1. Proof by Induction:

   Base Case: when $n = 2$, $T(n) = T(2) = 2 = 2 \lg 2 = 2$.

   Inductive Hypothesis: Assume for $n = 2^k$, $T(n = 2^k) = n \lg n = 2^k \lg 2^k$.

   Inductive Step: show that for $n = 2^{k+1}$, $T(n = 2^{k+1}) = 2^{k+1} \lg 2^{k+1}$:

$$\begin{aligned} T(2^{k+1}) &= 2T(2^{k+1}/2) + 2^{k+1} = 2T(2^k) + 2^{k+1} = 2(2^k \lg 2^k) + 2^{k+1} = 2^{k+1} \lg 2^k + 2^{k+1} \\ &= 2^{k+1}(\lg 2^k + 1) = 2^{k+1}(\lg 2^k + \lg 2) = 2^{k+1}(\lg 2^k 2) = 2^{k+1}(\lg 2^{k+1}). \end{aligned}$$

2. Consider the following table:

n	$InsertionSort : 8n^2$	$MergeSort : 64n \lg n$
2	32	128
4	128	512
8	512	1,536
16	2,048	4,096
32	8,192	10,240
40	12,800	13,624
41	13,448	14,058
42	14,112	14,495
43	14,792	14,933
44	15,488	15,374
64	32,768	24,576
128	131,072	57,344
512	2,097,152	294,912

   Therefore, it becomes clear that for $n > 43$, Merge Sort is a more efficient algorithm than Insertion Sort. Notice that $8n^2 = 64n \log n$ is a transcendental equation, and is therefore only solvable by numerical approximation methods.

3. (a) Since it takes $n-1$ time in the worst case to insert $A[n]$ into the sorted array $A[1..n-1]$, we get the recurrence

$$T(n) = \begin{cases} 0 & \text{if } n = 1, \\ T(n-1) + n - 1 & \text{if } n > 1. \end{cases}$$

   (b) By simple substitution, the solution of the recurrence is

$$\begin{aligned} T(n) &= T(n-1) + (n-1) \\ &= T(n-2) + (n-2) + (n-1) \\ &= \cdots \\ &= T(1) + 1 + 2 + \cdots + (n-2) + (n-1) \\ &= \frac{1}{2} n(n-1) = \Theta(n^2) \end{aligned}$$

4. Pseudocode of selection sort

```
SelectionSort(A, v)
n = length(A)
for j = 1 to n-1
    % Find the index of smallest element in carray A[j...n]
    smallest = j
    for i = j+1 to n
        if A[i] < A[smallest]
           smallest = i
        end if
    end for
    Swap elments A[j] and A[smallest]
end for
```

Correctness: loop invariant – At the start of each iteration of the for loop, $A[1, ..., j-1]$ is sorted.

Note that the algorithm only needs to run for the first $n-1$ elements, because the algorithm consistently walks the entire array, looking for the smallest element. Therefore, the largest element bubbles to the $A[n]$ position. Therefore, when the algorithm terminates, the $A[n]$ element is already the largest.

Complexity – Best Case: for SELECTION-SORT, the best case running time occurs when the input is already sorted. Then the "if-statement" is never true and the following line is never executed. Therefore

$$\begin{aligned} T(n) &= n + n - 1 + \sum_{j=1}^{n} j + \sum_{j=1}^{n-1} j + n - 1 \\ &= n + n - 1 + n(n+1)/2 + (n-1)n/2 + n - 1 \\ &= n^2 + 3n - 2 = \Theta(n^2) \end{aligned}$$

Complexity – Worse Case: the worst case occurs when the input for SELECTION-SORT is in reverse sorted order. Therefore the "if-statement" is always true and the following is always executed. Then the running time $T(n)$ differs from above by only one term:

$$\begin{aligned} T(n) &= n + n - 1 + \sum_{j=1}^{n} j + \sum_{j=1}^{n-1} j + \sum_{j=1}^{n-1} j + n - 1 \\ &= n + n - 1 + n(n+1)/2 + (n-1)n/2 + (n-1)n/2 + n - 1 \\ &= \frac{3}{2}n^2 + \frac{5}{2}n - 2 = \Theta(n^2) \end{aligned}$$

Therefore, both the best case and the worst case running time are $\Theta(n^2)$.

5. (a) The simplest algorithm starts with the first element as the minumum and then walks the array, comparing each element with the current minimum and updating if necessary. If the length of the array is n, this takes $(n-1)$ comparisons because the first element is never compared with itself. To find the maximum, the same algorithm is applied, only checking for the maximum instead of the minimum element. Therefore, the total number of comparisons is $(n-1)+(n-1) = 2(n-1) = 2n-2$.

   (b) Since the length of our array is a power of two, consider the following divide and conquer algorithm (initially, `low` $= 1$, and `high` $= n$):

```
MinMaxRec(A,low,high)
if low == high then
   return A[low] and A[low]
else
   mid = floor( (low+high)/2 )
   [m1,M1] = MinMaxRec(A,low,mid)
   [m2,M2] = MinMaxRec(A,mid+1,high)
   return min(m1, m2) and max(M1,M2)
end if
```

   (c) $T(n) = 2T(n/2) + 2$ with $T(2) = 1$.

   (d) We have several methods for solving this recurrence relation. Since we are given the form of the solution already, we might as well use substitution/induction:

   Base Case: $T(2) = 1 = \frac{3}{2} \cdot 2 - 2$.

   Inductive Hypothesis: Assume for $n = 2^k$, $T(2^k) = \frac{3}{2} \cdot 2^k - 2$. Then we need to show that for $n = 2^{k+1}$, $T(2^{k+1}) = \frac{3}{2} \cdot 2^{k+1} - 2$:

$$\begin{aligned} T(2^{k+1}) &= 2T(2^{k+1}/2) + 2 = 2T(2^k) + 2 = 2(\frac{3}{2} \cdot 2^k - 2) + 2 \\ &= 3 \cdot 2^k - 4 + 2 = 3 \cdot 2^k - 2 = \frac{3}{2} \cdot 2^{k+1} - 2 \end{aligned}$$

6. (a) *Algorithm idea:* Suppose we want to search S between indices ℓ and r. initially $\ell = 1$ and $r = n$. If $\ell > r$, we are done and answer that there is no index i such that $S[i] = i$. Else look at $S[m]$, where $m = \lfloor(\ell + r)/2\rfloor$. If $S[m] = m$, again we're done. Otherwise if $S[m] > m$, recursively search S between ℓ and $m-1$ (the left half), while if $S[m] < m$, recursively search A between $m+1$ and r (the right half).

   (b) *Correctness:* Why does this work? Because we are dealing with distinct and sorted integers $S[i]$. When we learn that $S[m] > m$, we know that $S[m+1] > m+1$, and so forth, and therefore, we only need to continue the search in the left half of the array, i.e., $S[\ell...m-1]$. Similarly for the case of $S[m] < m$.

   (c) *Pseudocode:*

```
FindEqIndex(S,1,n)
i = FindEqIndexRec(S,1,n)
if S[i] = i then
    print(''S[i] = i'')
else if S[i] > i and S[i-1] < i-1 then
    print(''No solution'')
else if S[i] < i and S[i+1] > i+1 then
    print(''No solution'')
else
    print(''Error'')
end if

FindEqIndexRec(S,low,high)
if (high-low) == 1 then
   return low
else if (high-low) mod 2 == 0 then
   i = (high-low)/2
else
   i = (high-low+1)/2
end if
if S[i] < i then
   i = FindEqIndexRec(S,low+i,high)
else if S[i] > i then
   i = FindEqIndexRec(S,low,high-i)
end if
return i
```

   (d) *Running time:* $T(n) = T(n/2) + 1 = \Theta(\lg n)$.