Problem 1: (5 pts) There are two basic functionalities associated with Queue data structure, lets call them In and Out. $\text{In}(x)$ causes element $x$ to enter the queue and $\text{Out}()$ takes out an element that was entered first among all existing elements.

Our algorithm for queue implementation using two stacks is simple. Name the stacks as $\text{IN\_stack}$ and $\text{OUT\_stack}$. As names suggest, whenever an element enters the queue it is pushed onto $\text{IN\_stack}$ and the elements leaving the queue are popped from $\text{OUT\_stack}$. If $\text{OUT\_stack}$ is empty, then all the elements from $\text{IN\_stack}$ are transferred to $\text{OUT\_stack}$ by successive POP and PUSH operations.

Complete algorithm is as follows.

\[
\text{In}(x) \\
\{
  \text{PUSH}(x, \text{IN\_stack})
\}
\]

\[
\text{Out}() \\
\{
  \text{IF (OUT\_stack not empty)}\\
  \text{THEN}\\
  \text{POP(OUT\_stack)}\\
  \text{ELSE}\\
  \text{WHILE (IN\_stack not empty)}\\
  \text{PUSH(POP(IN\_stack),OUT\_stack)}\\
  \text{POP(OUT\_stack)}
\}
\]

Observe that $\text{in}(x)$ is $\Theta(1)$, while $\text{out}()$ is $\Theta(n)$ in the worst case, where $n$ is the stack size. It is worthwhile to note that even though $\text{out}()$ is expensive in the worst case, it is just $\Theta(1)$ in the amortized sense. To clarify the point, lets consider a case when $\text{out}()$ operation corresponds to transferring $m$ elements from $\text{IN\_stack}$ to $\text{OUT\_stack}$. Observe that the next $m$ operations are just $\Theta(1)$. Hence the total cost of these $m$ successive $\text{Out}$ operations is $2m$. Thus on an average $\text{Out}$ operation is $\Theta(1)$. 
**Problem 2: (5 pts)** Observe that if we have some data structure in which an element can be inserted in the front or at the back, then the sorting of a given sequence can be done using the following algorithm

FOR($i = 1$ to $n$)  

\{  
IF ($a_i \leq a$)  
    Insert_front($a_i$)  
ELSE  
    Insert_back($a_i$)  
\}

The data structure that allows the required functionality is circular linked lists (discussed in the class). In this data structure each insert operation is $\Theta(1)$ and we need $n$ inserts. Hence the complexity of the complete sorting algorithm is $\Theta(n)$.

**Problem 3: (5 pts)** A simple and yet an efficient algorithm for palindrome verification is as follows. Let the given word be stored in $Llist_1$.

**STEP 1:** Invert list $Llist_1$ and store the inverted list in $Llist_2$ (this operation is discussed in the class). Let $h1$ and $h2$ be the head pointers for the $Llist_1$ and $Llist_2$, respectively.

**STEP 2:**
WHILE ($h1 \neq$ NULL)

\{  
IF ($h1$.$letter = h2$.$letter$)  
    $h1 = h1$.$next$  
    $h2 = h2$.$next$  
ELSE  
    return(Word is NOT palindrome)  
\}

return(Word is palindrome)

Observe that the STEP 1 is $\Theta(n)$ and traversing the lists in STEP 2 is also $\Theta(n)$. Hence the palindrome verification algorithm is $\Theta(n)$.

**Problem 4: (10 pts)** Let $f(x)$ and $g(x)$ be two polynomials of degree $n$. Without loss of generality, let $n$ be the poser of 2.

Now, let

\[
    f(x) = a_{n-1}x^{n-1} + \ldots + a_1x + a_0
\]

\[
    g(x) = b_{n-1}x^{n-1} + \ldots + b_1x + b_0.
\]

We define,

\[
    f_H(x) = a_{n-1}x^{\frac{n}{2}-1} + a_{n-2}x^{\frac{n}{2}-2} + \ldots + a_{\frac{n}{2}+1}x + a_{\frac{n}{2}}
\]
\[ f_L(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1x + a_0 \]
\[ f(x) = x^\frac{n}{2}f_H(x) + f_L(x). \]

Similarly,
\[ g(x) = x^\frac{n}{2}g_H(x) + g_L(x). \]

With this construction observe that
\[ f(x)g(x) = x^n f_H(x)g_H(x) + x^\frac{n}{2}[f_H(x)g_L(x) + f_L(x)g_H(x)] + f_L(x)g_L(x). \]

Observe that we have converted a polynomial multiplication problem having polynomials of degree \( n \) into four polynomial multiplication problems involving polynomials of degree \( \frac{n}{2} \).

Observe that dividing polynomials is \( O(n) \) and then we need to combine the terms with equal powers in polynomial products \( f_H(x)g_L(x) \) and \( f_L(x)g_H(x) \), which is also \( O(n) \). Thus, if \( T(n) \) denotes the time required to solve the problem, then we have the following recursion.

\[ T(n) = 4T\left(\frac{n}{2}\right) + O(n) \]
\[ = O(n^2) \quad \text{By Master’s Thm.} \]

Hence the above divide and conquer algorithm obtains the polynomial product is \( O(n^2) \) time.