目录

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。

2.需求分析

本项目即八大排序算法（冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序，基数排序）的基本实现：

> 随机函数产生10000个随机数，用不同排序算法进行排序，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。

3.设计思路

冒泡排序

最基础的排序算法，算法描述如下：

对需要排序的所有数执行以下操作：

- 从第0个数开始，各位置依次与后一个数进行比较后一个数比
  - 如果前一个数大，则不做操作
  - 如果后一个数大，则交换二者次序
- 完成一次全序列比较后，序列最后的数即此序列最大值（相当于一次冒泡）
- 继续进行下一次冒泡，参与冒泡的序列为上一轮序列去掉最大值
- 重复该过程直至参与冒泡序列长度为1，说明排序完成

选择排序

选择排序又称直接选择排序，是一种简单的排序方法，原理与冒泡排序类似，每次选择待排序序列的最大值/最小值，将其置于数列某一端，重复操作直到数列为升序/降序。与冒泡排序相比，选择排序一次全序列比较只交换一次，在比较过程中只是改变记录位置的标签，并不会交换值，知道全序列比较完后，再把标签记录的最值交换到序列一端。算法描述如下（升序排列）：

- 从当前待排序序列中选出最大值，与序列末端数交换
- 待排序序列去掉末端数
- 对新的待排序序列继续进行上述操作，直到待排序序列长度为1

直接插入排序

算法的基本思路为：按照大小，每一步将一个待排序序列中的数插入到已排序序列中，直到所有数全部完成排序，算法描述如下：

- 将待排序序列第一个数视为有序序列，将剩下的数视为无序序列，依次执行以下操作：
- 将无序序列第一个数从后向前依次与有序序列比较：
  - 大于：直接将该数从无序序列划为有序序列
  - 小于：用临时变量保存无序序列第一个数，有序序列后移一位，继续比较直至找到有序序列中大于该数的数，小于部分依次后移
- 将临时变量存储的数插入到有序序列中空出来的位置
- 有序序列增加一个数，无序序列减少第一个数
- 重复进行上述操作直至无序序列全部插入完成

希尔排序

希尔排序是插入排序的一种，又称缩小增量排序，与直接插入排序相比更为高效，原理相当于分组插入排序，将待排序数按一定间隔分为几个组，每组分别进行插入排序，然后逐步缩小间隔至1，完成排序。算法描述如下：

- 以待排序数组大小n的$\frac{1}{2}$作为初始间隔，对每组数组内进行插入排序
- 取新的间隔为原间隔的$\frac{1}{2}$，再对各组数组内进行插入排序
- 重复上述操作直至间隔为0

快速排序

快速排序是对冒泡排序的一种改进，基本思路为选取一个标志数， 通过一遍排序将要排序的数据分成两个部分，其中一个部分全部大于标志数，另一个部分全部小于标志数，在对两个部分进行同样操作，直至所有数排序完毕。算法描述如下（升序排列）：

- 选取待排序数组的第一个数作为标志数
- 对当前序列进行以下操作：
    - 从头开始遍历数组，将比标志数大的数移至高端
    - 从尾开始遍历数组，将比标志数小的数移至低端
- 当两端的遍历至同一数时，停止遍历，说明此时该数组已经被标志数分为了两个部分
- 分别对两个部分进行上述操作，直至进行操作的数组长度为1，即全部排序完成

堆排序

堆排序利用堆的特点进行排序。所谓堆，是一种完全二叉树，树的子节点均小于/大于其父节点，又称为大根堆/小根堆，即堆中数据的最大值/最小值一定在堆的根节点。利用堆的这一性质进行选择排序，就可以得到有序序列。算法描述如下（升序排列）：

- 建立最大堆
- 将堆顶元素抽出作为有序数组的最左端元素
- 调整堆为新的最大堆
- 重复上述操作直至堆中元素为零，排序完成

归并排序

归并排序是分治法的一个经典应用。其核心思想是将有序的子序列归并成为有序的序列，从而达到全序列排序的目的。归并排序首先要将全序列分成有序的子序列，算法描述如下（升序排列）：

- 通过递归将排序数列不断二分，直至无法再分
- 按照二分的逆序进行两两归并：
    - 为两个待归并序列设置两个头指针
    - 比较指针所指数大小，将较小的数放入新序列
    - 拥有较小数的数组和新序列数组指针后移一位，重复上一步比较过程
    - 当其中一个序列全部遍历完后，直接将另一个序列剩余部分复制到新序列，完成归并

基数排序

基数排序是分配式排序的一种，通过键值的某一资讯，将要排序的元素按类进行分配，再按类的顺序有序排在一起，从而达到排序的目的。用基数排序对数字进行排序的算法描述如下：

- 通从所有数的最低位/最高位开始，按照0-9的类别将所有数分类后输出
- 再+1/-1位按照同样分类法分类输出，直至最大数/最小数的最高位/最低位都进行过分类操作
- 该序列排序完毕

前者称为LSD(最低位优先法)，后者称为MSD（最高位优先法）

以LSD为例，假设原来有一串数值如下所示：

73, 22, 93, 43, 55, 14, 28, 65, 39, 81

首先根据个位数的数值，在走访数值时将它们分配至编号0到9的桶子中：

0

1 81

2 22

3 73 93 43

4 14

5 55 65

6

7

8 28

9 39

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

81, 22, 73, 93, 43, 14, 55, 65, 28, 39

接着再进行一次分配，这次是根据十位数来分配：

0

1 14

2 22 28

3 39

4 43

5 55

6 65

7 73

8 81

9 93

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

14, 22, 28, 39, 43, 55, 65, 73, 81, 93

排序完毕

八大算法比较

方法	平均情况	最好情况	最坏情况	空间辅助存储	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	稳定
基数排序[1]	$O(d(r+n))$	$O(d(rd+n))$	$O(d(r+n))$	$O(rd+n)$	稳定

4.核心代码说明

时间与交换次数的计算

时间

利用ctime库中的`clock()`函数分别记录下排序算法启动与结束的时间，二者差值即为耗时

交换次数

整型变量count初始化为0，在执行交换操作语句后对count进行自增操作，实现计数

冒泡排序

```
void BubbleSort(vector<int> &data){
  int count = 0;
  double time = 0,start,finish;
  int n = data.size();

  start = clock();
  //用循环逐个比较大小
  for (int i = 0; i < n-1; i++) {
    for (int j = 0; j < n -i- 1; j++) {
      if (data[j] > data[j + 1]) {  //将较大的元素向数组右侧移动
        swap(data[j], data[j + 1]);
        count++;
      }
    }

  }
```

```
  finish = clock();
  //得出时间
  time = finish - start;
  cout<<"冒泡排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"冒泡排序交换次数:  "<<count<<endl<<endl;
}
```

利用两层循环实现冒泡排序，外层循环用于实现对每一个位置的数都执行向后比较操作，内层循环用于控制每一次向后比较操作的比较次数，即`n-i-1`次，不包含后面i+1个已经排好序的数

选择排序

```
void SelectionSort(vector<int> &data){
  int key,count = 0,n = data.size();
  double time = 0,start,finish;

  start = clock();
  //用循环遍历数组，用key记录当前最小值位置
  for (int i = 0; i < n - 1; i++) {
    key = i;
    for (int j = i+1; j < n; j++) {
      if (data[key] > data[j]) key = j;//比较大小，更新key值
    }
    //将key记录的最小值交换到数组最左端
    if (key != i) {
      int temp = data[i];
      data[i] = data[key];
      data[key] = temp;
      count++;
    }
  }
  finish = clock();
  //得出时间
  time = finish - start;
  cout<<"选择排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"选择排序交换次数:  "<<count<<endl<<endl;
}
```

利用key变量来记录最小值位置，其余代码说明见注释

直接插入排序

```
void InsertionSort(vector<int> &data){
  int count = 0,n = data.size();
  double time,start,finish;

  start = clock();
  for (int i = 1; i < n; i++) {
    if (data[i] < data[i - 1]) {//若第i个元素小于第i-1个时，有序表后移一位
      int temp = data[i];
      int j = i - 1;
      data[i] = data[j];
```

```
      for (j; j >= 0; j--) {  //比较查找在有序表中插入的位置
        if (temp < data[j]) {
          data[j + 1] = data[j];
          count++;
        }
        else break;
      }
      data[j + 1] = temp;
    }
  }
  finish = clock();
  time = finish - start;
  cout<<"直接插入排序所用时间:    "<<time<<"ms"<<endl;
  cout<<"直接插入排序交换次数:    "<<count<<endl<<endl;
}
```

希尔排序

```
void ShellSort(vector<int> &data){
  int count = 0,gap,n = data.size();
  double time = 0,start,finish;

  start = clock();
  for (gap = n / 2; gap > 0; gap /= 2)
    for (int i = gap; i < n; i++)//从第gap的元素开始
      if (data[i] < data[i - gap]) {//按gap分组，分别对每组元素进行插入排序
        int temp = data[i];
        int j = i - gap;
        while (j >= 0 && data[j] > temp) {
          data[j + gap] = data[j];
          count++;
          j -= gap;
        }
        data[j + gap] = temp;
      }
  finish = clock();
  time =finish - start ;
  cout<<"希尔排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"希尔排序交换次数:  "<<count<<endl<<endl;
}
```

从第gap个元素开始进行分组比较操作可以减少`gap-1`次外层循环

快速排序

因为要实现计时与次数统计，同时要使用递归函数，所以用两个函数实现，`QuickSort`为快速排序的计时、计数函数，`Qsort`为快速排序的递归函数

```cpp
void QuickSort(vector<int> &data){
  int count = 0;
  double time = 0,start,finish;

  start = clock();
  Qsort(data, 0, data.size() - 1,count);
  finish = clock();
  time = finish - start;
  cout<<"快速排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"快速排序交换次数:  "<<count<<endl<<endl;
}
```

```cpp
//快速排序递归调用函数
void Qsort(vector<int> &array, int low, int high, int& count) {
  if (low >= high) return;
  int first = low, last = high;
  int key = array[first];//选取数组的第一个元素作为基准元素

  while (first < last) {
    while (first < last && array[last]>=key)
      --last;
    array[first] = array[last];//将第一个比key小的值移动到最低端
    while (first < last && array[first] < key)
      ++first;
    array[last] = array[first];//将第一个比key大的值移动到高端
    count += 2;
  }
  //将第一个比key大的位置补为key，相当于三次赋值完成了两次两两交换
  array[first] = key;
  Qsort(array, low, first - 1, count);
  Qsort(array, first + 1, high, count);
}
```

堆排序

堆排序使用了两个函数，`HeapSort`为主要函数，实现计时、计数以及取根节点排序功能，`MaxHeapify`功能为堆化，用于构建堆和每次抽取根节点后调整堆

```cpp
void HeapSort(vector<int> &data){
  int count = 0,n = data.size();
  double time,start,finish;

  start = clock();
  for (int i = n - 1; i >= 0; i--)
    MaxHeapify(data,n,i,count); //从子树开始向上整理整棵树
  //通过循环将堆中最大元素逐个放到数组最后
  while (n > 0) {
    swap(data[n - 1], data[0]);

    n--;
```

```
    MaxHeapify(data,n,0,count);
  }
  finish = clock();
  time = start - finish;
  cout<<"堆排序所用时间:        "<<time<<"ms"<<endl;
  cout<<"堆排序交换次数:        "<<count<<endl<<endl;
}
```

```
//堆化，将数组调整为最大堆
void MaxHeapify(vector<int> &array,int size,int element,int &count) {
  int l_child = element * 2 + 1, r_child = element * 2 + 2;
  //当子树均在范围内时，循环整理交换部分的子树，把元素放在正确位置
  while (r_child < size) {
    if (array[element] >= array[l_child] && array[element] >= array[r_child]) return;
    if (array[l_child] >= array[r_child]) {
      swap(array[element], array[l_child]);
      count++;
      element = l_child;
    }
    else {
      swap(array[element], array[r_child]);
      count++;
      element = r_child;
    }
    l_child = element * 2 + 1, r_child = element * 2 + 2;
  }
  //左子树还在范围内，整理左子树
  if (l_child < size&&array[l_child]>array[element]) {
    swap(array[element], array[l_child]);
    count++;
  }
  //到叶子节点，整理完毕
  return;
}
```

归并排序

归并排序包含三个函数，`MergeSort`实现计时、计数功能，`Msort`为递归函数，用递归方式实现二分及归并操作，`Merge`为归并的具体实现函数

```cpp
void MergeSort(vector<int> &data){
  int count = 0,n = data.size();
  double start,finish,time;
  vector<int> output(n);

  start = clock();
  Msort(data, output, 0, n - 1,count);
  data = output;
  finish = clock();
  time = finish - start;
  cout<<"归并排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"归并排序交换次数:  "<<count<<endl<<endl;
}
```

```cpp
//通过递归将待排序数列不断二分，进行排序
void Msort(vector<int> &data, vector<int> &output, int s, int t,int &count) {
  vector<int> temp(data.size());
  if (s == t) output[s] = data[s];
  else {
    int m = (s + t) / 2;
    Msort(data, temp, s, m,count);
    Msort(data, temp, m + 1, t,count);
    Merge(temp, output, s, m , t,count);
  }
}

```

```cpp
//将数组r1[i,m]与r2[m+1,n]合并为r3[i,n]
void Merge(vector<int> &data,vector<int> &output, int i, int m, int n,int &count) {
  int j, k;
  //依次取值比较，较小的放入r3
  for (j = m+1, k = i; i <= m&&j <= n; k++) {
    if (data[j] < data[i])
      output[k] = data[j++];
    else
      output[k] = data[i++];
    count++;
  }
  //处理比较完后的剩余数
  while (i <= m)
    output[k++] = data[i++];
  while (j <= n)
    output[k++] = data[j++];
}
```

基数排序

基数排序包含两个函数，`RadixSort`实现计时、计数、基数排序的分类、输出功能，`MaxBit`求出待排序数中最大位数，用于确定分类操作的次数

```
void RadixSort(vector<int> &data){
  int count = 0,n = data.size(),max = MaxBit(data),radix = 1,i,j,k;
  double start,finish,time;
  vector<int> temp(n),counts(10);

  start = clock();
  for (i = 1; i <= max; i++) {//根据最大位数决定排序次数
    for (j = 0; j < 10; j++)//将计数器置零
      counts[j] = 0;
    for (j = 0; j < n; j++) {//统计各个数字的数据个数
      k = (data[j] / radix) % 10;
      counts[k]++;
    }
    for (j = 1; j < 10; j++)//确定各个数字在排序完后数组中具体位置
      counts[j] += counts[j - 1];
    for (j = n - 1; j >= 0; j--) {//循环将数据读入temp
      k = (data[j] / radix) % 10;
      temp[counts[k] - 1] = data[j];
      count++;
      counts[k]--;
    }
    data = temp;
    radix *= 10;
  }
  finish = clock();
  time = finish - start;
  cout<<"基数排序所用时间:  "<<time<<"ms"<<endl;
  cout<<"基数排序交换次数:  "<<count<<endl<<endl;
}
```

```
//求出数据中的最大位数
int MaxBit(vector<int> &data) {
  int max = 1, flag = 10;
  for (int i = 0; i < data.size(); i++) {
    while (data[i] > flag) {
      flag *= 10;
      max++;
    }
  }
  return max;
}
```

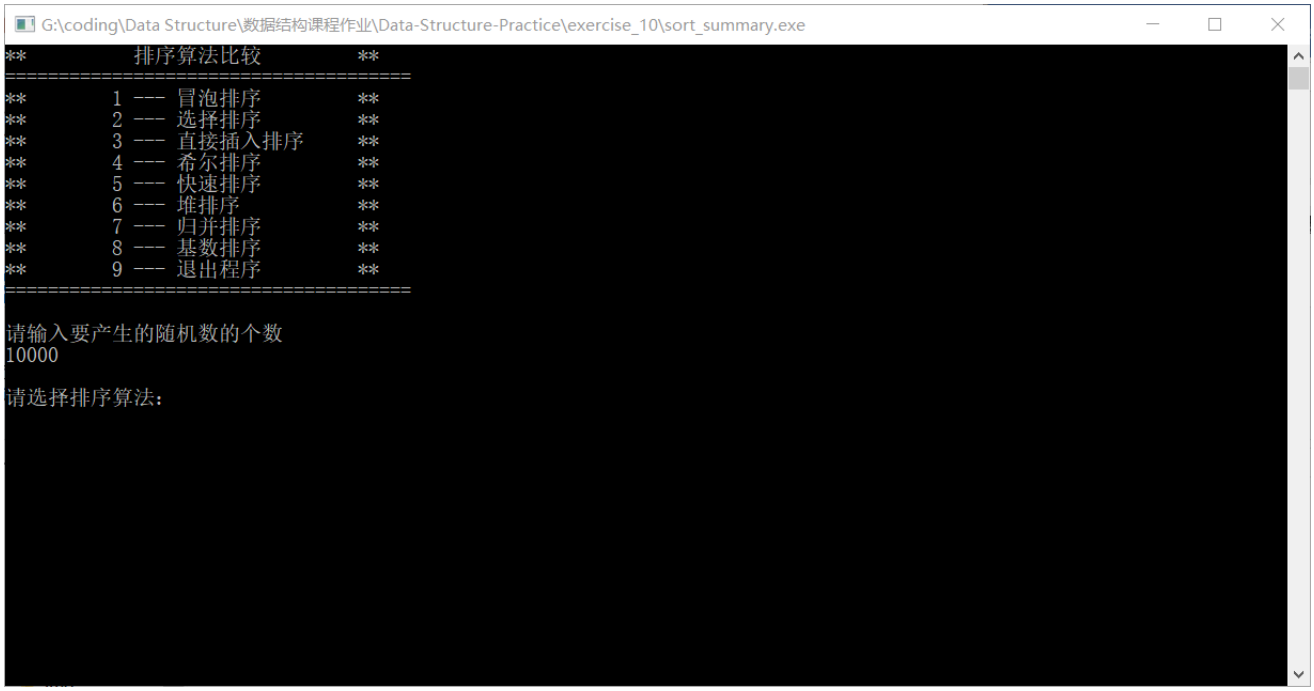
函数接口说明

返回值类型	成员函数名	参数	功能
void	showArray	(vector &array)	输出序列
void	BubbleSort	(vector &data)	冒泡排序
void	SelectionSort	(vector &data)	选择排序
void	InsertionSort	(vector &data)	直接插入排序
void	ShellSort	(vector &data)	希尔排序
void	Qsort	(vector &array, int low, int high, int& count)	快速排序递归函数
void	QuickSort	(vector &data)	快速排序主函数
void	MaxHeapify	(vector &array,int size,int element,int &count)	堆排序堆化
void	HeapSort	(vector &data)	堆排序主函数
void	Merge	(vector &data,vector &output, int i, int m, int n,int &count)	归并排序归并操作
void	Msort	(vector &data, vector &output, int s, int t,int &count)	递归实现二分及合并
void	MergeSort	(vector &data)	归并排序主函数
void	MaxBit	(vector &data)	基数排序求出最大数位数
void	RadixSort	(vector &data)	基数排序主函数

5.使用说明及函数功能演示

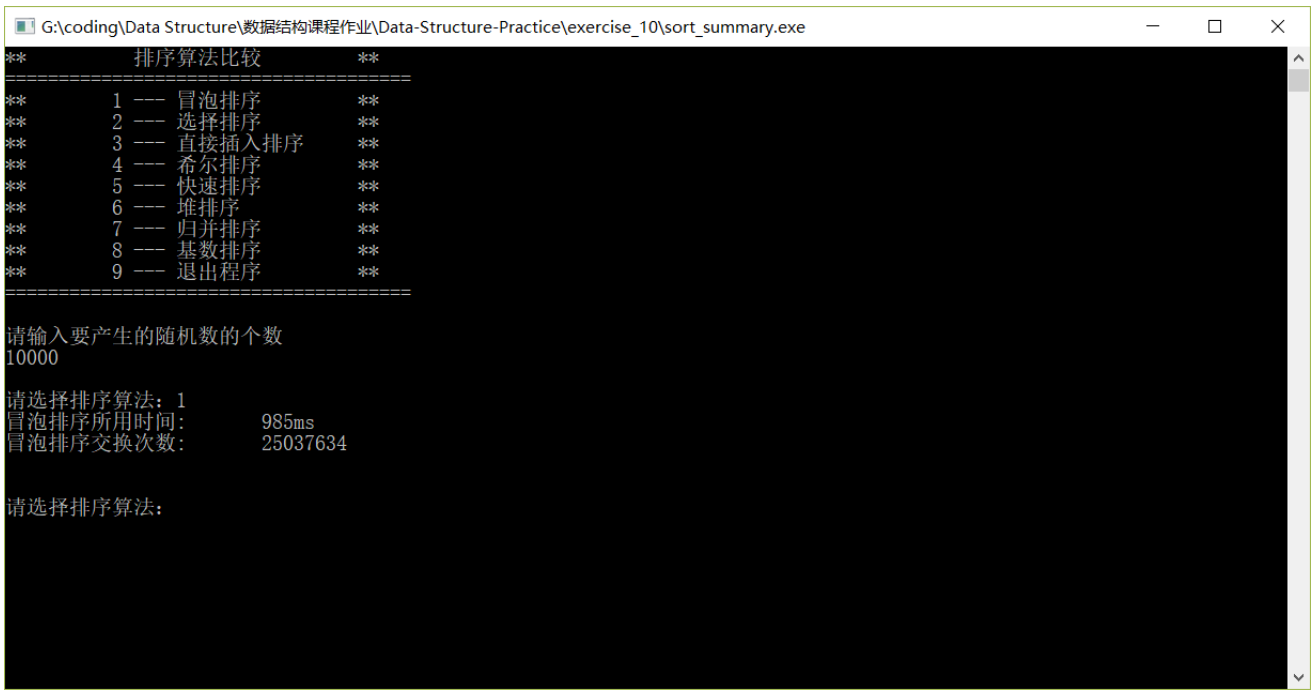
运行`sort_summary.exe`启动程序，按照系统提示进行操作

首先输入要产生的随机数个数



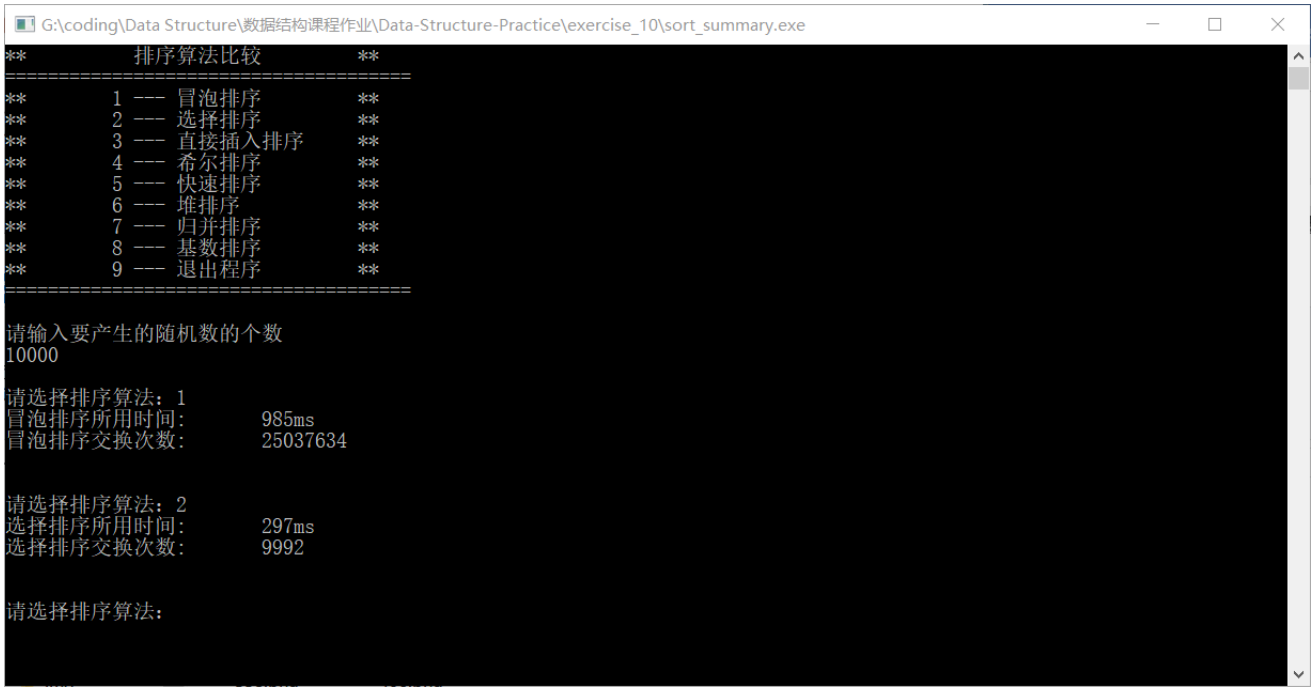
冒泡排序

输入1选择该功能，随后系统将会输出冒泡排序时间与交换次数



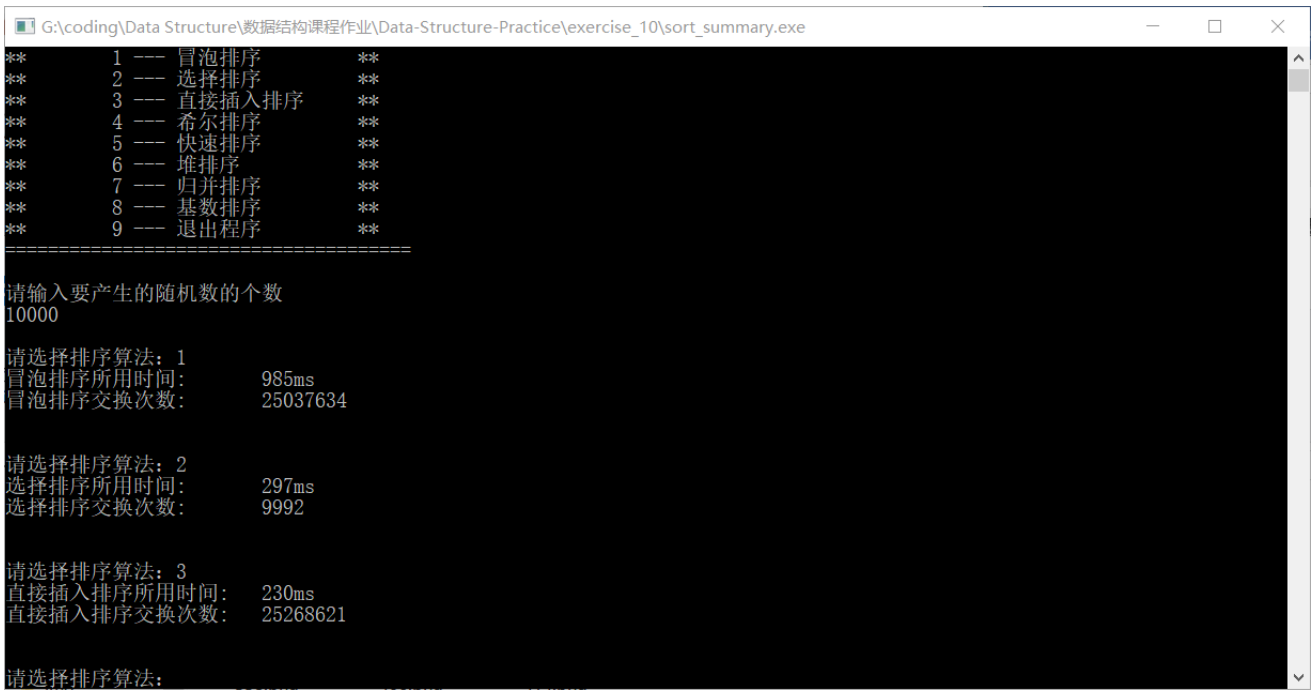
选择排序

输入2选择该功能，随后系统将会输出选择排序时间与交换次数



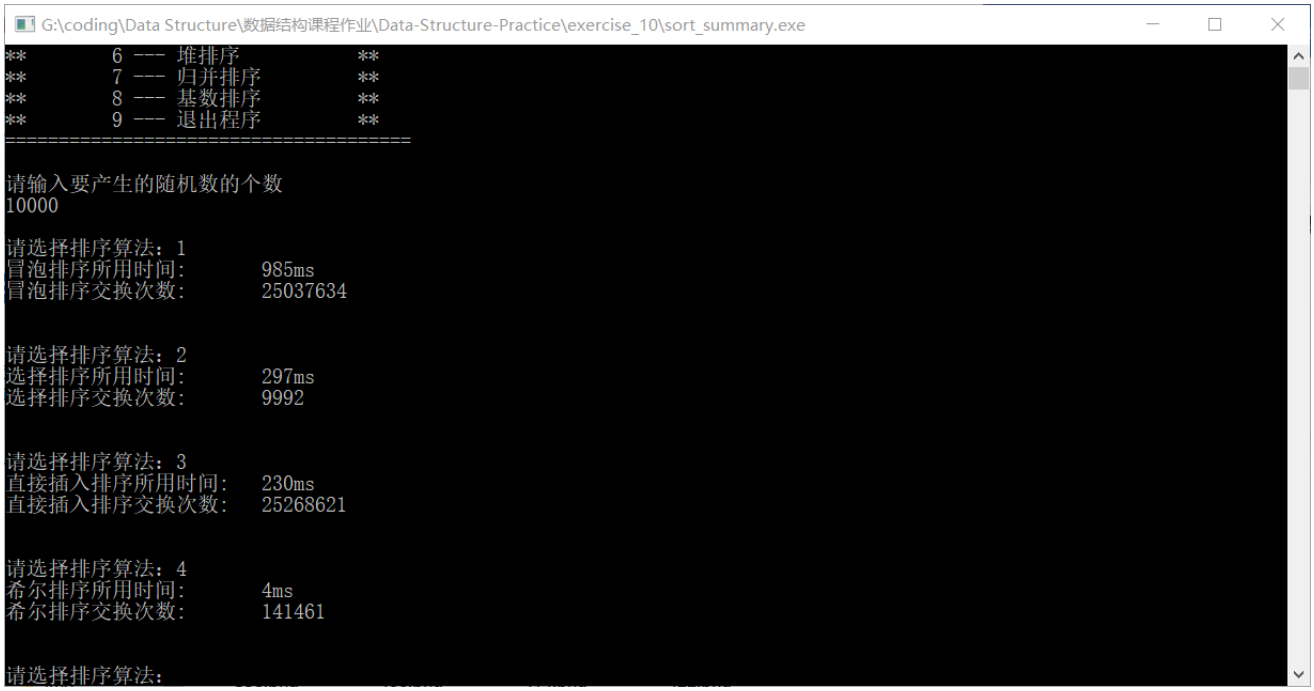
直接插入排序

输入3选择该功能，随后系统将会输出直接插入排序时间与交换次数



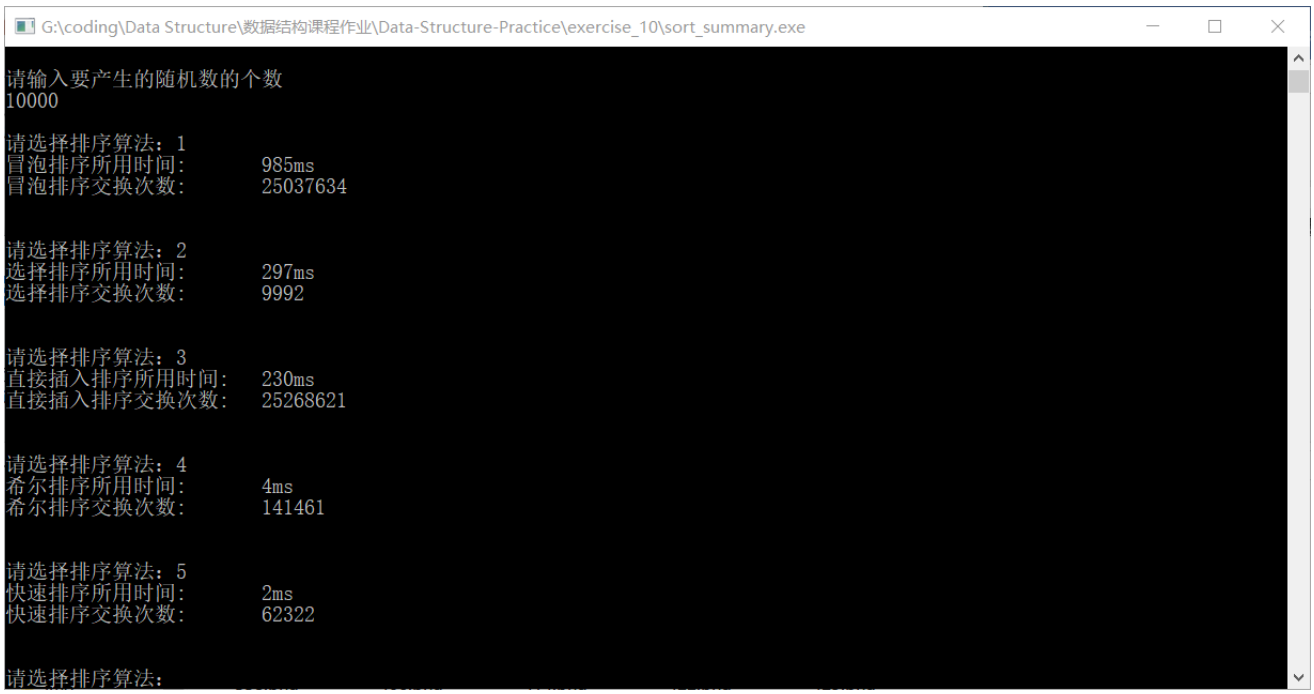
希尔排序

输入4选择该功能，随后系统将会输出希尔排序时间与交换次数



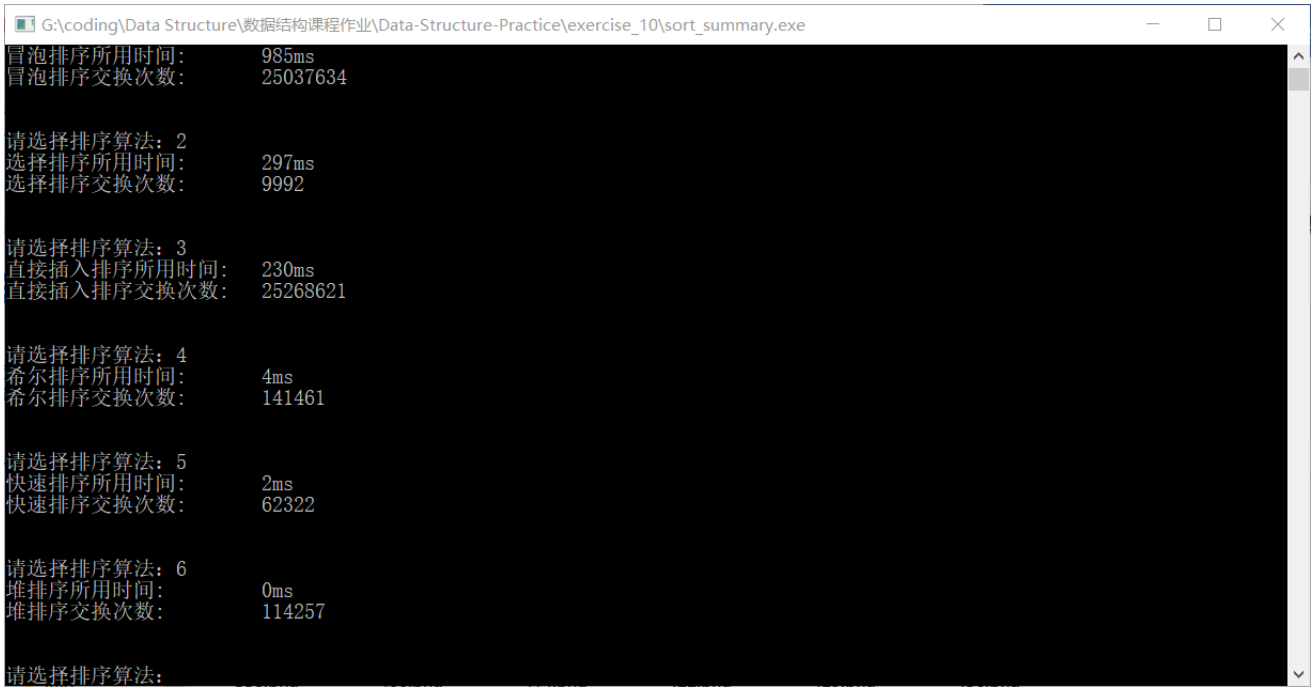
快速排序

输入5选择该功能，随后系统将会输出快速排序时间与交换次数



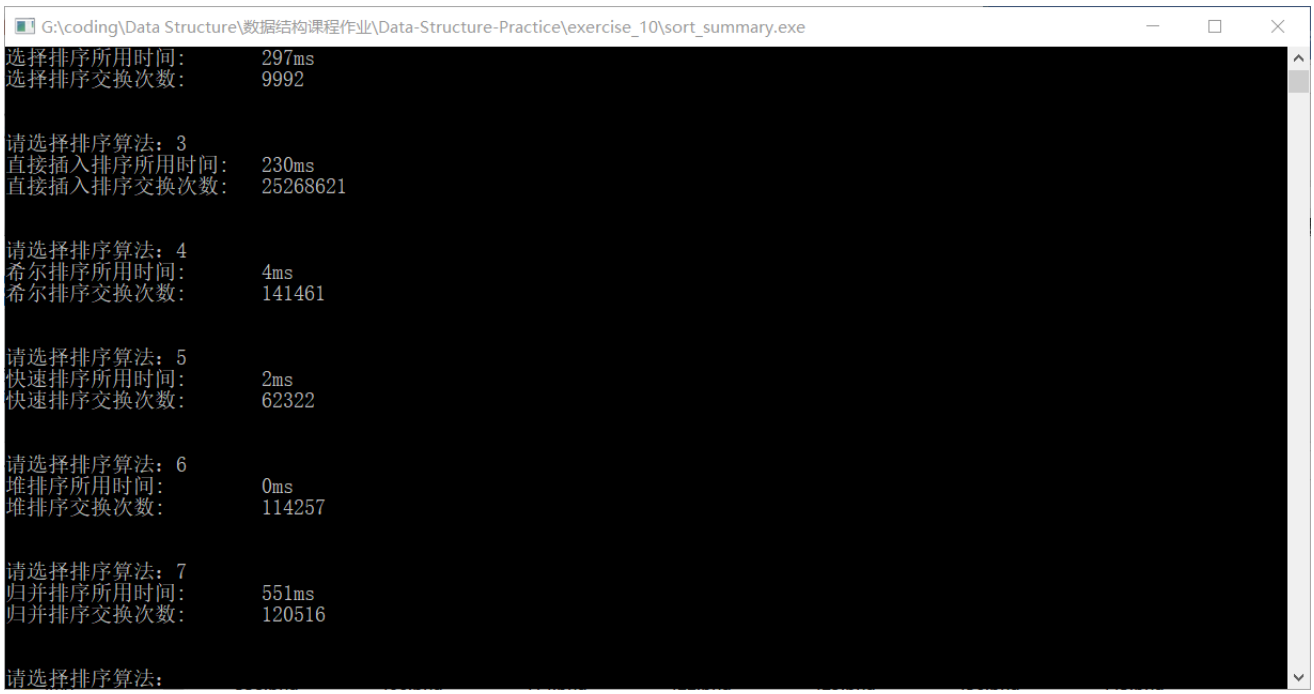
堆排序

输入6选择该功能，随后系统将会输出堆排序时间与交换次数



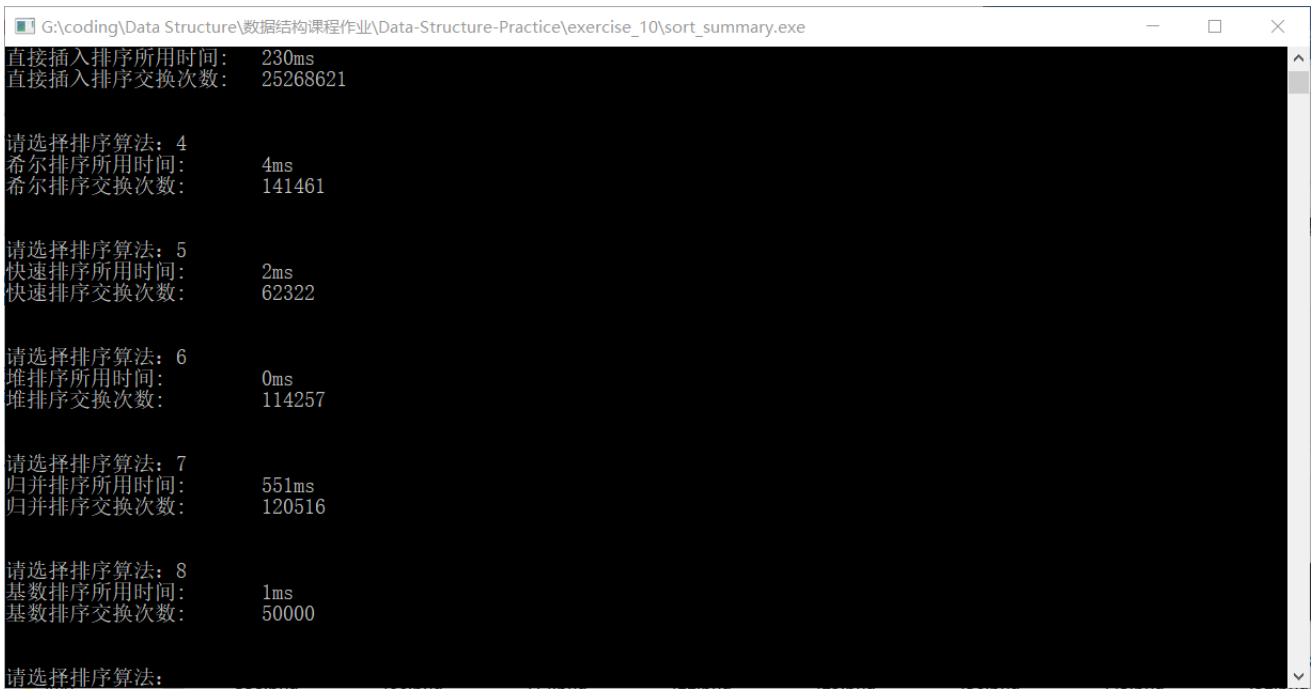
归并排序

输入7选择该功能，随后系统将会输出归并排序时间与交换次数



基数排序

输入8选择该功能，随后系统将会输出基数排序时间与交换次数



退出程序

输入9退出程序，需要再次运行则重启exe文件

运行环境说明

CPU:i7-6700HQ

MEM:16G DDR4 2133Hz

Compiler:TDM-GCC 4.9.2 32bit-Debug

每执行一种排序算法，都将重新生成输入个数的随机数，所以耗时和交换次数不一定完全契合时间复杂度分析结果

---

1. r为关键字基数（一般选择10），d为最大数位数，n为关键字的个数↩