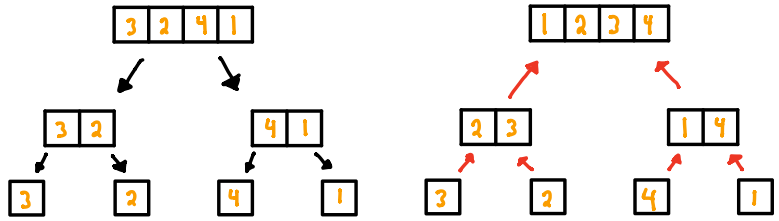
Merge Sort

- Chapter 12
- Each node on the tree (T) represents a recursive (recurrence or repetition) invocation (or call) of the merge sort algorithm.
  - recursive: function calls itself



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 $\lfloor \frac{n}{2} \rfloor$
- 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 $\lfloor n/2 \rfloor$ 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.

Analysis of Merge Sort

1) Divide and Conquer

2) Recursive

3) Stable

4) Not In-place

$\theta(n)$ Space Complexity

5) $\theta(n \log n)$ time - Complexity

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(\frac{n}{2}) + (c_2 + c_3)\cdot n + (c_1 + c_4) & \end{cases}$$

$$\Downarrow$$

$$2T(\frac{n}{2}) + c'n + c'', \quad \text{if } n > 1$$

$$\Downarrow$$

$$2T(\frac{n}{2}) + c'n, \quad \text{if } n > 1$$

For time Complexity we care about rate of growth of function for very high values of n, so we can simplify our calculation

Now we can solve this recurrence:

$$T(n) = 2T(\frac{n}{2}) + c'n, \quad \text{if } n > 1$$

$$= 2\left\{2T(\frac{n}{4}) + c'\cdot\frac{n}{2}\right\} + c'n$$

$$= 4T(\frac{n}{4}) + 2c'n$$

$$= 4\left\{2T(\frac{n}{8}) + c'\cdot\frac{n}{4}\right\} + 2c'n$$

$$= 8T(n/8) + 3c'n$$

$$= 16T(\frac{n}{16}) + 4c'n$$

$$= 2^k T(\frac{n}{2^k}) + k\cdot c'\cdot n$$

$\frac{n}{2^k} = 1 \rightarrow 2^k = n$

$\rightarrow k = \log_2 n$

$$= 2^{\log_2 n} T(1) + \log_2 n \cdot c'\cdot n$$

$$= nc + c'\cdot n\log n$$

- Now if we have to write in Big-Oh Notation we drop the lower order terms

$$= \theta(n \log n)$$

happens when base condition $n < 2$.

In Worst Case

c_1

Simple statements:
- Simple operation like assignments or comparisons.

Simple Statements:
- operate in constant time (c_1)

```
def Mergesort(A):
    n = length(A)
    if n < 2:                    <- Base Condition
        return
    mid = n//2
    Left = [0 : mid]             <- c2·n
    Right = [mid :]
    Mergesort(Left)              <- T(n/2)
    Mergesort(Right)             <- T(n/2)
    Merge(Left, right, A)        <- c3·n + c4
```

- performing this operation "n" times in every iteration.

Recursive Calls

Merge

$\Theta(n \log n)$ means:

$C_1 n \log n \leq T(n) \leq C_2 \cdot n \log n$, if $n \geq n_0$

$O(n \log n)$ means:

$T(n) \leq C \cdot n \log n$, if $n \geq n_0$

$\Theta(n)$ Space Complexity: A function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm

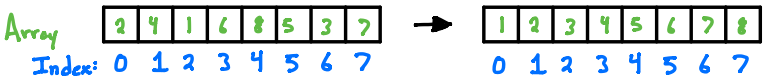
Time Complexity: A function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm

$\Theta(n)$ Space Complexity for our problem:

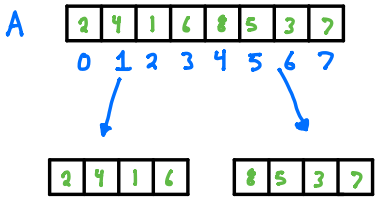
```
def Mergesort(A):
    n = length(A)
    if n < 2:
        return
    mid = n // 2
    Left = [0 : mid]
    Right = [mid :]
    Mergesort(Left)
    Mergesort(Right)
    Merge(Left, right, A)
```

The Extra memory we will consume will mostly be in the form of this Auxiliary Arrays

Merge Sort

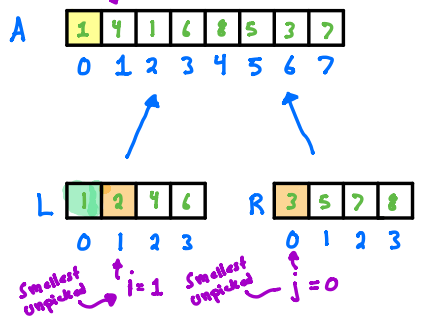


Unsorted Sorted



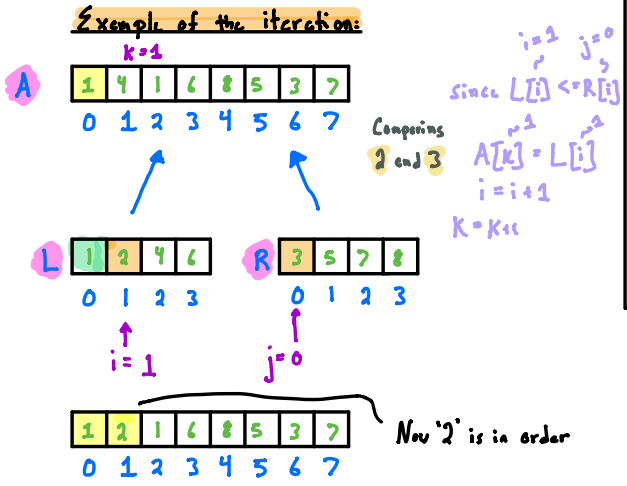
- Split array

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



~ Sorted (merge these to the original array)

Example of the iteration:

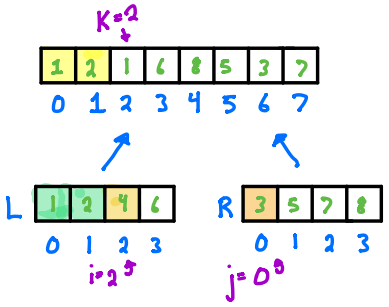


Now '2' is in order

Sudo Code:

```
def Merge(L,R,A):
    n L ← Length(L)      List Length
    n R ← 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:

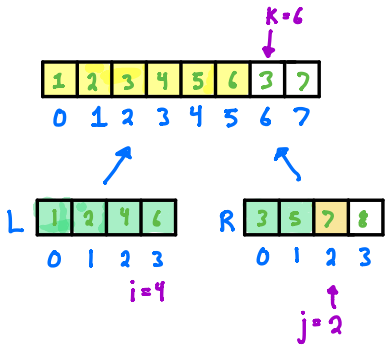


~ no we compare at i=2 and j=0 for position at K=2 index.

What happens when either i or j index is finished first? (are conditional `while (i < #nL and j < #R):` 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[i]
        j = j + 1
        k = k + 1
```

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



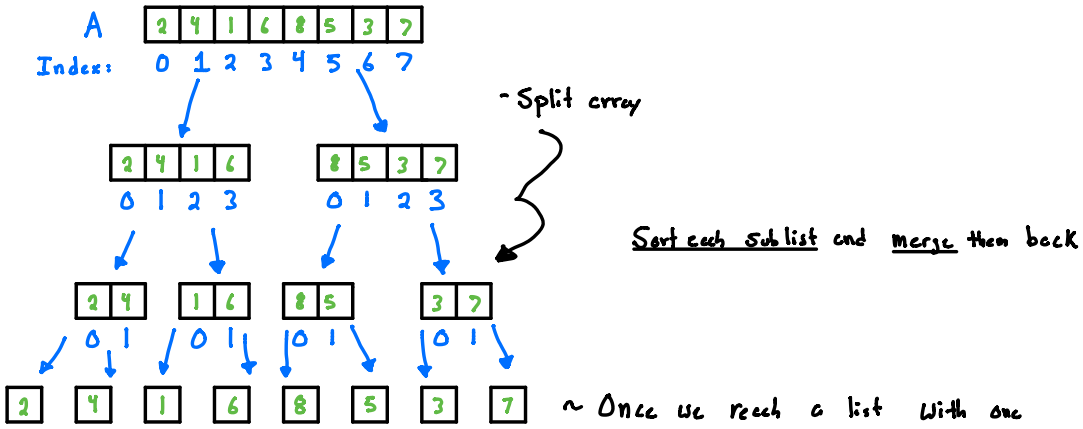
Python Code for Merge function:

```python
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]
```

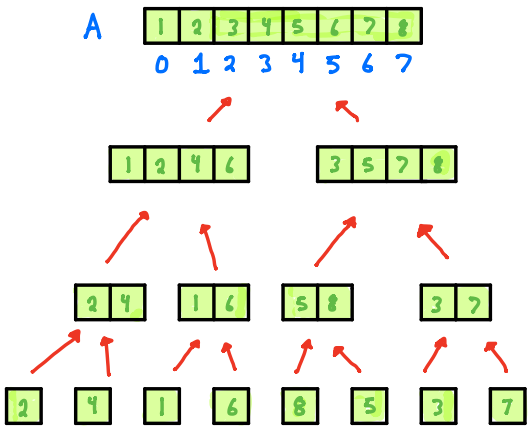
✓

~ 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.

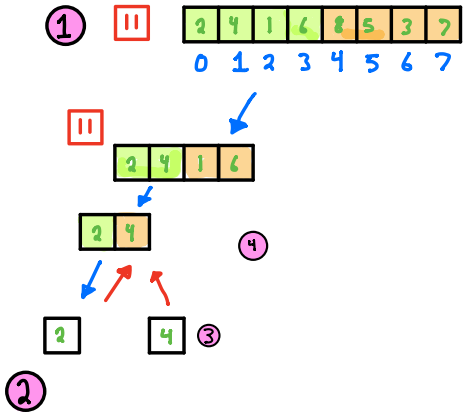


Merge Sort Function

Suedo Code:

```
def Mergesort(A):          # An array:
    n = length(A)
    if n < 2:              # <- Base Condition
        return
    mid = n // 2
    Left = [0 : mid]       # 2 arrays split original array in L and R halves.
    Right = [mid :]

    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




```
def Mergesort(A):          ← An array:
    n = length(A)          ~ length of Array.
    if n < 2:              ← Base Condition or the exit Condition from the recursion.
        return
    mid = n//2             ← Divide
    Left = [0 : mid]       ┐ 2 arrays split original array in L and R halves.
    Right = [mid : 0]      ┘
    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.
```

Recursive Call → Mergesort(Left)

3 Second Recursive Call → Mergesort(Right)
Will be made on the Right array

4 When both recursive Calls for this particular Sub list with two elements return back, Merge function will be called.

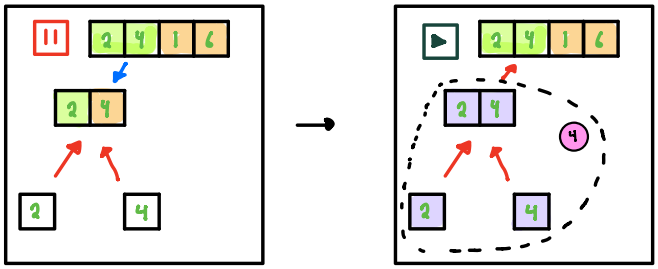
in 1 our function call of the current array is paused II and the median says hey, let me go and finish this particular function call and then ill come back to you. (We keep dividing the array by n/2 until our base condition is satisfied (n<2).

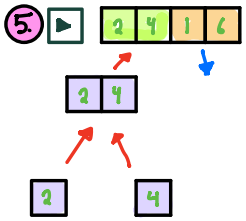
II : Stack of execution of the function called with these arrays as arguments are paused

↙ : recursive call

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

- Once Merge(Left, Right, A) is finished the control will return back to the execution of the previous sublist [2,4,1,6].



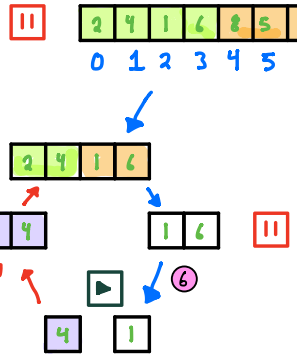


~ Now this sublist will have its control returned back

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

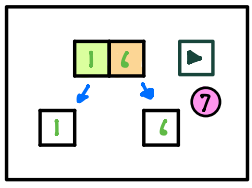
Biggest Takeaway

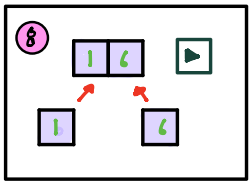
Original Array:



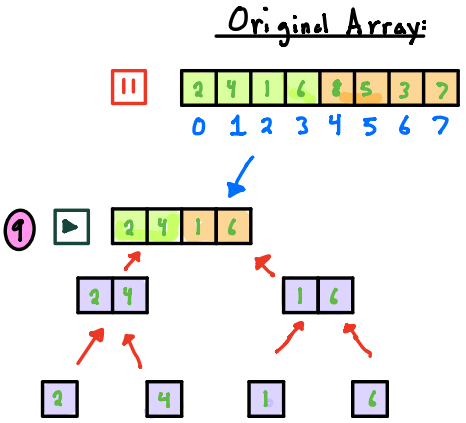
6 : Now, the array [1|6] is 11. and we dived the array, by making another recursive call (Mergesort(Left)).

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

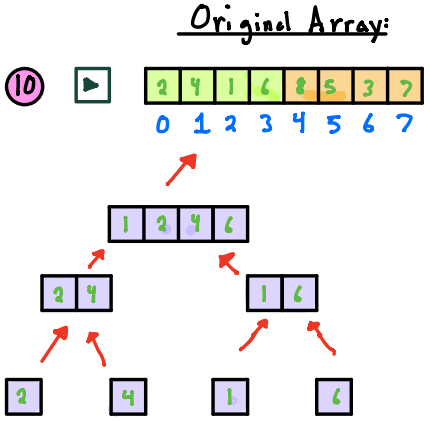




8 When both recursive calls for this particular sub list with two elements return back, Merge(Left, right, A) will be called.

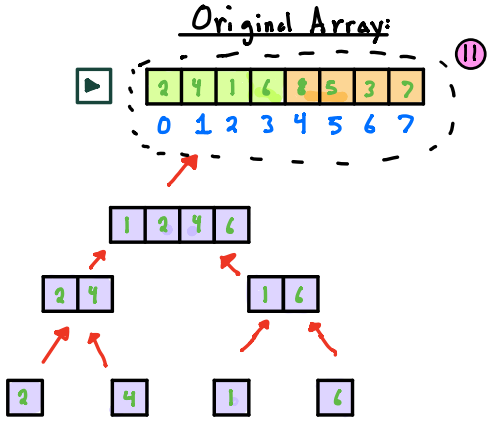


9: Then Control will return back to the array 2,4,1,6 and Merge(Left, right, A) will be called

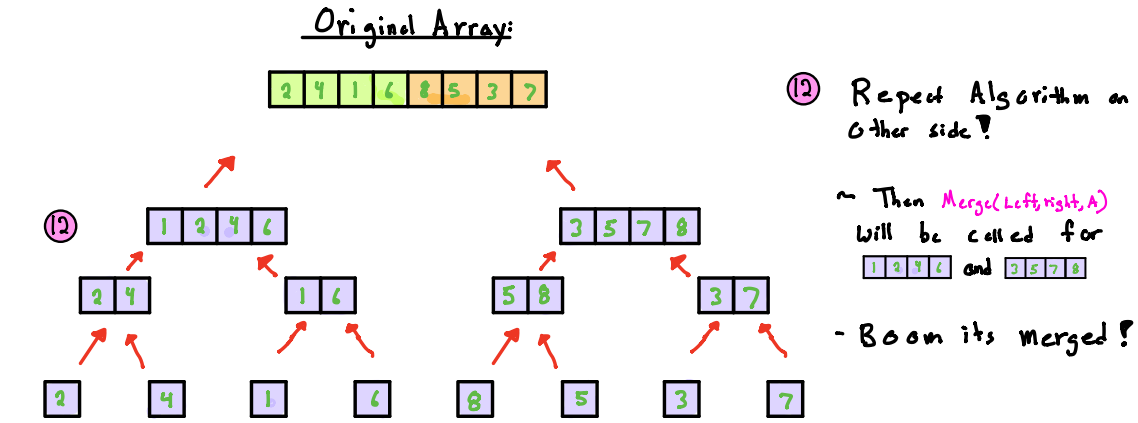


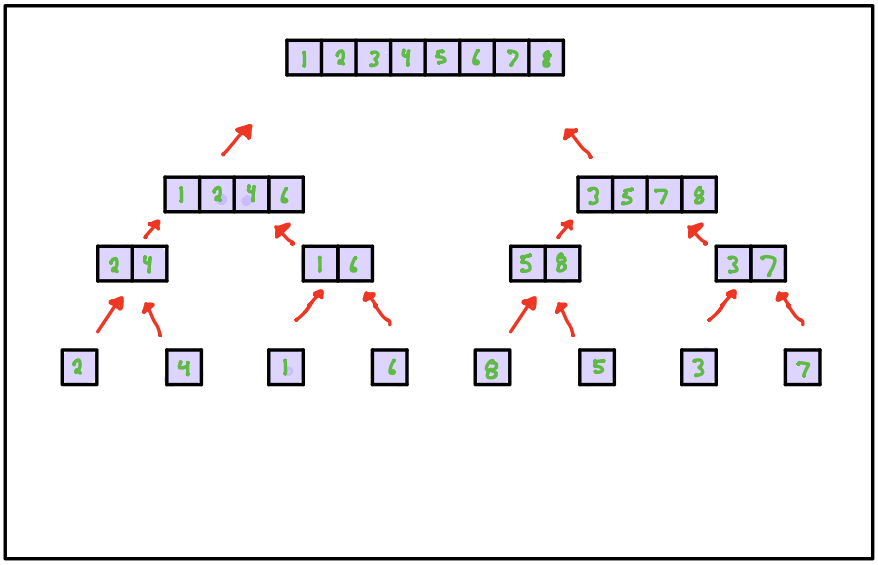
10 Once 1,2,4,6 will finish Merge(Left, right, A) the Control will return back to the function call corresponding to the original array

and then this guy



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





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]'
```