	19	68
桶排序	0ms	0	0
MSD基数排序	0ms	0	20
LSD基数排序	0ms	0	20
stl库排序	0ms		

4.2 100 个随机数测试

4.2.1 随机序列

**	随机序列		**
算法	时间	比较次数	移动次数
冒泡排序	1ms	4760	7737
选择排序	0ms	4950	279
直接插入排序	0ms	2675	2777
折半插入排序	0ms	524	2777
希尔排序	0ms	720	1113
堆排序	0ms	1024	1749
快速排序	0ms	861	417
归并排序	0ms	541	1344
桶排序	0ms	0	0
MSD基数排序	0ms	0	400
LSD基数排序	0ms	0	400
stl库排序	0ms		

4.2.2 升序序列

**	升序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	99	0
选择排序	0ms	4950	0
直接插入排序	0ms	99	198
折半插入排序	0ms	573	198
希尔排序	0ms	342	684
堆排序	0ms	1081	1920
快速排序	0ms	579	261
归并排序	0ms	316	1344
桶排序	0ms	0	0
MSD基数排序	0ms	0	400
LSD基数排序	0ms	0	400
stl库排序	0ms		

4.2.3 降序序列

**	降序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	4950	14850
选择排序	0ms	4950	150
直接插入排序	0ms	4950	5148
折半插入排序	0ms	480	5148
希尔排序	0ms	500	914
堆排序	0ms	944	1548
快速排序	0ms	765	352
归并排序	0ms	356	1344
桶排序	0ms	0	0
MSD基数排序	0ms	0	400
LSD基数排序	0ms	0	400
stl库排序	0ms		

4.3 1000 个随机数测试

4.3.1 随机序列

**	随机序列		**
算法	时间	比较次数	移动次数
冒泡排序	20ms	496872	766149
选择排序	3ms	499500	2985
直接插入排序	2ms	256380	257381
折半插入排序	2ms	8593	257381
希尔排序	0ms	13817	19720
堆排序	0ms	16859	27270
快速排序	1ms	13917	5605
归并排序	1ms	8711	19952
桶排序	0ms	0	0
MSD基数排序	1ms	0	6000
LSD基数排序	1ms	0	6000
stl库排序	2ms		

4.3.2 升序序列

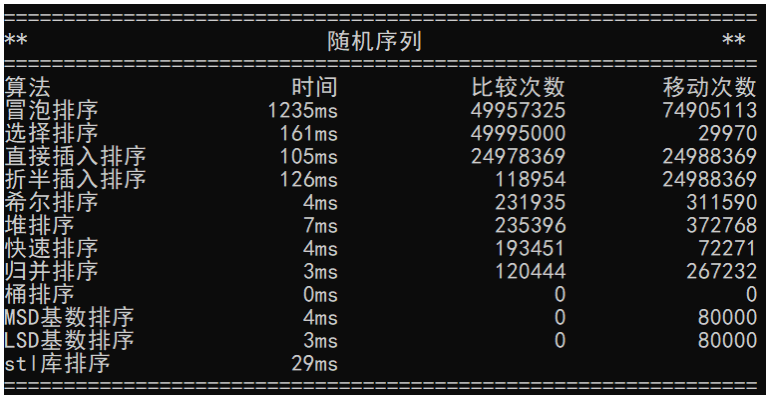
**	升序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	999	0
选择排序	2ms	499500	0
直接插入排序	0ms	999	1998
折半插入排序	0ms	8977	1998
希尔排序	0ms	5457	10914
堆排序	1ms	17583	29124
快速排序	0ms	8986	2509
归并排序	0ms	4932	19952
桶排序	0ms	0	0
MSD基数排序	1ms	0	6000
LSD基数排序	0ms	0	6000
stl库排序	1ms		

4.3.3 降序序列

**	降序序列		**
算法	时间	比较次数	移动次数
冒泡排序	24ms	499500	1498500
选择排序	3ms	499500	1500
直接插入排序	3ms	499500	501498
折半插入排序	2ms	7987	501498
希尔排序	1ms	8550	14834
堆排序	0ms	15965	24948
快速排序	1ms	12378	3504
归并排序	1ms	5044	19952
桶排序	0ms	0	0
MSD基数排序	1ms	0	6000
LSD基数排序	1ms	0	6000
stl库排序	1ms		

4.4 10000 个随机数测试

4.4.1 随机序列



**	随机序列		**
算法	时间	比较次数	移动次数
冒泡排序	1235ms	49957325	74905113
选择排序	161ms	49995000	29970
直接插入排序	105ms	24978369	24988369
折半插入排序	126ms	118954	24988369
希尔排序	4ms	231935	311590
堆排序	7ms	235396	372768
快速排序	4ms	193451	72271
归并排序	3ms	120444	267232
桶排序	0ms	0	0
MSD基数排序	4ms	0	80000
LSD基数排序	3ms	0	80000
stl库排序	29ms		

4.4.2 升序序列

**	升序序列		**
算法	时间	比较次数	移动次数
冒泡排序	0ms	9999	0
选择排序	171ms	49995000	0
直接插入排序	0ms	9999	19998
折半插入排序	2ms	123617	19998
希尔排序	1ms	75243	150486
堆排序	7ms	244460	395868
快速排序	1ms	123630	25902
归并排序	4ms	64608	267232
桶排序	1ms	0	0
MSD基数排序	6ms	0	80000
LSD基数排序	3ms	0	80000
stl库排序	10ms		

4.4.3 降序序列

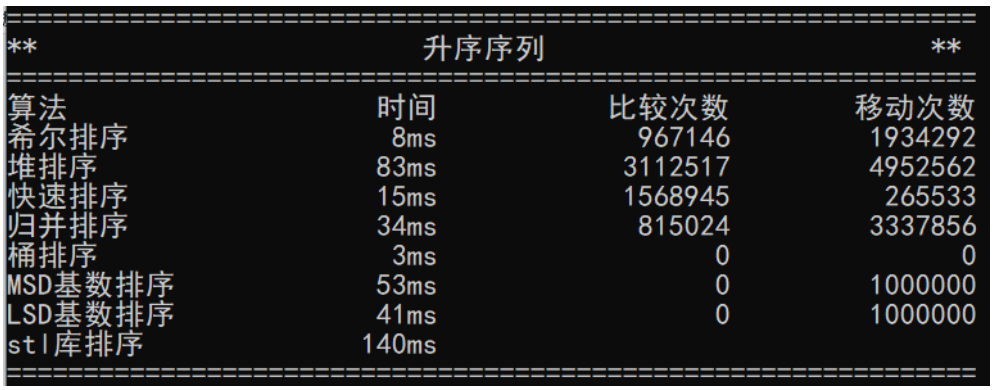
**	降序序列		**
算法	时间	比较次数	移动次数
冒泡排序	1847ms	49995000	149985000
选择排序	192ms	49995000	15000
直接插入排序	213ms	49995000	50014998
折半插入排序	168ms	113631	50014998
希尔排序	1ms	120190	204190
堆排序	5ms	226682	350088
快速排序	2ms	173556	35899
归并排序	2ms	69008	267232
桶排序	1ms	0	0
MSD基数排序	5ms	0	79998
LSD基数排序	2ms	0	80000
stl库排序	12ms		

4.5 100000 个随机数测试

4.5.1 随机序列

**	随机序列		**
算法	时间	比较次数	移动次数
希尔排序	51ms	3979361	4987896
堆排序	108ms	3019596	4724298
快速排序	39ms	2504746	872486
归并排序	43ms	1536178	3337856
桶排序	2ms	0	0
MSD基数排序	41ms	0	800000
LSD基数排序	35ms	0	800000
stl库排序	315ms		

4.5.2 升序序列



**	升序序列		**
算法	时间	比较次数	移动次数
希尔排序	8ms	967146	1934292
堆排序	83ms	3112517	4952562
快速排序	15ms	1568945	265533
归并排序	34ms	815024	3337856
桶排序	3ms	0	0
MSD基数排序	53ms	0	1000000
LSD基数排序	41ms	0	1000000
stl库排序	140ms		

4.5.3 降序序列

**	降序序列		**
算法	时间	比较次数	移动次数
希尔排序	12ms	1533494	2553946
堆排序	84ms	2926640	4492302
快速排序	24ms	2229743	358186
归并排序	36ms	853904	3337856
桶排序	1ms	0	0
MSD基数排序	59ms	0	999998
LSD基数排序	43ms	0	1000000
stl库排序	161ms		

4.6 1000000 个随机数测试

4.6.1 随机序列

**	随机序列		**
算法	时间	比较次数	移动次数
希尔排序	534ms	61279450	73492591
堆排序	1090ms	36794165	57145488
快速排序	346ms	41229765	9526837
归并排序	426ms	18673390	39902848
桶排序	15ms	0	0
MSD基数排序	265ms	0	8000000
LSD基数排序	338ms	0	8000000
stl库排序	2607ms		

4.6.2 升序序列

**	升序序列		**
算法	时间	比较次数	移动次数
希尔排序	84ms	11804265	23608530
堆排序	941ms	37692069	59363376
快速排序	141ms	18951444	2524285
归并排序	316ms	9884992	39902848
桶排序	27ms	0	0
MSD基数排序	626ms	0	10000000
LSD基数排序	337ms	0	10000000
stl库排序	1551ms		

4.6.3 降序序列

**	降序序列		**
算法	时间	比较次数	移动次数
希尔排序	154ms	17578626	29853914
堆排序	1100ms	36001436	55000224
快速排序	205ms	27296156	3524280
归并排序	403ms	10066432	39902848
桶排序	22ms	0	0
MSD基数排序	604ms	0	9999998
LSD基数排序	467ms	0	10000000
stl库排序	2081ms		