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： 肖杨

学 号： 1950430

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 3

2 设计 3

3 实现 4

4 测试 15

5 比较 17

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移动次数。其中，随机数的个数由用户定义，系统产生随机数。并且显示他们的移动次数。

1.2 文档内容

文档包括了总体系统的设计和每个排序方式的实现介绍，同时介绍了每种排序方式的实现效率，并给出了相应测试数据和对排序方式的比较分析。

2 设计

对于整个系统而言，首先接收数据数目，然后根据数据数目生成一个随机数数列，对于每种排序都使用相同的数列并在排序过程中记录比较次数、赋值次数、运行时间，最后将各个排序的结果输出。

对于随机数，每次调用系统时间进行随机数种子的设置，再按数组大小一一赋值。

对于各个排序，按照排序的基本实现方法进行实现，除去添加了用于记录次数的函数以外，完全为八种基本排序。

比较的数据由结构体记录如下：

struct Data

{

long long move = 0;//赋值、交换次数

long long cmp = 0;//比较次数

long long time = 0;//用时

}bubble,quick,selection,heap,insertion,shell,merge,radix;

3 实现

3.1 冒泡排序的实现

3.1.1 冒泡排序的算法

对于含有n个元素的序列，从下标为0的元素开始，依次与后续元素比较，若前一个元素大于后一个元素，则交换二者位置。如此比较，直到到达数组末尾下标为n-1的元素，此时数组末尾保存的为数组中的最大元素。随后，继续从下标为0的元素开始比较至下标为n-2的元素。第i+1次循环时，从0比较至n-i-1，直到i+1=n时不再继续比较，此时已经按照非降序排列。

考虑到可能出现某次交换后已经有序的情况，因此可以添加一个变量用于记录一趟比较后是否发生交换，若没有交换说明此时数组已经有序，直接退出循环即可。

冒泡排序是一种用时较长的排序算法，时间复杂度为O()，空间复杂度为O(1)。在不提前截断的情况下，数据的比较次数和初始排列无关，但数据的交换次数和初始排列有关。冒泡排序一趟循环后，保证相同的元素不会被交换，不会破坏其有序性，因此是稳定的。

3.1.2 核心代码

void BubbleSort(int nums[],const int size)

{

bool changed = false;

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

{

changed = false;

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

{

bubble.cmp++;

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

{

bubble.move++;

swap(nums[j], nums[j + 1]);

changed = true;

}

}

if (!changed)

{

break;

}

}

}

3.2 快速排序的实现

3.2.1 快速排序的算法

快速排序每次选择数据中的关键码，将其余数据中小于关键码的数据移动至数组中关键码左侧，大于关键码的数据移动至关键码的右侧，并对关键码左右的数据分别再进行一次快速排序。当关键码左侧、右侧均只有小于等于1个元素时，说明已局部有序，为递归中止条件，在下次递归时会直接返回。当递归全部结束时，说明排序完成。本程序中的关键码选择左边界的数据。

快速排序每次将数据分为三块，分别为左侧较小数据，中间已有序数据（关键码），右侧较大数据，再对其进行递归，平均时间复杂度为O()，平均空间复杂度为O()（递归占用空间）。最差情况下，快速排序会退化为冒泡排序，则时间复杂度为O()。

快速排序的比较次数和移动次数和初始排列关系较大，会影响到多次排序的关键码选择，对于相同数据的不同排列情况，快速排序的用时也是不同的，且由于其交换只保证区间和关键码的关系，不保证相同数据的顺序，已经有序的序列可能由于和关键码交换的顺序不同而被改变，因此是不稳定的。

3.2.2 核心代码

void QuickSort(int nums[], const int low, const int high)

{

if (low >= high)

return;

int i = low;

int j = high + 1;

int key = nums[low];

while (i < j)

{

while (nums[--j] > key)

{

quick.cmp++;

if (i == j)

{

break;

}

}//找到从右向左第一个小于key的数

if (i == j)

{

break;//j所指的数为key，右侧已全部大于key

}

quick.move++;

swap(nums[i], nums[j]);//否则交换，使nums[j]右侧全部大于key

while (nums[++i] < key)

{

quick.cmp++;

if (i == j)

{

break;

}

}

if (i == j)

{

break;

}

quick.move++;

swap(nums[i], nums[j]);//进行相反操作

}//此时key左侧全部小于key，右侧全部大于key

QuickSort(nums, low, j - 1);//缩小区间继续排序，直到升序排列

QuickSort(nums, j + 1, high);

}

3.3 选择排序的实现

3.3.1 选择排序的算法

对于含有n个元素的数组，选择排序通过循环遍历数组，每次循环找到最小的元素并将其交换至第一个元素处，若第一个元素为最小的元素则不进行交换。第i+1次循环时，从下标为i的元素开始向后比较，直到i+1=n时比较完毕，此时数组已有序。

简单选择排序也属于用时较长的排序，时间复杂度为O(), 空间复杂度为O(1)。显然，对于选择排序，每次排序都要找到剩余元素中的最小元素，必须遍历后续数据，因此比较次数和初始排列无关，但交换次数由于最小元素可能是第一个元素，和初始排列有关。对于相同的数据，由于其元素不一定是剩余元素中最小的元素，将会和后续最小的元素交换，从而破坏了有序性，因此选择排序是不稳定的。

3.3.2 核心代码

void SelectionSort(int nums[], const int size)

{

int k = 0;

for (int i = 0; i < size; i++)

{

k = i;

for (int j = i + 1; j < size; j++)

{

selection.cmp++;

if (nums[k] > nums[j])

{

k = j;

}

}

if (k != i)

{

selection.move++;

swap(nums[k], nums[i]);

}

}

}

3.4 堆排序的实现

3.4.1 堆排序的算法

堆排序首先需要根据初始数组建立最大堆，自下向上依次调整为堆。整体排序选择原地排序，不使用额外空间储存结果数组。建立堆后，堆的首个元素为最大元素，将其与最后一个元素进行交换，使堆大小减小1，并重新将剩余元素调整为堆。反复执行该操作，直到堆大小为1时排序完成。

堆排序利用了最大堆的性质，即每个元素都大于其子树上的元素，每次将最大元素交换至最后则使其不必占用额外空间（如使用最小堆实现升序排列，通常需要额外空间）。整体而言利用堆的性质加快了排序，时间复杂度为O()，空间复杂度为O(1).数据的比较次数和移动次数在建堆、调整为堆时受到初始排列影响，与初始排列有关。由于使用数组实现堆，且每次将最大元素移动至数组最后，其中的相同数据序列会在重新调整或移动最大元素后被破坏，因此是不稳定的。

3.4.2 核心代码

void ShiftDown(int nums[], int index, int max)

{

int child = 2 \* index + 1, first = nums[index];

while (child <= max)

{

heap.cmp += 3;

if (child < max&&nums[child] < nums[child + 1])

{

child++;

}//child指向较大孩子

if (first > nums[child])

{

break;

}

else

{

heap.move++;

nums[index] = nums[child];

index = child;

child = 2 \* child + 1;

}

}

heap.move++;

nums[index] = first;

}

void HeapSort(int nums[], const int size)

{

int i = (size - 2) / 2;

while (i >= 0)

{

ShiftDown(nums, i--, size - 1);

}

for (i = size - 1; i > 0; i--)

{

heap.move++;

swap(nums[i], nums[0]);//最大元素排到最后

ShiftDown(nums, 0, i - 1);

}

}

3.5 插入排序的实现

3.5.1 插入排序的算法

对于n个待排序元素，直接插入排序会从第2个元素开始，依次插入第i个元素。插入第i个元素时，前i-1个元素已经有序，通过不断向前比较找到第i个元素应插入的位置j后，将j到i-1的元素依次向后移动一个位置，并将第i个元素插入第j个位置。当插入到第n个元素时，排序结束。

直接插入排序的时间复杂度为O()，空间复杂度为O(1),最优情况下每个元素只需要一次比较，时间复杂度为O(n)。显然，由于元素依次向前比较，直到找到相应位置为止，比较和移动次数和初始排列有关。对于一些相同元素，插入排序会保证其相对位置不变，较小下标的元素始终保持在下标较大元素之前，因此直接插入排序是稳定的。

3.5.2 核心代码

void InsertionSort(int nums[], const int size)

{

for (int i = 0; i < size; i++)

{

int key = nums[i];

int j = i - 1;

insertion.cmp++;//作为进入条件的比较是否忽略对结论无影响

while (j >= 0 && nums[j] > key)

{

insertion.cmp++;

insertion.move++;

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

j--;

}

insertion.move++;

nums[j + 1] = key;

}

}

3.6 希尔排序的实现

3.6.1 希尔排序的算法

希尔排序，首先选择一个gap作为数据间隔，将全部元素划分为gap个子序列，将左右距离为gap的元素放在一个子序列中进行插入排序。随后，减小gap，重复进行子序列的划分和排序，直到gap为1时，最后进行一遍插入排序。此时由于多段子序列已经有序，整体相对有序，因此排序花费时间较直接插入排序较少。

对于gap序列的选择，有选择gap为n/2，每次使gap变为一半的取法，但这种序列在多数情况下会使奇偶位置的数据无法交叉，尤其是对于2^n（n取整数）个数据而言，奇偶位置直到最后一次才能交叉，整体有序性较差。对于gap序列，目前还没有最优解，常见已证明的优化后序列有Hibbard 增量序列、Knuth 增量序列、Gonnet 增量序列、Sedgewick 增量序列等。

本程序选择使用Sedgewick序列，其通项公式为：

hi=max(9∗4j−9∗2j+1,4k−3∗2k+1)

平均时间复杂度被估计为O()，最差时间复杂度为O（）, 空间复杂度为O(1)。

由于使用到插入排序的思想，数据比较次数和移动次数和初始排列有关。在每次改变增量时，都可能使相同元素的相对位置被改变，因此希尔排序是不稳定的。

3.6.2 核心代码

void ShellSort(int nums[], const int size)

{

int Sedgewick[30] =

{ 0, 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289,

64769, 146305, 260609, 587521, 1045505, 2354689, 4188161, 9427969, 16764929,

37730305, 67084289, 150958081, 268386305, 603906049, 1073643521 };

int index = 0, i = 0, j = 0;

int key;

int gap;

while (Sedgewick[index] < size / 2)

{

index++;

}

gap = Sedgewick[index];

while (gap > 0)

{

for (i = gap; i < size; i++)

{

j = i;

key = nums[j];

shell.cmp++;

while (j >= gap && nums[j - gap] > key)

{

shell.cmp++;

shell.move++;

nums[j] = nums[j - gap];

j -= gap;

}

nums[j] = key;

}

gap = Sedgewick[--index];

}

}

3.7 归并排序的实现

3.7.1 归并排序的算法

归并排序首先将n个元素分成两个含有n/2个元素的序列，再分别对两个子序列进行归并排序。当子序列的长度为1时，直接返回。两个子序列的归并排序都执行完毕后，再进行两个序列的合并，利用额外储存空间储存合并后的有序序列，并存回原有序列。进行两个含有n/2个元素序列的合并后，排序完成。

合并的过程中，两个待合并序列均有序，利用双指针法，分别比较当前指针所指向元素的大小，取较小者存入新数组并将指针后移，直到一个序列全部存入新数组，此时将另一个序列的剩余元素全部存入新数组，并将新数组的内容存回原有的地址。

归并排序不断将待排序数组二分后进行排序，时间复杂度为O()，递归时和合并时使用额外空间，空间复杂度为O(n)。由于每次比较都会有元素存入新数组，数据比较次数和移动次数和初始序列无关。每次归并时若出现相同元素，优先选择左侧元素，使得相同的左侧元素始终在左侧，因此归并排序是稳定的。

3.7.2 核心代码

void Merge(int nums[], const int left, const int middle, const int right)

{

int i = left, j = middle + 1;

const int size = right - left + 1;

int\* temp = new int[size];

int k = 0;

while (i <= middle && j <= right)

{

merge.cmp++;

merge.move++;

if (nums[i] <= nums[j])

{

temp[k++] = nums[i++];

}

else

{

temp[k++] = nums[j++];

}

}

while (i <= middle)

{

merge.move++;

temp[k++] = nums[i++];

}

while (j <= right)

{

merge.move++;

temp[k++] = nums[j++];

}

k = 0;

for (int p = left; p <= right; p++)

{

nums[p] = temp[k++];

}

delete[]temp;

}

void MergeSort(int nums[], const int left, const int right)

{

if (left < right)

{

int middle = (left + right) / 2;

MergeSort(nums, left, middle);

MergeSort(nums, middle + 1, right);

Merge(nums, left, middle, right);

}

}

3.8 基数排序的实现

3.8.1 基数排序的算法

基数排序和其他排序算法有着较大不同。基数排序没有数据之间的大小比较，只根据其本身的数据进行若干次收集。基数的选取方式很多，本项目选择2作为基数，从而能够利用位运算简化代码、提升效率。首先进行一遍遍历找到最大的数，得到其二进制位数i，然后进行i遍收集，第k遍收集根据其第k位的位数将其放入pots[0]或pots[1]，再将pots中的所有元素放回原数组进行下一次收集。

需要注意的是，若数组中有n个数，pots[n]数组中的每一个数组长度必须为n，从而使得收集时不会出现越界。

基数排序的平均时间复杂度为O(d(n+r))，在这里d为分配、收集次数，r为基数，对于本排序，可以认为平均时间复杂度为O()，Max为原始数据中最大的数，应向上取整表示最高二进制位数。由于开辟了两个大小为n的数组，空间复杂度为O(n)。基数排序不是基于比较的排序，所有数据的移动次数只取决于序列中的最大值，对于最大值相同、长度为n的序列，其排序过程的步数完全相同。由于每次排序时按从前到后的顺序进行收集，相同的数据会保持相同的相对顺序始终在同一个桶中，因此基数排序是稳定的。

3.8.2 核心代码

void RadixSort(int nums[],const int size)

{

int max = nums[0];

int i, j, k;

for (i = 0; i < size; i++)

{

if (max < nums[i])

{

max = nums[i];

}

}

int bits = 0, presentBit = 0;

int index = 0;

while (max > 0)

{

max >>= 1;

bits++;

}//得到二进制位数

int \*\*pots = new int\*[2];

pots[0] = new int[size];

pots[1] = new int[size];

int potsize[] = { 0,0 };

for (; presentBit < bits; presentBit++)

{

for (i = 0; i < size; i++)

{

index = (nums[i] >> presentBit) & 1;//优于余除2

pots[index][potsize[index]++] = nums[i];

radix.move++;

}

k = 0;

for (i = 0; i < 2; i++)

{

for (j = 0; j < potsize[i]; j++)

{

nums[k++] = pots[i][j];

radix.move++;

}

potsize[i] = 0;

}

}

delete[] pots[0];

delete[] pots[1];

delete[] pots;

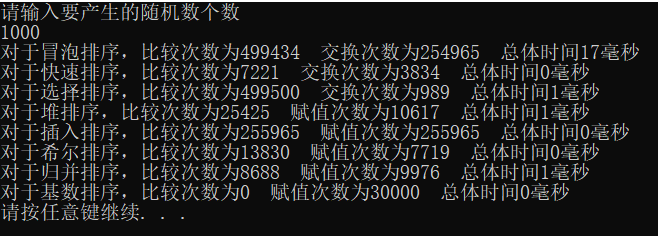
}

4 测试

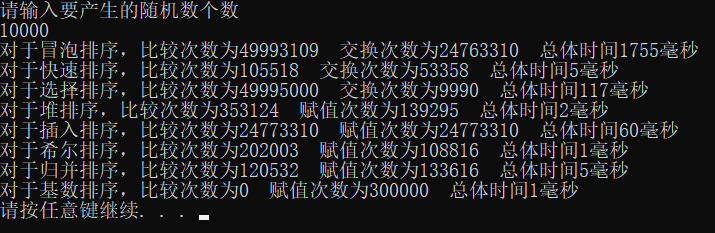
4.1 100个随机数



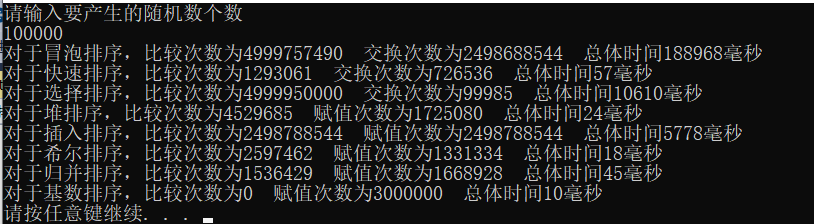
4.2 1000个随机数



4.3 10000个随机数



4.4 100000个随机数



5 比较

5.1 排序算法的比较

通过对八种排序算法在不同规模下的时间和比较、交换时间进行比较，得出结论：

1.冒泡排序、直接插入排序和简单选择排序都是最基本的排序方法。其中冒泡排序任何情况下的排序状态都是最差的，以至于不使用优化的情况下对于大量数据的排序时间和比较次数过长。直接插入排序和简单选择排序在排序时由于减少了数据的交换次数，因此对于大量数据的处理相对冒泡排序更为有效。

2.三者对于少量数据来说效率相对其他排序无较大差别，但占用内存较少，因此使用方便。

3.对于堆排序，其利用了堆的性质，因此在使用常量空间的情况下能够高效处理数据。但在数据较多时，由于单次调整的时间复杂度为O()，调整次数又较大，消耗时间较长。在10万级别的数据上该特征不够明显，若继续增加数据量其耗时提升较大而其他四种排序算法提升较少。

4.对于希尔排序，其利用了不同增量序列的性质，从而在使用常量空间的情况下使得其时间复杂度较低，但其排序时间波动很大，对于相同规模的不同数据差异很大。

5.对于快速排序，其利用额外空间换取时间，但其受到不同数据的影响很大，尤其是可能出现退化为冒泡排序的情况。但对于均匀分布的随机数据而言其平均排序效果较好，在不考虑极端情况的状况下可以使用快速排序。

6.对于归并排序和基数排序，二者属于稳定且高效的复杂排序。基数排序的优点在于对最大值和规模相同的数据其排序时间固定，尤其是对最大值较小的数据排序性能较好，而归并排序主要是保持了稳定性。

7.理论上，堆排序和希尔排序相比，堆排序的复杂度较低，但实际测试中耗时较长，这是因为和在合理的数据规模下差距不大，且实际执行时会有影响。1亿的规模下，刚刚超过，而规模再大时二者的排序效率则不够高，或由于内存限制不再使用内排序，因此表现出的执行效率希尔排序较高。

8.对于快速排序，在数据分散的情况下表现较好，而由于本实验中生成的随机数为32位int，最大数很容易长32位，使得总体排序时间较长，若限制随机数大小，可以发现基数排序应当效率高于快速排序。

5.2 排序方式的选择

对于规模较小的数据，若不要求稳定性，可以使用冒泡排序或简单选择排序；若要求稳定性或相对有序，可以使用直接插入排序。数据规模较大时，若数据随机分布且极端情况较少，可以使用快速排序；若对内存要求较高，可以使用空间复杂度为常数的堆排序和希尔排序；若规模更大且对时间要求较高或要求稳定性，可以采用归并排序或基数排序。