算法基础笔记

上官凝

2020 年 11 月 6 日

目录

算法书的：

1. 答案 1
2. 答案 2
3. cnblogs 的一些文章合集

第 1 章　基础知识

第 1.1 节　循环不变式

循环不变式与插入排序的正确性：必须证明三条性质

1. 【初始化】循环的第一次迭代之前，它为真
2. 【保持】如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真
3. 【终止】在循环终止时，不变式为我们提供一个有用的性质，该性质有助于证明算法是正确的

当循环是 for 循环时，在第一次开始迭代之前，我们将检查循环不变式的时刻是在对循环计数变量对初始赋值后，在循环头对第一次测试之前。

第 1.2 节　时间复杂度

Formula 1.2.1. 渐进紧确界

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 \in \mathcal{R}^+, s.t. \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Formula 1.2.2. 渐进上界

$$O(g(n)) = \{f(n) \mid \exists c, n_0 \in \mathcal{N}^+, s.t. \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

Formula 1.2.3. 渐进下界

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 \in \mathcal{N}^+, s.t. \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

第 2 章　以比较为基础的排序算法

先引入几个评判标准的概念：

1. 稳定性：(Stability) 假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变。[1]
2. 时间复杂度
3. 就地排序：只有常数个数的元素被存储在数组外面

以下六种方法，前三种（插入排序、选择排序和冒泡排序）是比较基础的排序算法，而后面三种（希尔排序、堆排序和快排）是效率比较高的。

第 2.1 节　插入排序

插入排序 (Insertion-Sort) 的总体思想是保持一个已经排序完成的 DONE 序列，类比摸扑克牌并将其顺序置入手牌中 时间复杂度：最好 $O(n)$，平均和最差都是 $O(n^2)$，就地排序，稳定排

Algorithm 1: Insertion-Sort(A)

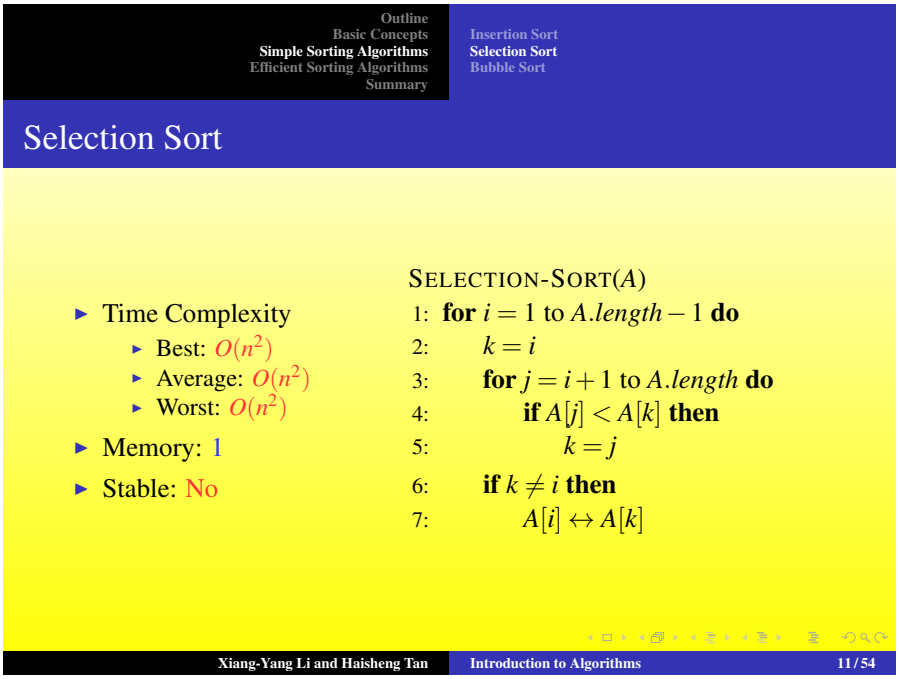
```
1 forall j = 2toA.length do
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1..j−1].
4     i = j − 1
5     while i > 0andA[i] > key do
6         A[i + 1] = A[i]
7         i = i − 1
8     A[i + 1] = key
```

序

[1] https://baike.baidu.com/item/排序算法稳定性

第 2.2 节 选择排序

与插入排序相反，选择排序 (SELECTION-SORT) 不再遍历已排序数组，而是遍历未排序数组，选出最小的元素，并将其放入新的数组里面。时间稳定性三个均为 $O(n^2)$，是就地排序但不是稳定排序



第 2.3 节 冒泡排序

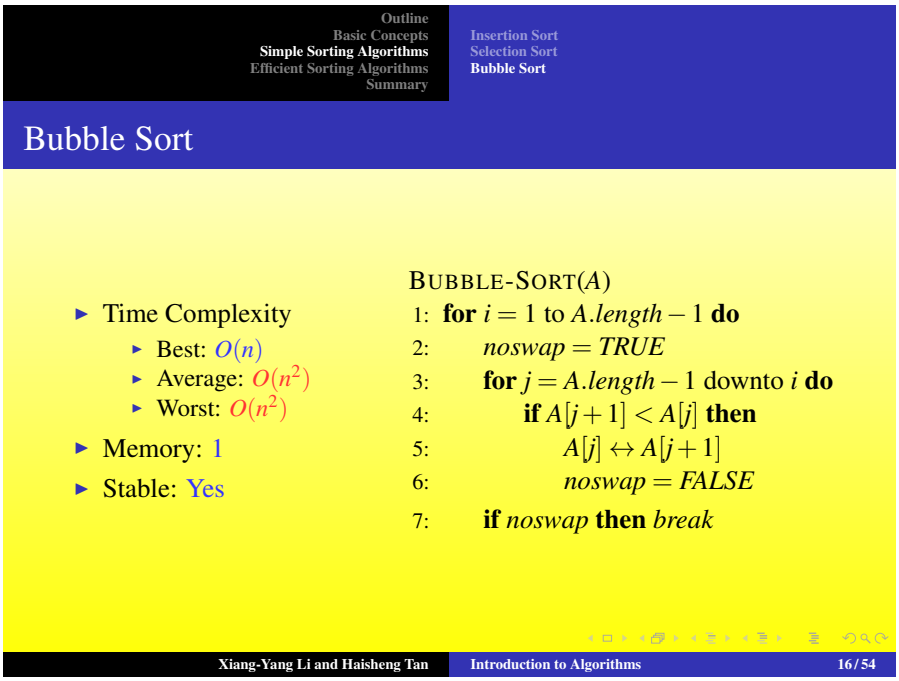
冒泡排序 (BUBBLE-SORT) 是两重循环比较相邻两个数据，将较大/小者下沉/上浮

第 2.4 节 希尔排序

希尔排序 (Shell-Sort) 是一种“少吃多餐”的插入排序算法，他需要选取一组递减的模长，在每一轮排序中，对同一“同余类”中的数据进行插入排序。有一个参考链接。希尔排序的时间复杂度依赖于间隔序列，属于就地排序，不是稳定排序。

为什么需要用到希尔排序呢?

1. 尽管插入排序的最坏情况时间复杂度是 $O(n^2)$，但是在数据总量较小时可以和 $O(n)$ 相接近
2. 当对一个较大的数组进行排序的时候，如果用一个足够大的排序间隔，每一个 sub-array 就会比较小
3. 在很多轮循环之后，尽管 gap 很小了，但是大部分元素也已经排好序了（所以是一个比较好的情况）



那么，如何选择模长呢？

1. $\left\lceil \frac{n}{2^k} \right\rceil$：时间复杂度 $\Theta(n^2)$
2. $2\left\lceil \frac{n}{2^k+1} \right\rceil + 1$：时间复杂度 $\Theta(n^{3/2})$
3. $2^k - 1$：时间复杂度 $\Theta(n^{3/2})$
4. $2^k + 1$ $(k \geq 0)$：时间复杂度 $\Theta(n^{3/2})$
5. 连续的形如 $2^p 3^q$ $(p, q$为素数$)$ 的 gap：时间复杂度 $\Theta(n \lg^2 n)$

第 2.5 节 堆排序

堆排序 (HEAP-SORT) 是一直维护一个二叉堆。然后重复选择排序，只是换做用二叉堆来找到最小/大元素（最大堆的 $A[1]$ 永远是最大的元素。注意到堆有一个很好的性质：with the array representation of an n-element heap, the leaves are the nodes indexed from $\lceil A.length/2 \rceil + 1$ to n, and each leaf is a 1-element max-heap to begin with. 时间复杂度：最大堆化 $O(\log n)$，建堆 $O(n)$；三个总复杂度都是 $O(n \log n)$。就地排序但不是稳定排序

2.5.1 优先队列

优先队列是一种抽象数据类型，或者说一种需求（如进程调度中的优先级分层调度）。优先队列是一种用来维护由一组元素构成的集合 S 的数据结构，其中的每一个元素都有一个相关的值，称为关键字 key。优先队列中的每个元素都有优先级，而优先级高（或者低）的将会先出队，而优先级相同的则按照其在优先队列中的顺序依次出队。优先队列一般用二叉堆实现（PPT 用的最大堆）

BUILD-MAX-HEAP(A)
1: $A.heap\text{-}size = A.length$
2: **for** $i = \lfloor A.length/2 \rfloor$ downto 1 **do**
3: MAX-HEAPIFY(A, i)

General idea: Same as selection sort, maintain the minimum (maximum) element by using heap.
MAX-HEAP: $A[1]$ always stores the largest number.

HEAPSORT(A)
1: BUILD-MAX-HEAP(A)
2: **for** $i = A.length$ downto 2 **do**
3: $A[1] \leftrightarrow A[i]$
4: $A.heap\text{-}size = A.heap\text{-}size - 1$
5: MAX-HEAPIFY($A, 1$)

Maintaining the Heap Property

Assumption: sub-trees rooted at LEFT(i) & RIGHT(i) are max-heaps.
MAX-HEAPIFY(A, i) // Input an an array and an index i
1: $l = \text{LEFT}(i)$;
2: $r = \text{RIGHT}(i)$
3: **if** $l \leq A.heap\text{-}size$ and $A[l] > A[i]$ **then**
4: $largest = l$
5: **else** $largest = i$
6: **if** $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$ **then**
7: $largest = r$
8: **if** $largest \neq i$ **then**
9: $A[i] \leftrightarrow A[largest]$
10: MAX-HEAPIFY($A, largest$)

第 2.6 节 快速排序

很多好的算法都是递归的。快排 (QUICK-SORT) 使用了分治思想。选择一个基准点，作为左右两个子数组的分界标准，然后原问题分解为两个较小的子问题，而由于左数组 $A[p..q-1]$ 的元素均小于 $A[q]$，而右数组则相反，故合并过程是平凡的。需要引起注意的是，快速排序的时间

Quicksort

General idea:
- **Arbitrarily choose** an element x in the unsorted set for comparison.
- **Divide** the unsorted elements into two parts: $\leq x$ and $> x$.
- **Recursively** use QUICKSORT for the above two parts.

QUICKSORT(A, p, r)
1: **if** $p < r$ **then**
2: $q = \text{PARTITION}(A, p, r)$
3: QUICKSORT($A, p, q-1$)
4: QUICKSORT($A, q+1, r$)

Partition

PARTITION(A, p, r)
1: $x = A[r]$ // pivot element
2: $i = p - 1$
3: **for** $j = p$ to $r - 1$ **do**
4: **if** $A[j] \leq x$ **then**
5: $i = i + 1$
6: $A[i] \leftrightarrow A[j]$
7: $A[i+1] \leftrightarrow A[r]$
8: **return** $i + 1$

依赖于划分是否平衡，而平衡与否又依赖于用于划分的元素。如果划分是平衡的，那么性能可以和归并排序一样，如果不是平衡的，那么其性能就接近于插入排序了。如果在算法的每一层递归上，划分都是最大程度不平衡的，那么算法的时间复杂度为 $\Theta(n^2)$（这个结论并不依赖于原始数据的整齐程度）。每一次都是平衡的划分，那么他的时间复杂度为 $\Theta(n \lg n)$。快排的平均时间复杂度更接近于其最好情况，也是 $O(n \lg n)$

第 3 章　线性时间排序

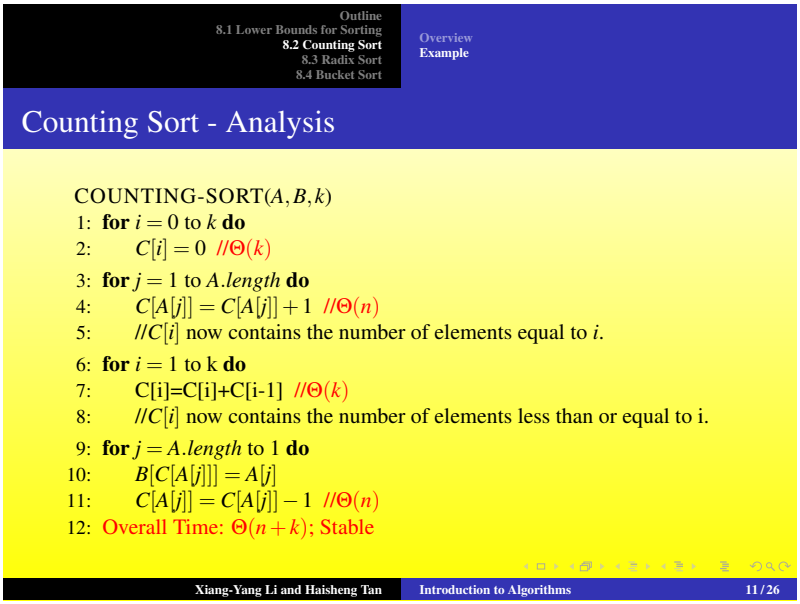
所有比较排序算法都有一个共同特点：排出来的顺序只依赖于对于输入元素之间的比较，他们的时间复杂度是 Ωn 的。比较排序算法可以被抽象地视作（二叉）决策树。

在不同对排序算法之间进行选择对时候，不只是要考虑其时间复杂度，还有一些其他因素。如 (characteristics of the implementations)、底层结构 (the underlying machine) 以及输入数据 (the input data)

第 3.1 节　计数排序

计数排序 (Counting-Sort) 对整数进行排序，它使用了一个额外的用于计数的数组 C，其长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上 1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。所以说，计数排序适用于数据量较小的情况。但是，计数排序可以用在基数排序算法中，能够更有效的排序数据范围很大的数组。

计数排序的基本思想是对于数据组中的某一个数据 A，统计出所有数据中比他小的数据个数为 m，则他就排在第 $m+1$ 个位置。具体操作是开一个宽度为可能出现的值域宽的统计数组，在遍历原始数据的过程中统计出来所有可能出现的值各自出现了多少次。



第 3.2 节 基数排序

基数排序 (RADIX-SORT) 是将整数按照位数切割成不同的数字，然后每个位数进行比较。它存在的前提是每一轮排序都是 stable 的，这样对低位进行的排序保证了在对高位进行排序时，高位相同的数据低位是有序的。收藏一个参考文章

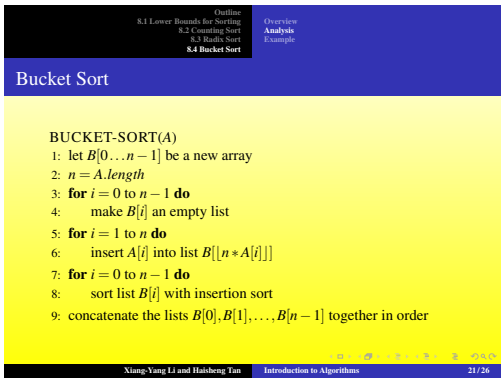
Algorithm 2: RADIX-SORT (A,d)

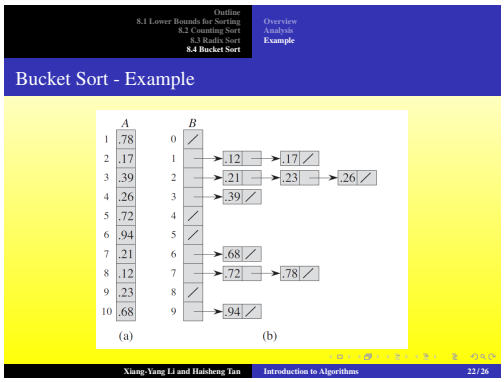
Input: A is the to-sort array, while d is the 数据位数

```
1 for i = 1 to d do
2     use a stable sort to sort array A on digit i
```

第 3.3 节 桶排序

桶排序 (BUCKET-SORT) 作用于一组服从在 $[0,1)$ 区间上均匀分布的数据。基本思路是将这个 $[0,1)$ 区间分割成几个等长的小区块，称为“桶”可以证明，桶排序的时间复杂度为 $\Theta(n)$。且即使





数据不服从均匀分布，桶排序也有可能在线性时间内完成，只要输入数据满足以下的性质：所有桶大小的平方和与总元素数呈线性关系

第 4 章　中位数和顺序统计量

本章将讨论一个由 n 个互异 (distinct) 元素构成的集合中选择第 i 个顺序统计量 ($X_{(i)}$) 的问题

第 4.1 节　最小值和最大值

显然，想要得到一组数据的 min 和 max，最少需要 $O(n)$ 的时间（因为所有数据总要遍历一遍），同时由于比较 $n-1$ 次总能出结果，故最好的次数就是 $n-1$。若想要同时找出最小值和最大值，那么就有一种好的算法，只需要最多 $3\lceil n/2 \rceil$ 次：对输入的一对元素相互进行比较，然后把较小的与当前最小值比较，较大的与当前最大值进行比较。

第 4.2 节　期望为线性时间的选择算法

如果想要找到这个集合中的 $X_{(k)}$，那么采取一个类似于快排的分治策略：取一个分割点，如果这个点的排序小于 k，那么就在高位集合递归，反之在低位集合