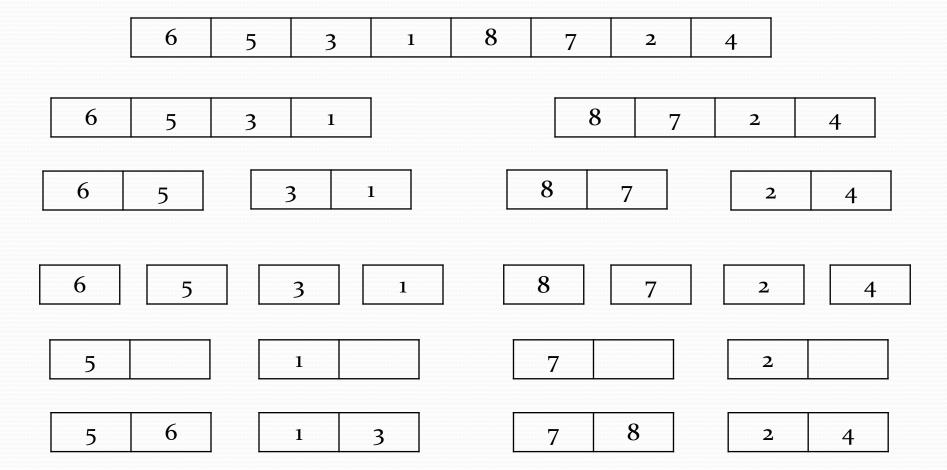
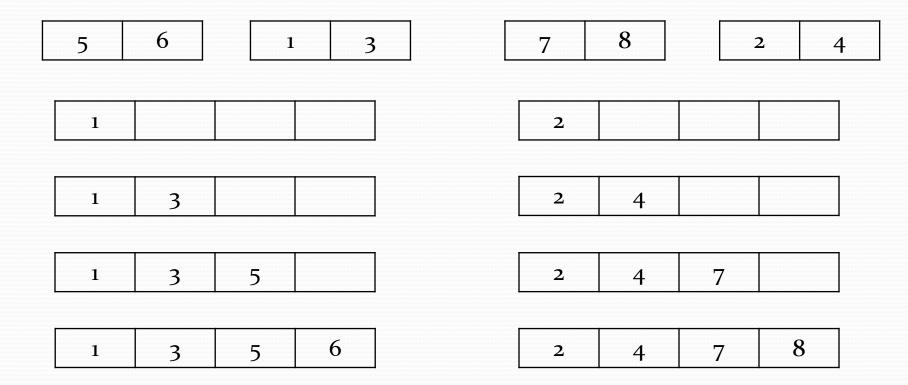
Computer Programming I
Sorting Algorithms, Part 2

Merge Sort Algorithm

- Start with an unordered list of elements.
- Repeatedly break this list into roughly equal parts.
- Continue until each part consists of only a single element.
 - A single element is sorted!
- Merge each sorted part (starting from single elements).
 - Put each element in the proper position
- https://www.youtube.com/watch?v=XaqR3G_NVoo

6 5 3 1 8 7 2 4





1	3	5	6			2	4	7	8
	1								
	1	2							
	1	2	3						
	1	2	3	4					
	1	2	3	4	5				
	1	2	3	4	5	6			
	1	2	3	4	5	6	7		
	1	2	3	4	5	6	7	8	

```
def merge sort(values, fromIndex, toIndex):
    if fromIndex < toIndex:
        #Find midpoint in the partition
        mid = (fromIndex + toIndex) // 2
        #Recursively sort left and right partitions
        merge sort(values, fromIndex, mid)
        merge sort(values, mid + 1, toIndex)
        #Merge left and right partition in sorted order
        merge(values, fromIndex, mid, toIndex)
```

```
#Add smallest element from left or right partition to merged values
while left pos <= mid and right pos <= toIndex:
    if values[left pos] < values[right pos]:</pre>
        merged values[merge pos] = values[left pos]
        left pos = left pos + 1
    else:
        merged values[merge pos] = values[right pos]
        right pos = right pos + 1
    merge pos = merge pos + 1
#If left partition is not empty, add remaining elements to merged values
while left pos <= mid:
    merged values[merge pos] = values[left pos]
    left pos = left pos + 1
    merge pos = merge pos + 1
```

```
#If right partition is not empty, add remaining elements to merged values
while right_pos <= toIndex:
    merged_values[merge_pos] = values[right_pos]
    right_pos = right_pos + 1
    merge_pos = merge_pos + 1

#Copy merge number back to values
merge_pos = 0
while merge_pos < merged_size:
    values[fromIndex + merge_pos] = merged_values[merge_pos]
    merge pos = merge pos + 1</pre>
```

Quick Sort

- In Quicksort we repeatedly partition the list into low and high parts (each part unsorted),
 - move large elements to one side of the list and small ones to the other
- Then recursively sort each of those parts.
- To partition the list, quicksort chooses a pivot to divide the values into low and high parts.
- https://www.youtube.com/watch?v=ywWBy6J5gz8&t=202s

Quick Sort – Algorithm

- 1. Choose a **Pivot** Element This element *will be* put somewhere in the "middle" of the list.
 - The left side of the list will hold all the elements smaller than or equal to the pivot element (low/left partition)
 - The right side will hold all elements larger than the pivot (high/right partition)
 - The pivot location may depend on the algorithm used. We'll use the middle element of a sub-list as the pivot.
- 2. Partition the current list in two parts: [elements <= pivot] [elements > pivot]
- 3. Recursively quicksort the left and right partitions (back to step 1)

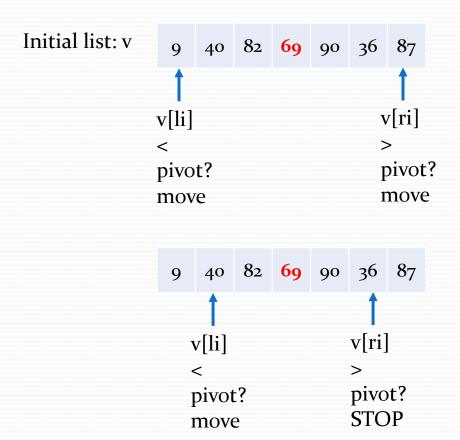
How to partition?

- The partitioning algorithm uses two index variables leftIndex and rightIndex, initialized to the left and right sides of the current elements being sorted.
- As long as the value at index leftIndex is less than the
 pivot value, the algorithm increments leftIndex, because
 the element should remain in the low partition.
- Likewise, as long as the value at index rightIndex is greater than the pivot value, the algorithm decrements rightIndex, because the element should remain in the high partition.

How to partition?

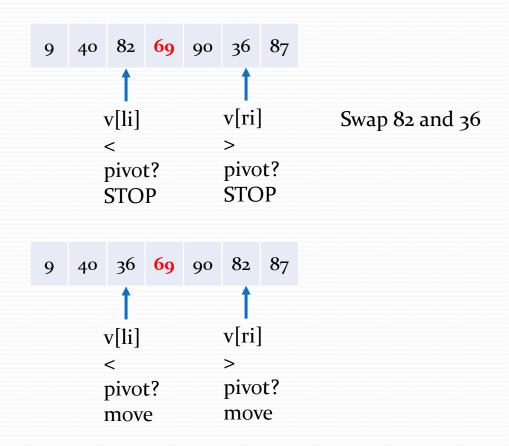
- Then, if leftIndex >= rightIndex, all elements have been partitioned, and the partitioning algorithm returns rightIndex, which is the index of the last element in the left partition.
- Otherwise, the elements at leftIndex and rightIndex are swapped to move those elements to the correct partitions.
- The algorithm then increments leftIndex, decrements rightIndex, and repeats.

Quicksort - Partition

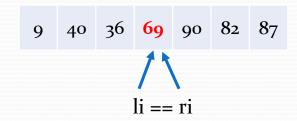


Pivot shown in red li = left index ri = right index

Quicksort - Partition



Quicksort - Partition



Partition point found



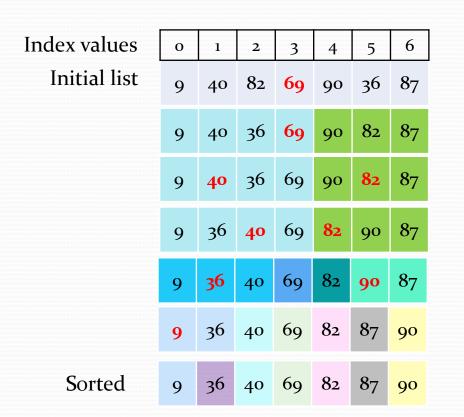
Left/low partition

Right/high partition

Repeat the process with the left partition

Repeat the process with the right partition

Quicksort



Pivot shown in red

Quick Sort Implementation

```
def quick sort(values, fromIndex, toIndex):
    splitPoint = 0
    #Base case: If there are 1 or zero items to sort, partition
    #is already sorted
    if fromIndex < toIndex:
        #Partition the data within the list. Value splitPoint
        #returned from partitioning is location of last item in
        #low partition
        splitPoint = partition(values, fromIndex, toIndex)
        #Recursively sort low partition (fromIndex to splitPoint)
        #and high partition (splitPoint + 1 to toIndex)
        quick sort(values, fromIndex, splitPoint)
        quick sort(values, splitPoint + 1, toIndex)
```

Quick Sort Implementation

```
def partition (values, fromIndex, toIndex):
    # Pick middle element as pivot
    midpoint = fromIndex + (toIndex - fromIndex) // 2
   pivot = values[midpoint]
    # Initialize variables
    done = False
    leftIndex = fromIndex
    rightIndex = toIndex
    while not done:
        # Increment leftIndex while values[leftIndex] < pivot
        while values[leftIndex] < pivot:</pre>
            leftIndex = leftIndex + 1
        # Decrement rightIndex while pivot < values[rightIndex]</pre>
        while pivot < values[rightIndex]:</pre>
            rightIndex = rightIndex - 1
```

Quick Sort Implementation

```
# If there are zero or one items remaining, all numbers are
# partitioned. Return rightIndex
if leftIndex >= rightIndex:
    done = True
else:
    # Swap values[leftIndex] and values[rightIndex]
    # update low and high
    swap(values, leftIndex, rightIndex)
    leftIndex = leftIndex + 1
    rightIndex = rightIndex - 1
```

return rightIndex