Below are four different functions for sorting a list of elements in increasing order. For simplicity, assume that the list only contain ints. For each, we are going to ask you about how the algorithm creates its output and about the worst case time complexity - or order of growth - of the algorithm. Answer the questions without running the code on your computer.

PROBLEM 3-1

Answer the following 5 questions based on this code.

```
def sort1(lst):
    swapFlag = True
    iteration = 0
    while swapFlag:
        swapFlag = False
        for i in range(len(lst)-1):
            if lst[i] > lst[i+1]:
                temp = lst[i+1]
                lst[i+1] = lst[i]
                lst[i] = temp
                swapFlag = True

        L = lst[:]  # the next 3 questions assume this line just executed
        iteration += 1
    return lst
```

PROBLEM 3-1 A (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the smallest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-1 B (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the largest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ◉ True ✔
- ○ False

You have used 1 of 1 submissions

PROBLEM 3-1 C (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the first `n+1` elements of the original list, `lst`, appear in the correctly sorted places in L. The "correctly sorted places" refers to the order of the elements in the list, not the index.

- (•) True ✔
- ( ) False

You have used 1 of 1 submissions

PROBLEM 3-1 D (1/1 point)

The function sorts the list `lst` in place without using a new list.

- (•) True ✔
- ( ) False

You have used 1 of 1 submissions

PROBLEM 3-1 E (1/1 point)

The complexity of this algorithm is:

O(n^2) ▾

You have used 1 of 1 submissions

PROBLEM 3-2

Answer the following 5 questions based on this code.

```
def sort2(lst):
    for iteration in range(len(lst)):
        minIndex = iteration
        minValue = lst[iteration]
        for j in range(iteration+1, len(lst)):
            if lst[j] < minValue:
                minIndex = j
                minValue = lst[j]
        temp = lst[iteration]
        lst[iteration] = minValue
        lst[minIndex] = temp

        L = lst[:]  # the next 3 questions assume this line just executed
```

```
    return lst
```

PROBLEM 3-2 A (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the smallest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- (•) True ✔
- ( ) False

You have used 1 of 1 submissions

PROBLEM 3-2 B (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the largest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-2 C (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the first `n+1` elements of the original list, `lst`, appear in the correctly sorted places in `L`. The "correctly sorted places" refers to the order of the elements in the list, not the index.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-2 D (1/1 point)

The function sorts the list `lst` in place without using a new list.

- ◉ True ✔
- ○ False

You have used 1 of 1 submissions

PROBLEM 3-2 E (1/1 point)

The complexity of this algorithm is:

O(n^2) ▾

You have used 1 of 1 submissions

PROBLEM 3-3

Answer the following 5 questions based on this code.

```
def sort3(lst):
    out = []
    for iteration in range(0,len(lst)):
        new = lst[iteration]
        inserted = False
        for j in range(len(out)):
            if new < out[j]:
                out.insert(j, new)
                inserted = True
                break
        if not inserted:
            out.append(new)

        L = out[:]  # the next 3 questions assume this line just executed
```

```
    return out
```

PROBLEM 3-3 A (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the smallest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-3 B (1/1 point)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the largest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-3 C (1 point possible)

When we reach the marked spot in the code, and the variable `iteration` has value `n`, the first `n+1` elements of the original list, `lst`, appear in the correctly sorted places in `L`. The "correctly sorted places" refers to the order of the elements in the list, not the index.

- ○ True
- ◉ False ✘

You have used 1 of 1 submissions

PROBLEM 3-3 D (1/1 point)

The function sorts the list `lst` in place without creating a new list.

- ○ True
- ◉ False ✔

Help

You have used 1 of 1 submissions

PROBLEM 3-3 E (1/1 point)

The complexity of this algorithm is:

O(n^2) ▾

You have used 1 of 1 submissions

PROBLEM 3-4

Answer the following 5 questions based on this code.

```
def sort4(lst):
    def unite(l1, l2):
        if len(l1) == 0:
            return l2
        elif len(l2) == 0:
            return l1
        elif l1[0] < l2[0]:
            return [l1[0]] + unite(l1[1:], l2)
        else:
            return [l2[0]] + unite(l1, l2[1:])

    if len(lst) == 0 or len(lst) == 1:
        return lst
    else:
        front = sort4(lst[:len(lst)/2])
        back = sort4(lst[len(lst)/2:])

        L = lst[:]  # the next 3 questions assume this line just executed
```

```
        return unite(front, back)
```

PROBLEM 3-4 A (1/1 point)

When we reach the marked spot in the code on the `n` th recursive call of `sort4`, the smallest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-4 B (1/1 point)

When we reach the marked spot in the code on the `n` th recursive call of `sort4`, the largest `n+1` elements of the sorted version of `lst` are in `L` in the correct order.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-4 C (1/1 point)

When we reach the marked spot in the code on the `n` th recursive call of `sort4` , the first `n+1` elements of the original list, `lst` , appear in the correctly sorted places in `L` . The "correctly sorted places" refers to the order of the elements in the list, not the index.

- ○ True
- ◉ False ✔

You have used 1 of 1 submissions

PROBLEM 3-4 D (1 point possible)

The function sorts the list `lst` in place without creating a new list.

- ◉ True ✖
- ○ False

You have used 1 of 1 submissions

PROBLEM 3-4 E (1/1 point)

The complexity of this algorithm is:

O(n log n) ▾

You have used 1 of 1 submissions