Learning Aims

Understand the need for standard searching algorithms

- Binary search and linear search
- Bubble sort

Implement binary and linear search.

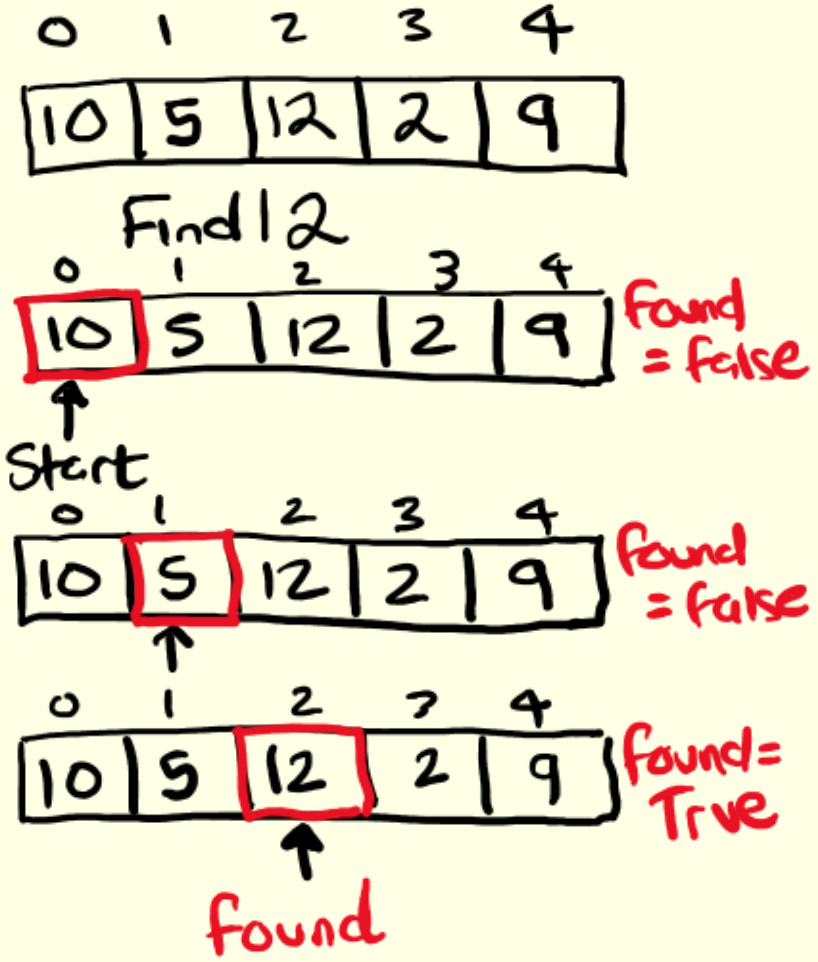
Searching Algorithms

The **linear search algorithm** looks through each item in a data list until a match is found. List does not need to be sorted and better with smaller lists

The **binary search algorithm** needs a sorted data list, then keeps splitting it in half until a match is found. Must be sorted. Better with large datasets.

Linear Search

- Starting at the first element
- Each item is checked
- until value is found
- or end of list reached and not found

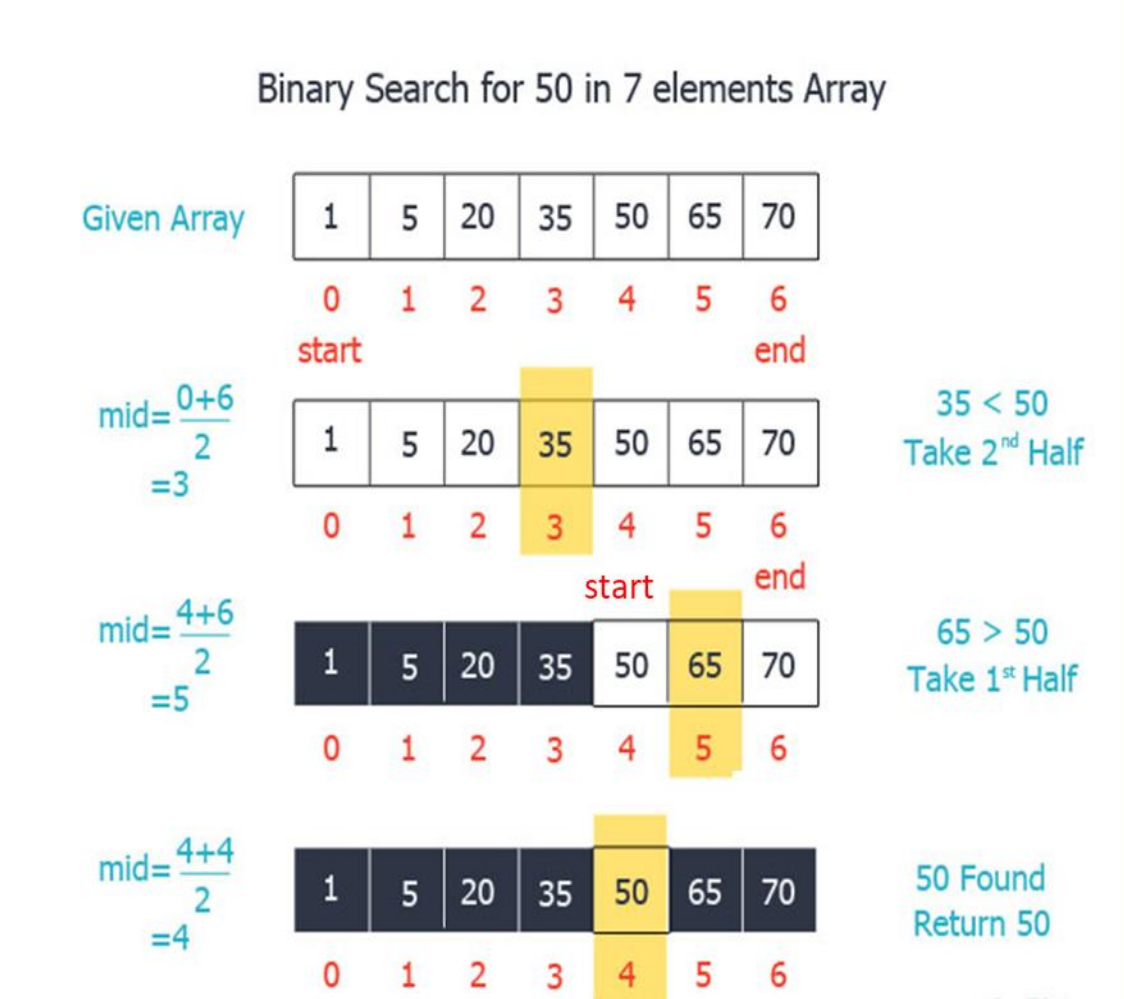


Linear Search Algorithm

```
alist = [‘A’, ‘F’, ‘B’, ‘E’, ‘D’,’G’,’C’]
position = 0
found = False
item =input()
WHILE position < len(alist) AND found == False
    IF  item == alist[position] then
            print (“Item found”)
            found = True
     ELSE
            position = position + 1
     ENDIF
END WHILE
If found == False then
     print(“Item not found”)
ENDIF
```

Binary Search Algorithm

- Binary can only search a list that has been **sorted.**
- It operates by finding the **midpoint**.
- It can then discard the side without the target.
- This finds the midpoint in the side with the target and discards half.
- This process continues until the target has been found.
- Binary Searching is quicker than linear.



Binary search algorithm

```
alist = [1,2,5,7,11,14]
item= input()
found = False
first = 0
last = len(alist) – 1
WHILE found = False AND first <= last
    midPoint = (first + last) DIV 2
    IF item ==alist[midpoint] then
            print (“item found at location”, midpoint)
            found = True
    ELSE
            IF item < alist[midpoint] then
                    last = midpoint - 1
            ELSE
                    first = midpoint + 1
            END IF
      END IF
END WHILE
if found == False then
        print(“Item not found”)
```

DIV is used so that it returns a whole number.
It will round down.
In python you use //

Model Answer

Show how a binary search would be performed on the array shown in Fig. 4.2 to find the value ‘duck’. [3]

midpoint = 0+8
= 4

wolf	monkey	lion	iguana	goat	giraffe	frog	elephant	duck

midpoint = 5+8
= 6

wolf	monkey	lion	iguana	goat	giraffe	frog	elephant	duck

Midpoint = 7+8
= 7

wolf	monkey	lion	iguana	goat	giraffe	frog	elephant	duck

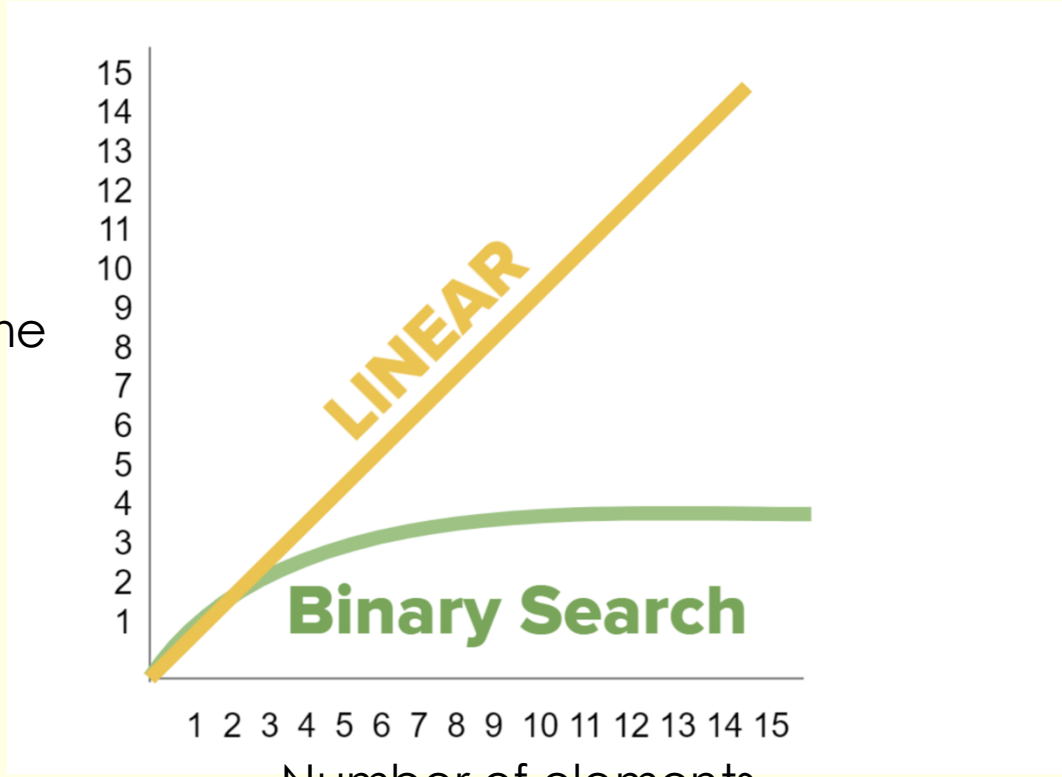
midpoint = 8+8
= 8

wolf	monkey	lion	iguana	goat	giraffe	frog	elephant	duck

Time Efficiency

Worst Case - Time complexity of linear search is O(n) - number of elements in a list or array increase, the number of comparisons also increases linearly for a linear search algorithm

Binary search has Worst case time complexity of O(log n) – meaning it runs in logarithmic time in the worst case. Binary search is faster than linear search except for small arrays.



Learning Aims

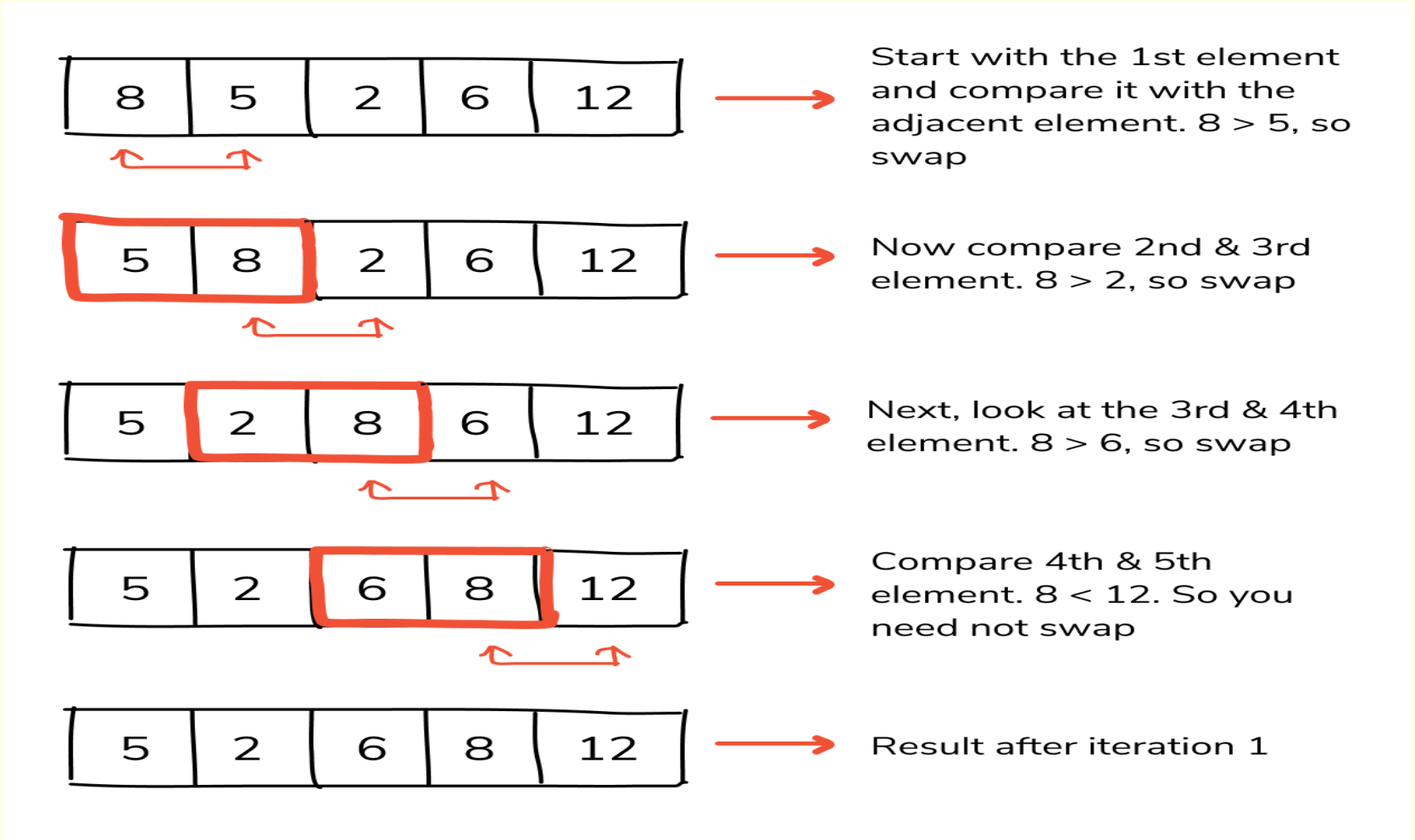
Understand the need for standard sorting algorithms

- BinaBubble sort
- Insertion sort
- Merge sort
- Quick sort

Implement bubble sort and insertion sort.

Bubble Sort

The **bubble sort algorithm** sorts a list of data by comparing each pair of values in sequence and swapping them if necessary. It keeps doing this until the list is sorted. It's easy to program but it's very slow.



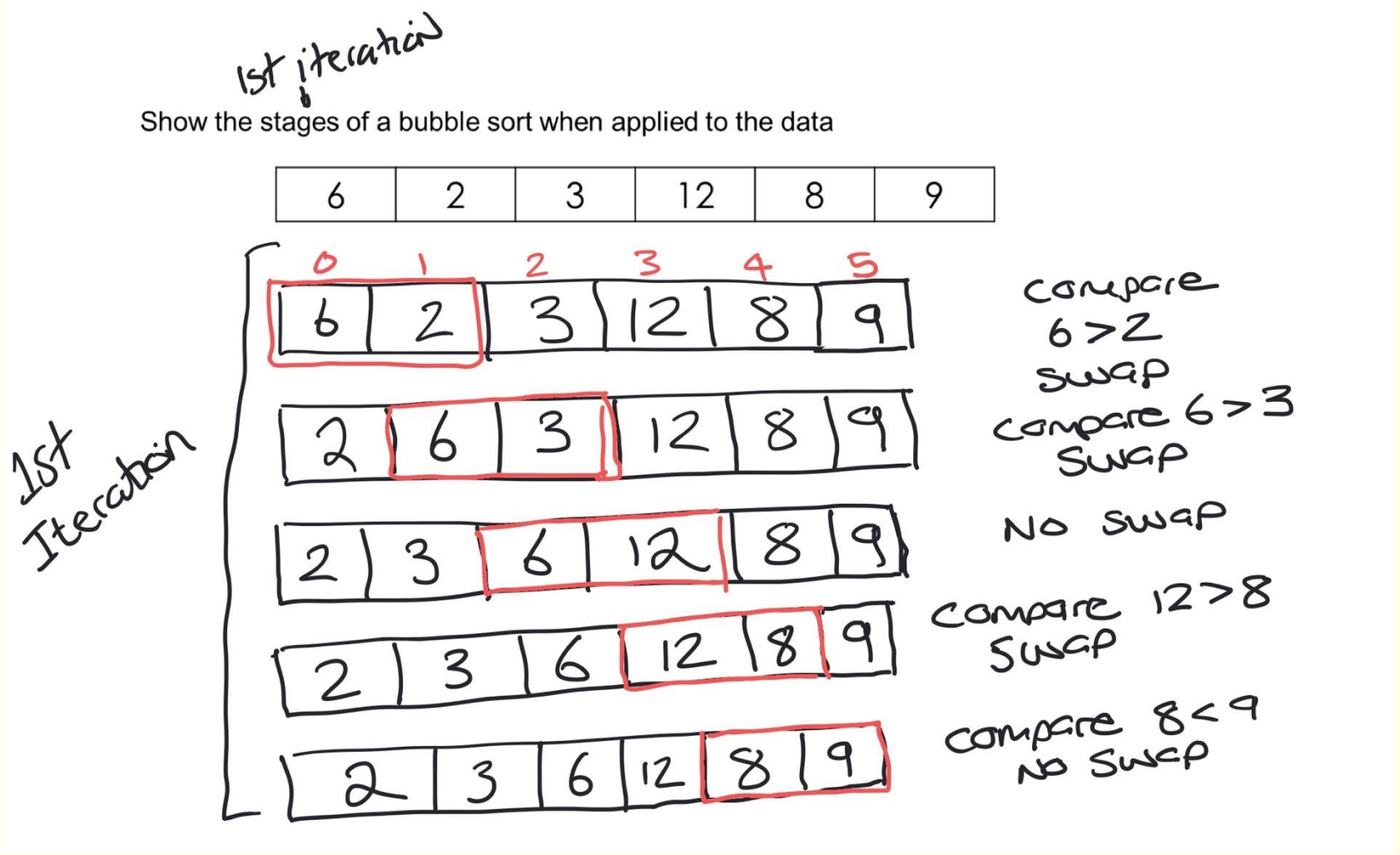
Bubble Sort Code

```
alist = [4, 2, 10, 5, 6]
swap = True
while swap == True
    swap = False
    for pos in range (0,len(alist)-1)
        if alist[pos] > alist[pos+1] then
            temp = alist[pos]
            alist[pos] = alist[pos+1]
            alist[pos+1] = temp
            swap = True
        endif
    next pos
endwhile
print(alist)
```

Will continue to loop through the list until there has not been any swaps

Swap items using a temp variable

Example Exam Question Solution



Bubble Sort Summary

+ The primary advantage of the bubble sort is that it is popular and easy to implement.

+ In the bubble sort, elements are swapped in place without using additional temporary storage.

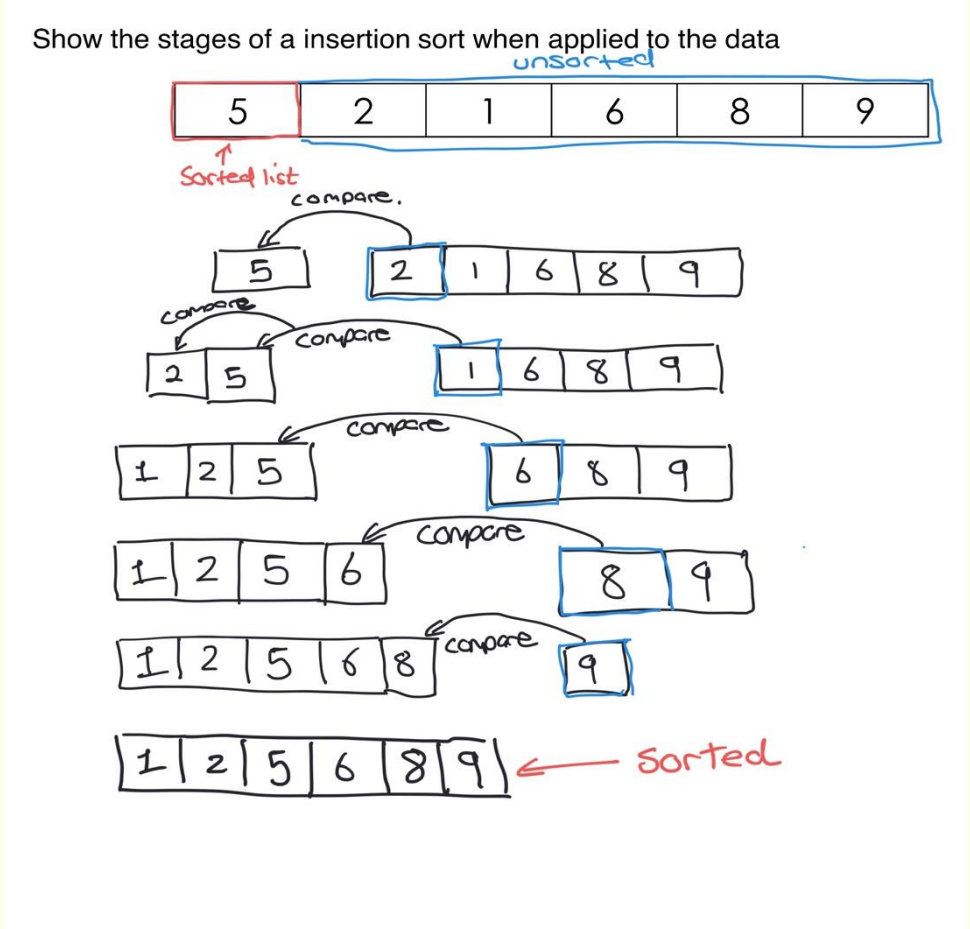
+The space requirement is at a minimum

- The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.

The **insertion sort algorithm** sorts a list of data by taking each item in sequence, moving along the rest of the list, and inserting the item in the correct place. It's fairly fast.

Sorted and unsorted sections of the list are separated by a pointer.

The current item compared with each item in the sorted section



Compare 2 with 5 swap

Compare with 1 with 5 and swap, Compare with 2 and swap

Compare with 6 with 5 no swap

Compare with 6 with 5 no swap

Compare with 9 with 8 no swap

Insertion Sort Algorithm

```
# Traverse through 1 to len(arr)
FOR pos = 1 to lengthOfarray:
    currentValue = array[position]


# Swap the currentValue with the next item along the
sorted list - swapping if next item in the sorted part of the list
> currentValue

    WHILE pos > 0 AND array[pos-1] > currentValue:
        array[pos] = array[pos-1]
        pos = pos – 1
     End while

     array[pos] = currentValue

next position
```

Starts at the second item and places it into a sorted sequence by performing consecutive swaps (within the while loop) until every item has been inserted, leaving behind a sorted array.

Insertion sort is, like bubble sort, a slow algorithm.

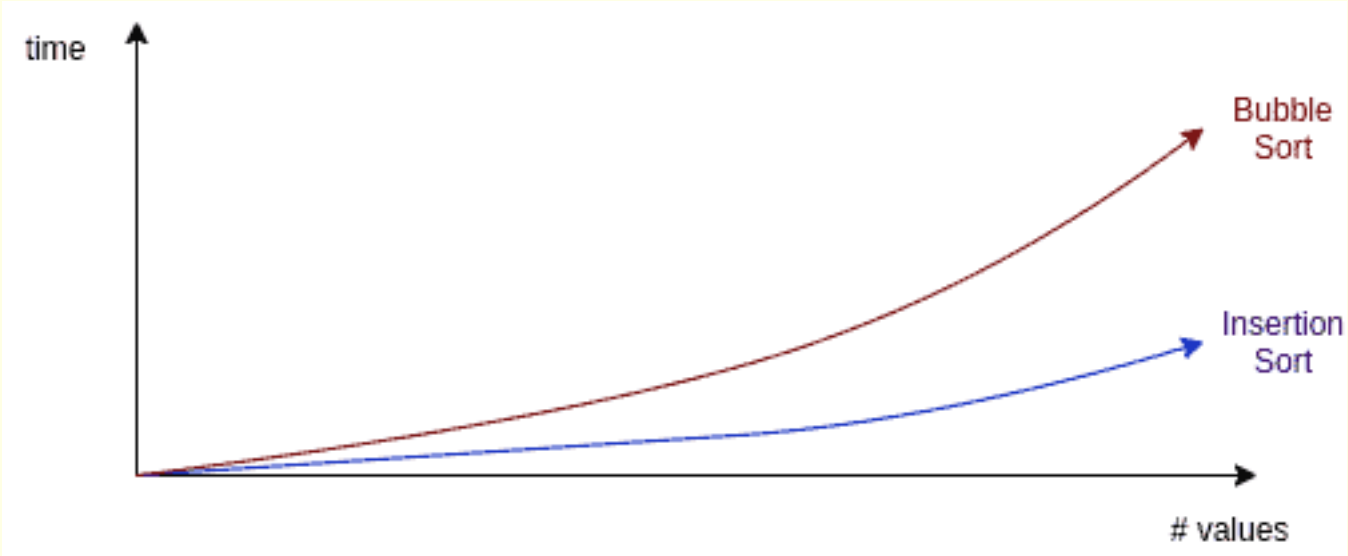
Insertion Sort Summary

+ The main advantage of the insertion sort is its simplicity.
+ It also exhibits a good performance when dealing with a small list.
+ The insertion sort is an in-place sorting algorithm, so the space requirement is minimal.
+ Insertion is good for lists that are almost sorted
+ Good for inserting new values into a sorted list
++More efficient than a bubble sort

Time complexity of bubble and insertion sorts

The bubble sort requires close to n passes through the list, with each pass requiring a maximum of n – 1 swaps. It is of order O(n2).

The insertion sort also has two nested loops and so has time complexity O(n2)
Already almost sorted, the time complexity is reduced to close to O(n).



Merge Sort Algorithm

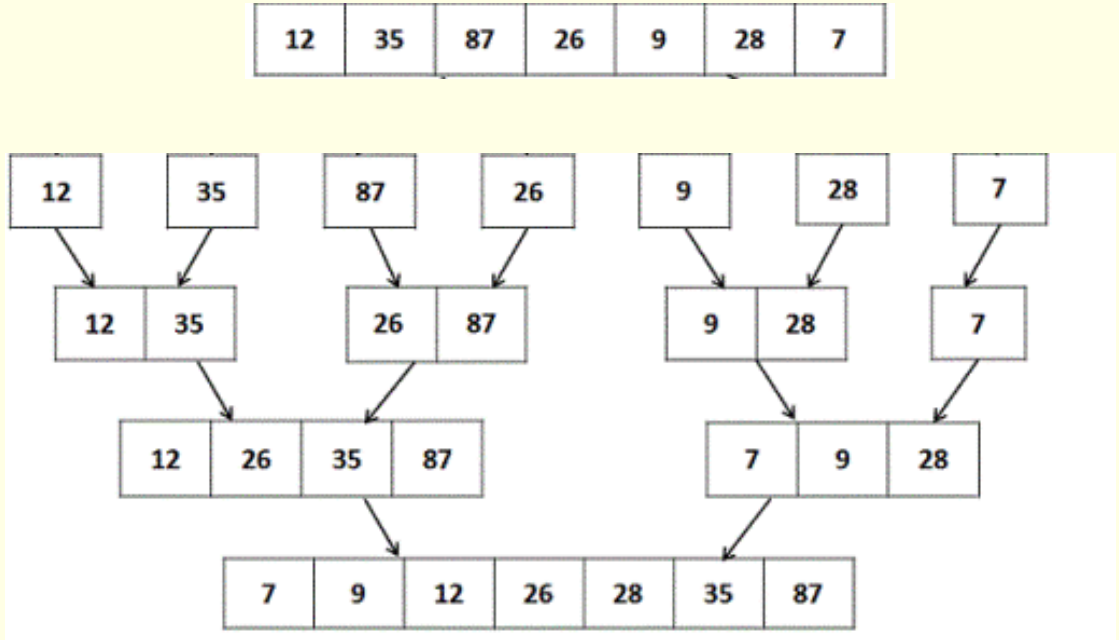
Merge Sort – This method breaks a list down into it's component parts and then builds the list up in order.

Step 1: Split the arrays into sub-arrays of 1 element.

Step 2: Take each sub-array and merge into a new, sorted array.

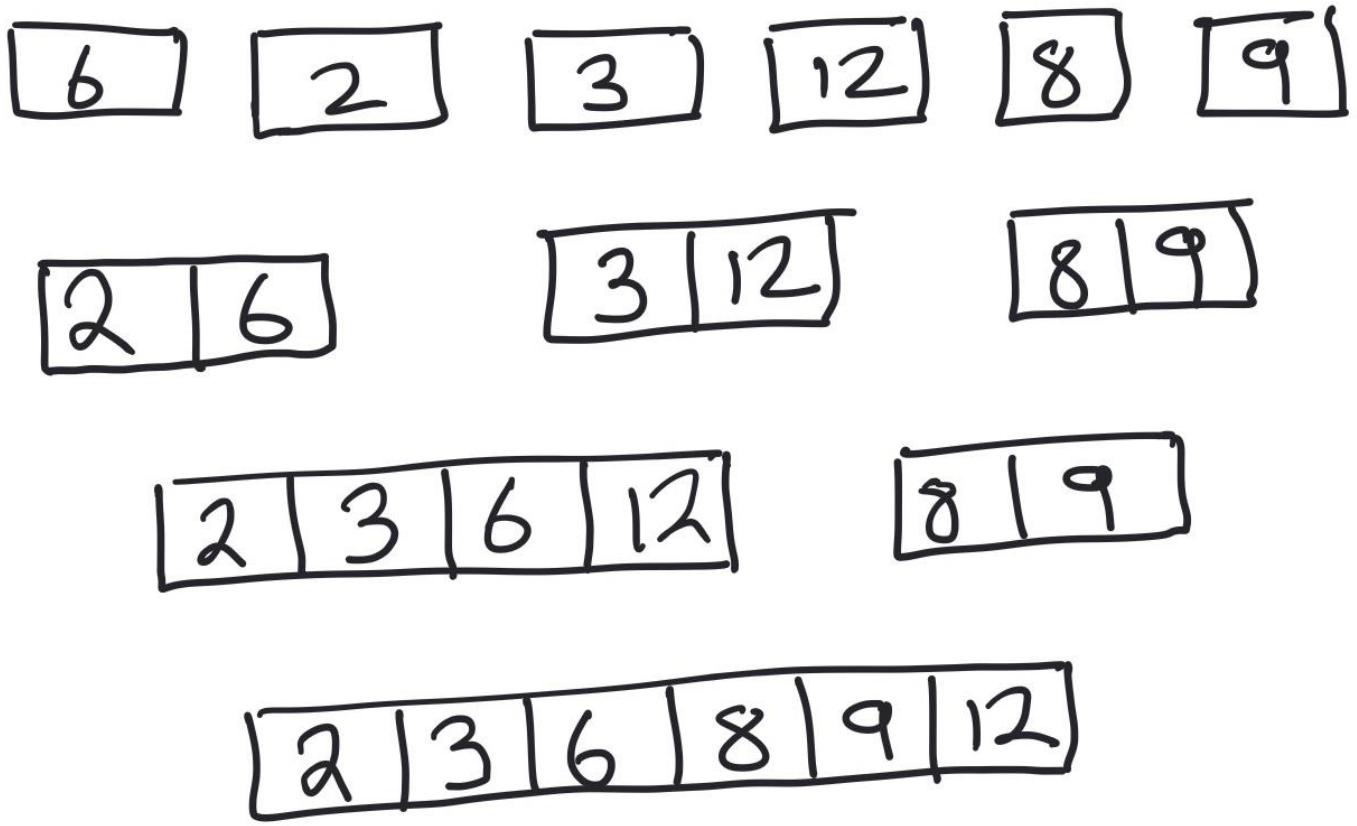
Step 3: Repeat this process until a final, sorted array is produced.

Output: A sorted array



Show the stages of a merge sort when applied to the data

6	2	3	12	8	9



Merge Sort Algorithm:

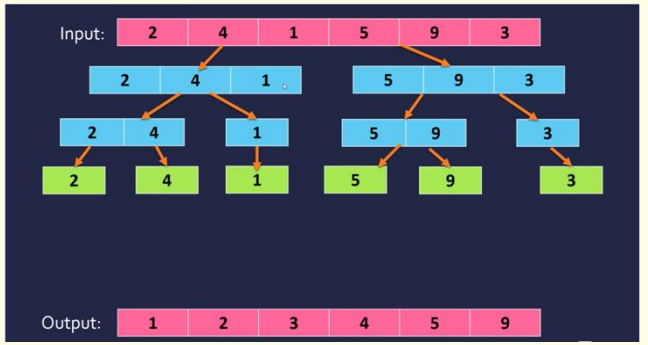
Merge sort is formed from two functions. One called **MergeSort** and another called **Merge**.

The algorithm for the merge_sort subroutine is recursive, used to divide the list until only length of 1

It uses the **merge** subroutine that merge the sublists that are created as the algorithm progresses.

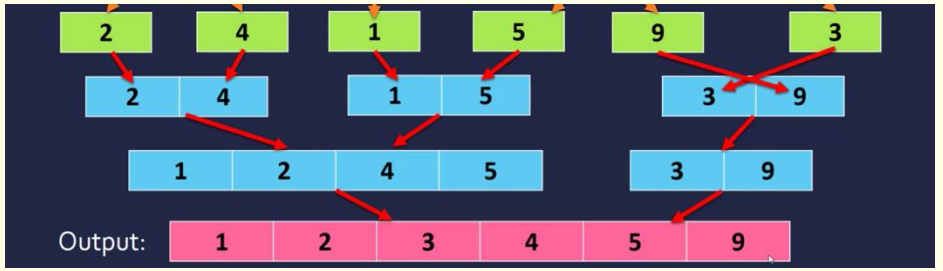
```
SUBROUTINE merge_sort(items)
    # Base case for recursion:
    # The recursion will stop when the list has been divided into single items
    if len(items) > 1 then
        midpoint = len(items) DIV 2 #div is // in python
        left_half = items[0:midpoint]   # Create left half list
        right_half = items[midpoint:LEN(items)]   # Create right half list

        merge_sort(left_half)  # Recursive call on left half
        merge_sort(right_half)  # Recursive call on right half

        merge(items, left_half, right_half)  # Call procedure to merge both halves
    ENDIF
ENDSUBROUTINE
```

MergeSort divides its input into two parts and recursively calls MergeSort on each of those two parts until they are of length 1 at which point Merge is called.



Merge puts groups of elements back together in a special way, ensuring that the final group produced is sorted.

merged - the list that will hold the ordered items after merging the two lists
list_1 - the first list to be merged
list_2 - the second list to be merged




```
SUBROUTINE merge (merged, list1, list2)

    index_1 = 0  # list_1 current position
    index_2 = 0  # list_2 current position
    index_merged = 0  # merged current position

    # While there are still items to merge
    WHILE index_1 < LEN(list_1) AND index_2 < LEN(list_2)

        # Find the lowest of the two items being compared
        # and add it to the new list
        IF list_1[index_1] < list_2[index_2] THEN
            merged[index_merged] = list_1[index_1]
            index_1 = index_1 + 1
        ELSE
            merged[index_merged] = list_2[index_2]
            index_2 = index_2 + 1
        ENDIF
        index_merged = index_merged + 1
    ENDWHILE

    # Add to the merged list any remaining data from list_1
    WHILE index_1 < len(list_1)
        merged[index_merged] = list_1[index_1]
        index_1 = index_1 + 1
        index_merged = index_merged + 1
    ENDWHILE

    # Add to the merged list any remaining data from list_2
    WHILE index_2 < len(list_2)
        merged[index_merged] = list_2[index_2]
        index_2 = index_2 + 1
        index_merged = index_merged + 1
    ENDWHILE
ENDSUBROUTINE
```

The first while loop
The values from list_1[index_1] and list_2[index_2] are compared
The lower of the two values is written to the merged list merged at the next empty position indicated by index_merged
The pointer for the list that the item came from is incremented by 1;
if the item came from list1, index_1 is incremented by 1;
if the item came from list_2, index_2 is incremented by 1
The comparison and merging continues until one list is completely empty

The second while loop:
Any remaining items from list_1 are added to merged; remember that they are already in the right order. This prevents unnecessary computational steps, as the subroutine doesn't need to compare items that are already sorted.

The third while loop:
Any remaining items from list_2 are added to merged; remember that they are already in the right order.

Time complexity of merge sort

The merge sort is another example of a divide and conquer algorithm, but in this case, there are n sublists to be merged, so the time complexity has to be multiplied by a factor of n.

The time complexity is therefore O(nlog n).

Space complexity

The amount of resources such as memory that an algorithm requires, known as the space complexity, is also a consideration when comparing the efficiency of algorithms. The bubble sort, for example, requires n memory locations for a list of size n.

The merge sort, on the other hand, requires additional memory to hold the left half and right half of the list, so takes twice the amount of memory space.

Merge Sort Summary

- Very fast
- Divide and Conquer Strategy
- Recursive algorithm

+ It can be applied to files of any size.
+Large lists
+Does not have to go through several times like the bubble

- Requires extra space
- Merge Sort requires more space than other sorts.

Quicksort

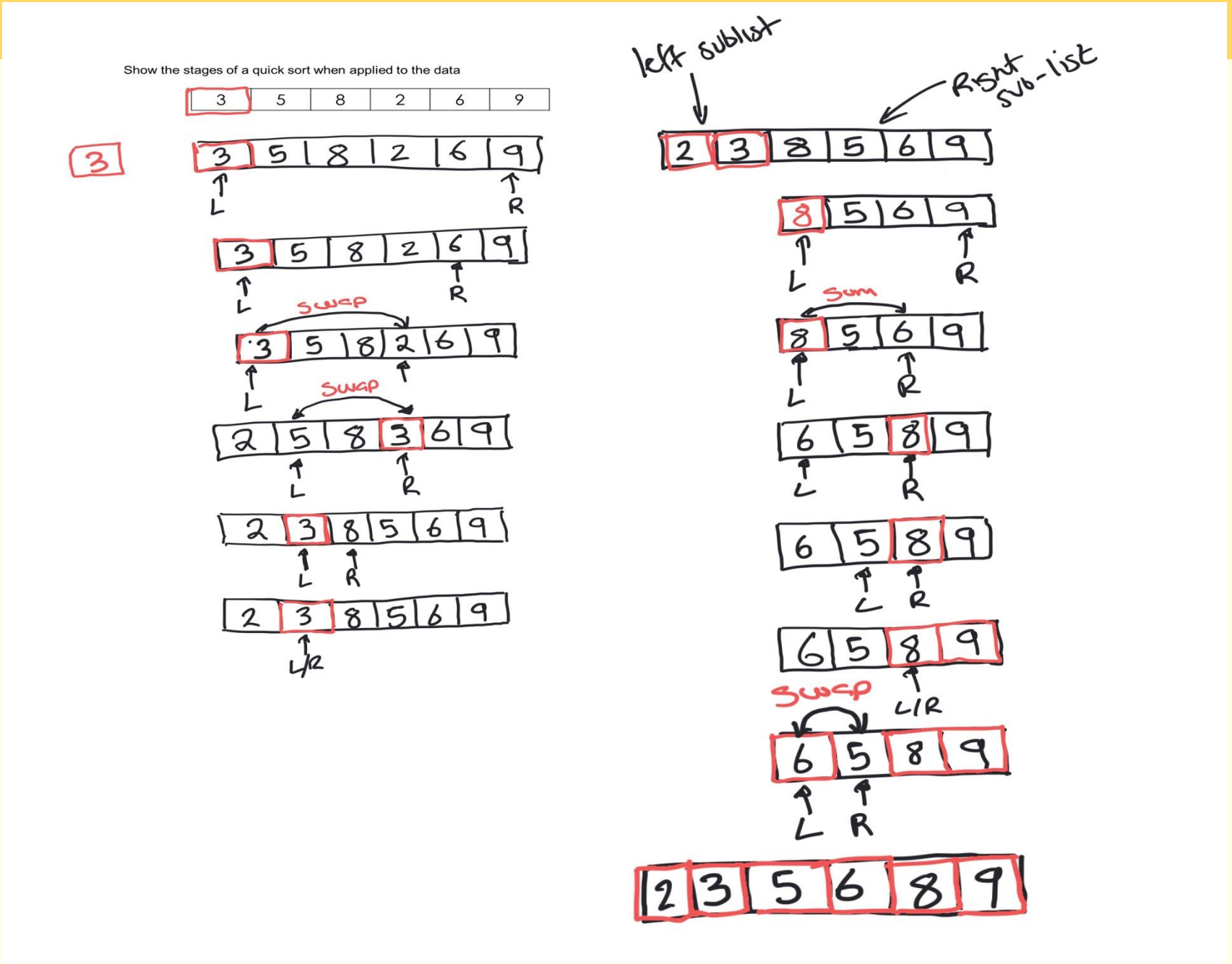
- Quick sort is a highly efficient sorting algorithm
- A **Divide and Conquer** Strategy
- Left and Right Pointers, and a pivot value
- If an item is smaller than pivot, move it to the left
- If an item is larger than pivot, move it to the right
- Recursive algorithm
- Quicksort partitions an array and then calls itself **recursively** twice to sort the two resulting subarrays.

Pivot

There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented above)
3. Pick a random element as pivot.
4. Pick median as pivot.

Example


Show the stages of a quick sort when applied to the data
3 5 8 2 6 9
3
swap
swap
left sublist
Right sub-list
Sum
swap
L
R
L/R

Quick Sort Algorithm

- Highlight first list element as start pointer, and last list element as end pointer
- Repeatedly compare numbers being pointed to…
- …if incorrect, swap and move end pointer
- …else move start pointer
- Split list into 2 sublists
- Quick sort each sublist
- Repeat until all sublists have only 1 number
- Combine sublists

Algorithm

- Choose a pivot
- Set a left pointer and right pointer
- If current pointer is the right pointer
  - If right pointer data is less than pivot
    - Swap with right pointer data with left pointer data
    - Move left pointer along by 1
    - Set left pointer as current
  - Otherwise
    - Move right pointer back by one
- If current pointer is the left pointer
  - If leftpointer data is greater than pivot
    - Swap with right pointer data with left pointer data
    - Move right pointer back by1
    - Set right pointer as current
  - Otherwise
    - Move left pointer along by 1

- Continue until leftpointer == RightPointer
- Slot the pivot data into the leftpointer

- Repeat steps on the left half and the right half of the list till the entire list is sorted.

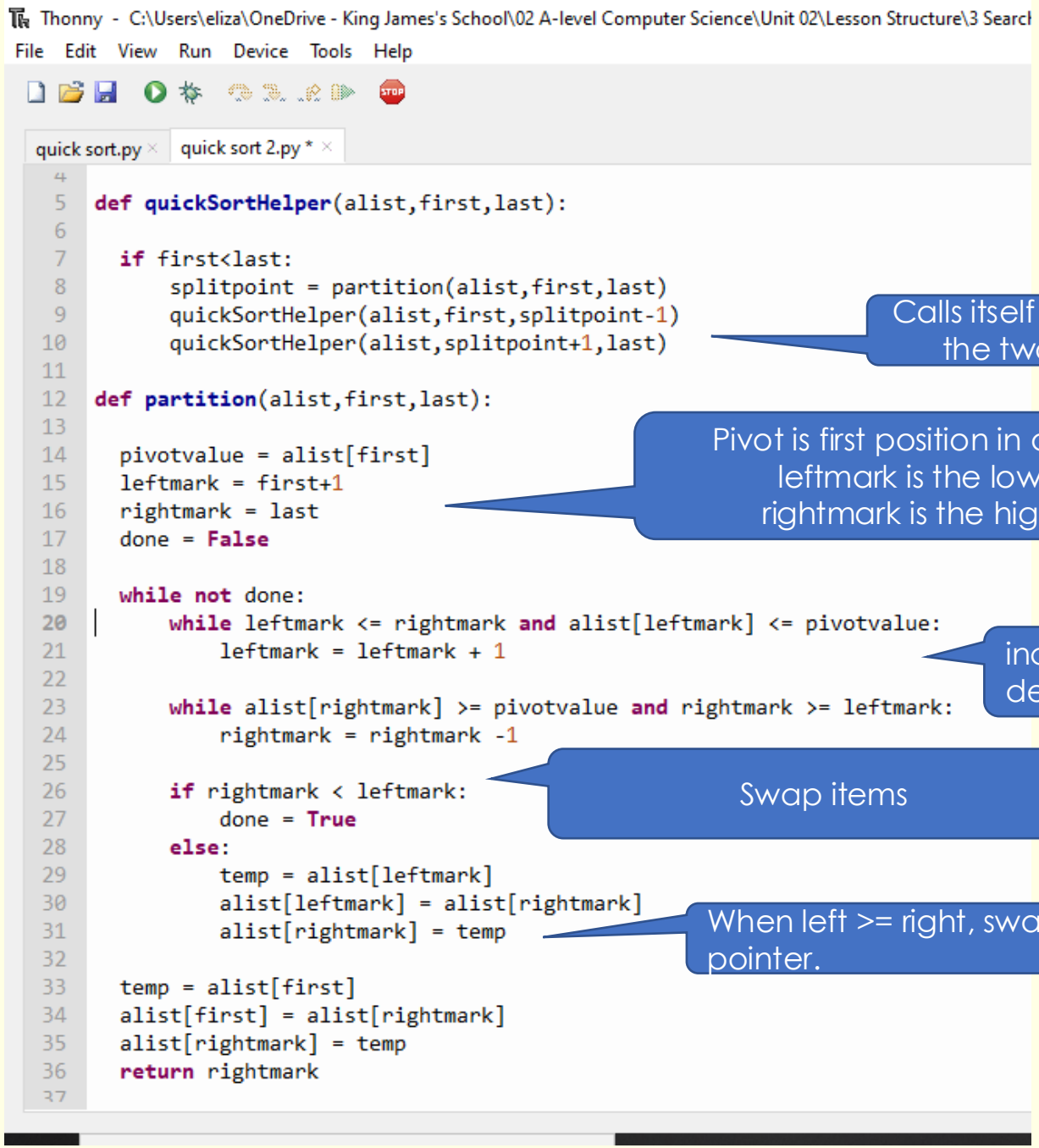
```
Pivot = Data(LeftPointer)
WHILE LeftPointer <> RightPointer
    IF CurrentPointer = "Right" THEN
        IF Data(RightPointer) < Pivot THEN
            Data(LeftPointer) = Data(RightPointer)
            LeftPointer = LeftPointer + 1
            CurrentPointer = "Left"
        ELSE
            RightPointer = RightPointer - 1
        END IF
    ELSEIF CurrentPointer = "Left" THEN
        IF Data(LeftPointer) > Pivot THEN
            Data(RightPointer) = Data(LeftPointer)
            RightPointer = RightPointer - 1
            CurrentPointer = "Right"
        ELSE
            LeftPointer = LeftPointer + 1
        END IF
    END IF
END WHILE
Data(LeftPointer) = Pivot
```

Choose a pivot
Set a left pointer and right pointer

If current pointer is the right pointer
If rightpointer data is less than pivot
Swap with left
Move left pointer along by 1
Set left pointer as current
Otherwise Move right pointer back by one

If current pointer is the left pointer
If leftpointer data is greater than pivot
Swap with right
Move right pointer back by1
Set right pointer as current
Otherwise Move left pointer along by one

Continue until leftpointer == RightPointer
Slot the pivot data into the leftpointer


Thonny - C:\Users\eliza\OneDrive - King James's School\02 A-level Computer Science\Unit 02\Lesson Structure\3 Searcl
File Edit View Run Device Tools Help
quick sort.py
quick sort 2.py *
def quickSortHelper(alist,first,last):
if first<last:
splitpoint = partition(alist,first,last)
quickSortHelper(alist,first,splitpoint-1)
quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
pivotvalue = alist[first]
leftmark = first+1
rightmark = last
done = False
while not done:
while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
leftmark = leftmark + 1
while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
rightmark = rightmark -1
if rightmark < leftmark:
done = True
else:
temp = alist[leftmark]
alist[leftmark] = alist[rightmark]
alist[rightmark] = temp
temp = alist[first]
alist[first] = alist[rightmark]
alist[rightmark] = temp
return rightmark
Calls itself recursively twice to sort the two resulting subarrays.
Pivot is first position in array
leftmark is the low
rightmark is the high
increment the left pointer and decrement the right pointer
Swap items
When left >= right, swap the pivot with either left or right pointer.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now at right place */
        pivot = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pivot
        quickSort(arr, pi + 1, high); // After pivot
    }
}
```

Quicksort Summary

+ The quick sort is regarded as the best sorting algorithm.
+ It is able to deal well with a huge list of items.
+ Because it sorts in place, no additional storage is required as well

- Inefficient in terms of memory for very large lists due to recursion (the stack can grow large with all the return addresses, variables etc that have to be stored for each recursive call).
- Sometimes it can grow too large causing a stack overflow (out of stack memory) error.
- Hard to program
- The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble If the list is already sorted than bubble sort is much more efficient than quick sort.

Summary of sort algorithms

- Bubble sort is the slowest of the sorts, with time complexity O(n2
- Insertion sort is O(n2) but if the list is already almost sorted, this reduces to O(n)
- Merge sort is O(n log n) but requires additional memory space for the merging process
- Quick sort is generally the fastest sort, but is dependent on using a pivot that is not close to the smallest or largest elements of the list. There are several methods for selecting a pivot to ensure this does not happen. It has average time complexity O(n log n). It does not require additional memory space.