Noel Ball
HW1

1.2-2.) Insertion sort runs in $8n^2$ steps, while merge sort runs in 64nlg(n) steps.
Simplifying the inequality in order for insertion sort to beat merge sort:

- $8n^2 < 64n\lg(n)$
- $n^2 < 8n\lg(n)$
- $n < 8\lg(n)$
- $2^n < n^8$

Plotting both equations it is clear the inequality is only true after n = 2 and until n = 43, therefore insertion sort beats merge sort at input sizes $2 \leq n \leq 43$.

1.2-3.) One algorithm takes $100n^2$ steps while another takes 2^n steps for a given n inputs.
Simplifying the inequality in order for the first algorithm to run faster:

- $100n^2 < 2^n$

Plotting both equations it is clear the inequality is true after n = 14, therefore n = 15 is the smallest value of inputs that the first algorithm will run faster, so $n \geq 15$.

1-1.) Largest size n of a problem an algorithm can solve taking f(n) microseconds for each function:

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lg(n)	$2^{\wedge}(10^6)$	$2^{\wedge}(6*10^7)$	$2^{\wedge}(3.6*10^9)$	$2^{\wedge}(8.6*10^{10})$	$2^{\wedge}(2.62*10^{12})$	$2^{\wedge}(3.15*10^{13})$	$2^{\wedge}(3.15*10^{15})$
sqrt(n)	10^{12}	$36*10^{14}$	$12.96*10^{18}$	$74.65*10^{20}$	$6.86*10^{24}$	$9.92*10^{26}$	$9.92*10^{30}$
n	10^6	$6*10^7$	$3.6*10^9$	$8.64*10^{10}$	$2.62*10^{12}$	$3.15*10^{13}$	$3.15*10^{15}$
nlg(n)	$6.27*10^4$	$2.8*10^6$	$1.33*10^8$	$2.75*10^9$	$7.62*10^{10}$	$7.97*10^{11}$	$6.85*10^{13}$
n^2	10^3	$7.75*10^3$	$6*10^4$	$2.94*10^5$	$1.61*10^6$	$5.61*10^6$	$5.61*10^7$
n^3	10^2	$3.91*10^2$	$1.53*10^3$	$4.42*10^3$	$1.38*10^4$	$3.16*10^4$	$1.47*10^5$
2^n	19	25	31	36	41	45	51
n!	9	11	12	13	15	16	17

2.1-1.) Illustrating Insertion-Sort on the array:
{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***}

2.1-2.) Loop Invariant of Sum-Array:
Initialization: Prior to the first iteration of the loop, sum correctly starts as 0 because no elements of A have been added.
Maintenance: Sum is only changed by adding the element A[i] indexed at the current loop iteration to itself. Therefore sum will include elements A[1 : i-1] prior to the current loop iteration, and the element added during the iteration will ensure sum includes elements A[1 : i] before the next iteration.
Termination: The loop terminates after i = n, and there are no more elements in A to add to the sum. Therefore the procedure returns the sum of every element in A[1 : n].

2.1-4.) Pseudocode for linear search, proving the algorithm is correct with a loop invariant:
linear search(A, n, x)

```
Index = NIL
For i = 1 to n
    If A[i] = x
        Index = i  // x found!
        Return index
Return index  // no x found
```

Initialization: Prior to the first iteration of the loop, none of the elements in array A have been searched, so the index starts as NIL because x has not been found.
Maintenance: Every iteration of the for loop will act as the next index to search in A. Before the current iteration, index is still NIL because x has not been found in the elements A[1 : i-1]. After the current iteration, if A[i] is not x, then the index will remain NIL before the next iteration. However if A[i] is x, then the index is stored and the loop terminates.
Termination: The loop terminates if during any iteration A[i] is x, or after the iteration i = n because an index for x was not found. The algorithm is correct because both cases of termination give us the desired output, either an index for x in array A, or NIL indicating x is not in array A.

2.2-2.) Pseudocode for selection sort, and the loop invariant maintained:

Selection Sort(A, n)

```
For i = 1 to (n-1)
    Key = A[i]  // element to be exchanged
    Small = key
    Index = i
    For j = (i+1) to n
        If A[j] < small
            Small = A[j]
            Index = j
    A[i] = small
    A[index] = key
```

Initialization: Prior to the first iteration of the loop, when i = 1, the subarray A[1 : i-1] contains no elements and is thus sorted.
Maintenance: Before the current iteration, A[1 : i-1] elements are in sorted order. First the value at A[i] and its index is stored along with it being the smallest element this iteration has searched. Then the nested for loop will search every element after A[i], and store the smallest and its index. Swapping A[i] with the smallest element (potentially even itself if nothing was found) ensures that A[1 : i] is sorted before the next iteration.
Termination: The loop only terminates after i = n-1, demonstrating elements A[1 : n-1] are sorted and smaller than A[n]. Therefore A[1 : n] is in sorted order without needing to iterate all n elements.
Worst Case Runtime: $\Theta(n^2)$. The best case runtime is not any better, given the first for loop always iterates n times, and the second for loop n-i times. There would be less calls to the lines in the if-statement, however this does not affect the order of growth.

2.2-3.) The average case of linear search will need to check half of n elements, while the worst case will search all n elements. Given the average case is still n elements with a coefficient of ½, the order of growth for both the average and worst case is $\Theta(n)$.

2.3-4.) Mathematical induction to prove the recurrence relation:
Assume for some positive integer $k \geq 1$, $T(n) = n\lg(n)$ for all $n = 2^k$. The base case $T(2) = 2$ can be represented as $T(2) = 2\lg(2) = 2$, demonstrating the statement holds for $k = 1$.
By substituting $n = 2^{k+1}$ into the recurrence relation $T(n) = 2T(n/2)+n$: $T(2^{k+1}) = 2T(2^k)+2^{k+1}$.
The inductive hypothesis states $T(2^k) = 2^k\lg(2^k) = 2^k k$, substituted above makes $T(2^{k+1}) = 2(2^k k)+2^{k+1} = 2^{k+1}k+2^{k+1} = 2^{k+1}(k+1)$. Rewriting $(k+1)$ as $\lg(2^{k+1})$ demonstrates $T(2^{k+1}) = 2^{k+1}\lg(2^{k+1})$ matches the form $T(n) = n\lg(n)$. Therefore $T(n) = n\lg(n)$ for all $n=2^k$, $k \geq 1$

2.3-7.) If insertion sort used binary search instead of linear search, the binary search would only produce a worst case runtime of Θ(lg(n)) all n iterations of the for loop, rather than linear search causing Θ(n) for all n iterations. This would improve insertion sort's worst case runtime to Θ(nlg(n)).

2.3-8.) Pseudocode of FindSum, looks for two elements in S that exactly sum to x:

```
FindSum(S, x)
    Mergesort(S)
    Left = 1
    Right = length(S)
    While left < right
        Sum = S[left] + S[right]
        If sum == x
            Return true
        Elif sum < x
            Left += 1
        Else
            Right -= 1
    Return false
```

FindSum will return true if any two elements sum to x, or false otherwise. The worst case runtime of merge sort is Θ(nlg(n)), and the while loops performs in linear time Θ(n), making FindSum's worst case runtime Θ(nlg(n)).

2-1a.) Time complexity of linear search on n/k sublists each length k:
Each sublist of length k would take $\Theta(k^2)$ worst case runtime during the linear search, with n/k sublists to sort, the total runtime is $\Theta(k^2*n/k) = \Theta(nk)$.

2-1b.) Time complexity of merging n/k sublists:
The merge can be demonstrated as a tree with lg(n/k) levels each with n elements to compare, resulting in Θ(nlg(n/k)).

2-1c.) Largest value of k as a function of n to match merge sorts runtime:
Due to the nk term in the modified algorithms runtime Θ(nk + nlg(n/k)), any value greater than k = lg(n) would result in a runtime greater than Θ(nlg(n)). When k = lg(n) the term nlg(n/k) = nlg(n/lg(n)) which asymptotically grows smaller than nlg(n) and is not considered in the Θ. Therefore k = lg(n) results in the same runtime as standard merge sort.

2-1d.) In practice, k should be a value smaller than lg(n) in order to improve the runtime of standard merge sort.

2-3a.) The Horner procedure has a running time of $\Theta(n)$ from the for loop iterating n times.

2-3b.) Naive polynomial-evaluation computing each term from scratch:

Naive-Horner(A, n, x)

```
P = 0
For k = 0 to n
    Exp = 1
    For i = 1 to k  //calculating kth exponential term
        Exp = exp * x
    P = p + A[k] * exp
```

This method of implementation requires a runtime of $\Theta(n^2)$ due to the nested for loop required to compute the x^k term.

2-3c.) Loop invariant of horner procedure, termination resulting in $p = \sum^{n}_{k=0} A[k]*x^k$:

Initialization: Since the summation has not added any terms, p = 0.

Maintenance: Before the ith iteration $p = \sum^{n-(i+1)}_{k=0} A[k+i+1]*x^k$, and by the end of the iteration $p = \sum^{n-i}_{k=0} A[k+i]*x^k$.

Termination: The loop terminates after i = 0, in which $p = \sum^{n-i}_{k=0} A[k+i]*x^k = \sum^{n}_{k=0} A[k]*x^k$.