Lab-2- Solutions

Problem 1

Method 1: Recursive Algorithms

```
Algorithm removeDuplicates(L)
	if L.isEmpty() return L
	return removeDupsRec(L, L.first()) // Start with the first node

Algorithm removeDupsRec(L, curr) // Current node is the first node here
	// L has only one element
	if L.isLast(curr) then // If current node is a last node then stop and return the single node
		return L
	return removeDupsHelper(L, curr.element(), L.after(curr))

Algorithm removeDupsHelper(L, currElem, p) // It takes the current and current next
	prev := L.before(p) // maintain the previous to track the point which is not duplicate
	if p.element() = currElem then  // Current next and current are same means duplicate found
		L.remove(p)  // [ 1 1 2 3 4], remove the duplicate current next, position 1.

	 // Start from the same position after removing duplicates [ 1  2 3 4], start 0 index
		p := prev
	if ! L.isLast(p) then // If the current element is not last recursively call itself
		removeDupsHelper(L, p, L.after(p))
```

In the worst case that no duplicates, the removeDupsHelper function would be called for each element in the list, and each call would iterate through the remaining part of the list. Therefore, the overall time complexity would be O(n^2), where n is the length of the list.

Method 2: Iterative Approach

```
Algorithm removeDups(L)
	if L.isEmpty() then return L
	curr := L.first()
	while ! L.isLast(curr) do
		currElem = curr.element()
		p := L.after(curr)
		while !L.isLast(p) do  // remove all elements = currElem
			if p.element() = currElem then
				prev := L.before(p)
				L.remove(p)
				p := prev
			if !L.isLast(p) then
				p := L.after(p)
		if p != curr ∧ currElem = p.element() then
			L.remove(p)
		if !L.isLast(curr) then
			curr := L.after(curr)
```

Problem 2:

```
Big O  Algorithm isPermutation(A, B)
1         if A.size() != B.size() then // If both are not same size return false
1                 return false
// Both list are empty due to same size. Empty is the permutation here. Return trus and stop
1         if A.isEmpty() then
1                 return true

1         p := A.first()  // Take the first element from A
n         if  ! isMemberOf(p.element(), B) then // If the first element is not in B, return false, stop
1                 return false

n         while ! A.isLast(p) do // Need to check the all the elements of A is in B
n                 p := A.after(p) // Check from the second element, first is verified in the previous logic
n*n               if  ! isMemberOf(p.element(), B) then // calling another helper function
1                         return false
1         return true

       Algorithm isMemberOf(elem, L)
1         curr := L.first()       // Get the first element from the Sequence B
1         if curr.element() = elem then // Both sequence matched return true
1                 return true
n         while ! L.isLast(curr) do // Traverse second to last element to check the existence of A element
n                 curr := L.after(curr) // Move to the next element in B
n                 if curr.element() = elem then // Match found return true
1                         return true
1         return false // Sequence A element is not in the Sequence B
```

Time Complexity: O(n^2), due to nested loop

Problem 3 – A

```
Algorithm max(L)
        p := L.first()
        max := p.element()
        while L.last() != p do
                p := L.after(p)
                if( p.element() > max)
                  max = p.element
        return max
```

Time Complexity: O(n), need to compare n items to find the max.

Problem 3 – B

Method 1:

```
Algorithm findMiddle(L)
  Input: Linked List
  Output: Middle Node

if L.isEmpty() then return null
  p := L.first()
  q := L.last()

//  If size is even L.after(p) != q becomes false, if size is odd p != q becomes false

while p != q ∧ L.after(p) != q do
   p := L.after(p)
   q := L.before(q)
return p
```

Method 2:

```
Algorithm findMiddle(L)
  Input: Linked List
  Output: Middle Node
        If L.isEmpty()
            return Null
        mid= (L.size()/2)  // Take the ceiling
        current = L.first()
        for i=1 to mid  do
          current= L.after(current)
        return current
```

Time Complexity: pointer moving one from left and another from right. To find the middle it requires O(n).

Task C:

```
Algorithm removeMiddle(L)
          p := findMiddle(L)
          return L.remove(p)
```