CS 171 - Lab 9

Professor Mark W. Boady and Professor Adelaida Medlock

Content by Mark Boady, Drexel University

Detailed instructions to the lab assignment are found in the following pages.

- Complete all the exercises and type your answers in the space provided.

What to submit:

- Lab sheet in PDF format.
- A screenshot with your code for question 8.

Submission must be done via Gradescope

- Please make sure you have tagged all questions and added your teammates if any
- We only accept submissions via Gradescope.

Student Name: **Tony Kabilan Okeke**

User ID (abc123): **tko35**

Possible Points: 103

Your score out of 103:

Lab Grade on 100% scale:

Graded By (TA Signature):

Question 1: 9 points

In this lab, we will look at two different methods of sorting.

First, we will examine how the algorithms work. In the final question, you will code each of the algorithms.

(a) (1 point) Sort the following list: `[17, 12, 10, 15, 0, 14, 13, 5, 0, 6]`
[0, 0, 5, 6, 10, 12, 13, 14, 15, 17]

(b) (1 point) Sort the following list: `[11, 0, 10, 12, 3, 17, 11, 15, 7, 8]`
[0, 3, 7, 8, 10, 11, 11, 12, 15, 17]

(c) (1 point) Sort the following list: `[10, 19, 17, 0, 16, 3, 20, 14, 9, 1]`
[0, 1, 3, 9, 10, 14, 16, 17, 19, 20]

(d) (1 point) Sort the following list: `[7, 16, 20, 0, 20, 10, 11, 18, 0, 1]`
[0, 0, 1, 7, 10, 11, 16, 18, 20, 20]

(e) (5 points) Think about how your group sorted the lists. Write out instructions for **How to Sort a List**.

- Create the empty output list
- Look for the smallest number in the list and move it to the first position in the sorted (output) list and remove it from the original list.
- Look for the next smallest number in the list, append it to the output list and remove it from the original list.
- Repeat this until all the numbers in the original list have been removed.

The first **Algorithm** we will look at is based on **merging**.

If we have two lists that are already sorted, it should be easy to **merge** them into one new list. Let's say we start with

```
A = [1, 3, 5]
B = [2, 6, 9]
```

We can **merge** these arrays by looking at the first elements and picking the smallest.

```
function Merge (A, B)
   C = []
   while len(A) > 0 and len(B) > 0 do
       if A[0] < B[0] then
          Add A[0] to end of C
          Remove A[0] from A
       else
          Add B[0] to end of C
          Remove B[0] from B
       end if
   end while
   while len(A) > 0 do
       Move Remaining A values to C
   end while
   while len(B) > 0 do
       Move Remaining B values to C
   end while
   return C
end function
```

The steps taken to merge the two lists are shown in the table below.

A	B	Comparison	C
[1,···]	[2,···]	1 < 2	[1]
[3,···]	[2,···]	3 < 2	[1, 2]
[3,···]	[6,···]	3 < 6	[1, 2, 3]
[5,···]	[6,···]	5 < 6	[1, 2, 3, 5]
[]	[6,···]	*len*(*B*) > 0	[1, 2, 3, 5, 6]
[]	[9,···]	*len*(*B*) > 0	[1, 2,3, 5, 6, 9]
[]	[]	Return	[1, 2, 3, 5,6, 9]

Question 2: 19 points

(a) (3 points) Is it possible for both the following **while** statements to run, or will at most one run during a specific call to the function? Explain your answer.

```
while len(A) > 0 do
    Move Remaining A values to C
end while
while len(B) > 0 do
    Move Remaining B values to C
end while
```

No, there is no case where both of these while loops would run.
The previous while loop runs as long as both lists are not empty. Either one or both of the lists (A and B) must be empty to exit the first while loop. As such, only one of the two loops above will be able to run in a single function call since one of the list must be empty before either of those loops are executed.

(b) (14 points) Replicate the Table from the previous page to show how merge works with these new inputs.

```
A = [5, 7, 9, 11]
B = [1, 7, 8]
```

A	B	Comparison	C
[5, ···]	[1, ···]	5 < 1	[1]
[5, ···]	[7, ···]	5 < 7	[1, 5]
[7, ···]	[7, ···]	7 < 7	[1, 5, 7]
[7, ···]	[8, ···]	7 < 8	[1, 5, 7, 7]
[9, ···]	[]	len(A) > 0	[1, 5, 7, 7, 9]
[11, ···]	[]	len(A) > 0	[1, 5, 7, 7, 9, 11]
[]	[]	return	[1, 5, 7, 7, 9, 11]

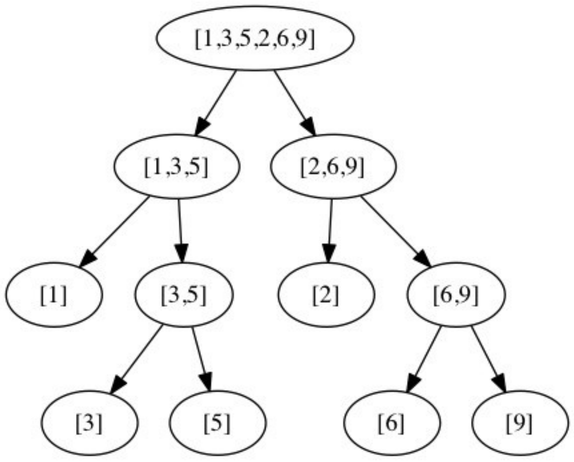
(c) (2 points) When you encounter an element that is the same in both lists, which list is the element removed from first?
The element is removed from the second list (B) first.

Question 3: 15 points

The **Merge** function can be used to sort. We take a list and divide it into smaller lists. Then we merge them together.

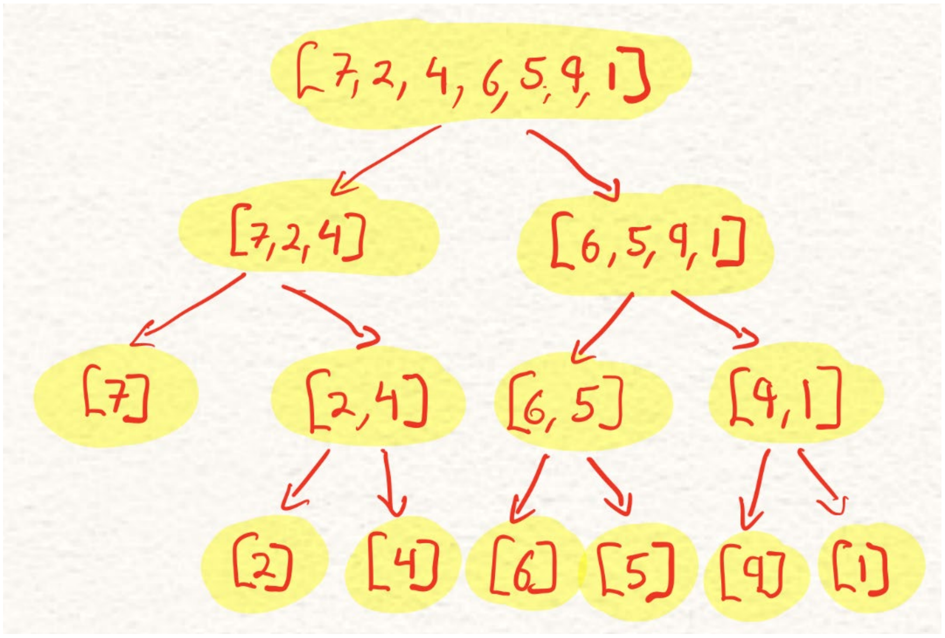
The **MergeSort** function takes a list. The list is divided in half repeatedly. When the list has been divided into single elements, it is merged back together.

The below tree shows how we divide up the array `[1,3,5,2,6,9]`



(a) (2 points) If the length of a list if **odd** does the extra element go on the **left** or **right**?
Extra elements go to the right if the list has an odd length.

(b) (13 points) Draw a tree diagram, like the one above, that shows how `[7,2,4,6,5,9,1]` is divided.



Question 4: 10 points

We can put all the pieces together to make the **Merge Sort** Algorithm.

```
function msort(X)
   if len(X)  > 1 then
      middle = len(X)  // 2
      A = msort(X[ : middle])
      B = msort(X[middle : ])
      C= merge(A, B)
      return C
   else
      return X
   end if
end function
```

Below is a trace of how this function sorts a list `[2, 4, 1, 3, 6]`

```
Calling msort ([2 ,4, 1, 3, 6])
Calling msort ([2, 4])
Calling msort ([2])
Calling msort ([4] )
Merge [2] and [4]
Calling msort ([1, 3, 6])
Calling msort ([1])
Calling msort ([3, 6])
Calling msort ([3])
Calling msort ([6])
Merge [3] and [6]
Merge [1] and [3, 6]
Merge [2, 4] and [1 ,3, 6]
```

(a) (10 points) Write out what the trace should look like out if we asked it to sort the list `[9, 5, 7, 2]`. This should look similar to the above example.

```
Calling msort ([9, 5, 7, 2])
Calling msort ([9, 5])
Calling msort ([9])
Calling msort ([5])
Merge [9] and [5]
Calling msort ([7, 2])
Calling msort ([7])
Calling msort ([2])
Merge [7] and [2]
Merge [9, 5] and [7, 2]
```

Question 5: 20 points

A second method to sort uses **Partitions.** Instead of dividing the list in half at the middle, we divide it into *smaller* and *larger* piles.

If a value is equal to the pivot, put in in the *larger* pile.

(a) (4 points) Partition the list `[68, 19, 86, 8, 64, 79, 77, 78, 18, 23]` on the pivot value `23`

Smaller Values: [19, 8, 18]

Pivot: 23

Larger Values: [68, 86, 64, 79, 77, 78, 23]

(b) (4 points) Partition the list `[74, 62, 99, 63, 52, 74, 27, 78, 32, 6]` on the pivot value `6`

Smaller Values: []

Pivot: 6

Larger Values: [74, 62, 99, 63, 52, 74, 27, 78, 32, 6]

(c) (4 points) Partition the list `[91, 8, 21, 86, 25, 73, 60, 49, 13, 46]` on the pivot value `46`

Smaller Values: [8, 21, 25, 13]

Pivot: 46

Larger Values: [91, 86, 73, 60, 49, 46]

(d) (4 points) Partition the list `[53, 36, 60, 23, 51, 39, 68, 71, 92, 41]` on the pivot value `41`

Smaller Values: [36, 23, 39]

Pivot: 41

Larger Values: [53, 60, 51, 68, 71, 92, 41]

(e) (4 points) Partition the list `[39, 9, 76, 30, 64, 26, 77, 13, 58, 38]` on the pivot value `38`

Smaller Values: [9, 30, 26, 13]

Pivot: 38

Larger Values: [39, 76, 64, 77, 58, 38]

Question 6: 6 points

Quicksort Partitions a List repeatedly until it is sorted. The algorithm is described below.

```
function qsort(A, start, stop)
    if start < stop then
        p = partition (A, start, stop)
        qsort(A, start, p - 1)
        qsort(A, p + 1, stop)
    end if
end function
```

Answer the following questions about this algorithm.

(a) (2 points) Does this function have a return value? Why or Why not?
The function doesn't have a return value because it is modifying the list A in place (changing the original object instead of making a copy). This is possible since lists are mutable objects.

(b) (2 points) Why don't either of the recursive calls actually include the value at position p?
After each call to the partition function, the pivot value is located at position p in the list, and all values before it are smaller, while all values after it are larger. So, the value at position p does not need to be included in subsequent calls to qsort since it is already in its sorted position.

(c) (2 points) Why don't we need to do anything when `start >= stop`?
It start >= stop, then the list will already be completely sorted, since p will either be equal to start or equal to stop.

Question 7: 4 points

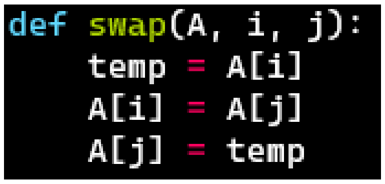
The final piece we need to sort is an algorithm to partition.

```
function partition (A, start, stop)
    Set pivot = A[stop]
    Set i = start
    for j in range(start, stop) do
        if A[j] ≤ pivot then
            Swap A[i] and A[j]
            i++
        end if
    end for
    Swap A[i] and A[stop]
    return i
end function
```

(a) (2 points) What value is selected as the **pivot** element in this algorithm?
The last value in a partition

(b) (2 points) How would you implement the pseudocode `Swap A[i] and A[j]`?

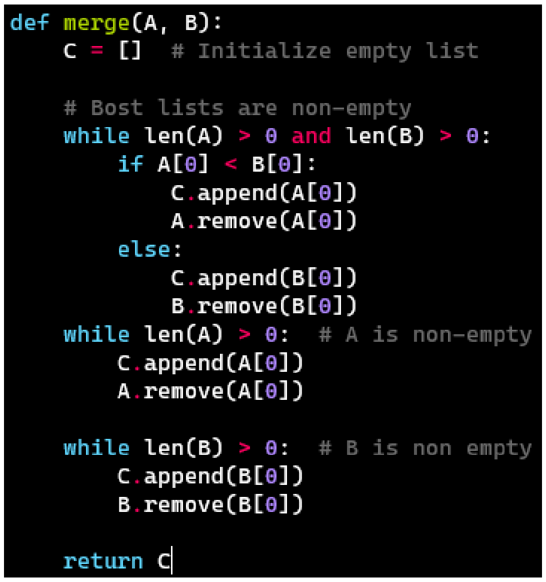
- Store the value of A[i] in a temporary variable
- Set A[i] equal to A[j]
- Set A[j] equal to the temporary variable



```
def swap(A, i, j):
    temp = A[i]
    A[i] = A[j]
    A[j] = temp
```

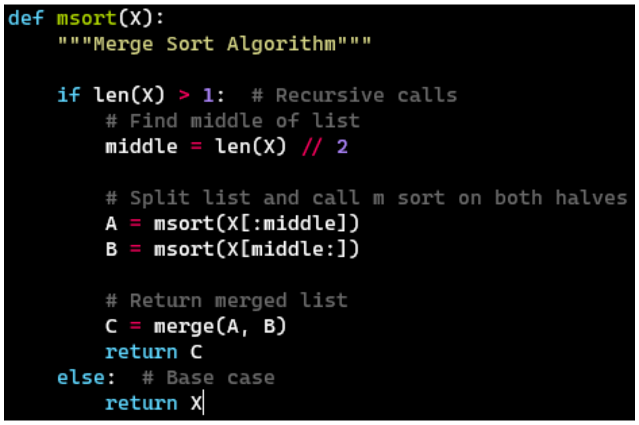
Question 8: 20 points

(a) (5 points) Implement the **Merge** function in Python.



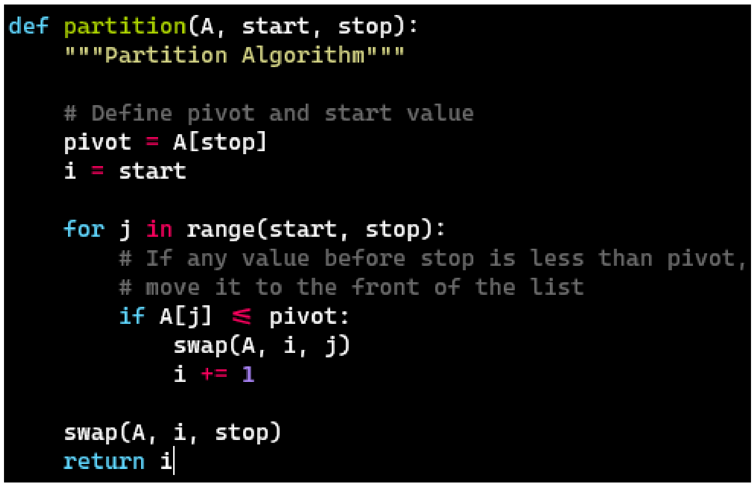
```python
def merge(A, B):
    C = []  # Initialize empty list

    # Bost lists are non-empty
    while len(A) > 0 and len(B) > 0:
        if A[0] < B[0]:
            C.append(A[0])
            A.remove(A[0])
        else:
            C.append(B[0])
            B.remove(B[0])
    while len(A) > 0:  # A is non-empty
        C.append(A[0])
        A.remove(A[0])

    while len(B) > 0:  # B is non empty
        C.append(B[0])
        B.remove(B[0])

    return C
```

(b) (5 points) Implement the **MSort** function in Python.



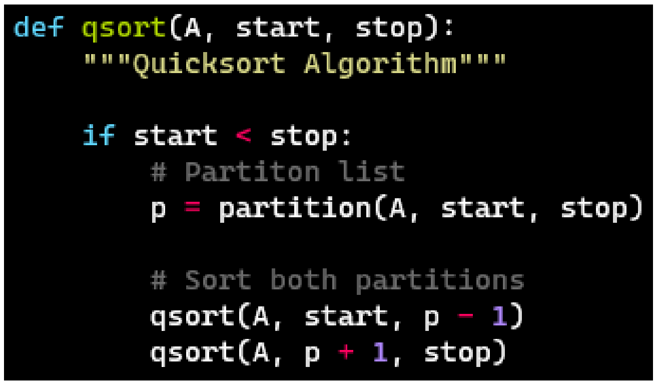
```python
def msort(X):
    """Merge Sort Algorithm"""

    if len(X) > 1:  # Recursive calls
        # Find middle of list
        middle = len(X) // 2

        # Split list and call m sort on both halves
        A = msort(X[:middle])
        B = msort(X[middle:])

        # Return merged list
        C = merge(A, B)
        return C
    else:  # Base case
        return X
```

(c) (5 points) Implement the **Partition** function in Python.



```python
def partition(A, start, stop):
    """Partition Algorithm"""

    # Define pivot and start value
    pivot = A[stop]
    i = start

    for j in range(start, stop):
        # If any value before stop is less than pivot,
        # move it to the front of the list
        if A[j] <= pivot:
            swap(A, i, j)
            i += 1

    swap(A, i, stop)
    return i
```

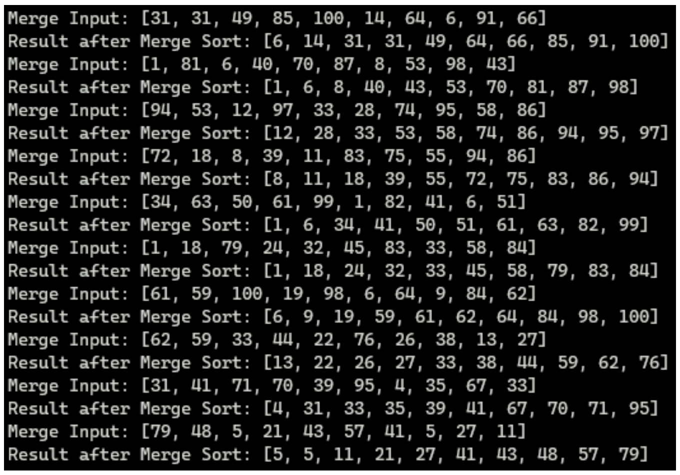
(d) (5 points) Implement the **Qsort** function in Python.



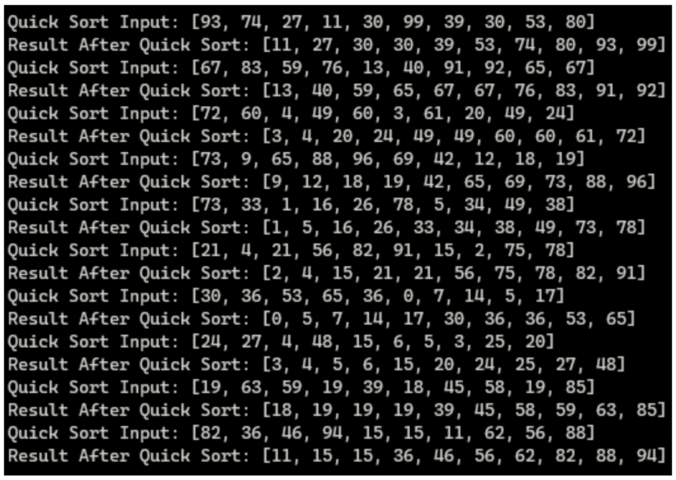
```
def qsort(A, start, stop):
    """Quicksort Algorithm"""

    if start < stop:
        # Partiton list
        p = partition(A, start, stop)

        # Sort both partitions
        qsort(A, start, p - 1)
        qsort(A, p + 1, stop)
```

Test your functions with the below code. Submit screenshots and output for each question. You may submit multiple screenshots if needed to show all code or output.

```
#Testing Sorts
import random
for x in range (0, 10):
     L = [random.randint (0, 100) for k in range(0, 10) ]
     print("Merge Input:", L)
     L = msort(L)
     print("Result after Merge Sort:", L)

for x in range(0, 10) :
     L = [random.randint(0, 100) for k in range(0,10)   ]
     print("Quick Sort Input:", L)
     qsort (L, 0, len(L) - 1)
     print("Result After Quick Sort:", L)
```



```
Merge Input: [31, 31, 49, 85, 100, 14, 64, 6, 91, 66]
Result after Merge Sort: [6, 14, 31, 31, 49, 64, 66, 85, 91, 100]
Merge Input: [1, 81, 6, 40, 70, 87, 8, 53, 98, 43]
Result after Merge Sort: [1, 6, 8, 40, 43, 53, 70, 81, 87, 98]
Merge Input: [94, 53, 12, 97, 33, 28, 74, 95, 58, 86]
Result after Merge Sort: [12, 28, 33, 53, 58, 74, 86, 94, 95, 97]
Merge Input: [72, 18, 8, 39, 11, 83, 75, 55, 94, 86]
Result after Merge Sort: [8, 11, 18, 39, 55, 72, 75, 83, 86, 94]
Merge Input: [34, 63, 50, 61, 99, 1, 82, 41, 6, 51]
Result after Merge Sort: [1, 6, 34, 41, 50, 51, 61, 63, 82, 99]
Merge Input: [1, 18, 79, 24, 32, 45, 83, 33, 58, 84]
Result after Merge Sort: [1, 18, 24, 32, 33, 45, 58, 79, 83, 84]
Merge Input: [61, 59, 100, 19, 98, 6, 64, 9, 84, 62]
Result after Merge Sort: [6, 9, 19, 59, 61, 62, 64, 84, 98, 100]
Merge Input: [62, 59, 33, 44, 22, 76, 26, 38, 13, 27]
Result after Merge Sort: [13, 22, 26, 27, 33, 38, 44, 59, 62, 76]
Merge Input: [31, 41, 71, 70, 39, 95, 4, 35, 67, 33]
Result after Merge Sort: [4, 31, 33, 35, 39, 41, 67, 70, 71, 95]
Merge Input: [79, 48, 5, 21, 43, 57, 41, 5, 27, 11]
Result after Merge Sort: [5, 5, 11, 21, 27, 41, 43, 48, 57, 79]
```



```
Quick Sort Input: [93, 74, 27, 11, 30, 99, 39, 30, 53, 80]
Result After Quick Sort: [11, 27, 30, 30, 39, 53, 74, 80, 93, 99]
Quick Sort Input: [67, 83, 59, 76, 13, 40, 91, 92, 65, 67]
Result After Quick Sort: [13, 40, 59, 65, 67, 67, 76, 83, 91, 92]
Quick Sort Input: [72, 60, 4, 49, 60, 3, 61, 20, 49, 24]
Result After Quick Sort: [3, 4, 20, 24, 49, 49, 60, 60, 61, 72]
Quick Sort Input: [73, 9, 65, 88, 96, 69, 42, 12, 18, 19]
Result After Quick Sort: [9, 12, 18, 19, 42, 65, 69, 73, 88, 96]
Quick Sort Input: [73, 33, 1, 16, 26, 78, 5, 34, 49, 38]
Result After Quick Sort: [1, 5, 16, 26, 33, 34, 38, 49, 73, 78]
Quick Sort Input: [21, 4, 21, 56, 82, 91, 15, 2, 75, 78]
Result After Quick Sort: [2, 4, 15, 21, 21, 56, 75, 78, 82, 91]
Quick Sort Input: [30, 36, 53, 65, 36, 0, 7, 14, 5, 17]
Result After Quick Sort: [0, 5, 7, 14, 17, 30, 36, 36, 53, 65]
Quick Sort Input: [24, 27, 4, 48, 15, 6, 5, 3, 25, 20]
Result After Quick Sort: [3, 4, 5, 6, 15, 20, 24, 25, 27, 48]
Quick Sort Input: [19, 63, 59, 19, 39, 18, 45, 58, 19, 85]
Result After Quick Sort: [18, 19, 19, 19, 39, 45, 58, 59, 63, 85]
Quick Sort Input: [82, 36, 46, 94, 15, 15, 11, 62, 56, 88]
Result After Quick Sort: [11, 15, 15, 36, 46, 56, 62, 82, 88, 94]
```