算法概论

第三讲：排序算法

薛健

Last Modified: 2018.12.9

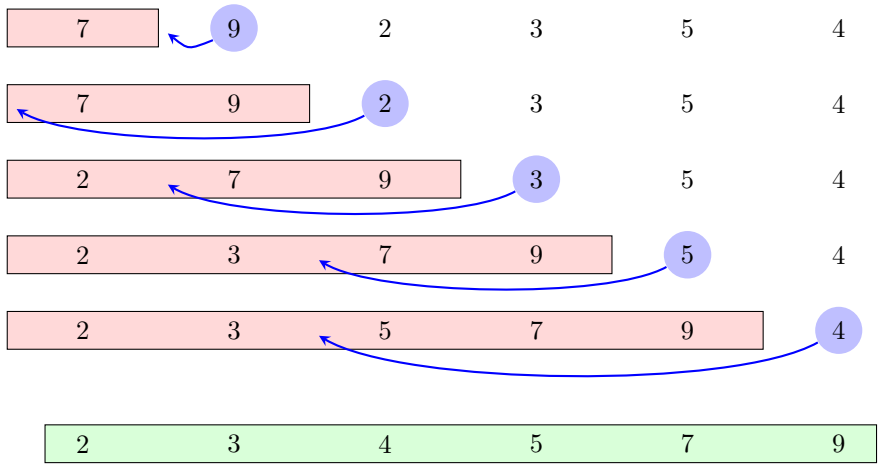
主要内容

1 简单排序算法

1.1 插入排序

基本策略

- 依次将序列元素插入到已排好序的部分：



算法

Algorithm `InsertionSort`$(E[], n)$

```
1 for xindex ← 1 to n − 1 do
2 |   current ← E[xindex];
3 |   xloc ← ShiftVac(E, xindex, current);
4 |   E[xLoc] ← current;
5 end
```

ShiftVac

Algorithm `ShiftVac`$(E[], xindex, x)$

```
1  vacant ← xindex;
2  xloc ← 0;
3  while vacant > 0 do
4  |   if E[vacant − 1] ≤ x then
5  |   |   xloc ← vacant;
6  |   |   break;
7  |   end
8  |   E[vacant] ← E[vacant − 1];
9  |   vacant ← vacant − 1;
10 end
11 return xloc;
```

复杂度分析

- **基本操作**：元素比较
- Worst-Case：
  - $W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$
- Average-Case：
  - 不妨假设序列元素各不相同
  - 对于 `ShiftVac`，将当前第 i 个待排序元素插入前 i 个排好序的元素序列中，共有 $i+1$ 个可能的插入位置，在每个位置插入的可能性为 $1/(i+1)$，则平均情况下比较次数为
    $$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$
  - `InsertionSort` 总共进行了 $n-1$ 次插入操作，比较次数总和为
    $$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{j} \approx \frac{n^2}{4} \in \Theta(n^2)$$

1.2 类似算法分析

类似算法的复杂度下界

- **类似算法**：每次比较之后仅交换一对相邻元素位置的排序算法
- 设原序列 $E = (x_1, x_2, \cdots, x_n)$，定义 π 为原序列的一种排列，即对于 $1 \le i \le n$，$\pi(i)$ 是 x_i 在排好序的序列中的实际位置

- 排列中的反序对 $(\pi(i), \pi(j))$：$i < j$ 且 $\pi(i) > \pi(j)$
- 上述算法的最少比较次数等于输入序列中反序对的个数
- 有一种排列其反序对个数为 $n(n-1)/2$，因此上述算法最坏情况下复杂度下界为 $n(n-1)/2 \in \Omega(n^2)$
- 平均情况？

Theorem 1.1. 任何使用比较为基本操作，并且每次比较后最多去除一个反序对的排序算法，输入规模为 n 时，在最坏情况下的最少比较次数为 $n(n-1)/2$，在平均情况下的最少比较次数为 $n(n-1)/4$

另一个类似算法：选择排序

Algorithm `SelectionSort`$(E[], n)$

```
1 for i ← 0 to n − 1 do
2 |   k ← i;
3 |   for j ← i to n − 1 do
4 |   |   if E[j] < E[k] then k ← j;
5 |   end
6 |   E[i] ↔ E[k];
7 end
```

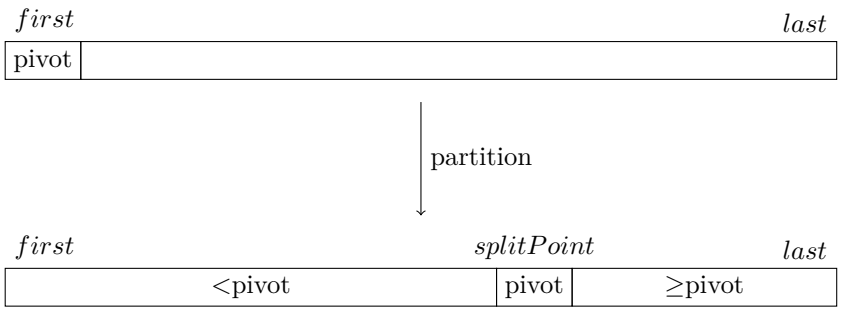
- Worst-Case：$W(n) = n(n-1)/2$
- Average-Case：$A(n) = n(n-1)/2$
- **优点**：不需频繁移动数组元素

2 用分治法设计排序算法

2.1 快速排序

基本策略

- 将原序列分解为两个子序列，使一个子序列里面的元素小于另一个子序列中的元素
- 对两个子序列分别排序 (递归求解)
- C. A. R. Hoare 在 1962 年提出

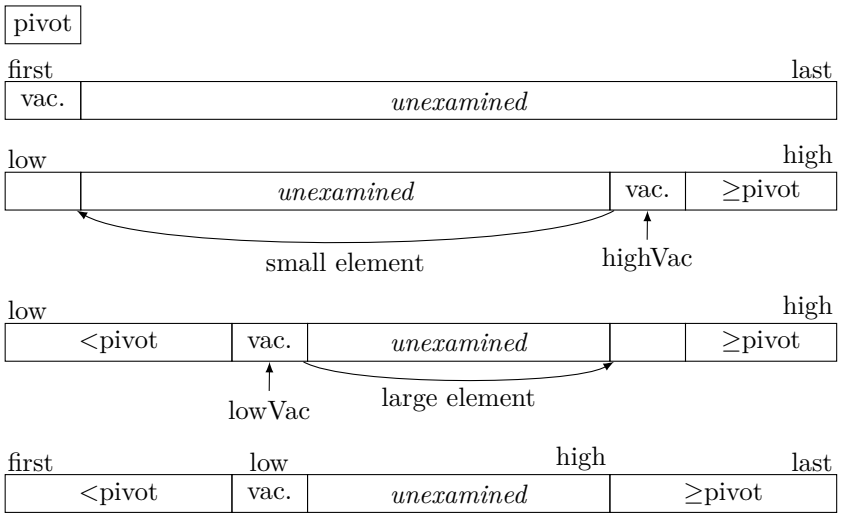


`QuickSort` 算法描述

Algorithm `QuickSort`$(E[], first, last)$

```
1 if first < last then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 end
```

In Place `Partition`



`Partition`

Procedure `Partition`$(E[], pivot, first, last)$

```
1 low ← first;
2 high ← last;
3 while low < high do
4 |   highVac ← ExtendLargeRegion(E, pivot, low, high);
5 |   lowVac ← ExtendSmallRegion(E, pivot, low + 1, highVac);
6 |   low ← lowVac;
7 |   high ← highVac − 1;
8 end
9 return low;
```

`ExtendLargeRegion`

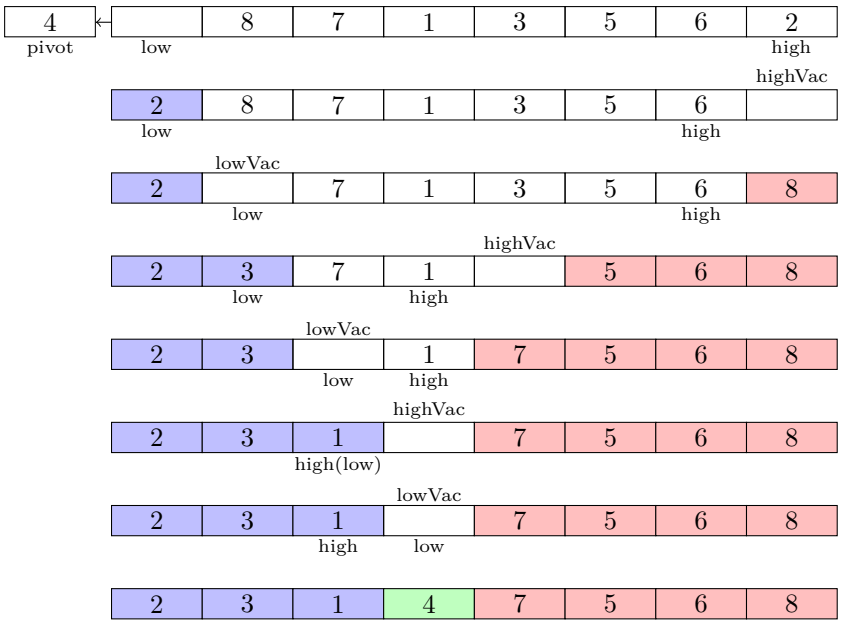
Procedure `ExtendLargeRegion`$(E[], pivot, lowVac, high)$

```
1  highVac ← lowVac;                          // In case no element < pivot.
2  curr ← high;
3  while curr > lowVac do
4  |  if E[curr] < pivot then
5  |  |  E[lowVac] ← E[curr];
6  |  |  highVac ← curr;
7  |  |  break;
8  |  end
9  |  curr ← curr − 1;
10 end
11 return highVac;
```

`ExtendSmallRegion`

Procedure `ExtendSmallRegion`$(E[], pivot, low, highVac)$

```
1  lowVac ← highVac;                          // In case no element ≥ pivot.
2  curr ← low;
3  while curr < highVac do
4  |  if E[curr] ≥ pivot then
5  |  |  E[highVac] ← E[curr];
6  |  |  lowVac ← curr;
7  |  |  break;
8  |  end
9  |  curr ← curr + 1;
10 end
11 return lowVac;
```



最坏情况复杂度分析

- `Partition`: 对于长度为 k 的序列，共需比较 $k-1$ 次；
- 最坏情况下的 `Partition` 结果是 $splitPoint = first$ or $last$，原序列被分成一个空序列和一个比原序列少一个元素 (pivot) 的子序列；

- 因此，最坏情况下时间复杂度为：

$$W(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- *QuickSort* 徒有虚名？

平均情况复杂度分析

- 我们已经知道：当一次比较最多仅能消除一个反序对的时候，平均情况下比较次数至少为 $n(n-1)/4$
- 对于 QuickSort 来说，一次比较后，元素在序列中可能移动相当大的距离，这意味着每次比较后可能消除多个反序对 (最多 $n-1$)
- 每次 Partition 之后得到的两个子序列各自的元素之间两两未经比较，因此可以认为子序列各种排序出现的概率保持相等
- 假设序列元素每种排列出现的概率相等，Partition 返回的 *splitPoint* 位置出现在每个序列元素位置的概率也相等，则：

$$A(n) = (n-1) + \sum_{i=0}^{n-1}\frac{1}{n}(A(i) + A(n-1-i)) \qquad \text{for } n \geq 2$$
$$A(1) = A(0) = 0$$

- 整理一下得：$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \quad \text{for } n \geq 1$

求 $A(n)$

- 做一个猜想：如果 Partition 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2) \xrightarrow{\text{主定理}} A(n) \in \Theta(n\log n)$$

Theorem 2.1. 对于如下定义的 $A(n)$：

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \quad \textit{for } n \geq 1$$

存在常数 c，使得 $A(n) \leq cn\ln n$.

- 一个更精确的近似：$A(n) \approx 1.386n\lg n - 2.846n$

Proof. 用数学归纳法。
奠基：$A(1) = 0 \leq c1\ln 1 = 0$.
归纳：

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}ci\ln i$$

用积分定界：

$$\sum_{i=1}^{n-1}ci\ln i \leq c\int_1^n x\ln x dx = c\left(\frac{1}{2}n^2\ln n - \frac{1}{4}n^2\right)$$

因此：

$$A(n) \leq n - 1 + \frac{2c}{n}\left(\frac{1}{2}n^2\ln n - \frac{1}{4}n^2\right) = cn\ln n + n(1 - \frac{c}{2}) - 1$$

当 $c \geq 2$ 时，$A(n) \leq cn\ln n$ □

更精确的分析：

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$

$$A(n-1) = n - 2 + \frac{2}{n-1}\sum_{i=1}^{n-2} A(i)$$

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

令：

$$B(n) = \frac{A(n)}{n+1}$$

即得：

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)} \qquad B(1) = 0$$

$$B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)} \approx 2(\ln n + 0.577) - \frac{4n}{n+1}$$

因此：

$$A(n) \approx 1.386n \lg n - 2.846n \qquad (\text{注：} \ln n = \frac{\lg n}{\lg e} \approx 0.693 \lg n)$$

改进

- 选择合适的 pivot
  - 随机选择；选择 $E[first]$、$E[(first + last)/2]$、$E[last]$ 的中间值
- 减少递归调用
  - 将递归变为循环；减小递归调用深度 (`SmallSort`)

Algorithm `QuickSort`$(E[], first, last)$

```
1 if last − first > smallSize then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 else
8 |   SmallSort(E, first, last);
9 end
```

改进 (cont.)

- 快速排序的递归深度：$O(n) \Rightarrow O(\lg n)$

```
Algorithm QuickSort2(E[], first, last)
1  while first < last do
2  |  pivot ← E[first];
3  |  splitPoint ← Partition(E, pivot, first, last);
4  |  E[splitPoint] ← pivot;
5  |  if splitPoint − first < last − splitPoint then
6  |  |  QuickSort2(E, first, splitPoint − 1);
7  |  |  first ← splitPoint + 1;
8  |  else
9  |  |  QuickSort2(E, splitPoint + 1, last);
10 |  |  last ← splitPoint − 1;
11 |  end
12 end
```

2.2 归并排序

归并操作 (Merge)

- 归并操作，或归并算法，指的是将两个有序序列合并成一个有序序列的操作
- 基本原理：
  1. 申请空间，使其大小为两个已经排序序列之和，用来存放合并后的序列；
  2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
  3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
  4. 重复上一步骤直到某一指针达到序列尾；
  5. 将另一序列剩下的所有元素直接复制到合并序列尾

递归描述

```
Algorithm Merge(A[], B[], C[])
1  if A is empty then rest of C ← rest of B;
2  else if B is empty then rest of C ← rest of A;
3  else
4  |  if first of A ≤ first of B then
5  |  |  first of C ← first of A;
6  |  |  Merge(rest of A, B, rest of C);
7  |  else
8  |  |  first of C ← first of B;
9  |  |  Merge(A, rest of B, rest of C);
10 |  end
11 end
```

非递归版本

Algorithm `Merge`$(A[], k, B[], m, C[])$

```
1  n ← k + m;
2  indexA ← indexB ← indexC ← 0;
3  while indexA < k and indexB < m do
4      if A[indexA] ≤ B[indexB] then
5          C[indexC] ← A[indexA];
6          indexA ← indexA + 1;
7          indexC ← indexC + 1;
8      else
9          C[indexC] ← B[indexB];
10         indexB ← indexB + 1;
11         indexC ← indexC + 1;
12     end
13 end
14 if indexA ≥ m then  Copy B[indexB, ..., m − 1] to C[indexC, ..., n − 1] ;
15 else  Copy A[indexA, ..., k − 1] to C[indexC, ..., n − 1] ;
```

归并操作时间复杂度分析

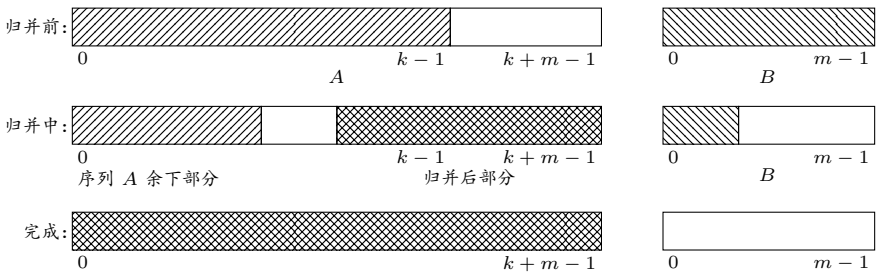
- Worst-Case：
  - 每做完一次比较，至少有一个元素被移到合并后的序列中
  - 最后一次比较时，至少还剩 2 个元素未移到合并序列中，这时合并序列中最多有 $n-2$ 个元素，加最后一次至多经过了 $n-1$ 次比较
  - $W(n) = n - 1 \in \Theta(n)$
- Average-Case?

Theorem 2.2. 任何对两个均包含 $k = m = n/2$ 个元素的有序序列，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

Corollary 2.3. 任何对两个分别包含 k 和 m 个元素的有序序列，并且 $|k-m| = 1$，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



最坏情况下额外占用的存储空间：$n/2 \in \Theta(n)$ 当 $k = m = n/2$

归并排序 (Mergesort)

- 快速排序最大的问题实际上是每次序列分割不一定能将原序列分成长度相等的两个子序列
- **归并排序**则每次将序列分割为恰好相等的两个子序列，分别对子序列进行递归排序，再将排好序的两个子序列通过归并操作合并成一个序列

Algorithm `Mergesort`$(E[], first, last)$

```
1 if first < last then
2 |   mid ← (first + last)/2;
3 |   Mergesort(E, first, mid);
4 |   Mergesort(E, mid + 1, last);
5 |   Merge(E, first, mid, last);
6 end
```

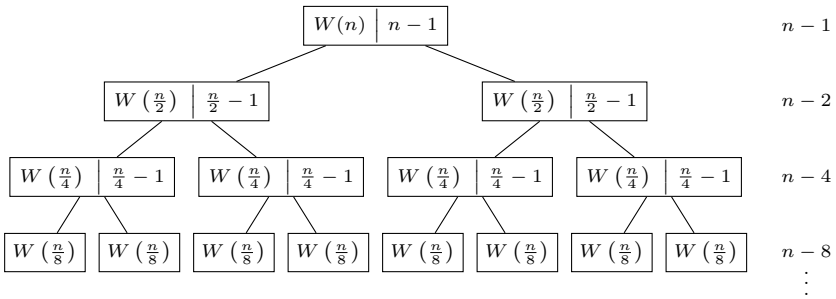
归并排序时间复杂度

- Worst-Case:

$$
\begin{aligned}
&W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \\
&W(1) = 0 \\
&\qquad \Downarrow \text{主定理} \\
&W(n) \in \Theta(n \log n)
\end{aligned}
$$

- 一个更精确的结果：$\lceil n \lg n - n + 1 \rceil \le W(n) \le \lceil n \lg n - 0.914n \rceil$

`Mergesort` 递归树



- 不包含叶结点的每层非递归开销为：$n - 2^d$
- 叶结点 $(W(1) = 0)$ 深度为 $\lceil \lg(n+1) \rceil - 1$ 或 $\lceil \lg(n+1) \rceil$
- 恰有 n 个叶结点
- 令递归树最大深度 (高度) 为 $D = \lceil \lg(n+1) \rceil$，在深度为 $D-1$ 层有 B 个叶结点，则在 D 层有 $n - B$ 个叶结点，在 $D-1$ 层有 $(n-B)/2$ 个非叶结点，且 $B = 2^D - n$ ($Why?$)

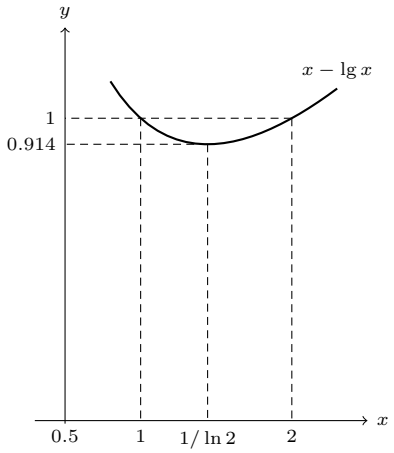
时间复杂度计算

$$
\begin{aligned}
W(n) &= \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2) \\
&= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2} \\
&= nD - 2^D + 1
\end{aligned}
$$

$$
\alpha = \frac{2^D}{n} \Rightarrow 1 \le \alpha < 2
$$

$$
W(n) = n\lg n - (\alpha - \lg\alpha)n + 1
$$

$$
0.914 \le \alpha - \lg\alpha \le 1
$$



$$
\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil
$$

一点讨论

- 快速排序时间复杂度：$W(n) \in \Theta(n^2),\ A(n) \approx 1.386n\lg n - 2.846n$
- 归并排序：$\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil$
- 是否意味着归并排序比快速排序更好？
  - 时间复杂度
  - 空间复杂度

Exercise (4). 试分析比较快速排序和归并排序在平均情况下的元素移动次数。

deadline: 2018.12.15

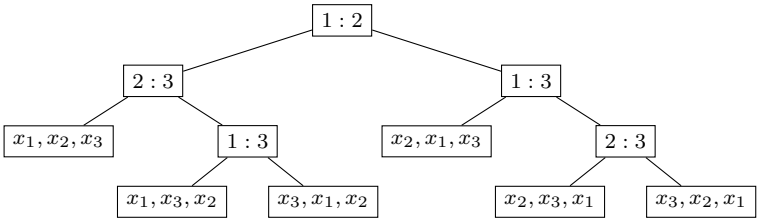
2.3 比较排序的复杂度下界

排序算法的决策树

- 任意一种以元素间的比较为基本操作的排序算法其执行过程都对应于一棵二叉决策树：
  - 内结点标注 $(i:j)$，表示数组元素 x_i 与 x_j 的一次比较操作，其左子树对应 $x_i < x_j$ 的情况下下一步要进行的比较操作（或输出结果操作），右子树对应 $x_i > x_j$ 的情况下下一步要进行的比较操作（或输出结果操作）

– 每个外结点表示一个排序结果

- 例：输入规模 $n = 3$ 时的决策树：



最坏情况复杂度下界

Lemma 2.4. 若高度为 h 的二叉树有 L 个叶结点，则有：$L \leq 2^h$ 及 $h \geq \lceil \lg L \rceil$

Lemma 2.5. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法其对应决策树的高度至少为 $\lceil \lg n! \rceil$

Theorem 2.6. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法在最坏情况下至少要执行 $\lceil \lg n! \rceil \approx \lceil n \lg n - 1.443n \rceil$ 次比较操作

$$\text{注：}\ \lg n! = \sum_{j=1}^{n} \lg j \geq n \lg n - (\lg e)n$$

平均情况复杂度下界

- 根据决策树定义，很显然是 2-tree，而每一条从根结点到叶结点的路径代表了一次成功的排序过程
- 设决策树外路径长度为 epl，叶结点个数为 L，则平均比较次数为 epl/L
- $\Longrightarrow$ 找到 epl 的下界：$epl \geq L \lg L$

Theorem 2.7. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法在平均情况下至少要执行 $\lg n! \approx n \lg n - 1.443n$ 次比较操作

2.4 堆排序

回顾：二叉堆 (Binary Heap)

- **堆** (Heap)：基于树的数据结构，其满足堆特性：若 B 是 A 的子结点，则 $\text{key}(A) \geq \text{key}(B)$ (大根堆)
- **二叉堆**：用完全二叉树来表达的堆结构，是实现优先队列的最有效的数据结构之一
- **二叉堆的基本操作**：
  - 往堆中添加元素 (heapify-up, up-heap, bubble-up)：
    1. 将新元素插入堆的尾部
    2. 将新元素与其父结点比较，若满足堆特性，则结束
    3. 否则，将其与父结点交换位置，并重复上一步骤
  - 提取并删除堆首元素 (heapify-down, down-heap, bubble-down)：
    1. 将堆尾元素放到堆首代替已删除的堆首元素
    2. 将其与子结点比较，若满足堆特性，则结束
    3. 否则，将其与子结点中的一个交换位置，使三者满足堆特性，并重复上一步骤
- ✠ **效率**：入队和出队的复杂度 $W(n) \in O(\log n)$

堆排序 (Heapsort) 算法要点

Algorithm `Heapsort`$(E[], n)$

```
1 从 E 构造堆 H;
2 for i = n down to 2 do
3 |   curMax ← GetMax(H);
4 |   DeleteMax(H);
5 |   E[i] ← curMax;
6 end
```

Algorithm `DeleteMax`(H)

```
1 拷贝堆 H 的底层最右边结点中的元素到 K 中;
2 删除堆 H 的底层最右边结点中的元素;
3 FixHeap(H, K);
```

堆的调整：`FixHeap`

Algorithm `FixHeap`(H, K)

```
1  if H 是叶结点 then
2  |   将 K 插入 Root(H);
3  else
4  |   取左右子树中根结点元素较大的那棵作为 largerSubHeap;
5  |   if K ≥Root(largerSubHeap) then
6  |   |   将 K 插入 Root(H);
7  |   else
8  |   |   将 Root(largerSubHeap) 插入 Root(H);
9  |   |   FixHeap(largerSubHeap, K);
10 |   end
11 end
```

堆的构造：`ConstructHeap`

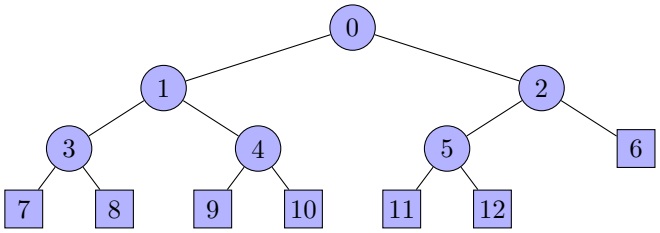
Algorithm `ConstructHeap`(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

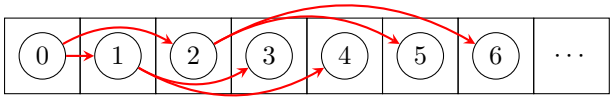
- 最坏情况复杂度分析：
  - `FixHeap`: 大约需要 $2\lg n$ 次比较
  - `ConstructHeap`: $W(n) = W(n-r-1)+W(r)+2\lg n$, r 为右子树结点数, 且 $n > 1$
  - $\Rightarrow W(n) \in \Theta(n)$ (*Why?*)

回顾：二叉堆 (完全二叉树) 的数组存储方式

Example 2.1 (完全二叉树).



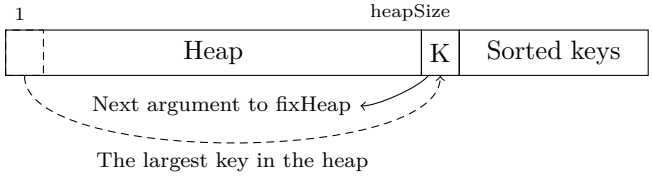
完全二叉树的顺序存储方式：



数组实现版本

Algorithm `Heapsort`$(E[], n)$

```
1 ConstructHeap(E, n);
2 for heapSize ← n down to 2 do
3 |   curMax ← E[0];
4 |   K ← E[heapSize − 1];
5 |   FixHeap(E, heapSize − 1, 0, K);
6 |   E[heapSize − 1] ← curMax;
7 end
```



`FixHeap` 的数组实现

Algorithm `FixHeap`$(E[], heapSize, root, K)$

```
1  left ← 2 ∗ root + 1, right ← 2 ∗ root + 2;
2  if left ≥ heapSize then E[root] ← K;
3  else
4  |   if left = heapSize − 1 then largerSubHeap ← left;
5  |   else if E[left] > E[right] then largerSubHeap ← left;
6  |   else largerSubHeap ← right;
7  |   if K ≥ E[largerSubHeap] then E[root] ← K;
8  |   else
9  |   |   E[root] ← E[largerSubHeap];
10 |   |   FixHeap(E, heapSize, largerSubHeap, K);
11 |   end
12 end
```

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`: $\in \Theta(n)$
- `FixHeap`: $2\lfloor \lg k \rfloor$

- 循环：$2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor$

$$\begin{aligned} 2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor &\le 2\int_1^n (\lg e)\ln x dx \\ &= 2(\lg e)(n\ln n - n) \\ &= 2(n\lg n - 1.443n) \end{aligned}$$

Theorem 2.8. `Heapsort` 排序算法在最坏情况下所需的比较操作次数为 $2n\lg n + O(n)$，其时间复杂度为 $\Theta(n\log n)$

平均情况?

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3 |   parent ← (vacant − 1)/2;
4 |   if K ≤ E[parent] then  E[vacant] ← K ;
5 |   else
6 |   |   E[vacant] ← E[parent];
7 |   |   BubbleUpHeap(E, root, K, parent);
8 |   end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到底层后使用 `BubbleUpHeap` 将 K “提升”(bubble up) 到适当位置
- 由于我们取的 K 是堆尾元素，应该是一个“相当小”的元素，因此在平均情况下应该用比空位下降更少的比较就能上升到合适位置
- 注意：最坏情况仍然需要 $2\lfloor \lg k \rfloor$ 次比较
- 能不能做得更好一些？

进一步的改进 —— Divide and Conquer

- Risky `FixHeap` 每次只下降当前堆 $\frac{h}{2}$ 高度 (可以称为 `Promote`)
- 当下降到当前空位父结点元素小于 K 时开始 `BubbleUpHeap`

Algorithm `Promote`$(E[], hStop, vacant, h)$

```
1  left ← 2 ∗ vacant + 1;
2  right ← 2 ∗ vacant + 2;
3  if h ≤ hStop then
4  |   vacStop ← vacant;
5  else if E[left] ≤ E[right] then
6  |   E[vacant] ← E[right];
7  |   vacStop ← Promote(E, hStop, right, h − 1);
8  else
9  |   E[vacant] ← E[left];
10 |   vacStop ← Promote(E, hStop, left, h − 1);
11 end
12 return vacStop;
```

更快的 FixHeap

Algorithm FixHeapFast($E[], n, K, vacant, h$)

```
1  if h ≤ 1 then
2  |  处理高度为 0 或 1 的情况;
3  else
4  |  hStop ← h/2;
5  |  vacStop ← Promote(E, hStop, vacant, h);
6  |  vacParent ← (vacStop − 1)/2;
7  |  if E[vacParent] ≤ K then
8  |  |  E[vacStop] = E[vacParent];
9  |  |  BubbleUpHeap(E, vacant, K, vacParent);
10 |  else
11 |  |  FixHeapFast(E, n, K, vacStop, hStop);
12 |  end
13 end
```

时间复杂度分析

- 直观地看:
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用,则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较,而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程: $T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$
- 假设 $T(h) \geq h$: $T(h) = \lceil h/2 \rceil + 1 + T(\lfloor h/2 \rfloor) \quad T(1) = 2 \Longrightarrow T(h) \approx h + \lceil \lg(h+1) \rceil \approx \lg(n+1) + \lg\lg(n+1)$

Theorem 2.9. 使用 FixHeapFast 来调整堆结构的堆排序算法在最坏情况下其时间复杂度为 $n\lg n + \Theta(n \log\log n)$

3 其他排序算法

3.1 希尔排序

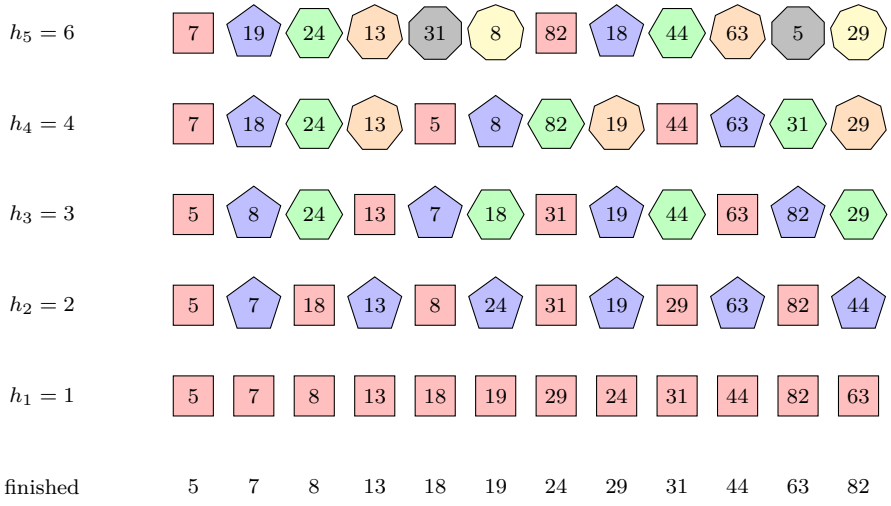
基本思想

- 回顾: **插入排序** (InsertionSort)
  - 如果原始数据的大部分元素已经有序，那么插入排序的速度很快 (因为需要移动的元素很少)
  - 从这个事实我们可以想到，如果原始数据只有很少元素，那么排序的速度也很快 (SmallSort)
  - 插入排序的低效之处在于每次比较交换相邻元素最多只能消除一个反序对
- **希尔排序** (Shell Sort)又称**缩小增量排序** (diminishing increment sort)，就是利用插入排序的特点设计的一种优秀的排序法，算法本身不难理解，也易于实现，而且速度很快。其基本原理为:

  - 指定一组递减增量序列 (最后一个增量为 1)
  - 依次取序列中的增量为矩阵行宽，将待排序序列按矩阵排列
  - 对矩阵每一列进行插入排序

- 算法名称取自其发明人 D. L. Shell 的名字，于 1959 年发表

示例



算法描述

Algorithm `Shellsort`$(E[], n, h[], t)$

```
1 for s ← t − 1 to 0 do
2 |  for xindex ← h[s] to n − 1 do
3 |  |  current ← E[xindex];
4 |  |  xloc ←ShiftVacH(E, h[s], xindex, current);
5 |  |  E[xloc] ← current;
6 |  end
7 end
```

`ShiftVacH`

Procedure `ShiftVacH`$(E[], h, xindex, cur)$

```
1 vacant ← xindex;
2 while vacant ≥ h do
3 |  if E[vacant − h] ≤ cur then
4 |  |  break;
5 |  end
6 |  E[vacant] ← E[vacant − h];
7 |  vacant ← vacant − h;
8 end
9 return vacant;
```

复杂度分析

- `Shellsort` 的时间复杂度取决于增量序列的选择，其分析难度较大，仍然是一个开放的问题

- 已知复杂度上界
  - [Pratt, 1971]：$\Theta(n(\log n)^2)$，
    增量序列：$1, 2, 3, 4, 6, 9, 8, 12, 18, 27, \cdots, 2^i 3^j$
  - [Papernov-Stasevich, 1965; Pratt, 1971]：$\Theta(n^{3/2})$，
    增量序列：$1, 3, 7, 15, \cdots, 2^k - 1$
  - [Sedgewick, 1982]：$O(n^{4/3})$，
    增量序列：$1, 8, 23, 77, 281, 1073, 4193, 16577, \cdots, 4^{j+1} + 3 \cdot 2^j + 1$
  - [Incerpi-Sedgewick, 1985]：对于任意 $\epsilon > 0$，必存在一种增量序列使得 `Shellsort` 的时间复杂度为 $O(n^{1+\epsilon/\log n})$，且只用 $\frac{8}{\epsilon^2}\log n$ 遍排序
- 已知复杂度下界
  - [Poonen, 1993]：对大小为 n 的序列进行 m 遍排序的 `Shellsort` 至少需要 $n^{1+c/\sqrt{m}}$ 次比较 $(c > 0)$
- 平均情况
  - [Knuth, 1973]：两遍 $(h, 1)$ `Shellsort` 需要 $2n^2/h + \sqrt{\pi n^3 h}$ 次比较 $(h \in O(n^{1/3}) \Rightarrow A(n) \in O(n^{5/3}))$
  - [Yao, 1980]：三遍 $(h, k, 1)$ `Shellsort` 需要 $\frac{2n^2}{h} + \frac{1}{k}\left(\sqrt{\frac{\pi n^3 h}{8}} - \sqrt{\frac{\pi n^3}{8h}}\right) + \psi(h, k)n$ 次比较
- 悬而未决的问题
  - 是否还存在更好的增量序列？
  - 最坏情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 平均情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 如果使用其他排序算法代替每遍排序使用的插入排序算法，时间复杂度是否还能降低？
- 参考文献

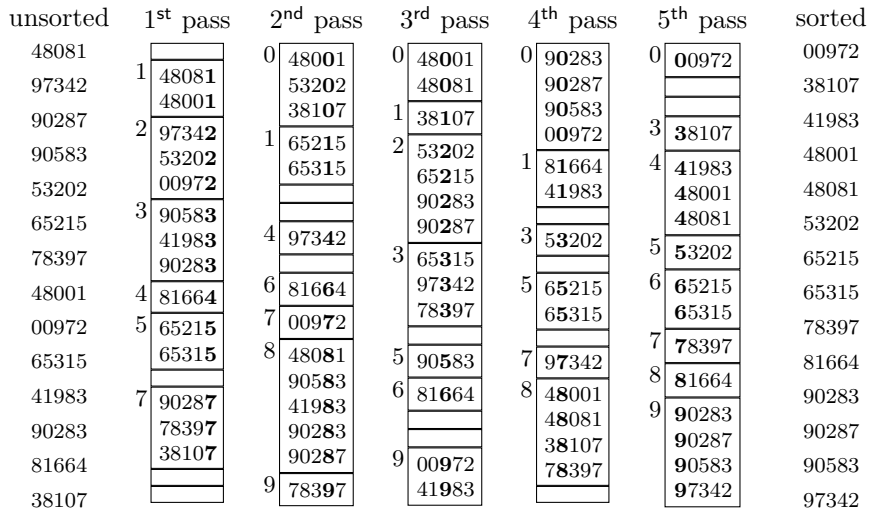
[1] Robert Sedgewick. *Analysis of Shellsort and Related Algorithms.* ESA '96: Fourth Annual European Symposium on Algorithms, 1996

3.2 基数排序

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- **桶排序** (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素
  - 算法的基本思想是：(1) 分配待排序元素到各个桶；(2) 对每个桶内分别进行排序；(3) 将排序结果组合到一起
  - 假定元素分配尽可能平均，每个桶内排序基于比较操作，则桶排序的时间复杂度可以达到 $O(n \log(n/k))$，其中 k 为桶的个数，若取 $k = n/c$，复杂度为 $O(n \log c) = O(n)$
- **基数排序** (Radix Sort)：一种复杂度能达到 $O(n)$ 的桶排序算法

基数排序



算法描述

- 用链表存储待排序元素和桶元素

Algorithm `RadixSort`(List L, int $radix$, int $numFields$)

```
1 List[] buckets ← new List[radix];
2 List newL ← L;
3 for field ← 0 to numFields do
4 |   初始化 buckets 中元素为空链表;
5 |   Distribute(newL, buckets, radix, field);
6 |   newL ← Combine(buckets, radix);
7 end
8 return newL;
```

`Distribute`

Procedure `Distribute`(List L, List[] $buckets$, $radix$, $field$)

```
1 List remL ← L;
2 while remL ≠ null do
3 |   K ← First(remL);                          /* 取表头元素 */
4 |   b ← MaskShift(field, radix, K);           /* 取当前位数字 */
5 |   buckets[b] ← Cons(K, buckets[b]);         /* 插入到相应桶 */
6 |   remL ← Rest(remL);                        /* 取表尾 */
7 end
```

`Combine`

Procedure `Combine`(List[] *buckets*, int *radix*)

```
for b ← radix − 1 down to 0 do
    remBucket ← buckets[b];
    while remBucket ≠null do
        K ← First(remBucket);
        L ← Cons(K, L);
        remBucket ← Rest(remBucket);
    end
end
return L;
```

复杂度分析

- **时间复杂度：**
  - `Distribute`: $\Theta(n)$
  - `Combine`: $\Theta(n)$
  - `RadixSort`:总共调用 $numFields$ 次 `Distribute` 和 `Combine`,一般情况下 $numFields$ 为常数，因此 `RadixSort` 复杂度为 $\Theta(n)$
- **空间复杂度：**
  - 额外空间主要用在存储元素的链表结构上，所需空间为 $\Theta(n)$

4 排序算法比较

排序算法的稳定性

- **排序算法的稳定性：**若待排序的序列中，存在多个具有相同键值的记录，经过排序，这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对次序发生了改变，则称该算法是不稳定的。
- **稳定性的好处：**排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。
- **何时需要稳定的排序算法：**不希望改变具有相同键值的记录原有的顺序。例如：对学生按成绩排序，对于成绩相同的学生，希望保持原有的学号顺序

排序算法比较

算法	最坏情况	平均情况	空间占用	稳定性
插入排序	$n^2/2$	$n^2/4$	$\Theta(1)$	是
选择排序	$n^2/2$	$n^2/2$	$\Theta(1)$	否
快速排序	$n^2/2$	$1.386n\lg n$	$\Theta(\log n)$	否
归并排序	$n\lg n$	$n\lg n$	$\Theta(n)$	是
堆排序	$2n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
快速堆排序	$n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
希尔排序	$\Theta(n\log^2 n)$	?	$\Theta(1)$	否
基数排序	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	是