山东师范大学 SHANDONG NORMAL UNIVERSITY

《算法设计与分析》
第3.2讲 基于比较的排序算法

山东师范大学信息科学与工程学院
段会川
2014年9月

目录

冒泡排序—算法描述

1. 比较相邻的元素。如果第一个比第二个大，就交换。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。
   - 第一遍后，最后的元素将是最大的元素。
3. 针对除最后排序好的元素重复以上的步骤。
   - 每一遍都必定排好当前最后的元素。
4. 持续对越来越少的前面未排好序的元素重复上面步骤，直到只剩一个元素。

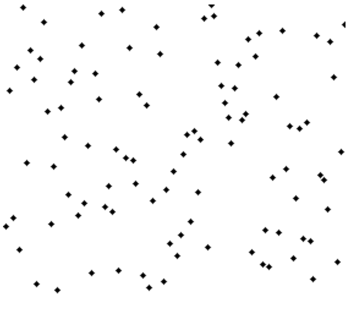
冒泡排序算法—伪代码

- **算法名称：冒泡排序 BubbleSort**
- **输入：n个数的数组a**
- **输出：排好序的a**

最好: O(n)

```
for i=n down to 2
    done = true
    for j=1 to i-1
        if a[j] > a[j+1] then
            swap(a[j], a[j+1])
            done = false
        end if
    end for //j
    if done then break
end for //i
```


冒泡排序算法—演示



冒泡排序算法—复杂度分析

- 基本操作是比较运算
- 最好情况(best case)
  - 元素已经是有序的，外循环只需执行第1轮，运行时间：
    $T_{best} = n - 1 = O(n)$
- 最坏情况(worst case)
  - 元素是倒序的，各次内循环均不中途结束：
    $T_{worst} = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2} = O(n^2)$
- 平均复杂度：$O(n^2)$

目录

- ☐ 冒泡排序
- ☐ 归并排序
- ☐ 插入排序
- ☐ 二叉树
- ☐ 二叉堆与堆排序
- ☐ 优先队列
- ☐ 快速排序
- ☐ 排序算法的复杂度上界
- ☐ 排序算法复杂度总结
- ☐ 矩阵乘法

归并排序(merge sort, 合并排序)—描述

- ☐ 归并排序由冯·诺伊曼于1945年提出，是一个典型的分治算法
  - ■ 分解：将排序数组分解为两个等大的子数组
  - ■ 解决：递归
  - ■ 合并：将两个已经有序的数组合并为一个有序的数组
    - ☐ 大小分别为m和n的有序数组可以在线性时间$O(m+n)$内合并

归并排序—算法伪代码

- ☐ 算法名称：归并排序(MergeSort)
- ☐ 输入：n个元素的数组a
- ☐ 输出：排序的a
- ☐ 1: MergeSort(a, low, high, b)
- ☐ 2: if high-low<=1 then return
- ☐ 3: mid = (low+high)/2 分解
- ☐ 4: MergeSort(a, low, mid, b)
- ☐ 5: MergeSort(a, mid+1, high, b) 解决
- ☐ 6: Merge(a, low, mid, high, b) 合并
- ☐ 7: copy(b, low, high, a)

P59

归并排序—归并算法伪代码

- ☐ 算法名称：合并两个有序数组(Merge)
- ☐ 输入：数组a，low到mid及mid+1到high间已排序
- ☐ 输出：数组a，从low到high是排序的
- ☐ 1: i0 = low, i1 = high i_0 i_1
- ☐ 2: for j = low to high
- ☐ 3: if i0 < mid and (i1 >= high or A[i0] <= A[i1]))
- ☐ 4: B[j] = A[i0]
- ☐ 5: i0 = i0 + 1
- ☐ 6: else
- ☐ 7: B[j] = A[i1]
- ☐ 8: i1 = i1 + 1
- ☐ 9: end if

归并排序—复杂度分析

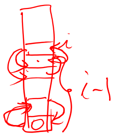
- ☐ 显然归并排序计算复杂度的递推式为
  - ■ $T(n) = 2T(n/2) + O(n)$
  - ■ 运用主定理
    $a = 2, b = 2, d = 1 = \log_b a = 1.$
    因而，$T(n) = O(n^d \log n) = O(n\log n)$
  - ■ 归并排序的合并阶段需要一个规模为n的数组，因而空间复杂度为$O(n)$
  - ■ 归并排序是基于比较的排序算法中的一个最优算法，因为可以证明基于比较的排序算法最少需要$\Omega(n\log n)$的复杂度

目录

- ☐ 冒泡排序
- ☐ 归并排序
- ☐ 插入排序
- ☐ 二叉树
- ☐ 二叉堆与堆排序
- ☐ 优先队列
- ☐ 快速排序
- ☐ 排序算法的复杂度上界
- ☐ 排序算法复杂度总结
- ☐ 矩阵乘法

插入排序算法—伪代码

1. 从第一个元素开始，该元素作为单个元素序列显然是已经排序好的
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描
3. 如果该元素（已排序）大于新元素，将该元素移到后一位置
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5



插入排序算法—伪代码

- 算法名称：插入排序 InsertionSort
- 输入：n个数的数组a
- 输出：排好序的a

```
for i=2 to n
    x = a[i]; j=i-1
    while j >= 1
        if a[j] > x then
            a[j+1] = a[j]
        else break
        j = j-1
    end while
    a[j+1] = x
end for //i
```


插入排序算法—演示

6 5 3 1 8 7 2 4

插入排序算法—复杂度分析

- 基本操作是元素的比较(或移动)
- 最好情况(best case)
  - 元素已经是有序的，每次内循环只执行1次比较，运行时间：

$$T_{best} = n - 1 = O(n)$$

- 最坏情况(worst case)
  - 元素是倒序的，各次内循环均将当前元素插入到位置1上：

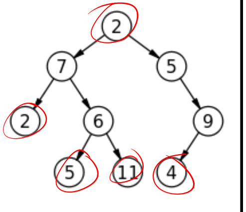
$$T_{worst} = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$$

- 平均复杂度：$O(n^2)$

目录

二叉树(binary tree)的基本概念

- 二叉树定义
  - 二叉树是一种树形结构，它的每一个结点（node）最多只有两颗有序的子树，即左子树和右子树。
  - 含有子树的结点称为父结点(parentsnode)，有父结点的结点称为子结点(childrennode)。
  - 无父结点的结点(只有一个)称为根结点(rootnode)，无子结点的结点成为叶结点(leafnode)。



二叉树(binary tree)的基本概念

- 结点的深度**d(depth)**
  - 是指从根结点到该结点的路径长度，因而根结点的深度为**0**。
  - 同一深度上结点的集合称为树的一个层**(level)**。
  - 深度(或层)**d**上的结点总数$\le 2^d$。
- 树的高度**h(height)**
  - 指的是从其根结点到最深的结点的路径长度，也就是树的最大深度，因而只有一个根结点的树的高度为**0**。

段数

二叉树(binary tree)的基本概念

- 满二叉树**(full binary tree, also proper binary tree, 2-tree, strictly binary tree)**:
  - 除叶结点外各结点均有两个子结点的二叉树。
- 完美二叉树**(perfect binary tree)**:
  - 最后的叶结点层**h**上的结点数达到最大值 2^h 的完全二叉树，它是完全二叉树的极端情况。
  - 完美二叉树每一层**d**上的结点数都达到最大值2^d。
  - 高度为**h**的完美二叉树的结点总数为:

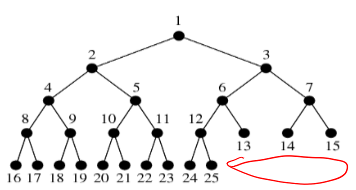
$$\sum_{d=0}^{h} 2^d = 2^{h+1} - 1$$

$2\times2^h-1$

2^h+2^h-1

二叉树(binary tree)的基本概念

- 完全二叉树**(complete binary tree, almost perfect binary tree,** 几乎完美的二叉树**)**:
  - 叶结点只在最后两层，倒数第**2**层的结点数达到最大且最后一层的叶结点连续排列在左侧的二叉树。



二叉树(binary tree)的基本概念

- 高度为**h**的完全二叉树的结点总数n为:
  - $2^h \le n \le 2^{h+1} - 1$，或$2^h \le n < 2^{h+1}$，即$h \le \log n < h + 1$。
  - 即结点数为**n**的完全二叉树的高度为: $h = \lfloor \log n \rfloor$。
- 结点的顺序编号
  - 对于有**n**个结点的完全二叉树，可以对其各结点按从上到下、从左到右的顺序进行编号，规定根结点的编号为**1**，则显然编号范围是**1≤i≤n**，其中**n**是结点总数。第**i**结点如果存在子结点，则左右子结点的编号分别是**2i**和**2i+1**，如果**i**结点不是根结点，则其父结点的编号为$\lfloor i/2 \rfloor$。
  - 因此完全二叉树可以用一个简单的数组来存储。

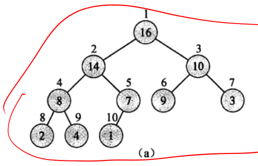
目录

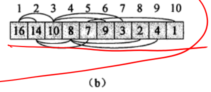
堆(heap)—基本概念

- 堆是计算机科学中一类特殊数据结构的统称。堆通常是一个可以被看做一棵树的数组对象。堆总是满足下列性质:
  - 堆中某个结点的值总是大于或小于其父结点的值;
  - 堆总是一颗完全树。
- 将根结点中的值最大的堆叫做最大堆或大根堆，根结点中的值最小的堆叫做最小堆或小根堆。
- 常见的堆有二叉堆、斐波那契堆等。

二叉堆

- 二叉堆是一棵完全二叉树，它满足如下堆特性：
  - 父结点的键值总是大于等于（二叉最大堆），或小于等于（二叉最小堆）任何一个子结点的键值，且每个结点的左子树和右子树都是一个二叉堆（都是最大堆或最小堆）。
  - 可以使用线性表存储。

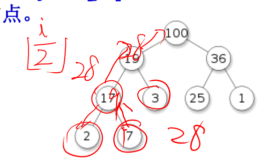




导论3版(中)，图6.1，P84

二叉最大堆的基本操作—结点上移(SiftUp)

- 基本思路
  - 假定对于某个i>1,结点H[i]变成了键值大于它父结点键值的元素，这样就违反了堆的特性。修复此堆的算法称为上移操作(SiftUp)。
  - SiftUp操作沿着从H[i]到根结点的惟一一条路径进行，不断将路径上结点H[j]的键值与其父结点H[⌊j/2⌋]的键值进行比较，如果H[j]>H[⌊j/2⌋]，则将其交换，直到H[j]≤H[⌊j/2⌋]或j是根结点。



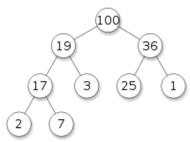
二叉最大堆的基本操作—SiftUp算法

算法名称：二叉堆SiftUp算法

输入：数组H[1..n]，除i元素外其他均满足堆的特性。

输出：上移H[i]（如果需要），直至不大于父结点。

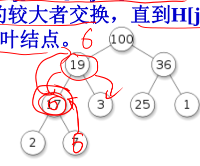
```
done←false
if i=1 then exit
repeat
    if key(H[i]) > key(H[⌊i/2⌋]) then
        H[i] ↔ H[⌊i/2⌋]
    else done←true
    i ←⌊i/2⌋
until i=1 or done
```



由于堆的高度为⌊logn⌋，SiftUp算法的复杂度为O(logn)。

二叉最大堆的基本操作—结点下移(SiftDown)

- 基本思路
  - 假定对于某个i>1,H[i]变成了键值小于它子结点键值的元素，这样就违反了堆的特性。修复此堆的算法称为下移操作(SiftDown)。
  - SiftDown操作沿着从H[i]到叶结点的惟一一条路径，不断将H[j]的键值与其子结点的键值H[2*j]和H[2*j+1]进行比较，如果H[j]小于H[2*j]和H[2*j+1]二者之一，则将其与H[2*j]和H[2*j+1]中的较大者交换，直到H[j]不小于H[2*j]和H[2*j+1]或j变成了叶结点。



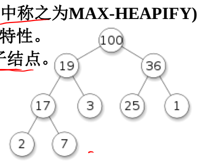
二叉最大堆的基本操作—SiftDown算法

算法名称：二叉堆SiftDown算法(《算法导论》中称之为MAX-HEAPIFY)

输入：数组H[1..n]，除i元素外其他均满足堆的特性。

输出：下移H[i]（如果需要），直至它不小于子结点。

```
done←false
if 2*i>n then exit
repeat
    i←2*i
    if i+1≤n and key(H[i+1] > key(H[i]) then i←i+1
    if key(H[⌊i/2⌋])< key(H[i]) then H[i] ↔ H[⌊i/2⌋]
    else done←true
until 2*i>n or done
```

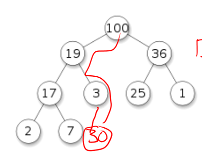


由于堆的高度为⌊logn⌋，SiftDown算法的复杂度为O(logn)。

二叉最大堆的基本操作—插入(Insert)

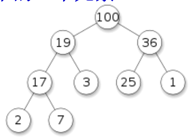
- 堆的插入算法用于实现给堆增加一个元素x，使其仍然满足堆的特性。
- 基本思路：
  - 将新元素加到堆的最后，然后使用上移算法SiftUp将其移动到合适的位置上。
- 可以使用插入算法将给定数组中的元素建立在堆数据结构中。

n n+1



二叉最大堆的基本操作—插入(Insert)

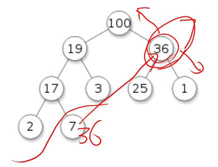
- 算法名称：二叉堆Insert的伪代码
  - 输入：堆H[1…n]和元素x。
  - 输出：新的堆H[1…n+1]，x是其中的一个元素。
  - 1：n←n+1
  - 2：H[n]←x
  - 3：SiftUp(H, n)



- 显然，Insert算法的复杂度与SiftUp的复杂度相同，即O(logn)。

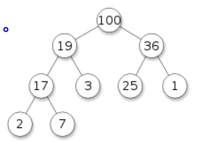
二叉最大堆的基本操作—删除(Delete)

- 堆的删除操作用于删除堆中某个编号的元素，但删除后要保证堆的特性仍能得到保持。
  - 删除操作应返回删除的结点值。
- 基本思路：
  - 将待删除的元素与最后一个元素进行交换，将堆的大小减1，然后再使用SiftUp或SiftDown算法对堆进行调整。



二叉最大堆的基本操作—删除(Delete)

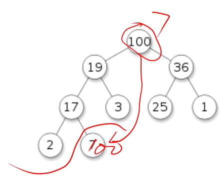
- 算法名称：二叉堆删除(Delete)算法伪代码
  - 输入：堆H[1…n]和1到n之间的i。
  - 输出：去掉i元素的新的堆H[1…n-1] 。
  - 1：x←H[i]; y←H[n]
  - 2：n←n-1
  - 3：if i=n+1 then exit
  - 4：H[i] ←y
  - 5：if key(y)≥key(x) then SiftUp(H,i)
  - 6：else SiftDown(H,i)



- 显然，Delete算法的复杂度与SiftUp或SiftDown的复杂度相同，即O(logn)。

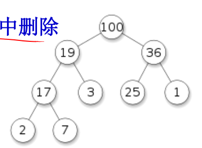
二叉最大堆的基本操作—删除最大值

- 删除最大值操作用来删除非空堆H中的最大值并保持堆的特性，同时返回最大键值结点中的数据项。
- 基本思路：
  - 堆的最大值是其根结点，删除最大值就是删除堆中的1号元素，因此只要用参数1调用算法Delete就可以了。



二叉最大堆的基本操作—删除最大值

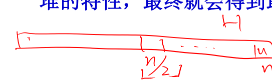
- 算法名称：删除最大值算法DeleteMax
  - 输入：堆H[1…n]
  - 输出：返回最大键值元素并将其从堆中删除
  - 1：x←H[1];
  - 2：Delete(H,1)
  - 3：return x

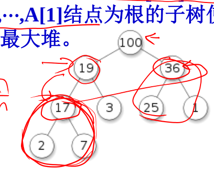


- 显然DeleteMax算法的复杂度就是Delete的复杂度，即O(logn)。

二叉最大堆的基本操作—创建堆

- 目的：将任意排列的n个元素的数组组织成一个堆。
- 基本思路：运用堆的特性和基本操作
  - 令H[1…n]为一个二叉堆，则A[[n/2]+1],A[[n/2]+2],…,A[n]对应于相应树的叶结点，因为最小编号的父结点就是最后一个元素n对应的父结点，其编号应为[n/2]。编号大于[n/2]的结点都应该是叶结点。这样，从A[[n/2]]开始依次调整以A[[n/2]],A[[n/2]-1],…,A[1]结点为根的子树使其满足堆的特性，最终就会得到最大堆。





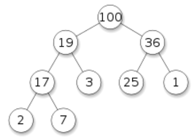
O(n)

二叉最大堆的基本操作—创建堆

- 算法名称：创建堆算法**MakeHeap**

输入：n个元素的数组A[1…n]。
输出：以A[1…n]构成的堆。

```
1:  for i←⌊n/2⌋ downto 1
2:     SiftDown(A,i)
3:  end for
```



二叉最大堆的基本操作—创建堆算法的复杂度

- 创建堆算法MakeHeap的运算时间可理解如下
  - 算法对$i = h - 1$到0的层上的结点进行循环处理
  - 第i层的深度为$h - i$，因而其上执行SiftDown算法的复杂度为$O(h - i)$
  - 第i层上共有2^i个结点
  - 因此第i层的总运算时间为$T(i) = 2^i O(h - i)$
  - 因而MakeHeap的运算时间为 → 令 $j = h - i$

$T(n) = \sum_{i=h-1}^{0} 2^i O(h-i) = \sum_{j=1}^{h} 2^{h-j} O(j)$

$= O\left(2^h \sum_{j=1}^{h} \frac{j}{2^j}\right) = O\left(2^{\lfloor \log n \rfloor} \sum_{j=1}^{h} \frac{j}{2^j}\right)$

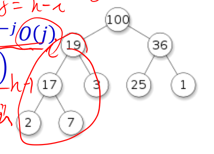
$= O\left(n \sum_{j=1}^{h} \frac{j}{2^j}\right)$.

h-i

0

log n

h-1



二叉最大堆的基本操作—创建堆的复杂度

- $T(n) = O\left(n \sum_{j=1}^{h} \frac{j}{2^j}\right)$的计算
  - 当$|x| \neq 1$时，$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$
  - 当$|x| < 1$时，$\sum_{i=0}^{\infty} x^i = \lim_{n\to\infty} \frac{x^{n+1}-1}{x-1} = \frac{1}{1-x}$
  - 等式两边对x求导，得$\sum_{i=0}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$
  - 两边同乘以x得，$\sum_{i=0}^{\infty} i x^i = \frac{x}{(1-x)^2}$
  - 因而$T(n) = O\left(n \sum_{j=1}^{\infty} j \left(\frac{1}{2}\right)^j\right) = O(2n) = O(n)$

堆排序算法

- 基本思想
  - 先用MakeHeap算法将输入的数组创建为一个堆。
  - 交换堆中的第一个元素和最后一个元素，则新的最后一个元素即是已经排好序的元素。
  - 将堆的大小减1，并对第1个元素执行SiftDown操作。
  - 继续这一过程，当堆的大小减到1时，即完成了对数组的排序。

堆排序算法

- 算法名称：堆排序**HeapSort**

输入：n个元素的数组A[1…n]
输出：以非降序排列的数组A

```
1:  MAKEHEAP(A)
2:  for i=n downto 2
3:     A[1] ↔ A[i]
4:     SiftDown(A[1…i-1],1)
5:  end for
```

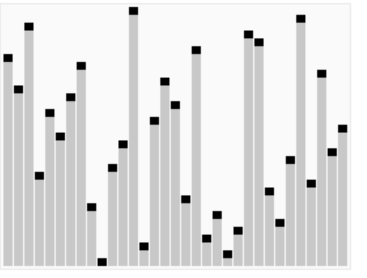

堆排序算法

- 时间复杂度
  - 算法中MakeHeap的复杂度为O(n)，循环要进行n-1次，每次的复杂度为O(logn)，因此HeapSort的时间复杂度为O(nlogn)。
- 空间复杂度
  - 算法需要存储n个元素的数组，这需要O(n)的复杂度
  - SiftDown需要一个辅助存储单元实现可能的父结点和子结点间的交换，这带来O(1)的空间开销

堆排序算法—演示

6 5 3 1 8 7 2 4

堆排序算法—演示



目录

- 冒泡排序
- 归并排序
- 插入排序
- 二叉树
- 二叉堆与堆排序
- 优先队列
- 快速排序
- 排序算法的复杂度上界
- 排序算法复杂度总结
- 矩阵乘法

优先队列

- **In computer science/data structures, a priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.**
- **While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.**

Wikipedia

优先队列

- 优先队列(**priority queue**)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**)或键。最大优先队列支持以下操作：
  - **INSERT(S, x)**：把元素x插入集合S中，等价于 $S = S \cup \{x\}$
  - **MAXIMUM(S)**：返回S中具有最大键值的元素
  - **EXTRACT-MAX(S)**：去掉并返回S中的具有最大键值的元素
  - **INCREASE-KEY(S, x, k)**：将元素x的键值增加到k，k≥x
- 实现：常用堆来实现，但也可使用数组或链表等实现，但效率较低
- 应用：作业调度、**Huffman**编码、**Dijkstra**算法

目录

- 冒泡排序
- 归并排序
- 插入排序
- 二叉树
- 二叉堆与堆排序
- 优先队列
- 快速排序
- 排序算法的复杂度上界
- 排序算法复杂度总结
- 矩阵乘法

快速排序算法

- 算法描述
  - 使用分治法策略把待排序数据序列分为两个子序列，步骤为：
  1. 挑出一个元素，称为"基准"(pivot)元素。枢轴
  2. 分区(partition)操作：将所有比基准值小的元素放在基准前面，所有比基准值大的元素放在其后面(相等的元素可放到任一边)。
  3. 将小于基准值的子序列和大于基准值的子序列对步骤2递归。
     - 当序列长度变为0或1时结束递归。

快速排序算法

- 算法名称：快速排序QuickSort

输入：n个元素的数组q
输出：排序的q

```
quicksort(q)
if length(q) ≤ 1 then return q
else
   select a pivot from q
   for each x in q except the pivot
      if x < pivot then add x to less
      if x ≥ pivot then add x to greater
   return concat(quicksort(less), pivot, quicksort(greater))
end if
```


快速排序算法

- 时间复杂度
  - 最好情况：$O(n\log n)$
  - 最坏情况：$O(n^2)$
  - 平均情况：$O(n\log n)$
- 空间复杂度
  - 朴素实现：需要额外的$O(n)$辅助空间
  - Sedgewick实现(1978)：需要额外的$O(\log n)$辅助空间

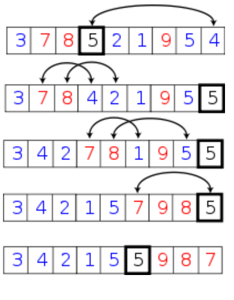
快速排序算法

- 发明人
  - Sir Charles Antony Richard Hoare (Tony Hoare或C. A. R. Hoare, 霍尔)
  - 1960年，26岁
  - 1980年图灵奖获得者
    - 程序设计语言的定义与设计方面的基础性贡献

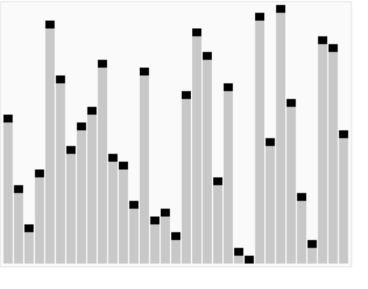
Sir Charles Antony Richard Hoare
1934.1.11

快速排序算法—演示

6 5 3 1 8 7 2 4



快速排序算法—演示



目录

排序算法的复杂度上界

- □ 任意待排序的n个数的数字序列共有n!中可能的组合状态
  - 如任意的3个数1,2,3共有6种状态

$x_1, \cdots x_n$

输入

1，2，3
1，3，2
2，3，1
2，1，3
3，1，2
3，2，1

$n!$ 输出

排序算法的复杂度上界

- □ 一个排序算法要对任意初始状态的n个数进行排序，就必须能够将其n!种可能的初始组合都能正确地排序
  - 基于比较的排序算法，执行一次基本的比较操作会将被排序的数据分成两种情况
  - 如果一个基于比较的排序算法在$f(n)$步内完成比较，则它最多可以区分$2^{f(n)}$种可能的数据情况，而它要具备对n个数进行排序的能力，则必有：$2^{f(n)} \geq n!$，即$f(n) \geq \log(n!)$

$\Theta(n\log n)$　$f(n)$　$2^{f(n)}$

log(n!)的复杂度

- $n! < n^n \Rightarrow \log(n!) < n\log n \Rightarrow \log(n!) = O(n\log n).$
- $n! = 1 \cdot 2 \cdot \cdots \cdot \left\lceil\frac{n}{2}\right\rceil \cdot \underbrace{\left(\left\lceil\frac{n}{2}\right\rceil + 1\right) \cdot \cdots \cdot (n-1) \cdot n}_{n/2项},$

$$n! > \underbrace{\left(\left\lceil\frac{n}{2}\right\rceil + 1\right) \cdot \cdots \cdot (n-1) \cdot n}_{n/2项} > \left(\frac{n}{2}\right)^{\frac{n}{2}},$$

$$\log(n!) > \frac{n}{2}\log\left(\frac{n}{2}\right) = \frac{1}{2}(n\log n - n),$$

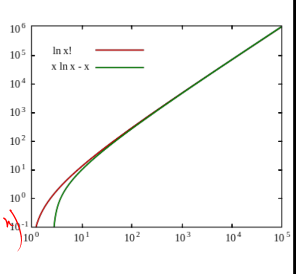
$$\lim_{n\to\infty}\frac{n\log n}{\frac{1}{2}(n\log n - n)} = 2 < \infty \therefore \log(n!) = \Omega(n\log n).$$

因此：$\log(n!) = \Theta(n\log n)$

n!的近似式—斯特灵近似公式

- □ **Stirling近似公式是*n*很大时*n*!的近似式**
  - **James Stirling，苏格兰数学家(1692—1770)**
  - $\lim_{n\to\infty}\frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$
  - $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
  - $\ln n! \sim n\ln n - n + \frac{1}{2}\ln n + \frac{1}{2}\ln(2\pi).$
  - $\ln n! = n\ln n - n - O(\ln n)$

$\log(n!) = \Theta(n\log n)$



排序算法的复杂度上界

- □ 因此，基于比较的排序算法在最坏情况下至少需要$\log(n!) = \Theta(n\log n)$的复杂度才能完成对n个数的排序
  - 也就是说，基于比较的排序算法在最坏情况下的复杂度不能低于$\Theta(n\log n)$
  - 或者说，基于比较的排序的最优算法的最坏复杂度为$\Theta(n\log n)$
  - 归并排序的复杂度为$O(n\log n)$，因而它是最优的
  - 堆排序的复杂度为$O(n\log n)$，因而它也是最优的

目录

排序算法复杂度总结

算法名称	选择法	冒泡法	插入排序	快速排序	合并排序	堆排序
最坏情况复杂度	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(n\log n)$
平均情况复杂度	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
最好情况复杂度	$O(n^2)$	$O(n)$	$O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
空间复杂度	$O(1)$	$O(1)$	$O(1)$	$O(n)$ $O(\log n)$	$O(n)$	$O(1)$

目录

矩阵乘法—一般方法 $O(n^2)$

Kara~~stu~~tsuba $O(n^{\log 3}) \doteq O(n^{1.59})$

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} (AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\ (AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np} \end{pmatrix}$$

一般矩阵乘法要产生n^2个元素，而每个元素需要n次乘法，因而复杂度为n^3

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}.$$

矩阵乘法—分块方法

- 将两个待乘的矩阵分别分解为4个大小为$n/2 \times n/2$的子矩阵

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

  - 分块后需要进行8次$n/2 \times n/2$子矩阵乘法和4次$n/2 \times n/2$子矩阵的加法，而加法的复杂度为$O(n^2)$
  - 因而复杂度递推公式为：$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$

递推式的通解—主定理

- 主定理(master theorem)
  - 如果对于常数$a > 0, b > 1, d \geq 0$, 有
    $T(n) = aT(\lceil n/b \rceil) + O(n^d)$
  - 则有：$T(n) = \begin{cases} O(n^d) & 当 d > \log_b a \\ O(n^d \log n) & 当 d = \log_b a \\ O(n^{\log_b a}) & 当 d < \log_b a \end{cases}$
- 当分治法每次将问题分解为a个规模为n/b的子问题，而每个子问题可以在$O(n^d)$时间内求解时，显然该分治法的运算时间的递推式便是
  - $T(n) = aT(\lceil n/b \rceil) + O(n^d)$
  - 因而可以套用主定理求解

矩阵乘法—分块方法复杂度

- 复杂度递推式
  - $T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$
  - $a = 8, b = 2, d = 2 < \log_b a = 3$
  - 所以：$T(n) = O\left(n^{\log_b a}\right) = O(n^3)$

矩阵乘法—Strassen算法

- 德国数学家Volker Strassen于1969年提出了一个快速的矩阵乘法算法

$$\mathbf{C} = \mathbf{AB} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

$$\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in R^{2^{n-1} \times 2^{n-1}}$$

$$\mathbf{C}_{1,1} = \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \qquad \mathbf{C}_{1,2} = \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2}$$

$$\mathbf{C}_{2,1} = \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \qquad \mathbf{C}_{2,2} = \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2}$$

矩阵乘法—Strassen算法

- 德国数学家Volker Strassen于1969年提出了一个快速的矩阵乘法算法

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

矩阵乘法—Strassen算法

- 德国数学家Volker Strassen于1969年提出了一个快速的矩阵乘法算法

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\
P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\
P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\
P_4 &= D(G - E)
\end{aligned}$$

P67

  - 该算法包括7次规模为$\frac{n}{2}$的矩阵乘法和18次加法

矩阵乘法—Strassen算法

- Strassen算法复杂度的递推式
  - $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$
- 运用主定理
  - $a = 7, b = 2, d = 2 < \log_b a = \log_2 7 \approx 2.81$
  - 所以：$T(n) = O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$

目录