Data Structure Note 4 Sorting

Part I. Sorting Overview

📢 **Sorting Algorithms:**

- Sorting with Comparison:

  lower bound: $O(NlogN)$

  - Insertion Sort:

    $O(N^2)$

    Space Complexity: $O(1)$

  - Merge Sort:

    $O(NlogN)$

    Space Complexity: $O(N)$

  - Heap Sort:

    $O(NlogN)$

  - Quick Sort:

    Expected case: $\Theta(NlogN)$

    Worst case: $\Theta(N)$

    Space Complexity: $O(1)$

- Sorting without Comparison:

  - Counting Sort
  - Radix Sort

Part II. Quick Sort

> Basic Idea of Quick Sort
>
> 1. Divide: **Partition** the array into two subarrays around a **pivot** `x` such that **elements in lower subarray** $\leq x$, **elements in upper subarrys** $\geq x$.
>
> 
>
>
> 2. Conquer: Recursively sort the two subarrays

Expected conquer cost: $O(n)$

3. Combine: Trivial(微不足道的)

1.Time complexity analysis(Naive):

- Expected Case:

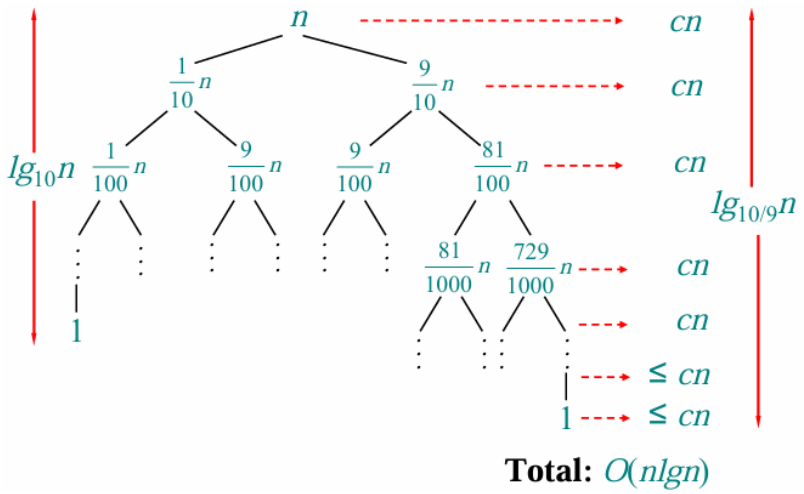
We suppose we can get the ideal case, in which the pivot divides the array into exactly two identically half subarrays.

$$T(n) = 2T(\frac{N}{2}) + O(N)$$

We can use the master method to analyze the time complexity.

$$T(N) = Nlog(N)$$

Or we can suppose that pivot divides the array into exactly one subarray with `k` of the total elements and the other with `1-k` of the elements, $0 < k < 1$



We assume $\lambda = max\{k, 1-k\}$

Tree Height: $log_{\frac{1}{\lambda}} N$

Cost for each layer: cn

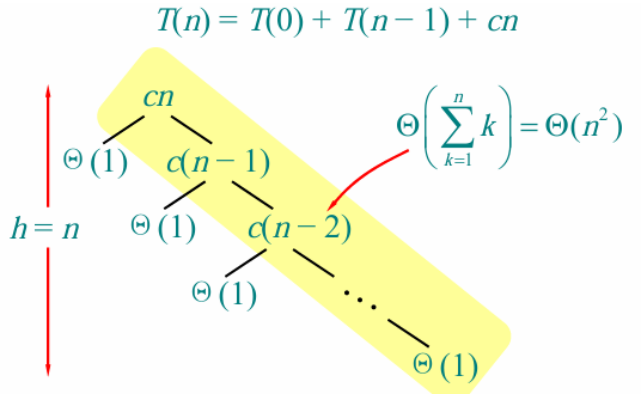
$$\Longrightarrow T(N) = \Theta(NlogN)$$

- Worst case:

We suppose we can get the worst case, in which the pivot divides the array into exactly one subarray with 0 element and the other with `n-1` element.

$$T(N) = T(0) + T(N-1) + \Theta(N)$$

$$T(N) = \Theta(1) + T(N-1) + \Theta(N) = \Theta(N^2)$$

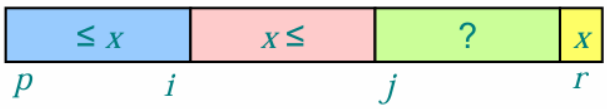


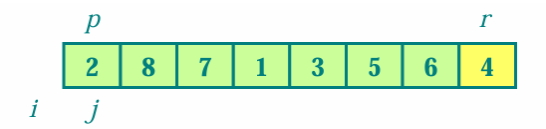
2.Quick Sort Implementation

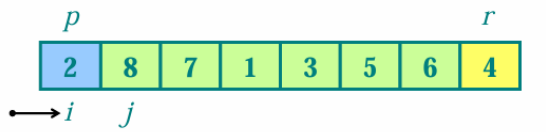
```
QUICKSORT(A, p, r)
1. if p < r
2.     then q ← PARTITION(A, p, r)
3.          QUICKSORT(A, p, q − 1)
4.          QUICKSORT(A, q + 1, r)
```

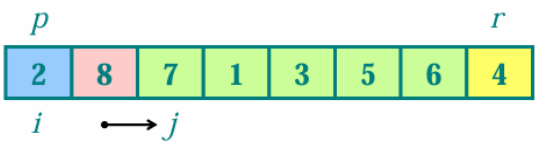
Initial call: QUICKSORT$(A, 1, n)$

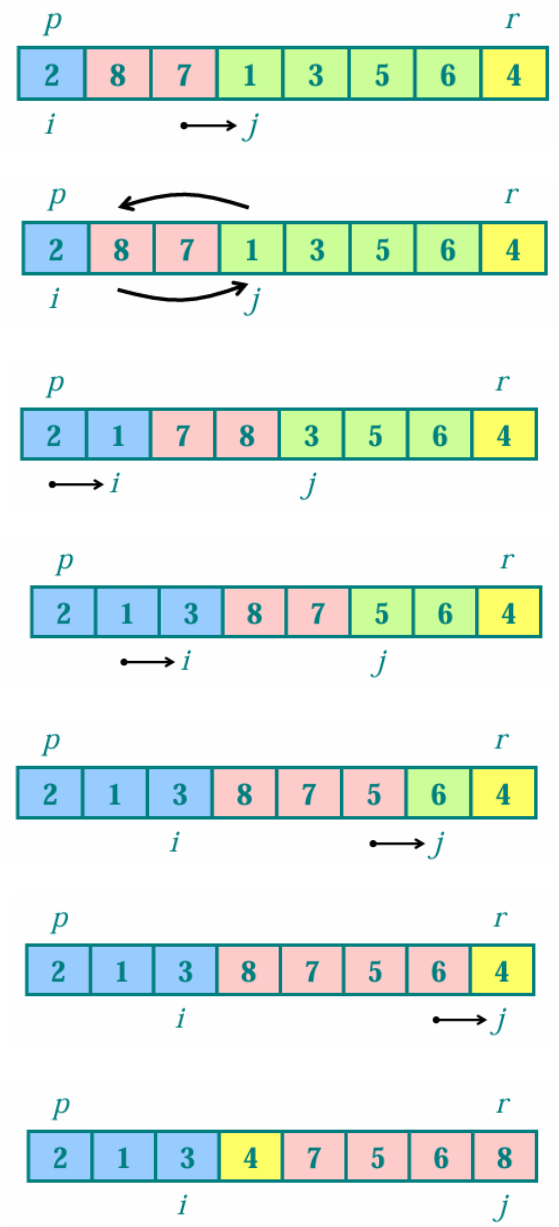
```
PARTITION(A, p, r)    //A[p ... r]
1. x ← A[r]    //pivot = A[p]
2. i ← p − 1
3. for j ← p to r − 1
4.     do if A[j] ≤ x
5.         then i ← i + 1
6.              exchange A[i] ↔ A[j]
7. exchange A[r] ↔ A[i + 1]
8. return i + 1
```











We can use the last element as pivot, implemented in the way which the pseudocode describes:

Quick Sort1

```
template<typename T>
void Quick_sort_1(std::vector<T>& arr){

    std::function<void(int, int)> helper = [&](int left, int right) -> void{
        // right is exclusive
        if(left >= right - 1){
            return;
        }

        // partition, choose arr[right - 1] to be the pivot
        T pivot = arr[right - 1];
        // i points to the last element we have explored that's smaller than the pivot
        int i = left - 1;
        // j points to the first element that we haven't explored.
```

```
        int j = left;

        while(j < right - 1){
            if(arr[j] > pivot){
                j++;
            }
            else{
                std::swap(arr[i + 1], arr[j]);
                i++;
                j++;
            }
        }

        std::swap(arr[i + 1], arr[right - 1]);

        helper(left, i + 1);
        helper(i + 2, right);
    };

    helper(0, arr.size());
}
```

We can also use the first element as pivot, implemented in a slightly different way.

Quick Sort2

```
template<typename T>
void Quick_sort_2(std::vector<T>& arr){

    std::function<void(int, int)> helper = [&](int left, int right) -> void{
        // right is exclusive
        if(left >= right - 1){
            return;
        }

        // partition, choose arr[left] to be the pivot
        T pivot = arr[left];

        int small_point = left + 1;
        int big_point = right - 1;
        /**We can guarantee that:
         * When index < small_point, elements must be smaller than the pivot
         * When index > big_point, elements must be bigger than the pivot
        */
        while(small_point <= big_point){
            if(arr[small_point] < pivot){
```

```
                small_point++;
            }
            else{
                std::swap(arr[small_point], arr[big_point]);
                big_point--;
            }
        }

        std::swap(arr[left], arr[small_point - 1]);

        helper(left, small_point - 1);
        helper(small_point, right);
    };

    helper(0, arr.size());
}
```

Average case: no human effect(not $nlgn$ exactly)

Expected case: with human effect

3.Strict Analysis of Quick Sort

7.4.2　期望运行时间

我们已经从直观上了解了为什么 RANDOMIZED-QUICKSORT 的期望运行时间是 $O(n \lg n)$：如果在递归的每一层上，RANDOMIZED-PARTITION 将任意常数比例的元素划分到一个子数组中，则算法的递归树的深度为 $\Theta(\lg n)$，并且每一层上的工作量都是 $O(n)$。即使在最不平衡的划分情况下，会增加一些新的层次，但总的运行时间仍然保持是 $O(n \lg n)$。要准确地分析 RANDOMIZED-QUICKSORT 的期望运行时间，首先要理解划分操作是如何进行的；然后，在此基础之上，推导出期望运行时间的一个 $O(\lg n)$的界。有了这一期望运行时间的上界，再加上 7.2 节中得到的最好情况界 $\Theta(n \lg n)$，我们就能得到 $\Theta(n \lg n)$这一期望运行时间。在这里，假设待排序的元素始终是互异的。

运行时间和比较操作

QUICKSORT 和 RANDOMIZED-QUICKSORT 除了如何选择主元元素有差异以外，其他方面完全相同。因此，我们可以在讨论 QUICKSORT 和 PARTITION 的基础上分析 RANDOMIZED-QUICKSORT。其中，RANDOMIZED-QUICKSORT 随机地从子数组中选择元素作为主元元素。

QUICKSORT 的运行时间是由在 PARTITION 操作上所花费的时间决定的。每次对 PARTITION 的调用时，都会选择一个主元元素，而且该元素不会被包含在后续的对 QUICKSORT 和 PARTITION 的递归调用中。因此，在快速排序算法的整个执行期间，至多只可能调用 PARTITION 操作 n 次。调用一次 PARTITION 的时间为 $O(1)$再加上一段循环时间。这段时间与第 3～6 行中 **for** 循环的迭代次数成正比。这一 **for** 循环的每一轮迭代都要在第 4 行进行一次比较：比较主元元素与数组 A 中另一个元素。因此，如果我们可以统计第 4 行被执行的总次数，就能够给出在 QUICKSORT 的执行过程中，**for** 循环所花时间的界了。

引理 7.1 *当在一个包含 n 个元素的数组上运行 QUICKSORT 时，假设在 PARTITION 的第 4 行中所做比较的次数为 X，那么 QUICKSORT 的运行时间为 $O(n+X)$。*

证明 根据上面的讨论，算法最多对 PARTITION 调用 n 次。每次调用都包括一个固定的工作量和执行若干次 **for** 循环。在每一次 **for** 循环中，都要执行第 4 行。 ■

因此，我们的目标是计算出 X，即所有对 PARTITION 的调用中，所执行的总的比较次数。我们并不打算分析在每一次 PARTITION 调用中做了多少次比较，而是希望能够推导出关于总的比较次数的一个界。为此，我们必须了解算法在什么时候对数组中的两个元素进行比较，什么时候不进行比较。为了便于分析，我们将数组 A 的各个元素重新命名为 z_1，z_2，…，z_n，其中 z_i 是数组 A 中第 i 小的元素。此外，我们还定义 $Z_{ij}=\{z_i, z_{i+1}, \cdots, z_j\}$为 z_i 与 z_j 之间(含 i 和 j)的元素集合。

算法什么时候会比较 z_i 和 z_j 呢？为了回答这个问题，我们首先注意到每一对元素至多比较一次。为什么呢？因为各个元素只与主元元素进行比较，并且在某一次 PARTITION 调用结束之后，该次调用中所用到的主元元素就再也不会与任何其他元素进行比较了。

我们的分析要用到指示器随机变量(见 5.2 节)。定义

$$X_{ij} = \mathrm{I}\{z_i \text{ 与 } z_j \text{ 进行比较}\}$$

其中我们考虑的是比较操作是否在算法执行过程中任意时间发生，而不是局限在循环的一次迭代或对 PARTITION 的一次调用中是否发生。因为每一对元素至多被比较一次，所以我们可以很容易地刻画出算法的总比较次数：

$$X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$$

对上式两边取期望，再利用期望值的线性特性和引理 5.1，可以得到：

$$\mathrm{E}(X) = \mathrm{E}\Big[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\Big] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} \qquad (7.2)$$

上式中的 $\Pr\{z_i$ 与 z_j 进行比较$\}$还需要进一步计算。在我们的分析中，假设 RANDOMIZED-PARTITION 随机且独立地选择主元。

让我们考虑两个元素何时不会进行比较的情况。考虑快速排序的一个输入，它是由数字 1 到 10 所构成(顺序可以是任意的)，并假设第一个主元是 7。那么，对 PARTITION 的第一次调用就将这些输入数字划分成两个集合：{1，2，3，4，5，6}和{8，9，10}。在这一过程中，主元 7 要与所有其他元素进行比较。但是，第一个集合中任何一个元素(例如 2)没有(也不会)与第二个集合中的任何元素(例如 9)进行比较。

通常我们假设每个元素的值是互异的，因此，一旦一个满足 $z_i<x<z_j$ 的主元 x 被选择后，我们就知道 z_i 和 z_j 以后再也不可能被比较了。另一种情况，如果 z_i 在 Z_{ij} 中的所有其他元素之前被选为主元，那么 z_j 就将与 Z_{ij} 中除了它自身以外的所有元素进行比较。类似地，如果 z_j 在 Z_{ij} 中其他元素之前被选为主元，那么 z_j 将与 Z_{ij} 中除自身以外的所有元素进行比较。在我们的例子中，值 7 和 9 要进行比较，因为 7 是 $Z_{7,9}$ 中被选为主元的第一个元素。与之相反的是，值 2 和 9 则始终不会被比较，因为从 $Z_{2,9}$ 中选择的第一个主元为 7。因此，z_i 与 z_j 会进行比较，当且仅当

Z_{ij} 中将被选为主元的第一个元素是 z_i 或者 z_j。

我们现在来计算这一事件发生的概率。在 Z_{ij} 中的某个元素被选为主元之前，整个集合 Z_{ij} 的元素都属于某一划分的同一分区。因此，Z_{ij} 中的任何元素都会等可能地被首先选为主元。因为集合 Z_{ij} 中有 $j-i+1$ 个元素，并且主元的选择是随机且独立的，所以任何元素被首先选为主元的概率是 $1/(j-i+1)$。于是，我们有：

$$\begin{aligned}\Pr\{z_i \text{ 与 } z_j \text{ 进行比较}\} &= \Pr\{z_i \text{ 或 } z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \Pr\{z_i \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &\quad + \Pr\{z_j \text{ 是集合 } Z_{ij} \text{ 中选出的第一个主元}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1} \qquad (7.3)\end{aligned}$$

上式中第二行成立的原因在于其中涉及的两个事件是互斥的。将公式(7.2)和公式(7.3)综合起来，有：

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

在求这个累加和时，可以将变量做个变换($k=j-i$)，并利用公式(A.7)中给出的有关调和级数的界，得到：

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n) \qquad (7.4)$$

于是，我们可以得出结论：使用 RANDOMIZED-PARTITION，在输入元素互异的情况下，快速排序算法的期望运行时间为 $O(n \lg n)$。

Part III. Heap and Heap Sort

1.Heap

1. Defination
   a. **Max-heap** property is that for every node i other than the root

$$A[parent(i)] \geq A[i]$$

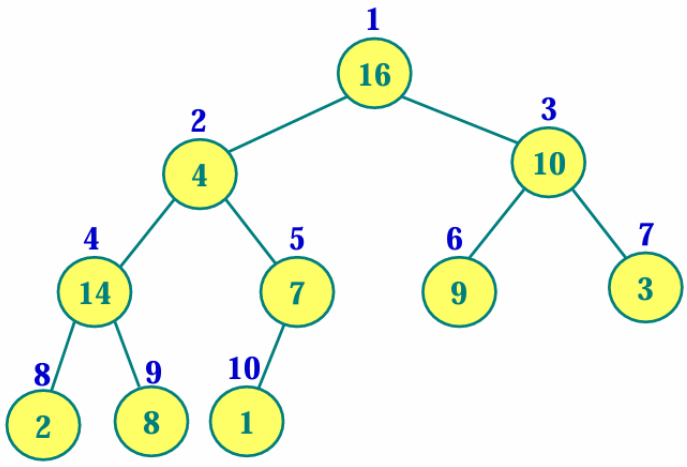
   b. **Min-heap** property is that for every node i other than the root

$$A[parents(i)] \leq A[i]$$

2. Full Binary Tree Implementation（以最小堆为例）

We can use a full binary tree to implement the priority queue.

The full binary tree can be implemented conveniently simply using an array. We set the array 's first element `arr[0]` to be the sentinel, and other elements with the binary tree encoded.



For an element, say its index in the array is `i` . `arr[i]` 's parent is `arr[i/2]` , and the left child(if exist) is `arr[2 * i]` , the right child(if exist) is `arr[2 * i + 1]` .

3. **Heapify Operation** to turn unordered tree into Heap

   Suppose that we get an arbitrary binary tree, we want to convert it to a min-heap or max-heap.How can we operate?

   We can first make sure that subtrees in the bottom are a max-heap, then gradually go up.

   a. **MAX-HEAPIFY(A, i)**

      Suppose that we want to make sure that the element in current index `i` (`arr[i]`) is at the right position of its own subtree, in the other word, **bigger than both of its children.**

      We can do as follows:

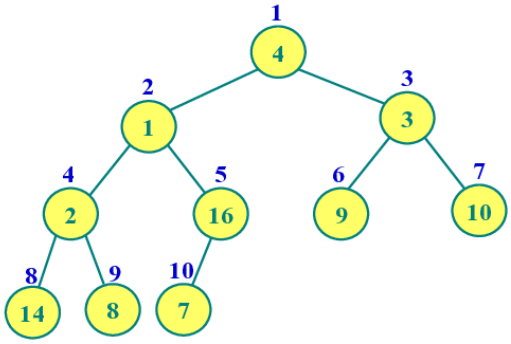
```
MAX-HEAPIFY(A, i)
1    l ← LEFT(i)
2    r ← RIGHT(i)
3    if l ≤ heap-size[A] and A[l] > A[i]
4        then largest ← l
5        else largest ← i
6
7    if r ≤ heap-size[A] and A[r] > A[largest]
8        then largest ← r
9
10   if largest ≠ i
11       then exchange A[i] ↔ A[largest]
12       MAX-HEAPIFY(A, largest)
```

   b. **Build max-heap**

      Notice that all of the leaves now are exactly a max-heap.So we can start from the last but one（倒数第二）layer to start our max-heapify operation.(the element `A[length / 2]`)

Build max-heap(A):

```
1   heap-size[A] ← length[A]
2   for i ←   A ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```



2.Heaplify Operation Complexity Analysis

我们可以用下面的方法简单地估算 BUILD-MAX-HEAP 运行时间的上界。每次调用 MAX-HEAPIFY 的时间复杂度是 $O(\lg n)$，BUILD-MAX-HEAP 需要 $O(n)$ 次这样的调用。因此总的时间复杂度是 $O(n \lg n)$。当然，这个上界虽然正确，但不是渐近紧确的。

我们还可以进一步得到一个更紧确的界。可以观察到，不同结点运行 MAX-HEAPIFY 的时间与该结点的树高相关，而且大部分结点的高度都很小。因此，利用如下性质可以得到一个更紧确的界：包含 n 个元素的堆的高度为 $\lfloor \lg n \rfloor$（见练习 6.1-2），高度为 h 的堆最多包含 $\lceil n/2^{h+1} \rceil$ 个结点（见练习 6.3-3）。

在一个高度为 h 的结点上运行 MAX-HEAPIFY 的代价是 $O(h)$，我们可以将 BUILD-MAX-HEAP 的总代价表示为

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

最后的一个累积和的计算可以用 $x=1/2$ 带入公式（A.8）得到，则有

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

于是，我们可以得到 BUILD-MAX-HEAP 的时间复杂度：

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

因此，我们可以在线性时间内，把一个无序数组构造成为一个最大堆。

3.Prority Queue

1. Defination:

A priority queue is a data structure for maintaining a set of S of elements, each with an associated value called a *key*.

We usually use a heap to implement the priority queue.

2. Basic operations（以最大优先级队列为例）：

📢
- **INSERT(S, x):** inserts the element x into the set S.
- **MAXIMUM(S):** returns the element S with the largest key.
- **EXTRACT-MAX(S):** removes and returns the element S with the largest key.
- **INCREASE-KEY(S, x, k):** increase the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.
- **DECREASE-KEY(S, x, k):** decrease the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.

a. Insert(S, x)

> 将插入的元素x放入完全二叉树的最后一个位置，对于这个位置再进行“向上冒泡”

```
MAX-HEAP-INSERT(A, key)
1. heap-size[A] ← heap-size[A] + 1
2. A[heap-size[A]] ← −∞
3. HEAP-INCREASE-KEY(A, heap-size[A], key)
```

$O(log(N))$

b. Extract-Max(S)

> 返回并且移除根节点，将完全二叉树最后一个元素放入根节点，在对于根节点进行“向下冒泡”（Max-Heapify(S, 0)）

```
HEAP-EXTRACT-MAX(A)
1. if heap-size[A] < 1
2.     then error "heap underflow"
3. max ← A[1]
4. A[1] ← A[heap-size[A]]
5. heap-size[A] ← heap-size[A] − 1
6. MAX-HEAPIFY(A, 1)
7. return max
```

$O(log(N))$

c. Maximum(S)

> 返回根节点即可

d. Increase-Key(S, x, k)

> 将这个元素“向上冒泡”

```
HEAP-INCREASE-KEY(A, i, key)
1. if key < A[i]
2.     then error "new key is smaller than current key"
3. A[i] ← key
4. while i > 1 and A[PARENT(i)] < A[i]
5.     do exchange A[i] ↔ A[PARENT(i)]
6.     i ← PARENT(i)
```

$O(log(N))$

e. Decrease-Key(S, x, k)

> 将这个元素“向下冒泡”,其实就是Max-Heapify(S, k)

4.Heap Sort

初始时候，堆排序算法利用 BUILD-MAX-HEAP 将输入数组 $A[1..n]$建成最大堆，其中 $n=A.length$。因为数组中的最大元素总在根结点 $A[1]$中，通过把它与 $A[n]$进行互换，我们可以让该元素放到正确的位置。这时候，如果我们从堆中去掉结点 n(这一操作可以通过减少 $A.heap\text{-}size$ 的值来实现)，剩余的结点中，原来根的孩子结点仍然是最大堆，而新的根结点可能会违背最大堆的性质。为了维护最大堆的性质，我们要做的是调用 MAX-HEAPIFY(A，1)，从而在 $A[1..n-1]$上构造一个新的最大堆。堆排序算法会不断重复这一过程，直到堆的大小从 $n-1$ 降到 2。(准确的循环不变量定义见练习 6.4-2。)

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

图 6-4 给出了一个在 HEAPSORT 的第 1 行建立初始最大堆之后，堆排序操作的一个例子。图 6-4 显示了第 2～5 行 **for** 循环第一次迭代开始前最大堆的情况和每一次迭代之后最大堆的情况。

HEAPSORT 过程的时间复杂度是 $O(n\lg n)$，因为每次调用 BUILD-MAX-HEAP 的时间复杂度是 $O(n)$，而 $n-1$ 次调用 MAX-HEAPIFY，每次的时间为 $O(\lg n)$。

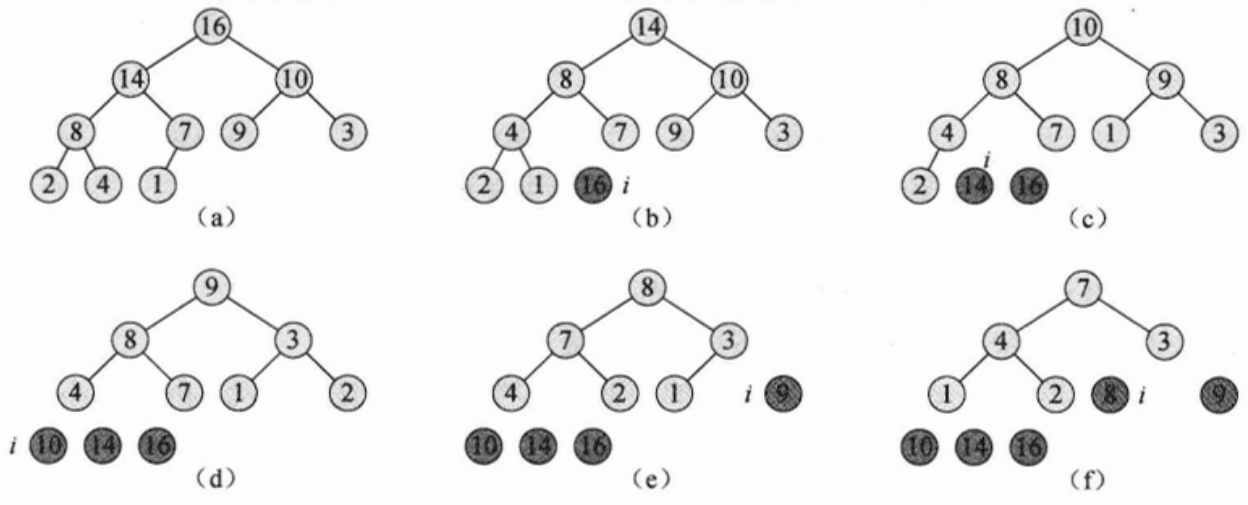


图 6-4 HEAPSORT 的运行过程。(a)执行堆排序算法第 1 行，用 BUILD-MAX-HEAP 构造得到的最大堆。(b)～(j)每次执行算法第 5 行，调用 MAX-HEAPIFY 后得到的最大堆，并标识当次的 i 值。其中，仅仅浅色阴影的结点被保留在堆中。(k)最终数组 A 的排序结果

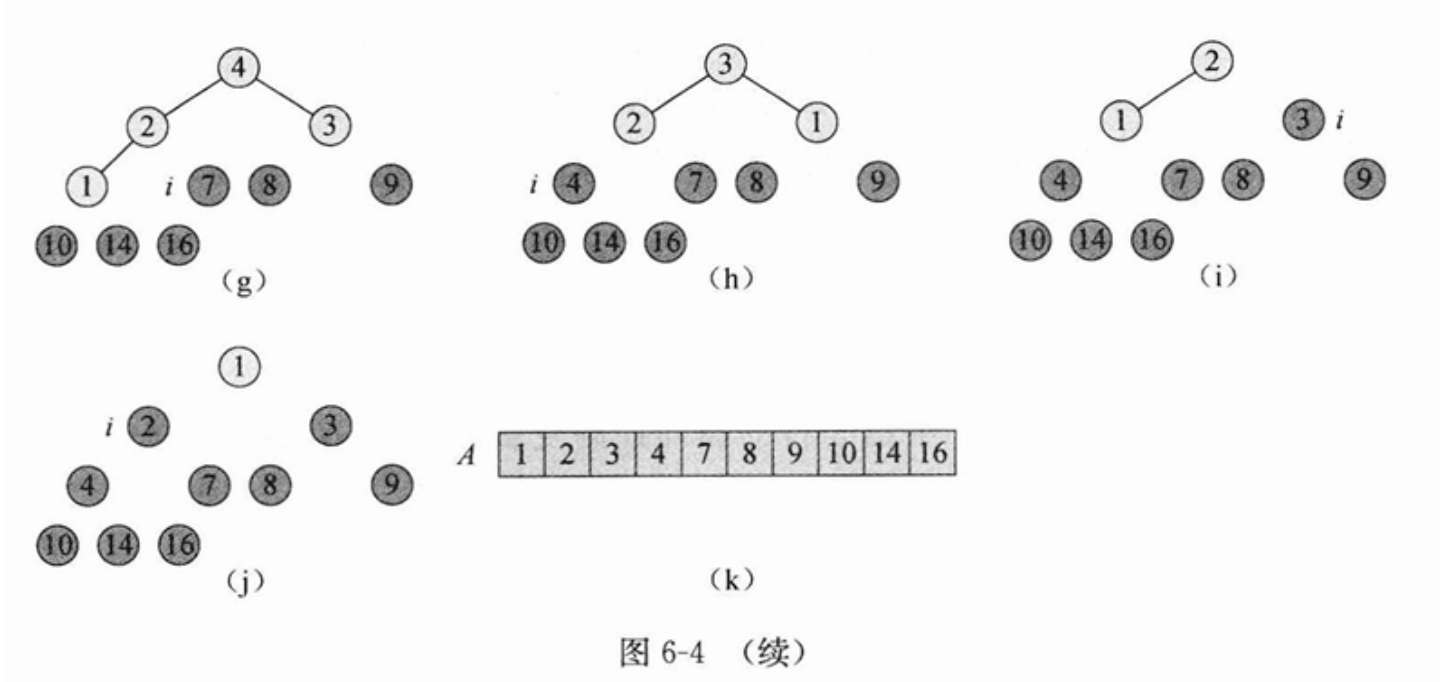


图 6-4 （续）

关键：建堆和清空堆的时间复杂度

虽然说堆插入和删除的时间复杂度都是 $O(lg(N))$,但是对于这n个元素，总共的插入的复杂度（建堆）和删除的复杂度（清空堆）的时间复杂度却是不同的。

直观的理解是插入时实际的操作是从下往上冒泡，路径长度也就是最后所到位置的子树的高度，路径长较短。而删除时候的操作是把最后一个元素放到根节点向下冒泡，所需时间较长。

所以说建堆是 $O(N)$,而清空堆是 $O(Nlg(N))$ 。堆排序的复杂度是 $O(Nlg(N))$

实现自己的堆

Part IV. Comparison Sort Summary

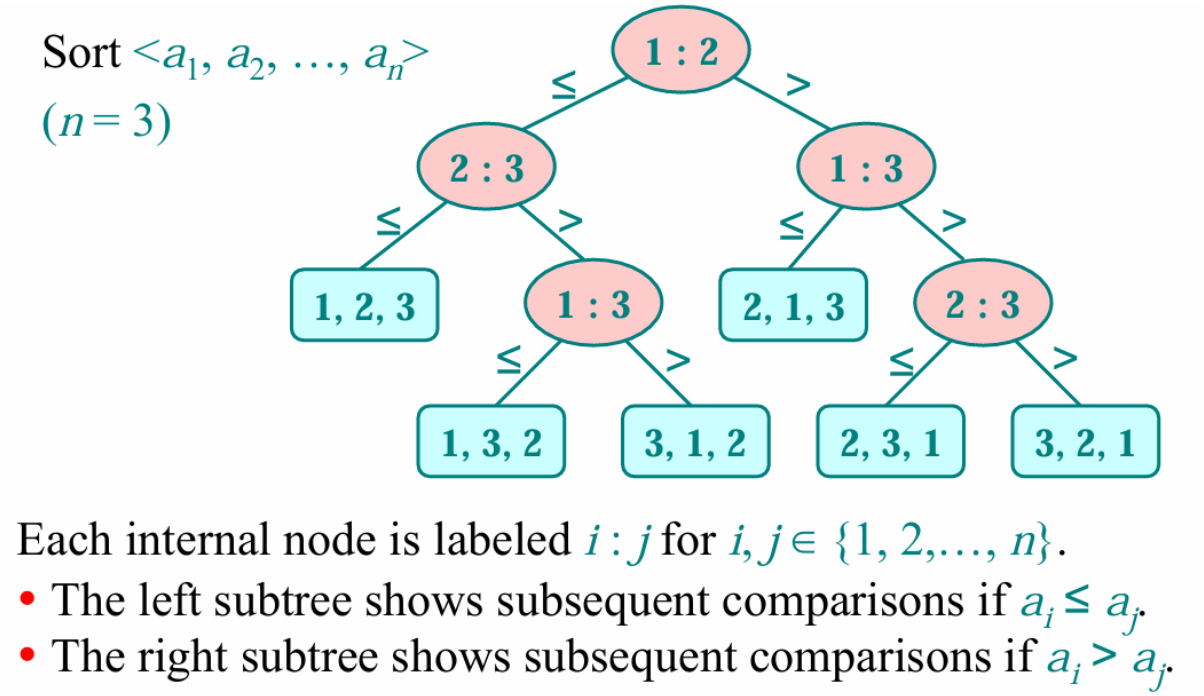
1.Different comparion sort algorithms comparison

Running time	Worst-case	Average-case	In place
Heap sort	*nlgn*	*nlgn*	*Yes*
Quick sort	n^2	*nlgn*	*Yes*
Insertion sort	n^2	n^2	*Yes*
Merge sort	*nlgn*	*nlgn*	*No*

2.Complexity Lower Bound of Comparion Sorts

- All of our algorithms above used comparisons.
- We will prove that all of our algorithms above have a running time lower bound $\Omega(nlgn)$.

1. Sort using Decision Tree



Each leaf contains a permutation $< \pi(1), \pi(2), \pi(3), ...\pi(n) >$ to indicate that the ordering $< a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)} >$ has been established.

The complexity of the sort algorithm is just the height of the Decision Tree.

Suppose that the tree's height is h, the tree of height h can have at most 2^h leaves, while the decision tree needs $n!$.

$$
\begin{aligned}
2^h &\geq n! \\
h &\geq lg(n!) \\
&= lg(n) + lg(n-1) + lg(n-2) + ... + lg2 + lg1 \\
&\geq lg(n) + lg(n-1) + lg(n-2) + ... + lg(\frac{n}{2}) \\
&= \frac{n}{2} lg(\frac{n}{2}) \\
&= \Omega(nlg(n))
\end{aligned}
$$

Part V. Sorting in linear time

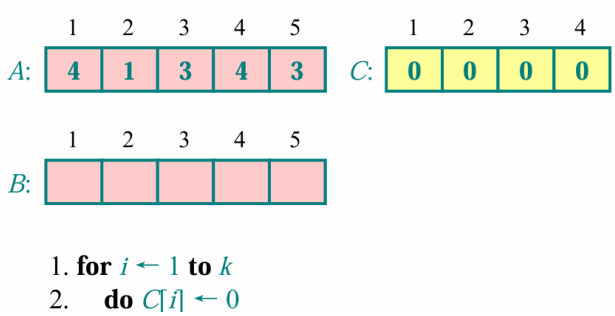
1.Counting Sort

> Input: A[1 ⋯ n], where A[j] ∈ {1, 2, ⋯, k}. Output: B[1 ⋯ n], sorted.
>
> Auxiliary storage: C[1 ⋯ k].

1. Operation Process
   - Loop1:

代码块

```
1   for i <- i to k:
2       do C[i] <- 0
```

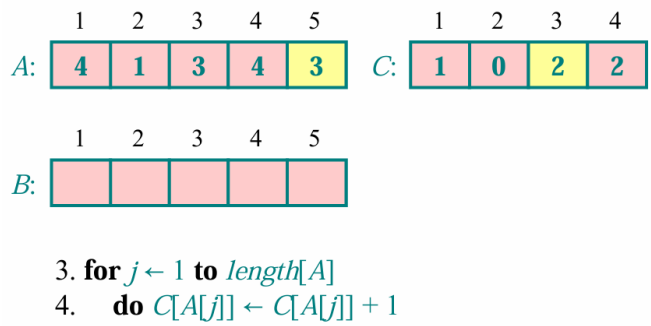


1. **for** $i \leftarrow 1$ **to** k
2. **do** $C[i] \leftarrow 0$

- Loop2:

代码块

```
1   for j <- 1 to length[A]
2       do C[A[j]] <- C[A[j]] + 1
```

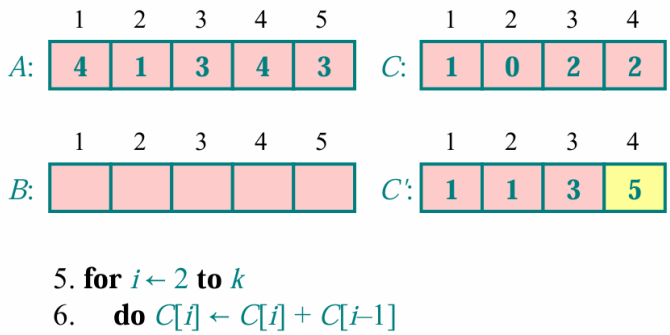


3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$

- Loop 3:

代码块

```
1   for i <- 2 to k
2       do C[i] = C[i] + C[i - 1]
```

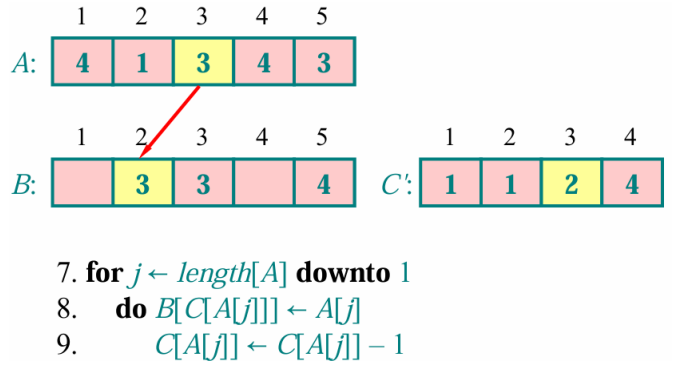


5. **for** $i \leftarrow 2$ **to** k
6. **do** $C[i] \leftarrow C[i] + C[i–1]$

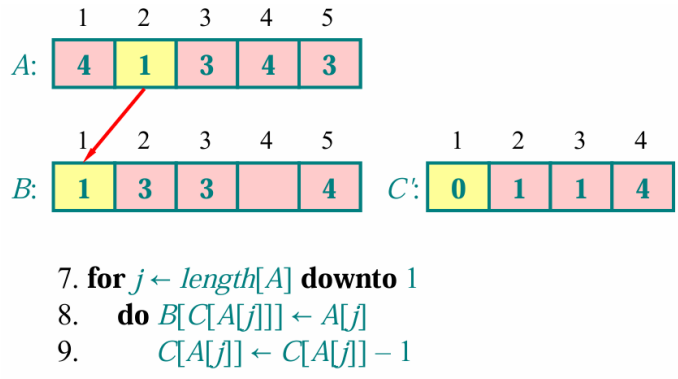
- Loop 4:

代码块

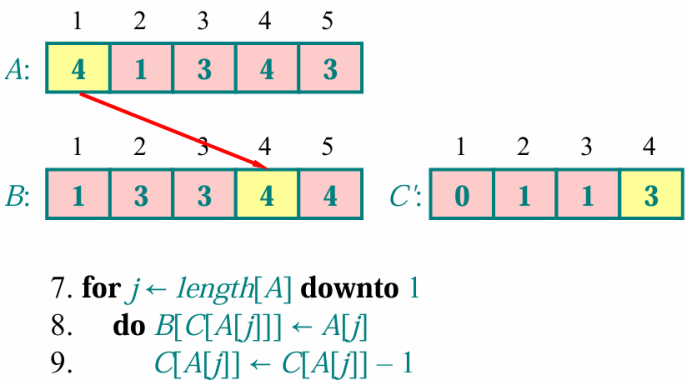
```
for j <- length[A] downto 1
    do B[C[A[j]]] <- A[j]
        C[A[j]] <- C[A[j]] - 1
```



7. **for** $j \leftarrow length[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$



7. **for** $j \leftarrow length[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$



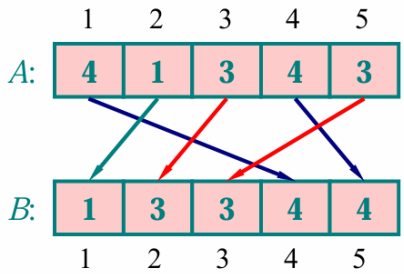
7. **for** $j \leftarrow length[A]$ **downto** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

2. Complexity Analysis

```
COUNTING-SORT(A, B, k)
1. for i ← 1 to k                          } Θ(k)
2.     do C[i] ← 0
3. for j ← 1 to length[A]                  } Θ(n)
4.     do C[A[j]] ← C[A[j]] + 1
5. for i ← 2 to k                          } Θ(k)
6.     do C[i] ← C[i] + C[i–1]
7. for j ← length[A] downto 1              } Θ(n)
8.     do B[C[A[j]]] ← A[j]
9.        C[A[j]] ← C[A[j]] – 1
                                           Θ(n + k)
```

3. Stabliblity

Counting sort is a ***stable*** sort: it preserves the input order among equal elements.

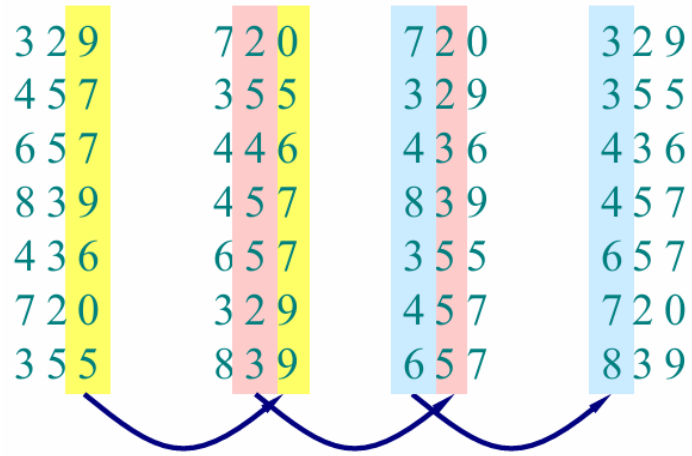


2.Radix Sort

> Digit-by-digit sort.

Hollerith’ s original (bad) idea: sort on most significant digit first.

Good idea: Sort on least-significant digit first with auxiliary stable sort. （counting sort one by one)



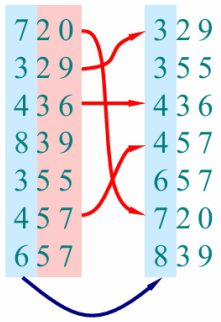
Correctness of radix sort

Induction on digit position

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t.

Two numbers that differ in digit t are correctly sorted.

Two numbers equal in digit t are put in the same order as the input ⇨ correct order.



RADIX-SORT(A, d)
1. **for** $i \leftarrow 1$ **to** d
2. **do** use a stable sort to sort array A on digit i

$\Theta(d(n+k))$ Each digit can take on up to k possible values

when d is constant and $k = O(n)$ ⇨ $\Theta(n)$

📢 Stability of Quick Sort:

- Insertion sort is stable
- Merge sort is stable
- Heap sort is not stable
- Quick sort is not stable