Chapter 2. Getting Started

2.1-1.
$A = \{31, 41, 59, 26, 41, 58\}$
$\Rightarrow A = \{26, 31, 41, 59, 41, 58\}$
$\Rightarrow A = \{26, 31, 41, 41, 59, 58\}$
$\Rightarrow A = \{26, 31, 41, 41, 58, 59\}$

2.1-2.

Algorithm 1: Insertion sort(A), nonincreasing order

```
for j = 2 to A.length do
    key = A[j];
    // Insert A[j] into the sorted sequence A[1, ..., j - 1];
    i = j - 1;
    while i = 0 and A[i] < key do
        A[i + 1] = A[i];
        i = i - 1;
    end
    A[i + 1] = key;
end
```

2.1-3.

Algorithm 2: Linear search(A, ν)

```
for j = 1 to A.length do
    if A[j] = ν then
        return j
    end
end
return NIL
```

The loop invariant of this algorithm is as follows:
At the start of each iteration of the for loop, the subarray $A[1, \cdots, j-1]$

does not contain ν. The three necessary properties of the loop invariant can be proved as:
Initialization: This is trivial, since nonempty array does not contain anything.
Maintenance: Before the loop, the subarray $A[1, \cdots, j-1]$ does not contain ν. If $A[j] = \nu$, the algorithm returns and the loop breaks. Otherwise, $A[1, \cdots, j]$ does not contain ν, maintaining the loop invariant.
Termination: Finally, once the loop is completed, A does not contain ν, so NIL should be returned.

2.1-4.

Algorithm 3: Add two binary integers(A, B)

```
carry = 0;
let C be a new array with (n + 1) element;
for j = A.length to 0 do
    C[j+1] = A[j] + B[j] + carry;
    carry = (A[j] + B[j] + carry) / 2;
end
C[0] = carry;
return C
```

2.2-1.
$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$.

2.2-2.

Algorithm 4: Selection sort(A)

```
for i = 1 to A.length − 1 do
    min = ∞, index = i + 1;
    for j = i + 1 to A.length do
        if min > A[j] then
            min = A[j], index = j;
        end
    end
    swap(A[j], A[index]);
end
```

The loop invariant of selection sort is that, at the start of each iteration of the for loop, the subarray $A[1, \cdots, j-1]$ consists of the least $j-1$ elements of A. It doesn't need to run for all n elements since after $n-1$ iteration it will be already guaranteed that $A[n]$ is larger than any other elements in $A[1, \cdots, n-1]$. The performance for the best case and the worst case are both $\Theta(n^2)$.

2.2-3.
$n/2$ elements are needed to be checked on the average. In worst case n checks are required. They're both $\Theta(n)$.

2.2-4.
Making loop invariant to satisfy algorithm requirement would decrease the best-case running time.

2.3-1.
$A = \{3, 41, 52, 26, 38, 57, 9, 49\}$
$\Rightarrow A = \{3, 41, 26, 52, 38, 57, 9, 49\}$
$\Rightarrow A = \{3, 26, 41, 52, 9, 38, 49, 57\}$
$\Rightarrow A = \{3, 9, 26, 38, 41, 49, 52, 57\}$

2.3-2.

Algorithm 5: Merge(A, p, q, r)

```
n_1 = q - p + 1;
n_2 = r - q;
let L[1, ..., n_1] and R[1, ..., n_1] be new arrays;
for i = 1 to n_1 do
    L[i] = A[p + i - 1];
end
for j = 1 to n_2 do
    R[i] = A[q + j];
end
i = 1, j = 1;
for k = p to r do
    if L[i] < R[j] or j = n_2 + 1 then
        A[k] = L[i];
        i = i + 1;
    else
        A[k] = R[j];
        j = j + 1;
    end
end
```

2.3-3.
For $n = 2$, $T(n) = 2$.
Suppose the statement holds for $n \le 2^k$. Then for $n = 2^{k+1}$, $T(n) = 2 \cdot (2^k \cdot k) + 2^{k+1} = 2^{k+1} \cdot (k+1)$.

2.3-4.
The recurrence formula for the worst-case running time of the modified version of insertion sort is $T(n) = T(n-1) + (n-1)$.

2.3-5.

Algorithm 6: BinarySearch(A, p, q, ν)

```
if p > q then
  | return;
m = (p + q)/2;
if A[m] > ν then
  | BinarySearch(A, p, m - 1, ν);
else if A[m] < ν then
  | BinarySearch(A, m+1, n, ν);
else
  | return m;
end
```

The worst-case running time of binary search is $\Theta(\log n)$, because for each recursion the range being searched is halved.

2.3-6.
No, the number of swaps required for each insertion makes the worst-case running time still $\Theta(n^2)$.

2.3-7.

Algorithm 7: TwoSum(S, x)

```
MergeSort(S);
left = 1, right = S.length;
while left < right do
  | if S[left] + S[right] < x then
  |   | left = left + 1;
  | else if S[left] + S[right] > x then
  |   | right = right - 1;
  | else
  |   | return true;
  | end
end
return false;
```

The worst-case running time of this algorithm is $\Theta(n \log n)+\Theta(n) = \Theta(n \log n)$.

2-1.
a. $\Theta(k^2) \cdot n/k = \Theta(nk)$.

b. n/k sublists can be merged by merging the two adjacent sublists recursively; The time complexity becomes $\Theta(n \log n/k)$.
c. $nk + n\log(n/k) = n \log n \Rightarrow k = \log k$, hence the largest value is $k = 1$.
d. In practice, analyzing operation count is necessary to decide the optimal value of k.

2-2.
a. We need to prove that $A'[1], \cdots, A'[n]$ is an rearrangement of $A[1], \cdots, A[n]$.
b. The loop invariant is that, at the start of each iteration of the outer inner loop, $A[j]$ is smallest among $A[j, \cdots, n]$. The three necessary properties of the loop invariant can be proved as:
Initialization: This is trivial, since $A[n]$ is smallest among $A[n]$.
Maintenance: In the inner loop, if $A[j]$ is smallest among $A[j, \cdots, n]$, the element is not swapped and the loop invariant is maintained. Otherwise, $A[j]$ and $A[j+1]$ is swapped so that $A[j+1]$ is placed in $A[j]$, maintaining the loop invariant.
Termination: Finally, once the loop is completed, $A[i+1]$ is the smallest element of $A[i+1, \cdots, n]$.
c. The loop invariant is that, at the start of each iteration of the outer for loop, $A[1, \cdots, i-1]$ consists of the least i member among $A[1, \cdots, n], insortedorder$. The three necessary properties of the loop invariant can be proved as:
Initialization: This is trivial, since nonempty array does not contain anything.
Maintenance: In the inner loop, the smallest element in $A[i+1, \cdots, n]$ is repeatedly swapped so that it is placed in $A[i]$, maintaining the loop invariant.
Termination: Finally, once the loop is completed, $A[1, \cdots, n]$ is sorted.
d. $\Theta(n^2)$, same with the running time of insertion sort.

2.3.
a. $\Theta(n)$.

b.

Algorithm 8: Naive polynomial evaluation(P)

```
y = 0;
for i = n downto 0 do
    term = a_n;
    for j = 1 to n do
        term = term · x;
    end
    y = y + term;
end
return y;
```

The running time of this algorithm is $\Theta(n^2)$, significantly worse than Horner's rule.
c. At termination, $i = -1$, so plugging i in the formula gives $y = \sum_{k=0}^{n} a_k x^k$.
d. Direct calculation verifies correctness of the algorithm.

2.4.
a. $(2,3)$, $(0,4)$, $(1,4)$, $(2,4)$, $(3,4)$.
b. $\{n, n-1, \cdots, 1\}$. This have $n(n-1)/2$ inversions.
c. The running time of insertion sort is proportional to number of inversions, because it is number of total swaps required.
d. The number of inversion can be simply calculated by tweaking merge sort:

Algorithm 9: MergeSort(A, p, r)

```
if p < r then
    q = (p + r) / 2, count = 0;
    count = count + MergeSort(A, p, q);
    count = count + MergeSort(A, q + 1, r);
    count = count + Merge(A, p, q, r);
    return count;
else
    return 0;
end
```

Algorithm 10: Merge(A, p, q, r)

```
n_1 = q − p + 1;
n_2 = r − q;
count = 0;
let L[1, ···, n_1] and R[1, ···, n_1] be new arrays;
for i = 1 to n_1 do
    L[i] = A[p + i - 1];
end
for j = 1 to n_2 do
    R[i] = A[q + j];
end
i = 1, j = 1;
for k = p to r do
    if L[i] < R[j] or j = n_2 + 1 then
        A[k] = L[i];
        i = i + 1;
    else
        A[k] = R[j];
        j = j + 1;
        count = count + 1;
    end
end
return count;
```

The algorithm runs in $\Theta(n \log n)$ time.