1. Linked List Questions

a.

how function would be called:

```
int find_size( LL.get_first() , LL.get_last()->next )
      // get_first() returns a pointer to first node of LinkedList
      // get_last() returns a pointer to the last node
      // this function assumes there is a trailer node
```

the implemented function:

```
int find_size(Node* first_node, Node* trailer ) /* returns number
                                       of nodes in linked list */

      {
            if ( first_node == trailer )
                  return 0;
            else
                  return find_size( first_node->next, trailer) + 1;
      }
```

b.

$T(0) = 0$
$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
$T(n-2) = T(n-3) + 1$
…

c.

$T(n) = T(n-1) + 1 \quad = T(n-1) + 1$
$ = T(n-2) + 1 + 1 \quad = T(n-2) + 2$
$ = T(n-3) + 1 + 1 + 1 \quad = T(n-3) + 3$
$k_{max} = n$
$T(n-k) = k \longrightarrow n$ (since k is at max value)

$\therefore$ **O(n)**

2. Max value in an array

```
int find_max( int* array , int size , int n=0)
{
      if( size < 1)     // make sure there is a valid size
            throw(InvalidSize);

      if ( n < size -1 )
      {
            int next = find_max( array, size, n+1);
            int max = array[n];

            if (next > max) return next;
            else return max;
      }

      else return array[n];
}
```

a.

$T(0) = 1$
$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
$T(n-2) = T(n-3) + 1$
…

- Last iteration of recursive function will be a constant
- Going through the recursive function, there are several operations but they are all constant, which for the sake of simplicity can be reduced to "+1"

b.

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

$k_{max} = n$
$T(n-k) = T(0) + k = 1 + k \longrightarrow 1 + n$

$\therefore$ **O(n)**

3.

The most suitable data structure to determine if a string is a palindrome, is a string. The string class is essentially abstraction of an array of characters. This class also has several useful functions and features, such as `size()` which returns the size of the string, `push_back()` which allows new characters to be added to a string, and the overloaded `[]` bracket operator which allow for character by character access for each index in the string. This function would be linear, O(n).

Below is how the string class would be implemented to see if a string is a palindrome:

```
string str1;      // assume it’s initialized with some value
string str2;

// must first remove spaces from str1
for (int i = 0; i < str1.size(); ++i)
{
      if (str1[i] !=“ “)
            str2.push_back(str1[i]);
}

// testing to see if it is a palindrome
is_palindrome = true;
for (int i = 0; i< str2.size()/2; ++i)
{
      if (str2[i] != str2[str2.size()-1-i]) is_palindrome = false;
}

// return value of is_palindrome
```

4. **C-5.2 p.224**

Assuming *pop* returns the top value, and removes the top element

Steps:

- **I.** Pop the first (top) value of from the stack, *S.* Check to see if it is the value you're looking for. If it is, indicate that it has been found (you could set a boolean variable, `is_found = true`).
- **II.** Put this popped element into the first spot in the queue, *Q.*
- **III.** repeat steps **I.** and **II.** until S is empty
- **IV.** Pop the first value from *Q*, put it in *S*. Repeat this until *Q* is empty. At this point, *S* should be in reverse order.
- **V.** Pop the first (top) value from *S*, and put it back into *Q*. Repeat this until *S* is empty.
- **VI.** Pop the first value from *Q*, put it back into *S*. Repeat this until *Q* is empty. At this point, *S* should be back in it's original order. The boolean variable will tell whether the certain value was found within *S*.

5. Amortized Cost Analysis

a. Doubling strategy

- If the stack isn't full, the push operation is constant, O(1).
- if the stack is full, a new array of double the size must me made. Then each element from the previous array must be copied over. This task is O(n).
- Since the increase in size happens less regularly as *n* gets bigger, the amortized cost is O(1).

Expansion of array only happens when i-1 is a power of 2. To insert *n* elements takes $\log_2(n)$ expansions.

Cost of *n* push operations:

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \log_2(n) \rfloor} 2^j < n + 2n = 3n$$

Divide 3n by cost of single push operation

3n/n = 3 —> O(1)

b. Incremental strategy

- For every new element added to the stack, the stack array must be increased. To do this a new array with size + c must be made, and all values from the old array must be copied over. This task is O(n).
- Since this increase occurs for every *n* elements, the the amortized cost is O(n).

To insert *n* elements takes $k = n/c$ expansions.
The time it takes to insert these *n* elements is proportional to

$$T(n) = (k!)\, c + n < k^2 = n^2/c$$

Divide n^2 / c by cost of single push operation

$(n^2/c)/n = n/c$ —> O(n)

6. Stack ADT

```
class StackADT {
private:
      queue Q;

public:
      bool isEmpty()
      {
            if ( Q.isEmpty() ) return true;
      }

      void pop()
      {
            if (Q.isEmpty) throw(StackEmptyException);

            queue temp;
            while( !Q.isEmpty() )
                  temp.push( Q.pop() );
            temp.pop();
            while( !temp.isEmpty() )
                  Q.push( temp.pop() );
      }

      object_type top()
      {
            if (Q.isEmpty() ) throw(StackEmptyException);

            queue temp;
            while( !Q.isEmpty() )
                  temp.push( Q.pop() );
            return temp.first();
      }

      object_type topAndPop()
      {
            object_type temp = this->top();
            this->pop();
            return temp;
      }

      void push(object)
      {
            if (Q.isFull() ) throw(StackFullException);
            queue temp;
            while ( !Q.isEmpty() )
                  temp.push( Q.pop() );
            temp.push(object);
            while (!temp.isEmpty()
                  Q.push( temp.pop() );
      }
}
```

Function	Runtime (n is size of stack)	Big-O
push()	T(n) = 4n + 2	O(n)
pop()	T(n) = 4n + 2	O(n)

7. C-5.8 p.224

- **I.** Start with a mathematical string in postfix form.
- **II.** Scan the string from left to right into a stack. Keep scanning until you scan an operator.
- **III.** When you reach an operator, pop the two most recent entries from the stack.
  - **a.** evaluate the expression in the form:

    ```
    result = second_elm_from_top [operator] first_elm_from_top
    ```

  - **b.** push the result to the top of the stack.
- **IV.** Repeat steps **II.** and **III.** until the entire string has been traversed. After this, the answer to the mathematical expression is stored on the top of the stack.

8. Quick Sort Algorithm

a.

Best/Average Case: (both have same runtime)

input, array a: {10,9,8,7,6,5,4,3,2,1}

pivots points (1st, 2nd, 3rd, etc.) = 10,1,9,2,8,3,7,4,6,10

Worst Case:

input array: {1,2,3,4,5,6,7,8,9,10}

pivots points (1st, 2nd, 3rd, etc.) = 1,2,3,4,5,6,7,8,9,10

b.

Best:

2*T(n/2) —> since 1 array is split into 2 smaller arrays

c*n —> due to copy operation

$T(n) = 2*T(n/2) + c*n$

$= 2*(2*T(n/4) + c*n/2) + c*n$ *substituting for T(n/2)*

$= 2^2*T(n/4) + 2*c*n$

$= 2^2*(2*T(n/8) +c*n/4) + 2*c*n$ *substituting for T(n/4)*

$= 2^3*T(n/8) + 3*c*n$

$= 2^k*T(n/2^k) + k*c*n$

$k_{max} = \log_2(n)$

—> $T(n) = n*T(1) + c*n*\log_2(n)$

—> **$O(n*\log_2(n))$**

Worst:

$T(n) = T(n-1) + T(1) + c*n$

$= [T(n-2)+T(1)+c*(n-1)]+T(1)+c*n$ *substituting for T(n-1)*

$= T(n-2) + 2*T(1) + c*(n-1 + n)$

$= [T(n-3)+T(1)+c*(n-2)]+2*T(1)+c*(n-1+n)$ *substituting*

$= T(n-3) + 3*T(1) + c*(n-2 + n-1 + n)$

$= T(n-k) + k*T(1) + c*(n-k+1 + n-k+2 + n-k+3 \ldots)$

$c*(n-k+1 + n-k+2 + n-k+3 \ldots)$ —> summation from k = 0 to n-1 of (n-k)

—> $(n-1)*(n)/2$

—> **$O(n^2)$**

9. Merge Sort

a.

$T(n) = 2*T(n/2) + n$
$T(n/2) = 2*T(n/4) + n/2$
$T(n/4) = 2*T(n/8) + n/4$

b.

$T(n) = 2*[\ 2*T(n/4) + n/2\ ] + n$
$= 2*2*T(n/4) + n + n$
$= 2^2*T(n/2^2) + 2*n$
$= 2*2*[\ 2*T(n/8) + n/4] + n + n$
$= 2^3*T(n/2^3) + 3n$

$k_{max} = \log_2(n)$

$T(n/2^{k-1}) = 2^k*T(n/2^k) + k*n$

$= n*T(1) + n*\log_2(n)$

c.

Best, Worst, Average: **$O(\ n*\log_2(n)\ )$**