****

**算法分析与设计预习报告**

****

**学 院： 电子信息与人工智能学院**

**专业名称： 计算机科学与技术**

**班 级： 计算机203**

**学 号： 202007020625**

**姓 名： 於俊涛**

**任课老师： 齐 勇**

# 实验1 排序算法性能分析实验

班级 计算机203 学号 202007020625 姓名 於俊涛 角色 coder

1. **实验目的**
2. 掌握八种排序算法原理；
3. 掌握不同排序算法时间效率的经验分析方法，验证理论分析与经验分析的一致性；
4. **实验内容与方法**

排序问题要求我们按照升序排列给定列表中的数据项，目前为止，已有多种排序算法提出。本实验要求掌握选择排序、冒泡排序、归并排序、快速排序、插入排序、堆排序、基数排序、桶排序算法原理，并进行代码实现。通过对大量样本的测试结果，统计不同排序算法的时间效率与输入规模的关系，通过经验分析方法，展示不同排序算法的时间复杂度，并与理论分析的基本运算次数做比较，验证理论分析结论的正确性。

1. **实验步骤与过程**
2. 独立算法性能分析
3. 冒泡排序
4. 伪代码

BUBBLESORT(A)

for i = 1 to A.length-1

for j = A.length downto i + 1

if A[j] < A[j - 1]

exchange A[j] with A[j - 1]

1. 算法性能分析

①时间复杂度分析：

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数C和记录移动次数M均达到最小值：



所以，冒泡排序最好的时间复杂度为O(n)。

若初始文件是反序的，需要进行n-1趟排序。每趟排序要进行n-i次关键字的比较( 1 ≤ i ≤ n − 1 ) (1≤i≤n-1)(1≤i≤n−1)，且每次比较都必须移动记录三次来达到交换记录位置。在这种情况下，比较和移动次数均达到最大值：



冒泡排序的最坏时间复杂度为O( n2)。

综上，因此冒泡排序总的平均时间复杂度为O(n2)。

1. 选择排序
2. 伪代码

SELECTSORT(ele)

for i=0 to n-2

min=i

for j= i+1 to n-1

if ele[min]>ele[j] min=j

swap(ele[i],ele[min])

1. 算法性能分析

①时间复杂度分析：

选择排序的交换操作介于0与n − 1次之间。选择排序的比较操作为n(n − 1) /2次之间。选择排序的赋值操作介于0和3(n − 1)次之间。比较次数为O(n2)。比较次数与关键字的初始状态无关，总的比较次数N = ( n − 1 ) + ( n − 2 ) + . . . + 1 = n ( n − 1 ) / 2。交换次数O(n)，最好情况是已经有序，交换0次；最坏情况交换n − 1次，逆序交换n /2次。因此选择排序的时间复杂度为O(n2)。

1. 插入排序
2. 伪代码

INSERTION-SORT(A)

for j=2 to A.length:

key=A[j]

//将A[j]插入已排序序列A[1..j-1]

i=j-1

while i>0 and A[i]>key

A[i+1]= A[i]

i=i-1

A[i+1]=key

1. 算法性能分析

①时间复杂度分析

在插入下标为i的元素的时候（假设该元素为x），R0 ,R1 ,R3 ,RI-1已经有序了，那么x实际上有n + 1个备选位置可以插入，按照独立分布来说，每个位置的概率都是1/i+1，那么这n + 1个位置从左到右对应的比较次数为（i，i-1，...，1）。那么我们就可以得到一趟插入排序的平均时间复杂度为:



接下来，我们对n − 1趟结果进行求和，得到最终时间复杂度为：T(n)=O（n2）。

1. 快速排序
2. 伪代码

QUICKSORT(SeqList R,int low,int high)

int pivotpos;//划分后的基准记录的位置

if(low<high)

//仅当区间长度大于1时才需排序

pivotpos = Partition(R,low,high);

//对R[low...high]进行划分

QuickSort(R,low,pivotpos-1);

QuickSort(R,pivotpos+1,high);

1. 算法性能分析

①时间复杂度分析：

快速排序的一次划分算法从两头交替搜索，直到low和high重归，因此其时间复杂度是O(n)，而整个快速排序算法的时间复杂度与划分的趟数有关。最理想的情况是，每次划分所选择的中间数恰好将当前序列几乎等分，经过log n趟划分，便可得到长度为1的子表。这样，整个算法的时间复杂度为O(n log n)。

最坏的情况是，每次所选的中间数是当前序列中的最大或最小元素，这使得每次划分所得的子表中一个为空表，另一子表的长度为原表的长度-1。这样，长度为n的数据表的快速排序需要经过n趟划分，使得整个排序算法的时间复杂度为O(n2)。

综上快速排序的平均时间复杂度也是O(n log n)。

1. 归并排序
2. 伪代码

MERGE(sourceArr,tempArr,sIndex,midIndex,eIndex)

i = sIndex

j = midIndex+1

k = sIndex

//取出两个有序子序列中最小的一个元素，放入新的有序数列中

while i! = midIndex+1 and j!=eIndex+1

if sourceArr[i] < sourceArr[j]

tempArr[k] = sourceArr[i]

i++

else

tempArr[k] = sourceArr[j]

j++

k++

//前面的有序数列中还有元素

while i != midIndex+1

tempArr[k++] = sourceArr[i++]

//后面的有序数列还有元素

while j != eIndex+1

tempArr[k++] = sourceArr[j++]

//将有序数列拷贝给原数列

for m = sIndex to eIndex

sourceArr[m] = temp[m]

//归一化

MERGESORT(sourceArr,tempArr,sIndex,eIndex)

//类似二分查找一样，每次取半

if sIndex < eIndex

mid = sIndex + (eIndex - sIndex)/2

MERGESORT(sourceArr,tempArr,sIndex,mid)

MERGESORT(sourceArr,tempArr,mid+1,eIndex)

MERGE(sourceArr,tempArr,sIndex,mid,eIndex)

1. 算法性能分析

①时间复杂度分析：

不妨假设一个序列有n个数的排序时间为T(n)，T(n)是一个关于n的函数，随着n的变化而变化。那么我们将n个数的序列分为两个n /2的序列。则有：

T(n) = 2 ∗T(n/2) + t ( t为归并时间 )

由于归并时，两个子序列已经组内排好序了，那我们将两个排好序的序列组归成一个大的有序序列，只需要一个if循环即可。if循环中有n个数需要比较，所以时间复杂度为n。则有：

T(n) =2 ∗ T(n/2) + n

我们再将两个n /2的序列再分成4个n /4的序列。则有：

T(n/2) =2 ∗T(n/4) + n/2

代入并化简得：

T(n) = 4 ∗T(n/4) + 2 n

不难求得，整体时间消耗有如下表达式：

T(n)=n∗log2 n

综上，归并排序的时间复杂度为：O(n∗log n)

1. 基数排序
2. 伪代码

RADIXSORT(A,d)

for i = 1 to d

use a stable sort to sort array A on digit i

1. 算法性能分析

①时间复杂度分析：

基数排序的时间复杂度是O(k\*n)，其中n是排序元素个数，k是数字位数。k的大小取决于数字位的选择（比如比特位数）和待排序数据所属数据类型的全集的大小；k决定了进行多少轮处理，而n是每轮处理的操作数目。

以排序n个不同整数来举例，假定这些整数以B为底，这样每位数都有B个不同的数字，k= logBN,N是待排序数据类型全集的势。虽然有B个不同的数字，需要B个不同的桶，但在每一轮处理中，判断每个待排序数据项只需要一次计算确定对应数位的值，因此在每一轮处理的时候都需要平均n次操作来把整数放到归适的桶中去，所以就有：

k ≈ logBN

所以，基数排序的平均时间T就是：

T ≈ n ∗logBN

1. 堆排序
2. 伪代码

HEAP-SORT(A):

BUILD-MAX-HEAP(A); //构建最大堆

for i = A.length downto 2:

exchange A[i] and A[1];

A.heap-size = A.heap-size-1;

MAX-HEAPIFY(i);

构建大顶堆伪代码：

BUILD-MAX-HEAP(A):

A.heap-size = A.length;

//heap-size代表整个数组中在堆中的元素个数

for i = A.length/2 downto 1:

MAX-HEAPIFY(i)

维护大顶堆伪代码：

MAX-HEAPIFY(A, i): //维护堆性质的关键， 用于检测是否满足堆的性质

l = left(i);

r = right(i); //记录左右孩子的下标

if l <= A.heap-size and A[l] >= A[i]:

largest = l; //记录根节点和左右孩子中最大数的下标

else :

largest = r;

if r <= A.heap-size and r >= A[largest]:

largest = r;

if i != largest:

exchange A[i] and A[largest];

MAX-HEAPIFY (A, largest);

1. 算法性能分析

①时间复杂度分析：

  堆排序的排序过程主要分为构建初始堆和进行排序两部分，接下来分别进行时间复杂度分析：

a.初始化建堆：

  初始化建堆只需要对二叉树的非叶子节点调用调整堆函数，由下至上，由右至左选取非叶子节点来进行调整。那么倒数第二层的最右边的非叶子节点就是最后一个非叶子结点。

  假设高度为k kk，则从倒数第二层右边的节点开始，这一层的节点都要执行子节点比较然后交换（如果顺序是对的则无需交换）；倒数第三层则会选择其子节点进行比较和交换，如果没交换就可以不用再执行下去了。如果交换了，那么又要选择一支子树进行比较和交换；高层也是这样逐渐递归。那么总的时间计算为：

T = ( k − i )2i − 1

其中i表示第几层，2i − 1表示该层上元素个数，k − i表示子树上要下调比较的次数。则有：

T = 2k−k −1

又因为k为完全二叉树的深度，则有k=log n，把此式带入得：

T=n−（log n）−1

则堆排序初始堆的时间复杂度为：O(n)。

b. 排序重建堆

在取出堆顶点放到对应位置并把原堆的最后一个节点填充到堆顶点之后，需要对堆进行重建，只需对堆的顶点调用调整堆函数。每次重建意味着有一个节点出堆，所以需要将堆的容量减一。调整堆函数的时间复杂度k=log n，k为堆的层数。所以在每次重建时，随着堆的容量的减小，层数会下降，函数时间复杂度会变化。重建堆一共需要n−1次循环，每次循环的比较次数为log i，则有：

T=log2+log3+⋯⋯+log（n−1）+log n=log n!

又因为



则有：



即log n!与n log n是同阶函数；

则堆排序的时间复杂度为O(n log n)。

1. 桶排序
2. 伪代码

def BucketSort(nums):

buckets = [0] \* 10 # 建立桶

for i in nums:

buckets[i - 50] += 1 # 索引计算

for i in range(len(buckets)): # 输出排序后的值

while buckets[i] != 0:

print(i + 50)

buckets[i] -= 1

1. 算法性能分析

①时间复杂度

桶排序的时间复杂度和上面的计数排序是一样的,同样也是线性级别的,但是也是增加了桶的时间,所以也是O(n+k)。

②空间复杂度

上面我们已经说过了,桶排序本身也是一个通过空间来换取时间的算法,所以很明显他的空间复杂度就会很高.并且这个空间复杂度主要就取决于桶的数量以及桶的范围,所以假设有k个桶的话,那么空间复杂度就为O(n+k)。