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 $n = 2^{k+1}$, $T(2^{k+1}) = 2^{k+1} \lg 2^{k+1}$:

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

2. First, with $n = 2^k$ for $k = 1, 2, \ldots, 8$, we generate the following tables for functions $8n^2$ and $64n \lg n$:

n	$InsertSort : 8n^2$	$MergeSort : 64n \lg n$
2	32	128
4	128	512
8	512	1,536
16	2,048	4,096
32	8,192	10,240
64	32,768	24,576
128	131,072	57,344
512	2,097,152	294,912

   From the table, it becomes clear that starting at $n = 2^6 = 64$, MergeSort is a more efficient algorithm than InsertSort.

3. (a) Pseudocode:

```
InsertSortRecur(A,n)
if n > 1
   InsertSortRecur(A,n-1)
   // Insert A[n] to the sorted A[1...n-1]
   key = A[n]
   i = n-1
   while i > 0 and A[i] > key
       A[i+1] = A[i]
       i = i-1
  end while
  A[i+1] = key
end if
```

   (b) 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

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

(c) By iteration, the solution of the recurrence is

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

(d) Complexity of INSERT-SORT is $\frac{3}{4} \cdot n^2$ in average case. And complexity of INSERT-SORT-RECUR is $\frac{1}{2} \cdot n^2$. Both algorithm complexity are $O(n^2)$. Therefore, they cost the same.

4. (a) Pseudocode of selection sort

```
SelectionSort(A)
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.

(b) Complexity

- Best Case: 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

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

- 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:

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

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 $T(2) = 1$. For initial case, n=2=2^1 and d=1 which means there are only one element in base case. Under this condition, low is equal to high and return A[low] for both min and max case. For general case, there are two comparisons of min(m1, m2) and max(M1, M2).

   (d) We have several methods for solving this recurrence relation.

   Method 1: since we are given the form of the solution already, we might as well use the induction:

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

   Inductive hypothesis: Assume for $n = 2^k$, $T(2^k) = \frac{3}{2}\cdot 2^k - 2$.

   Inductive step: show that for $n = 2^{k+1}$, $T(2^{k+1}) = \frac{3}{2}\cdot 2^{k+1} - 2$:

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

   Method 2: Use the iteration to solve the recursion directly:

$$\begin{aligned} T(n) &= 2T(n/2)+2 \\ &= 2[2T(n/2^2)+2]+2 = 2^2T(n/2^2)+2^2+2 \\ &= \cdots \\ &= 2^{k-1}T(n/2^{k-1}) + 2^{k-1} + \cdots + 2^2 + 2 \\ &= 2^{k-1}T(2) + 2^{k-1} + \cdots + 2^2 + 2 \\ &= 2^{k-1} + 2^{k-1} + \cdots + 2^2 + 2 \\ &= 2^{k-1} + \frac{2^k - 1}{2-1} - 1 \\ &= \frac{3}{2}n - 2 \end{aligned}$$