HW4

6.4

P90 6.4：2

6.4-2 试分析在使用下列循环不变量时，HEAPSORT 的正确性：

在算法的第 2～5 行 **for** 循环每次迭代开始时，子数组 $A[1..i]$是一个包含了数组$A[1..n]$中第 i 小元素的最大堆，而子数组 $A[i+1..n]$包含了数组 $A[1..n]$中已排序的$n-i$个最大元素？

```
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)
```

开始时，i=A.len，A[i+1..n]无元素，循环不变式成立。

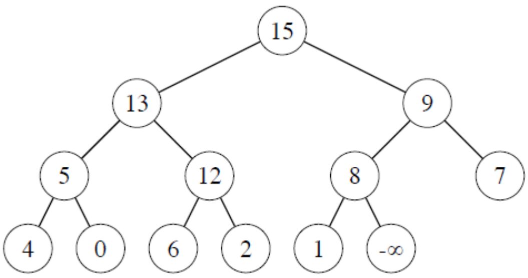
假设i=k时循环不变式成立，下面证i=k-1时仍成立（k=A.len..3）：

i=k循环开始，由假设，A[1..k]是包含全局第k小元素的max-heap，A[k+1..n]是n-k个全局最大元素。全局第k小元素是A[1..k]中最大元素，为max-heap根A[1]。i=k循环过程中，A[1]和A[k]交换，然后重新调整A[1..k-1]为max-heap。故i=k-1循环开始，A[k,n]是n-k+1个全局最大元素，A[1..k-1]是max-heap，A[1]是全局第k-1小元素，循环不变式仍成立。得证。

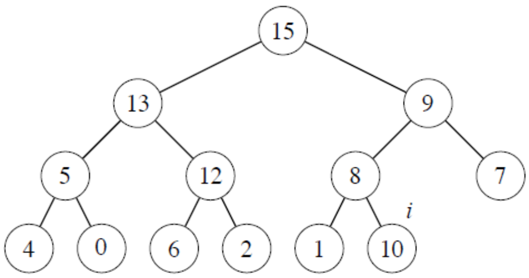
6.5

P92 6.5：2，6

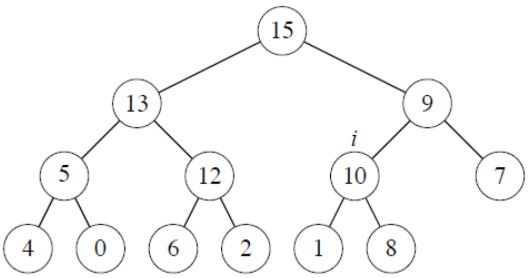
6.5-2 试说明 MAX-HEAP-INSERT(A，10)在堆 $A=\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\rangle$上的操作过程。



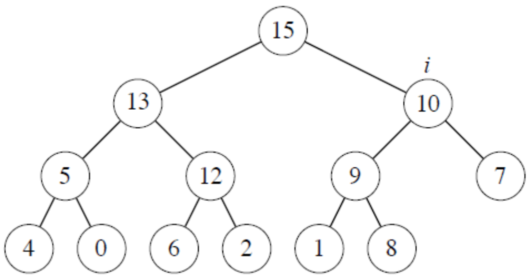
(a)



(b)



(c)



(d)

6.5-6 在 HEAP-INCREASE-KEY 的第 5 行的交换操作中，一般需要通过三次赋值来完成。想一想如何利用 INSERTION-SORT 内循环部分的思想，只用一次赋值就完成这一交换操作？

HEAP-INCREASE-KEY(A, i, key)

1 **if** $key < A[i]$

2   **error** "new key is smaller than current key"

3 $A[i] = key$

4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$

5   exchange $A[i]$ with $A[\text{PARENT}(i)]$

6     $i = \text{PARENT}(i)$

```
def HeapIncreaseKey(A,i,key):
    # increase A[i] to key

    if A[i]>=key:
        return

    parentI = Tree.Parent(i)
    while parentI>=0 and A[parentI]<=key:
        # 一次赋值
        A[i] = A[parentI]
        i = parentI
        parentI = Tree.Parent(i)
    # 最后记得放key
    A[i] = key
    return A
```

7.1

P97 7.1：4

7.1-4 如何修改 QUICKSORT，使得它能够以非递增序进行排序？

升序排的Partition()：

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p-1
3  for j = p to r-1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i+1] with A[r]
8  return i + 1
```

第4行改为if A[j]>=x即可。

7.2

P100 7.2：3，5

7.2-3 证明：当数组 A 包含的元素不同，并且是按降序排列的时候，QUICKSORT 的时间复杂度为 $\Theta(n^2)$。

A开始时已有序，按书上算法选A[r]作划分元，将导致最坏划分，n个元素划分后一组n-1，一组0元素。$T(n)=T(n-1)+T(0)+\Theta(n)$，$T(n)=\Theta(n^2)$。

7.2-5 假设快速排序的每一层所做的划分的比例都是 $1-\alpha:\alpha$，其中 $0<\alpha\leqslant 1/2$ 且是一个常数。试证明：在相应的递归树中，叶结点的最小深度大约是 $-\lg n/\lg\alpha$，最大深度大约是 $-\lg n/\lg(1-\alpha)$（无需考虑整数舍入问题）。

$0<\alpha\le\frac{1}{2}$，则$1-\alpha\ge\alpha$，α分支最早结束，对应最小深度k；$1-\alpha$分支最晚结束，对应最大深度l。

$n\alpha^k\le 1$，$k\ge -\lg n/\lg\alpha$，即最小深度。同理，最大深度$-\lg n/\lg(1-\alpha)$。

8.1

P108 8.1：3

8.1-3 证明：对 $n!$ 种长度为 n 的输入中的至少一半，不存在能达到线性运行时间的比较排序算法。如果只要求对 $1/n$ 的输入达到线性时间呢？$1/2^n$ 呢？

这种的“一半，1/n，$1/2^n$”都是针对n!种排序结果。输入长度为n，输出结果有$m = n!/2$，$n!/n$，$n!/2^n$时，都不存在线性时间的比较排序算法。证明：设比较排序等价的决策树高h，则算法代价$\Omega(h)$。m是最后一层节点数，故$2^h \geq m$，$h \geq \lg m = \Omega(n \lg n)$。所以对所给m，算法代价$\Omega(n \lg n)$，不存在线性时间算法。

$\lg m = \Omega(n \lg n)$是因为：

$$\begin{aligned}\lg \frac{n!}{2} &= \lg n! - 1 \geq n \lg n - n \lg e - 1 \\ \lg \frac{n!}{n} &= \lg n! - \lg n \geq n \lg n - n \lg e - \lg n \\ \lg \frac{n!}{2^n} &= \lg n! - n \geq n \lg n - n \lg e - n\end{aligned}$$

8.2

P110 8.2：4

8.2-4 设计一个算法，它能够对于任何给定的介于 0 到 k 之间的 n 个整数先进行预处理，然后在 $O(1)$时间内回答输入的 n 个整数中有多少个落在区间$[a..b]$内。你设计的算法的预处理时间应为 $\Theta(n+k)$。

预处理：对n个整数作counting sort，得前缀和数组C[0..k]，C[i] = #(n个数中小于等于i的数)。$\Theta(n+k)$时间。

#(落在[a..b]内的数) = C[b]-C[a-1]。O(1)时间。

HW5

8.3

P112 8.3：4

8.3-4 说明如何在 $O(n)$时间内，对 0 到 n^3-1 区间内的 n 个整数进行排序。

（1）将n个数转成n进制：$n^3 - 1$转为n进制共三位(n-1,n-1,n-1)，所以n个数中的任意一个数最多需要运算3次。这一步的复杂度O(3n)。

（2）对这n个三位数使用radix sort：每一位排序采用counting sort，每一位取值范围0..n-1，共三位，故这一步的复杂度O(3(n+n))。

总复杂度O(9n)=O(n)。

更一般的思路 by #151

8.3-4　对于本题，输入规模：n

最大数据为 n^3-1，~~数位~~ 用二进制表示数位为 $\lceil \lg(n^3-1) \rceil = \lceil 3\lg n \rceil$

使用基数排序，由引理8.4　$b = \lceil 3\lg n \rceil = \cancel{\Theta(\lg n)} \geq \lfloor \lg n \rfloor$

因此 r 取 $\lfloor \lg n \rfloor$

$\Rightarrow$ 时间复杂度为 $\Theta((b/r)(n+2^r)) = \Theta\left(\frac{3\lg n}{\lg n}(n+n)\right) = \Theta(n)$

具体方法为将这些数当作 $\underbrace{2^{\lfloor \lg n \rfloor}}_{= n}$ 进制的数使用基数排序，稳定排序算法采用计数排序.

8.4

P114 8.4：2

8.4-2 解释为什么桶排序在最坏情况下运行时间是 $\Theta(n^2)$？我们应该如何修改算法，使其在保持平均情况为线性时间代价的同时，最坏情况下时间代价为 $O(n\lg n)$？

Bucket sort最坏情况：所有数据集中在一个桶，bucket sort退化为insert sort，最坏时间O(n^2)。

改进：桶内排序由insert sort改为最坏$O(n\lg n)$的排序算法，如merge/heap sort。

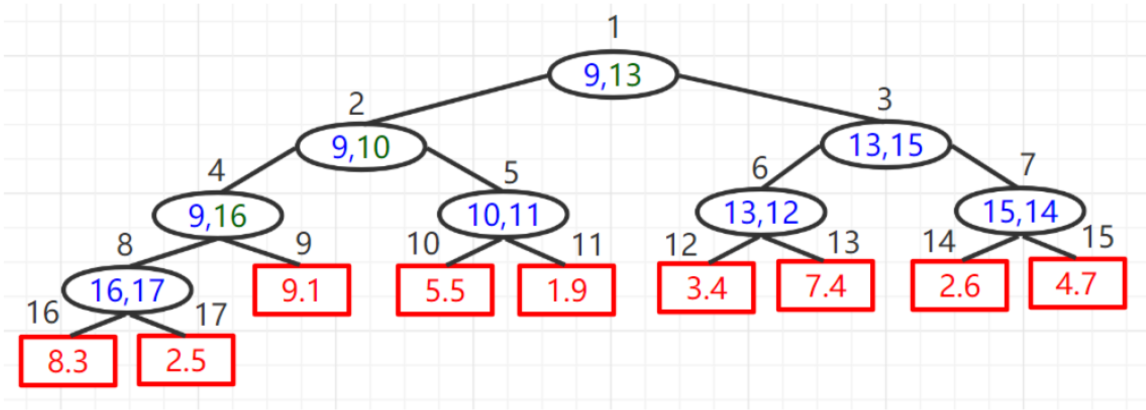
9.1

P120 9.1：1

9.1-1 证明：在最坏情况下，找到 n 个元素中第二小的元素需要 $n+\lceil \lg n \rceil - 2$ 次比较。（提示：可以同时找最小元素。）

类比课上讲的Max_SecondMax()算法：

算法：A[1..n]，利用二叉树找出A的max和secondMax。



构建二叉树：非叶节点数=n-1，总节点数m=2n-1

从尾节点往前对节点赋值：A元素占据全部叶节点，非叶节点记录下标对。

找到max：根节点下标对的第一个下标就是maxId。至此，除根节点外每两个节点要比较一次，共(m-1)/2 = n-1次。

找secondMax：secondMax一定出自max击败的节点，如例子中13,10,16号节点。二叉树高度$h = \lfloor \log_2 m \rfloor$，找secondMax共比较$\Theta(h)$次。

总比较次数$= n - 1 + \Theta(h) = \Theta(n)$。

得最坏比较次数=$n + ceil(\lg n) - 2$的Min_SecondMin()算法。

> 严谨地，还应该证明最坏情况至少要这么多次比较。这是否成立暂时未知，如果不成立，最坏比较次数可能更小，题目描述就应该改为：给出一个最坏比较次数是…的算法。

9.2

P123 9.2：3

9.2-3 给出 RANDOMIZED-SELECT 的一个基于循环的版本。

原始的random select程序是递归版的：

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q-p+1
if i == k          // the pivot value is the answer
    return A[q]
else if i < k
    return RANDOMIZED-SELECT(A, p, q-1, i)
else return RANDOMIZED-SELECT(A, q+1, r, i-k)
```

改为循环的，就是改成非递归版：

```python
def RandomSelect(A,p,r,i):
    while True:
        # p==r检查要放到循环内
        if p==r:
            return A[p]

        q = RandomPartition(A,p,r)
        k = q-p+1
        if i==k:
            return A[q]
        elif i<k:
            r = q-1
        else:
            i-=k
            p = q+1
```

按书上的算法，当初始的r-p+1<i时，最终p==r，i>k，算法直接返回A[p]，所以非递归版p==r的检查要放到循环内。

或者，一开始就将r-p+1<i的情况除外，这样p==r时，i==k，可省略p==r检查：

```python
def RandomSelect(A,p,r,i):
    if r-p+1<i:
        raise Exception("len(A)<i")
    while True:
        q = RandomPartition(A,p,r)
        k = q-p+1
        if i==k:
            return A[q]
        elif i<k:
            r = q-1
        else:
            i-=k
            p = q+1
```

9.3

P124 9.3：5

9.3-5　假设你已经有了一个最坏情况下是线性时间的用于求解中位数的“黑箱”子程序。设计一个能在线性时间内解决任意顺序统计量的选择问题算法。

如果采用 *9.3 最坏情况为线性时间的选择算法*，虽然能在线性时间内解决顺序统计量的问题，但没体现题中所给的中位数黑盒程序的作用。所以，题目所指的算法应该是：

```
SELECT'(A, p, r, i)
if p = r
   then return A[p]
x ← MEDIAN(A, p, r)
q ← PARTITION(x)
k ← q − p + 1
if i = k
   then return A[q]
elseif i < k
   then return SELECT'(A, p, q − 1, i)
else return SELECT'(A, q + 1, r, i − k)
```

就是将random select的random partition，改为中位数作划分元。因为中位数是有序数组中间的数，故T(n)=T(n/2)+O(n)=O(n)。