排序算法设计与实现

20123211 高雨晴

0. 概述

概要说明本实验的内容、主要方法或技术、以及实验结果

- 实验内容：实现四种排序算法，并分别用10~100000的随机数据测试其运行时间（每增长十倍测试一次）。
- 实验方法：使用C++语言，通过MinGW-w64 C/C++编译器
- 实验结果：在大多数情况下，快速排序 > 归并排序 > 插入排序 > 冒泡排序

1. 实验设计

1.1 算法实现原理

1.1.1 冒泡排序

遍历数组，将最大的数不断向后交换，使第一轮遍历中最大数在数组的最后一位。之后再对前n-1个数进行这个过程，以此类推。

1.1.2 插入排序

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。

1.1.3 归并排序

采用分治法，先将数组拆成小部分，每个部分分别排好序后，再按照大小顺序逐渐合成原来的数组。

1.1.4 快速排序

通过一次排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小，再按这种方法对这两部分数据分别进行快速排序。

1.2 算法设计与分析

在算法设计与分析里面详细解释算法的细节，核心代码或伪代码，并对算法的时间复杂度进行分析

1.2.1 冒泡排序

```
// 冒泡排序
void BubbleSort(int array[], int n){
    for (int i=0; i<n-1; i++) { // 共遍历 n-1 次
        for (int j=0; j<n-i-1; j++) {  // 第i次遍历 n-i-1 个元素
            if (array[j]>array[j+1])
                swap(array[j],array[j+1]);
        }
    }
}
```

最坏情况：T(n) = O(n^2)

平均情况：T(n) = O(n^2)

稳定度：稳定

1.2.2 插入排序

```
// 插入排序
void InsertionSort(int a[], int n){
    for (int j=1; j<n; j++) {
        int key = a[j]; // 待排序组的第一个元素
        int i = j-1; // j-1是有序组的最后一个元素的下标
        while (i>=0 && a[i]>key) {
            a[i+1] = a[i];
            i--;// 不是合适位置，则有序数组向后移动
        }
        a[i+1] = key;// 找到合适位置，插入
    }
}
```

最坏情况：T(n) = O(n^2)

平均情况：T(n) = O(n^2)

稳定度：稳定

1.2.3 归并排序

```
//
void merge(int arr[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = 0; // i 为第一组起点， j 为第二组起点，k指向temp当前放到哪个位置
    int* temp = new int[high - low + 1];// 申请动态内存
    while (i <= mid && j <= high)
    {// 将两个序列循环比较，从小到大填入temp
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
        temp[k++] = arr[j++];
    }
    while (j <= high) // 如果比完后第二组还有剩，全都填入temp
        temp[k++] = arr[j++];
    while (i <= mid)// 如果比完后第一组还有剩，全都填入temp
        temp[k++] = arr[i++];
    for (i = low, k = 0;i <= high;i++, k++) // 将排好序后的temp填回原数组
        arr[i] = temp[k];
    delete[]temp; // 释放new分配的对象数组指针指向的内存
}
// 归并排序
void MergeSort(int arr[], int low, int high)
{
    // 递归
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort(arr, low, mid);
```

```
        MergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}
```

平均情况：T(n) = O(nlogn)

最坏情况：T(n) = O(nlogn)

稳定性：稳定

1.2.4 快速排序

```
// 快速排序
void QuickSort(int a[], int start, int end) {
    // 若数组长度小于等于1，返回
    if(start>=end)
    return;
    // 初始化：设置数组的第一个元素为pivot，i指向第二个元素，j指向最后一个元素
    int i = start+1 , j = end;
    int pivot = a[start];
    // j一直减小直到指向比pivot小的元素，i一直增大直到指向比pivot大的元素，都停下后交换所指元
素，直到i和j相遇
    while( i <= j ) {
        while( i<=j && a[j] >= pivot ){
            j--;
        }
        while( i<=j && a[i] <= pivot ){
            i++;
        }
        if(i<=j){
            swap(a[i],a[j]);
            i++;j--;
        }
    }
    // 将pivot和ij相遇处的元素交换，将数组一分为二，再进行快速排序
    a[start] = a[j];
    a[j] = pivot;
    QuickSort(a,start,j);
    QuickSort(a,j+1,end);
}
```

平均情况：T(n) = O(nlogn)

最坏情况：T(n) = O(n^2)

稳定性：不稳定

2. 实验结果说明与分析

2.1 实验设置

```
int main() {
    int cases[6] = {10,100,1000,10000,100000};
    for (int c=0; c<5; c++) {
        int size = cases[c];
        int *array = genRandomArray(size); // 随机生成数组，因为是伪随机所以能够保证每次测试生成的数组是一样的
        int array1[size];

        cout << "Array size:" << size << endl;

        // 计时器设置，可以精确到微秒
        _LARGE_INTEGER time_start;  //开始时间
        _LARGE_INTEGER time_over;   //结束时间
        double dqFreq;      //计时器频率
        LARGE_INTEGER f;    //计时器频率
        QueryPerformanceFrequency(&f);
        dqFreq=(double)f.QuadPart;


        // time of bubble sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        BubbleSort(array1, size);
        QueryPerformanceCounter(&time_over);

        cout << "Bubble sort:" << (double)1000*(time_over.QuadPart-time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of insertion sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        InsertionSort(array1, size);
        QueryPerformanceCounter(&time_over);
        cout << "Insertion sort:" << (double)1000*(time_over.QuadPart-time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of merge sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        MergeSort(array1, 0, size-1);
        QueryPerformanceCounter(&time_over);
        cout << "Merge sort:" << (double)1000*(time_over.QuadPart-time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of quick sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        QuickSort(array1, 0, size-1);
        QueryPerformanceCounter(&time_over);
        cout << "Quick sort:" << (double)1000*(time_over.QuadPart-time_start.QuadPart)/dqFreq << "ms" << endl;

        cout << "\n" << endl;
```

```
    }
    system("pause");
    return 0;
}
```

2.2 实验结果以及相应的分析

一般用图、表等方式展示结果，并对出现的结果进行分析

数组大小	冒泡排序	插入排序	归并排序	快速排序
10	0.0005	0.0003	0.0063	0.0006
100	0.0211	0.0071	0.0635	0.0086
1000	1.7971	0.6207	0.668	0.1006
10000	299.967	57.6972	6.4148	1.1038
100000	36191.6	5128.25	70.5749	13.0043

单位：ms.

数组大小 n 越小，越适合用冒泡排序和插入排序；数组大小 n 越大，越适合用归并排序和快速排序。

3. 总结

冒泡排序和插入排序的平均时间复杂度为 O(n^2)，归并排序和快速排序的平均时间复杂度为 O(nlogn)，从实验结果来看，符合理论规律。

4. 附录

```
# include <iostream>
# include <ctime>
# include <cstdlib>
# include <bits/stdc++.h>
# include <windows.h>

using namespace std;

static const int MAX = 1000000;

// generate random array
int *genRandomArray(int size) {
    static int array[MAX];
    srand((int)time(0)); // ?
    for (int i=0; i<size; i++)
        array[i] = rand() % size;
    return array;
}
// swap fuction
void swap(int p, int q) {
    int temp;
```

```
    temp = p;
    p = q;
    q = temp;
}
// bubble sort
void BubbleSort(int array[], int n){
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<n-i-1; j++) {
            if (array[j]>array[j+1])
                swap(array[j],array[j+1]);
        }
    }
}
// insertion sort
void InsertionSort(int a[], int n){
    for (int j=1; j<n; j++) {
        int key = a[j];
        int i = j-1;
        while (i>=0 && a[i]>key) {
            a[i+1] = a[i];
            i--;
        }
        a[i+1] = key;
    }
}
// merge sort
void merge(int arr[], int low, int mid, int high)
{
    int i = low, j = mid + 1, k = 0;
    int* temp = new int[high - low + 1];
    while (i <= mid && j <= high)
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
        temp[k++] = arr[j++];
    }
    while (j <= high)
        temp[k++] = arr[j++];
    while (i <= mid)
        temp[k++] = arr[i++];
    for (i = low, k = 0;i <= high;i++, k++)
        arr[i] = temp[k];
    delete[]temp;
}

void MergeSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort(arr, low, mid);
        MergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
```

```
}
// quick sort
void QuickSort(int a[], int start, int end) {
    if(start>=end)
    return;
    int i = start+1 , j = end;
    int pivot = a[start];
    while( i <= j ) {
        while( i<=j && a[j] >= pivot ){
            j--;
        }
        while( i<=j && a[i] <= pivot ){
            i++;
        }
        if(i<=j){
            swap(a[i],a[j]);
            i++;j--;
        }
    }
    a[start] = a[j];
    a[j] = pivot;
    QuickSort(a,start,j);
    QuickSort(a,j+1,end);
}


int main() {
    int cases[6] = {10,100,1000,10000,100000};
    for (int c=0; c<5; c++) {
        int size = cases[c];
        int *array = genRandomArray(size);
        int array1[size];

        cout << "Array size:" << size << endl;

        _LARGE_INTEGER time_start;  //开始时间
        _LARGE_INTEGER time_over;   //结束时间
        double dqFreq;      //计时器频率
        LARGE_INTEGER f;    //计时器频率
        QueryPerformanceFrequency(&f);
        dqFreq=(double)f.QuadPart;


        // time of bubble sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        BubbleSort(array1, size);
        QueryPerformanceCounter(&time_over);

        cout << "Bubble sort:" << (double)1000*(time_over.QuadPart-
time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of insertion sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
```

```
        InsertionSort(array1, size);
        QueryPerformanceCounter(&time_over);
        cout << "Insertion sort:" << (double)1000*(time_over.QuadPart-
time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of merge sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        MergeSort(array1, 0, size-1);
        QueryPerformanceCounter(&time_over);
        cout << "Merge sort:" << (double)1000*(time_over.QuadPart-
time_start.QuadPart)/dqFreq << "ms" << endl;

        // time of quick sort
        memcpy(array1, array, size * sizeof(int));
        QueryPerformanceCounter(&time_start);
        QuickSort(array1, 0, size-1);
        QueryPerformanceCounter(&time_over);
        cout << "Quick sort:" << (double)1000*(time_over.QuadPart-
time_start.QuadPart)/dqFreq << "ms" << endl;

        cout << "\n" << endl;
    }
    system("pause");
    return 0;
}
```