



MERRIMACK COLLEGE

CSC 6013

Week 7

Divide-and-Conquer Algorithms

Algorithms and Discrete Structures - Dr. Paulo Fernandes

Presentation Agenda

Week 7

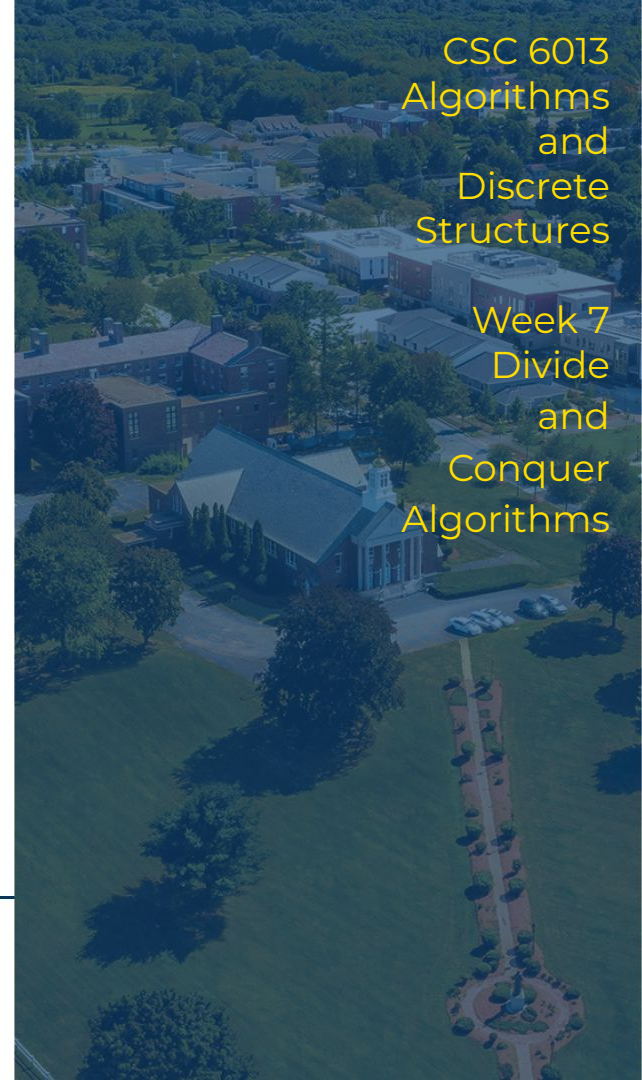
- Divide-and-Conquer
 - a. Meet the family
 - b. Basic Principles
 - c. Recursion and Iteration
- Examples
 - a. Array sum
 - b. Height of Binary Tree
 - c. Mergesort
 - d. Quicksort
- This Week's tasks



MERRIMACK COLLEGE

CSC 6013
Algorithms
and
Discrete
Structures

Week 7
Divide
and
Conquer
Algorithms



Divide-and-Conquer Algorithms

Solve the problem by solving each of its parts.

Some of the better known algorithms belong to the ...-and-conquer family.

Some of the more effective algorithms are divide-and-conquer.

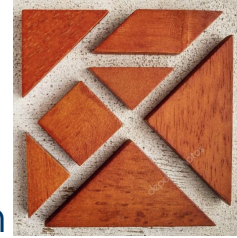
- **Meet the family**
 - Decrease-and-Conquer
 - Divide-and-Conquer
 - Transform-and-Conquer
- **Divide-and-Conquer**
 - Basic Principles
 - Recursion and Iteration



Meet the ...-and-conquer family

While in military, politics, management, etc., the usage of divide-and-conquer became extremely popular, and effective, in algorithms we have a subdivision of the algorithm techniques:

- **Divide-and-conquer**
 - When the original problem is broken into complementary parts;
- **Decrease-and-conquer**
 - When at each time the problem becomes smaller, but not into complementary problems;
- **Transform-and-conquer**
 - When the problem not always becomes smaller.



MERRIMACK COLLEGE

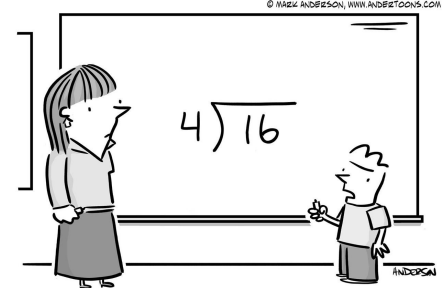
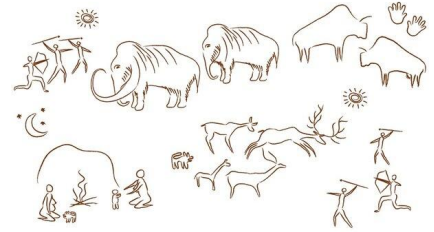
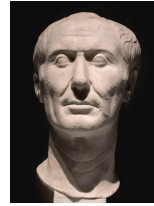
This division is acknowledged by great authors (as our textbook ones: Cormen, Leiserson, Rivest, and Stein), but it is not unanimous.

The ...-and-conquer family

Probably, one of the most known algorithmic techniques is the **divide-and-conquer**. It has been talked about consistently (documented with this name) at least since the times of Julius Caesar, i.e., 2000 years ago!

However, there are other similar approaches to solve problems, and those are known as the **...-and conquer family** in the algorithm analysis context.

Divide-and-conquer is the big brother of the ...-and-conquer family. Most of the more effective algorithms are divide-and-conquer ones.



Divide-and-Conquer Algorithms

Solve the problem by solving each of its parts.

Some of the better known algorithms belong to the ...-and-conquer family.

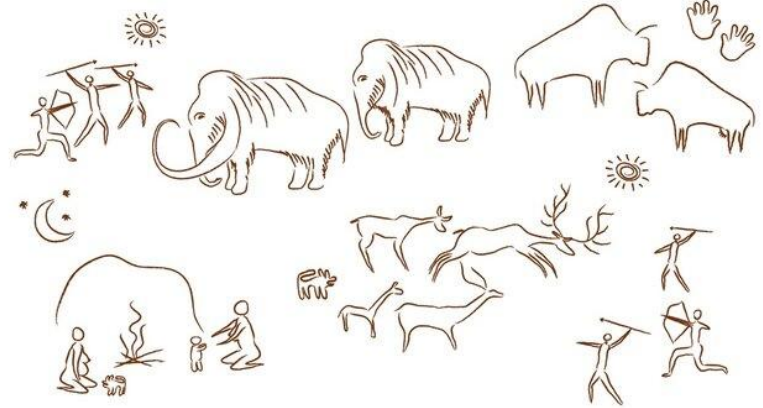
Some of the more effective algorithms are divide-and-conquer.

- Meet the family
 - Decrease-and-Conquer
 - Divide-and-Conquer
 - Transform-and-Conquer
- **Divide-and-Conquer**
 - Basic Principles
 - Recursion and Iteration



Divide-and-Conquer

Tackling a problem by dividing it into subproblems is so intuitive that sometimes we wonder if this technique can be said to be invented. Splitting our tasks into complementary subtasks has been done perhaps even before we learned how to speak.



If you have to carry a large lego statue, it does make sense to break it into pieces to be able to carry it.

If you have a pile of boxes to move upstairs, it makes sense to not carry them all together.



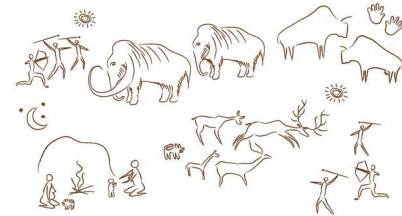
MERRIMACK COLLEGE

Julius Caesar was very smart, but not the creator of Divide-and-Conquer.

Divide-and-Conquer

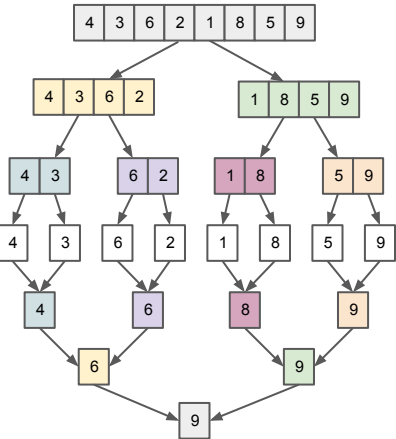
While decrease-and-conquer can be implemented **recursively or iteratively**, the very large majority of divide-and-conquer algorithms are implemented **recursively** because it is very natural keep on breaking a problem into parts until it becomes trivial.

Also it is much natural to break into halves, as we will see, it makes the algorithms simpler and more efficient.



With some effort we can always manage, often with large effort and no gain, to turn a divide-and-conquer implementation into a non-recursive (iterative) version.

```
1 def Max(A, start, end):
2     if (start == end):
3         return end
4     else:
5         mid = (end+start)//2
6         fst = Max(A, start, mid)
7         lst = Max(A, mid+1, end)
8         return fst if A[fst] > A[lst] else lst
9
10 from random import randint
11 A = [randint(0,100000) for _ in range(1000)]
12 i = Max(A, 0, len(A)-1)
13 print("The maximum number is", A[i], "at index", i)
```



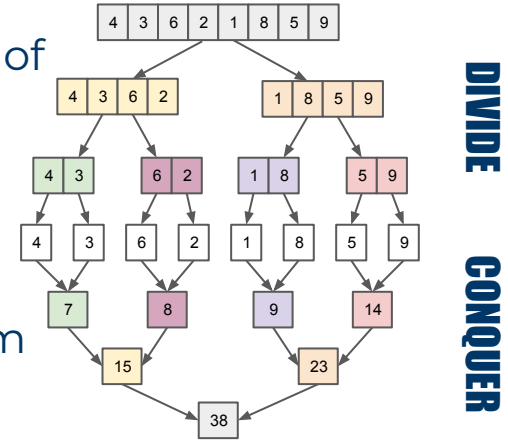
A collection of hand-drawn illustrations of prehistoric life. The drawings are in a simple, sketchy style. At the top left, two figures are shown one holding a spear and the other a bow. To their right is a sun. In the center are two mammoths. To the right of the mammoths are two bison and two handprints. Below the mammoths is a crescent moon. In the bottom left, a figure is shown with a large animal skin or hide. To the right of this is a small pile of what looks like food or tools. In the center bottom are two horses. To the right of the horses is a deer. In the bottom right, two figures are shown one holding a spear and the other a bow. A sun is also present in the bottom right area.

- **Array sum**
- Mergesort
- Quicksort
- Height of Binary Tree



Example 1 - Array sum

- How to sum all elements of and array **A** composed of **n** elements?
- You can go iteratively adding up all of them, sure, and this is not bad at all, but there is divide and conquer way to do it as well.
- What if we say we split the array in two and the sum is the sum of the two parts.
 - ... and we keep on doing that until we have a single element where the obvious sum is the element itself.

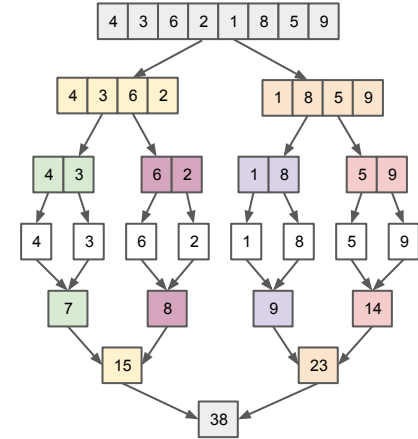


```
1 def arraySum(A, start, end):  
2     if (start == end):  
3         return A[start]  
4     else:  
5         mid = (start+end)//2  
6         return arraySum(A, start, mid) + arraySum(A, mid+1, end)
```



Example 1 - Array sum

The sum all elements of and array **A** composed of **n** elements is given by recursively add the two halves of the array, until we have a single element where the obvious sum is the element itself.



```
1 def arraySum(A, start, end):
2     if (start == end):
3         return A[start]
4     else:
5         mid = (start+end)//2
6         return arraySum(A, start, mid) + arraySum(A, mid+1, end)
7
8 from random import shuffle
9 A = list(range(1000))
10 shuffle(A)
11 print("The sum is:", arraySum(A, 0, 999))
12 print("... and using Gauss it should be:", (999 * 1000)//2)
```



- $T(n) = 2 T(n/2) + 2$
- $T(1) = 0$

Master method:

- $a=2, b=2, f(n)=n^0, \log_2=1$

→ if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$



- Array sum
- **Height of Binary Tree**
- Mergesort
- Quicksort

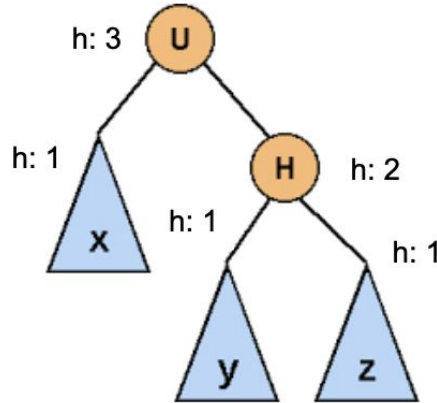


Example 2 - Height of Binary Tree

- Giving a tree with n nodes
- What is the height of the tree (height of the root)?

The algorithm simply consider the height of a node as plus one of the tallest child subtree.

The height of a tree is equal to the maximum depth of this tree, since both are the distance between the root and the deepest leaf.



```
1 class Node:
2     def __init__(self, d):
3         self.data = d
4         self.Left = None
5         self.Right = None
```

```
7 def heightTree(node):
8     if (node == None):
9         return -1
10    else:
11        left = heightTree(node.Left)
12        right = heightTree(node.Right)
13        if (left < right):
14            return right + 1
15        else:
16            return left + 1
```



Example 2 - Height of Binary Tree

- $T(n) = 2T(?) + 1$
- $T(1) = 0$

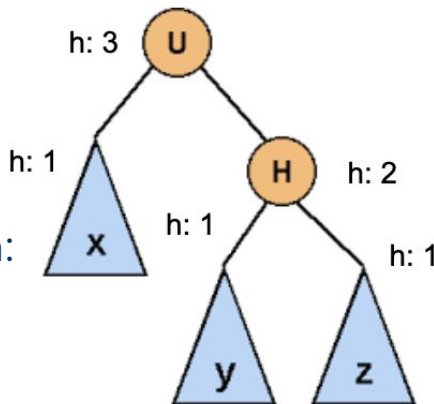
Worst case:

- $T(n) = T(n-1) + 1$
- Back-substitution:
 - $O(n)$

Best case:

- $T(n) = 2T(n/2) + 1$
- Master method:
 - $a=2, b=2, f(n)=1, \log_2 2=1$

➡ if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$



```
1 class Node:
2     def __init__(self, d):
3         self.data = d
4         self.Left = None
5         self.Right = None
```

```
7 def heightTree(node):
8     if (node == None):
9         return -1
10    else:
11        left = heightTree(node.Left)
12        right = heightTree(node.Right)
13        if (left < right):
14            return right + 1
15        else:
16            return left + 1
```



A collection of hand-drawn illustrations of prehistoric life, including mammoths, bison, deer, and humans, set against a dark blue background. The drawings are in a simple, sketchy style using white and yellow lines. The scene includes a sun, a crescent moon, and various animals in different poses, suggesting a narrative of daily life or a hunt.

- Array sum
- Height of Binary Tree
- **Mergesort**
- Quicksort

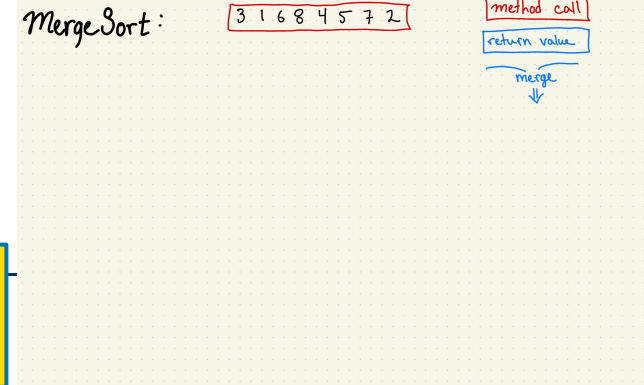
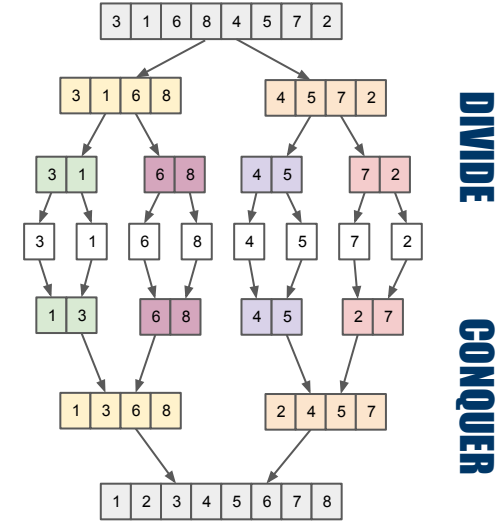


Example 3 - Mergesort

- Giving an array $A = [a_1, a_2, \dots, a_n]$
- Deliver a permutation of A where $a_i \leq a_j$ if $i < j$

The Mergesort performs the sort:

- splitting the array in two halves and keep splitting until you end up with a single element, thus sorted;
 - this is the **divide** part;
- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;
 - this is the **conquer** part.



MERRIMACK COLLEGE

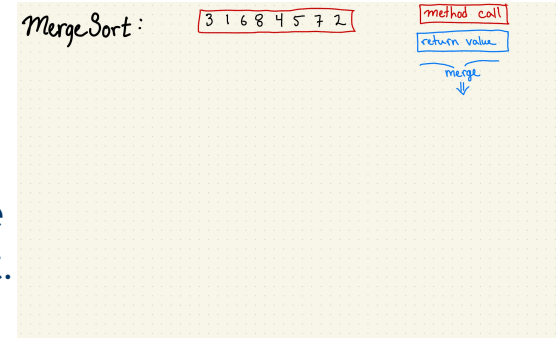
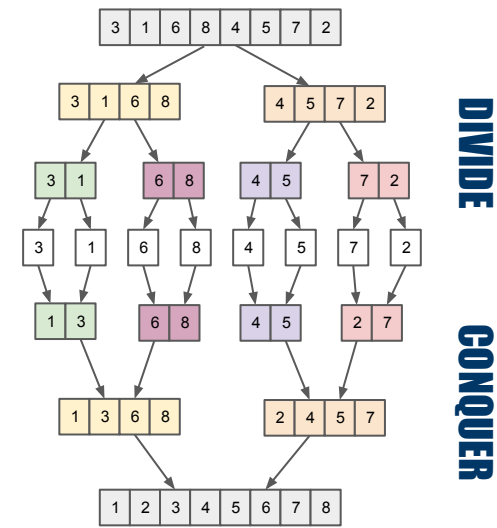
Mergesort is very time efficient.

Example 3 - Mergesort

The Mergesort performs the sort:

- splitting the array in two halves and keep splitting until you end up with a single element, thus sorted;
 - this is the **divide** part.
- if it is not a single element:
 - split in two;
 - recursive call on left;
 - recursive call on right;
 - call the merge (conquer) part.

```
1 def mergesort(A):  
2     if len(A) <= 1:  
3         return A  
4     else:  
5         mid = len(A) // 2  
6         left = mergesort(A[:mid])  
7         right = mergesort(A[mid:])  
8         return merge(left, right)
```



Example 3 - Mergesort

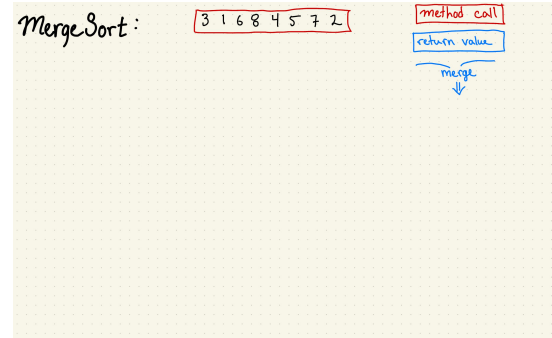
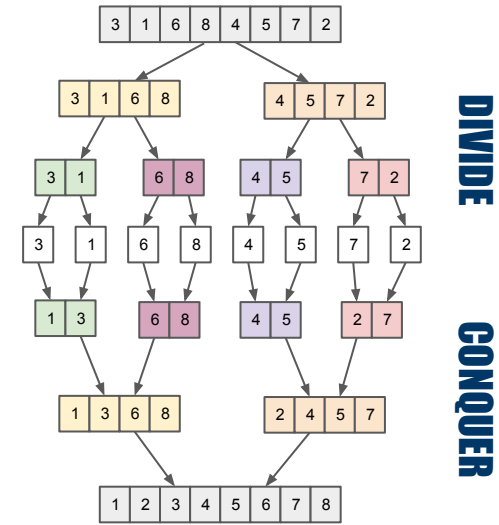
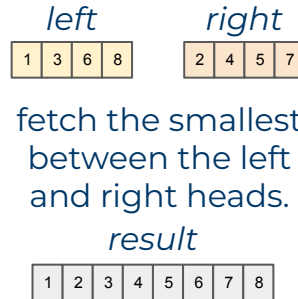
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



MERRIMACK COLLEGE

Note that the input are two arrays, and a third one is returned (requires more memory).

Example 3 - Mergesort

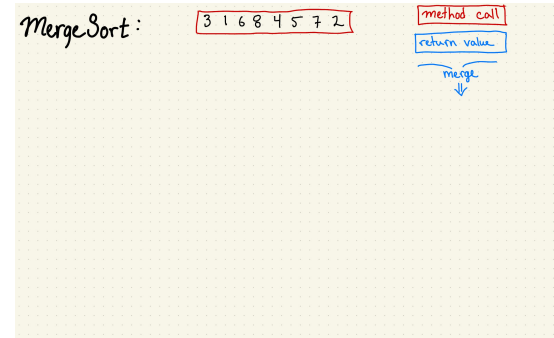
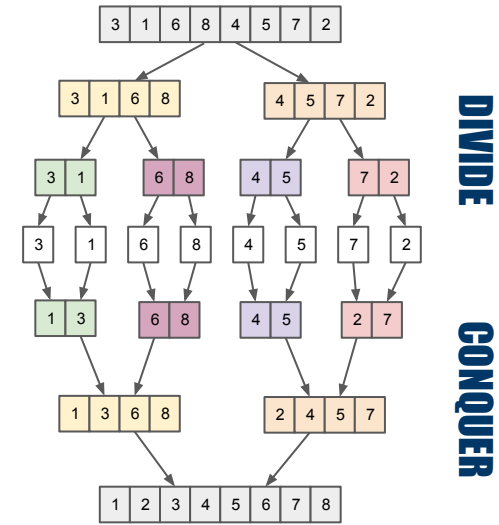
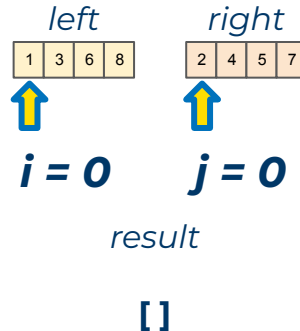
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

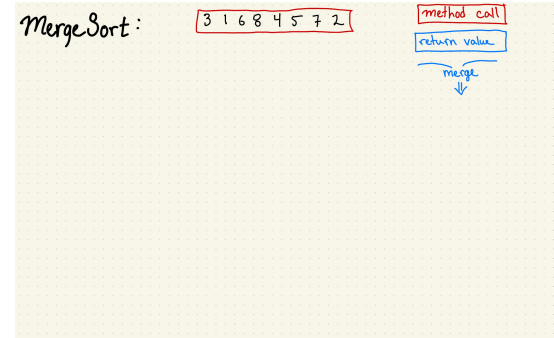
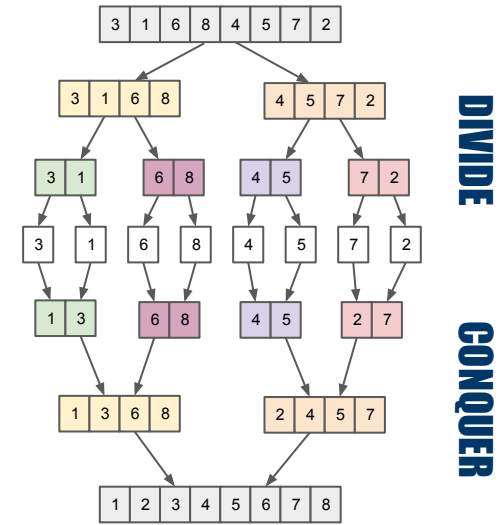
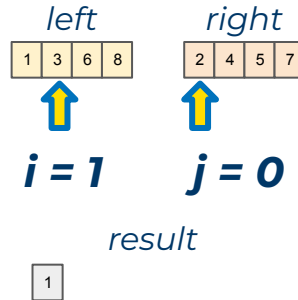
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

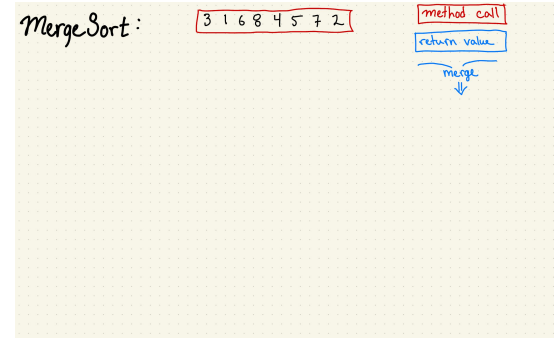
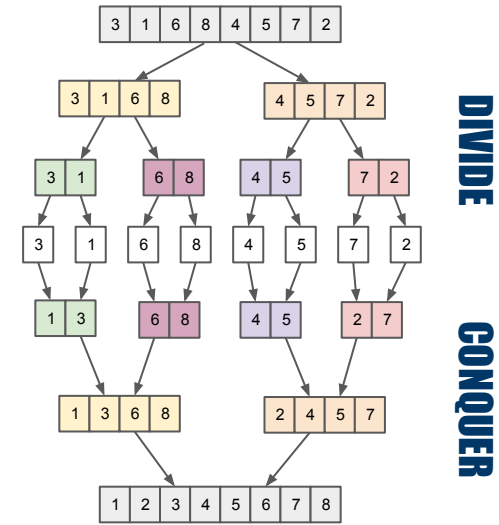
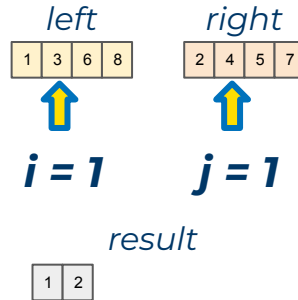
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

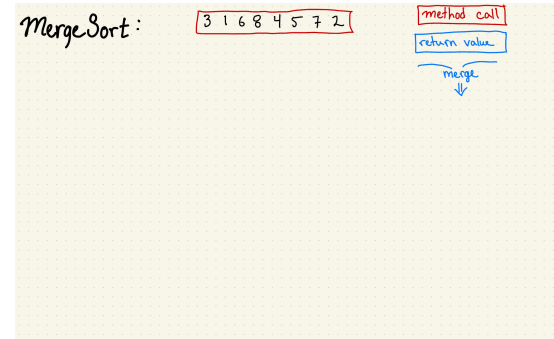
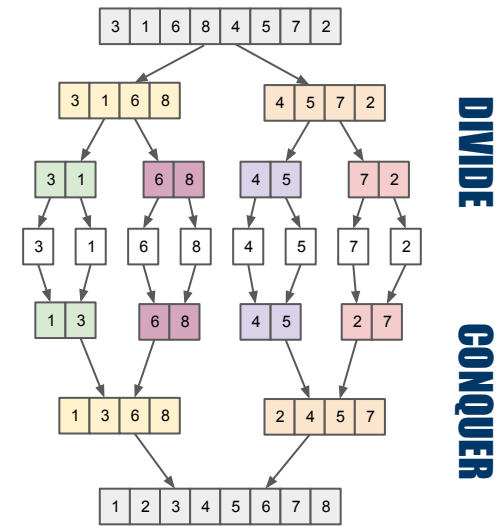
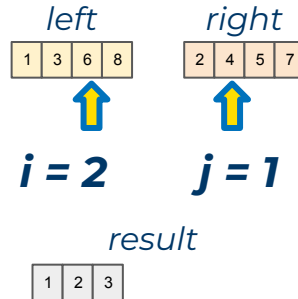
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

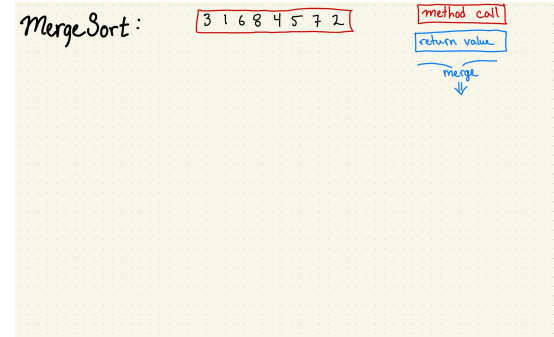
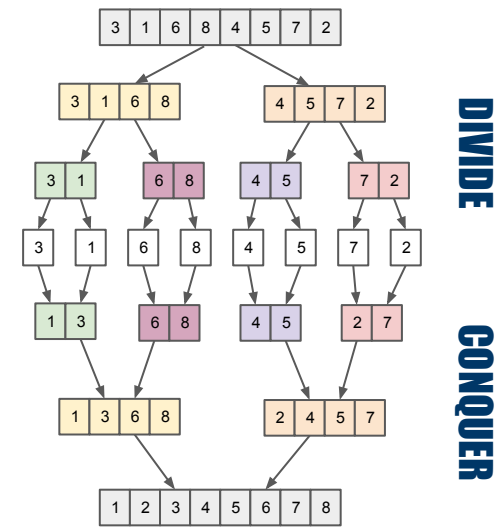
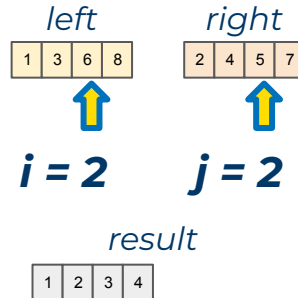
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

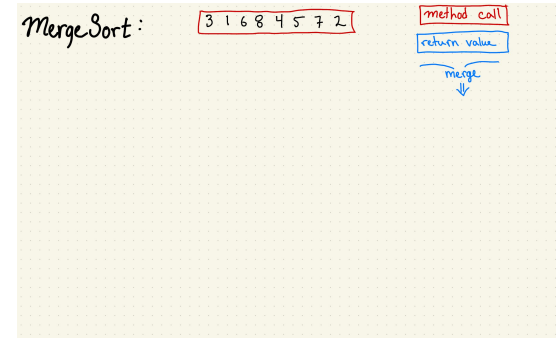
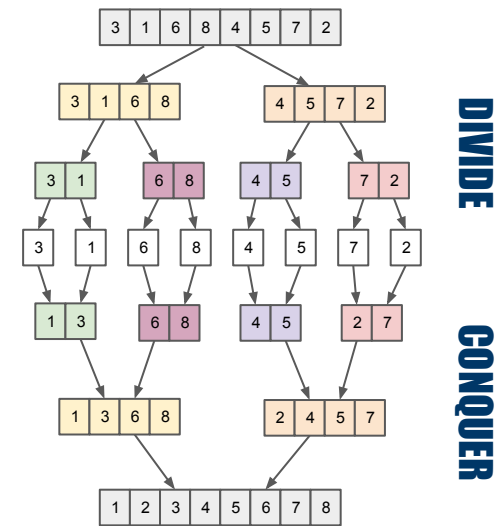
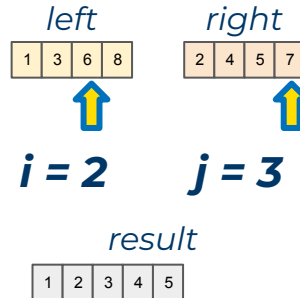
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

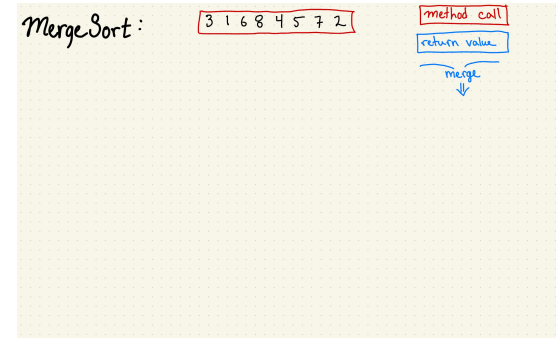
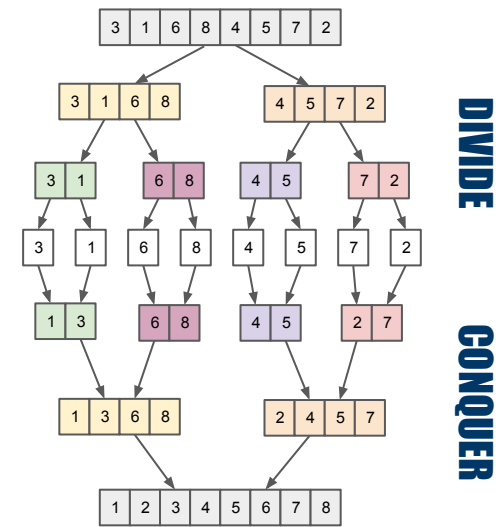
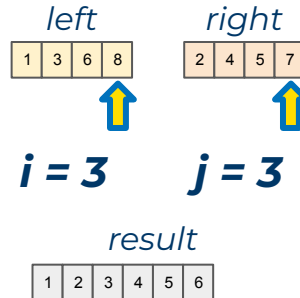
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

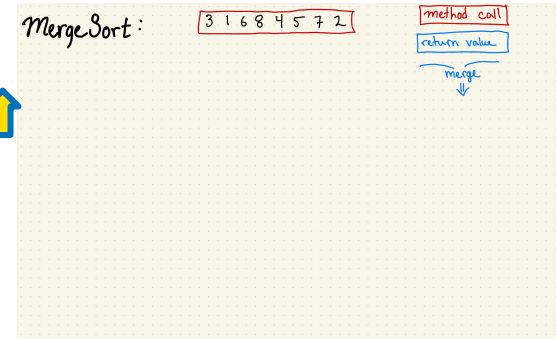
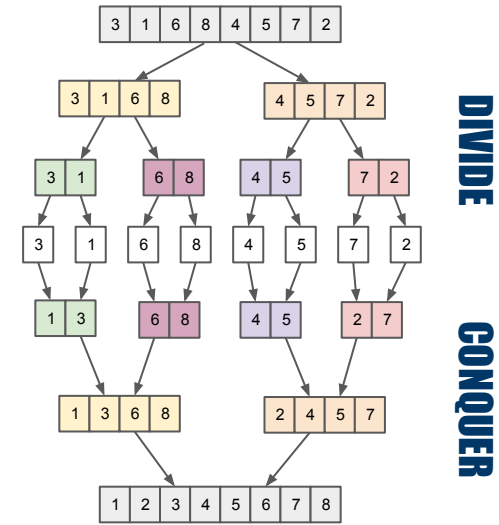
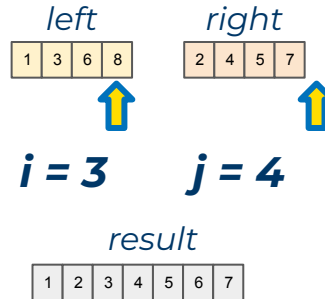
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort

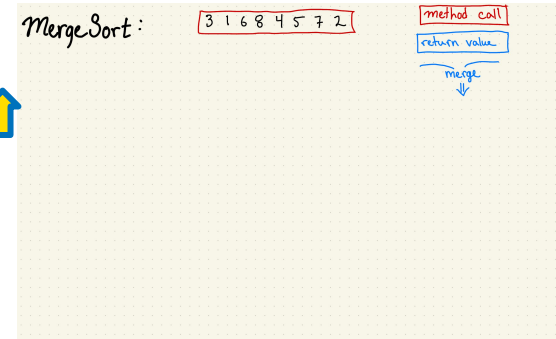
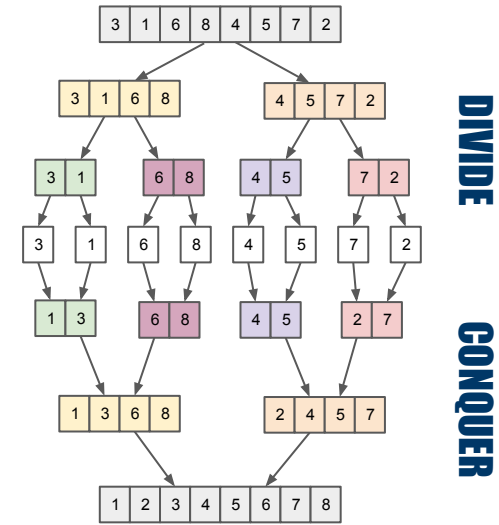
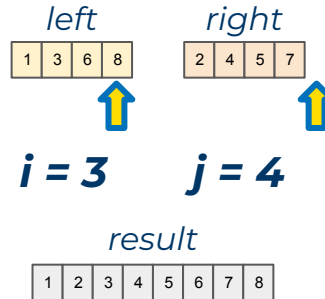
The Mergesort performs the sort:

- merging the sorted halves into a sorted array repeatedly, until the whole array is sorted;

```
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if (left[i] <= right[j]):
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
```



- this is the **conquer** part.



Example 3 - Mergesort


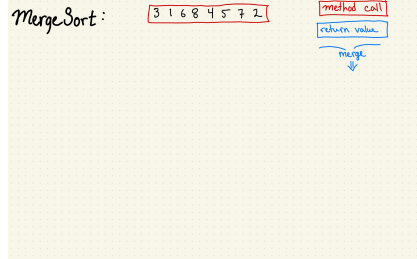
- $T(n) = 2 T(n/2) + n$
- $T(1) = 0$

Master method:

- $a=2, b=2, f(n)=n, \log_2 2=1$

➔ if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$

The Mergesort algorithm requires some extra memory for the array copies created, thus, the memory complexity is also equal to $O(n \log n)$.



```
1 def mergesort(A):
2     if len(A) <= 1:
3         return A
4     else:
5         mid = len(A) // 2
6         left = mergesort(A[:mid])
7         right = mergesort(A[mid:])
8         return merge(left, right)
9
10 def merge(left, right):
11     result, i, j = [], 0, 0
12     while i < len(left) and j < len(right):
13         if left[i] <= right[j]:
14             result.append(left[i])
15             i += 1
16         else:
17             result.append(right[j])
18             j += 1
19     result += left[i:]
20     result += right[j:]
21     return result
22
23 A = [3, 1, 6, 8, 4, 5, 7, 2]
24 A = mergesort(A)
25 print(A)
```



A collection of prehistoric cave paintings, including animals like mammoths, bison, and deer, and human figures in various poses, set against a dark background.

- Array sum
- Height of Binary Tree
- Mergesort
- **Quicksort**

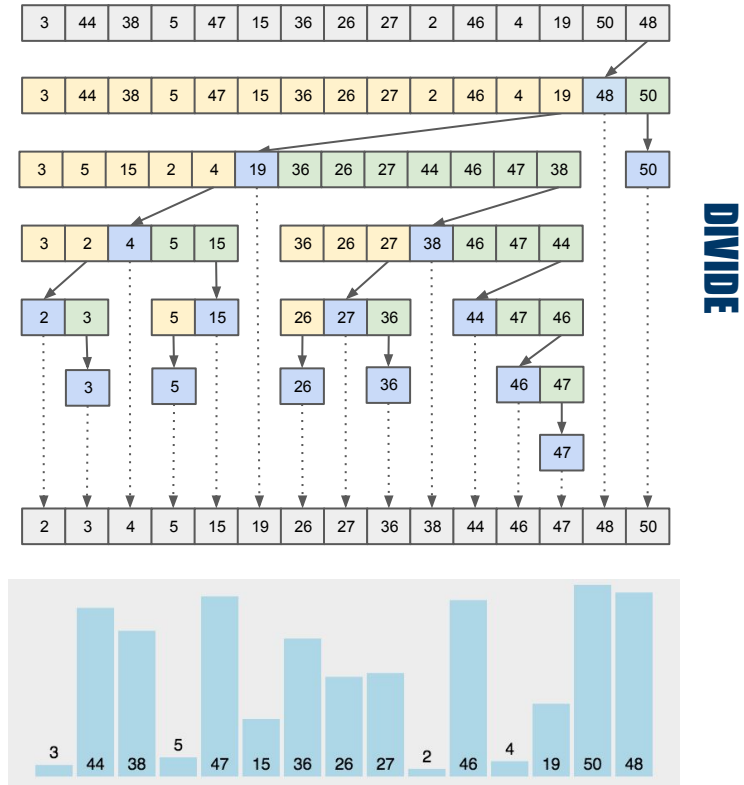


Example 4 - Quicksort

- Giving an array $\mathbf{A} = [a_1, a_2, \dots, a_n]$
- Deliver a permutation of \mathbf{A} where $a_i \leq a_j$ if $i < j$

The Quicksort performs the sort:

- calling recursively a Lomuto partition (or other splitting an array in two two parts where one has smaller elements than the other using a pivot) until you have a single element part;
 - this is the **divide** part;
- Once this is done, the sort is done;
 - this is the (empty) **conquer** part.



Example 4 - Quicksort

The Quicksort performs the sort:

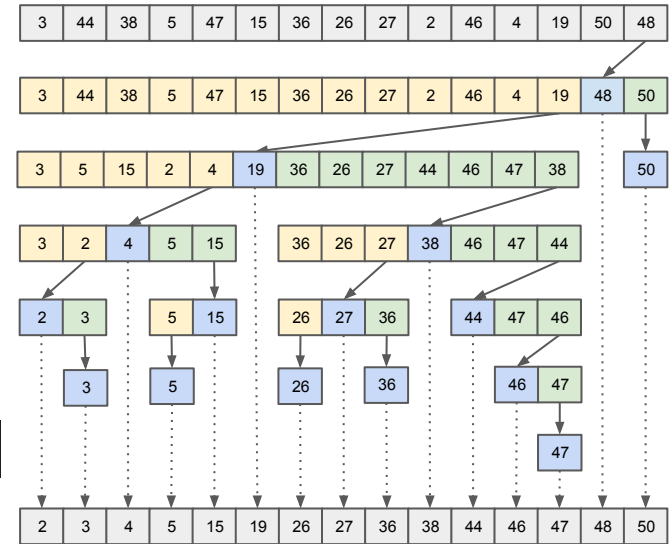
- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

```
1 def lomuto(A, left, right):  
2     p = A[right]  
3     i = left  
4     for j in range(left, right):  
5         if A[j] < p:  
6             A[i], A[j] = A[j], A[i]  
7             i += 1  
8     A[i], A[right] = A[right], A[i]  
9     return i
```

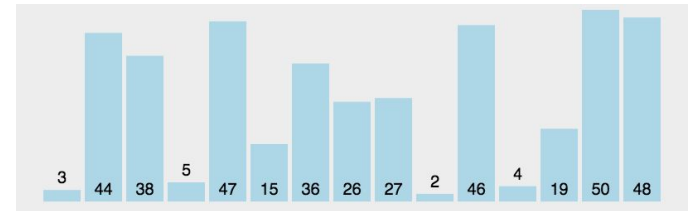
```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]  
18 quicksort(A, 0, len(A)-1)  
19 print(A)
```



```
11 def quicksort(A, left, right):  
12     if (left < right):  
13         mid = lomuto(A, left, right)  
14         quicksort(A, left, mid-1)  
15         quicksort(A, mid+1, right)
```



DIVIDE



MERRIMACK COLLEGE

Here we employ the Lomuto partition as seen before.

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

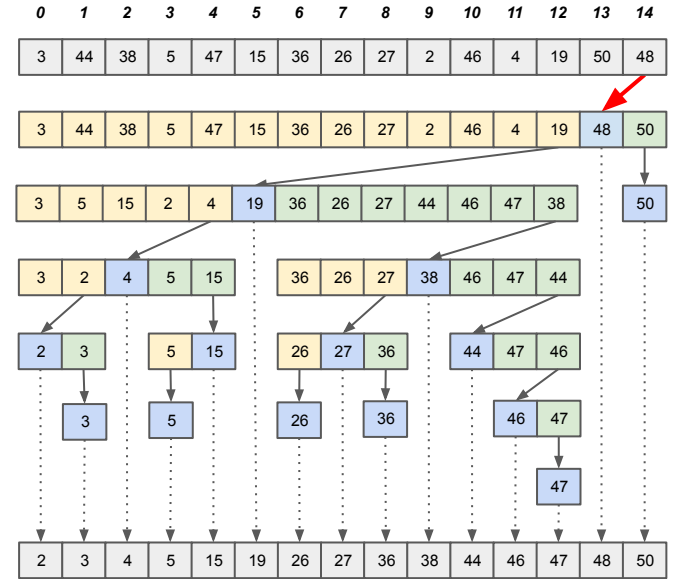
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



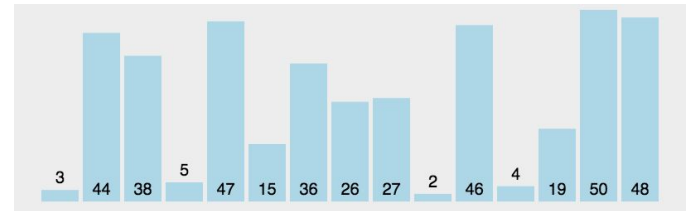
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for whole
A (15 elements)
left = 0 - right = 14

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 1 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

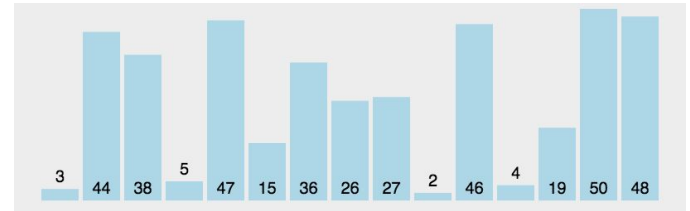
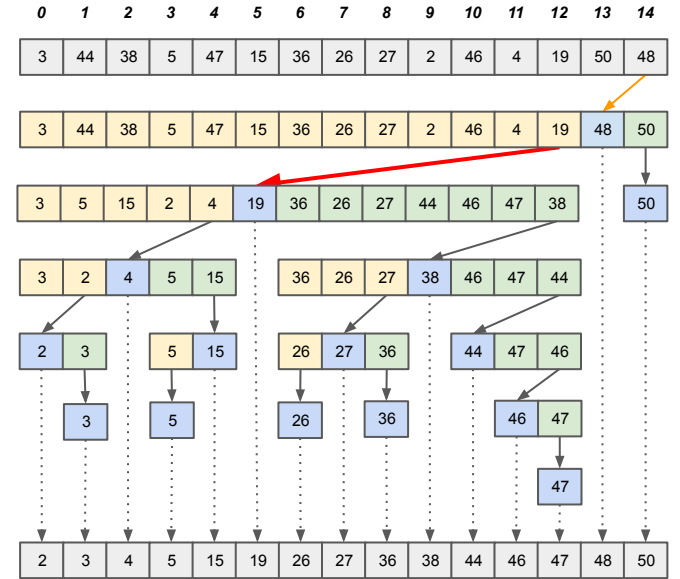
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```

Call Lomuto for just 13 elements of **A**
left = 0 - right = 12

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```



Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

```

1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
    
```



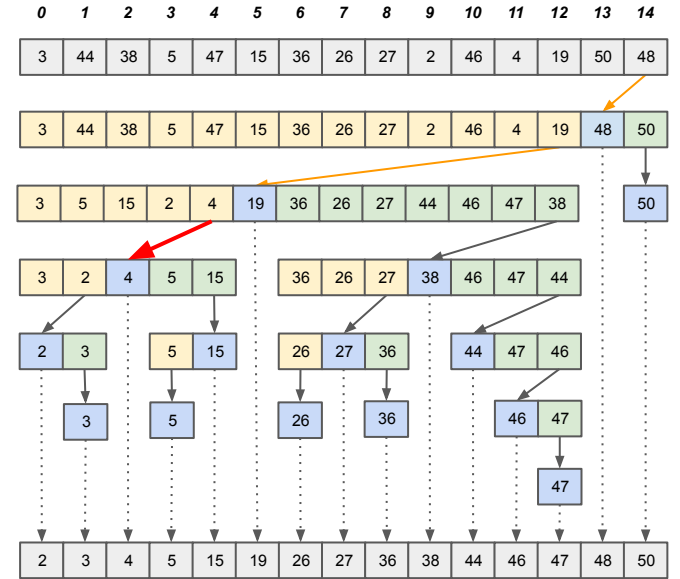
```

11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
    
```

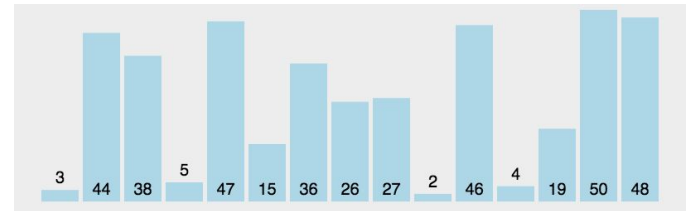
Call Lomuto for just 5 elements of **A**
left = 0 - right = 4

```

17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
    
```



DIVIDE



MERRIMACK COLLEGE

Step 3 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

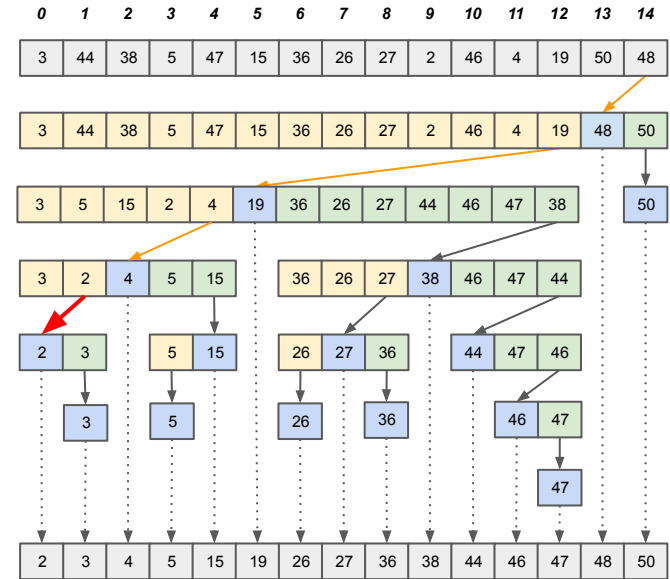
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



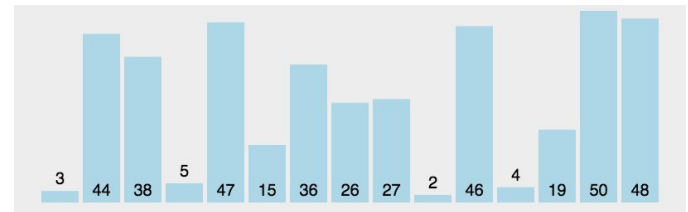
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 2 elements of **A**
left = 0 - right = 1

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 4 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

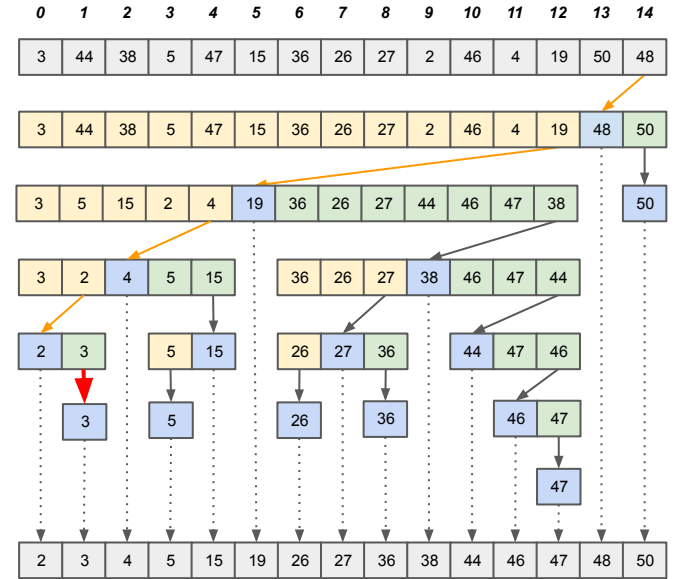
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



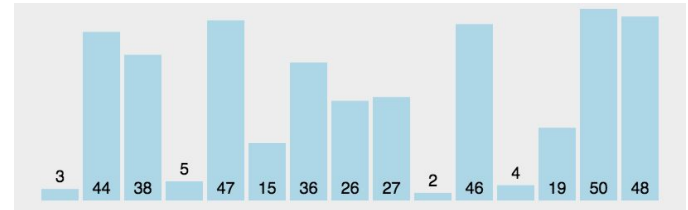
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
element of **A**
left = 2 - right = 2

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 5 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

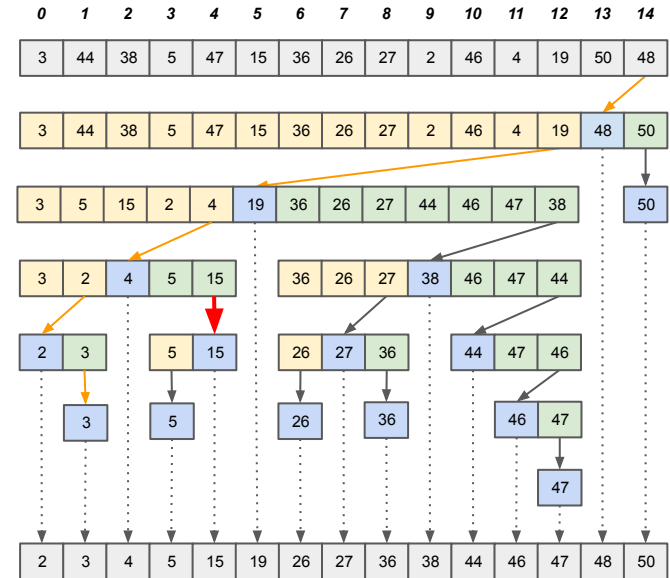
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



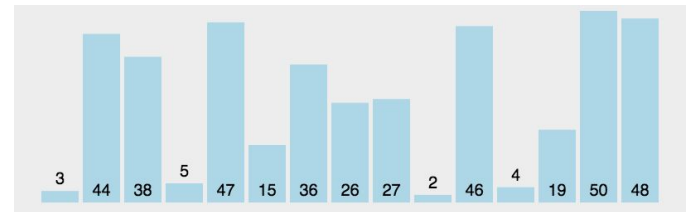
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 2 elements of **A**
left = 3 - right = 4

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 6 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

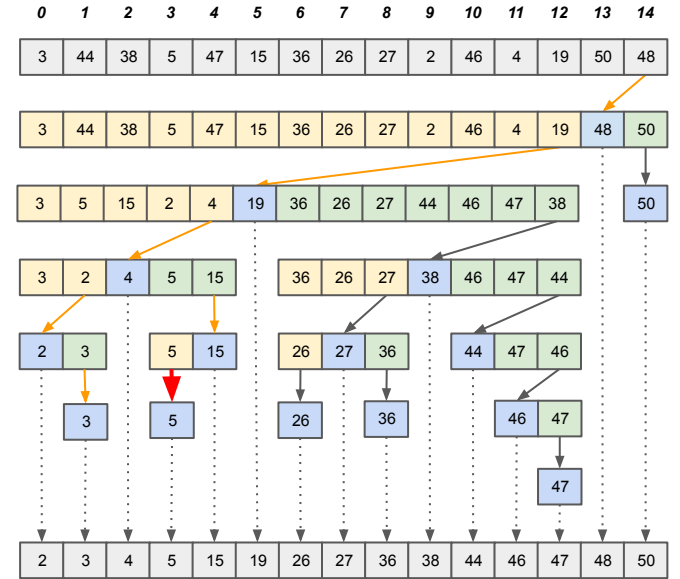
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



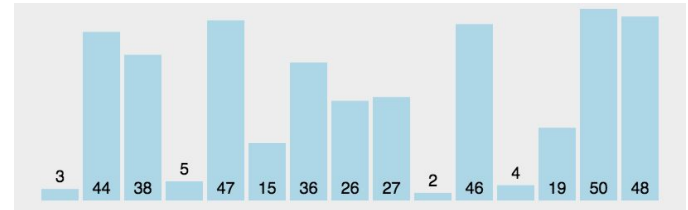
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
element of **A**
left = 3 - right = 3

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 7 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

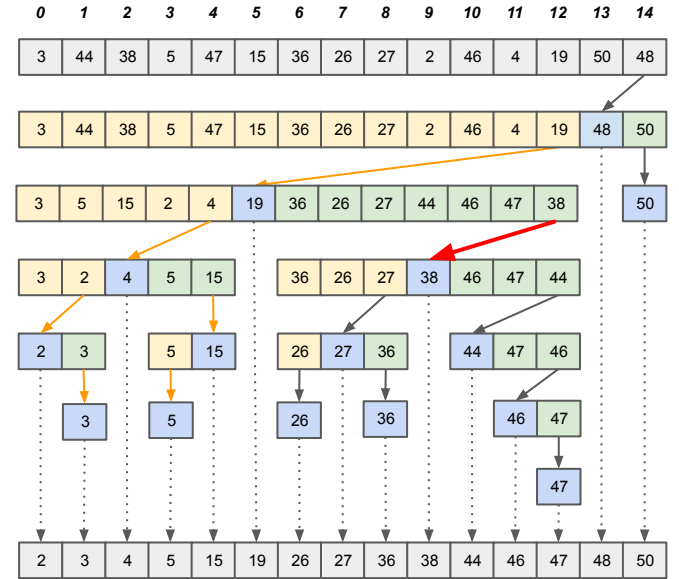
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



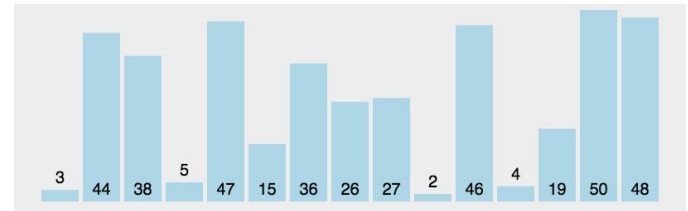
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 7
elements of **A**
left = 6 - right = 12

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 8 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

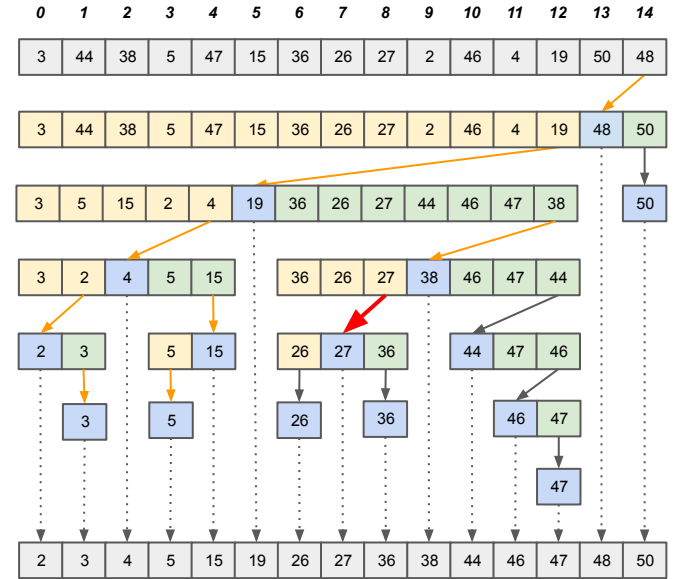
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



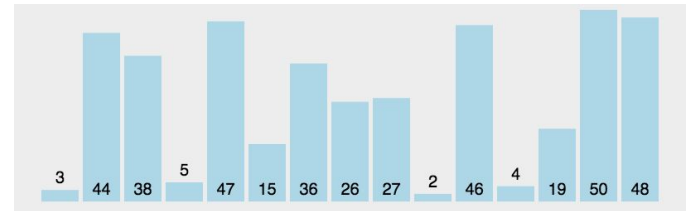
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 3
elements of **A**
left = 6 - right = 8

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 9 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

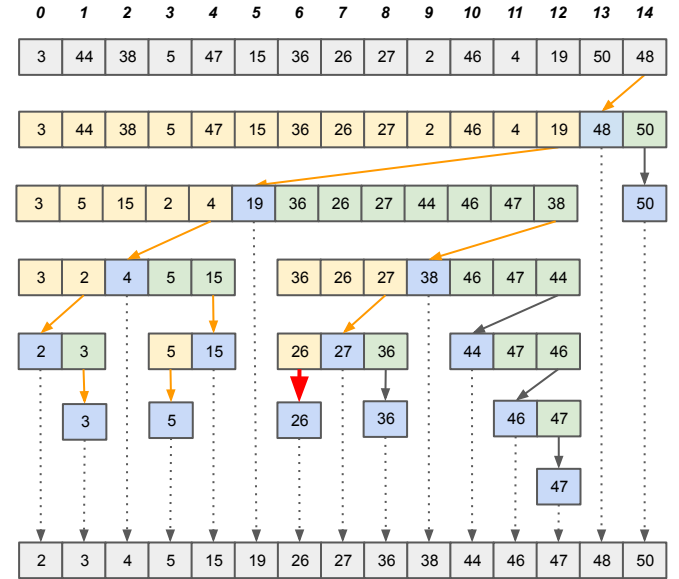
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



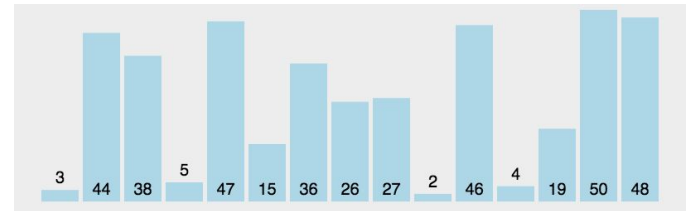
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
elements of **A**
left = 6 - right = 6

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 10 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

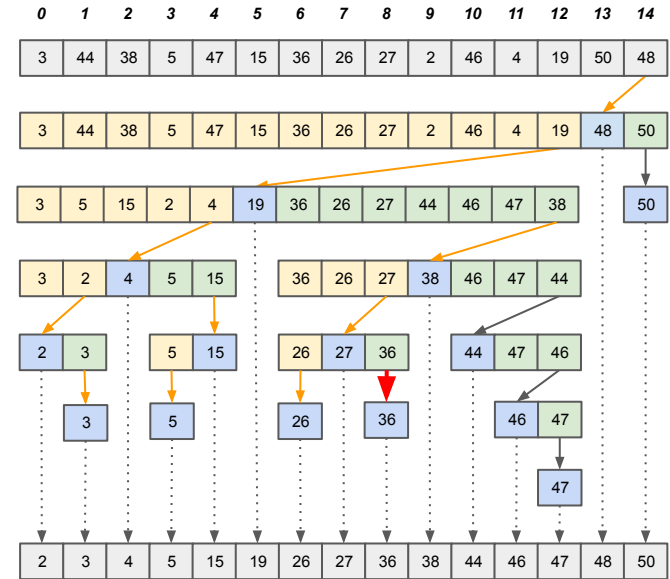
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



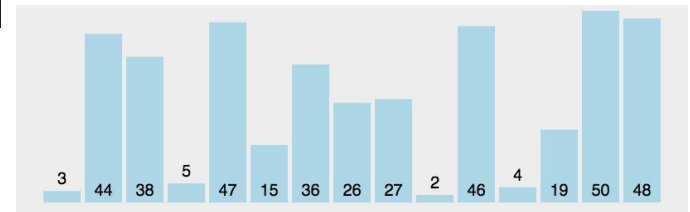
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
element of **A**
left = 8 - right = 8

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 11 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

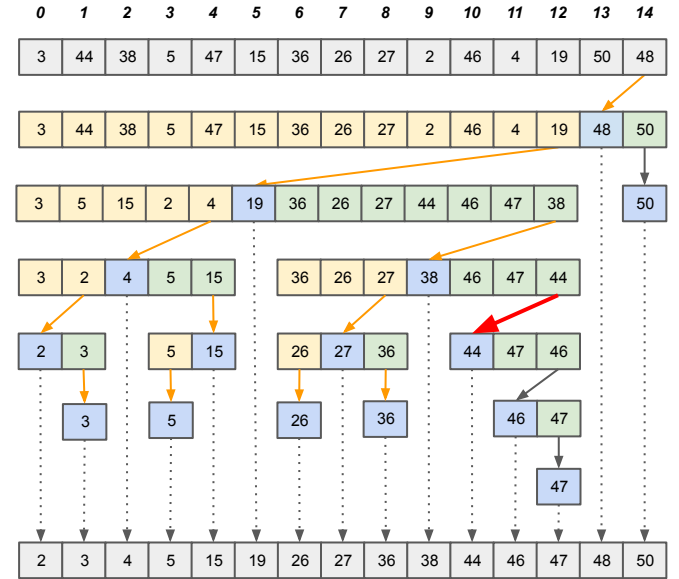
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



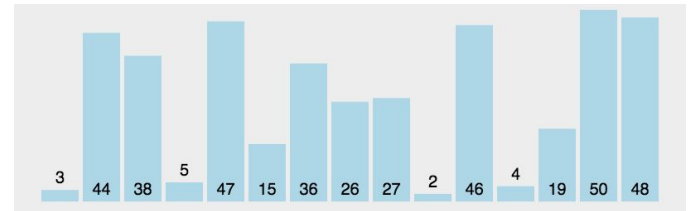
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 3
elements of **A**
left = 10 - right = 12

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 12 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

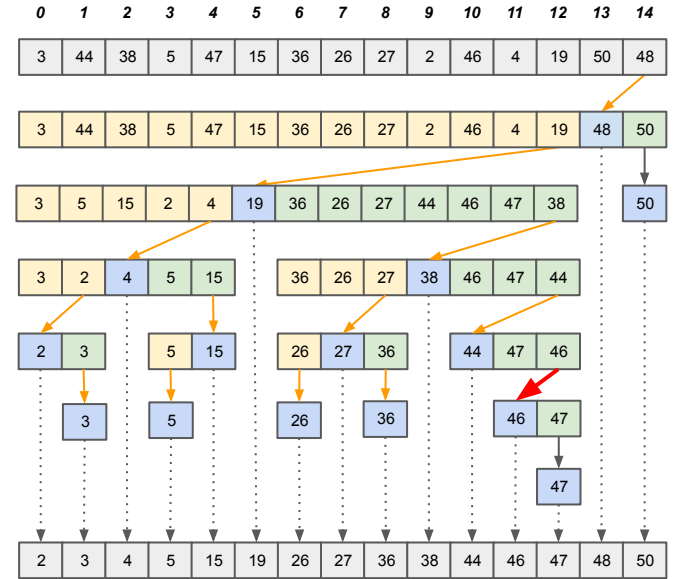
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



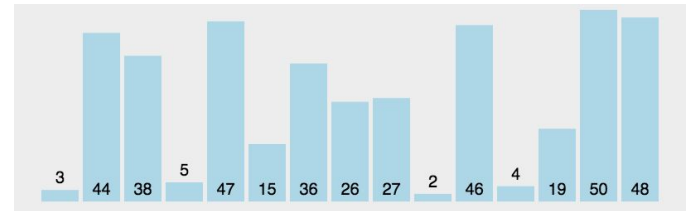
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 2
elements of **A**
left = 11 - right = 12

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 13 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

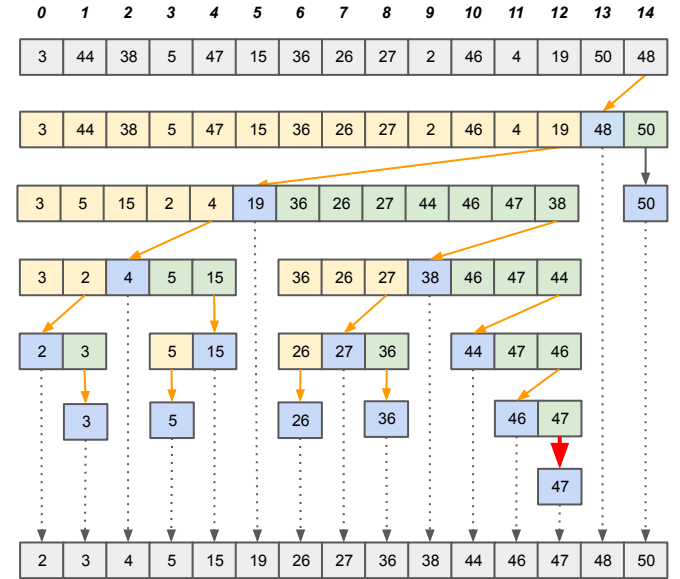
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



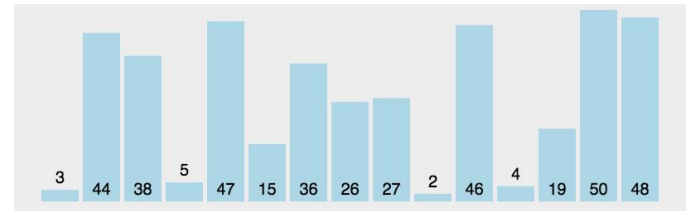
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
element of **A**
left = 12 - right = 12

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



DIVIDE



MERRIMACK COLLEGE

Step 14 of 15

Example 4 - Quicksort

The Quicksort performs the sort:

- calling recursively a Lomuto partition until you have a single element part;
 - this is the **divide** part.

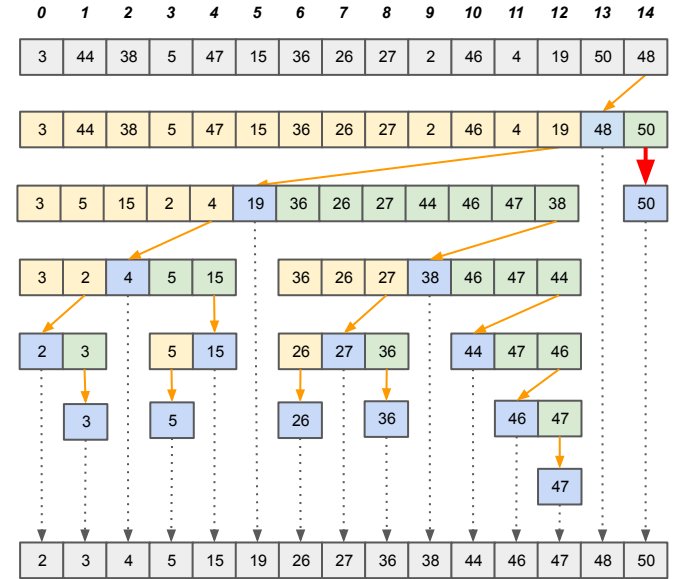
```
1 def lomuto(A, left, right):
2     p = A[right]
3     i = left
4     for j in range(left, right):
5         if A[j] < p:
6             A[i], A[j] = A[j], A[i]
7             i += 1
8     A[i], A[right] = A[right], A[i]
9     return i
```



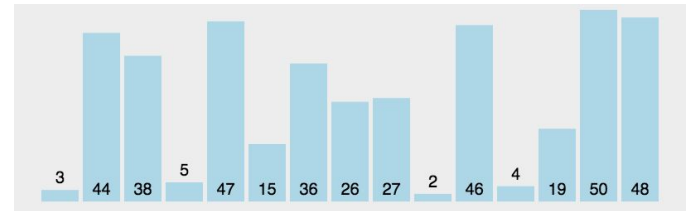
```
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
```

Call Lomuto for just 1
element of **A**
left = 14 - right = 14

```
17 A = [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48]
18 quicksort(A, 0, len(A)-1)
19 print(A)
```



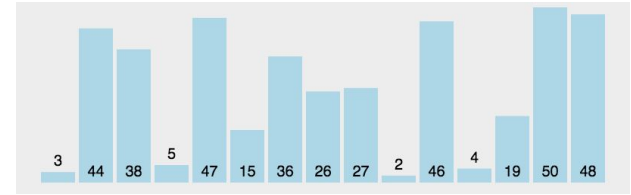
DIVIDE



MERRIMACK COLLEGE

Step 15 of 15

Example 4 - Quicksort



- $T(n) = 2T(?) + n - 1$
- $T(1) = 0$

Worst case:

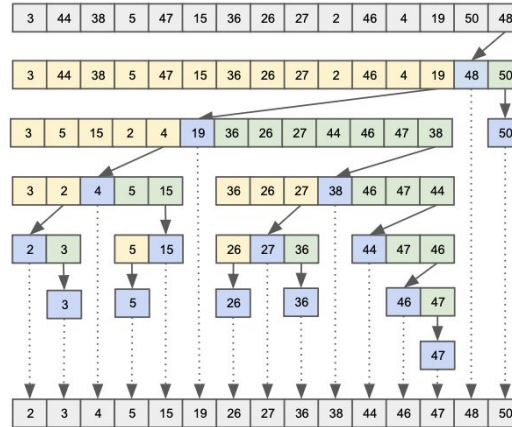
- $T(n) = T(n-1) + n - 1$
- Back-substitution:
 - $O(n^2)$

Best case:

- $T(n) = 2T(n/2) + n - 1$
- Master method:
 - $a=2, b=2, f(n)=n, \log_2 2=1$

if $f(n) < n^{\log_b a}$ then $T(n) = O(n^{\log_b a})$
 if $f(n) = n^{\log_b a}$ then $T(n) = O(n^{\log_b a} \log n)$
 if $f(n) > n^{\log_b a}$ then $T(n) = O(f(n))$

It does not require extra memory, thus the memory complexity is constant: $O(1)$.



```

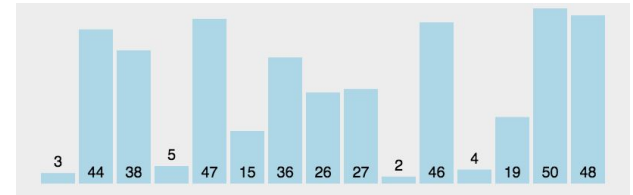
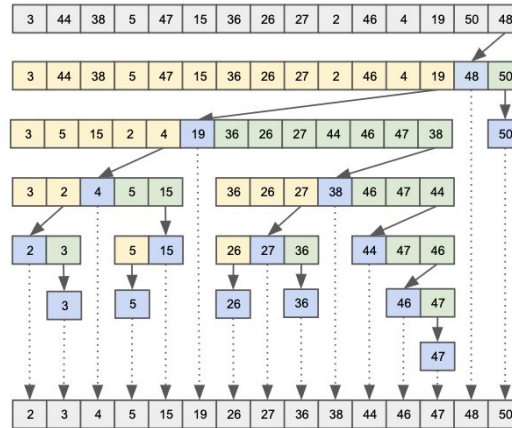
1  def lomuto(A, left, right):
2      p = A[right]
3      i = left
4      for j in range(left, right):
5          if A[j] < p:
6              A[i], A[j] = A[j], A[i]
7              i += 1
8      A[i], A[right] = A[right], A[i]
9      return i
10
11 def quicksort(A, left, right):
12     if (left < right):
13         mid = lomuto(A, left, right)
14         quicksort(A, left, mid-1)
15         quicksort(A, mid+1, right)
    
```



Example 4 - Quicksort

This Quicksort implementation performs the sort calling recursively a Lomuto partition until you have a single element part.

Other implementations are possible (see the video below, where the two indices i and j are used in a different way.



```
1 def lomuto(A, left, right):  
2     p = A[right]  
3     i = left  
4     for j in range(left, right):  
5         if A[j] < p:  
6             A[i], A[j] = A[j], A[i]  
7             i += 1  
8     A[i], A[right] = A[right], A[i]  
9     return i  
10  
11 def quicksort(A, left, right):  
12     if (left < right):  
13         mid = lomuto(A, left, right)  
14         quicksort(A, left, mid-1)  
15         quicksort(A, mid+1, right)
```

Video (an alternative implementation for the partition algorithm): [Quicksort In Python Explained](#).



This Week's tasks

- In-class Exercise E#7
- Coding Project P#7
- Quiz Q#7

Tasks

- Fill the worksheet as required.
- Develop 2 Python programs:
 - number of digits in a binary expansion;
 - sum of squares of positive Integers.
- Quiz #7 about this week topics.

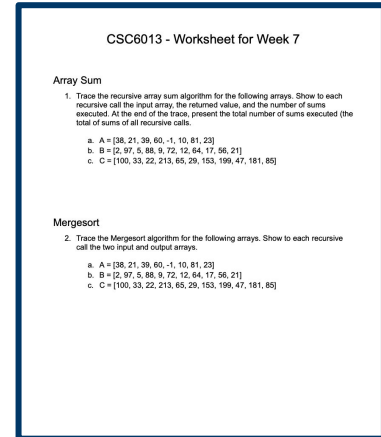


In-class Exercise - E#7

You have to submit a **.pdf** file with your answers. Feel free to type it or hand write it, but you have to submit a single **.pdf** with your answers.

Download the pdf ([link here also](#)) and perform the following:

- (1) Trace the **recursive array sum** algorithm for the following arrays. Show to each recursive call the input array, the returned value, and the number of sums executed. At the end of the trace, present the total number of sums executed (the total of sums of all recursive calls).
 - A = [38, 21, 39, 60, -1, 10, 81, 23]
 - B = [2, 97, 5, 88, 9, 72, 12, 64, 17, 56, 21]
 - C = [100, 33, 22, 213, 65, 29, 153, 199, 47, 181, 85]
- (2) Trace the **Mergesort** algorithm for the following arrays. Show to each recursive call the two input and output arrays.
 - A = [38, 21, 39, 60, -1, 10, 81, 23]
 - B = [2, 97, 5, 88, 9, 72, 12, 64, 17, 56, 21]
 - C = [100, 33, 22, 213, 65, 29, 153, 199, 47, 181, 85]



MERRIMACK COLLEGE

This task counts towards the In-class Exercises grade and the deadline is This Friday.

Seventh Coding Project - P#7

Develop one Python program to perform the Quicksort, but instead of sorting the elements in ascending order (from the smallest to the largest), the elements are sorted in the decrescent order (from the larger to the smallest). Your program also have to compute and print at the end the number of swaps performed and the number of recursive calls.

Test your program over the following arrays:

- A = [38, 21, 39, 60, -1, 10, 81, 23]
- B = [2, 97, 5, 88, 9, 72, 12, 64, 17, 56, 21]
- C = [100, 33, 22, 213, 65, 29, 153, 199, 47, 181, 85]

You have to submit the code (**.py** file) of your algorithm, plus the **.pdf** of the output values of total number swaps and recursive call for the required tests.



MERRIMACK COLLEGE

This assignment counts towards the Projects grade and the deadline is Next Monday.

Seventh Quiz - Q#7

- The seventh quiz in this course covers the topics of Week 7;
- The quiz will be available this Friday, and it is composed by 10 questions;
- The quiz should be taken on Canvas (Module 7), and it is not a timed quiz:
 - You can take as long as you want to answer it (a quiz taken in less than one hour is usually a too short time);
- The quiz is open book, open notes, and you can even use any language Interpreter to answer it;
- Yet, the quiz is evaluated and you are allowed to submit it only once.

Your task:

- Go to Canvas, answer the quiz and submit it within the deadline.



MERRIMACK COLLEGE

This quiz counts towards the Quizzes grade and the deadline is Next Monday.

” **Welcome to CSC 6013**

- **Do In-class Exercise E#7 until Friday;**
- **Do Quiz Q#7 (available Friday) until next Monday;**
- **Do Coding Project P#7 until next Monday.**

**Next Week - Transform-and-conquer algorithms
... and the final exam!**



MERRIMACK COLLEGE