11.12 Insertion Sort

- Another simple, but inefficient, sorting algorithm
- First iteration takes the second element in the array and, if it's less than the first element, swaps it with the first element
- Second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order
- At the ***i***th iteration of this algorithm, the first ***i*** elements in the original array will be sorted

11.12 Insertion Sort (cont.)

- Consider the following array

```
34 56 14 20 77 51 93 30 15 52
```

- Insertion sort looks at the first two elements of the array, `34` and `56`
  - Already in order, so the algorithm continues
  - If they were out of order, the algorithm would swap them
- Next iteration looks at the third value, `14`
  - Less than `56`, so the program stores `14` in a temporary variable and moves `56` one element to the right
  - The algorithm then checks and determines that `14` is less than `34`, so it moves `34` one element to the right
  - The algorithm has now reached the beginning of the array, so it places `14` in element 0
- The array now is

```
14 34 56 20 77 51 93 30 15 52
```

11.12 Insertion Sort (cont.)

- Next iteration stores `20` in a temporary variable
  - Compares `20` to `56` and moves `56` one element to the right because it's larger than `20`
  - Next, compares `20` to `34`, moving `34` right one element
  - When the algorithm compares `20` to `14`, it observes that `20` is larger than `14` and places `20` in element 1
- The array now is

```
14 20 34 56 77 51 93 30 15 52
```

- Using this algorithm, at the ***i***th iteration, the first ***i*** elements of the original array are sorted, but may not be in their final locations
  - Smaller values may be located later in the array

11.12.1 Insertion Sort Implementation

Note: The last two lines of source code in this example have been modified from the print book so you can execute the example inside the notebook.

Function `insertion_sort`

- In each iteration, variable `insert` holds the value of the element that will be inserted into the sorted portion of the array
- Variable move_item keeps track of where to insert the element
- The nested `while` loop locates the position where `insert`'s value should be inserted
- After the nested loop ends, the next statement inserts the element into place

In [1]:

```
# insertionsort.py
"""Sorting an array with insertion sort."""
import numpy as np
from ch11utilities import print_pass

def insertion_sort(data):
    """Sort an array using insertion sort."""
    # loop over len(data) - 1 elements
    for next in range(1, len(data)):
        insert = data[next]  # value to insert
        move_item = next  # location to place element

        # search for place to put current element
        while move_item > 0 and data[move_item - 1] > insert:
            # shift element right one slot
            data[move_item] = data[move_item - 1]
            move_item -= 1

        data[move_item] = insert  # place inserted element
        print_pass(data, next, move_item)  # output pass of algorithm
```

In [2]:

```
def main():
    data = np.array([34, 56, 14, 20, 77, 51, 93, 30, 15, 52])
    print(f'Unsorted array: {data}\n')
    insertion_sort(data)
    print(f'\nSorted array: {data}\n')
```

In [3]:

```
# call mainto execute the sort
main()
```

```
Unsorted array: [34 56 14 20 77 51 93 30 15 52]

after pass 1: 34  56* 14  20  77  51  93  30  15  52
              --
after pass 2: 14* 34  56  20  77  51  93  30  15  52
              --  --
after pass 3: 14  20* 34  56  77  51  93  30  15  52
              --  --  --
after pass 4: 14  20  34  56  77* 51  93  30  15  52
              --  --  --  --
after pass 5: 14  20  34  51* 56  77  93  30  15  52
              --  --  --  --  --
after pass 6: 14  20  34  51  56  77  93* 30  15  52
              --  --  --  --  --  --
after pass 7: 14  20  30* 34  51  56  77  93  15  52
              --  --  --  --  --  --  --
after pass 8: 14  15* 20  30  34  51  56  77  93  52
              --  --  --  --  --  --  --  --
after pass 9: 14  15  20  30  34  51  52* 56  77  93
              --  --  --  --  --  --  --  --  --

Sorted array: [14 15 20 30 34 51 52 56 77 93]
```

11.12.2 Big O of the Insertion Sort

- Insertion sort also runs in O(n^2) time
- Algorithm contains nested loops
- The outer `for` loop iterates `len(data) - 1` times, inserting an element into the appropriate position among the elements sorted so far
  - `len(data) - 1` is equivalent to $n - 1$ (as `len(data)` is the size of the array)
- The nested `while` loop iterates over the preceding elements in the array
  - Worst case, this loop requires $n - 1$ comparisons
  - Each iteration runs in O(n) time
- In Big O notation, nested loops mean that you must multiply the number of comparisons
  - For each iteration of an outer loop, there will be a certain number of iterations of the inner loop
  - In this algorithm, for each O(n) iterations of the outer loop, there will be O(n) iterations of the inner loop
  - Multiplying these values results in O(n^2)