Python

```python
#N^2, because the use of indexing/getitem
def copy(ll_A: LinkedList) -> LinkedList:
    ll_B = LinkedList()

    for i in range(ll_A.size()):
        ll_B.add(ll_A[i])

    return ll_B

#N. Huge savings from not using indexing.
def copy(ll_A: LinkedList) -> LinkedList:
    ll_B = LinkedList()

    curr = ll_A.head
    while curr != None:
        ll_B.add(curr.getData())
        curr = curr.getNext()

    return ll_B
```

Python

```python
def is_sorted(ll_A: LinkedList) -> Boolean:
      # if size <= 1, then list is automatically sorted
      if ll_A.size() <= 1:
            return True

      current: Node = ll_A.head
      current_next: Node = current.getNext()
      while current_next is not None:
            if current.getData() > current_next.getData():
                  return False
            current = current.getNext()
            current_next = current_next.getNext()
      return True
```

Python

```python
#N^2
def strictly_greater(ll_A: LinkedList, ll_B: LinkedList) -> LinkedList:
    for i in range(ll_A.size()):
        if ll_B[i] != None and ll_A[i] <= ll_B[i]:
            return False
    return True

#N
def strictly_greater(ll_A: LinkedList, ll_B: LinkedList) -> LinkedList:
    curr_A, curr_B = ll_A.head, ll_B.head

    while curr_A != None and curr_B != None:
        if curr_A.getData() <= curr_B.getData():
            return False
        curr_A = curr_A.getNext()
        curr_B = curr_B.getNext()

    return True
```

Python

```python
# Solution A
def max_at_each_position(A: LinkedList, B: LinkedList) -> LinkedList:
    max_list = LinkedList()
    size_A = A.size()
    size_B = B.size()
    max_size = size_A if size_A > size_B else size_B
    for i in range(max_size):
        value_A = A[i] if i < size_A else None
        value_B = B[i] if i < size_B else None
        if value_A is not None and value_B is not None:
            max_value = value_A if value_A > value_B else value_B
        elif value_A is None:
            max_value = value_B

        else:
            max_value = value_A

        max_list.add(max_value)

# Solution B
def max_at_each_position(A: LinkedList, B: LinkedList) -> LinkedList:
      max_list = LinkedList()
      a_curr = A.head
      b_curr = B.head
      while a_curr is not None or b_curr is not None:
            if a_curr is None:
                  max_list.add(b_curr.getData())
            elif b_curr is None:
                  max_list.add(a_curr.getData())
            else:
                  max_value = max(a_curr.getData(), b_curr.getData())
                  max_list.add(max_value)
            a_curr = a_curr.getNext()
            b_curr = b_curr.getNext()
      return max_list
```

Python

```python
# Solution A
def remove_dupes(ll_A: LinkedList) -> LinkedList:
	non_duplicates = []  # could also use 'set' instead
	current = ll_A.head
	ll_B = LinkedList()
	while current is not None:
		current_data = current.getData()
		if current_data not in non_duplicates:
			ll_B.add(current_data)
			non_duplicates.append(current_data)
		current = current.getNext()
	return ll_B

# Solution B
def remove_dupes(ll_A: LinkedList) -> LinkedList:
	ll_B = LinkedList()
	current = ll_A.head
	while current is not None:
		# since the ll_B is in ascending order, if a value is repeated,
the same value would have been already added to the end of ll_B the first time
it appeared in ll_A.
		if ll_B.tail is None or ll_B.tail.getData() != current.getData():
			ll_B.add(current.getData())
		current = current.getNext()
	return ll_B
```

Python

```python
def merge(LinkedListA: LinkedList, LinkedListB: LinkedList) -> LinkedList:
    merged_list = LinkedList()
    currentA = LinkedListA.head
    currentB = LinkedListB.head
    # Iterate until we add all from one list
    while currentA is not None and currentB is not None:
        if currentA.getData() <= currentB.getData():
            merged_list.add(currentA.getData())
            currentA = currentA.getNext()

        else:
            merged_list.add(currentB.getData())
            currentB = currentB.getData()

    # Add the rest of the elements from the other list
    while currentA is not None:
        merged_list.add(currentA.getData())
        currentA = currentA.getNext()

    while currentB is not None:
        merged_list.add(currentB.getData())
        currentB = currentB.getData()

    return merged_list
```

Python

```
def repeat(ll_A, ll_B):
	big_curr = ll_B.head
	small_curr = ll_A.head
	new_ll = LinkedList()
	while small_cur is not None and big_curr is not None:
		# Track which LinkedList has the larger and smaller curr value
		if big_curr.getData() < small_curr.getData():
			temp = big_curr
			big_curr = small_curr
			small_curr = temp
		# Case we have a unique value that hasn't already been added to
new_ll
		if new_ll.tail is None or new_ll.tail.getData() !=
small_curr.getData():
			# Case we see a value that is in both LinkedLists
			if small_curr.getData() == big_curr.getData():
				new_ll.add(small_curr.getData())
		# Keep iterating down the list with a smaller value, until we get
to a point where the data at small_cur > big_curr
		small_curr = small_cur.getNext()
	return new_ll
```