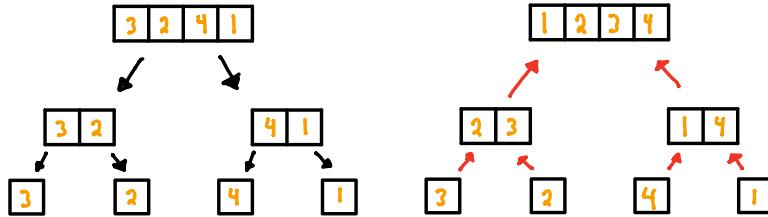


Merge Sort

- Chapter 12
- Each node on the tree (T) represents a recursive invocation (or call) of the merge sort algorithm.

 $\left\{ \begin{array}{l} \text{recurrence} \\ \text{or} \\ \text{repetition} \end{array} \right.$

 $\left\{ \begin{array}{l} \text{function calls itself} \end{array} \right.$



a.) Input sequences processed at each Node of T b.) Output sequences generated at each node of T

- Split object(list) into two sequences and sort each one, then merge them at the end.
- Dividing the sequence by $\lceil \frac{n}{2} \rceil$
- Very Efficient for large data sets.
- Divide and Conquer Algorithm

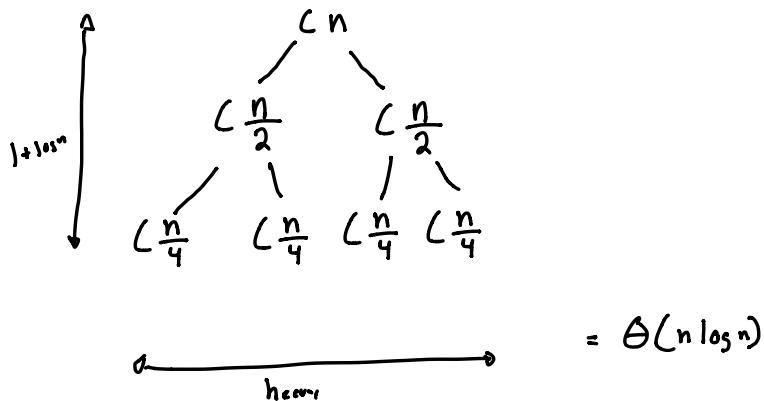
1. **Divide:** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S ; that is, S_1 contains the first $\lceil n/2 \rceil$ elements of S , and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
2. **Conquer:** Recursively sort sequences S_1 and S_2 .
3. **Combine:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

The running time of Merge Sort:

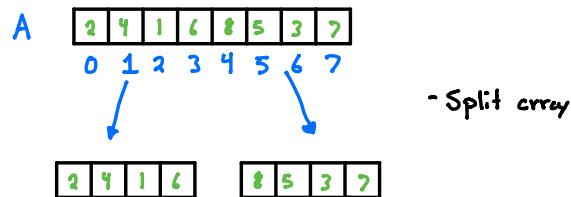
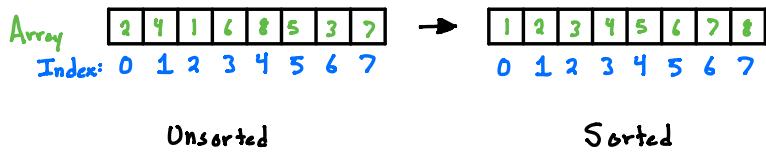
Proposition 12.1: The merge-sort tree associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.

~ Add more here

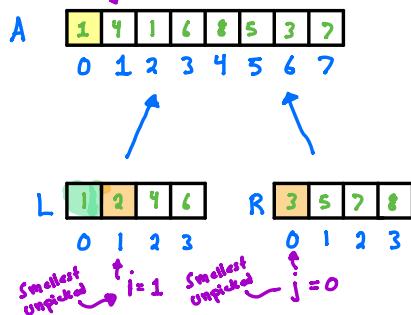
$$\begin{aligned}\text{Complexity: } T(n) &= \underbrace{C_1}_{\text{Divide}} + \underbrace{2T\left(\frac{n}{2}\right)}_{\text{Recursion}} + \underbrace{C \cdot n}_{\text{merge}} \\ &= 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)\end{aligned}$$



Merge Sort



K=1 ~ marks the position that needs to be filled in A.



~ Sorted (merge these to the original array)

Example of the iteration:

$k=1$

1	4	1	6	5	3	7
0	1	2	3	4	5	6

A

Comparing 2 and 3

$i=1$ $j=0$

L R

1	2	4	6
0	1	2	3

3	5	7	8
0	1	2	3

$i=K[i]$

$i=1$ $j=0$

Now '2' is in order

```

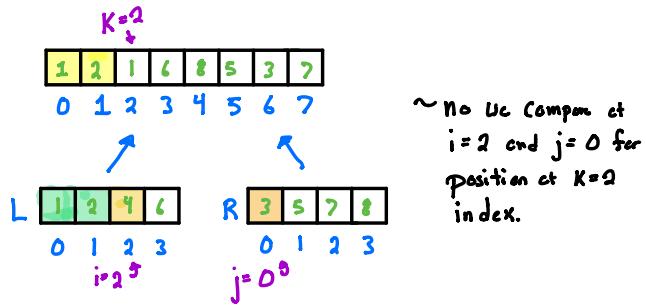
def Merge(L,R,A):
    nL ← Length(L)      List Length
    nR ← Length(R)      List Length
    i = j = k = 0

    While (i < nL and j < nR):
        If (L[i] <= R[j]):   ~ Comparing smallest unpicked in L with the smallest unpicked in R.
            A[k] = L[i]
            i = i + 1
        Else:
            A[k] = R[j]
            j = j + 1

    K = K+1      ← moving to next index in our sorted list

```

next iteration:



~ No we compare at
i=2 and j=0 for
position at K=2
index.

What happen when either i or j index is finished first? (are conditional while($i < \#nL$ and $j < \#nR$): is false)

`def Merge(L, R, A):`

Left-length = `len(Left)`

Right-length = `len(Right)`

$i = j = k = 0$

`while (i < Left-length and j < Right-length):`

`if (L[i] <= R[j]):`

`A[k] = L[i]`
 `i = i + 1`

`else:`
 `A[k] = R[j]`
 `j = j + 1`

`k = k + 1`

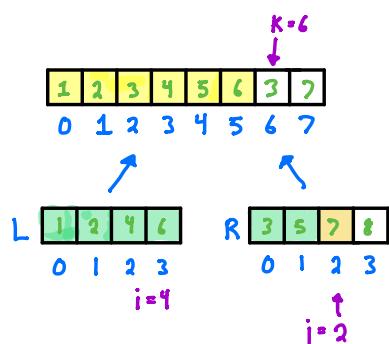
`while (i < #nL):`

`A[k] = L[i]`
 `i = i + 1`
 `k = k + 1`

`while (j < #nR):`

`A[k] = R[j]`
 `j = j + 1`
 `k = k + 1`

Once were out of
this while loop only
one while loop at
the bottom will execute



Python Code for Merge function:

```
def Merge (Left, Right, Array):
    Left_length = len(Left)
    Right_length = len(Right)
    i = j = k = 0
    while (i < Left_length and j < Right_length):
        if Left[i] <= Right[j]:
            Array[k] = Left[i]
            i = i + 1
        else:
            Array[k] = Right[j]
            j = j + 1
        k = k + 1
    # in case one of the letters runs out before the other
    while i < Left_length:
        Array[k] = Left[i]
        i = i + 1
        k = k + 1
    while j < Right_length:
        Array[k] = Right[j]
        j = j + 1
        k = k + 1
    return Array

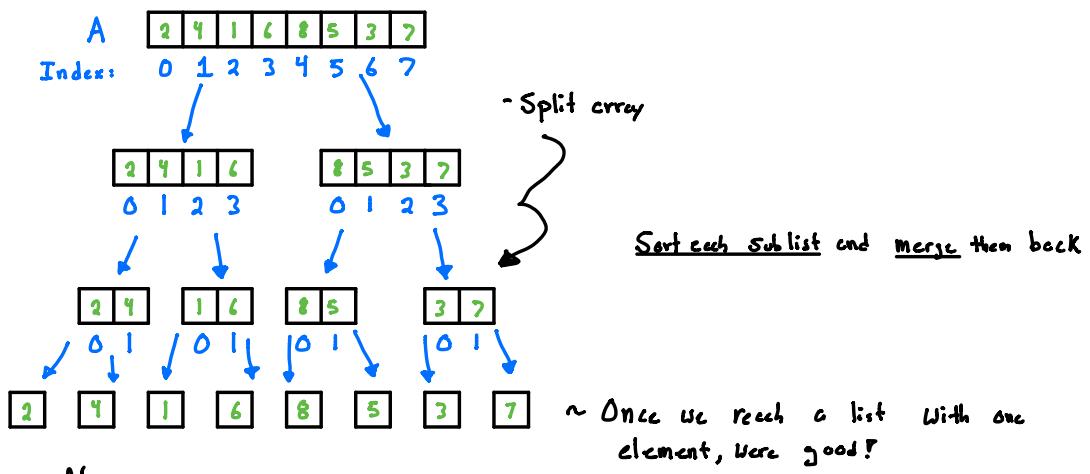
# Example:
a = [1,2,1,6,8,5,3,7]
l = [1,2,4,6]
r = [3,5,7,8]
print(Merge(l,r,a))

# Results
[1, 2, 3, 4, 5, 6, 7, 8]
```

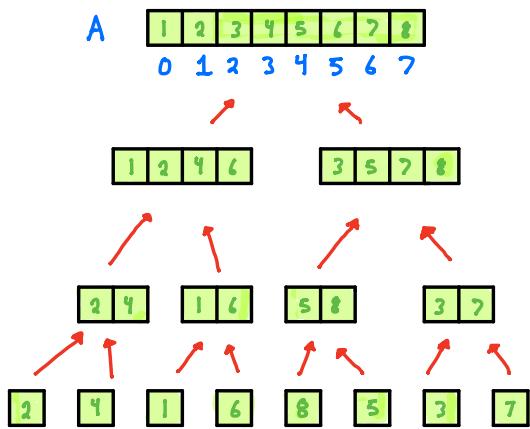
✓

*~ Takes in an unsorted Array,
Left and Right sides of
the Unsorted Array and
Sorts them and returns
a Sorted array.*

Problem Solved



Now we can merge them.

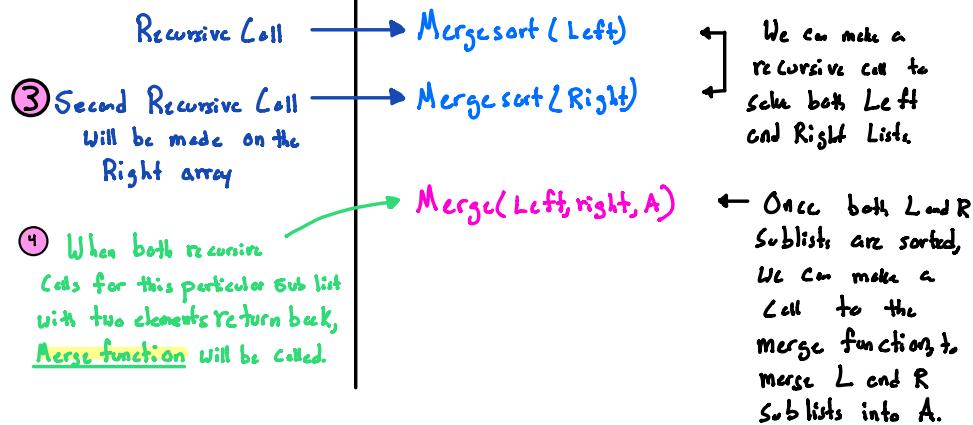
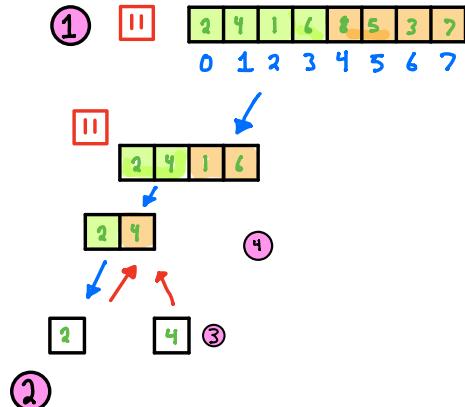


Merge Sort Function

Sudo Code:

```
def MergeSort(A):  
    An array:  
    n = length(A)  
    if n < 2:      ← Base Condition  
        return  
    mid = n // 2  
    Left = [0: mid]  
    Right = [mid:]  
    ↗ 2 arrays split original array in L and R halves.  
    MergeSort(Left) ] ↗ We can make a recursive call to solve both Left and Right Lists.  
    MergeSort(Right)  
  
Merge(Left, right, A) ← Once both L and R Sublists are sorted,  
                      we can make a call to the merge function to merge L and R Sublists into A.
```

Step by Step



in ① our function call of the current array is paused () and the machine says hey let me go and finish this particular function call and then ill come back to you / we keep dividing the array by $n/2$ until our base condition is satisfied ($n < 2$).

(): Step of execution of the function called with these arrays as arguments are paused.

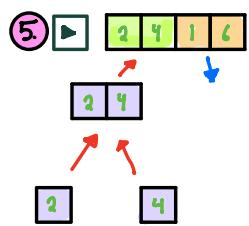
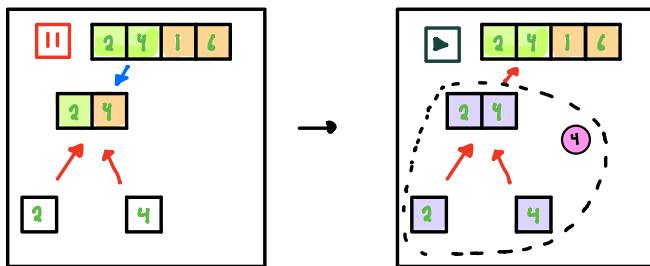
↓: Recursive Call

② This array with one element base condition will be true, so this call will exit

```

def MergeSort(A):
  ↗ An array:
  n = length(A) ~ length of Array.
  if n < 2: ← Base Condition or the
    return exit Condition from the
    mid = n // 2 ← Divide recursion.
  Left = [0: mid] ← 2 arrays split original
  Right = [mid: n] array in L and R halves.
  ↗ We can make a recursive call to solve both Left and Right Lists.
  ↗ Merge(Left, right, A) ← Once both L and R Sublists are sorted, we can make a call to the Merge function to merge L and R Sublists into A.
  
```

- Once $\text{Merge}(\text{Left}, \text{Right}, A)$ is finished the control will return back to the execution of the previous subset $[2, 4, 1, 6]$. (▶)



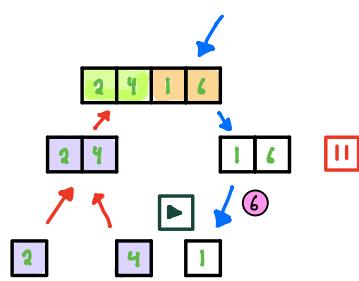
~ Now this sublist will have its Control returned back

- ⑤ : Now this guy will make the second Merge Sort call. It will call Mergesort(right), which will pass in the right array (which has $[1, 6]$), and then we're at the top of our def MergeSort(A) function.

~~BIGGEST TAKEAWAY~~

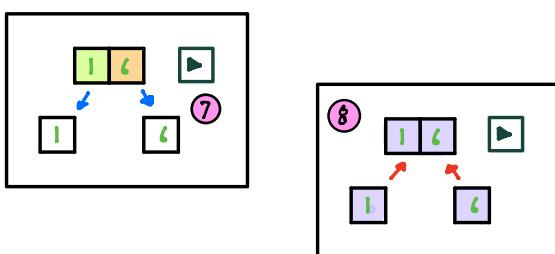
Original Array:

II [2 4 1 6 8 5 3 7]
 0 1 2 3 4 5 6 7



- ⑥ : Now, the array [1, 6] is II and we divide the array, by making another recursive call (Mergesort(Left)).

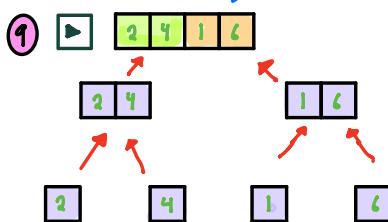
- ⑦ Since the single element is < 2 our Base Condition will make the program to the next line, which will execute the recursive call (Mergesort(Right)) on the right side of the array, which is 6.



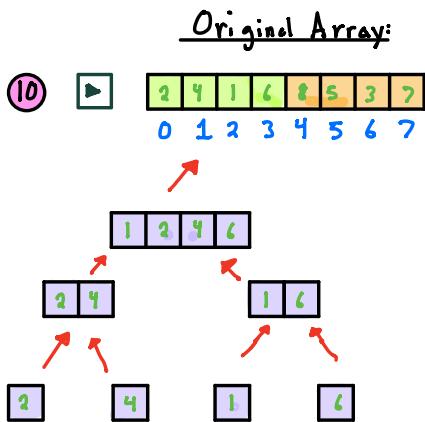
- ⑧ When both recursive calls for this particular sublist with two elements return back, Merge(Left, right, A) will be called.

Original Array:

II	2	4	1	6	8	5	3	7
	0	1	2	3	4	5	6	7

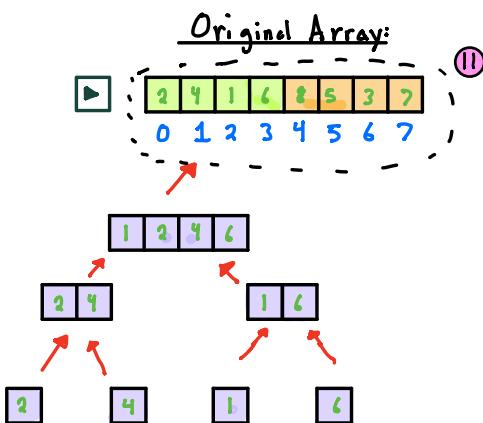


①: Then Control will return back to the array 2,4,1,6 and Merge(left,right,A) will be called



⑩ Once 1,2,4,6 will finish Merge(left,right,A) the Control Will return back to the function call corresponding to the Original array

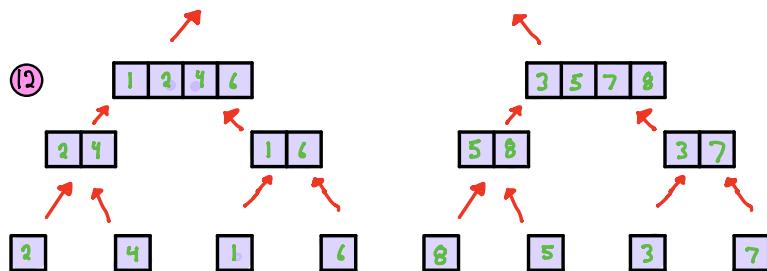
and then this guy



⑪ And then this guy will make another recursive call to do the right side of our Original array.

Original Array:

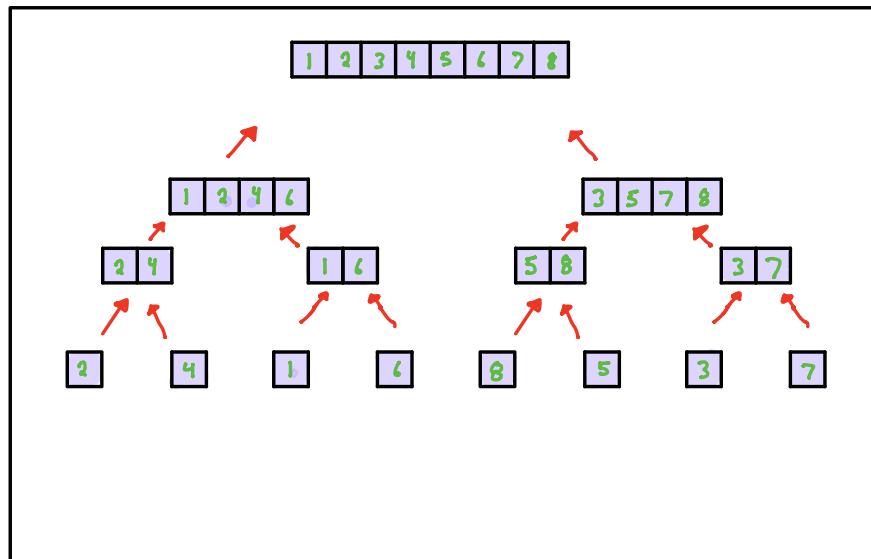
2 4 1 6 8 5 3 7



(2) Repeat Algorithm on other side!

~ Then `Merge(left,right,A)` will be called for [1, 4, 6] and [3, 5, 7]

- Boom its merged!



Sorted Array Using merge sort

Python Code for Merge-Sort Function

```
def Merge_Sort(Array):  
    '''Recursive function to sort an Array'''  
  
    n = len(Array)  
  
    if n < 2:  
  
        return  
    middle = n // 2  
  
    # have to make right and left subarrays  
    Left = Array[0:middle]  
  
    Right = Array[middle:]  
  
    #Recursive functions  
  
    Merge_Sort(Left)  
  
    Merge_Sort(Right)  
  
    # calling the merge function  
    Merge(Left, Right, Array)
```

Full Python Code For Merge Sort Algorithm

```
def Merge(Left, Right, Array):
    Left_length = len(Left)
    Right_length = len(Right)
    i = j = k = 0
    while (i < Left_length and j < Right_length):
        if Left[i] <= Right[j]:
            Array[k] = Left[i]
            i = i + 1
        else:
            Array[k] = Right[j]
            j = j + 1
        k = k + 1
    # in case one of the letters runs out before the other
    while i < Left_length:
        Array[k] = Left[i]
        i = i + 1
        k = k + 1
    while j < Right_length:
        Array[k] = Right[j]
        j = j + 1
        k = k + 1
    return Array

def Merge_Sort(Array):
    '''Recursive function to sort an Array'''
    n = len(Array)

    if n < 2:
        return
    middle = n // 2

    # have to make right and left subarrays
    Left = Array[0:middle]
    Right = Array[middle:]

    #Recursive functions
    Merge_Sort(Left)
    Merge_Sort(Right)

    # calling the merge function
    Merge(Left, Right, Array)

a = [2,4,1,6,8,5,3,7]
Merge_Sort(a)
print(a)
'Results: [1, 2, 3, 4, 5, 6, 7, 8]'
```

Sorted !