Question 2:
1.

```
(defun partition-list (lst)
 (labels ((partition-step (unprocessed list1 list2)
      (cond ((null unprocessed)
         (list list1 list2))  ; Return both lists when no more items
        ((null (cdr unprocessed))  ; Only one item left
         (partition-step nil
               (append list1 (list (car unprocessed)))
               list2))
        (t  ; Two or more items - add to each list
         (partition-step (cddr unprocessed)
               (append list1 (list (car unprocessed)))
               (append list2 (list (cadr unprocessed))))))))
  (format t "Starting partition of: ~a~%" lst)
  (let ((result (partition-step lst nil nil)))
   (format t "Partitioned into:~%List 1: ~a~%List 2: ~a~%~%"
       (car result) (cadr result))
   result)))
```

2. In merge sort, after partitioning the list into two halves, the next step is to recursively sort each half. Once both halves are sorted, they are merged back together in sorted order.

3.

```
(defun merge-sorted-lists (list1 list2)
 (cond ((null list1) list2)
    ((null list2) list1)
    ((< (car list1) (car list2))
     (cons (car list1) (merge-sorted-lists (cdr list1) list2)))
    (t
        (cons (car list2) (merge-sorted-lists list1 (cdr list2))))))
```

End Conditions:

- If one of the lists is empty, append the remaining elements from the other list to the processed list.
- If both lists are empty, the merger is complete.

4.

```
;; Function to merge two sorted lists
(defun merge-sorted-lists (list1 list2)
  (cond ((null list1) list2)
        ((null list2) list1)
        ((< (car list1) (car list2))
         (cons (car list1) (merge-sorted-lists (cdr list1) list2)))
        (t
         (cons (car list2) (merge-sorted-lists list1 (cdr list2))))))


;; Function to sort a list using merge sort
(defun mergesort (lst)
  (cond ((null lst) nil)                  ; empty list
        ((null (cdr lst)) lst)            ; single element
        (t (let* ((partitioned (partition-list lst))
                  (left (car partitioned))
                  (right (cadr partitioned)))
             (merge-sorted-lists (mergesort left)    ; recursively sort left
                                 (mergesort right)))))) ; recursively sort right


;; Function to partition a list into two sublists
(defun partition-list (lst)
  (labels ((partition-step (unprocessed list1 list2)
        (cond ((null unprocessed)
               (list list1 list2))  ; Return both lists when no more items
              ((null (cdr unprocessed))  ; Only one item left
               (partition-step nil
                       (append list1 (list (car unprocessed)))
                       list2))
              (t  ; Two or more items - add to each list
               (partition-step (cddr unprocessed)
                       (append list1 (list (car unprocessed)))
                       (append list2 (list (cadr unprocessed))))))))
  (format t "Starting partition of: ~a~%" lst)
  (let ((result (partition-step lst nil nil)))
   (format t "Partitioned into:~%List 1: ~a~%List 2: ~a~%~%"
       (car result) (cadr result))
   result)))
```

Question 3:

```
(defun partition-list (unprocessed list1 list2)
(cond
;; Case 1 If unprocessed is empty, return the two lists
((null unprocessed) (list list1 list2))
;; Case 2 If there is only one item, add it to the first list
((null (cdr unprocessed)) (list (cons (car unprocessed) list1) list2))
;; Case 3 Recursive step: Take the first two items, add one to each list
(t (partition-list (cddr unprocessed) (cons (car unprocessed) list1) (cons (cadr unprocessed)
list2)))))
```

```
;; Function to sort a pair of numbers
(defun sort-pair (a b)
  (if (< a b)
      (list a b)
      (list b a)))

;; Function to create initial sorted pairs with step tracking
(defun make-sorted-pairs (lst)
  (labels ((make-pairs-step (processed remaining)
             (cond ((null remaining)
                    processed)
                   ((null (cdr remaining))
                    (format t "Current pairs: ~a~%" processed)
                    (format t "Remaining: ~a~%" remaining)
                    (cons (list (car remaining)) processed))
                   (t
                    (let ((new-pair (sort-pair (car remaining) (cadr remaining))))
                      (format t "Current pairs: ~a~%" processed)
                      (format t "Remaining: ~a~%~%" remaining)
                      (make-pairs-step
                       (cons new-pair processed)
                       (cddr remaining)))))))
    (reverse (make-pairs-step nil lst))))

;; Function to merge two sorted lists
(defun merge-sorted-lists (list1 list2)
  (cond ((null list1) list2)
        ((null list2) list1)
        ((< (car list1) (car list2))
         (cons (car list1) (merge-sorted-lists (cdr list1) list2)))
        (t
         (cons (car list2) (merge-sorted-lists list1 (cdr list2))))))

;; Function to merge pairs of sorted lists
(defun merge-pass (lists)
  (cond ((null lists) nil)
        ((null (cdr lists)) lists)
        (t (cons (merge-sorted-lists (car lists) (cadr lists))
                 (merge-pass (cddr lists))))))

;; Main mergesort function that shows steps
(defun mergesort-with-steps (lst)
  (format t "Starting with list: ~a~%~%" lst)
  ;; Step 1: Create sorted pairs with detailed steps
  (let ((pairs (make-sorted-pairs lst)))
    (format t "~%Final sorted pairs: ~a~%~%" pairs)

    ;; Step 2: Merge pairs until we have one sorted list
    (do ((current-lists pairs (merge-pass current-lists)))
        ((null (cdr current-lists)) (car current-lists))
      (format t "After merge pass: ~a~%~%" (merge-pass current-lists)))))

;; Test function
(defun test-mergesort ()
  (let ((test-list '(1 7 2 1 8 6 5 3 7 9 4)))
    (format t "Original list: ~a~%" test-list)
  (format t "Final sorted list: ~a~%" (mergesort-with-steps test-list))))


;; Run the test

(test-mergesort)
```

Question 4:

1. **While insertion sort is in progress, we track 2 lists: the sorted items and the unsorted items. What should these look like when the process starts and when the process ends? Can we use that information to decide termination?**
    a. Process starts:
        Sorted items list: Empty
        Unsorted items list: items that need to be sorted.
    b. Process ends:
        Sorted items list: expand and sorted in order because of items inserted from the unsorted list.
        Unsorted items list: Empty
    c. The information to decide the termination: when the unsorted list is empty and the sorted list is fully sorted using the insertion sort.

2. **In each pass we start with two lists. At the end of the pass, we would have moved one more item from the sorted to the unsorted list. When is this process trivially accomplished? What will the recursive call look like?**
    a. The insertion process is trivially accomplished when the unsorted list has only one item left, as that item can be directly appended to the sorted list without needing any further comparisons.
    b. When the unsorted list has only one item left, that item is inserted directly into the sorted list, and the recursive call ends.
        Example:
            (insertion-sort '(5) '(1 3 4 7)) => (1 3 4 5 7)
3. **Moving requires an operation that can insert an item into a sorted list in ascending order. This can be done easily using the predefined LAST function in Common Lisp (look for this in the LISP text; LAST and BUTLAST are like the reverse of CAR and CDR). Again, to represent this insertion process, we need to track the items that have been examined, the item to be inserted, and the items yet to be examined. There are two ways in which the process can terminate – what are they?**
    The insertion process terminates:
    - The unsorted List is empty
    - When the item to be inserted in the unsorted list is larger than all remaining items in the Sorted List, allowing it to be appended at the end.
4. **Code:**

```
(defun insert-item (item sorted)
        (cond
                ((null sorted) (list item)) ; If the sorted list is empty, return the item as the only
        element.
                ((< item (car sorted)) (cons item sorted)) ; If the item is smaller than the first
        element, insert it at the front.
                (t (cons (car sorted) (insert-item item (cdr sorted)))))) ; recursively insert into the rest
        of the list.

(defun insertion-sort (unsorted sorted)
        (cond
                ((null unsorted) sorted) ; If unsorted list is empty, return the sorted list.
                ((null (cdr unsorted)) (insert-item (car unsorted) sorted)) ; If unsorted list has only
        one item, insert it into the Sorted List.
                (t (insertion-sort (cdr unsorted) (insert-item (car unsorted) sorted))))) ; Insert the first
        item into sorted list.

(defun sort-list (lst)
        (insertion-sort lst nil)) ; Start sorting with an empty sorted list.
```