Internal sorting

2019.11.23

[Summary 2](#_Toc25441397)

[1. Insert sorting 3](#_Toc25441398)

[2. Bubble sorting 4](#_Toc25441399)

[3. Selection sorting 5](#_Toc25441400)

[4. Shell sort 6](#_Toc25441401)

[5.Merge sort 7](#_Toc25441402)

[6.Quicksort 8](#_Toc25441403)

[7.heapsort 9](#_Toc25441404)

[8.Binsort and Radix sort 10](#_Toc25441405)

[Bucket sort 10](#_Toc25441406)

[Radix sort 10](#_Toc25441407)

[Proof of Lower Bounds for Sorting 12](#_Toc25441408)

Summary

Clue:

(1)How they perform

(2)Is it stable? if not ->how to change it?

(3)Time cost analysis

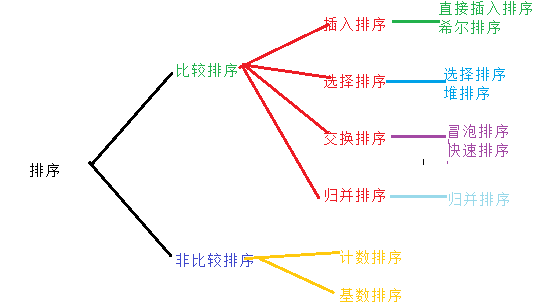
a)best-case cost

b)worst-case cost

c)average-case cost

how the above be calculated and under what situation

(4)Space cost





Stability

unstable选择排序、快速排序、希尔排序、堆排序【快、希、选、堆（“快些选一堆”）】

stable:冒泡排序、插入排序、归并排序、基数排序

1. Insert sorting

**（1）算法思想**：

插入排序，第i(i\_start=0)轮表示从第a[i+1]开始，把a[i+1]放到它前面序列中比它小和比它大的两个元素之间。

具体实现是从a[i+1]开始，该元素和它前面相邻的元素做大小比较，如果小于前面那个元素，就交换位置。

例如:原始序列：49、38、65、97、76、13、27、49

第0轮 **38** 49 {65、97、76、13、27、49}

第1轮 38 49 **65**

第2轮 38 49 65 **97**

第3轮 38 49 65 **76** 97

第4轮 **13** 38 49 65 76 97

第5轮 13 **27** 38 49 65 76 97

第6轮 13 27 38 49 **49** 65 76 97

**（2）Is it stable?**

A sorting algorithm is said to be stable if **it does not change the relative ordering of records with identical key values**. Many, but not all, of the sorting algorithms are stable, or can be made stable with minor changes.

交换位置iff该元素比前面元素小（相等是不交换位置的），因此是稳定排序

**（3）Time cost analysis**

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

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

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

else break;

}

简化之后：

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

for(int j=i;j>0&&(a[j]<a[j-1]);j--)

swap(a,j,j-1);

}

主要执行操作：元素大小比较。注意一旦a[j]>=a[j-1]就不用继续比较了，因为前面是排好序的。

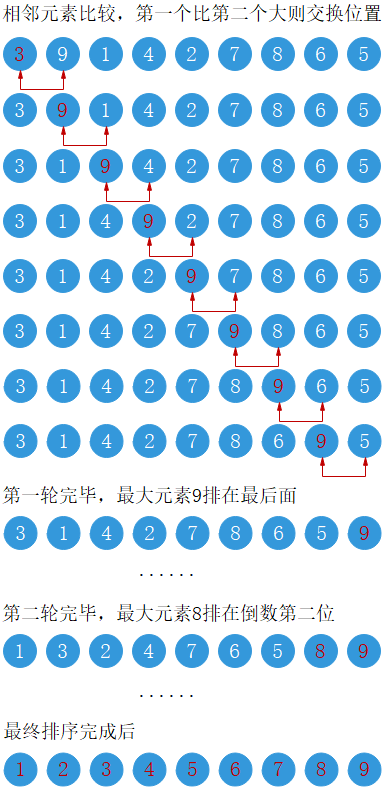
a)best-case cost : 元素已正序排好。这个时候，刚一进入内层循环就退出，因此执行次数（比较次数）是n-1次，因此时间复杂度是O(n).

b)worst-case cost: 元素是逆序的，每个元素都要移到数组最前端。这个时候，执行次数（比较次数）= 1+2+…+n-1 ≈ n2/2 = O(n2)

c)average-case cost:数组前i-1条元素中有一半的元素比第i条大，即时间代价为最坏情况的一半，即n2/4 = O(n2)

**（4）Space cost**

需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)

2. Bubble sorting  
**（1）算法思想**：  
  a）比较相邻的元素。如果第一个比第二个大，就交换他们两个。  
  b） 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。  
  c）针对所有的元素重复以上的步骤，除了最后一个。  
  d）持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要交换。

举例：见右图

**（2）Is it stable?**

很显然，我们不会对两个相同的值进行交换，因此冒泡排序是稳定的

**（3）Time cost analysis**

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

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

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

swap(a,j,j+1);

主要执行操作：元素大小比较。注意不管序列是怎样，都是要比较n(n-1)/2 次。因此best-case cost = worst-case cost=average-case cost= O(n2)

**（4）Space cost**

需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)

3. Selection sorting

**（1）算法思想**：选择排序，从头至尾扫描序列，找出最小的一个元素，和第一个元素交换，接着从剩下的元素中继续这种选择和交换方式，最终得到一个有序序列。

例如:原始序列：49、38、65、97、76、13、27、49

a）在进行选择排序过程中分成有序和无序两个部分，开始都是无序序列

结果：49、38、65、97、76、13、27、49

b）从无序序列中取出最小的元素13，将13**同无序序列第一个元素交换**，此时产生仅含一个元素的有序序列，无序序列减一

结果：{13、} {38、65、97、76、49、27、49}

c）从无序序列中取出最小的元素27，将27同无序序列第一个元素交换，此时产生仅两个元素的有序序列，无序序列减一

结果：{13、27、} {65、97、76、49、38、49}

…..

结果：{13、27、38、49、49、65、76、97}

**（2）Is it stable?**

举另外一个例子，序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

The new low value is set only if it is actually less than the previous one, but the direction of the search is from the bottom of the array.

The algorithm will be stable if “less than” in the check becomes **“less than or equal to”** for selecting the low key position

**（3）Time cost analysis**

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

swap(a,i,mini\_num(a,i,n)); //由下面函数可以看出每次要比较n-i次

int mini\_num(int data[],int s,int n){ //找出a[s]到a[n]区间最小元素下标

int min = 9999999; int index;

for(int i=s;i<n;i++)

if(data[i]<min) {

min = data[i];

index = i;

}

return index;

}

主要执行操作：元素大小比较。注意不管序列是怎样，都是要比较=n(n-1)/2 次。因此best-case cost = worst-case cost=average-case cost= O(n2)

**（4）Space cost**

需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)

4. Shell sort

**（1）算法思想**：希尔排序，也称递减增量排序算法(diminishing increment sort) ，是插入排序的一种更高效的改进版本。定义一个间隔序列来表示排序过程中进行比较的元素之间有多远的间隔，每次**将具有相同间隔的数分为一组，进行插入排序**，大部分场景中，间隔是可以提前定义好的，也可以动态生成。希尔排序的实质就是**分组的插入排序**。



**（2）Is it stable?**

相同的值可能被分到不同的组里面  把相对顺序打乱，因此希尔排序不稳定。

The sublist sorts are done independently, and it is quite possible to swap an element in one sublist ahead of its equal value in another sublist. Once they are in the same sublist, they will retain this (incorrect) relationship.

**（3）Time cost analysis**

主要执行操作：元素大小比较。时间复杂度与选取的增量有关。

{1,2,4,8,...}这种序列并不是很好的增量序列，使用这个增量序列的时间复杂度（最坏情形）是O(n²)

Hibbard提出了另一个增量序列{1,3,7，...,2^k-1}，这种序列的时间复杂度(最坏情形)为O(n^1.5)，

Sedgewick提出了几种增量序列，其最坏情形运行时间为O（n^1.3）,其中最好的一个序列是{1,5,19,41,109,...}

**（4）Space cost**

需要一个临时变量用来交换数组内数据位置，所以空间复杂度为O(1)

5.Merge sort

**（1）算法思想**：分而治之。（整体分两个过程：分解，合并）

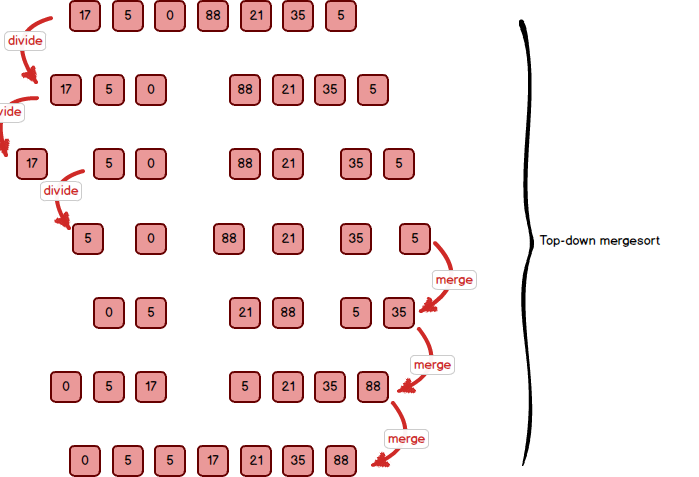
递归法原理如下（假设序列共有n个元素）：

将原始序列从中间分为左、右两个子序列，此时序列数为2

将左序列和右序列再分别从中间分为左、右两个子序列，此时序列数为4

重复以上步骤，直到每个子序列都**只有一个元素**，可认为每一个子序列都是有序的

最后依次进行归并操作，直到序列数变为1



**（2）Is it stable?**

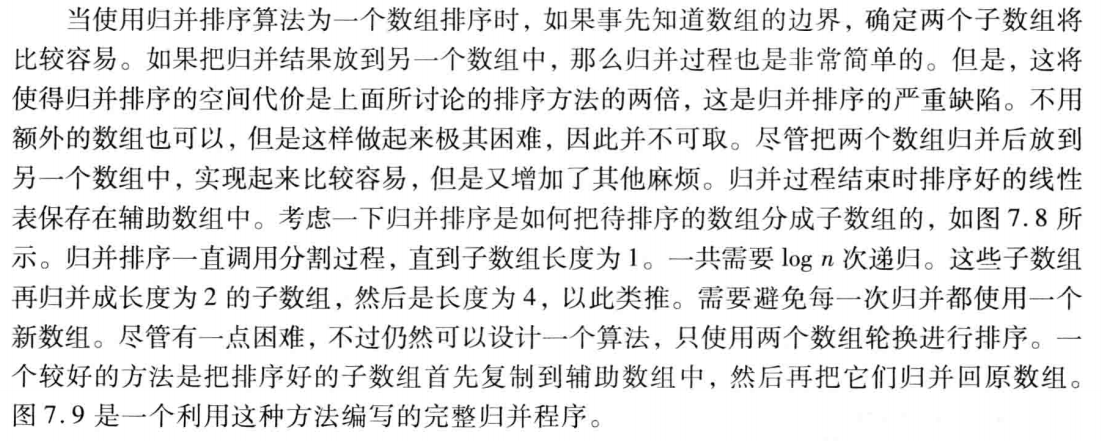
在合并过程中，如果左边序列中的元素和右边序列中元素相等，我们把处在前面的序列的元素保存在结果序列的前面（先拿左边的，再拿右边的），因此是稳定的。

**（3）Time cost analysis**

the depth of the recursion is **log n** for n elements. The first level of recursion can be thought of as working on one array of size n, the next level working on two arrays of size n=2, the next on four arrays of size n=4, and so on. The bottom of the recursion has n arrays of size 1. Thus, n arrays of size 1 are merged (requiring Θ(n) total steps), n=2 arrays of size 2 (again requiring Θ(n) total steps), n=4 arrays of size 4, and so on. **At each of the log n levels of recursion, Θ(n) work is done, for a total cost of Θ(n log n)**.

This cost **is unaffected by the relative order of the values** being sorted, thus this analysis holds for the best, average, and worst cases.

**（4）Space cost**



总结：需要一个临时数组储存归并的结果，空间复杂度为O(n)，时间复杂度为O(nlogn) 。可以将空间复杂度由 O(n) 降低至 O(1)，然而相对的时间复杂度则由 O(nlogn) 升至 O(n²)

6.Quicksort

**（1）算法思想**：（可参考《啊哈算法》）  
快速排序有两个方向，左边的i下标一直往右走，当a[i] <= a[pivot]，其中pivot是中枢元素的数组下标，一般取为数组第0个元素。而右边的j下标一直往左走，当a[j] > a[pivot]。如果i和j都走不动了，i <= j，交换a[i]和a[j],重复上面的过程，直到i > j。 交换a[j]和a[pivot]，完成一趟快速排序。

**（2）Is it stable?**

不稳定：在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为**5** 3 3 4 **3** 8 9 10 11，现在中枢元素5和3（第5个元素，下标从1开始计）交换就会把元素3的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和a[j] 交换的时刻。

**（3）Time cost analysis**

主要操作是：寻找pilot（常数复杂度，一般选第一个元素或中间那个元素）+把元素划分到pilot的左右边（需要进行元素间大小比较）

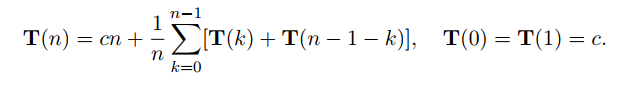
a)best-case cost :

Quicksort’s best case occurs when **findpivot** always breaks the array into two equal halves. Quicksort repeatedly splits the array into smaller partitions. In the best case, the result will be log *n* levels of partitions, with the top level having one array of size *n*, the second level two arrays of size *n=*2, the next with four arrays of size *n=*4, and so on. Thus, at each level, all partition steps for that level do a total of *n* work, for an overall cost of *n* log *n* work when Quicksort finds perfect pivots.

b)average-case cost:

We make one reasonable simplifying assumption: At each partition step, the pivot is equally likely to end in any position in the (sorted) array. In other words, the pivot is equally likely to break an array into partitions of sizes 0 and n−1, or 1 and n−2,and so on.

Given this assumption, the average-case cost is computed from the following equation:



This equation says that there is one chance in n that the pivot breaks the array into subarrays of size 0 and n − 1, one chance in n that the pivot breaks the array into subarrays of size 1 and n−2, and so on. The expression “T(k) +T(n−1−k)” is the cost for **the two recursive calls to Quicksort on two arrays of size k and n−1−k**. The initial cn term is the cost of **doing the findpivot and partition steps**, for some constant c. The closed-form solution to this recurrence relation is Θ(n log n).

Thus, Quicksort has average-case cost Θ(n log n)

c)worst-case cost

pivot值没选好，划分使得一遍只有一个元素，另一边有n-1和元素，因此没戏规模只减少了一个元素，此时总时间代价为O(n2)

**（4）Space cost**

快速排序的空间复杂度可以理解为递归的深度，而递归的实现依靠栈，平均需要递归logn次，所以平均空间复杂度为O(logn)

7.heapsort

**（1）算法思想**：

a)把要排序的数组转化为最大堆（建堆）

b)选取堆顶元素，把它“删去”，并把数组中最后一个元素放到第一个位置

（相当于把第一个位置与最后一个位置的元素交换位置）

c)重新建堆（下滤过程）

**（2）Is it stable?**

不稳定。我们知道堆的结构是节点i的孩子为2 \* i和2 \* i + 1节点，大顶堆要求父节点大于等于其2个子节点，小顶堆要求父节点小于等于其2个子节点。在一个长为n 的序列，堆排序的过程是从第n / 2开始和其子节点共3个值选择最大（大顶堆）或者最小（小顶堆），这3个元素之间的选择当然不会破坏稳定性。但当为n / 2 - 1， n / 2 - 2， … 1这些个父节点选择元素时，就会破坏稳定性。有可能第n / 2个父节点交换把后面一个元素交换过去了，而第n / 2 - 1个父节点把后面一个相同的元素没 有交换，那么这2个相同的元素之间的稳定性就被破坏了。

**（3）Time cost analysis**

时间消耗主要在初始化堆过程和每次选取最大数后重新建堆的过程，初始化建堆时的时间复杂度为O(n)，更改堆元素后重建堆的时间复杂度为O(nlogn) （如果要删除第k大的元素并重新建堆，则需要O(n+klogn)），所以堆排序的平均、最好、最坏时间复杂度都为O(nlogn)

**（4）Space cost**

堆排序是就地排序，空间复杂度为常数O(1)

8.Binsort and Radix sort

### Bucket sort

**（1）算法思想：**将待排序元素划分到不同的桶。将元素均匀的分在一个区间[a,b]上，在[a,b]之间放置一定数量的桶

**a)**先扫描一遍序列求出最大值 maxV 和最小值 minV ，设桶的个数为 k ，则把区间 [minV, maxV] 均匀划分成 k 个区间，每个区间就是一个桶。将序列中的元素分配到各自的桶。

b)对每个桶内的元素进行排序。可以选择任意一种排序算法。

c)将各个桶中的元素合并成一个大的有序序列。

**（2）Is it stable?**

Yue. Equal values that come later are appended to the list.

**（3）Time cost analysis &（4）Space cost**

假设数据是均匀分布的，则每个桶的元素平均个数为 n/k 。假设选择用快速排序对每个桶内的元素进行排序，

那么每次排序的时间复杂度为 O(n/klog(n/k)) 。

总的时间复杂度为 O(n)+O(m)O(n/klog(n/k)) = O(n+nlog(n/k)) = O(n+nlogn-nlogk) 。

当 k 接近于 n 时，桶排序的时间复杂度就可以金斯认为是 O(n) 的。

即桶越多，时间效率就越高，而桶越多，空间就越大。

### Radix sort

**（1）算法思想：**

a) 找到位数最大的整数的位数max

b)将所有待排序整数（注意，必须是非负整数）统一为位数相同（max）的整数，位数较少的前面补零。

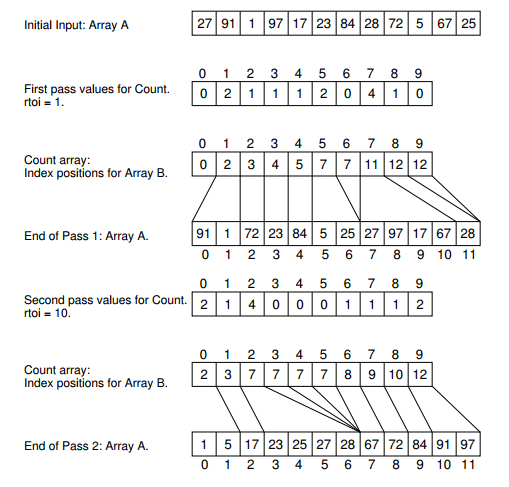
c) 从最低位开始，依次进行一次**稳定排序**。这样从最低位一直到最高位排序完成以后，整个序列就变成了一个有序序列。

注意：同一数位的排序子程序要用稳定排序，因为稳定排序能将上一次排序的成果保留下来。

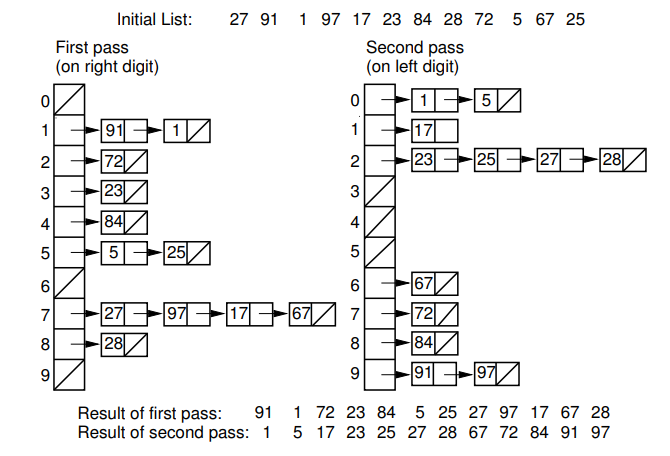
举例，有一个整数序列，0, 123, 45, 386, 106，下面是排序过程：

第一次排序，个位，000 123 045 386 106 第二次排序，十位，000 106 123 045 386

第三次排序，百位，000 045 106 123 386 最终结果，0, 45, 106, 123, 386, 排序完成。



linked-list based



**（2）Is it stable?**

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序，最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以其是稳定的排序算法。

While the processing is from bottom to top, the bins are also filled from bottom to top, preserving relative order.

**（3）Time cost analysis &（4）Space cost**

基数排序对于 n 个记录，执行一次分配和收集的时间为O(n+r)，如果关键字有 d 位，则要执行 d 遍，所以总的时间复杂度为 O(d(n+r))。该算法的空间复杂度就是在分配元素时，使用的桶空间，空间复杂度为O(r+n)=O(n)

此外，还介绍计数排序(Counting Sort)算法，并将三种排序算法比较如下：

计数排序是一种O(n)的排序算法，其思路是开一个长度为 maxValue-minValue+1 的数组，然后分配。扫描一遍原始数组，以当前值- minValue 作为下标，将该下标的计数器增1。收集。扫描一遍计数器数组，按顺序把收集值。

举个例子， nums=[2, 1, 3, 1, 5] , 首先扫描一遍获取最小值和最大值，

maxValue=5 , minValue=1 ，于是开一个长度为5的计数器数组 counter ，

1. 分配。统计每个元素出现的频率，得到 counter=[2, 1, 1, 0, 1] ，如 counter[0] 表示值 0+minValue=1 出现了2次。

2. 收集。 counter[0]=2 表示 1 出现了两次，那就向原始数组写入两个1，

3. counter[1]=1 表示 2 出现了1次，那就向原始数组写入一个2，依次类推，最终原始数组变为 [1,1,2,3,5]

计数排序本质上是一种特殊的桶排序，当桶的个数最大的时候，就是计数排序。



Proof of Lower Bounds for Sorting

An analysis for the cost of a *problem* as opposed to an *algorithm*. The upper bound for a problem can be defined as the **asymptotic cost of the fastest known algorithm**. **The lower bound defines the best possible efficiency for *any* algorithm that solves the problem,** including algorithms not yet invented.

A simple estimate for a problem’s lower bound can be obtained by measuring the size of the input that must be read and the output that must be written. **Certainly no algorithm can be more efficient than the problem’s I/O time.** From this we see that **the sorting problem cannot be solved by *any* algorithm in less than Ω(*n*) time** because it takes at least *n* steps to read and write the *n* values to be sorted.

This section presents one of the most important and most useful proofs in computer science: **No sorting algorithm based on key comparisons can possibly be faster than Ω(*n* log *n*) in the worst case.** This proof is important for three reasons.

First, knowing that widely used sorting algorithms are asymptotically（渐进地） optimal is reassuring（使人安心的）. Second, this proof is one of the few non-trivial lower-bounds proofs that we have for any problem; that is, this proof provides one of the relatively few instances where our lower bound is tighter than simply measuring the size of the input and output. As such, it provides a useful model for proving lower bounds on other problems. Finally, knowing a lower bound for sorting gives us a lower bound in turn for other problems whose solution could be used as the basis for a sorting algorithm. **The process of deriving asymptotic bounds for one problem from the asymptotic bounds of another is called a reduction.**

**Except for the Radix Sort and Binsort, all of the sorting algorithms presented in this chapter make decisions based on the direct comparison of two key values**. For example, Insertion Sort sequentially compares the value to be inserted into the sorted list until a comparison against the next value in the list fails. In contrast, Radix Sort has no direct comparison of key values. All decisions are based on the value of specific digits in the key value, so it is possible to take approaches to sorting that do not involve key comparisons. Of course, **Radix Sort in the end does not provide a more efficient sorting algorithm than comparison-based sorting. Thus, empirical evidence suggests that comparison-based sorting is a good approach.**

The proof that any comparison sort requires Ω(*n* log *n*) comparisons in the worst case is structured as follows. First, comparison-based decisions can be modeled as the branches in a tree. This means that any sorting algorithm based on comparisons between records can **be viewed as a binary tree whose nodes correspond to the comparisons, and whose branches correspond to the possible outcomes**. Next, the minimum number of leaves in the resulting tree is shown to be the factorial of *n*. Finally, **the minimum depth of a tree with *n*! leaves is shown to be in Ω(*n* log *n*)**. Before presenting the proof of an Ω(*n* log *n*) lower bound for sorting, we first must define the concept of a **decision tree**. **A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions.** Each (binary) decision is represented by a branch in the tree. For the purpose of modeling sorting algorithms, we count all comparisons of key values as decisions. If two keys are compared and the first is less than the second, then this is modeled as a left branch in the decision tree. In the case where the first value is greater than the second, the algorithm takes the right branch.

**Any sorting algorithm based on comparisons can be modeled by a decision tree in this way, regardless of the size of the input.** Thus, all sorting algorithms can be viewed as algorithms to **“find” the correct permutation of the input that yields a sorted list**. Each algorithm based on comparisons can be viewed as proceeding by making branches in the tree based on the results of key comparisons, and each algorithm can terminate once a node with a single permutation has been reached. How is the worst-case cost of an algorithm expressed by the decision tree? The decision tree shows the decisions made by an algorithm for all possible inputs of a given size. Each path through the tree from the root to a leaf is one possible series of decisions taken by the algorithm. **The depth of the deepest node represents the longest series of decisions required by the algorithm to reach an answer.**

Here are some important facts worth remembering:  
**• A binary tree of height *n* can store at most 2*n -* 1 nodes.  
• Equivalently, a tree with *n* nodes requires at least *d*log(*n* + 1)*e* levels.**

What is the minimum number of nodes that must be in the decision tree for any comparison-based sorting algorithm for *n* values? Because sorting algorithms are in the business of determining which unique permutation of the input corresponds to the sorted list, the decision tree for any sorting algorithm must contain at least one leaf node for each possible permutation. There are ***n*! permutations for a set of *n* numbers** (see Section 2.2). Because **there are at least *n*! nodes in the tree, we know that the tree must have Ω(log *n*!) levels**. From Stirling’s approximation (Section 2.2), we know **log *n*! is in Ω(*n* log *n*)**. **The decision tree for any comparison-based sorting algorithm must have nodes Ω(*n* log *n*) levels deep.** Thus, **in the worst case, any such sorting algorithm must require Ω(*n* log *n*) comparisons**. Any sorting algorithm requiring Ω(*n* log *n*) comparisons in the worst case requires Ω(*n* log *n*) running time in the worst case. Because any sorting algorithm requires Ω(*n* log *n*) running time, the problem of sorting also requires Ω(*n* log *n*) time. We already know of sorting algorithms with O(*n* log *n*) running time, so we can conclude that the problem of sorting requires Θ(*n* log *n*) time.