MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT9133 Semester 2 2019
Programming Foundations in Python

Week 12:
Divide-and-Conquer and Recursion

Jojo Wong

Gavin Kroeger

GROUP OF EIGHT AUSTRALIA

Module 6 Synopsis

- Module 6 is aimed to introduce you with:
  - Concepts of divide-and-conquer
  - Definitions of recursion
    - Base case
    - Recursive case
    - Convergence
  - Iterative solutions versus recursive solutions
  - Recursive sorting algorithms:
    - Merge Sort
    - Quick Sort

- There are no interviews for this week.
- Please check the consultation table on Moodle
  - Some of our booked consultation rooms were taken by other units during swotvac.
- SETU is open, please fill it in.
  - You can win gift cards by submitting the SETU.

- Upon completing this module, you should be able to:
  – Recognise computational problems that are solvable using the divide-and-conquer or recursive approach
  – Implement a recursive solution for a specific problem
  – Contrast the two advanced sorting algorithms that adopt the recursive approach (Merge Sort and Quick Sort)

Concepts of Divide-and-Conquer

- Divide-and-Conquer:
  - Solving a complex problem by breaking it into smaller manageable sub-problems
  - Sub-problems can then be solved in a similar way (with the same solution)
  - Sub-solutions are then combined to produce the final solution for the original problem

- Basic example: Binary Search
  - "Repeatedly" divides the (sorted) list into two sublists until the target item is found

Binary Search: "Iterative" Approach

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

    # repeatedly divide the list into two halves
    # 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
```

Binary Search: “Recursive” Approach

```
def rec_binary_search(the_list, target_item):
    # repeatedly divide the list into two halves
    # as long as the target item is not found

    # the list cannot be further divided i.e. item is not found
    if len(the_list) == 0:
        return False
    else:
        # find the mid position
        mid = len(the_list) // 2

        # check if target item is equal to middle item
        if the_list[mid] == target_item:
            return True
        # check if target item is less than middle item
        # search lower half
        elif target_item < the_list[mid]:
            smaller_list = the_list[:mid]
            return rec_binary_search(smaller_list, target_item)
        # check if target item is greater than middle item
        # search upper half
        else:
            smaller_list = the_list[mid+1:]
            return rec_binary_search(smaller_list, target_item)
```

Recursive Function:
A function that calls itself repeatedly.

Concepts of Recursion

- Recursion:
  - A divide-and-conquer approach for solving computational problems
  - Each problem is “recursively” decomposed into sub-problems (which have the same properties the original problem but smaller in size)
  - When the sub-problems have reached the *simplest* form, i.e. a known solution can be defined
  - The known solutions of these sub-problems are then recomposed together to produce the solution of the original problem

Cannot be further decomposed

Solution is known

- Three key requirements:
  - Base case: The recursive function must have a base case (i.e. the simplest form)
  - Convergence: The recursive function must be able to decompose the original problem into sub-problems; and must be converging towards the base case
  - Recursive case: The recursive function must call itself recursively to solve the sub-problems

Example: Recursive Addition

- How could we solve an *addition* problem recursively?

```
5 + 1 =
(4 + 1) + 1 =
((3 + 1) + 1) + 1 =
(((2 + 1) + 1) + 1)+ 1 =
((((1 + 1) + 1) + 1) + 1) + 1 = 6
```

```
def recursive_addition(a, b):
    if a == 0:
        return b
    return recursive_addition(a-1, b) + b
```

Partial solution

```
def recursive_addition(a, b):
    if a == 0:
        return b
    return recursive_addition(a-1, b+1)
```

Complete solution

Iterative Solution vs Recursive Solution

- How could we solve a factorial problem (`n!`) *iteratively*?
- E.g. `5! = 5 * 4 * 3 * 2 * 1`

```
def iterative_factorial(n):

    factorial = 1

    while n > 0:
        factorial *= n
        n -= 1

    return factorial
```

Iterative Solution vs Recursive Solution

- How could we solve a factorial problem (`n!`) *recursively*?
- E.g. `5! = 5 * 4!`

```
5! = 5 * (4 * 3!)
5! = 5 * (4 * (3 * 2!))
5! = 5 * (4 * (3 * (2 * 1!)))
5! = 5 * (4 * (3 * (2 * (1))))
```

```python
def recursive_factorial(n):

    if n == 1:
        return 1
    return n * recursive_factorial(n-1)
```

convergence

base case

recursive case

Review Questions:
Part 1

What does the given recursive function do? (Assume that a and b are positive integers.)

```
def mystery_func1(a, b):
    if a == 0:
        return 0
    return b + mystery_func1(a-1, b)
```

A. Addition
B. Subtraction
C. Multiplication
D. Division
E. None of the above

What does the given recursive function do? (Assume that n is an positive integer.)

```
def mystery_func2(n):
    if n == 1:
        return 1
    return n + mystery_func2(n-1)
```

A. Adding up from 1 to n
B. Subtracting 1 from n
C. Multiplying from 1 to n
D. Dividing n with 1
E. None of the above

What is the result of the given recursive function? (Assume a = 2 and b = 3.)

```
def mystery_func3(a, b):
    if b == 0:
        return 1
    if b % 2 == 0:
        return mystery_func3(a*a, b//2)
    return mystery_func3(a*a, b//2) * a
```

A. 5
B. 6
C. 8
D. 16
E. None of the above

What is the result of the given recursive function? (Assume a = “1234”.)

```
def mystery_func4(a):
    if len(a) == 1:
        return a
    return mystery_func4(a[1:]) + a[0]
```

A. “1234”
B. “4321”
C. “4”
D. “1”
E. None of the above

How many recursive calls and base cases are there for the given function?

```
def mystery_func5(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return mystery_func5(n-1) + mystery_func5(n-2)
```

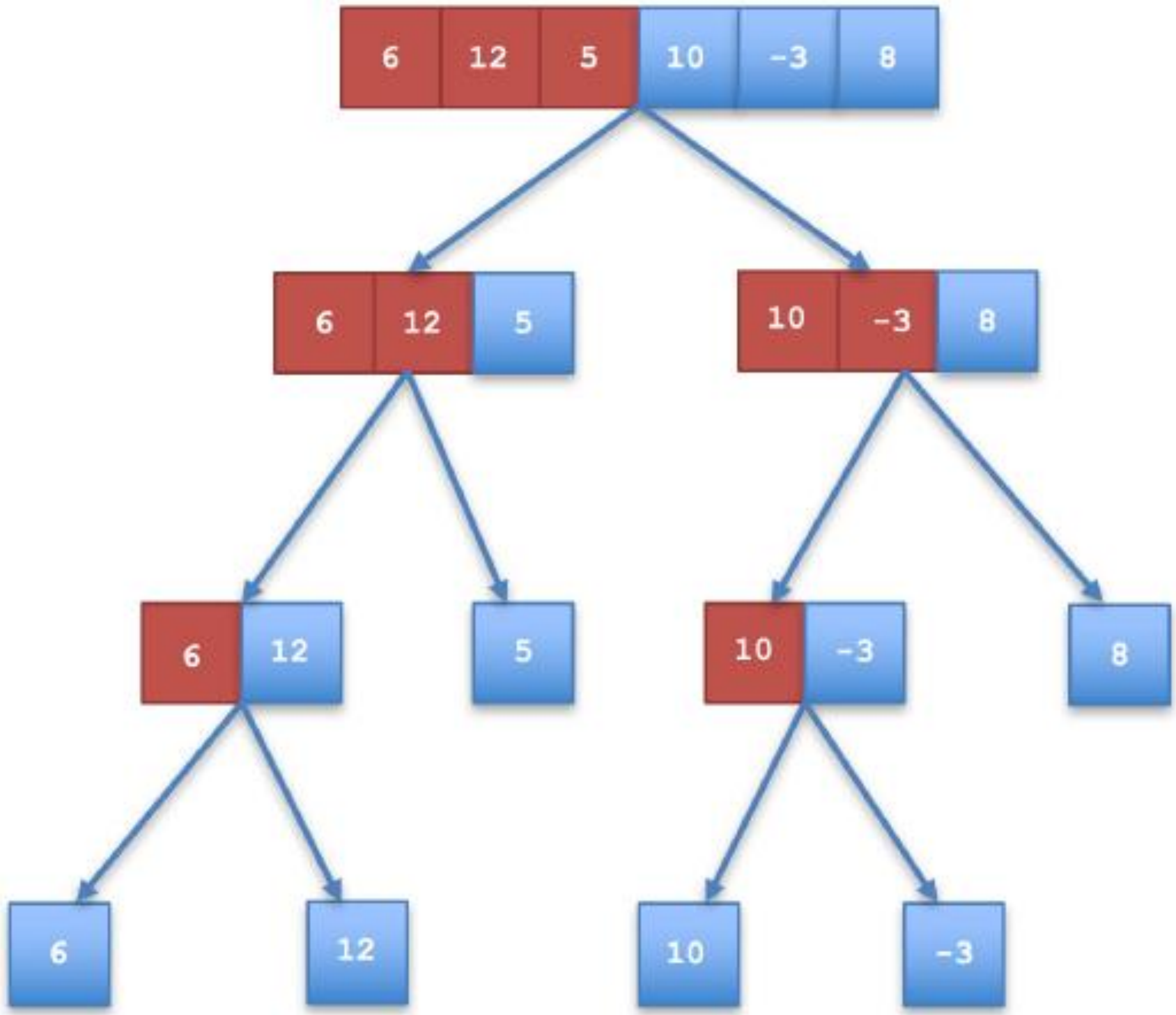
A. 1 and 1
B. 1 and 2
C. 2 and 1
D. 2 and 2
E. None of the above

Recursive Sorting:
Merge Sort

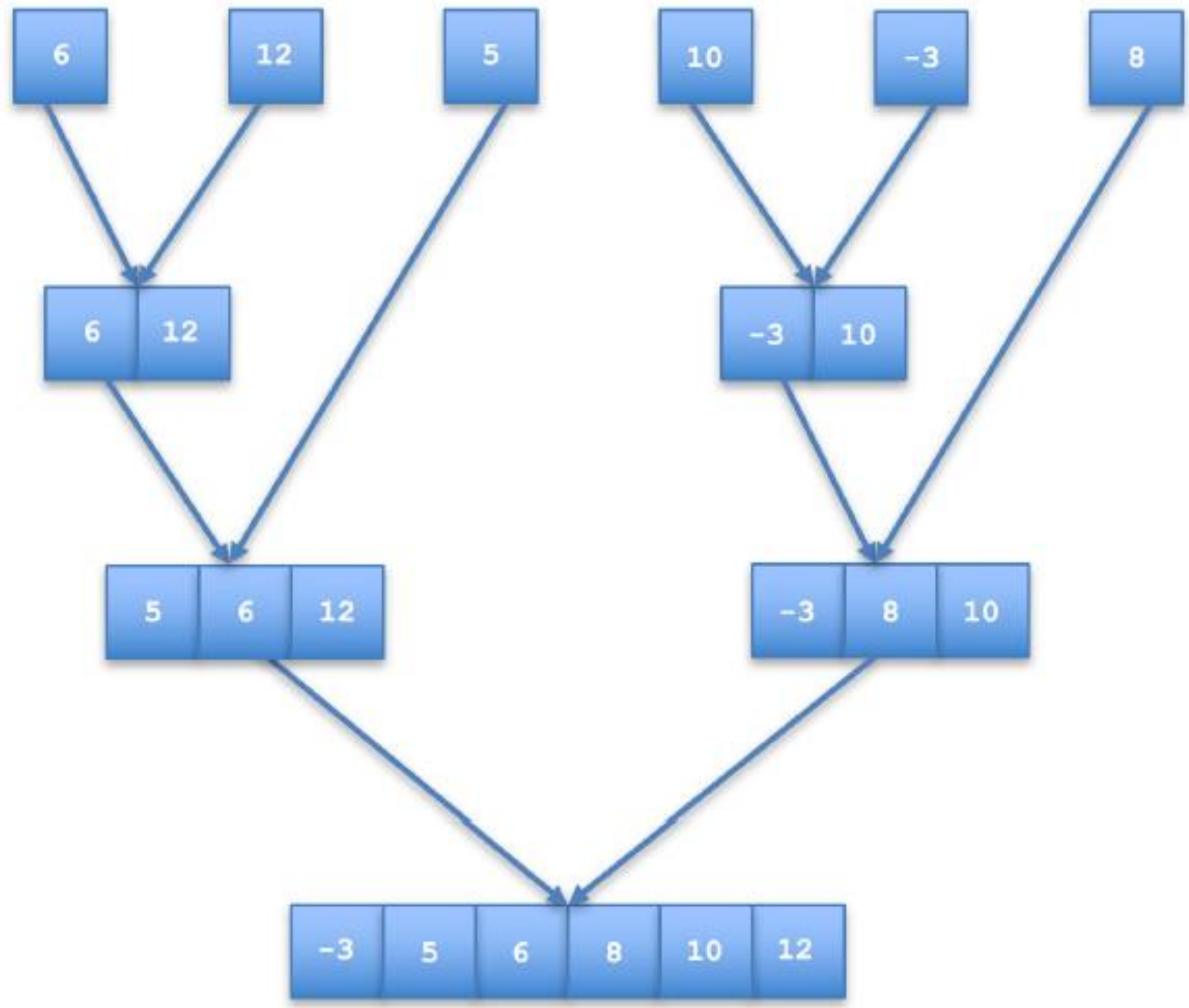
- Basic ideas:
  - Splits an unsorted list into two halves “recursively” until there is only one element left in each sublist
  - Sublists are then sorted and merged until the complete sorted list is obtained

- Divide and conquer:
  - Base case: A sublist with length of one (considered *sorted*)
  - Divide: Recursively identify the middle point of a list and divide into two halves
  - Conquer: Sort the smaller sublists
  - Combine: Merge the sorted sublist into one complete list

Merge Sort: Splitting



Merge Sort: Merging



Merge Sort: Implementation

```python
def merge_sort(the_list):
    # obtain the length of the list
    n = len(the_list)

    if n > 1: # check for base case
        # find the middle of the list
        mid = n // 2

        # based upon the middle index, two sublists are then created
        # from the 0 index until the mid-1 index
        left_sublist = the_list[:mid]
        # from the mid index until the n - 1 index
        right_sublist = the_list[mid:]
        print("Splitting: " + str(left_sublist) + " and " + str(right_sublist))

        # merge sort is called again on the two new created sublists
        merge_sort(left_sublist)
        merge_sort(right_sublist)
```

Merge Sort: Implementation (continue)

```
# sort and merge
print("Merging " + str(left_sublist) + " with " + str(right_sublist))
i = 0 # index for left sublist
j = 0 # index for right sublist
k = 0 # index for main list


while i < len(left_sublist) and j < len(right_sublist):
    if left_sublist[i] <= right_sublist[j]:
        the_list[k] = left_sublist[i]
        i += 1
    else:
        the_list[k] = right_sublist[j]
        j += 1
    k += 1

# insert the remaining elements into main list
while i < len(left_sublist):
    the_list[k] = left_sublist[i]
    i += 1
    k += 1

while j < len(right_sublist):
    the_list[k] = right_sublist[j]
    j += 1
    k += 1

print("After merging the list is: " + str(the_list))
```

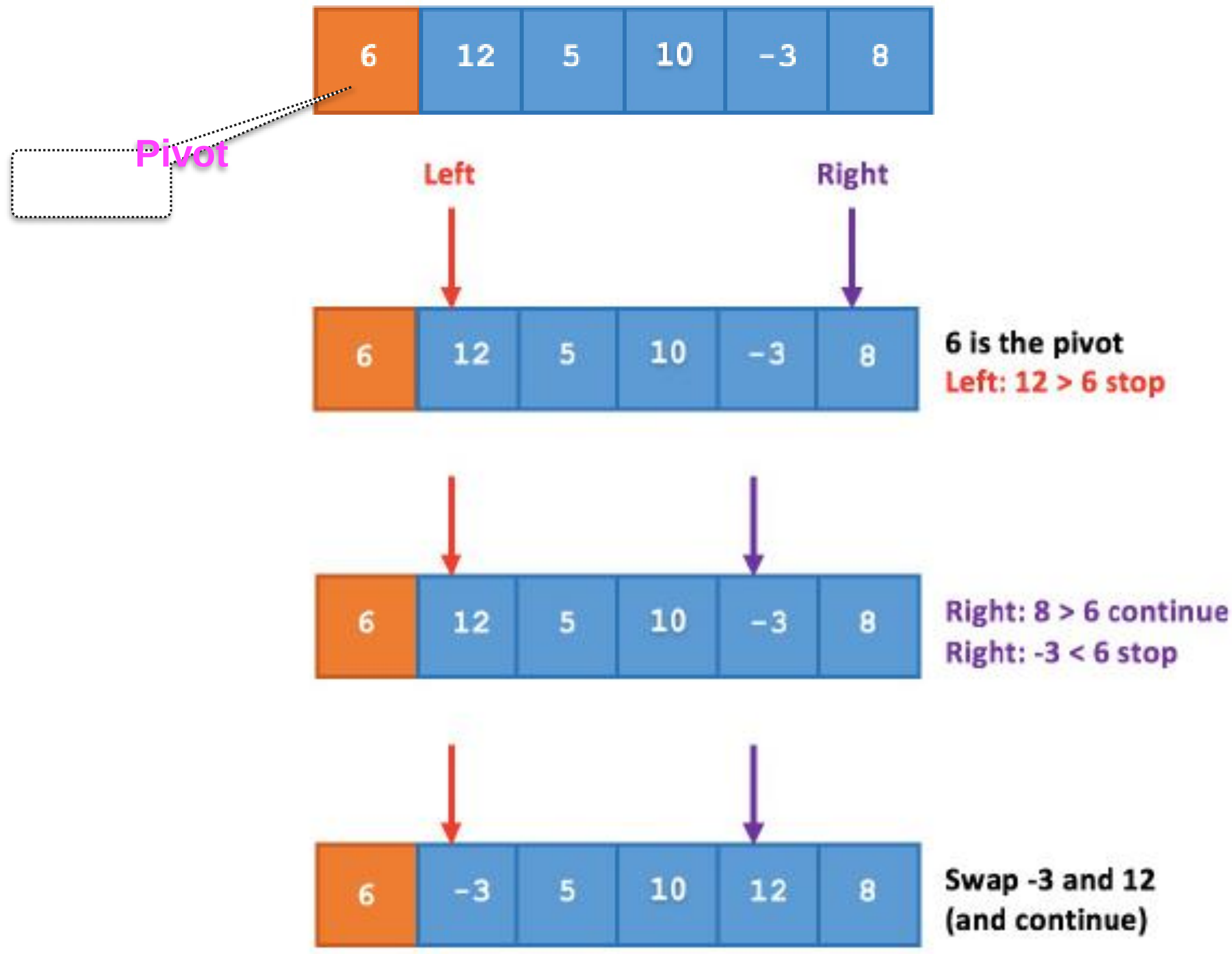
Recursive Sorting:
Quick Sort

- Basic ideas:
  - Similarly to Merge Sort
  - Major computation is performed in "partitioning" (dividing the list into two partitions)

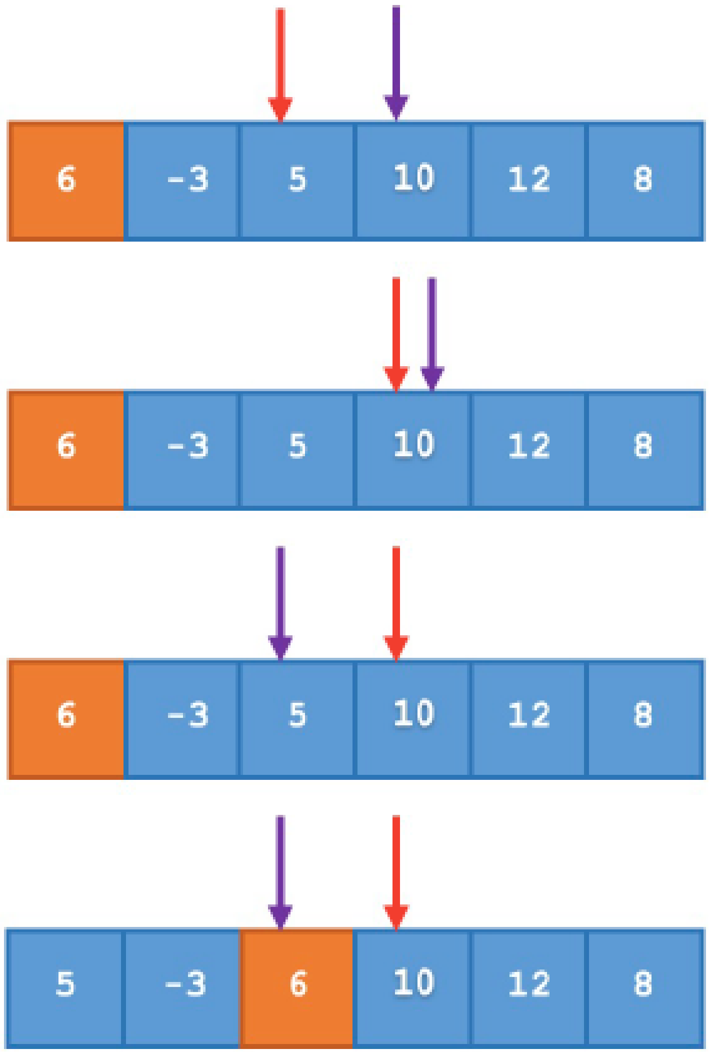
first element as "pivot"

- Divide and conquer:
  - Divide: Select a "pivot" to serve as the partition point
    - Elements smaller than the pivot are relocated to the left of the pivot
    - Elements greater than the pivot are relocated to the right
  - Conquer: Recursively partition the sublists based on the pivot chosen for each sublist
  - Combine: No computation needed
  - Base case: A sublist with length of one (considered *sorted*) or with zero length

Quick Sort: Partitioning



Quick Sort: Partitioning (continue)



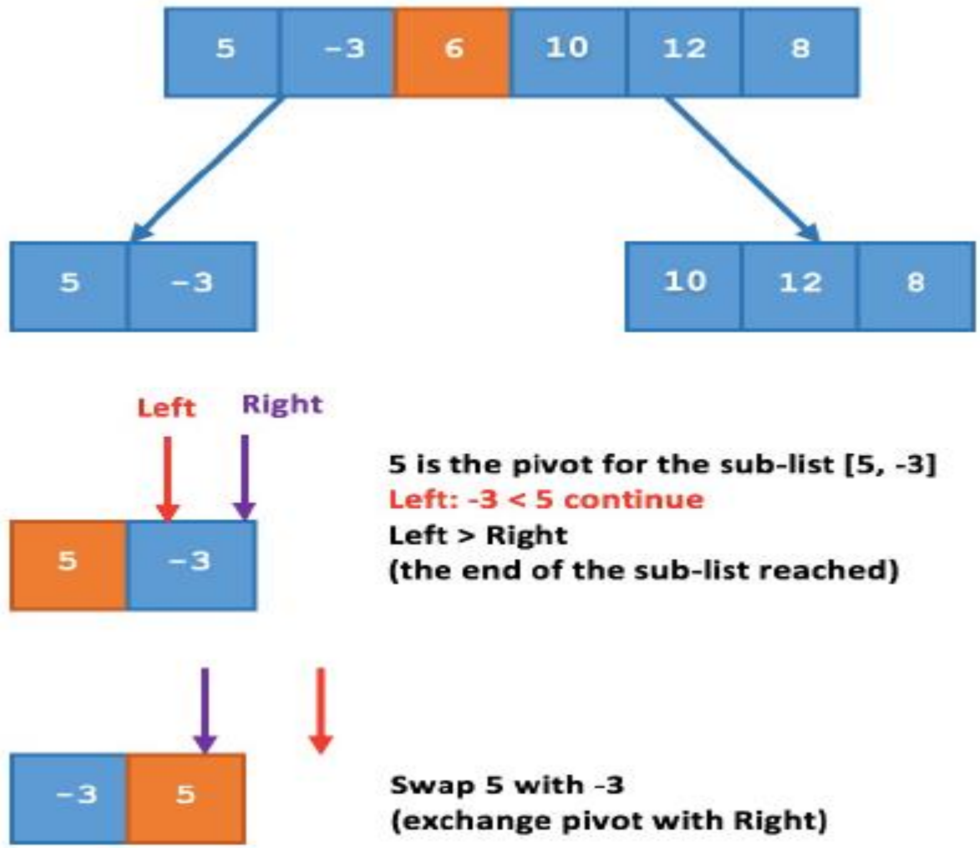
Left: 5 < 6 continue
Left: 10 > 6 stop

Right: 10 > 6 continue
Right: 5 < 6 stop

Right < Left
(partition point found)

Swap 6 with 5
(swap pivot with Right)

Quick Sort: Partitioning (continue)



At this stage, another two sub-lists will be created: [-3] and [ ]. Since these are the base case, the partitioning process terminates here. Note that the original list is considered partially sorted at this point. The other sub-list [10, 12, 8] will be sorted using the same partitioning procedure until the original list is completely sorted.

Quick Sort: Implementation

```
def quick_sort(the_list):
    # pass the indices of first and last elements of the list
    first = 0
    last  = len(the_list) - 1
    quick_sort_aux(the_list, first, last)

def quick_sort_aux(the_list, first, last):
    # if it is not the base case
    if first < last:
        # find the partition point
        partition_point = partitioning(the_list, first, last)

        print("partition at index: ", partition_point)
        print(the_list)
        print ("after partitioning: " + str(the_list[first:partition_point]) +
               " and " + str(the_list[partition_point+1:last+1]))

        # call the quick sort function again on the new sublists
        quick_sort_aux(the_list, first, partition_point - 1)
        quick_sort_aux(the_list, partition_point + 1, last)
```

Quick Sort: Implementation (continue)

```
def partitioning(the_list, first, last):
    # take first element of the list is the pivot
    pivot_value = the_list[first]
    print("pivot: ", pivot_value)

    # these two indices will help us in locating the index point
    # where the list will be partitioned
    left_index = first + 1
    right_index = last

    complete = False

    while not complete:
        # start with the left index and keep on incrementing it
        # until a value greater than the pivot value is found
        while left_index <= right_index and
              the_list[left_index] <= pivot_value:
            left_index += 1

        # now look for element from the right of the list
        # which is smaller than the pivot value
        while right_index >= left_index and
              the_list[right_index] >= pivot_value:
            right_index -= 1

        # check whether left and right indices have crossed each other
        # if that is the case exit the while loop
        if right_index < left_index:
            complete = True
        else:
        # otherwise swap the two elements
            the_list[left_index], the_list[right_index]
                 = the_list[right_index], the_list[left_index]

    # swap the pivot element with the element of the right index
    the_list[first], the_list[right_index]
         = the_list[right_index], the_list[first]

    # return right index which is the partition point
    return right_index
```

MONASH University

Review Questions:
Part 2

Given a list of integers below after the first partitioning of running Quick Sort, which of the integers could likely to be the *pivot*?

```
2, 5, 3, 7, 10, 12, 9
```

A. 7
B. 10
C. Either 7 or 10
D. Neither 7 nor 10

Given a list of integers below to be sorted using Quick Sort. How would the two partitions be if ‘4’ is chosen as the *pivot*?

```
4, 5, 8, 10, 12, 9, 11
```

A. Two almost even partitions
B. Left partition with 0 elements
C. Right partition with 0 elements
D. Not sure

Week 12 Summary

- We have discussed:
  - Concepts of recursion
  - Iterative solutions vs recursive solutions
  - Recursive sorting algorithms (MergeSort, QuickSort)