关于多种排序方式的算法效率 实践报告

Jammy Zeta

目录

一. 摘要

排序在算法中一直起着举足轻重的作用，而如今排序的方式也层出不穷，这些排序算法思想各异，各有千秋。因此本篇报告将对常见的几种排序算法：冒泡排序（BubbleSort）、选择排序（SelectSort）、插入排序（InsertSort）、快速排序（QuickSort）、C 语言库函数中的快速排序（qSort）、归并排序（MergeSort）、计数排序（CountSort） 从排序**无序数组**与**几乎有序数组**两个角度进行时间效率上的对比分析。

二. 问题分析

本报告旨在对各个排序算法的时间效率进行分析，因此需要对每个排序算法的排序过程计时，以**排序过程时长**作为衡量时间效率的标准。为了**避免偶然性**同时**放大**算法之间的**差异**，我们构造了大小（元素个数）分别为 1000，10000，100000，1000000 的数组。其次为了判断这些排序算法**应对不同数组**的能力，我们将数组分为**无序**和**几乎有序**数组。同时为了避免其他因素的影响，我们需要**控制变量**，即必须做到在每种情况下每个排序算法操作的均是**同一**数组。那么就需要将各个排序算法**函数化**，这样即可在不同情况下**方便地调用**，而无需频繁修改代码。

三. 算法*说明

1. 本算法主要包含**构造数组、调用排序函数（将数组复原+排序计时+验证排序结果正确性）**以及**各个排序函数**三个部分
2. 本算法中的无序数组由 rand()函数随机产生，几乎有序数组是由无序数组产生后先经过**计数排序（在无序数组的排序计时中计数排序所花时间最少）**变为有序数组而后随机选取 50 对数组元素进行互换来模拟的
3. 本算法所采用的计时函数 clock()位于<time.h>，单位为毫秒（ms）
4. 由于需要控制变量（同一数组），因此本算法中有**两个主要数组**，其中一个为**原数组**，另一个数组用于**排序**，即在**每次**排序函数调用前需要将其**复原**到原数组
5. 本算法中每个排序函数均由本人编写，因此为防止排序函数中存在纰漏，特此在每次排序结束后添入了评判部分，若排序结果存在问题则会报错，具体代码如下：

```
for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf(" （） Sort ERROR!\n");break;}
    if(i==n)
    printf(" （） Sort time-consuming:%d ms\n",end-start);
```

6. 本算法源代码中的/*or*/表示**代码的切换处**，用于切换**无序数组**或**几乎有序数组**的排序

7. 在排序**较大**数组时会发生**栈溢出**的情况，本算法采取调用 malloc()**和** free() 函数的方式通过**向计算机借用储存空间**来解决该问题：

```
int *s=malloc(n*sizeof(int));
  ……
free (s);
```

*算法完整源代码详见**附录**

四. 测试结果展示

Ⅰ. 无序数组

1.1 数组元素个数：1000

```
The number of elements in the array:1000
BubbleSort time-consuming:5 ms
SelectSort time-consuming:5 ms
InsertSort time-consuming:3 ms
QuickSort time-consuming:0 ms
# qsort time-consuming:0 ms
MergeSort time-consuming:0 ms
CountSort time-consuming:0 ms
```

1.2 数组元素个数：10000

```
The number of elements in the array:10000
BubbleSort time-consuming:432 ms
SelectSort time-consuming:442 ms
InsertSort time-consuming:136 ms
QuickSort time-consuming:1 ms
# qsort time-consuming:1 ms
MergeSort time-consuming:2 ms
CountSort time-consuming:0 ms
```

1.3 数组元素个数：100000

```
The number of elements in the array:100000
BubbleSort time-consuming:40806 ms
SelectSort time-consuming:33536 ms
InsertSort time-consuming:11045 ms
QuickSort time-consuming:17 ms
# qsort time-consuming:21 ms
MergeSort time-consuming:41 ms
CountSort time-consuming:3 ms
```

1.4 数组元素个数：1000000

```
The number of elements in the array:1000000
QuickSort time-consuming:345 ms
# qsort time-consuming:145 ms
MergeSort time-consuming:247 ms
CountSort time-consuming:11 ms
```

注：此时冒泡排序、选择排序、插入排序排序时间均超过 10 min.

Ⅱ.几乎有序数组

2.1 数组元素个数：1000

```
The number of elements in the nearly orderly array:1000
BubbleSort time-consuming:5 ms
SelectSort time-consuming:2 ms
InsertSort time-consuming:1 ms
QuickSort time-consuming:0 ms
# qsort time-consuming:0 ms
MergeSort time-consuming:0 ms
CountSort time-consuming:0 ms
```

2.2 数组元素个数：10000

```
The number of elements in the nearly orderly array:10000
BubbleSort time-consuming:406 ms
SelectSort time-consuming:215 ms
InsertSort time-consuming:2 ms
QuickSort time-consuming:12 ms
# qsort time-consuming:0 ms
MergeSort time-consuming:1 ms
CountSort time-consuming:0 ms
```

2.3 数组元素个数：100000

```
The number of elements in the nearly orderly array:100000
BubbleSort time-consuming:22927 ms
SelectSort time-consuming:12096 ms
InsertSort time-consuming:4 ms
QuickSort time-consuming:10 ms
# qsort time-consuming:3 ms
MergeSort time-consuming:11 ms
CountSort time-consuming:1 ms
```

2.4 数组元素个数：1000000

```
The number of elements in the nearly orderly array:1000000
QuickSort time-consuming:270 ms
# qsort time-consuming:27 ms
MergeSort time-consuming:119 ms
CountSort time-consuming:9 ms
```

注：此时冒泡排序、选择排序、插入排序排序时间均超过 10 min.

五．测试结果分析

先对上述结果进行整理绘表：

关于不同排序方式排序不同情况数组所需的时间统计表(单位:ms)

排序方式 \ 数组种类	1000		10000		100000		1000000	
	有序	几乎有序	有序	几乎有序	有序	几乎有序	有序	几乎有序
Bubble[O(n^2)]	5	5	432	406	40806	22927	-	-
Select[O(n^2)]	5	2	442	215	33536	12096	-	-
Insert[O(n^2)]	3	1	136	2	11045	4	-	8
Quick[O($n\log_2 n$)]	0	0	1	12	17	10	345	270
#qsort[O($n\log_2 n$)]	0	0	1	0	21	3	145	27
Merge[O($n\log_2 n$)]	0	0	2	1	41	11	247	119
Count[O(n)]	0	0	0	0	3	1	11	9

注：“-”表示该种情况下排序时间超过 10 min.

显然在上述所有算法中时间效率最高的算法非**计数排序（CountSort）**莫属，其实从理论上分析确实也是计数排序的时间复杂度最低，仅为 O(n)。

另外冒泡、选择排序很明显在数组非常庞大时不论是无序数组还是几乎有序数组效率均很低，从理论上来讲它们的时间复杂度确实也是最高的。

插入排序虽然复杂度与冒泡、选择排序的平均时间复杂度一致，而且在数组无序时效率确实也很低，然而当数组几乎有序时效率却显得异常之高，事实上这是因为插入排序的最优时间复杂度仅为 O(n)。

其次快速排序*效率虽高，但其稳定性不高。至于自己编写的快速排序与 C 语言库函数中的 qSort 相比有略微差别，总体相差不大，但自己编写的快速排序函数必然还是存在可优化之处的。

归并排序可见是一种效率很高而且稳定性也不错的排序方式，但对空间要求比较高。

计数排序的思想确实精妙，而且避免了非常繁琐的比较，从时间角度上来说计数排序确实是一种不二之选。然而其缺点也是显然的，首先排序数组中不能存在负数及小数，另外当数组内元素的数值很大时会造成极大的空间浪费，甚至会发生栈溢出的状况。

因此每种排序算法各有千秋，**本报告的分析也并不能说明某一种排序算法是完全优于其他排序算法的。**

我们理应**按需所取。**

*事实上快速排序一开始在排序几乎有序数组(100000)时发生了栈溢出，然而库函数中的 qSort 函数却不会发生栈溢出，这不禁引起了我的思考。后来经过上网查询以及对自己排序算法的分析后可以发现我的算法取的都是第一位数组元素作为基准数从而导致其在排序几乎有序数组时的递归次数将非常大，最后发生栈溢出。得知其本质原因后，我对自己的排序算法进行了优化，即每次随机取数组的一位元素作为基准数，这样就大大提高了快速排序算法的稳定性。

附录（源代码）

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void swap(int *x, int *y);
void BubbleSort(int arr[],int n);
void SelectSort(int arr[],int n);
void InsertSort(int arr[],int n);
void QuickSort(int arr[],int head,int foot);
void MergeSort(int arr[],int temparr[],int first,int last);
void merge(int arr[],int temparr[],int first,int mid,int last)
;
void CountSort(int arr[],int n);
int cmp(const void*a,const void *b)
{return *(int *)a-*(int *)b;}
int main() {
    srand(time(0));
    const int n=1000;

    // int *s=malloc(n*sizeof(int));
    // int *cup=malloc(n*sizeof(int));
    /*or*/
    int s[n],cup[n],i;

    for(int i=0;i<n;++i)
    s[i]=cup[i]=rand();
    /*or*/
    // for(int i=0;i<n;++i)
    // s[i]=rand();
    // CountSort(s,n);
    // for(int i=0;i<50;++i)
    // swap(&s[rand()%n],&s[rand()%n]);
    // for(int i=0;i<n;++i)
```

```
    // cup[i]=s[i];

    printf("The number of elements in the unordered array:%d\n
",n);
    /*or*/
    // printf("The number of elements in the nearly orderly ar
ray:%d\n",n);

    clock_t start,end;

    start=clock();
    BubbleSort(cup,n);
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("BubbleSort ERROR!\n");break;}
    if(i==n)
    printf("BubbleSort time-consuming:%d ms\n",end-start);

    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    SelectSort(cup,n);
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("SelectSort ERROR!\n");break;}
    if(i==n)
    printf("SelectSort time-consuming:%d ms\n",end-start);

    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    InsertSort(cup,n);
```

```
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("InsertSort ERROR!\n");break;}
    if(i==n)
    printf("InsertSort time-consuming:%d ms\n",end-start);

    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    QuickSort(cup,0,n-1);
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("QuickSort ERROR!\n");break;}
    if(i==n)
    printf("QuickSort time-consuming:%d ms\n",end-start);

    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    qsort(cup,n,sizeof(cup[0]),cmp);
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("qsort ERROR!\n");break;}
    if(i==n)
    printf("# qsort time-consuming:%d ms\n",end-start);

    int temparr[n];
    /*or*/
    // int *temparr=malloc(n*sizeof(int));

    for(int i=0;i<n;++i)
```

```
    temparr[i]=0;
    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    MergeSort(cup,temparr,0,n-1);
    end=clock();

    // free(temparr);

    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("MergeSort ERROR!\n");break;}
    if(i==n)
    printf("MergeSort time-consuming:%d ms\n",end-start);

    for(int i=0;i<n;++i)
    cup[i]=s[i];
    start=clock();
    CountSort(cup,n);
    end=clock();
    for(i=1;i<n;++i)
    if(cup[i]<cup[i-1])
    {printf("CountSort ERROR!\n");break;}
    if(i==n)
    printf("CountSort time-consuming:%d ms\n",end-start);

    // free(s);
    // free(cup);

    return 0;
}
void swap(int *x, int *y)
{
    int cup;
```

```
    cup=*x;
    *x=*y;
    *y = cup;
}
void BubbleSort(int arr[],int n)
{
    for(int i = 0; i < n; ++i)
    for(int j = n-1; j > i; --j)
    if(arr[j] < arr[j-1]) swap(&arr[j], &arr[j-1]);
}
void SelectSort(int arr[],int n)
{
    for(int i=0;i<n;++i)
        for(int j=i+1;j<n;++j)
            if(arr[i]>arr[j])
            swap(&arr[i],&arr[j]);
}
void InsertSort(int arr[],int n)
{
    for(int i=1;i<n;++i)
    {
        int j=i,temp=arr[i];
        while(arr[--j]>temp)
        swap(&arr[j],&arr[j+1]);
    }
}
void QuickSort(int arr[],int head,int foot)
{
    if(head>=foot) return;
    int first=head;
    int last=foot;
    srand(time(0));
    int mid=rand()%(foot-head+1)+head;
    int pivot=arr[mid];
```

```
    while(mid<last&&arr[last]>=pivot)
        --last;
    swap(&arr[mid],&arr[last]);
    while(first<last)
    {
        while(first<last&&arr[first]<=pivot)
            ++first;
        swap(&arr[first],&arr[last]);
        while(first<last&&arr[last]>=pivot)
            --last;
        swap(&arr[first],&arr[last]);
    }
    QuickSort(arr,head,last-1);
    QuickSort(arr,first+1,foot);
}
void MergeSort(int arr[],int temparr[],int first,int last)
{
    if (first>=last) return;
    int mid=(first+last)/2;
    MergeSort(arr,temparr,first,mid);
    MergeSort(arr,temparr,mid+1,last);
    merge(arr,temparr,first,mid,last);
}
void merge(int arr[],int temparr[],int first,int mid,int last)
{
    int former=first,latter=mid+1,cnt=first;
    while(former<=mid||latter<=last)
    {
        if(latter>last||arr[former]<arr[latter]&&former<=mid)
            temparr[cnt++]=arr[former++];
        else
        temparr[cnt++]=arr[latter++];
    }
```

```
   for(int i=first;i<=last;++i)
        arr[i]=temparr[i];
}
void CountSort(int arr[],int n)
{
    int max=arr[0];
    for(int i=1;i<n;++i)
    if(max<arr[i]) max=arr[i];
    int cnt[max+1],num=0;
    for(int i=0;i<max+1;++i)
        cnt[i]=0;
    for(int i=0;i<n;++i)
        ++cnt[arr[i]];
    for(int i=0;i<=max;++i)
        if(cnt[i])
            while(cnt[i]--)
            arr[num++]=i;
}
```