hw1

2.1-1

31	41	59	26	41	58
31	41	59	26	41	58
31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

2.1-3

在每次循环体的迭代中，进入循环时的不变条件是：不存在索引 k < j，使得 A[k] = v。为了继续下一次循环迭代，需要确保对于当前的 j 值，A[j] ≠ v。如果循环通过第 5 行退出，那么在前一行刚刚在 i 中放入了一个可接受的值。如果循环通过耗尽所有可能的 j 值而退出，那么没有任何索引具有值 j，因此将 NIL 放入 i 是正确的。

```
function FindIndex(A, v):
    i = NIL
    j = 0
    while j < length(A):
        if A[j] == v:
            i = j
            break
        j = j + 1
    return i
```

2.2-2

伪代码：

```
function SelectionSort(A):
    for i = 1 to n - 1 do
        min = i
        for j = i + 1 to n do
            // 寻找第 i 个最小元素的索引
            if A[j] < A[min] then
                min = j
            end if
        end for
        Swap A[min] and A[i]
    end for
```

在 lines 1 到 10 的 for 循环的每次迭代中，子数组 A[1::i − 1] 包含 A 中的 i − 1 个最小元素，并以升序排列。经过 n − 1 次循环迭代后，A 中最小的 n − 1 个元素以升序排列在数组的前 n − 1 个位置上，因此第 n 个元素必然是最大的元素。因此不需要再次运行循环。选择排序的最佳情况和最坏情况运行时间都是 Θ(n^2)。这是因为无论元素的初始排列如何，在主要的 for 循环的第 i 次迭代中，该算法总是检查剩余的 n−i 个元素以找到最小的剩余元素。

2.2-3

平均需要检查输入序列元素个数（期望）：

$$E(steps) = A.length(1-p)^{A.length} + \sum_{k=1}^{A.length} k(1-p)^{k-1}p$$

最坏情况是要检查所有可能位置，需要Θ(A.length).

证明：

$$\begin{aligned}
\mathbf{E}(steps) &= A.length(1-p)^{A.length} + \sum_{k=1}^{A.length} k(1-p)^{k-1}p \\
&= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length}\sum_{k=s}^{A.length} (1-p)^{k-1}p \\
&= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} p \sum_{k=s}^{A.length} (1-p)^{k-1} \\
&= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} p\frac{(1-p)^{s-1}-(1-p)^{A.length}}{1-(1-p)} \\
&= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} (1-p)^{s-1}-(1-p)^{A.length} \\
&= A.length(1-p)^{A.length} + \sum_{s=1}^{A.length} (1-p)^{s-1} - \sum_{s=1}^{A.length} (1-p)^{A.length} \\
&= \sum_{s=1}^{A.length} (1-p)^{s-1} \\
&= \frac{1-(1-p)^{A.length}}{1-(1-p)} \\
&= \frac{1}{p} - \frac{(1-p)^{A.length}}{p}
\end{aligned}$$

在数组中确切存在一个你要查找的元素，然后每个位置等可能地包含该元素。在这种情况下，最坏情况行为不会改变，期望运行时间是

$$\sum_{i=1}^{A.length} i/A.length = (A.length + 1)/2$$

这使得期望情况的渐近复杂度为 Θ(A.length)。

2.3-2

```
Merge(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1..n1] and R[1..n2] be new arrays
for i = 1 to n1 do
    L[i] = A[p + i - 1]
end for
for j = 1 to n2 do
    R[j] = A[q + j]
end for
i = 1
j = 1
k = p
while i ≠ n1 + 1 and j ≠ n2 + 1 do
    if L[i] ≤ R[j] then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    end if
    k = k + 1
end while
if i == n1 + 1 then
    for m = j to n2 do
        A[k] = R[m]
        k = k + 1
    end for
end if
if j == n2 + 1 then
    for m = i to n1 do
        A[k] = L[m]
        k = k + 1
    end for
end if
```

2.3-6

使用二分查找并不会改善最坏情况下的运行时间。插入排序需要将大于关键值的每个元素都复制到数组中的相邻位置。进行二分查找可以告诉我们需要复制多少个元素，但无法消除必须进行的复制操作。