# 8种排序算法的比较案例

## 问题描述

随机函数产生10000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，系统产生随机书。并且显示他们的比较次数。

## 算法介绍

**1. 冒泡排序**

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，我想你是不会再无聊地把他们俩交换一下的；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. 选择排序**

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，如果当前元素比一个元素小，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。比较拗口，举个例子，序列5 8 5 2 9， 我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

**3. 直接插入排序**

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。当然，刚开始这个有序的小序列只有1个元素，就是第一个元素。比较是从有序序列的末尾开始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

**4. 希尔排序**

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。所以，希尔排序的时间复杂度会比o(n^2)好一些。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的。

**5. 快速排序**

快速排序有两个方向，左边的i下标一直往右走，当a[i] <= a[center_index]，其中center_index是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j] > a[center_index]。如果i和j都走不动了，i <= j, 交换a[i]和a[j],重复上面的过程，直到i>j。 交换a[j]和a[center_index]，完成一趟快速排序。在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11， 现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和a[j]交换的时刻。

**6. 堆排序**

我们知道堆的结构是节点i的孩子为2*i*和2i+1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n的序列，堆排序的过程是从第n/2开始和其子节点共3个值选择最大(大顶堆)或者最小(小顶堆),这3个元素之间的选择当然不会破坏稳定性。但当为n/2-1, n/2-2, ...1这些个父节点选择元素时，就会破坏稳定性。有可能第n/2个父节点交换把后面一个元素交换过去了，而第n/2-1个父节点把后面一个相同的元素没有交换，那么这2个相同的元素之间的稳定性就被破坏了。所以，堆排序不是稳定的排序算法。

**7. 归并排序**

归并排序是把序列递归地分成短序列，递归出口是短序列只有1个元素(认为直接有序)或者2个序列(1次比较和交换),然后把各个有序的段序列合并成一个有序的长序列，不断合并直到原序列全部排好序。可以发现，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也没有人故意交换，这不会破坏稳定性。那么，在短的有序序列合并的过程中，稳定没有受到破坏，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序也是稳定的排序算法。

**8. 基数排序**

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序，最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以其是稳定的排序算法。

## 功能实现及代码分析

### 1、进入项目

进入项目，生成随机数，选择排序方式。

```
 int iCount = 0;
time_t before, after;
int size = 0;

int main() {
    int array[MAX];
    cout << "**                  排序算法比较            **\n";
    cout << "============================================\n";
    cout << "**             1 --- 冒泡排序              **\n";
    cout << "**             2 --- 选择排序              **\n";
    cout << "**             3 --- 直接插入排序          **\n";
    cout << "**             4 --- 希尔排序              **\n";
    cout << "**             5 --- 快速排序              **\n";
    cout << "**             6 --- 堆排序                **\n";
    cout << "**             7 --- 归并排序              **\n";
    cout << "**             8 --- 基数排序              **\n";
    cout << "**             9 --- 退出程序              **\n";
    cout << "============================================\n\n";
    while (1) {
        cout << "请输入要产生的随机数个数:";
        cin >> size;
        if (size > 0 && size <= MAX) break;
        cout << "输入的数字大小在0~10000，请重新输入！" << endl;
    }

    while (1) {
        set(array, size);
        cout << "\n请选择排序算法:";
        char select;
        cin >> select;
        iCount = 0;
        switch (select) {
            case '1': {
                before = clock();
                bubbleSort(array);
                after = clock();
                cout << "冒泡排序算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '2': {
                before = clock();
                simpleSelectSort(array);
                after = clock();
                cout << "选择排序算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '3': {
                before = clock();
                insertSort(array);
                after = clock();
                cout << "选择直接插入算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '4': {
                before = clock();
                shellSort(array);
                after = clock();
                cout << "选择希尔算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
```

```
            case '5': {
                before = clock();
                quickSort(array, 1, size);
                after = clock();
                cout << "快速排序算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '6': {
                before = clock();
                heapSort(array);
                after = clock();
                cout << "选择排序算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '7': {
                before = clock();
                mergeSort(array);
                after = clock();
                cout << "选择归并算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '8': {
                before = clock();
                radixSort(array);
                after = clock();
                cout << "选择基数算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
                cout << "交换次数:" << iCount << endl;
                break;
            }
            case '9': {
                return 0;
            }
            default: {
                cout << "输入错误！" << endl;
                break;
            }

        }

    }
}
```

**创建随机数组**

```
void set(int a[], int size) {
    for (int i = 0; i < size; i++)
        a[i] = rand();
}
```

## 2、时间计算

结束时间与开始时间之差，除以CLOCKS_PER_SEC获得真实时间。各算法计时方式类似，后面不再赘述。

```
before = clock();
bubbleSort(array);
after = clock();
cout << "冒泡排序算法排序所用时间为:\t" << difftime(after, before) / CLOCKS_PER_SEC << "秒\n";
```

## 3、冒泡排序

重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来，直到没有再需要交换的元素，即数列排序完成。

```
void bubbleSort(int a[]) { //冒泡排序
    int exchange = size;
    int bound;
    while (exchange != 0) {
        bound = exchange;
```

```
        exchange = 0;
        for (int i = 1; i < size; i++) {
            if (a[i] > a[i + 1]) {  //如果前一个数大于后面的数，交换
                int tmp = a[i];
                a[i] = a[i + 1];
                a[i + 1] = tmp;
                exchange = i;
                iCount++;
            }
        }
    }
}
```

## 4、选择排序

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。
2. 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
3. 重复第二步，直到所有元素均排序完毕。

```
void simpleSelectSort(int a[]) { //简单选择排序

    for (int i = 1; i < size; i++) {
        int index = i;
        for (int j = i + 1; j <= size; j++) //在无序区中选择最小数
            if (a[j] < a[index])
                index = j;
        if (index != i) {
            int tmp = a[i];
            a[i] = a[index];
            a[index] = tmp;
            iCount++;
        }
    }
}
```

## 5、直接插入排序

1. 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
2. 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置，如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。

```
void insertSort(int a[]) { //直接插入排序
    int j;
    for (int i = 2; i <= size; i++) {
        a[0] = a[i];
        for (j = i - 1; a[0] < a[j]; j--) { //寻找未排序的数
            a[j + 1] = a[j];
            iCount++;
        }
        a[j + 1] = a[0];
    }
}
```

## 6、希尔排序

1. 选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1。
2. 按增量序列个数k，对序列进行k 趟排序。
3. 每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```
void shellSort(int a[]) { //希尔排序
    int d; //增量
    int i, j;
    for (d = size / 2; d >= 1; d = d / 2) {
        for (i = d + 1; i < size; i++) {
            a[0] = a[i];
            for (j = i - d; j > 0 && a[0] < a[j]; j = j - d) //后移
                a[j + d] = a[j];
            a[j + d] = a[0];
```

```
            iCount++;
        }
    }
}
```

## 7、快速排序

1. 从数列中挑出一个元素，称为 “基准”。
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。

```
void quickSort(int a[], int first, int end) { //快速排序
    if (first < end) {
        int pivot = partition(a, first, end); //分区和返回基准
        quickSort(a, first, pivot - 1); //左侧快速排序
        quickSort(a, pivot + 1, end); //右侧快速排序
        iCount++;
    }
}

int partition(int a[], int first, int end) {//快速排序一次划分
    int i = first, j = end;
    while (i < j) {
        while (i < j && a[i] <= a[j]) { //右侧扫描
            j--;
        }
        if (i < j) {
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
            i++;
            iCount++;
        }

        while (i < j && a[i] <= a[j]) { //左侧扫描
            i++;
        }
        if (i < j) {
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
            j--;
            iCount++;
        }
    }
    return i; //返回基准
}
```

## 8、堆排序

1. 创建一个堆H[0..n-1]。
2. 把堆首（最大值）和堆尾互换。
3. 把堆的尺寸缩小1，并调用shift函数，目的是把新的数组顶端数据调整到相应位置。
4. 重复步骤2，直到堆的尺寸为1。

```
void heapSort(int a[]) { //堆排序
    for (int i = size / 2; i >= 1; i--)
        sift(a, i, size);
    for (int i = 1; i < size; i++) //移去对顶并重建大根堆
    {
        int tmp = a[1];
        a[1] = a[size - i + 1];
        a[size - i + 1] = tmp;
        sift(a, 1, size - i);
        iCount++;
    }
}

void sift(int a[], int k, int m) { //筛选法调整堆
    int i = k, j = 2 * k; //j指向左孩子
```

```
    while (j <= m) {
        if (j < m && a[j] < a[j + 1]) {
            j++;
        }
        if (a[i] > a[j])
            break;
        else {
            int tmp = a[i];
            a[i] = a[j];
            a[j] = tmp;
            i = j;
            j = 2 * i;
            iCount++;
        }
    }
}
```

## 9、归并排序

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列。
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置。
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置。
4. 重复步骤3直到某一指针达到序列尾。
5. 将另一序列剩下的所有元素直接复制到合并序列尾。

```
bool mergeSort(int a[]) {
    int *p = new int[size];
    if (p == NULL) return false;
    mergeSortN(a, 0, size - 1, p);
    iCount++;
    delete[] p;
    return true;
}

void mergeSortN(int a[], int first, int last, int temp[]) {
    if (first < last) {
        int mid = (first + last) / 2;
        mergeSortN(a, first, mid, temp);    //左边有序
        mergeSortN(a, mid + 1, last, temp); //右边有序
        merge(a, first, mid, last, temp); //再将二个有序数列合并
        iCount++;
    }
}

void merge(int a[], int first, int mid, int last, int temp[]) {//将有二个有序数列合并
    int i = first, j = mid + 1;
    int m = mid, n = last;
    int k = 0;

    while (i <= m && j <= n) {
        if (a[i] <= a[j]) temp[k++] = a[i++];
        else temp[k++] = a[j++];
        iCount++;
    }

    while (i <= m) {
        temp[k++] = a[i++];
        iCount++;
    }
    while (j <= n) {
        temp[k++] = a[j++];
        iCount++;
    }

    for (i = 0; i < k; i++) {
        a[first + i] = temp[i];
        iCount++;
    }
}
```

## 10、基数排序

1. 将给出的序列元素的个位进行收集，然后放到对应的位置（0-9序列），并根据个位排出大小，形成了一个序列。

2. 收集十位，根据第一步产生的序列放到对应的位置，形成一个新的序列。
3. 收集百位，根据第二步产生的序列放到对应的位置，形成一个新的序列。
4. 以此类推直至完成序列排序。

```
void radixSort(int *pDataArray) {
    int *radixArrays[10];    //分别为0~9的序列空间
    for (int i = 0; i < 10; i++) {
        radixArrays[i] = (int *) malloc(sizeof(int) * (size + 1));
        radixArrays[i][0] = 0;    //index为0处记录这组数据的个数
    }

    for (int pos = 1; pos <= 10; pos++)    //从个位开始到31位
    {
        for (int i = 0; i < size; i++)    //分配过程
        {
            int num = getNumInPos(pDataArray[i], pos);
            int index = ++radixArrays[num][0];
            radixArrays[num][index] = pDataArray[i];
        }

        for (int i = 0, j = 0; i < 10; i++)    //收集
        {
            for (int k = 1; k <= radixArrays[i][0]; k++) { pDataArray[j++] = radixArrays[i][k]; }
            radixArrays[i][0] = 0;    //复位
        }
    }
}

int getNumInPos(int num, int pos) { // 找到num的从低到高的第pos位的数据
    int temp = 1;
    for (int i = 0; i < pos - 1; i++) {
        temp *= 10;
    }
    return (num / temp) % 10;
}
```

## 总结

| 排序 | 平均时间 | 最差情形 | 稳定性 | 额外空间 | 备注 |
| --- | --- | --- | --- | --- | --- |
| 冒泡 | O(n^2) | O(n^2) | 稳定 | O(1) | n小时较好 |
| 交换 | O(n^2) | O(n^2) | 不稳定 | O(1) | n小时较好 |
| 选择 | O(n^2) | O(n^2) | 不稳定 | O(1) | n小时较好 |
| 插入 | O(n^2) | O(n^2) | 稳定 | O(1) | 大部分已排序时较好 |
| 基数 | O(logrd) | O(logrd) | 稳定 | O(n) | d是关键字项数（0-9）r是基数 |
| 希尔 | O(nlogn) | O(n^s) | 不稳定 | O(1) | s是所选分组 1<s<2 |
| 快速 | O(nlogn) | O(n^2) | 不稳定 | O(nlogn) | n大时较好 |
| 归并 | O(nlogn) | O(nlogn) | 稳定 | O(1) | n大时较好 |
| 堆 | O(nlogn) | O(nlogn) | 不稳定 | O(1) | n大时较好 |

## 用例演示

```
**                  排序算法比较                  **
==============================================
**                1 --- 冒泡排序                  **
**                2 --- 选择排序                  **
**                3 --- 直接插入排序              **
**                4 --- 希尔排序                  **
**                5 --- 快速排序                  **
**                6 --- 堆排序                    **
**                7 --- 归并排序                  **
**                8 --- 基数排序                  **
**                9 --- 退出程序                  **
```

```
==========================================

请输入要产生的随机数个数:10000

请选择排序算法:1
冒泡排序算法排序所用时间为:  0.363478秒
交换次数:24873991

请选择排序算法:2
选择排序算法排序所用时间为:  0.093741秒
交换次数:9984

请选择排序算法:3
选择直接插入算法排序所用时间为:    0.071835秒
交换次数:24846084

请选择排序算法:4
选择希尔算法排序所用时间为:  0.00278秒
交换次数:119992

请选择排序算法:5
快速排序算法排序所用时间为:  0.001574秒
交换次数:59508

请选择排序算法:6
选择排序算法排序所用时间为:  0.002051秒
交换次数:124303

请选择排序算法:7
选择归并算法排序所用时间为:  0.001983秒
交换次数:277232

请选择排序算法:8
选择基数算法排序所用时间为:  0.004183秒
交换次数:0

请选择排序算法:9

Process finished with exit code 0
```