MONASH
INFORMATION
TECHNOLOGY

FIT9133 Semester 2 2019
Programming Foundations in Python

Week 10:
Searching and Sorting Algorithms

Chunyang Chen

Module 4 Synopsis

- Module 4 is aimed to introduce you with:
  - Python library and packages
    - Standard packages: Math and Random
    - External packages: NumPy, SciPy, Matplotlib, Pandas
  - Searching algorithms
    - Linear Search
    - Binary Search
  - Sorting algorithms
    - Bubble Sort
    - Selection Sort
    - Insertion Sort

Module 4 Learning Objectives

- Upon completing this module, you should be able to:
  - Utilise a number of useful Python packages for scientific computation and basic data analysis
  - Recognise a suitable algorithm for solving a particular computational problem
  - Contrast different algorithms for searching and sorting

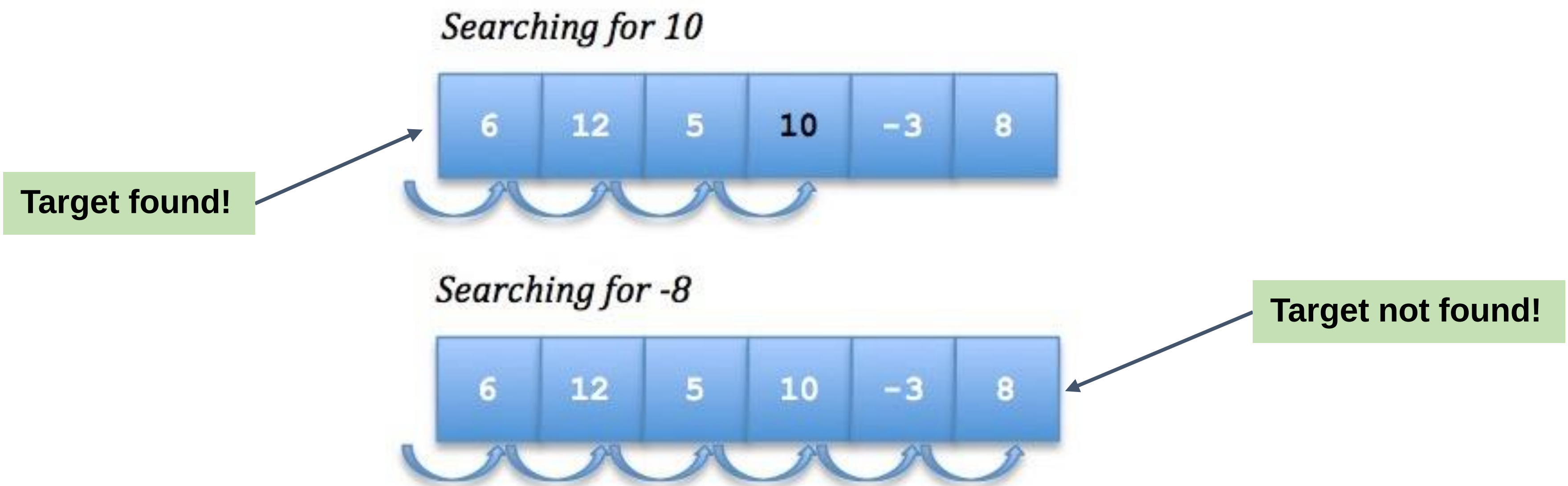
Searching Algorithms

- Searching:
  - A process of finding a particular data item (or a group of data items) within a sequence-based collection based on certain criteria
  - Search criteria are defined by some form of search key
    - Primitive data types: a single key value
    - Complex data types: a number of attributes

- Basic algorithms:
  - Linear Search
  - Binary Search

Linear Search

- Basic concept:
  - Begin with the first item in the collection (e.g. a list)
  - Each item is compared with the "target" item in turn until:
    - The "target" item is found; or
    - The end of the collection is reached (i.e. the "target" item does not exist)



- Implementation:

```
def linear_search(the_list, target_item):

    # obtain the length of the_list
    n = len(the_list)

    for i in range(n):
        # if the target is found
        if the_list[i] == target_item:
            return True

    # search through the list
    # the target is not found
    return False
```

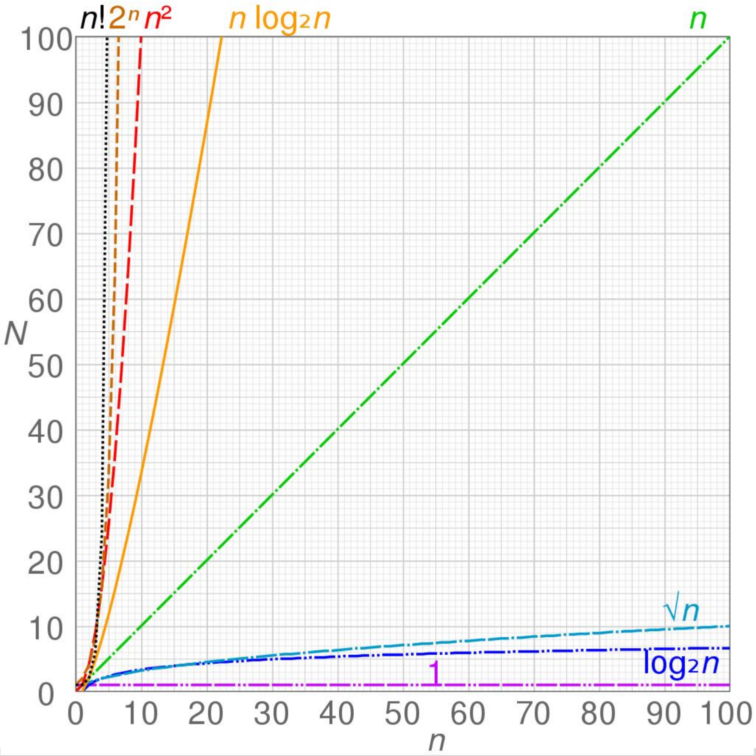
```
>>> char_list = ['p', 'y', 't', 'h', 'o', 'n']
>>> result = linear_search(char_list, 't')
>>> print(result)
>>> True
>>> result = linear_search(char_list, 'T')
>>> print(result)
>>> False
```

Shortcoming of linear search

- Inefficient performance
  - Given a list (n) of items and a query value
    - Best case: 1 operation
    - Worst case: n operations

- The time complexity is the computational complexity that describes the amount of time it takes to run an algorithm.
- Big O notation
  - The Big O notation defines an upper bound of an algorithm
    - O(*1*): constant time
    - O(*n*): linear time
    - O(n^2): quadratic time
    - O(log *n*): logarithmic time
    - …

Not tested!



- Basic concept:
  - Begin by choosing an item that divides the collection (the list) into two halves
  - The “middle” item is compared with the “target” item
  - Three possible conditions:
    - The “middle” item is the “target” item
    - The “target” item is less than the “middle” item
    - The “target” item is greater than the “middle” item
  - If the “target” item < the “middle” item:
    - Search the lower half (excluding the “middle” item)
  - If the “target” item > the “middle” item:
    - Search the upper half (excluding the “middle” item)

Pre-condition: The collection (the list) must be sorted.

(More on) Binary Search

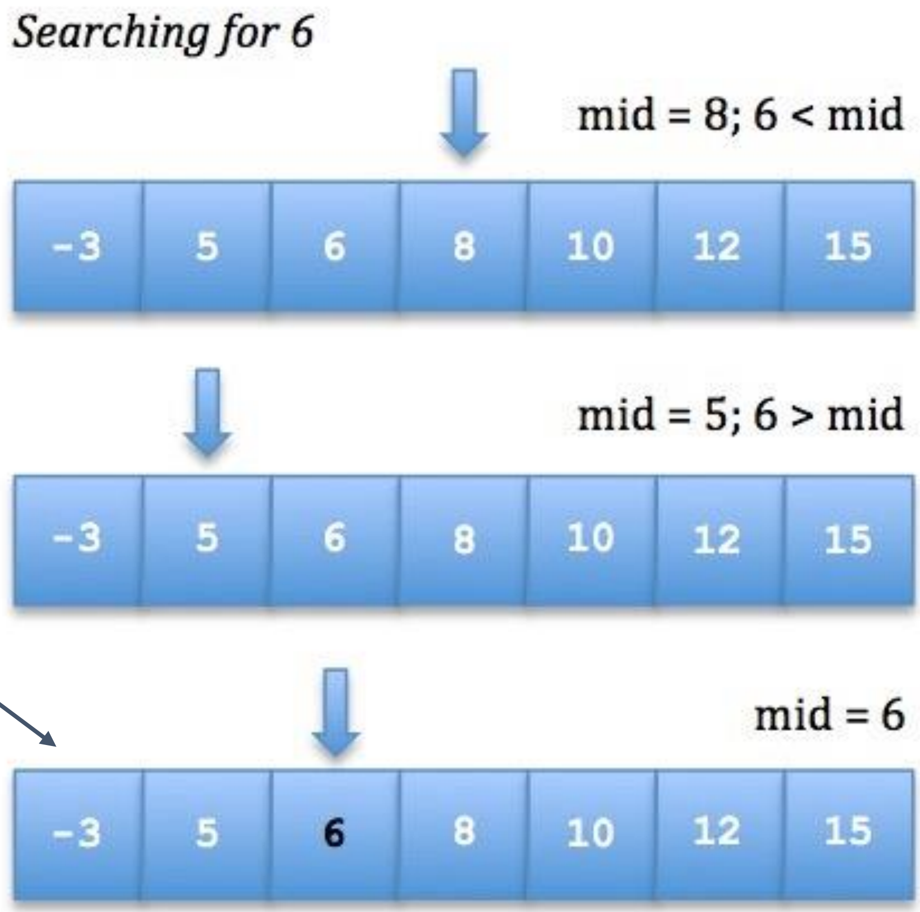
Search for 47

0	4	7	10	14	23	45	47	53

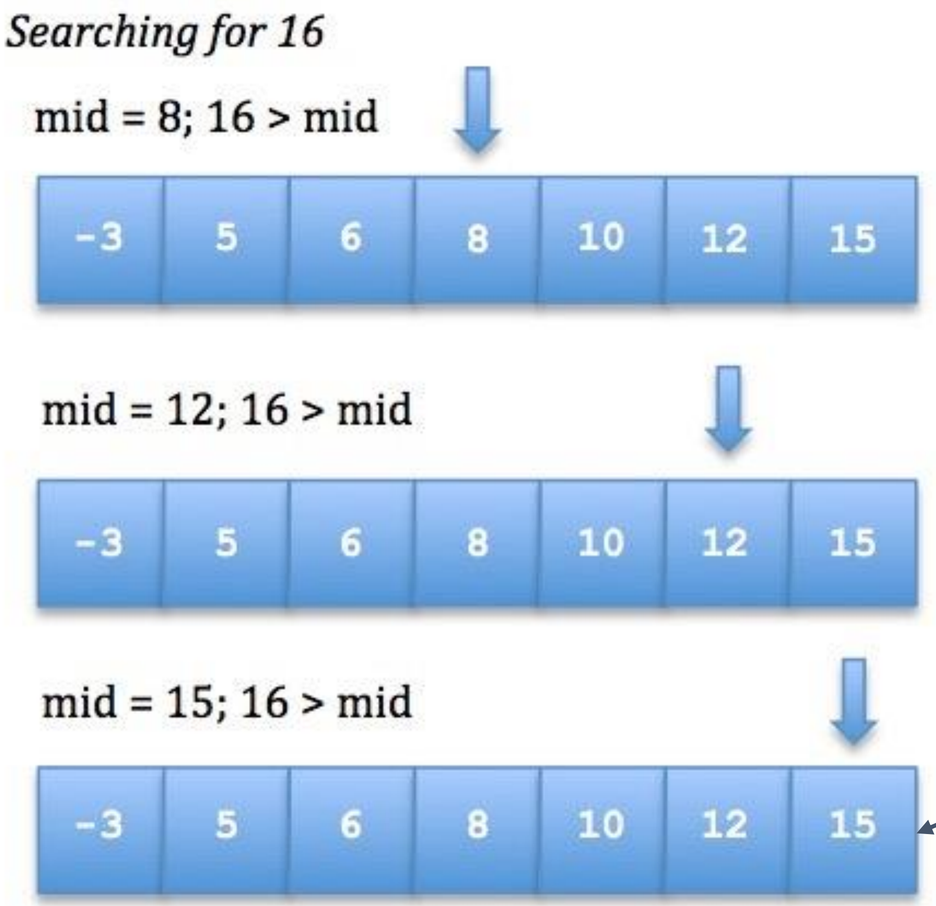
https://brilliant.org/wiki/binary-search/

(More on) Binary Search

Target found!



(More on) Binary Search



Target not found!

- Implementation:

```
def binary_search(the_list, target_item):
    low = 0
    high = len(the_list)-1

    # repeatedly divide the list in half
    # as long as the target item is not found
    while low <= high:

        # find the mid position
        mid = (low + high) // 2

        if the_list[mid] == target_item:
            return True
        elif target_item < the_list[mid]:
            high = mid - 1 # search lower half
        else:
            low = mid + 1  # search upper half

    # the list cannot be further divided
    # the target is not found
    return False
```

Time complexity $O(\log n)$:

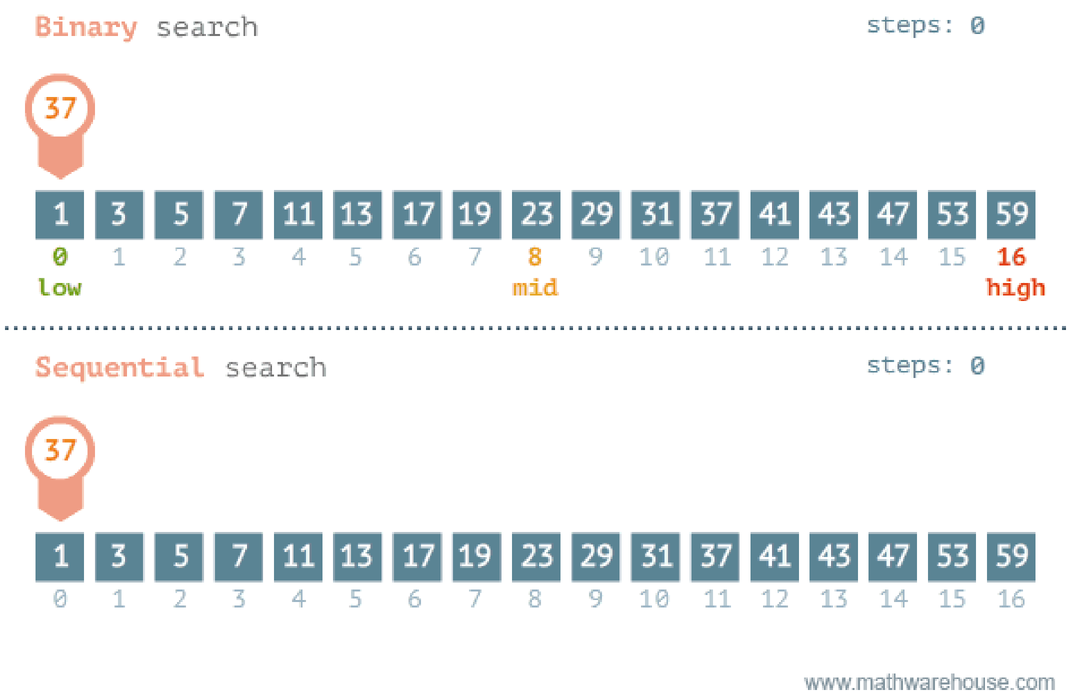
Review Questions:
Part 1

Review Question 1

Given a query 37 to be searched by a sorted list of values:

[1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59],

How many steps does linear search and binary search take?



4 steps !!!

https://www.mathwarehouse.com/programming/gifs/binary-vs-linear-search.php

Sorting Algorithms

Concepts of Sorting

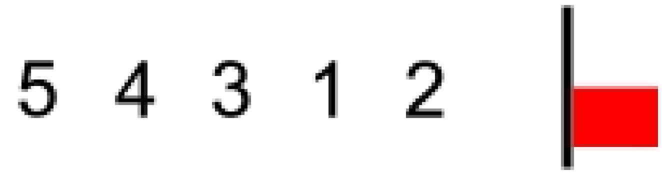
- Sorting:
  - A process of re-ordering or re-arranging data items within a collection based on certain characteristics/attributes
  - Reordering is based on some form of sort key
    - Primitive data types: a single key value
    - Complex data types: a number of attributes
  - Performance is measured by the total numbers of comparison and re-ordering (or swapping) involved

- Basic algorithms:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

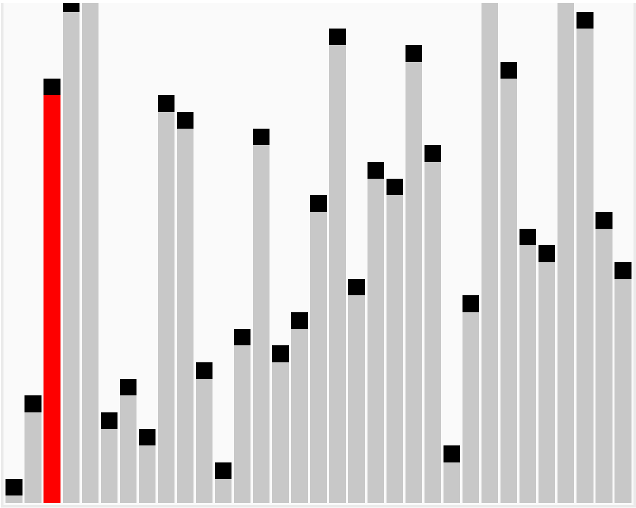
- Basic concept:
  - "Bubble up" larger items to the "top" or the end of the collection
  - "Sink down" smaller items to the "bottom" or the front of the collection
  - Every adjacent pair of items is compared; if out of order, swap them
  - Completing one iteration of traversing the entire collection, the next largest item will be in place
  - n-1 iterations are required for n items in the collection

- Limitation:
  - If the collection is in a completely reverse order, the total number of swaps can be expensive (worst case)

(More on) Bubble Sort

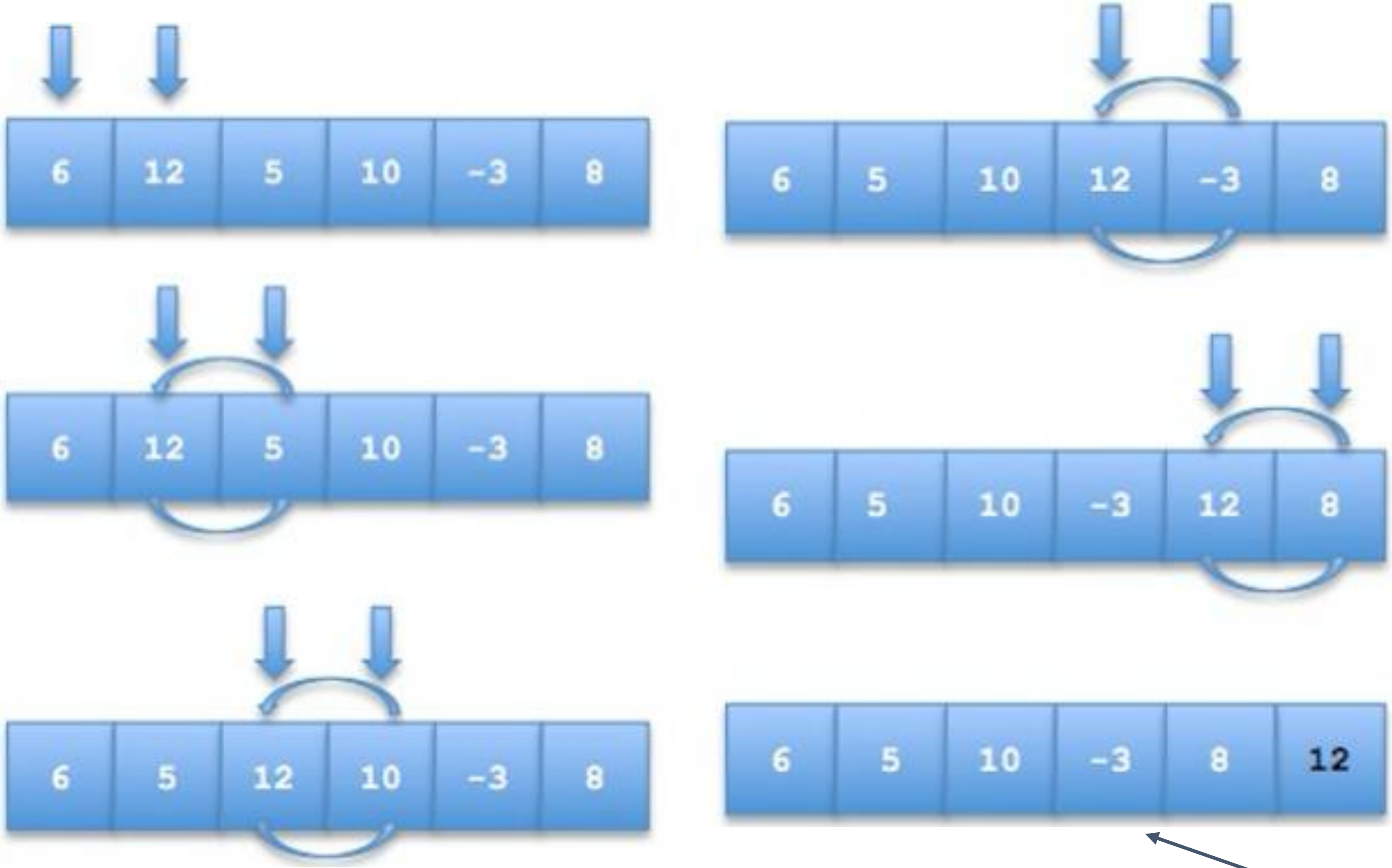


https://commons.wikimedia.org/wiki/File:Bubble_sort_with_flag.gif



https://www.reddit.com/r/gifs/comments/36z58n/bubble_sort/

(More on) Bubble Sort



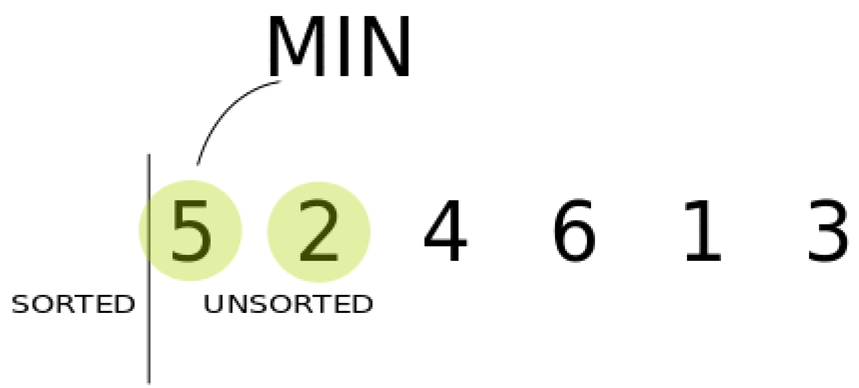
After 1st iterations

- Implementation:

```
def bubble_sort(the_list):

    # obtain the length of the list
    n = len(the_list)

    # perform n-1 iterations
    for i in range(n-1, 0, -1):

        # for each iteration
        # move the next largest item to the end
        for j in range(i):

            # swap if two adjacent items are
            # out of order
            if the_list[j] > the_list[j+1]:
                temp = the_list[j]
                the_list[j] = the_list[j+1]
                the_list[j+1] = temp
    return the_list
```

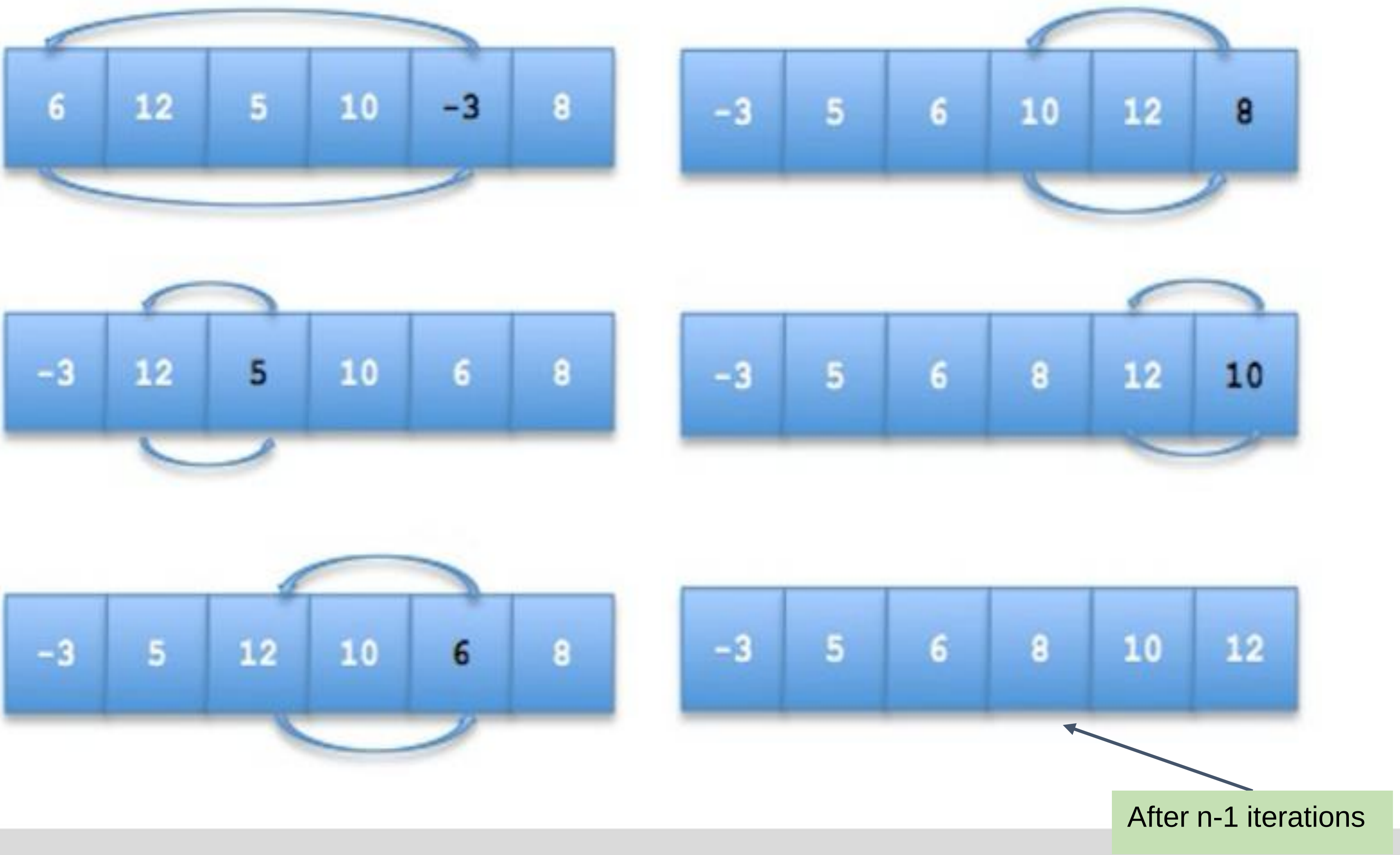
Time complexity
O(n^2)

- Basic concept:
  - Find the next smallest item (or the largest) in each iteration
  - Place the smallest item (or the largest) at the correct position at the end of each iteration
  - Only one swap is required at the end of each iteration
  - n-1 iterations are required for n items in the collection



https://codepumpkin.com/selection-sort-algorithms/

(More on) Selection Sort



- Implementation:

```
def selection_sort(the_list):

    # obtain the length of the list
    n = len(the_list)

    # perform n-1 iterations
    for i in range(n-1):

        # assume item at index i as the smallest
        smallest = i

        # check if any other item is smaller
        for j in range(i+1, n):
            if the_list[j] < the_list[smallest]:
                # update the current smallest item
                smallest = j

        # place the current smallest item
        # in its correct position
        the_list[smallest], the_list[i] = \
            the_list[i], the_list[smallest]
    return the_list
```

Characteristics of Selection Sort

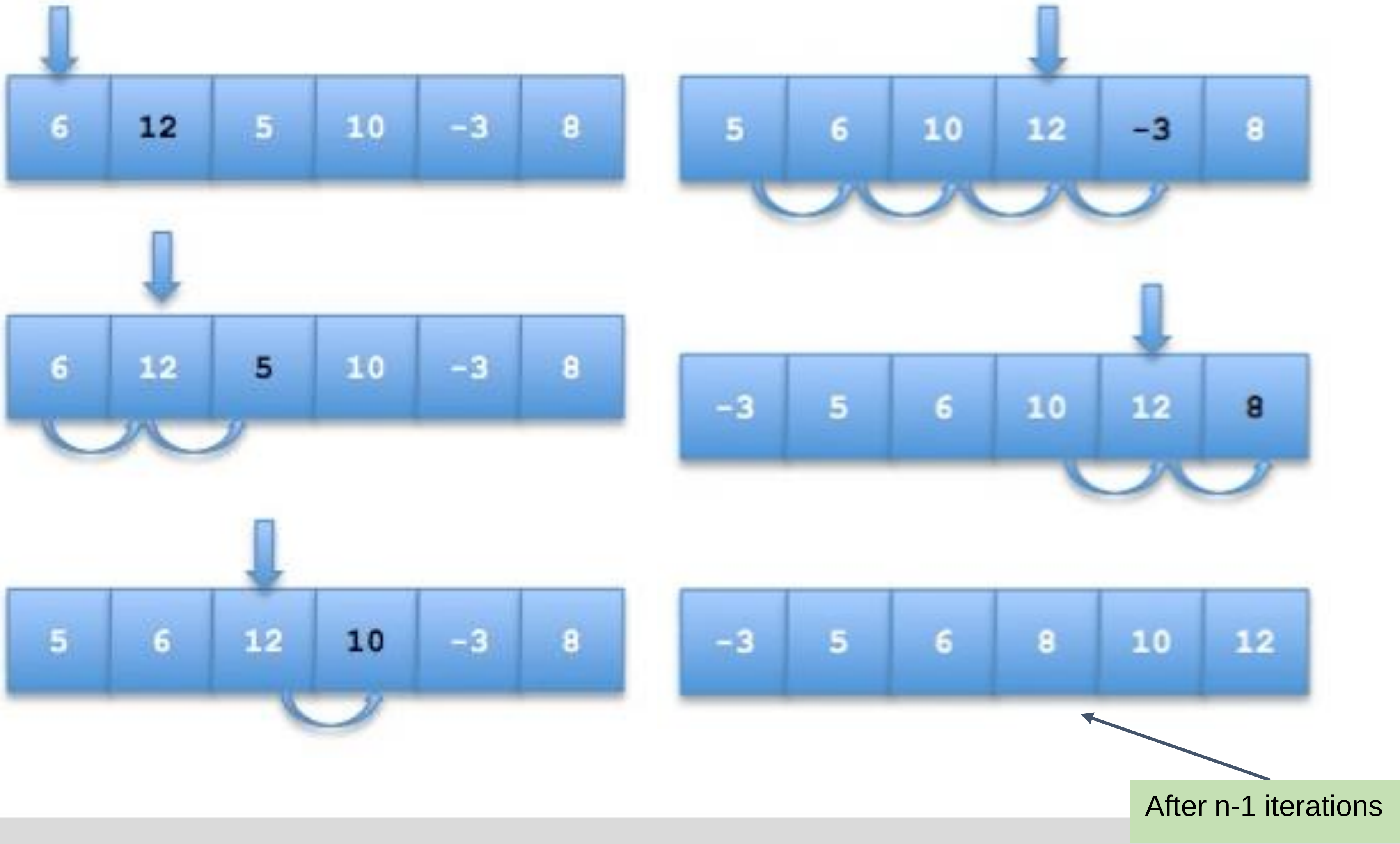
- Comparison with Bubble Sort:
  - Total number of comparisons is the same
  - Total number of swaps is reduced to only one in each iteration
  - Slightly more efficient than Bubble Sort

- Time complexity:
  - O(n^2):

- Basic concept:
  - Maintain two sublists for the collection to be sorted
  - Pick each of the items from the "unsorted" sublist
  - Insert each of these items into the correct position within the "sorted" sublist
  - Shifting (re-ordering) is needed to make "space" for insertion

https://stackoverflow.com/questions/15799034/insertion-sort-vs-selection-sort/15799689#15799689

(More on) Insertion Sort



- Implementation:

```
def insertion_sort(the_list):

    # obtain the length of the list
    n = len(the_list)

    # begin with the first item in the list
    # assume as the only item in the sorted sublist
    for i in range(1, n):

        # indicate the current item to be positioned
        current = the_list[i]

        # find the correct position where the current
        # item should be placed in the sorted sublist
        pos = i
        while pos > 0 and the_list[pos-1] > current:
            # shift items in the sorted sublist
            # for those larger than the current item
            the_list[pos] = the_list[pos-1]
            pos -= 1

        # place the current item in its correct position
        the_list[pos] = current
    return the_list
```

- Comparison with Bubble Sort and Selection Sort:
  - Total numbers of comparison and reordering (shifting) could be reduced in particular if the collection is almost sorted
  - If the collection is already sorted, only n-1 times of comparison is required and no re-ordering is needed
- Time complexity:
  - O(n^2)

Review Questions:
Part 2

A *stable* sorting algorithm preserves the relative order of elements with the same value. Is Selection Sort stable?

```
def selection_sort(the_list):
    n = len(the_list)

    for i in range(n-1):
        smallest = i

        for j in range(i+1, n):
            if the_list[j] < the_list[smallest]:
                smallest = j

        the_list[smallest], the_list[i] = \
            the_list[i], the_list[smallest]
```

A. Yes

B. No

C. Not sure

Would Insertion Sort be more efficient compare to Bubble Sort and Selection Sort if the given list is in the total reverse order?

```
def insertion_sort(the_list):
    n = len(the_list)

    for i in range(1, n):
        current = the_list[i]
        pos = i
        while pos > 0 and the_list[pos-1] > current:
            the_list[pos] = the_list[pos-1]
            pos -= 1

        the_list[pos] = current
```

A. Yes
B. No
C. Not sure

Week 8 Summary

- We have discussed:
  - Searching algorithms
  - Sorting algorithms
- Next week:
  - Testing and Exception Handling

Please fill in the SETU for giving us feedback