项目文档

谈瑞

项目十：几种排序算法比较

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 专业班级：软件工程4班  电话：18936361545 | 学号：1452775  电子邮件：tanrui106@163.com | 课目：数据结构课程设计  个人网站：http://guitoubing.top |
|  |  |  |

目录

[项目简介 3](#_Toc502775382)

[项目概要 3](#_Toc502775383)

[项目功能及要求 3](#_Toc502775384)

[项目结构 4](#_Toc502775385)

[性能分析 4](#_Toc502775386)

[特点比较 5](#_Toc502775387)

[项目类的实现 7](#_Toc502775388)

[Sorts类 7](#_Toc502775389)

[代码分析 8](#_Toc502775390)

[冒泡排序 8](#_Toc502775391)

[选择排序 8](#_Toc502775392)

[直接插入排序 9](#_Toc502775393)

[希尔排序 9](#_Toc502775394)

[快速排序 10](#_Toc502775395)

[堆排序 10](#_Toc502775396)

[归并排序 11](#_Toc502775397)

[基数排序 12](#_Toc502775398)

[一些思考 14](#_Toc502775399)

[快速排序为什么快？ 14](#_Toc502775400)

[称球问题 14](#_Toc502775401)

[快速排序的实质及比较 14](#_Toc502775402)

[实际情况 15](#_Toc502775403)

[运行测试 16](#_Toc502775404)

[排序数数量为10000时 16](#_Toc502775405)

[排序数量为100000时 17](#_Toc502775406)

# 项目简介

## 项目概要

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，希尔排序，堆排序，归并排序，基数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。

## 项目功能及要求

实现对八种排序算法的花费时间、查找次数的比较，并分析出各种排序算法的优劣

# 项目结构

本项目对八种排序方法进行了实现，输出了对10000个随机数进行排列时的花费时间及交换次数，同时对每种算法的实现方法进行了思考，分析了各类算法的优劣，而本项目在真正实现时也并非是对原标准算法照搬照抄，而是先理解其原理而后以自己的想法组织代码，因此在性能、代码形式等方面可能会与标准算法有或多或少的差异。

下面用表格的形式对八种算法进行一下比较：

## 性能分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 时间复杂度 | 花费时间/s | 交换次数/s | 空间复杂度 | 备注 |
| N=10000 | | | | | |
| 冒泡排序 | O(n2) | 0.245 | 24952888 | O(1) |  |
| 选择排序 | O(n2) | 0.108 | 9985 | O(1) |  |
| 直接插入排序 | O(n2) | 0.143 | 25087851 | O(1) |  |
| 希尔排序 | O(n\*log2n) | <0.002 | 52484 | O(1) |  |
| 快速排序 | O(n\*log2n) | <0.001 | 71515 | O(log2n)~O(n) |  |
| 堆排序 | O(n\*log2n) | 0.542 | 9999 | O(1) |  |
| 归并排序 | O(n\*log2n) | <0.001 | 1998 | O(n) |  |
| 基数排序 | O(n\*logrm) | 0.066 | 13857818 | O(n) | r为基数，m为堆数 |
| N=100000 | | | | | |
| 冒泡排序 | O(n2) | 28.017 | -1788922785 | O(1) | 排序次数溢出 |
| 选择排序 | O(n2) | 10.535 | 99986 | O(1) |  |
| 直接插入排序 | O(n2) | 11.463 | -1794262757 | O(1) | 排序次数溢出 |
| 希尔排序 | O(n\*log2n) | 0.019 | 647341 | O(1) |  |
| 快速排序 | O(n\*log2n) | 0.007 | 937215 | O(log2n)~O(n) |  |
| 堆排序 | O(n\*log2n) | \ | \ | O(1) | 所需额外空间过多，开辟失败 |
| 归并排序 | O(n\*log2n) | 0.021 | 17335 | O(n) |  |
| 基数排序 | O(n\*logrm) | 6.227 | 1389435727 | O(n) |  |

## 特点比较

|  |  |  |  |
| --- | --- | --- | --- |
|  | 核心思想 | 优点 | 缺点 |
| 冒泡排序 | 冒泡排序可以说是想法最简单的排序了，它重复的遍历序列，每次遍历将对每两个元素进行比较，较大的将后移，一次遍历下来，最大的元素浮到序列顶端 | 稳定的排序算法 | 慢 |
| 选择排序 | 选择排序基本思想是将序号从i到n的元素序列中的具有最小排序码的元素上调，调至子序列首位，重复操作直到i=n-1，排序即结束 | 移动的次数是固定的，对于n个元素就只需要n-1次移动 | 不稳定的排序算法  ，需要很多次比较 |
| 直接插入排序 | 插入排序，个人认为与选择排序的算法思想相似，均是每次将子序列长度加1后，进行调整。基本思想是第i次插入元素时，后面i-1个元素已经是排好序的了（这里采用了倒序插入，即从大到小），此时将其插入到其应该在的位置，如此反复n次序列就已经排好序了 | 稳定的排序算法，速度较快 | 需要进行大量的数据移动工作，这对于数组来说不友好，但是可以用链表解决这一问题 |
| 希尔排序 | 希尔排序是直接插入排序的一种改进方法，又叫缩小增量排序，其将序列按照一定的增量进行分组，对每组使用直接插入排序，逐渐缩小增量，当增量减小为1时，整个序列便排好序了 | 速度较快  数据移动量较少 | 不稳定的排序算法 |
| 快速排序 | 快速排序的基本思想是对序列进行分层，以序列第一个元素为基准，将排序码大的元素后移，排序码小的前移，此时该基准元素的位置已经是固定下来了，再对左右两个子序列递归前面的操作，最终即可得到排好序的序列了 | 速度极快  数据移动量较少 | 不稳定的排序算法 |
| 堆排序 | 堆排序时选择排序的一种，利用完全二叉树构建的最小堆，每次取堆顶元素放入新序列中并删除该元素，直到最小堆被清空，新序列即为所需序列 | 速度较快 | 不稳定的排序算法  额外空间很多  由于需要对堆进行经常性的删除操作，因此速度也不如快排 |
| 归并排序 | 归并排序使用分治法，对序列进行递归分割，直到子序列为1-2个元素，而后逆递归进行序列合并，递归结束后则可以得出有序的序列 | 速度较快  稳定的排序算法 | 需要开辟一个与原数组大小相同的数组 |

续表：

|  |  |  |  |
| --- | --- | --- | --- |
| 基数排序 | 基数排序采用分配的方法，对序列按照一定的基数进行分类，此项目中是按照序列中元素的各位位值进行排序，而后对每类元素递归进行此操作，当取到基数所能取得的最大值时，对每个子类采用直接插入排序的方法排序，最终即可得到有序序列 | 稳定的排序算法  速度较快 | 选择合适的基数以及将序列分成多少类需要考虑 |

# 项目类的实现

## Sorts类

protected成员

public成员

|  |  |
| --- | --- |
| 类成员 | 作用 |
| Sorts(); | 构造函数 |
| ~Sorts(); | 析构函数 |
| void getResult(int operate,  double\_time); | 获取某一排序的结果，包括运行时间、交换次数等 |
| void SetOperate(); | 选择排序方法 |
| void BubbleSort(); | 冒泡排序 |
| void SelectSort(); | 选择排序 |
| void InsertSort(); | 直接插入排序 |
| void ShellSort(); | 希尔排序 |
| void FastSort(); | 快速排序 |
| void HeapSort(); | 堆排序 |
| void MergeSort(); | 归并排序 |
| void RadixSort(); | 基数排序 |
| void InsertSort  (int left, int right); | 选择排序 | |
| void FastSort(int\* nums,  int left, int right); | 快速排序 | |
| void MergeSort(int left,  int right); | 归并排序 | |
| void Merge(int left,  int flag, int right); | 归并函数，是对两个子序列进行合并的函数 | |
| void RadixSort(int \* count,  int radix,  int left, int right); | 基数排序 | |
| int \*nums; | 序列数组 | |
| int search\_count,  swap\_count, sort\_type; | 搜索次数、交换次数、交换类型 | |
| ofstream out\_file; | 输出的文件流 | |
| bool ifPrint; | 是否输出至文件 | |
|  |  | |

# 代码分析

### 冒泡排序

void Sorts::BubbleSort() {

// 冒泡排序可以说是想法最简单的排序了，它重复的遍历序列，每次遍历将对每两个元素进行比较，较大的将

//后移，一次遍历下来，最大的元素浮到序列顶端，其时间复杂度为O(n2)

for (int i = 0; i < MAXAMOUNT - 1; i++) {

for (int j = 0; j < MAXAMOUNT - i - 1; j++) {

search\_count++;

if (nums[j]>nums[j+1]){

swap\_count++;

int temp = nums[j];

nums[j] = nums[j+1];

nums[j+1] = temp;

}

}

}

}

### 选择排序

void Sorts::SelectSort() {

// 选择排序基本思想是将序号从i到n的元素序列中的具有最小排序码的元素上调，调至子序列首位，重复操作

//直到i=n-1，排序即结束，其时间复杂度为O(n2)

for (int i = 0; i < MAXAMOUNT; i++) {

int min = i;

for (int j = i; j < MAXAMOUNT; j++){

search\_count++;

if (nums[min] > nums[j]){

min = j;

}

}

if (min != i){

//当子序列最小排序码元素不是首元素即需要进行调整交换

swap\_count++;

int temp = nums[min];

nums[min] = nums[i];

nums[i] = temp;

}

}

}

### 直接插入排序

void Sorts::InsertSort() {

InsertSort(1, MAXAMOUNT-1);

}

void Sorts::InsertSort(int left, int right) {

// 插入排序，个人认为与选择排序的算法思想相似，均是每次将子序列长度加1后，进行调整。时间复杂度也为O(n2)

//基本思想是第i次插入元素时，后面i-1个元素已经是排好序的了（这里采用了倒序插入，即从大到小），此时将其插

//入到其应该在的位置，如此反复n次序列就已经排好序了

// 从输出可以看到选择排序查找次数大约为50005000次，而插入排序查找次数大约为25002500次，这是由于插入

//排序的最内层循环还有个跳出语句，根据数的随机性，其跳出的概率大约是50%

for (int i = left; i <= right; i++) {

int j = i-1;

for (; j >= 0; j--) {

swap\_count++;

search\_count++;

if (nums[j] <= nums[j+1]){

break;

}

int temp = nums[j];

nums[j] = nums[j+1];

nums[j+1] = temp;

}

}

}

### 希尔排序

void Sorts::ShellSort() {

//希尔排序是直接插入排序的一种改进方法，又叫缩小增量排序，其将序列按照一定的增量进行分组，对每组使用直接插入排序，逐渐缩小增量，当增量减小为1时，整个序列便排好序了

int flag = MAXAMOUNT, temp, j;

while (flag!=1){

flag = (int)(ceil(flag/3) + 1);

for (int i = flag; i<MAXAMOUNT; i++){

search\_count++;

if (nums[i]<nums[i-flag]){

swap\_count++;

temp = nums[i], j = i-flag;

while (j >= 0 && temp < nums[j]){

nums[j+flag] = nums[j];

j-=flag;

}

nums[j+flag] = temp;

}

}

}

### 快速排序

void Sorts::FastSort() {

FastSort(nums, 0, MAXAMOUNT-1);

}

void Sorts::FastSort(int \*nums, int left, int right) {

// 快速排序的基本思想是对序列进行分层，以序列第一个元素为基准，将排序码大的元素后移，排序码小的

//前移，此时该基准元素的位置已经是固定下来了，再对左右两个子序列递归前面的操作，最终即可得到排

//好序的序列了

if (left >= right){

return;

}

int temp = nums[left];

int flag = left;

for (int i = left; i <= right; i++){

search\_count++;

if (nums[i] < temp){

//如果排序码比基准元素小，则将其移至基准元素前，若比基准元素大，则不需要移动

swap\_count++;

if (flag == i - 1){

//该元素与基准元素相邻时，直接调换二者位置

nums[flag++] = nums[i];

nums[i] = temp;

}

else{

//该元素与基准元素不相邻时，则将该元素与基准元素后一位调换位置，而后再调换该元素与基准元素

int temp2 = nums[i];

nums[i] = nums[flag + 1];

nums[flag + 1] = temp;

nums[flag++] = temp2;

}

}

}

//对左右两个子序列递归调用FastSort操作

FastSort(nums, left, flag-1);

FastSort(nums, flag + 1, right);

}

### 堆排序

void Sorts::HeapSort() {

Heap heap(nums, MAXAMOUNT);

// Heap heap(nums, 10);

for (int i = 1; i < heap.currentSize; ++i) {

heap.swap(i, heap.currentSize);

}

// heap.printHeap();

search\_count = heap.getSearchCount();

swap\_count = heap.getSwapCount();

for (int i = 0; i < heap.currentSize; ++i) {

nums[i] = heap.elems[i];

}

}

### 归并排序

void Sorts::MergeSort() {

MergeSort(0, MAXAMOUNT-1);

}

void Sorts::MergeSort(int left, int right) {

// 归并排序的基本思想时将序列不断折中，当某一序列折中为“1个元素”+“2个元素”或者“2个元素”+“1个元素”或

//者“2个元素”+“2个元素”时为递归结束条件，此时对2个元素的进行排序，1个元素的无操作，而后调用Merge函数

//将这两个子序列归并起来，如此反复递归便可最终得到有序序列

search\_count++;

if (right == left){

//1个元素的子序列直接返回不需要操作

return;

} else if (right - left == 1){

//2个元素的子序列若有序则直接返回，否则交换后返回

if (nums[left] > nums[right]){

swap\_count++;

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

}

return;

}

int flag = (right + left)/2;

MergeSort(left, flag);

MergeSort(flag+1, right);

Merge(left, flag, right);

//分割出的两个子序列通过这个函数归并到一起

}

void Sorts::Merge(int left, int flag, int right) {

// 归并函数是归并排序算法中的主要部分，其主要思想就是将逐个遍历两个子序列，依次选择较小的元素放入新开辟

//的数组中去，直到某一子序列到了末尾位置，此时将另一个序列剩余的元素直接放入序列中，最后将该数组赋值回原

//数组。即可得到两子序列归并后的序列

int temp = left, tempFlag = flag;

int \*copyNums = (int\*)malloc(sizeof(int) \* (right - left + 1));

for (int i = 0; i < right-temp+1; i++) {

if ((nums[left] < nums[flag+1] || flag == right)&&(left != tempFlag+1)){

copyNums[i] = nums[left];

left++;

} else{

copyNums[i] = nums[flag+1];

flag++;

}

}

for (int j = temp; j <= right; j++) {

nums[j] = copyNums[j-temp];

}

free(copyNums);

}

### 基数排序

void Sorts::RadixSort() {

int count[MAXAMOUNT];

for (int i = 0; i < MAXAMOUNT; ++i) {

count[i] = 0;

}

RadixSort(count, 1, 0, MAXAMOUNT-1);

}

void Sorts::RadixSort(int \*count, int radix, int left, int right) {

// 基数排序的主要思想是用若干个基数将序列区分为若干个类（形象来说，就是将相同类的元素放到同一个桶中），

//而后对每类元素进行排序

// 这里基数选用的是元素的位数，由于rand()随机函数产生的随机数范围在0-RAND\_MAX之间，而RAND\_MAX的最大

//值为2147483647（此为stdlib.h中宏定义的一个字符常量），因此这里我定义的MAXRADIX为10

// 基数排序每次递归仍然是对序列分层，而后将分好层的序列左子序列使用插入排序算法排序，右子序列继续递归分层，直到radix

//取到MAXRADIX，此时对剩下的序列直接利用插入排序算法排序

if (radix == MAXRADIX){

InsertSort(left, right);

return;

}

int tempLeft = left;

for (int i = left; i <= right; ++i) {

if ((int)(nums[i]/(pow(10, radix))) == 0){

//10的radix次方为每次分层的依据，找到所有小于此值的元素，将其移动到序列的左边

if (tempLeft != i){

int temp = nums[tempLeft];

nums[tempLeft] = nums[i];

nums[i] = temp;

}

tempLeft++;

count[radix]++;

}

}

InsertSort(left, count[radix]+left-1);

RadixSort(count, radix+1, left+count[radix], right);

}

# 一些思考

## 快速排序为什么快？

### 称球问题

想到这个问题是看了一篇名叫《[数学之美：快排为什么那样快](http://mindhacks.cn/2008/06/13/why-is-quicksort-so-quick/)》的文章，里面讨论了关于快速排序之所以快却又不那么快的原因。这里做一些简短记录。

里面讲到了一个经典的智力题，即“称球问题”。12个球里面有一个坏球，用天平最少多少次可以找出这个球，并确定该球是轻了还是重了。原文看得我有点混，这里提一个较为简单的想法（当然比文中方法要来的劣），由于天平具有平衡、失衡2种可能结果，现将12个球分成3组，每次称量可确定坏球在某个分组内。不失一般性，设分成A(4)、B(4)、C(4)，括号内表示分组内球的数量，称量次数为n，对A和B进行称量（n=1），则会有两种情况：“A!=B”和“A=B” （概率各为1/2，下面括号内分数均表示条件概率）。

1. 当A=B时，说明坏球在C组，此时C分成三组C1(2)、C2(1)、C3(1)三组，对C1两个球称量(n=2)，有相等（1/2）与不等（1/2）两种情况：ⅰ若相等，则需要将剩下的两个球分别与C1中两个球比较，n=3（1/2）或4（1/2）；ⅱ若不等，C1中任取一个与C2比较，比较一次即可得出坏球，n=3（1）
2. 当A!=B时，说明坏球在A组或是B组，重复①操作，则可能需要①中的n（1/2）或二倍①中的n（1/2）

经上述讨论，可总结出：（**而此题若是已知坏球是轻还是重，则只需要3次便可以确定坏球是哪个。**）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 次数 | 3 | 4 | 6 | 8 |
| 概率 | 9/16 | 3/16 | 3/16 | 1/16 |

### 快速排序的实质及比较

排序的实质：N个元素一共有N！种排列方法，排序就是要找到特定的一种排序。快排基于比较的思想正是利用上述思想，每次对于比较两个数时只有“>=”和“<”两种情况，且对于特定的pivot轴元素，其概率是近似相等的都为1/2，意即没有哪个分支是其弱点，若是每次轴元素与所有元素比较时均是上面的完美情况，那么N个元素排查结束只需要“log2N!”次，当N很大时，其要小于N\*log2N次，这说明最优情况要远小于其平均时间复杂度，而最坏情况毋庸置疑是N2。而快排快就快在其取得较优的情况的可能性很大，而取得较坏的情况的可能性很小。

冒泡排序慢是因为其进行了过多的重复比较；选择排序是因其存在弱点分支，即找到最大或最小的元素这个事件发生与否的概率是不均等的；插入排序与选择排序一样，在i个空位中寻找到自己的空位是不常发生的。

## 实际情况

实际操作中，我发现快速排序并不总是“快”的，看一下希尔排序、快速排序、归并排序在1亿个数以下的排序时间（由于我写的排序算法与各类算法官方版本有出入，因此这里的时间消耗也与标准时间消耗有出入，当然比各类算法的完美版本要来的慢得多）：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 数量/万 | | 10万 | 50万 | 100万 | 500万 | 1000万 | 2000万 | 5000万 |
| 时间/s | 希尔排序 | 0.033 | 0.132 | 0.291 | 1.677 | 3.876 | 7.585 | 17.845 |
| 快速排序 | 0.016 | 0.077 | 0.186 | 1.673 | 5.039 | 17.601 | 86.694 |
| 归并排序 | 0.016 | 0.127 | 0.262 | 1.422 | 2.752 | 6.009 | 12.322 |

500万个元素以下，快速排序还是很好的，但是超过500万后，快速排序并不如希尔排序和归并排序来的快，究其原因我认为有以下几点：

1. 快速排序运用的递归是其最大的短板，当N极大时，需要的辅助栈空间也将变得极大，使得快速排序在大数排序中显得有些力不从心。若是采用非递归法，其运行时间应该会大有改善，有时间可研究研究。
2. 这里的枢轴值没有采用media-of-three方法，即从开头-中间-末尾选择中间值作为轴值，使得其划分恶化的情况出现了较大的可能性。
3. 希尔排序之所以在N极大时仍然能保持良好的运行时间，我想是其没有使用递归栈
4. 归并排序本身将主要操作转移到了合并子序列上，其运行时间随N的增大变化较为固定，原因是其最终仍然转化为2个和1个元素或2个和2个元素的子序列排序归并问题
5. 最后一个原因，我感觉也是最重要的一个原因，rand()函数产生的随机数在千万级数量下，会有很多很多的重复值，这种情况对于稳定算法是很友好的，而对于快排这种不稳定算法却是很糟糕的。

# 运行测试

## 排序数数量为10000时

## 排序数量为100000时

当排序量上升到百万、千万级时，冒泡、选择、直接插入、堆排序、基数排序都将很展现出很低的效率，我跑了很久每个结果，这里便没再测试贴出（见“[思考](#_一些思考)”中的[大数排序](#_实际情况)）。