8中排序算法的比较案例

1.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，系统产生随机书。并且显示他们的比较次数。

2.各个算法简介

- **冒泡排序**
  它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。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。
- **选择排序**
  选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。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，然后，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。以此类推，直到所有元素均排序完毕。
- **直接插入排序**
  插入排序（英语：Insertion Sort）是一种简单直观的排序算法。它的工作原理是通过构建有序序列，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。插入排序在实现上，通常采用in-place排序，因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。
- **希尔排序**
  希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。希尔排序是非稳定排序算法。 希尔排序是基于插入排序的以下两点性质而提出改进方法的：
  - 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率
  - 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位
- **快速排序**
  快速排序（英语：Quicksort），又称划分交换排序（partition-exchange sort），一种排序算法，最早由东尼·霍尔提出。事实上，快速排序通常明显比其他 O(n*log n)算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。
- **堆排序**
  堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积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。
- **归并排序**
  归并操作（merge），也叫归并算法，指的是将两个已经排序的序列合并成一个序列的操作。归并排序算法依赖归并操作。
- **基数排序**
  基数排序（英语：Radix sort）是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。基数排序的发明可以追溯到1887年赫尔曼·何乐礼在打孔卡片制表机（Tabulation Machine）上的贡献。
  它是这样实现的：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。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。

> 以上算法简介均摘自Wikipedia&百度百科

3.程序结构

- **声明的数据结构**

```
#define MAX 0x7fffffff
int myTemp[10000];                                    //临时暂用数组1
```

```
int myTemp1[10000];                                     //临时暂用数组2
int Count=0;                                            //记录交换数的次数
class mySystem{                                         //抽象出整个程序系统
public:
    mySystem();
    ~mySystem(){};
    void processing();                                  //运行排序系统
    void randomGenerate(int num);                       //随机产生随机数
    int* getNums(){return numbers;}                     //获得需要排序的数组
    int* bubbleSort(int Nums[],int length);             //冒泡排序
    int* simSelSort(int Nums[],int length);             //简单选择排序
    int* insertSort(int Nums[],int length);             //插入排序
    void heapSort(int Nums[],int length);               //堆排序
    int* shellSort(int Nums[],int length);              //希尔排序
    void mergeSort(int answers[],int start,int end);    //归并排序
    void quickSort(int answers[],int low,int high);     //快速排序
    int* radixSort(int Nums[],int length);              //桶排序
    int* Copy(int Nums[],int length);                   //获得原数组的拷贝
    void display(int* myArr);
private:
    int numbers[10000];                                 //保存产生的随机数组
    int num;                                            //产生的随机数个数
    void ajustDown(int Nums[],int i,int length);        //堆排序需要的自上而下的调整
    void buildMaxHeap(int Nums[],int length);           //堆排序需要的建立最大堆函数
    void QuickSort(int answer[],int low,int high);      //快速排序需要的递归调用函数
    void _Merge(int Nums[],int start,int middle,int end); //归并排序需要的子函数
    void MergeSort(int Nums[],int start,int end);         //归并排序需要调用的子函数
    int getMaxLen(int Nums[],int length);               //获得数组中数的最大长度
};
```

- **产生随机数**
  使用srand种下时间种子，产生num个随机数储存在numbers数组中

```
void mySystem::randomGenerate(int num) {
    srand(time(NULL));                                  //用时间作为随机种子
    for(int i=0;i<num;i++){                             //产生num个随机数
                numbers[i]=rand();
    }
}
```

- **处理用户输入调用排序函数**
  检查用户是否合法，对用户输入进行检查，同时使用switch跳转表调用排序函数

```
void mySystem::processing() {                           //处理所有的排序算法调用
    int operand=0;
    while(operand!=9){
        cout<<"请选择排序算法:";
        cin>>operand;
        if(operand<1||operand>9){                       //检查输入是否合法
            cout<<"您输入的数字不在1-9之间，请重新输入"<<endl;
            cin.clear();
            cin.ignore();
            continue;
        }
        switch (operand){
            case 1:{
                auto ptr=bubbleSort(numbers,num);
                free(ptr);
                break;
            }
            case 2:{
                auto ptr=simSelSort(numbers,num);
                free(ptr);
                break;
            }
```

```
            case 3:{
                auto ptr=insertSort(numbers,num);
                free(ptr);
                break;
            }
            case 4:{
                auto ptr=shellSort(numbers,num);
                free(ptr);
                break;
            }
            case 5:{                                        //拷贝数组以免排序时改变原数组顺序
                auto ptr=Copy(numbers,num);
                quickSort(ptr,0,num-1);
                free(ptr);
                break;
            }
            case 6:{                                        //拷贝数组以免排序时改变原数组顺序
                auto ptr=Copy(numbers,num);
                heapSort(ptr,num);
                free(ptr);
                break;
            }
            case 7:{                                        //拷贝数组以免排序时改变原数组顺序
                auto ptr=Copy(numbers,num);
                mergeSort(ptr,0,num-1);
                free(ptr);
                break;
            }
            case 8:{
                auto ptr=radixSort(numbers,num);
                free(ptr);
                break;
            }
        }
    }
}
```

- **拷贝数组函数**
  由于很多排序中需要保留原排序数组，所以需要拷贝一个新的数组供排序使用，参数传入需要拷贝的数组和数组的大小。

```
int* mySystem::Copy(int Nums[],int length){             //拷贝数组并返回指向拷贝数组的指针
    auto ptr=(int*)malloc(sizeof(int)*length);
    for(int i=0;i<length;i++){
        ptr[i]=Nums[i];
    }
    return ptr;
}
```

- **计时结构** 由于每个算法都需要计时并输出整个排序的时间，所以这里抽象出一个专门计时的结构。

> 由于此处已经将计时结构抽象出来描述，所以后面8个排序算法中都不对计时结构进行特殊说明

```
clock_t start,end;                                  //开始计时时间和结束计时时间
start=clock();
// sort code
//...
end=clock();
double duration=(double)(end-start)/CLOCKS_PER_SEC;             //需要除CLOCKS_PER_SEC获得真实时间
cout<<"冒泡排序所用时间："<<duration<<endl;
```

- **冒泡排序**
  冒泡排序算法的运作如下：（从后往前）
  - 比较相邻的元素。如果第一个比第二个大，就交换他们两个。

- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。 其时间复杂度为o(n^2)

```
int* mySystem::bubbleSort(int Nums[],int length){
    int* Answer=Copy(Nums,length);
    clock_t start,end;                                  //开始计时时间和结束计时时间
    start=clock();                                      //记录开始时间
    for(int i=1;i<length;i++){
        for(int j=0;j<length-1;j++){
            if(Answer[i]<Answer[j]){                    //如果这个数比之前那个数小就交换
                swap(Answer[i],Answer[j]);
                Count++;                                //交换一次计数器加一
            }
            else{
                break;
            }
        }
    }
    end=clock();                                        //记录结束时间
    double duration= (double)(end - start) / CLOCKS_PER_SEC;    //计算整个过程时间
    cout<<"冒泡排序所用时间: "<<duration<<endl;
    cout<<"冒泡排序交换次数: "<<Count<<endl;
    Count=0;                                            //计数器清零备下次使用
    return Answer;                                      //返回排序后数组并在主函数中释放内存
}
```

- **简单选择排序**
  选择排序的思想是这样的，新建一个数组answer，第一次从原数组中选出最小的放在answer中，并把原数组中的那个最小的给剔除（赋值为0x7fffffff,即为signed int max)，然后接着从原数组中找到最小的放在answer的第二个位置上，依次最终得到从小到大的排序数组answer.时间复杂度是o(n^2).

```
int* mySystem::simSelSort(int Nums[],int length){
    int* temp=Copy(Nums,length);
    int* answer=(int*)malloc(sizeof(int)*length);
    clock_t start,end;
    start=clock();
    for(int i=0;i<length;i++){
        int min=MAX,index=0;                            //将最小数设为无穷大数
        for(int j=0;j<length;j++){                      //从所有数中寻找最小数保存其下标
            if(temp[j]<min){
                min=temp[j];
                index=j;                                //保存最小数下标
            }
        }
        answer[i]=min;                                  //将最小数赋值给我们排到的位置
        Count++;                                        //计数器加一
        temp[index]=MAX;                                //将原来的那个最小数作废（设为无穷大数）
    }
    end=clock();
    free(temp);
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"选择排序所用时间: "<<duration<<endl;
    cout<<"选择排序交换次数: "<<Count<<endl;
    Count=0;
    return answer;
}
```

- **直接插入排序**
  直接插入排序的思想是这样的：从第2个元素开始，如果它比前一个元素小，那么就把这个元素前移动。怎么移动是个重点，这里需要一个temp，保存我们要往前移动的这个元素，然后依次将temp与它之前的元素比较，如果temp比比较的元素还小那就接着向前比对并把比较的元素后移到后面的元素位中（因为上一个元素已经后移了所以此时不会被覆盖）。当temp比比较的元素大的时候停止，并把temp的值赋给当前位置，重复该过程直到移动到最后一个元素为止。复杂度依旧

是o(n^2)

```
int* mySystem::insertSort(int Nums[],int length){
    int* answer=Copy(Nums,length);
    clock_t start,end;
    start=clock();
    for(int i=1;i<length;i++){
        if(answer[i]<answer[i-1]){                    //如果当前位置数比他之前的数小
            int guard=answer[i];                      //将当前位置的数设置为哨兵
            int j=i-1;                                //从它前面的那个数开始比较
            while(guard<answer[j]&&j){                //如果该数比它前面那个数小就与前面那个数交换
                Count++;                              //计数器加一
                answer[j+1]=answer[j];                //交换两个数
                j--;                                  //接着寻找下一个数
            }
            Count++;                                  //循环里少加了一次计数
            answer[j+1]=guard;                        //将哨兵赋值到排序好的位置
        }
    }
    end=clock();
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"选择排序所用时间："<<duration<<endl;
    cout<<"选择排序交换次数："<<Count<<endl;
    Count=0;
    return answer;
}
```

- **堆排序**

  这个堆排序实际上还是对直接插入排序的优化。我们这样想：在直接插入排序的过程中，我们要进行n-1次的选取数，选出最小的数，但是实际上，有些比较我们已经做过了，之后再接着做这种比较就是一种浪费了。那么有没有什么办法储存这种比较的结果呢？有的。那就是用树来储存这种结构，于是就引出了堆排序的思想，堆排序首先建立一个最大堆，其中最大堆就是对于一颗完全二叉树，其所有的父节点的值都大于等于其子节点，这样的树称为最大堆。然后我们将最大堆中的第一个元素取出来与最后一个元素交换，这样，我们就选出了最大的那个元素放在最后一位，然后我们再次对1到n-1个元素构建最大堆，其实这里的2-n-1个元素都已经满足最大堆的性质，只要将换过来的第一个元素放进这个堆里就行了。依次n-1次操作，排序完毕。

  而构造堆排序需要两个功能：自下而上调整堆ajustDown函数和调用ajustDown函数自倒数第二行树的最后一个节点开始调整生成最大堆。

```
void mySystem::ajustDown(int Nums[],int i,int length){
    int temp=Nums[i];                                          //取其根节点
    for(int largest=2*i+1;largest<length;largest=largest*2+1){ //一直调整到根节点
        if(largest!=length-1&&Nums[largest]<Nums[largest+1]){  //如果其右子树的值比左子树的值大就取右子树
            largest++;                                         //否则取左子树值（即取子树最大的值）
        }
        if(temp<Nums[largest]){                                //将子树最大的值与其父节点的值交换
            Nums[i]=Nums[largest];
            i=largest;
            Count++;                                           //计数器加一
        }
        else{
            break;
        }
    }
    Nums[i]=temp;                                              //将根节点复位
    Count++;                                                   //计数器加一
}
void mySystem::buildMaxHeap(int Nums[],int length){            //调用ajustdown函数构建最大堆
    for(int i=length/2-1;i>=0;i--){                            //从树的倒数第二层的第一个节点开始调整
        ajustDown(Nums,i,length);
    }
}
void mySystem::heapSort(int Nums[],int length){
    clock_t start,end;
    start=clock();
```

```
    buildMaxHeap(Nums,length);                                  //构造最大堆
    for(int i=length-1;i>0;i--){                                //每次移除一个数，然后接着构建最大堆
        swap(Nums[0],Nums[i]);                                  //将根节点放到末尾
        ajustDown(Nums,0,i);                                    //排序前n-1个数
    }
    end=clock();
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"堆排序所用时间："<<duration<<endl;
    cout<<"堆排序交换次数："<<Count<<endl;
    Count=0;
}
```

- **希尔排序**
  希尔排序的思想就是直接插入排序的进化，就是把整个数组分成一小部分一小部分，然后进行插入排序算法，然后不断的合并再进行插入排序，因为这样减小了平均每个元素要往前移动的平均步数，所以比直接插入排序的平均表现要好一些。至于具体算法，则是设置一个gap，一般从gap=length / 2开始，对第一个元素和第1+gap个元素排序，对2和第2+gap个，依次类推，共length / 2个小组进行插入排序，然后将gap>>1,重复上述操作，直到gap==0为止。其实当gap=1的时候就等于直接插入排序了，但是此时整个数组基本上都排好了，所以总步数实际比直接插入排序少，整体算法可以类比直接插入排序的算法结构。

```
int* mySystem::shellSort(int Nums[],int length){
    clock_t start,end;
    start=clock();
    int* answer=Copy(Nums,length);
    for(int gap=(length>>1);gap>0;(gap>>=1)){            //初始步长设为length/2，每次步长乘2
        for(int i=0;i<gap;i++){                          //即将整个数组分为length / 2组排序
            for(int j=i+gap;j<length;j+=gap){
                if(answer[j]<answer[j-gap]){             //如果当前位置数比他之前的数小
                    int temp=answer[j];                  //设置哨兵
                    int pos=j-gap;
                    while(pos>=0&&answer[pos]>temp){     //步骤和插入排序一样，只是此时的步长为gap
                        Count++;
                        answer[pos+gap]=answer[pos];
                        pos-=gap;
                    }
                    Count++;
                    answer[pos+gap]=temp;
                }
                Count++;
            }
        }
    }
    end=clock();
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"希尔排序所用时间："<<duration<<endl;
    cout<<"希尔排序交换次数："<<Count<<endl;
    Count=0;
    return answer;
}
```

- **快速排序**
  c++的algorithm库中提供的qsort函数就是用的快排，快排的思想是递归操作，不断的将排序数组分成更小的数组然后排序。对于每一次排序，先令起始元素为i，结尾元素为j，同时将第一个元素设置为key，作为二分数组的界限。具体操作如下：
  - 先从左边开始向右找，找到第一个比key小的元素，并将其与i所在位置的元素交换，并将j设置为找到该元素的位置的下标。
  - 再从右边开始向左找，找到第一个比key大的元素，并将其与j所在位置的元素交换，并将i设置为找到该元素的位置的下表。
  - 重复上述操作，直到i==j为止，并将key值赋给下标为i（或者j）的位置。
  - 此时数组分为三部分，前后两部分继续递归。结束条件为begin>=end.

```
void mySystem::QuickSort(int answer[],int low,int high){
    if(high-low<=0){                                        //没有需要排序的数就返回
        return;
    }
    int i=low,j=high;                                       //设置指向数组两头的指针
    int key=answer[i];                                      //key值取数组的第一个值
    while(i<j){                                             //当两个指针没有重合时
        while(i<j&&answer[j]>=key){                         //从后面开始寻找比key小的第一个数
            j--;
        }
        Count++;                                            ///////////////计数器加一
        answer[i]=answer[j];
        while(i<j&&answer[i]<=key){                         //从前面开始寻找比key大的第一个数
            i++;
        }
        Count++;                                            //计数器加一
        answer[j]=answer[i];
    }
    answer[i]=key;                                          //将key值复位
    QuickSort(answer,low,i-1);                              //将当前数组分成两组继续递归分组排序
    QuickSort(answer,i+1,high);
}
void mySystem::quickSort(int answer[],int low,int high){
    clock_t start,end;
    start=clock();
    QuickSort(answer,low,high);
    end=clock();
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"快速排序所用时间: "<<duration<<endl;
    cout<<"快速排序交换次数: "<<Count<<endl;
    Count=0;
}
```

- **归并排序**

  基本思想就是先从一小部分开始排序，然后合并两个数组，不断地合并，最后形成一个排好序的数组。这里算法的实现实际上是倒着来的，不断的将数组递归成小的数组，直到只有一个元素的时候，然后再依据合并时的规则不断形成更大的数组，最后形成一个排好序的数组。合并的规则也十分简单，第一个数组的该位置元素小就这边先上，然后index1++，第二个数组index2位置的元素小就第二个数组元素先上，然后index2++，直到index1>=middle||index2>=end，最后将那些没有放完的元素一股脑直接塞入数组中（不需要考虑此时的排序问题，因为剩下的元素已经排好序了，而且肯定比原来末尾的最后的一个元素大。

```
void mySystem::_Merge(int Nums[],int start,int middle,int end){
    if(end-start<=0){
        return;
    }
    int first=start,index=start,second=middle+1;    //first指向第一个数组，second指向第二个数组
    while(first<=middle&&second<=end){               //两个数组的指针都没有指向该数组末尾时
        if(Nums[first]<Nums[second]){                //如果第一个数组的当前数比第二个数组的当前数小
            myTemp[index++]=Nums[first++];           //取第一个数组的当前数，并将第一个数组的指针向后移动一位
        }
        else{
            myTemp[index++]=Nums[second++];          //否则取第二个数组当前数，并将第二个数组的指针后移一位
        }
        Count++;                                     //计数器加一
    }
    while(first<=middle){                            //如果是第一个数组没有放完就把第一个数组的剩下所有数按顺序放进
        myTemp[index++]=Nums[first++];
        Count++;
    }
    while(second<=end){                              //如果是第二个数组没有放完就把第二个数组的剩下所有数按顺序放进
        myTemp[index++]=Nums[second++];
        Count++;
    }
    for(int i=start;i<=end;i++){                     //将暂存数组里的数拷贝回使用的数组里
```

```
        Nums[i]=myTemp[i];
    }
}
void mySystem::MergeSort(int Nums[],int start,int end){
    if(start>=end){
        return;
    }
    int middle=((start+end)>>1);                    //将数组均分为两份进行递归排序
    MergeSort(Nums,start,middle);
    MergeSort(Nums,middle+1,end);
    _Merge(Nums,start,middle,end);                  //将分成的两份排序好的数组归并
}
void mySystem::mergeSort(int Nums[],int start,int end){
    memset(myTemp,0,10000);                         //将临时数组清零
    clock_t begin,terminal;
    begin=clock();
    MergeSort(Nums,start,end);                      //调用归并排序
    terminal=clock();
    double duration= (double)(terminal - begin) / CLOCKS_PER_SEC;
    cout<<"归并排序所用时间："<<duration<<endl;
    cout<<"归并排序交换次数："<<Count<<endl;
    Count=0;
}
```

- **堆排序**
  其实我原来一直以为基数排序就是计数排序，后来才发现基数排序还是非常需要技巧的。。在讲基数排序之前大家需要先知道什么是桶排序，我大概描述一下吧：如果我要排1-5000内产生的1000个数，那么我可以产生1-500，501-1000，...,4501-5000，这样10个桶，然后将这1000个元素按照入桶的规则将这1000个数放入10个桶中，这样我平均每个桶只要排100个元素就可以了，最后按照桶的大小依次输出。但是桶排序的最优解仅存在于数据分布是均匀的，但是实际上这个条件是很难满足的，很多数据都是类似正态分布的，很多元素都入了一个桶。这个时候桶排序差不多就退化成了普通排序。那其实基数排序就是建立在桶排序的思想上，基数排序仅能排列正整数的顺序（负整数其实也能排但其实也是将其化归为正整数排序）。下面是基数排序的具体思路：
  - 先产生10个桶，从0-9。
  - 从个位数开始，按照个位数的大小0-9，按照数组中各个数的出现顺序放入对应10个桶中。 将10个桶中的数按照从0号桶-9号桶
  - 重复n次上述操作，n=数组中最大元素的位数，此时排序完毕。

```
int mySystem::getMaxLen(int Nums[],int length){
    int max=0,maxLen=0;
    for(int i=0;i<length;i++){                      //找到所有数中位数最长的那个
        if(Nums[i]>max){
            max=Nums[i];
        }
    }
    while(max){                                     //使用不断/10计算出其最大位数是maxLen
        max/=10;
        maxLen++;
    }
    return maxLen;                                  //返回该数组中的最长位数
}
int* mySystem::radixSort(int Nums[],int length){
    clock_t start,end;
    int* answer=Copy(Nums,length);
    int log=1,index=0;
    int* Element[10];                               //创建一个10*（length+1）的二维数组数组作为桶
    start=clock();
    int maxLen=getMaxLen(answer,length);
    for(int i=0;i<10;i++){                          //每行的长度都为length+1
        Element[i]=(int*)malloc(sizeof(int)*(1+length));
        Element[i][0]=0;                            //用每个桶的第一位存该桶内存储的数的数目
    }
    for(int q=0;q<maxLen;q++){
        for(int i=0;i<length;i++){
            int temp=++Element[(answer[i]/log)%10][0];  //计算出当前数所属的桶号
```

```
            Element[(answer[i]/log)%10][temp]=answer[i];     //放入桶中
        }
        for(int i=0,index=0;(i<10)&&(index<length);i++){
            for(int j=1;j<=Element[i][0];j++){
                answer[index++]=Element[i][j];          //按照桶的顺序将按照第q+1位大小排序后的数组放回answer中
            }
            Element[i][0]=0;                            //将二维数组的每行首位清零
        }
        log*=10;                                        //继续对下一个最高位进行排序
    }
    end=clock();
    double duration= (double)(end - start) / CLOCKS_PER_SEC;
    cout<<"基数排序所用时间: "<<duration<<endl;
    cout<<"基数排序交换次数: "<<Count<<endl;
    Count=0;
    return answer;
}
```

4.程序运行测试

- **初始欢迎界面**

```
/Users/kirito/CLionProjects/untitled/cmake-build-debug/untitled
**                  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:
```

- **用户输入需要产生的随机数(包括非法输入)**

```
请输入要产生的随机数个数: x
输入的个数不符合标准，请输入1-10000间的数字
请输入要产生的随机数个数: 332423432432423
输入的个数不符合标准，请输入1-10000间的数字
请输入要产生的随机数个数: 10000
请选择排序算法:
```

- **各个排序算法依次调用结果**

```
请输入要产生的随机数个数: 10000
请选择排序算法:1
冒泡排序所用时间: 0.308664
冒泡排序交换次数: 25331252
请选择排序算法:2
选择排序所用时间: 0.207131
选择排序交换次数: 96965
请选择排序算法:3
选择排序所用时间: 0.083972
选择排序交换次数: 25341236
请选择排序算法:4
希尔排序所用时间: 0.003393
希尔排序交换次数: 324735
请选择排序算法:5
```

```
快速排序所用时间: 0.002394
快速排序交换次数: 63392
请选择排序算法:6
堆排序所用时间: 0.002664
堆排序交换次数: 129204
请选择排序算法:7
归并排序所用时间: 0.004288
归并排序交换次数: 133616
请选择排序算法:8
基数排序所用时间: 0.005034
基数排序交换次数: 0
请选择排序算法:9

Process finished with exit code 0
```