数据结构课程设计

1751650 蒋伟博

8 种排序算法的比较案例

0. 项目简介

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，系统产生随机数。并且显示他们的比较次数。

1. 算法分析

接下来我们将逐个分析排序算法的特点、时间复杂度和空间复杂度。

1. 冒泡排序

冒泡排序可能是最简单、最容易实现的算法了，它走访要排序的数列，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。这样从头到尾的走访过程要执行n次，并且不必每次从头执行到尾。
假设共有n个数字需要进行排序，那么在第i次排序中，我们扫描第1至第(n-i+1)个数字，并且在必要时交换相邻的两个元素。经过这样一次排序，第(n-i+1)个数字就变为正确的了，于是我们在下一次排序中就不需要再访问这个元素了，也就是说，冒牌排序每次扫描的数字都比上一次少一个。这其实是一种“狗熊掰棒子”的做法，我们每次循环至少可以得到一个在剩余的玉米中最大的一个，并将它放在正确的位置上。
代码展示如下

```
        for (int sizeNow = size; sizeNow >= 2; sizeNow--)
                for (int idx = 0; idx < sizeNow - 1; idx++)
                        if (ans[idx] > ans[idx + 1])
                        {
                                swap(ans[idx], ans[idx + 1]);
                                swaptime++;
                        }
}
```

可见，冒泡排序确实是一种简单易懂的算法。但是相对来说，冒泡排序的时间复杂度极为不理想。在最坏情况下，即要排序的数字完全是倒序排列，这样我们每执行一次循环，只能将一个数字放在正确的位置上，并且每次比较都需要交换两个数字，这是一笔非常大的开销，它的时间复杂度为O(n^2)。即使对于平均情况而言，它的时间复杂度也为O(n^2)。
冒泡排序对空间的要求不高，既不需要辅助数组，也不执行递归调用，所以它的空间复杂度为O(1)。
由于我们只在后一个元素大于前一个元素的前提下，才会交换两个元素的位置，所以冒泡排序是一种稳定的排序算法。

2. 选择排序

选择排序与冒泡排序有些相似之处，而且也很容易理解。选择排序同样要执行n次遍历操作，每次遍历都要找出当前数组中最大（或最小）的元素，并将它与当前数组的尾部元素交换位置。同样地，选择排序不需要每次从头到尾遍历，它在第i次只遍历第1到第(n-i+1)个元素，从中找出最大的，将它与第(n-i+1)个元素交换位置。
直观来看，选择排序也是一种“狗熊掰棒子”算法，但是它实际上省掉了每次“用更大的棒子替换手中的棒子”这个过程，它相当于狗熊默默地在心中记下最大的棒子的位置，并在检查完所有剩余的棒子后拿走那个最大的棒子。
代码如下

```
void Sort::selection()
{
	int minIndex;
	for (int startPos = 0; startPos < size - 1; startPos++)
	{
		minIndex = startPos;
		for (int idx = startPos + 1; idx < size; idx++)
			if (ans[idx] < ans[minIndex])
				minIndex = idx;
		if (minIndex != startPos)
		{
			swap(ans[minIndex], ans[startPos]);
			swaptime++;
		}
	}
}
```

由于省掉了每次比较后的交换过程（实际上选择排序总共只执行(n-1)交换），所以面对相同的数据量，选择排序远快于冒泡排序，这一点在前面的展示中也体现出来。但是，虽然速度快了不少，选择排序与冒泡排序的时间复杂度是一样的。在最坏情况下，时间复杂度为O(n^2)，平均情况下时间复杂度也为O(n^2)。
选择排序的空间复杂度也为O(1)，同样不要辅助数组和递归调用。
由于选择排序不能保证在交换两个元素时不破坏相同元素的前后顺序，所以是不稳定的。

3. 直接插入排序

插入排序同样是简单易懂的排序方法之一，其中心思想为将每个数字不断插入已排序好的一个数列中，并保证插入后的数列仍为有序数列。实现方法也是简单的：我们从待排序数列的第二个元素开始（因为第一个元素自身必为有序数列），每次将其与它前一个元素比较，若它比前一个元素小，则将它们交换位置，直至不能交换。因为第i个元素的前(i-1)个数字为有序的，所以当第i个元素交换停止时，前i个数字为有序的。
代码如下

```
void Sort::insertion()
{
	for (int newArraySize = 0; newArraySize < size; newArraySize++)
	{
		int idxToInsert = newArraySize;
		for (; idxToInsert > 0; idxToInsert--)
		{
			if (source[newArraySize] < ans[idxToInsert - 1])
				ans[idxToInsert] = ans[idxToInsert - 1];
```

```
                        else break;
                }
                ans[idxToInsert] = source[newArraySize];
        }
}
```

直接插入排序的时间复杂度与前两种排序方法一致，最坏情况下为O(n^2)，平均情况下为O(n^2)。空间复杂度的情况也一样，为O(1)。
直接插入排序是顺序地选择元素并与前一个元素比较，并且只在后一个元素小于前一个元素时才进行交换，所以是稳定的。

值得说明的情况

介绍了冒泡排序、选择排序和直接插入排序，我们能够发现，这三种简单排序算法的时间复杂度情况完全一致，而理论证明这并不是巧合。在这里我不会证明这一点，但会做出简要的阐述：我们规定对于数组A中，当i>j时，A[i]<A[j]的一组数字成为逆序，而这种数对的个数成为逆序数。可以证明：

> N个互异数的数组平均逆序数为N(N-1)/4

而以上三种简单排序方法，每次交换（或比较）都只能直接（或间接）消除一个逆序数，所以他们的平均时间复杂度为O(n^2)。事实上，我们还有一个定理：

> 通过交换相邻元素进行排序的任何算法平均需要O(n^2）时间

这指出了：如果我们想要突破O(n^2)的时间界，那么每次交换都要消除不止一个逆序。

4. 希尔排序

希尔排序(Shellsort)的名字来源于它的发明者Donald Shell，它也是最早突破二次时间界的算法之一。它通过比较相距一定间隔的元素来工作，每趟比较所用的距离随着算法的进行而减小，直到只比较相邻元素的最后一趟排序为止。
希尔排序需要使用一个序列`h1, h2, h3, ..., ht`，这个序列中每个元素对应希尔排序每次比较所用的距离。我们从ht开始，到h1结束。
在希尔排序的每趟排序中，我们使用直接插入排序，所以自然可以推出：只要h1=1，那么任何一个序列都是可行的（甚至这个序列只有h1）。但是，有些序列比另一些序列更好，消耗的时间更短。
在使用增量hk的一趟排序后，所有相隔hk的元素都被排序，此时称数列时hk-排序的。希尔排序得以成立的保证在于：

> 一个hk-排序的数列将一直保持它的hk-排序性

在此我们不做证明。
代码如下

```
        for(int gap=size/2; gap>0; gap/=2)
        {
                //shell排序内部是插入排序
                for (int i = gap;i < size;++i)
                {
                        for (int j = i;j >= gap;j -= gap)
```

```
                        {
                                if (ans[j] < ans[j - gap])
                                {
                                        swap(ans[j], ans[j - gap]);
                                        swaptime++;
                                }
                        }
                }
                if(gap==1) break;//gap==1等效于一次插入排序，一定现在已经排序好了
        }
}
```

希尔排序通过比较两个间隔较远的元素，使得一次交换能够消除多个逆序，大大提高了运行速度。但是，根据使用的增量序列的不同，得到的时间界也有所不同。可以证明：

> 使用希尔增量时希尔排序的最坏情形运行时间为O(n^2)

如果我们换一组增量，如我在实现中使用到的Hibbard增量，这种增量形如1, 3, 7, ..., 2^k-1，在这种增量序列下最坏的时间复杂度为O(n^3/2)，这是经过严格证明的；而平均情况下的时间复杂度无法进行证明，但是在实践中人们大致上估计出了其平均时间复杂度为O(n^5/4)。如果使用Sedgewick增量，平均时间复杂度可以进一步优化到O(n^7/6)，最坏时间复杂度为O(n^4/3)。
希尔排序的空间复杂度为O(1)。
由于希尔排序每次比较、交换的两个元素位置间隔较远，不能保证不破坏相同元素的前后顺序，所以是不稳定的。

5. 快速排序

顾名思义，快速排序是目前在实践中最快的已知算法，但是由于需要进行高度优化才能达到最优的效果。在之前的比较中，当随机数量达到10^7时，快速排序相对其它使用比较来进行排序的算法，在时间上处于领先的地位。
快速排序是一种递归的算法，它的操作十分简单，设有数组A：

1. 如果A中的元素个数为0或1，则返回
2. 取A中任一元素v，称之为枢纽元
3. 将A - {v}(A中剩余的元素)分成两个不相交的集合：A1={x <= v}和A2={x > v}。
4. 返回{quicksort(A1), v, quicksort(A2)}。

至此，我们就可以的到排序后的数组A了。
对于快速排序的理论分析是比较容易的，但是在实践中，却很难实现，或者说使快速排序达到一个较好的时间界。这种困难在于，对枢纽元的选择。
通过选择不同的枢纽元，我们可以对数组A进行不同的划分。最理想的划分情况应该是：将大约一半的元素划入数组A1，另一半划入A2，很像一棵保持平衡的二叉查找树。这就要求我们每次选择合理的枢纽元。不幸的是，我们并不能一眼发现那个恰为中位数的枢纽元，我们只能尽可能地去寻找接近的数字；而且这个寻找的时间必须是常数的，因为我们不能在这上面浪费太多时间。
一种简单但是不好的方法是：每次选择第一个元素作为划分，这对于随机的数列（比如本题目）是勉强可以接受的，但是对于一些已经预排序的数列或者倒序的数列来说，则是相当糟糕的。对于预排序数列，虽然递归一直在进行，元素之间的比较也一直在进行，但是到最后算法没有做任何事情；对于倒序数列，这样划分下的快速排序实质上退化为冒泡排序，时间复杂度达到了O(n^2)，这显然是不能接受的。
另一种更好的方法是：随机从要排序的序列中选择一个元素，作为枢纽元。因为是随机地选择，所以几乎不可

能一直选择到最坏的选择，所以平均来看，这样的选择是可以接受的。我在实现中就是选用了这种方法。
代码如下

```
int quickSort(int *a,int left, int right)
{
        if(left>=right) return 0;
        int swaptime=0;
        int i = left, j = right;//分别指向数组的头尾
        int key=a[left];
        while (i < j)//若两数字交错，则退出循环
        {
                while (i<j && key < a[j])
                        --j;//找到右侧第一个小于基准元的数
                if(i<j)
                {
                        a[i++]=a[j];
                        swaptime++;
                }
                while (i<j && key > a[i])
                        ++i;//找到左侧第一个大于基准元的数
                if(i<j)
                {
                        a[j--]=a[i];
                        swaptime++;
                }
        }
        a[i]=key;
        swaptime++;

        swaptime+=quickSort(a,left, i - 1);//继续处理左半部分
        swaptime+=quickSort(a,i + 1, right);//继续处理右半部分
        return swaptime;
}
void quickSort(int *source,int size,int *&ans,int& swaptime,clock_t& time)
{
        ans = new int[size];
        memcpy(ans, source, size * sizeof(int));
        swaptime = 0;
        clock_t startTime, endTime;
        startTime = clock();

        swaptime=quickSort(ans,0,size-1);

        endTime = clock();
        time = endTime - startTime;
}
```

与之前介绍的算法有所不同的是，快速排序作为一种递归算法需要一个驱动程序，在这里是quick()，它负责处理一些准备工作，并调用第一个递归函数。
在具体的实现中，枢纽元选取后就不应该继续留在待处理的序列中了，因为处理时可能移动枢纽元的位置，导致不可预料的错误。正确的做法是，将选渠道的枢纽元放到待处理序列的尾部（或首部），在处理时避开枢纽

元的位置即可。在我的代码中，是将枢纽元放到待处理序列的最右端（尾部），然后处理left到right-1位置上的数字。
正如之前所言，快速排序在最坏的情况下，时间复杂度为O(n^2)；在平均情况下，时间复杂度可以达到O(nlogn)，可以说是十分理想了。
由于快速排序为递归函数，所以需要消耗栈空间。在最坏情况下，空间复杂度为O(n)，此时完全是按照{1, n-1}的数量划分数组；在平均情况下，空间复杂度为O(logn)。
快速排序在每次遍历待排序数组时，都是由两头向中间移动，交换元素时可能破坏相同元素的前后顺序，所以是不稳定的。

6. 堆排序

堆是一棵完全二叉树，分为最大堆和最小堆；以最小堆为例，它的父节点永远小于等于其儿子节点的值。这两点可以总结为堆序性和结构性。除此之外，堆的删除和插入算法的时间复杂度都为O(logn)，且当我们不断删除（弹出）堆的根节点时，得到的序列是有序的，这启发我们利用堆的性质发展处一种算法，这就是堆排序。
正如之前所说，堆排序是容易的，从总体来看，它包含两个步骤：

1. 建堆
2. 不断删除堆顶元素（根节点），直至堆为空。

当然，每删除一次堆顶元素，都要执行一次下滤保证堆序性。
我们将每次从堆中删除的元素按顺序排列，就可以得到一个有序数列。值得注意的是，由于我们每删除一个堆中的元素，堆原本所占用的数组就会空出一个位置来（这是堆的结构性保证的），位于新的堆的最后一个元素的后一个位置。我们完全可以将删除的元素放在空出的位置上，从而节省一定的空间。但是由于这样做，我们实质上是将删除产生的序列倒序排放了，所以建堆时要好好规划堆的性质。比如想要得到一个升序的序列，那就要建立一个最大堆，这样我们每次得到堆中最大的元素，并将它们从数组的尾部向前排列，这样最终得到的序列才会是升序的。
代码如下

```
int percDown(int* a,int downIndex, int endIndex)
{
        int swaptime=0;
        int childIndex;//它将指向两个儿子中较大的那一个
        for (int i = downIndex;2 * i + 1 < endIndex;i = childIndex)
        {
                childIndex = 2 * i + 1;//现在指示的是左儿子的坐标
                                                        /*存在的前提下，若右儿子
大于左儿子，则将childIndex改为左儿子坐标*/
                if (childIndex != endIndex - 1 && a[childIndex] < a[childIndex +
1])
                        ++childIndex;
                if (a[i] < a[childIndex])
                {
                        swap(a[i], a[childIndex]);//元素下滤
                        swaptime++;
                }
                else//下滤的元素找到了合适的位置；恢复了堆序性
                        break;
        }
        return swaptime;
}
```

```
void heapSort(int *source,int size,int *&ans,int& swaptime,clock_t& time)
{
        ans = new int[size];
        memcpy(ans, source, size * sizeof(int));
        swaptime = 0;
        clock_t startTime, endTime;
        startTime = clock();

        for (int i = size / 2;i >= 0;--i)//建堆,最大的元素位于根部
                swaptime+=percDown(ans,i, size);
        for (int i = size - 1;i > 0;--i)
        {
                /*将堆中最大元素排在排序区头部，排序区整体上为由小到大*/
                swap(ans[i], ans[0]);
                swaptime++;
                swaptime+=percDown(ans,0, i);
        }

        endTime = clock();
        time = endTime - startTime;
}
```

我们将下滤的算法单独摘出来，是为了便于理解，增加代码的可读性。对于时间复杂度的分析也是容易的：建堆的时间复杂度为O(n)；每次下滤的时间复杂度为O(logn)，删除的过程共执行(n-1)次下滤，则总的时间复杂度为O(nlogn)。在最坏情况下和平均情况下的时间复杂度都是如此。
堆排序不是递归算法，也不需要额外的存储空间（辅助数组），所以其空间复杂度为O(1)。
由于无法保证相同元素的出堆顺序，所以堆排序是不稳定的排序算法。

7. 归并排序

归并排序是建立在归并操作基础上的一种排序方法，而归并操作是典型的分治算法的应用。除此之外，我们所熟知的二分查找也是一种典型的分治算法应用。在归并操作中，我们有三个主要步骤：

1. 分解：将要解决的问题划分为几个要解决的子问题
2. 求解：当子问题足够小时，对问题进行求解
3. 合并：将子问题的解按顺序合并，即得到原问题的解

具体到归并排序中，我们这样操作：将一个大的待排序数组一分为二，得到两个子数组，再将两个子数组分别一分为二，得到四个子数组.....如此操作，直到得到的子数组中只有一个元素，此时子数组中的排序任务自动完成。接下来我们将排好序的子数组按递归的层次顺序进行合并，最终就得到了一个有序的数组。
代码如下

```
void mergeSort(int *a,int startidx,int length)
{
        if(length<=1) return;
        mergeSort(a,startidx,length/2);
        mergeSort(a,startidx+length/2,length-length/2);

        int *ans=new int[length];
        int left=0,right=length/2,ansidx=0;
```

```
        while(left!=length/2&&right!=length)
        {
                if(a[startidx+left]<=a[startidx+right])
                {
                        ans[ansidx]=a[startidx+left];
                        left++;
                        ansidx++;
                }
                //两个if就要判断循环条件
                if(left!=length/2&&a[startidx+left]>=a[startidx+right])
                {
                        ans[ansidx]=a[startidx+right];
                        right++;
                        ansidx++;
                }
        }
        for(; left!=length/2; left++)
                ans[ansidx++]=a[startidx+left];
        for(; right!=length; right++)
                ans[ansidx++]=a[startidx+right];

        for(int i=0; i<length;i++)
                a[startidx+i]=ans[i];
        delete[] ans;
}
void mergeSort(int *source,int size,int *&ans,int& swaptime,clock_t& time)
{
        ans = new int[size];
        memcpy(ans, source, size * sizeof(int));
        swaptime = 0;
        clock_t startTime, endTime;
        startTime = clock();

        mergeSort(ans,0,size);

        endTime = clock();
        time = endTime - startTime;
}
```

相比其它算法，归并排序的算法实现略微复杂一些。首先，归并排序是一种递归算法，所以它需要一个驱动函数；其次，合并两个有序数组的操作略显冗长，所以我将其独立为一个函数。另外值得注意的一点是，辅助数组名为tempArr，当归并排序的所有递归操作完成后，它实际上也是有序的了（因为我们确保了每次合并只会在要合并的数组自己的部分进行，不会干扰到其它区域）。所以如果单独调用归并排序，就可以省掉“将排序好的元素从临时数组中拷贝回操作数组”这一操作，但是这里因为是使用类内成员进行操作，所以必须进行拷贝，不然check操作会出现错误。

归并操作的时间复杂度是极为优秀的。由于我们将子数组细分到只有一个元素为止，所以实质上不存在最坏情况与平均情况之分：因为根本没有排序这个操作。将待排序数组进行划分的时间复杂度为O(logn)，而合并两个有序数组的时间复杂度为O(n)，且一次合并对应一次划分，所以总的时间复杂度为O(nlogn)。

归并操作的辅助数组占用的空间为O(n)，递归时消耗的栈空间为O(logn)，所以其总的时间复杂度为O(n)。可见，在已给出的几种算法中，归并排序的空间复杂度是最高的。但是相对的，归并排序在实践中的速度仅次于快速排序，是一种十分高效、容易理解的排序算法。

由于归并排序不能保证在合并操作时，两个分别处于不同数组的相同元素的前后顺序保持不变，所以是不稳定的。

8. 基数排序

基数排序有些类似于桶排序，但是具体操作步骤不太一样。有些人将基数排序视为桶排序，我认为这是不严格的，具体原因在这一部分的最后会变得清楚。
基数排序是一种分配式排序，例如对于整数，我们要做的是按不同的位（个位、十位等）的数值，将待排序序列分为十组（0~9），将分好组的元素按顺序拷贝回原数组，再按下一位用同样的方法分组.....以此类推，当我们按最后一位分类完毕后，整个序列就变为有序的了。
按照由低位到高位和高位到低位的次序不同，基数排序分为两类：MSD(最高位优先)和LSD(最低位优先)。我们在这里使用的算法是LSD，即分类由最低位增加至最高位，当最高位分类结束后，序列即为有序。
在具体操作中，我们申请一个大的数组空间（与待排序序列占用空间一样大）作为桶，用它来存放中间分类过程的结果，并且用一个10个int大小的数组记录每个数字对应元素的个数，并将它转化为每个数字对应元素在桶中的结束位置。这样我们就可以将待排序数组中的元素放入桶中相应的位置上了，之后再按顺序（必须保证这一点）取出，放回原数组即可。
从表面上看，基数排序的原理似乎难以理解，甚至可能去怀疑基数排序究竟有没有效果——答案是肯定的，奥秘就在于桶中的元素按顺序取出这一点。可以使用数学归纳法证明：

1. 取最低位(第1位)进行基数排序，得到的结果，仅看最低位，序列是有序的；
2. 假设对第i位做过基数排序后，仅看前i位，序列是有序的，那么对于第(i+1)位进行基数排序时，有：
   当数字的第(i+1)位相等时，它们之间的相对位置关系不会改变，此时仅看前(i+1)位，它们仍为有序的；
   当数字的第(i+1)位不相等时，它们将按照第(i+1)位的大小进行排序，此时仅看前(i+1)位，它们仍为有序的。
   如此一来，当我们进行完第(i+1)位的排序后，仅看前(i+1)位，序列就是有序的。
3. 由1和2知，待排序序列(最大数字为n位)经过n位基数排序后将是有序的。

所以，保证当数字的第(i+1)位相等时，它们之间的相对位置关系不会改变这一点的关键，就在于将一次基数排序排序好的序列从桶中取出时，保证顺序不会改变。
代码如下

```
int maxBit(int *a,int size)
{
        int maxNum = 0;
        for (int i = 0;i < size;++i)
        {
                if (maxNum < a[i])
                        maxNum = a[i];
        }
        int max = 1, ruler = 10;
        while (maxNum >= ruler)
        {
                ++max;
                ruler *= 10;
        }
        return max;
}
void radixSort(int *source,int size,int *&ans,int& swaptime,clock_t& time)//LSD方法实现
```

```
{
        ans = new int[size];
        memcpy(ans, source, size * sizeof(int));
        swaptime = 0;
        clock_t startTime, endTime;
        startTime = clock();

        int max = maxBit(source,size);
        int* bucket = new int[size];//桶（将来的所有桶都在里面分配空间）
                                                                /*计数器，第一阶段
计算某位为index的数据数量，第二阶段标记每个桶的结束位置*/
        int* count = new int[10];

        for (int i = 0, radixNum = 1;i < max;++i)//按位数进行max次排序，radixNum为基
数
        {
                for (int j = 0;j < 10;++j)//每次排序前初始化计数器
                        count[j] = 0;
                for (int j = 0;j < size;++j)
                {
                        /*拿到数据第i位上的数字作为索引*/
                        int index = (ans[j] / radixNum) % 10;
                        ++count[index];//数据量加1
                }
                for (int j = 1;j < 10;++j)//为每个桶分配空间
                        count[j] = count[j - 1] + count[j];//桶的结束位置

                for (int j = size - 1;j >= 0;--j)//将数据放入对应的桶中，保持稳定性
                {
                        int index = (ans[j] / radixNum) % 10;
                        /*count[index]-1是数据在桶中的位置*/
                        bucket[count[index] - 1] = ans[j];
                        --count[index];//对应数据量减1
                }
                for (int j = 0;j < size;++j)
                        ans[j] = bucket[j];//将桶中数据拷贝到操作数组中
                radixNum *= 10;//基数增加
        }

        delete[] bucket;
        delete[] count;
        count = bucket = NULL;

        endTime = clock();
        time = endTime - startTime;
}
```

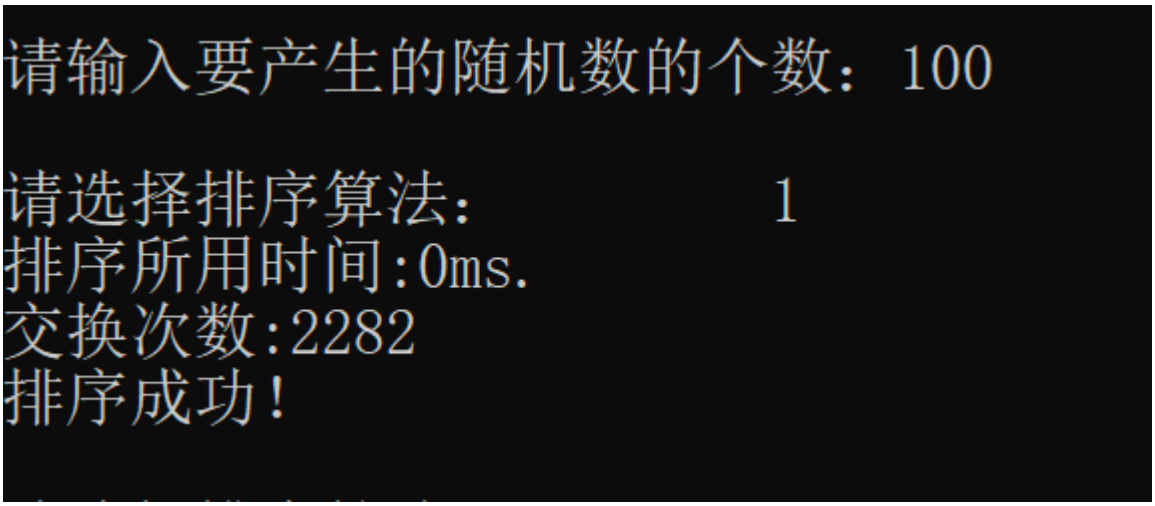
其中，maxBit的作用是计算待排序序列中最大元素的位数。
基数排序的时间复杂度为O(d(n+r))，其中d为最大元素的位数，r为基数的个数，这里为10(0~9)。
基数排序的空间需求在于两个辅助数组，分别用于储存中间过程的结果和每个基数对应的开始位置，故空间复杂度为O(n+r)。
由于我们保证了前i位有序的数字的相对位置不变，所以基数排序是一种稳定的排序算法。

算法分析结果总结

排序方式	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
冒泡排序	O(n^2)	O(n^2)	O(1)	稳定
选择排序	O(n^2)	O(n^2)	O(1)	不稳定
直接插入排序	O(n^2)	O(n^2)	O(1)	稳定
希尔排序	O(n^7/6)	O(n^4/3)	O(1)	不稳定
快速排序	O(nlogn)	O(n^2)	O(logn)	不稳定
堆排序	O(nlogn)	O(nlogn)	O(1)	不稳定
归并排序	O(nlogn)	O(nlogn)	O(n)	不稳定
基数排序	O(d(n+r))	O(d(n+r))	O(n+r)	稳定

2. 项目运行效果

- 输入随机数个数，并选择排序类型
  在程序运行之初，用户首先会被要求输入随机数的个数。输入完毕后，会被要求选择排序的类型。

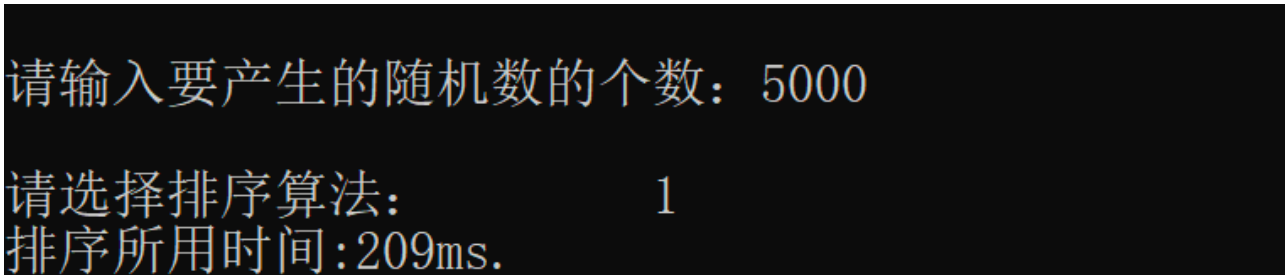


经过实践发现，项目示例中的比较次数可能存在错误，但这并不影响我们比较各种排序算法的时间复杂度；而各种排序方案，除了基数排序外，在空间上的差距并不大。

3. 时间测试

在本项测试中，将不同的随机数数量设置为不同的等级，规定某一排序算法消耗时间超过1s，即告“淘汰”，无法进入下一等级的测试。

- 5000个随机数



```
交换次数:6181638
排序成功!

请选择排序算法:          2
排序所用时间:34ms.
交换次数:4985
排序成功!

请选择排序算法:          3
排序所用时间:28ms.
交换次数:6186638
排序成功!

请选择排序算法:          4
排序所用时间:89ms.
交换次数:60885
排序成功!

请选择排序算法:          5
排序所用时间:2ms.
交换次数:27578
排序成功!

请选择排序算法:          6
排序所用时间:3ms.
交换次数:57126
排序成功!

请选择排序算法:          7
排序所用时间:3ms.
交换次数:0
排序成功!

请选择排序算法:          8
排序所用时间:1ms.
交换次数:0
排序成功!
```

- 20000个随机数

```
请输入要产生的随机数的个数：20000

请选择排序算法：          1
排序所用时间:3807ms.
交换次数:99999589
排序成功!

请选择排序算法：          2
排序所用时间:532ms.
交换次数:19991
排序成功!

请选择排序算法：          3
排序所用时间:382ms.
交换次数:100019589
排序成功!

请选择排序算法：          4
排序所用时间:1208ms.
交换次数:377989
排序成功!

请选择排序算法：          5
排序所用时间:4ms.
交换次数:131100
排序成功!

请选择排序算法：          6
排序所用时间:11ms.
交换次数:268510
排序成功!
```

```
排序成功!

请选择排序算法:          7
排序所用时间:13ms.
交换次数:0
排序成功!

请选择排序算法:          8
排序所用时间:4ms.
交换次数:0
排序成功!
```

- 50000个随机数

```
请输入要产生的随机数的个数: 50000

请选择排序算法:          1
排序所用时间:22851ms.
交换次数:625534181
排序成功!

请选择排序算法:          2
排序所用时间:3316ms.
交换次数:49986
排序成功!

请选择排序算法:          3
排序所用时间:2346ms.
交换次数:625584181
排序成功!

请选择排序算法:          4
排序所用时间:6448ms.
交换次数:1185053
排序成功!

请选择排序算法:          5
排序所用时间:9ms.
交换次数:365324
排序成功!
```

```
请选择排序算法:          6
排序所用时间:35ms.
交换次数:737537
排序成功!

请选择排序算法:          7
排序所用时间:31ms.
交换次数:0
排序成功!

请选择排序算法:          8
排序所用时间:5ms.
交换次数:0
排序成功!
```