8.4 When implementing Insertion Sort, a binary search could be used to locate the position within the first $i-1$ elements of the array into which element $i$ should be inserted. Why would using such a binary search not speed up the asymptotic running time for Insertion Sort?

Although the position to insert could be found in $\Theta(\log(i))$, shifting the elements to make room for the insert will still require $\Theta(i)$ — the same as the original algorithm. So the overall runtime will (up to big-$\Theta$) be unchanged.

8.5 Figure 8.5 shows the best-case number of swaps for Selection Sort as $\Theta(n)$. This is because the algorithm does not check to see if the $i$th record is already in the $i$th position; that is, it may perform unnecessary swaps.

(a) Modify the algorithm so that it does not make unnecessary swaps.

The original selection sort algorithm is

```c
int minSubscript;
for (int i = 0; i < n-1; i++) {
    minSubscript = i;
    for (int j = i+1; j < n; j++)
        if (list[j] < list[minSubscript])
            minSubscript = j;
    swap(list[i], list[minSubscript]);
}
```

We can eliminate unnecessary swaps by testing whether $i$ is the same as $\text{minSubscript}$:

```c
int minSubscript;
for (int i = 0; i < n-1; i++) {
    minSubscript = i;
    for (int j = i+1; j < n; j++)
        if (list[j] < list[minSubscript])
            minSubscript = j;
    if (i != minSubscript)
        swap(list[i], list[minSubscript]);
}
```

(b) What is your prediction regarding whether this modification actually improves running time?
Asymptotically, there will be no difference. In the asymptotic analysis, the principle contribution to the run time is the inner for loop, which is still \( \Theta(n - i) \). In practice lists for which few swaps are required might see a slight improvement in overall runtime, while lists for which many swaps are required might see an even slighter degradation in run time.

8.8 Assume \( L \) is an array, \( L\.length \) returns the number of records in the array, and \( \text{qsort}(L, i, j) \) sorts the records of \( L \) from \( i \) to \( j \) using the Quicksort algorithm (that we discussed in class). What is the average-case time complexity for each of the following code fragments?

(a) \[
\text{for (i = 0; i < L\.length; i++)}
\text{\hspace{1em} \text{qsort}(L, 0, i)}
\]

The implementation of quicksort we discussed in class chooses the pivot to be the larger of the first two distinct elements in the list. Since \( L[0:i-1] \) is sorted, if we assume the elements of the list are all distinct, this will result in choosing the pivot to be the second or third smallest element in \( L_{previous\_pivot:i} \). (Whether it's the second or third depends on the value of \( L[i] \).) So each second recursive call will be on a list of length \( previous\_length - 1 \) or \( previous\_length - 2 \). In either case the height of the tree of recursive calls will be \( \Theta(i) \). So each call in the for loop, \( \text{qsort}(L, 0, i) \), will be \( \Theta(i^2) \), and the total run time will be

\[
\sum_{i=0}^{n-1} i^2, \text{ which is } \Theta(n^3).
\]

(Here \( n = L\.length \).) Note that as long as all of the elements of \( L \) are distinct, this analysis applies regardless of the original order. So our runtime is best case, average case, and worst case.

(b) \[
\text{for (i = 0; i < L\.length; i++)}
\text{\hspace{1em} \text{qsort}(L, i, L\.length-1)}
\]

In this case we sort the entire list in the first call to \( \text{qsort} \), and all of the subsequent calls will call \( \text{qsort} \) on a sorted list. So the total run time will be

\[
n \log(n) + \sum_{i=1}^{n-1} (n - i)^2, \text{ which is } \Theta(n^3).
\]

Once again, if we assume that all of the elements of the list are distinct, this is best case, average case, and worst case.

8.10 Graph \( f_1(n) = n \log(n), f_2(n) = n^{1.5}, \) and \( f_3(n) = n^2 \) in the range \( 1 \leq n \leq 1000 \) to visually compare their growth rates. Typically, the constant factor in the running-time expression for an implementation of Insertion Sort will be less than the constant factors for Shellsort or Quick sort. How many times greater can the constant factor be for Shellsort to be faster than Insertion Sort when \( n = 1000 \)? How many times greater can the constant factor be for Quick sort to be faster than Insertion sort when \( n = 1000 \).

All three functions are increasing and concave up on the interval \([1, 1000]\). Furthermore \( f_1(n) < f_2(n) \leq f_3(n) \) on the interval.
If $T_1(n)$ is the average-case runtime of quicksort, $T_2(n)$ is the worst-case runtime of shellsort, and $T_3(n)$ is the worst-case runtime of insertion sort, we assume that

$$T_1(n) \approx c_1 f_1(n), T_2(n) \approx c_2 f_2(n), \text{ and } T_3(n) \approx c_3 f_3(n),$$

for large $n$ and some positive constants $c_1$, $c_2$, and $c_3$. So in order for Shellsort to be faster than Insertion sort when $n = 1000$, we want

$$1 > \frac{T_3(1000)}{T_2(1000)} \approx \frac{c_3 1000^2}{c_2 1000^{1.5}}.$$

Simplifying we get

$$1 > \frac{c_3 \sqrt{1000}}{c_2}$$

or

$$c_2 > c_3 \sqrt{1000}.$$ 

So as long as $c_2$ is at least $\sqrt{1000} \approx 31.6$ times bigger than $c_3$, we’d expect shellsort to be faster than insertion sort.

Similar reasoning applies for quicksort and insertion sort. Here we want

$$1 > \frac{T_3(1000)}{T_1(1000)} \approx \frac{c_3 1000^2}{c_1 1000 \log(1000)}.$$ 

Simplifying we get

$$1 > \frac{1000 c_3}{\log(1000) c_1}$$

or

$$c_1 > \frac{1000}{\log(1000)} c_3.$$ 

So as long as $c_1$ is at least $\frac{1000}{\log(1000)} \approx 100$ times bigger than $c_3$, we’d expect quicksort (on average) to be faster than insertion sort.