排序方法

1. 需求分析

排序是计算机程序设计中的一种重要操作，其功能是对一个[数据元素](http://baike.so.com/doc/6144813-6357989.html)集合或序列重新排列成一个按数据元素某个相知有序的序列。排序分为两类:内排序和外排序。

内部排序是指待排序列完全存放在内存中所进行的排序过程，适合不太大的元素序列。

其中快速排序的是目前排序方法中被认为是最好的方法。

[内部排序方法](http://baike.so.com/doc/1524299-1611470.html):

1.[插入排序](http://baike.so.com/doc/4301762-4505553.html)([直接插入排序](http://baike.so.com/doc/6834898-7052121.html));

2.快速排序;

3.[选择排序](http://baike.so.com/doc/5722712-5935443.html)([简单选择排序](http://baike.so.com/doc/5992150-6205119.html));

4.[归并排序](http://baike.so.com/doc/4608506-4820714.html);

5.冒泡排序;

6.[希尔排序](http://baike.so.com/doc/6137233-6350396.html);

[希尔排序](http://baike.so.com/doc/6137233-6350396.html)是对[直接插入排序](http://baike.so.com/doc/6834898-7052121.html)方法的改进。

7.[堆排序](http://baike.so.com/doc/1008529-1066354.html);

1. 概要设计

1.直接插入排序（Straiht Insertion Sort）

　　　　算法描述：如果有一个已经排好序的序列 {R(20),R(35),R(88)},当要插入一个R(66)时，需要与各个元素进行比较，R(35)<R(66)<R(88)，所以应该插在R(35)与R(88)直接。

　　　　算法开始时,取一个元素为原序列，然后重复执行上面的方法，将每个元素插入到序列中。

（1）.直接插入排序（Straiht Insertion Sort）

　　　　算法描述：如果有一个已经排好序的序列 {R(20),R(35),R(88)},当要插入一个R(66)时，需要与各个元素进行比较，R(35)<R(66)<R(88)，所以应该插在R(35)与R(88)直接。

　　　　算法开始时,取一个元素为原序列，然后重复执行上面的方法，将每个元素插入到序列中。

//插入排序

void InsertSort(int a[], int n)

{

int i, j;

int temp;

for(i = 1; i < n; i++)

{

temp = a[i];

j = i-1;

while(j >= 0 && a[j] > temp;)

{

a[j+1] = a[j];

--j;

}

a[j+1] = temp;

}

}

//递归的插入排序

void RecursiveInsertSort(int a[], int n)

{

int i, j, key;

if(n > 1)

{

RecursiveInsertSort(a,n-1);

}

key = a[n-1];

i = n-2;

while(i >= 0 && a[i] > key)

{

a[i+1] = a[i];

i--;

}

a[i+1] = key;

}

//折半插入排序

void BinInsertSort(int a[], int n)

{

int i,j;

for(i = 1; i < n; i++)

{

// 在a[0..i-1]中折半查找插入位置使a[high]<=a[i]<a[high+1..i-1]

int low = 0, high = i-1, m = 0;

while(low <= high)

{

m = m + (high-low)/2;

if (a[i] < a[m]) high = m-1;

else low = m+1;

}

// 向后移动元素a[high+1..i-1]，在a[high+1]处插入a[i]

int x = a[i];

for (j = i-1; j > high; j--)

a[j+1] = a[j];

a[high+1] = x; // 完成插入

}

}

此算法的时间复杂度为O(n2)

**2.快速排序**

快速排序是一种基于交换的排序方法，最常见的有冒泡排序（BubbleSort），快速排序（改进的冒泡排序）(QuickSort)

下面先说冒泡排序：

冒泡排序的基本思想是在一次排序中，将最大的元素沉入底部，然后缩小范围，继续进行。

具体的说：取第一个元素，然后与第二个元素进行比较，如果比第二个大，那么交换，否则，不交换，然后取第二个元素与第三个元素比较，同样用前面的方法，大则交换，直到将最大的元素交换到最底部，这是第一遍排序结束，然后，缩小范围，从第二个元素开始，在此运用上面的一遍排序方法。直到范围缩小为一个元素的时候，排序结束。

快速排序是不稳定的。最理想情况算法时间复杂度O(nlog2n)，最坏O(n^2)。

//快速排序

void QuickSort(int a[], int low, int high)

{

if(low < high)

{

// 划分

int pivot = a[low];

int i = low; int j = high;

while(i < j)

{

while(i<j && a[j] >= pivot) j--;

a[i] = a[j];

while(i<j && a[i] <= pivot) i++;

a[j] = a[i];

}

a[i] = pivot;

// 对子序列快排

QuickSort(a, low, i-1);

QuickSort(a, i+1, high);

}

}

//快速排序法的划分操作

int Partition(int vec[],int low,int high)

{

//任选元素作为轴，这里选首元素

int pivot = vec[low];

while(low < high)

{

while(low < high && vec[high] >= pivot)

high--;

vec[low] = vec[high];

while(low < high && vec[low] <= pivot)

low++;

vec[high] = vec[low];

}

//此时low==high

vec[low] = pivot;

return low;

}

//in-place partition algorithm

int in\_place\_partition(int\* array, int left, int right)

{

int pivot\_index = (right-left)/2;

int pivot = array[pivot\_index];

swap(array[pivot\_index], array[right]);

int index = left;

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

{

if (array[i] < pivot) //升序

{

swap(array[index], array[i]);

++index;

}

}

swap(array[right], array[index]);

return index;

}

void Qsort(int\* array, int left, int right)

{

if (left >= right)

return;

int index = Partition(array, left, right);

Qsort(array, left, index - 1);

Qsort(array, index + 1, right);

}

**3.选择排序**

选择排序（Selection Sort）的基本思想是，每一趟排序在n-i+1（i=1，2，3....，n-1）中选取关键字最小的记录作为有序序列的第i个记录。

(1)最为简单的是简单选择排序（Sample Selection sort）

//选择排序

void SelectSort(int a[],int n)

{

int temp;

int i,j;

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

{

int k = i;

for(j = i+1; j < n; j++)

if(a[j] < a[k]) k = j;

if(k != i)

{

temp = a[i];

a[i] = a[k];

a[k] = temp;

}

}

}

**4.归并排序**

归并排序（Merging sort），所谓归并，简单的讲，就是将两个有序的序列，合成一个新的有序表。我们用这个思想，可以把一个n个元素的序列，看成n个长度为1的子序列，然后利用归并排序，两两合并，然后的到了n/2个长度为2的子序列，再次进行合并，重复上面的步骤，直到合并为一个序列，则归并完成。归并排序的时间复杂度无论是在最好情况下还是在最坏情况下均是O(nlog2n)。

归并排序是稳定的排序算法

//将有序序列a[low..mid]和a[mid+1..high]归并到a[low..high]。

void Merge(int a[], int low, int mid, int high)

{

// 归并到b[]

int i = low;

int j = mid+1;

int k = 0;

int \*b = new int[high-low+1];

while(i <= mid && j <= high)

{

if (a[i] <= a[j]) { b[k++] = a[i++]; }

else { b[k++] = a[j++]; }

}

// 归并剩余元素

while(i <= mid) b[k++] = a[i++];

while(j <= high) b[k++] = a[j++];

// 从b[]复制回a[]

for(i = 0; i <= high-low; ++i)

a[low+i] = b[i];

delete []b;

}

//归并排序

//http://en.wikipedia.org/wiki/Mergesort

void MergeSort(int a[], int low, int high)

{

if(low >= high) return;

else

{

int mid = (low+high)/2;

MergeSort(a,low,mid);

MergeSort(a,mid+1,high);

Merge(a,low,mid,high);

}

}

//自底向上的归并排序

void MergeSort2(int a[], int n)

{

int s,i,t = 1;

while(t < n)

{

s = t; t = s\*2;

for(i=0; i+t<=n; i+=t)

Merge(a,i,i+s-1,i+t-1);

if(i+s < n)

Merge(a,i,i+s-1,n-1);

}

}

归并排序在最坏的情况下都是O(NlogN)的时间复杂度，缺点是Merge的时候要有O(N)的额外的空间，如何改进？

//reverse array

void reverse(int arr[], int size)

{

int left = 0;

int right = size -1;

while(left < right)

{

int temp = arr[left];

arr[left++] = arr[right];

arr[right--] = temp;

}

}

// swap [arr,arr+headSize) and [arr+headSize,arr+headSize+endSize)

void SwapMemory(int arr[], int headSize, int endSize)

{

reverse(arr, headSize);

reverse(arr + headSize, endSize);

reverse(arr, headSize + endSize);

}

//原地归并

void Inplace\_Merge(int arr[], int beg, int mid, int end)

{

int i = beg; // 指示有序串1

int j = mid + 1; // 指示有序串2

while(i < j && j <= end)

{

while(i < j && arr[i] <= arr[j])

{

++i;

}

int index = j;

while(j <= end && arr[j] <= arr[i])

{

++j;

}

SwapMemory(&arr[i], index-i, j-index);//swap [i,index) and [index,j)

i += (j-index);

}

}

//原地归并排序

void Inplace\_MergeSort(int arr[], int beg, int end)

{

if(beg < end)

{

int mid = (beg + end) / 2;

Inplace\_MergeSort(arr, beg, mid);

Inplace\_MergeSort(arr, mid+1, end);

Inplace\_Merge(arr, beg, mid, end);

}

}

**5.冒泡排序（Bubble Sort）**

冒泡排序方法是最简单的排序方法。这种方法的基本思想是，将待排序的元素看作是竖着排列的“气泡”，较小的元素比较轻，从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理，就是自底向上检查一遍这个序列，并时刻注意两个相邻的元素的顺序是否正确。如果发现两个相邻元素的顺序不对，即“轻”的元素在下面，就交换它们的位置。显然，处理一遍之后，“最轻”的元素就浮到了最高位置；处理二遍之后，“次轻”的元素就浮到了次高位置。在作第二遍处理时，由于最高位置上的元素已是“最轻”元素，所以不必检查。一般地，第i遍处理时，不必检查第i高位置以上的元素，因为经过前面i-1遍的处理，它们已正确地排好序。  
冒泡排序是稳定的。算法时间复杂度是O(n^2)。

//冒泡排序（相邻比较法）

void BubbleSort(int a[],int n)

{

int i,j;

int temp;

for(i = 0; i < n-1; i++)

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

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

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

//改进的冒泡排序

void ImprovedBubbleSort(int a[], int n)

{

int i,j;

int temp;

bool change = false;

for(i = 0; i < n-1; i++)

{

change = false;

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

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

{

temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

change = true;

}

if ( !change ) break;

}

}

//双向冒泡排序（鸡尾酒排序）

void CocktailSort(int a[], int n)

{

int top = n - 1;

int bottom = 0;

bool flag = true;

int i, j;

while(flag)

{

flag = false;

//从小到大，升序

for(i = bottom; i < top; i++)

{

if(a[i] > a[i+1])

{

swap(a[i], a[i+1]);

flag = true;

}

}

bottom++;

//从大到小，降序

for(j = top; j > bottom; j--)

{

if(a[j] < a[j-1])

{

swap(a[j], a[j-1]);

flag = true;

}

}

top--;

}

}

**6.希尔排序（Shell Sort）**

先将待排序列分割成若干个子序列，分别进行直接插入排序，基本有序后再对整个序列进行直接插入排序。  
希尔排序是不稳定的。时间复杂度大约为O(n^3/2)。

//希尔排序：shell排序的核心仍然使用插入排序

void ShellSort(int a[], int n)

{

int dk = n/2;

int i,j;

while(dk >= 1)

{

// 一趟希尔排序，对dk个序列分别进行插入排序

for(i = dk; i < n; ++i)

{

int x = a[i];

for (j = i-dk; j >= 0 && a[j] > x; j -= dk )

a[j+dk] = a[j];

a[j+dk] = x;

}

dk = dk/2; // 缩小增量

}

}

**7.堆排序（Heap Sort）**

堆排序是一种树形选择排序，在排序过程中，将A[n]看成是完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。对N个元素从小到大排序建立大根堆，然后交换堆顶与最后一个元素，将剩下的N-1个元素调整为大根堆，执行N-1此这样的操作。  
堆排序是不稳定的。算法时间复杂度O(nlogn)。

//调整大根堆

void HeapAdjust(int data[],int nStart, int nLen)

{

int nMaxChild = 0;

int Temp;

while((2\*nStart+1) < nLen)

{

nMaxChild = 2\*nStart+1;

if((2\*nStart+2) < nLen)

{

//比较左子树和右子树,记录最大值的Index

if (data[2\*nStart+1] < data[2\*nStart+2])

{

nMaxChild = 2\*nStart+2;

}

}

//change data

if(data[nStart] < data[nMaxChild])

{

//交换nStart与nMaxChild的数据

Temp = data[nStart];

data[nStart] = data[nMaxChild];

data[nMaxChild] = Temp;

//堆被破坏,需要重新调整

nStart = nMaxChild;

}

else

{

//比较左右孩子均小则堆未破坏,不再需要调整

break;

}

}

}

//堆排序 从小到大排序建立大顶堆

void HeapSort(int data[],int nLen)

{

int i;

int nTemp;

//建立堆

for(i = nLen/2-1; i >= 0; i--)

{

HeapAdjust(data, i, nLen);

}

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

{

//交换堆顶元素和最后一个元素

nTemp = data[0];

data[0] = data[i];

data[i] = nTemp;

//将data[0...i]重写建成大根堆

HeapAdjust(data, 0, i);

}

}

1. 详细设计
2. 调试分析
3. 用户手册
4. 测试结果
5. 附录