Question 1. [Masoumeh]

(a) **Base case** when \( n = 2 \):

\[(2! = 2 \times 1 = 2) < (2^2 = 4)\]

**Inductive Hypothesis**: assume that \( n! < n^n \) for some integer \( n > 1 \)

Now, using the inductive hypothesis, we need to prove that the above holds for \( n + 1 \) i.e. that \((n + 1)! < (n + 1)^{n+1}\)

**Proof**: We first reduce the equation in such a way that enables us to use the inductive hypothesis i.e.

\[(n + 1)! < (n + 1)^{n+1}\]
\[(n + 1)n! < (n + 1)(n + 1)^n\]
\[n! < (n + 1)^n\]

The above equation holds because using the inductive hypothesis \( n! < n^n \) and the fact that \( n^n < (n + 1)^n \) we deduce that:

\[n! < n^n < (n + 1)^n\]

and hence that \( n! < n^n \) for all integers \( n > 1 \).
(b) **Base case** when \( n = 1 \):

21 divides \( 4^{1+1} + 5^{2-1} = 16 + 5 = 21 \)

**Inductive Hypothesis:** assume that 21 divides \( 4^{n+1} + 5^{2n-1} \) for an integer \( n > 0 \)

Now, using the inductive hypothesis, we need to prove that the above holds for \( n + 1 \) i.e. that 21 divides \( 4^{n+2} + 5^{2n+1} \)

**Proof:** We first reduce the problem in such a way that enables us to use the inductive hypothesis. Here we add and subtract the term \( 4^{n+1} \cdot 5^2 \) and we obtain:

\[
(4^{n+2} + 5^{2n+1}) + 4^{n+1} \cdot 5^2 - 4^{n+1} \cdot 5^2
\]

\[
5^2(4^{n+1} + 5^{2n-1}) + 4^{n+2} - 4^{n+1} \cdot 5^2
\]

\[
25(4^{n+1} + 5^{2n-1}) - 4^{n+1}(5^2 - 4)
\]

\[
25(4^{n+1} + 5^{2n-1}) - 4^{n+1}(21)
\]

The above expression is divisible by 21 because from the inductive hypothesis we know that the first term is divisible by 21 and we see that the second term is divisible by 21 as well. Therefore the whole expression is divisible by 21 and hence 21 divides \( 4^{n+1} + 5^{2n-1} \) for all integers \( n > 0 \).

**Question 2. [Masoumeh]**

**R-3.20**

- The first assignment, int \( a \), takes constant time \( O(1) \).

- The comparison, \( i < n \), will be repeated \( n + 1 \) times within the for loop and hence is \( O(n) \).

- The counter, \( i \), will be initialized and incremented \( n \) times and this takes \( O(n) \).
The assignment within the for loop, $a = i$, will be repeated $n$ times and hence is $O(n)$.

Therefore the algorithm runs in $O(1 + n + n + n)$ i.e. $\in O(n)$.

R-3.21

- The for loop of this algorithm will repeat for $n/2$ times instead of $n$ times. Otherwise it's identical to the previous algorithm.

Therefore the algorithm runs in $O(1 + n/2 + n/2 + n/2)$ i.e. $\in O(n)$.

R-3.22

- The for loop of this algorithm will repeat for $n^2$ times.

As before, the algorithm runs in $O(1 + n^2 + n^2 + n^2)$ i.e. $\in O(n^2)$.

R-3.23

- The outer for loop will be repeated $n$ times. For each iteration of the outer loop, $i$ operations will be performed where $i$ varies between 1 and $n$.

The algorithm runs in $O(1 + 2 + 3 + \ldots + (n - 1) + n)$ i.e. $O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n(n+1)}{2}\right) \in O(n^2)$.

R-3.24

- The outer for loop will be repeated $n^2$ times. For each iteration of the outer loop, $i$ operations will be performed where $i$ varies between 1 and $n^2$.

The algorithm runs in $O(1 + 2 + 3 + \ldots + (n^2 - 1) + n^2)$ i.e. $O\left(\sum_{i=1}^{n^2} i\right) = O\left(\frac{n^2(n^2+1)}{2}\right) \in O(n^4)$. 

3
\[ \sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots + (n-1)^2 + n^2 = \frac{n(n + 1)(2n + 1)}{6} = \frac{2n^3 + 3n^2 + n}{6} \in O(n^3) \]

**Question 3.** [Masoumeh]

```java
func popN(List L, int n): returns List  begin;
    if(isempty(L) || (n < 1))
        return newList();
    else
        SubList = popN(sub(L), n-1);
        push(top(L), SubList);
        return SubList;
    end if;
end func;

func merge(List A, List B):  begin;
    if(isempty(B))
        return;
    else if (isempty(A))
        A = B;
    else if(top(B) < top(A))
        push(top(B), A);
    ```
merge(sub(A),sub(B));
else
merge(sub(A),B);
end if;
end func;

Analysis of LmergeSort

- length(L) is \( O(n) \).
- popN(L,n) is \( O(n) \).
- merge(A,B) is \( O(n) \).

Hence the recurrence relation is:

\[
T(n) = 2T(n/2) + O(n) + O(n) + O(n)
\]
i.e.

\[
T(n) = 2T(n/2) + kO(n)
\]
where \( k \) is a constant.

by iteratively substituting for \( T(n/2) \) in the recurrence relation above we’ll obtain the following after \( i \) times:

\[
T(n) = 2^iT(n/2^i) + kin
\]
assuming that \( n = 2^h \) for some integer \( h \), the above process will terminate after \( h \) times. \( h = \log(n) \) and hence we obtain the following:

\[
T(n) = 2^{\log(n)}(O(1)) + kn\log(n)
\]

5
\[ = nO(1) + kn\log(n) \]

This means that the Big-Oh running time for LmergeSort is \( O(n\log n) \).

**Question 4. [Gregory]**

Implement the recursive adaptive integration routine in Java, using the provided templates.

We can calculate the area of a trapezoid by finding the lengths of its two parallel sides, averaging them (by summing them and dividing by two), and then multiplying the result by the width of the trapezoid. Since the parallel lines in the trapezoids for the adaptive integration routine are both vertical, we can find the length of each by simply examining the value of \( f(hi), f(lo), \) and \( f(mid) \) as appropriate. The width of a trapezoid is just \( hi - lo \) (or \( hi - mid \), or \( mid - lo \), as appropriate). We fill in the AdaptiveInt method from the Adapt.java file as follows:

```java
public static double
AdaptiveInt(IntegFunction F, double lo, double hi, double tol) {

    double mid, area1, area2;

    mid = (hi + lo)/2;

    // We can calculate a trapezoid's area by finding the length
    // of each side of the trapezoid, dividing by 2 to get the
    // "average length", and then multiplying this result by
    // the width (which is (hi-lo).
    //
    //
```
Thus, we can use the following equation:

\[
((\text{abs}(f(lo)-0) + \text{abs}(f(hi)-0)) / 2) \times (hi - lo)
\]

\[
\text{area1} = ((\text{Math.abs}(F.\text{valueAt}(lo)) + \text{Math.abs}(F.\text{valueAt}(hi)))
/ 2) + (hi - lo);
\]

We use similar calculations here for the two smaller trapezoids that we look at, and determine the sum of both their areas.

\[
\text{area2} = (((\text{Math.abs}(F.\text{valueAt}(lo)) + \text{Math.abs}(F.\text{valueAt}(mid)))
/ 2) \times (mid - lo))
+ (((\text{Math.abs}(F.\text{valueAt}(mid)) + \text{Math.abs}(F.\text{valueAt}(hi)))
/ 2) \times (hi - mid));
\]

if (Math.abs(area1 - area2) < tol)
    return area2;
else
    return AdaptiveInt(F, lo, mid, tol) +
    AdaptiveInt(F, mid, hi, tol);
}

Output from the Logarithmic function:

Log test: range 1 to 4
area: 3.462892142331044 tol:1.0
area: 3.6517578602094254 tol:0.1
area: 3.6646865467253944 tol:0.01
area: 3.6704238342997 tol:0.0010
area: 3.671597911977007 tol:1.0E-4
area: 3.671841158610979 tol:1.0E-5
area: 3.6718995174926974 tol:1.00000000000002E-6
area: 3.6719117646196794 tol:1.00000000000002E-7
area: 3.671914264165523 tol:1.00000000000002E-8
area: 3.671914748931292 tol:1.00000000000003E-9

Output from the Linear function:

Log test: range 1 to 4
area: 7.5 tol:1.0
area: 7.5 tol:0.1
area: 7.5 tol:0.01
area: 7.5 tol:0.0010
area: 7.5 tol:1.0E-4
area: 7.5 tol:1.0E-5
area: 7.5 tol:1.00000000000002E-6
area: 7.5 tol:1.00000000000002E-7
area: 7.5 tol:1.00000000000002E-8
area: 7.5 tol:1.00000000000003E-9

Output from the Quadratic function:

Log test: range 1 to 4
area: 21.28125 tol:1.0
area: 21.0703125 tol:0.1
area: 21.017578125 tol:0.01
Question 5. [Gregory]

Write a pseudocode algorithm to determine if a list of integers contains a duplicate item, running in $O(n \log(n))$ time.

There are numerous good solutions to this problem. One simple solution is to sort the list and then run through the sorted list linearly searching for duplicate elements. We know that there exist sorting algorithms that run in $O(n \log(n))$ time, such as MergeSort for example. The process of searching through the sorted list looking for duplicates can be done in $O(n)$ time, because we need to look at each of the $n$ elements in the list only once. Thus, the total running time for such an algorithm would be $O(n \log(n) + n)$, which reduces to simply $O(n \log(n))$ (because the function $n \log(n)$ is bigger than the function $n$ as long as we’re looking at large values of $n$). The pseudocode for this algorithm would look as follows:

```pseudocode
function HasDuplicate(IntegerList list) {
    list = MergeSort(list) // sort our list in $O(n \log(n))$ time

    for (i=1 to (size of list)-1):
        if (list[i] equals list[i+1]) then
```
return true       // we've found a duplicate!

// If we get to this line, then we've run through the entire
// list and never found any duplicates. So there must
// not be any duplicates at all.
return false
}