Sction B3 Sorting

Bubble Sort

Bubble sort is the simplest and slowest algorithm used for sorting. As its worst-case performance is O(n^2), it should be used for small datasets. For a list of size n, bubble sort will have n-1 passes - each of which will focus only on unsorted data.
On pass one, the highest value in the list is repeatedly swapped with adjacent neighbours until it is at the end of the list.

```
def bubble_sort(lst):
    last_element_index = len(lst) - 1
    for pass_number in range(last_element_index, 0, -1):
        for index in range(pass_number):
            if lst[index] > lst[index + 1]:
                lst[index], lst[index + 1] = lst[index + 1], lst[index]
        print(f"pass: {-(pass_number - len(lst))}, list: {lst}")

print(f"pass: 0, list: {data}")
bubble_sort(data)
```

with output

```
pass: 0, list: [27, 13, 23, 1, 12, 14]
pass: 1, list: [13, 23, 1, 12, 14, 27]
pass: 2, list: [13, 1, 12, 14, 23, 27]
pass: 3, list: [1, 13, 12, 14, 23, 27]
pass: 4, list: [1, 12, 13, 14, 23, 27]
pass: 5, list: [1, 12, 13, 14, 23, 27]
```

Selection Sort

```
def selection_sort(lst):
    for slot in range(len(lst) - 1, 0, -1):
        max_index = 0
        for location in range(1, slot + 1):
            if lst[location] > lst[max_index]:
                max_index = location
        lst[slot], lst[max_index] = lst[max_index], lst[slot]
        print(f"filling_slot: {-(slot - len(lst))}, list: {lst}")
```

with output

```
start:           list: [27, 12, 23, 1, 19, 14]
filling_slot: 1, list: [14, 12, 23, 1, 19, 27]
filling_slot: 2, list: [14, 12, 19, 1, 23, 27]
filling_slot: 3, list: [14, 12, 1, 19, 23, 27]
filling_slot: 4, list: [1, 12, 14, 19, 23, 27]
filling_slot: 5, list: [1, 12, 14, 19, 23, 27]
```

Selection sort's worst-case performance is O(n^2). Its worst performance is similar to bubble sort and it should not be used for sorting larger datasets. Still, selection sort is a better designed algorithm than bubble sort and its average performance is better than bubble sort due to the reduction in the number of exchanges.

Quicksort

Divide and Conquer
Quicksort operates by selecting a 'pivot' value from the data and partitioning the data into two sub-arrays on either side of the selected pivot. Each of the sub-arrays are then sorted recursively using the same approach. Sorting is done "in place" by exchange rather than by copying to a destination array. As such, it is near optimally efficient in terms of space complexity - O(1).
While, in the worst case, quicksort has O(n^2) time complexity, on average, the algorithm has O(n log n) complexity (depending on how ordered the data is).

A pseudocode rendering of the algorithm might look like:

1. If the data to be sorted has less than two elements, return.
2. Pick a value called a pivot in the data following a partitioning scheme (discussed below).
3. Partition the range by reordering elements so that all elements with value less than the selected pivot are positioned before the pivot (to the left in the GIF above) and all elements greater than the pivot are after it (to the right in the GIF).
4. Now recursively apply the algorithm to sort the two ranges either side of the pivot location.

Pivot Points
Selection of a pivot point follows one of several partition schemes:

- choose the last (or first) element of the array (Lomuto partition scheme),
- choose the middle element,
- choose a random element,
- choose the median of the first, middle and last element (so called

"median of three").

```
def quicksort(array, first, last):
    '''
    Sort the range array[first, last] in-place with vanilla QuickSort

    :param array:  the list of numbers to sort
    :param first: the first index from array to begin sorting from,
                must be in the range [0, len(array))
    :param last: the last index from array to stop sorting at
                must be in the range [first, len(array))
    :return:    nothing, the side effect is that array[first, last] is sorted
    '''
    if first >= last:
        return

    i, j = first, last
    # Selected with Lomuto partition scheme
    pivot = array[last]

    while i <= j:
        while array[i] < pivot:
            i += 1
        while array[j] > pivot:
            j -= 1

        if i <= j:
            array[i], array[j] = array[j], array[i]
            i, j = i + 1, j - 1
    quicksort(array, first, j)
    quicksort(array, i, last)
```

```
>>> data = [10, 9, 5, 2, 6, 4, 7, 1]
>>> qsort(data, 0, len(data) - 1)
>>> print (data)
```

```
[1, 2, 4, 5, 6, 7, 9, 10]
```

Note that the data is sorted in place - there is no returned value from the function.

Insertion Sort and Beyond

Insertion sort is outperformed by quicksort for medium and large arrays of data. But for very small data sets (perhaps 'n' < 1000 or so), insertion sort may outperform quicksort - at least in part because it is iterative rather than recursive and dodges the overhead of recursive calls.
Insertion sort requires a constant, O(1), amount of space (memory) to execute. When input data is reverse sorted, worst case time complexity is O(n2). However, time complexity approaches O(n) when the data is already mostly sorted. In particular, insertion sort has time complexity O(kn) when any element in no more than 'k' places from its sorted position.

A simple pseudocode formulation might look like:

```
# until there is no more unsorted data:
#
#     check current value against the value of the highest sorted element
#
#     if larger
#         consider this to be correctly positioned and go to next iteration
#
#     else
#         remove unsorted value from the input
#         iterate through the sorted data to find correct position
#         shift the larger, sorted data to make room
#         insert the value into the sorted data at the correct location
```

Sorting is typically done "in place", its space complexity is $O(1)$.

```
def insertion_sort (lst):
    '''
    Sort the list in-place with vanilla insertion sort.

    :param list: the list of items to sort
    :return:     nothing, the side effect is that list is sorted
    '''

    # We ignore element 0 as being the only element already (trivially)
    # sorted.
    for i in range (1, len(lst)):

        j = i - 1
        next_element = lst[i]

        while (lst[j] > next_element) and (j >= 0):
            lst[j+1] = lst[j]
            j = j - 1

        lst[j + 1] = next_element
```

Of course, it's possible to write more "pythonically" - although this can obscure the details of the algorithm. This version uses enumerate() cleverly but adopts a subtly different approach to moving the new value into position:

```
def insertion_sort(lst):
    for i, value in enumerate(lst):
        for j in range(i - 1, -1, -1):
            if lst[j] > value:
                lst[j + 1] = lst[j]
                lst[j] = value
```