Searching and Sorting

1.

While loop version:

```
def linear_search(lst, value):
    """ (list, object) -> int

    Return the index of the last occurrence of value in lst, or return
    -1 if value is not in lst.

    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    2
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """

    i = len(lst) - 1  # The index of the next item in lst to examine.

    # Keep going until we reach the end of lst or until we find value.
    while i != -1 and lst[i] != value:
        i = i - 1

    # If we fell off the end of the list, we didn't find value.
    if i == -1:
        return -1
    else:
        return i
```

For loop version:

```
def linear_search(lst, value):
    """ (list, object) -> int

    Return the index of the last occurrence of value in lst, or return
    -1 if value is not in lst.

    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    2
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """

    # The first index is included, the second is not, and the third is the
```

```
    # increment.
    for i in range(len(lst) - 1, -1, -1):
        if lst[i] == value:
            return i

    return -1
```

Sentinal version:

```
def linear_search(lst, value):
    """ (list, object) -> int

    Return the index of the last occurrence of value in lst, or return
    -1 if value is not in lst.

    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    2
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """

    # Add the sentinel at the beginning.
    lst.insert(0, value)

    i = len(lst) - 1

    # Keep going until we find value.
    while lst[i] != value:
        i = i - 1

    # Remove the sentinel.
    lst.pop(0)

    # If we reached the beginning of the list we didn't find value.
    if i == 0:
        return -1
    else:
        # When we inserted, we shifted everything one to the right. Subtract
1
        # to account for that.
        return i - 1
```

2.

If there are duplicates, the one at the highest index is found.

3.

The question asks **roughly** how many times we need to search in order to make sorting the list first (on the order of N log_2 N steps) and then using binary search (on the order of log_2 N steps) faster than than just using linear search (on the order of N steps).

If there are k searches, then using linear search takes on the order of k * N steps.

It takes N log_2 N steps to sort a list with N items. In addition to the sorting time, there are all the binary searches to account for, each taking log_2 N steps, so this approach takes on the order of N log_2 N + k * N log_2 N steps.

```
N log_2 N + k * log_2 N < k * N
  [subtract k * log_2 N from both sides]
N log_2 N < k * N -  k * log_2 N
  [simplify]
N log_2 N < k * (N - log_2 N)
  [divide both sides by (N - log_2 N)]
(N log_2 N) / (N - log_2 N) < k
  [N is much larger than log_2 N, so we simplify the denominator and multiply
the numerator by 2
  (remember, we want a rough answer)]
(2 * N log_2 N) / N < k
  [simplify]
2 * log_2 N < k
```

The actual threshold is slightly smaller, but this is a fine rough estimate.

4.

a. Selection sort:

```
[6, 5, 4, 3, 7, 1, 2]
[1, 5, 4, 3, 7, 6, 2]
[1, 2, 4, 3, 7, 6, 5]
[1, 2, 3, 4, 7, 6, 5]
[1, 2, 3, 4, 5, 6, 7]
# Because selection sort doesn't stop even though the list is sorted,
# there is one more iteration.
[1, 2, 3, 4, 5, 6, 7]
```

b. Insertion sort:

```
[6, 5, 4, 3, 7, 1, 2]
[5, 6, 4, 3, 7, 1, 2]
[4, 5, 6, 3, 7, 1, 2]
[3, 4, 5, 6, 7, 1, 2]
[3, 4, 5, 6, 7, 1, 2]  # The 7 doesn't move on this iteration.
[1, 3, 4, 5, 6, 7, 2]
[1, 2, 3, 4, 5, 6, 7]
```

5.

a.

```
until all items are sorted:
    # sweep through the list.
    for each pair of items in the unsorted part of the list:
        if the pair is out of order:
            swap them
```

b.

After the first sweep, the largest item must have been swapped to the end of the list. After the second sweep, the second-largest item must have been swapped to the second-last spot. So there is a sorted section accumulating at the end of the list, and we'll keep track of the end of the unsorted section.

```
# The end of the unsorted section.  The largest item will be placed here.
end = len(lst) - 1

# Keep going until there is only one item to consider.
while end != 0:
    # sweep through the list.
    for each pair of items in the unsorted part of the list:
        if the pair is out of order:
            swap them

    end = end - 1
```

Now we consider "for each pair of items in the unsorted part of the list". Here, we compare each item to the one that follows it. The last pair we compare is lst[end – 1] and lst[end].

```
# The end of the unsorted section.  The largest item will be placed here.
end = len(lst) - 1

# Keep going until there is only one item to consider.
while end != 0:
    # sweep through the list.
    for i in range(0, end):
        if lst[i - 1] > lst[i]:
            swap them

    end = end - 1
```

Swapping is straightforward:

```
# The end of the unsorted section.  The largest item will be placed here.
end = len(lst) - 1

# Keep going until there are either 0 or 1 items to consider.
# (The 0 case is for the empty list.)
```

```
while end > 0:
    # sweep through the list.
    for i in range(0, end):
        if lst[i] > lst[i + 1]:
            tmp = lst[i + 1]
            lst[i + 1] = lst[i]
            lst[i] = tmp

    end = end - 1
```

c and d. (using doctest)

```
def bubble_sort(lst):
    """ (list) -> NoneType

    Reorder the items in L from smallest to largest.

    >>> L = [3, 4, 7, -1, 2, 5]
    >>> bubble_sort(L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    >>> L = []
    >>> bubble_sort(L)
    >>> L
    []
    >>> L = [1]
    >>> bubble_sort(L)
    >>> L
    [1]
    >>> L = [2, 1]
    >>> bubble_sort(L)
    >>> L
    [1, 2]
    >>> L = [1, 2]
    >>> bubble_sort(L)
    >>> L
    [1, 2]
    >>> L = [3, 3, 3]
    >>> bubble_sort(L)
    >>> L
    [3, 3, 3]
    >>> L = [-5, 3, 0, 3, -6, 2, 1, 1]
    >>> bubble_sort(L)
    >>> L
    [-6, -5, 0, 1, 1, 2, 3, 3]
    """

    # The end of the unsorted section.  The largest item will be placed here.
    end = len(lst) - 1

    # Keep going until there are either 0 or 1 items to consider.
    # (The 0 case is for the empty list.)
    while end > 0:
        # sweep through the list.
        for i in range(0, end):
            if lst[i] > lst[i + 1]:
```

```
                tmp = lst[i + 1]
                lst[i + 1] = lst[i]
                lst[i] = tmp

        end = end - 1
```

6.

a.

```
until all items are sorted:
    # sweep through the list from the end to the beginning.
    for each pair of items in the unsorted part of the list:
        if the pair is out of order:
            swap them
```

b.

After the first sweep, the largest item must have been swapped to the front of the list. After the second sweep, the second-largest item must have been swapped to the second spot. So there is a sorted section accumulating at the front of the list, and we'll keep track of the beginning of the unsorted section.

```
# The beginning of the unsorted section.  The next-smallest item will be
placed here.
beginning = 0

# Keep going until there is only one item to consider.
while beginning < len(lst):
    # sweep through the list from the beginning to the front.
    for each pair of items in the unsorted part of the list:
        if the pair is out of order:
            swap them

    beginning = beginning + 1
```

Now we consider "for each pair of items in the unsorted part of the list". Here, we compare each item to the one that precedes it. The last pair we compare is lst[beginning + 1] and lst[beginning].

```
# The beginning of the unsorted section.  The largest item will be placed
here.
beginning = 0

# Keep going until there is only one item to consider.
while beginning < len(lst):
    # sweep through the list.
    for i in range(len(lst) - 1, beginning, -1):
        if lst[i] < lst[i - 1]:
            swap them

    beginning = beginning + 1
```

Swapping is straightforward:

```
# The beginning of the unsorted section.  The largest item will be placed
here.
beginning = 0

# Keep going until there is only one item to consider.
while beginning < len(lst):
    # sweep through the list.
    for i in range(len(lst) - 1, beginning, -1):
        if lst[i] < lst[i - 1]:
            tmp = lst[i - 1]
            lst[i - 1] = lst[i]
            lst[i] = tmp

    beginning = beginning + 1
```

c and d. (using doctest)

```
def bubble_sort(lst):
    """ (list) -> NoneType

    Reorder the items in L from smallest to largest.

    >>> L = [3, 4, 7, -1, 2, 5]
    >>> bubble_sort(L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    >>> L = []
    >>> bubble_sort(L)
    >>> L
    []
    >>> L = [1]
    >>> bubble_sort(L)
    >>> L
    [1]
    >>> L = [2, 1]
    >>> bubble_sort(L)
    >>> L
    [1, 2]
    >>> L = [1, 2]
    >>> bubble_sort(L)
    >>> L
    [1, 2]
    >>> L = [3, 3, 3]
    >>> bubble_sort(L)
    >>> L
    [3, 3, 3]
    >>> L = [-5, 3, 0, 3, -6, 2, 1, 1]
    >>> bubble_sort(L)
    >>> L
    [-6, -5, 0, 1, 1, 2, 3, 3]
    """

    # The beginning of the unsorted section.  The largest item will be placed
```

```
    # here.
    beginning = 0

    # Keep going until there is only one item to consider.
    while beginning < len(lst):
        # sweep through the list.
        for i in range(len(lst) - 1, beginning, -1):
            if lst[i] < lst[i - 1]:
                tmp = lst[i - 1]
                lst[i - 1] = lst[i]
                lst[i] = tmp

        beginning = beginning + 1
```

7.

The results:

```
    n     bubble  select  insert  builtin
=======================================
   10        0.0     0.0     0.0     0.0
 1000       85.7    78.6    31.6     0.2
 2000      332.2   317.0   128.2     0.4
 3000      792.5   683.8   285.8     0.6
 4000     1400.3  1246.1   517.6     0.9
 5000     2111.7  1968.4   836.2     1.2
10000     8762.9  7625.9  3118.3     2.5
```

Bubble sort is the worst of all of them. This is probably because selection sort performs 1 swap to put an item in its place, but bubble sort may have many swaps on a single sweep.

8.

On the first iteration, binary search is performed on only 2 items; on each iteration, the size of the sublist being searched grows by one. So N log_2 N is an upper bound on the estimate of the overall running time, but not an egregious one.

9.

A third way to do it, as practiced by many novice programmers (including ourselves back when we started!) is to make a guess, and then keep fussing with the code until it works. We have learned by experience that this almost never saves time, and often can triple or quadruple the amount of time we spend.

We personally prefer the first version, because there are only a small number of ways for the loop to be done, but many ways for it to continue.

10.

There is an error in this question: it should be about function merge on page 261, not function mergesort.

In order to ensure that no extend call is needed, the new version of the loop must process every element in both L1 and L2. The only way for the loop to be terminated, then, is when i1 len(L1) and i2 len(L2). We continue as long as that is not true:

```
    newL = []
    i1 = 0
    i2 = 0

    # For each pair of items L1[i1] and L2[i2], copy the smaller into newL.
    while not (i1 == len(L1) and i2 == len(L2)):
        # append the smaller of L1[i1] and L2[i2] to newL; if one of the
lists has
        # no items left, copy from the other one.
```

There are now two reasons to append L1[i1] to newL: either i2 == len(L2), or both i1 and i2 are still valid and L1[i1] <= L2[i2]. In all other cases, we append L2[i2] to newL:

```
    newL = []
    i1 = 0
    i2 = 0

    # For each pair of items L1[i1] and L2[i2], copy the smaller into newL.
    while not (i1 == len(L1) and i2 == len(L2)):
        if i2 == len(L2) or \
           (i1 != len(L1) and L1[i1] <= L2[i2]):
            newL.append(L1[i1])
            i1 += 1
        else:
            newL.append(L2[i2])
            i2 += 1

    return newL
```