项目说明文档

数据结构课程设计

——8 种排序算法的比较案例

作 者 姓 名：安江涛

学　　号：1952560

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 分析

1.1 项目简介

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。

2 设计

2.1 算法设计

实现 8 中排序方法，分别是冒泡排序、插入排序、选择排序、希尔排序、快速排序、堆排序、归并排序、基数排序。

2.2 生成随机数

```
Vector<int> get_random(int size) {
    std::default_random_engine e;
    std::uniform_int_distribution<int> u(-1e9-7, 1e9+7);
    e.seed(time(0));
    Vector<int> v;
    for (int i = 0; i < size; i++)
        v.push_back(u(e));
    return v;
}
```

2.3 测试函数

```
#include"../H/Sort.h"
#include<vector>
#include<limits.h>
#include<iomanip>

bool checkCin(){
    if(std::cin.fail()){
        std::cin.clear();
        std::cin.ignore(INT_MAX,'\n');
        std::cout<<"input error!please re-enter!"<<'\n';
        return false;
    }
    return true;
}

bool checkSort(const Vector<int> &v){
    for(int i=0;i<v.size()-1;i++){
        if(v[i]>v[i+1])
            return false;
    }
    std::cout<<"the vector has sorted!"<<'\n';
    return true;
}

void instructions(){
    std::string str[12];
    str[0]="**        sorting algorithm comparison     **";
    str[1]="============================================";
    str[2]="**         1---bubblesort                  **";
    str[3]="**         2---selectionsort               **";
    str[4]="**         3---insertionsort               **";
    str[5]="**         4---shellsort                  **";
    str[6]="**         5---quicksort                   **";
    str[7]="**         6---heapsort                    **";
    str[8]="**         7---mergesort                     **";
    str[9]="**         8---radixsort                  **";
    str[10]="**          9---quiting the system          **";
    str[11]="============================================";
    for(int i=0;i<12;i++)
        std::cout<<str[i]<<'\n';
}

int main() {
    instructions();
```

```
std::cout<<"please enter the number of random numbers to be generated:";
int random_num=0;
std::cin>>random_num;
while(!checkCin()){
    std::cin>>random_num;
}
Vector<int> v = get_random(random_num);
int order=0;
while(true){
    Vector<int> v_copy=v;
    clock_t start,end;
    double cost;
    for(int i=0;i<v_copy.size();i++){
        if(v_copy[i]>v_copy[i+1]){
            std::cout<<'\n'<<"the vector has not sorted!"<<'\n';
            break;
        }
    }
    std::cout<<std::setw(15)<<"please select the sorting algorithm:";
    std::cin>>order;
    while(!checkCin()||order<1||order>9){
        std::cin>>order;
    }
    switch (order)
    {
    case 1:
        start=clock();
        bubbleSort(v_copy);
        end=clock();
        cost=end-start;
        //checkSort(v_copy);
        std::cout<<std::setw(15)<<"bubblesort takes time:"<<cost<<" ms"<<'\n'<<'\n';
        break;
    case 2:
        start=clock();
        selectionSort(v_copy);
        end=clock();
        cost=end-start;
        //checkSort(v_copy);
        std::cout<<std::setw(15)<<"selectionsort takes time:"<<cost<<" ms"<<'\n'<<'\n';
        break;
```

```
case 3:
    start=clock();
    insertionSort(v_copy);
    end=clock();
    cost=end-start;
    //checkSort(v_copy);
    std::cout<<std::setw(15)<<"insertionsort takes time:"<<cost<<" ms"<<'\n'<<'\n';
    break;
case 4:
    start=clock();
    shellSort(v_copy);
    end=clock();
    cost=end-start;
    //checkSort(v_copy);

    std::cout<<std::setw(15)<<"shellsort takes time:"<<cost<<" ms"<<'\n'<<'\n';



    break;
case 5:
    start=clock();
    quickSort(v_copy,0,v_copy.size()-1);
    end=clock();
    cost=end-start;
    checkSort(v_copy);
    //std::cout<<std::setw(15)<<"quicksort takes time:"<<cost<<" ms"<<'\n'<<'\n';
    break;
case 6:
    start=clock();
    heapSort(v_copy);
    end=clock();
    cost=end-start;
    //checkSort(v_copy);
    std::cout<<std::setw(15)<<"heapsort takes time:"<<cost<<" ms"<<'\n'<<'\n';
    break;
```

```
            case 7:
                start=clock();
                mergeSort(v_copy,0,v_copy.size()-1);
                end=clock();
                cost=end-start;
                //checkSort(v_copy);
                std::cout<<std::setw(15)<<"mergesort takes time:"<<cost<<" ms"<<'\n'<<'\n';
                break;
            case 8:
                start=clock();
                radixSort(v_copy);
                end=clock();
                cost=end-start;
               // checkSort(v_copy);
                std::cout<<std::setw(15)<<"radixsort takes time:"<<cost<<" ms"<<'\n'<<'\n';
                break;
            case 9:
                exit(0);
            default:
                break;
            }
        }
    }
```

3 实现

3.1 冒泡排序

冒泡排序（Bubble Sort）也是一种简单直观的排序算法。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。这个算法的名字由来是因为越小的元素会经由交换慢慢"浮"到数列的顶端。

作为最简单的排序算法之一，冒泡排序给我的感觉就像 Abandon 在单词书里出现的感觉一样，每次都在第一页第一位，所以最熟悉。冒泡排序还有一种优化算法，就是立一个 flag，当在一趟序列遍历中元素没有发生交换，则证明该序列已经有序。但这种改进对于提升性能来说并没有什么太大作用。

3.1.1 算法步骤

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2. 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。这步做完后，最后的元素会是最大的数。

3. 针对所有的元素重复以上的步骤，除了最后一个。

4. 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。

3.1.2 核心代码

```
template<typename T>
void bubbleSort(Vector<T>& v) {
    if (v.size() <= 1)
        return;
    for (int i = 0; i < v.size() - 1; i++) {
        for (int j = 0; j < v.size() - i - 1; j++) {
            if (v[j] > v[j + 1]) {
                swap<T>(v[j], v[j + 1]);
                times++;
            }
        }
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.1.3 复杂度分析

最坏时间复杂度----O(n^2)

最优时间复杂度---O(n)

平均时间复杂度---O(n^2)

空间复杂度---O(1)

稳定性---稳定

3.2 选择排序

选择排序是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度。所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。

3.2.1 算法步骤

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。

3.2.2 核心代码

```
template<typename T>
void selectionSort(Vector<T>& v) {
    if (v.size() <= 1)
        return;
    for (int i = 0; i < v.size() - 1; i++) {
        int min_index = i;
        for (int j = i + 1; j < v.size(); j++) {
            if (v[j] < v[min_index])
                min_index = j;
        }
        if (min_index != i){
            swap<T>(v[i], v[min_index]);
            times++;
        }
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.2.3 复杂度分析

最坏时间复杂度----O(n^2)
最优时间复杂度---O(n)
平均时间复杂度---O(n^2)
空间复杂度---O(1)
稳定性---不稳定

3.3 插入排序

插入排序的代码实现虽然没有冒泡排序和选择排序那么简单粗暴，但它的原理应该是最容易理解的了，因为只要打过扑克牌的人都应该能够秒懂。插入排序是一种最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。

插入排序和冒泡排序一样，也有一种优化算法，叫做拆半插入。

3.3.1 算法步骤

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。）

3.3.2 核心代码

```
template<typename T>
void insertionSort(Vector<T>& v) {
    if (v.size() <= 1)
        return;
    for (int i = 1; i < v.size(); i++) {
        int key = v[i];
        int j = i - 1;
        while (j >= 0 && key < v[j]) {
            v[j + 1] = v[j];
            j--;
            times++;
        }
        times++;
        v[j + 1] = key;
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.3.3 复杂度分析

最坏时间复杂度----O(n^2)

最优时间复杂度---O(n)

平均时间复杂度---O(n^2)

空间复杂度---O(1)

稳定性---稳定

3.4 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。但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；

希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录"基本有序"时，再对全体记录进行依次直接插入排序。

3.4.1 算法步骤

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。

3.4.2 核心代码

```
template<typename T>
void shellSort(Vector<T>& v) {
    for (int gap = v.size() >> 1; gap > 0; gap >>= 1) {
        for (int i = gap; i < v.size(); i++) {
            int j = i;
            while (j - gap >= 0 && v[j] < v[j - gap]) {
                times++;
                swap<T>(v[j], v[j - gap]);
                j -= gap;
            }
        }
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.4.3 复杂度分析

最坏时间复杂度----O(nlog2n)

最优时间复杂度---O(n)
平均时间复杂度---O(nlog2n)
空间复杂度---O(1)
稳定性---不稳定

3.5 快速排序

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要 O(nlogn) 次比较。在最坏状况下则需要 O(n2) 次比较，但这种状况并不常见。事实上，快速排序通常明显比其他 O(n logn) 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。快速排序使用分治法（Divide and conquer）策略来把一个串行（list）分为两个子串行（sub-lists）。快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。

快速排序的名字起的是简单粗暴，因为一听到这个名字你就知道它存在的意义，就是快，而且效率高！它是处理大数据最快的排序算法之一了。虽然 Worst Case 的时间复杂度达到了 O(n^2)，但是人家就是优秀，在大多数情况下都比平均时间复杂度为 O(n logn) 的排序算法表现要更好，可是这是为什么呢，在《算法艺术与信息学竞赛》中的答案：

快速排序的最坏运行情况是 O(n^2)，比如说顺序数列的快排。但它的平摊期望时间是 O(nlogn)，且 O(nlogn) 记号中隐含的常数因子很小，比复杂度稳定等于 O(nlogn) 的归并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于归并排序

3.5.1 算法步骤

1. 从数列中挑出一个元素，称为 "基准"（pivot）;

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（recursive）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；

3.5.2 核心代码

```
template<typename T>
void quickSort(Vector<T>& v, int start, int end) {
    if (start >= end)
        return;
    int i = start, j = end, base = v[i];
    while (i < j) {
        while (v[j] >= base && i < j)
            j--;
        v[i] = v[j];
        times++;
        while (v[i] <= base && i < j)
            i++;
        v[j] = v[i];
        times++;
    }
    v[i] = base;
    times++;
    quickSort(v, start, i - 1);
    quickSort(v, i + 1, end);
    if(start==0&&end==v.size()-1){
        std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
        times=0;
    }
}
```

3.5.3 复杂度分析

最坏时间复杂度----O(n^2)

最优时间复杂度---O(nlogn)

平均时间复杂度---O(nlogn)
空间复杂度---O(nlogn)
稳定性---不稳定

3.6 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。堆排序可以说是一种利用堆的概念来排序的选择排序。分为两种方法：

1. 大顶堆：每个节点的值都大于或等于其子节点的值，在堆排序算法中用于升序排列；

2. 小顶堆：每个节点的值都小于或等于其子节点的值，在堆排序算法中用于降序排列；

3.6.1 算法步骤

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 fitdown(0)，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。

3.6.2 核心代码

```
template<typename T>
void fitdown(int ind, Vector<T>& v, int end) {
    if ((ind << 1) + 1 >= end)
        return;
    T val1 = v[(ind << 1) + 1];
    times++;
    if ((ind << 1) + 2 >= end) {
        if (v[ind] < val1) {
            v[(ind << 1) + 1] = v[ind];
            v[ind] = val1;
        }
    }
    else {
        T val2 = v[(ind << 1) + 2];
        if ((val2 < val1 || val2 == val1) && v[ind] < val1) {
            v[(ind << 1) + 1] = v[ind];
            v[ind] = val1;
            fitdown((ind << 1) + 1, v, end);
        }
        else if (val1 < val2 && v[ind] < val2) {
            v[(ind << 1) + 2] = v[ind];
            v[ind] = val2;
            fitdown((ind << 1) + 2, v, end);
        }
    }
}

template<typename T>
void heapSort(Vector<T>& v) {
    for (int i = (v.size() >> 1) - 1; i >= 0; i--) {
        fitdown(i, v, v.size());
    }
    for (int i = v.size() - 1; i > 0; i--) {
        swap<T>(v[0], v[i]);
        times++;
        fitdown(0, v, i);
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.6.3 复杂度分析

最坏时间复杂度----O(nlogn)
最优时间复杂度---O(nlogn)
平均时间复杂度---O(nlogn)
空间复杂度---O(1)
稳定性---不稳定

3.7 归并排序

归并排序（Merge sort）是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

作为一种典型的分而治之思想的算法应用，归并排序的实现由两种方法：

自上而下的递归（所有递归的方法都可以用迭代重写，所以就有了第 2 种方法）；

自下而上的迭代；

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 O(nlogn) 的时间复杂度。代价是需要额外的内存空间。

3.7.1 算法步骤

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。

3.7.2 核心代码

```
template<typename T>
void mergeSort(Vector<T>& v, int start, int end) {
    if (start >= end)
        return;
    int len = end - start, mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    mergeSort(v, start1, end1);
    mergeSort(v, start2, end2);
    Vector<T> reg;
    while (start1 <= end1 && start2 <= end2){
        times++;
        reg.push_back(v[start1] < v[start2] ? v[start1++] : v[start2++]);
    }
    while (start1 <= end1){
        times++;
        reg.push_back(v[start1++]);
    }
    while (start2 <= end2){
        times++;
        reg.push_back(v[start2++]);
    }
    for(int i=0;i<reg.size();i++)
    v[start + i] = reg[i];
    if(start==0&&end==v.size()-1){
        std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
        times=0;
    }
}
```

3.7.3 复杂度分析

最坏时间复杂度----O(nlogn)

最优时间复杂度---O(nlogn)

平均时间复杂度---O(nlogn)

空间复杂度---O(n)

稳定性---稳定

3.8 基数排序

基数排序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、计数排序、桶排序这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

基数排序：根据键值的每位数字来分配桶；

计数排序：每个桶只存储单一键值；

桶排序：每个桶存储一定范围的数值；

3.8.1 算法步骤

1.取得数组中的最大数，并取得位数；
2.对数位较短的数前面补零；
3.分配，先从个位开始，根据位值(0-9)分别放到 0~9 号桶中;
（如果想要试下你负数的排序，需要 20 个桶）
4.收集，再将放置在 0~9 号桶中的数据按顺序放到数组中;
5.重复 3~4 过程，直到最高位，即可完成排序。

3.8.2 核心代码

```
void radixSort(Vector<int> &v){
    int len = v.size();
    if(len <= 1) return;
    int res = 10, div = 1;
    bool is_not_max_digit = true;
    Vector<int> counter[20];
    while(is_not_max_digit){
        is_not_max_digit = false;
        for(int i = 0; i < len; i++){
            if(abs(v[i]) / res) is_not_max_digit = true;
            int index = v[i] % res / div+10;
            if(v[i]<0)
                index=9-abs(v[i])%res/div;
            counter[index].push_back(v[i]);
            times++;
        }
        int radix = 0, index = 0;
        for(int i = 0; i < len;){
            times++;
            for(;index<counter[radix].size();index++){
                v[i+index]=counter[radix][index];
            }
            i+=index;
            index=0;
            radix++;
        }
        for(int i = 0; i < 20; i++){
            counter[i].clear();
        }
        res *= 10; div *= 10;
    }
    std::cout<<"The number of comparison for selection sorting is:"<<times<<'\n';
    times=0;
}
```

3.8.3 复杂度分析

最坏时间复杂度----O(n*d)

最优时间复杂度---O(n*d)
平均时间复杂度---O(n*d)
空间复杂度---O(d)
稳定性---稳定

4 输出测试

4.1 随机数数组的大小为 100

```
**         sorting algorithm comparison     **
===============================================
**          1---bubblesort                   **
**          2---selectionsort                **
**          3---insertionsort                **
**          4---shellsort                    **
**          5---quicksort                    **
**          6---heapsort                     **
**          7---mergesort                    **
**          8---radixsort                    **
**          9---quiting the system           **
===============================================
please enter the number of random numbers to be generated:100
please select the sorting algorithm:1
The number of comparison for selection sorting is:2630
bubblesort takes time:1 ms

please select the sorting algorithm:2
The number of comparison for selection sorting is:96
selectionsort takes time:1 ms

please select the sorting algorithm:3
The number of comparison for selection sorting is:2729
insertionsort takes time:1 ms

please select the sorting algorithm:4
The number of comparison for selection sorting is:432
shellsort takes time:2 ms

please select the sorting algorithm:5
The number of comparison for selection sorting is:400
quicksort takes time:1 ms

please select the sorting algorithm:6
The number of comparison for selection sorting is:614
heapsort takes time:0 ms

please select the sorting algorithm:7
The number of comparison for selection sorting is:672
mergesort takes time:1 ms

please select the sorting algorithm:8
The number of comparison for selection sorting is:1080
radixsort takes time:3 ms

please select the sorting algorithm:9
```

4.2 随机数数组的大小为 1000

```
please enter the number of random numbers to be generated:1000
please select the sorting algorithm:1
The number of comparison for selection sorting is:245762
bubblesort takes time:13 ms

please select the sorting algorithm:2
The number of comparison for selection sorting is:992
selectionsort takes time:21 ms

please select the sorting algorithm:3
The number of comparison for selection sorting is:246761
insertionsort takes time:16 ms

please select the sorting algorithm:4
The number of comparison for selection sorting is:8994
shellsort takes time:1 ms

please select the sorting algorithm:5
The number of comparison for selection sorting is:5408
quicksort takes time:0 ms

please select the sorting algorithm:6
The number of comparison for selection sorting is:9454
heapsort takes time:3 ms

please select the sorting algorithm:7
The number of comparison for selection sorting is:9976
mergesort takes time:23 ms

please select the sorting algorithm:8
The number of comparison for selection sorting is:9180
radixsort takes time:1 ms

please select the sorting algorithm:9

D:\Course-design-of-data-structure>
```

4.3 随机数数组的大小为 10000

```
please enter the number of random numbers to be generated:10000
please select the sorting algorithm:1
The number of comparison for selection sorting is:24911132
bubblesort takes time:764 ms

please select the sorting algorithm:2
The number of comparison for selection sorting is:9990
selectionsort takes time:356 ms

please select the sorting algorithm:3
The number of comparison for selection sorting is:24921131
insertionsort takes time:224 ms

please select the sorting algorithm:4
The number of comparison for selection sorting is:146250
shellsort takes time:6 ms

please select the sorting algorithm:5
The number of comparison for selection sorting is:70137
quicksort takes time:14 ms

please select the sorting algorithm:6
The number of comparison for selection sorting is:127667
heapsort takes time:21 ms

please select the sorting algorithm:7
The number of comparison for selection sorting is:133616
mergesort takes time:13 ms

please select the sorting algorithm:8
The number of comparison for selection sorting is:90180
radixsort takes time:12 ms

please select the sorting algorithm:9

D:\Course-design-of-data-structure>
```

4.4 随机数数组的大小为 100000

```
please enter the number of random numbers to be generated:100000
please select the sorting algorithm:1
The number of comparison for selection sorting is:2507211152
bubblesort takes time:74516 ms

please select the sorting algorithm:2
The number of comparison for selection sorting is:99988
selectionsort takes time:34583 ms

please select the sorting algorithm:3
The number of comparison for selection sorting is:2507311151
insertionsort takes time:21844 ms

please select the sorting algorithm:4
The number of comparison for selection sorting is:3105080
shellsort takes time:77 ms

please select the sorting algorithm:5
The number of comparison for selection sorting is:853031
quicksort takes time:26 ms

please select the sorting algorithm:6
The number of comparison for selection sorting is:1609507
heapsort takes time:46 ms

please select the sorting algorithm:7
The number of comparison for selection sorting is:1668928
mergesort takes time:111 ms

please select the sorting algorithm:8
The number of comparison for selection sorting is:900180
radixsort takes time:48 ms

please select the sorting algorithm:9

D:\Course-design-of-data-structure>
```