Arrays: Mergesort

Agenda

- Explore the Merge sort algorithm

- Understand the following aspects
  - Algorithm mechanism and pseudocode
  - Algorithm iterations on varying input
  - Algorithm time and space complexity

Merge sort

- Merge sort is a classic sort algorithm.
  - Optimal sort algorithm.
  - Used in system sort functions in major programming languages:
    - Java (sorting objects)
    - Python (stable sort)

- Algorithm scheme:
  1. Divide given array into two halves.
  2. Sort each half recursively.
  3. Merge the two sorted halves.

Merge sort

- Characteristics:
  - Merge sort is a comparison-based sort.
  - Regular Merge sort requires an auxiliary array for the merge operations.
  - In-place merge complicates algorithm, avoided.

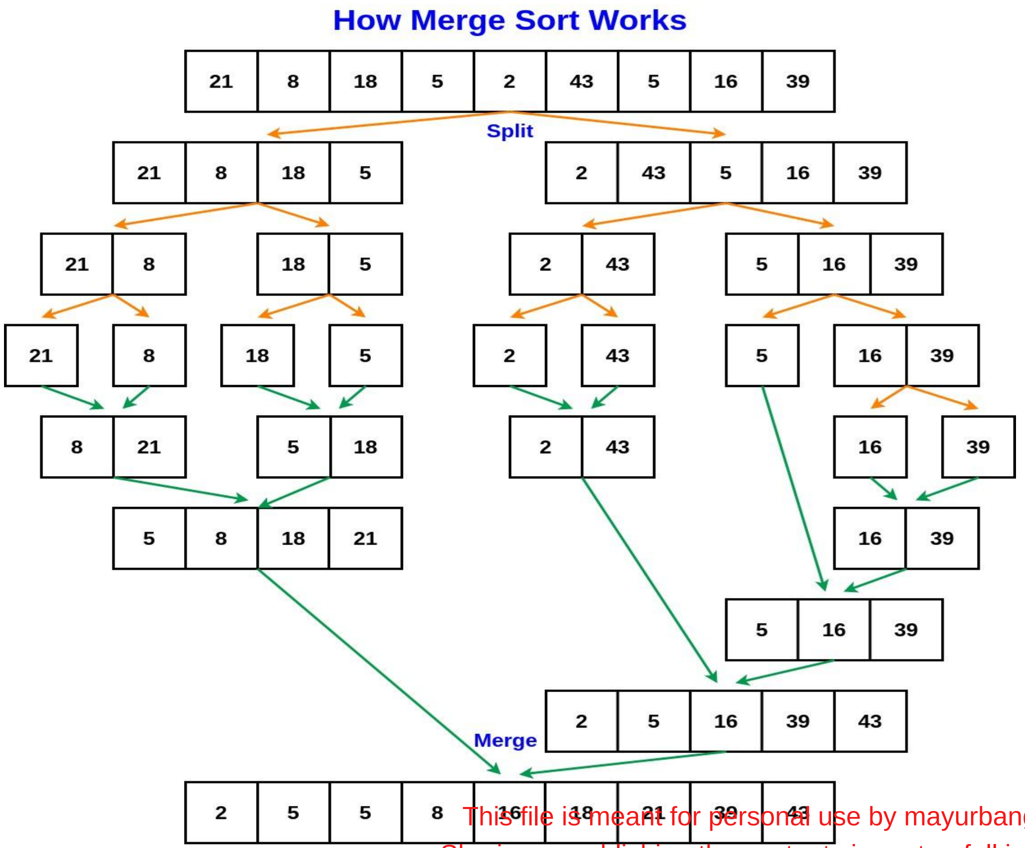
- Comparison operation:
  - Defined as required for the data type:
    - Numbers
    - Strings
    - Objects: by attributes

Merge sort

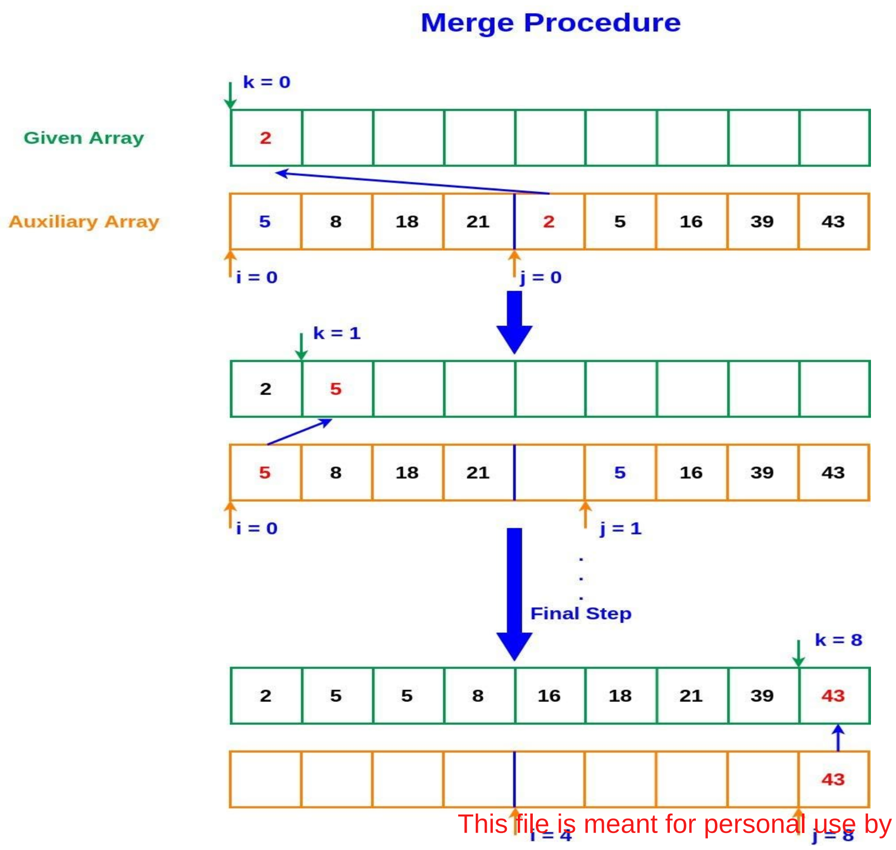
- Merge sort uses the divide-and-conquer technique.
    - Divide the array into two halves.
    - Recursively sort the two halves.

- Iterative version of Merge sort also possible.

Merge sort


How Merge Sort Works
21 8 18 5 2 43 5 16 39
Split
21 8 18 5
2 43 5 16 39
21 8
18 5
2 43
5 16 39
21
8
18
5
2
43
5
16 39
8 21
5 18
2 43
16
39
5 8 18 21
16 39
5 16 39
2 5 16 39 43
Merge
2 5 5 8 16 18 21 39 43

Merge procedure



Merge sort pseudocode

Code For Merge Sort

```
def merge(array, aux, lo, mid, hi):
     for k in range(lo, hi+1):


            i = lo

           j = mid+1

    for k in range(lo, hi+1):
         if i > mid:
            array[k] = aux[j]
             j+=1
         elif j > hi:
            array[k] = aux[i]
            i += 1
        elif aux[j] < aux[i]:
            array[k] = aux[j]
            j += 1
            else:

              array[k] = aux[i]
                  i += 1
```

```
def merge_sort(array, aux, lo, hi):
            if hi <= lo:
               return

      mid = lo + (hi - lo) // 2

   merge_sort(array, aux, lo, mid)
   merge_sort(array, aux, mid+1, hi)

   merge(array, aux, lo, mid, hi)



     def wrapper_sort(array):

             aux = []


         aux.extend(array)
merge_sort(array, aux, 0, len(array)-1)
```

Merge sort iterations

Merge Sort Iterations On Array A (With Auxiliary Array AUX)

Array Indexes

	0	1	2	3	4	5	6	7	8	
	21	8	18	5	2	43	5	16	39	(VIOLET) Initial Array A
merge(A, AUX, 0, 0, 1)	8	21	18	5	2	43	5	16	39	(GREY) No Movement
merge(A, AUX, 2, 2, 3)	8	21	5	18	2	43	5	16	39	
merge(A, AUX, 0, 1, 3)	5	8	18	21	2	43	5	16	39	
merge(A, AUX, 4, 4, 5)	5	8	18	21	2	43	5	16	39	

(RED)
Currently Merged
Sub-Array

	0	1	2	3	4	5	6	7	8
	5	8	18	21	2	43	5	16	39
merge(A, AUX, 6, 6, 6)	5	8	18	21	2	43	5	16	39
merge(A, AUX, 7, 7, 8)	5	8	18	21	2	43	5	16	39
merge(A, AUX, 4, 6, 8)	5	8	18	21	2	5	16	39	43
merge(A, AUX, 0, 4, 8)	2	5	5	8	16	18	21	39	43
(GREEN) Final Sorted Array	2	5	5	8	16	18	21	39	43

Merge sort iterations

Merge Sort Iterations On Pre-Sorted Array A
(With Auxiliary Array AUX)

Array Indexes

	0	1	2	3	4	5	6	7	8	
	2	5	5	8	16	18	21	39	43	(VIOLET) Initial Array A
merge(A, AUX, 0, 0, 1)	**2**	**5**	5	8	16	18	21	39	43	
merge(A, AUX, 2, 2, 3)	2	5	**5**	**8**	16	18	21	39	43	
merge(A, AUX, 0, 1, 3)	**2**	**5**	**5**	**8**	16	18	21	39	43	
merge(A, AUX, 4, 4, 5)	2	5	5	8	**16**	**18**	21	39	43	

	0	1	2	3	4	5	6	7	8
	2	5	5	8	**16**	**18**	21	39	43
merge(A, AUX, 6, 6, 6)	2	5	5	8	16	18	**21**	39	43
merge(A, AUX, 7, 7, 8)	2	5	5	8	16	18	21	**39**	**43**
merge(A, AUX, 4, 6, 8)	2	5	5	8	**16**	**18**	**21**	**39**	**43**
merge(A, AUX, 0, 4, 8)	**2**	**5**	**5**	**8**	**16**	**18**	**21**	**39**	**43**
(GREEN) Final Sorted Array	2	5	5	8	16	18	21	39	43

Merge sort iterations

Merge Sort Iterations On Reverse-Sorted Array A (With Auxiliary Array AUX)

Array Indexes

	0	1	2	3	4	5	6	7	8
(VIOLET) Initial Array A	43	39	21	18	16	8	5	5	2
merge(A, AUX, 0, 0, 1)	39	43	21	18	16	8	5	5	2
merge(A, AUX, 2, 2, 3)	39	43	18	21	16	8	5	5	2
merge(A, AUX, 0, 1, 3)	18	21	39	43	16	8	5	5	2
merge(A, AUX, 4, 4, 5)	18	21	39	43	8	16	5	5	2

	0	1	2	3	4	5	6	7	8
	18	21	39	43	8	16	5	5	2
merge(A, AUX, 6, 6, 6)	18	21	39	43	8	16	5	5	2
merge(A, AUX, 7, 7, 8)	18	21	39	43	8	16	5	2	5
merge(A, AUX, 4, 6, 8)	18	21	39	43	2	5	5	8	16
merge(A, AUX, 0, 4, 8)	2	5	5	8	16	18	21	39	43
(GREEN) Final Sorted Array	2	5	5	8	16	18	21	39	43

Merge sort: Complexity

- Analyzing time complexity:
  - Best case
  - Worst case
  - Average case

- Assume that the number of steps taken by the sort procedure (not merge), on an array of size N is C(N).

- Mergesort has 2 steps:
  - Sort 2 sub-arrays recursively: C(N/2) + C(N/2)
  - Merge 2 sub-arrays of size N/2: N

Merge sort: Complexity

- Analyzing time complexity:
  - Simplifying assumption: $N = 2^k$, for some k
  - From the previous slide,

$$C(N) = 2C(N/2) + N$$

Dividing both sides by N,

$$C(N) / N = C(N/2) / (N/2) + 1$$

$$C(N) / N = C(N/4) / (N/4) + 1 + 1 \quad \text{(Telescoping Property)}$$

....

$$C(N) / N = C(N/N) / (N/N) + k \qquad \text{(Remember, } N = 2^k\text{)}$$

$$C(N) / N = 0 + \log_2 N \qquad \text{(Sorting 1 element!!)}$$

$$C(N) = N\log_2 N$$

Merge sort: Complexity

- Analyzing time complexity:
  - The following can be demonstrated:
    - Worst case: $N\log_2 N$
    - Average case: $N\log_2 N$
    - Best case: $N\log_2 N$

Merge sort: Complexity

- Analyzing space complexity
  - Since the recursive sort and merge are executed throughout the sort:
    - Recursive sort: Constant space
    - Merge operation: Space proportional to N

  - Best case: space proportional to N
  - Worst case: space proportional to N
  - Average case: space proportional to N
  - The constant involved may change, from case to case.

Summary

We explored the Merge sort algorithm as follows:

- Algorithm mechanism and pseudocode
- Algorithm iterations on varying input
- Time and space complexity

Thank You