빅데이터 혁신공유대학

파이썬으로 배우는 데이터 구조

한동대학교 전산전자공학부

김영섭 교수

Data Structures in Python Chapter 5 - 2

- **Merge sort**
- Quick sort Algorithm
- Quick sort Analysis
- Empirical Analysis

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

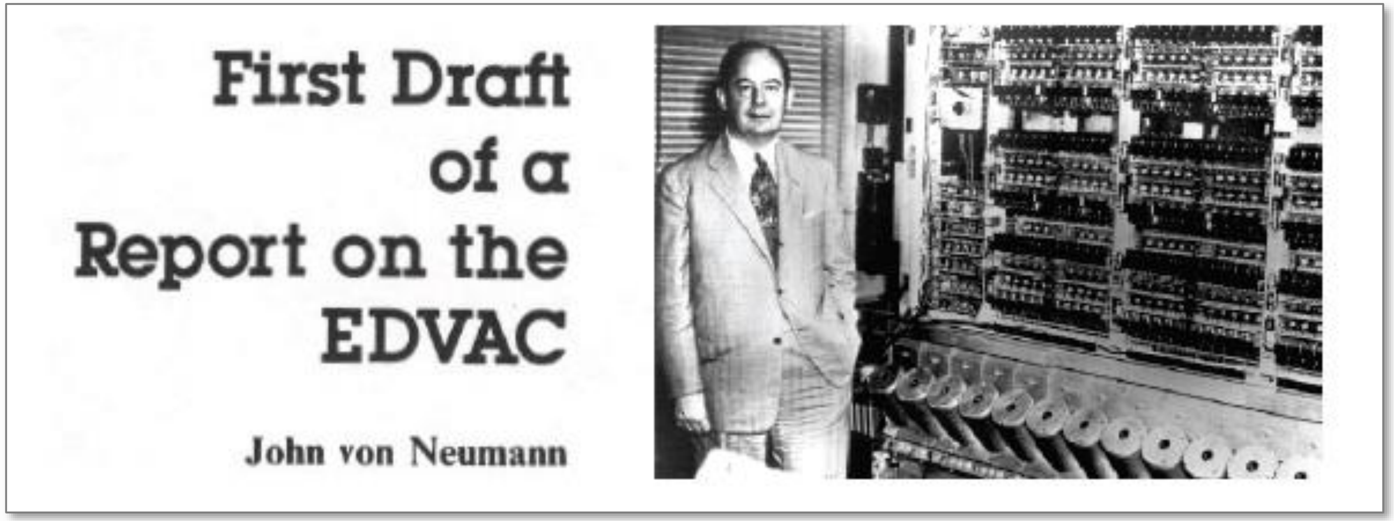
Agenda & Readings

- Agenda
  - Merge sort - $O(n \log n)$ sorting algorithm
- Reference:
  - Problem Solving with Algorithms and Data Structures
    - Chapter 5 Search, Sorting and Hashing
    - Analysis of merge sort
    - [알고리즘] 합병정렬

Merge sort

- Divide and conquer algorithm
  - We have already seen the **divide and conquer** algorithm using binary search on a sorted collection of items.
- Recursive or non-recursive(Iteration) implementation
- It was implemented on the first general purpose computer and is still running.

the first general purpose computer and its inventor,



Merge sort: Algorithm

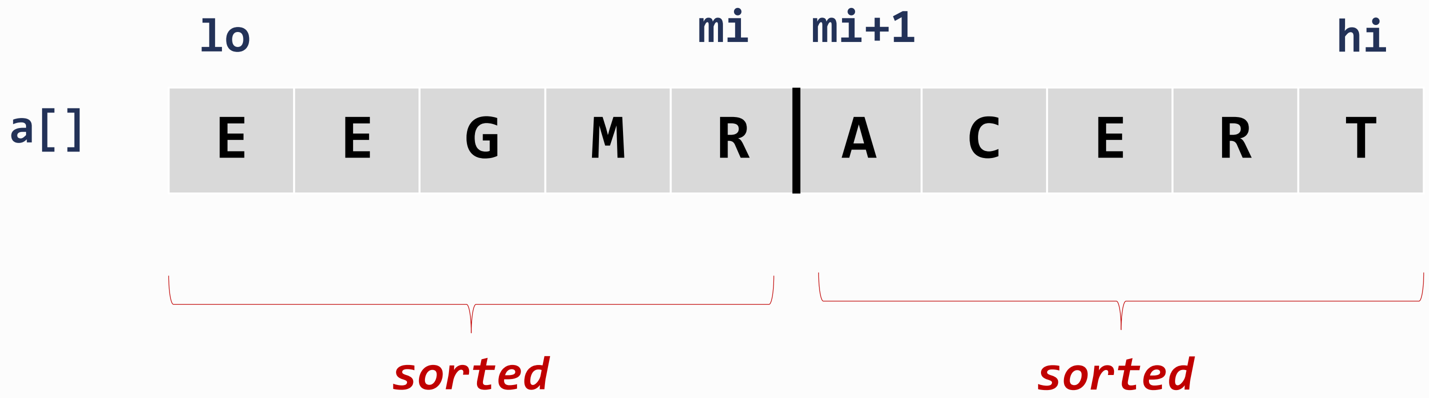
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

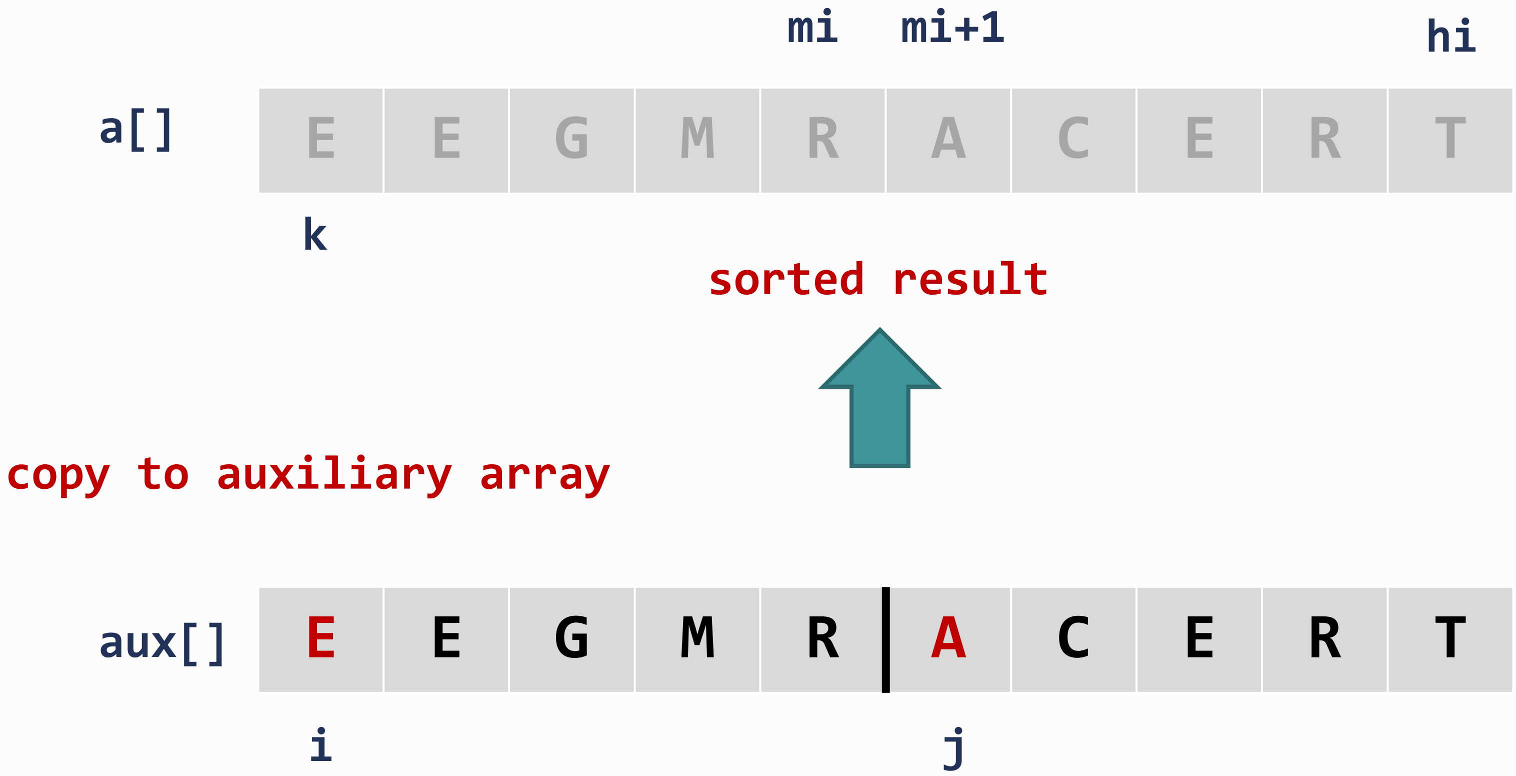
	lo				mi	mi+1				hi
a[]	E	E	G	M	R	A	C	E	R	T

copy to auxiliary array

aux[]	E	E	G	M	R	A	C	E	R	T

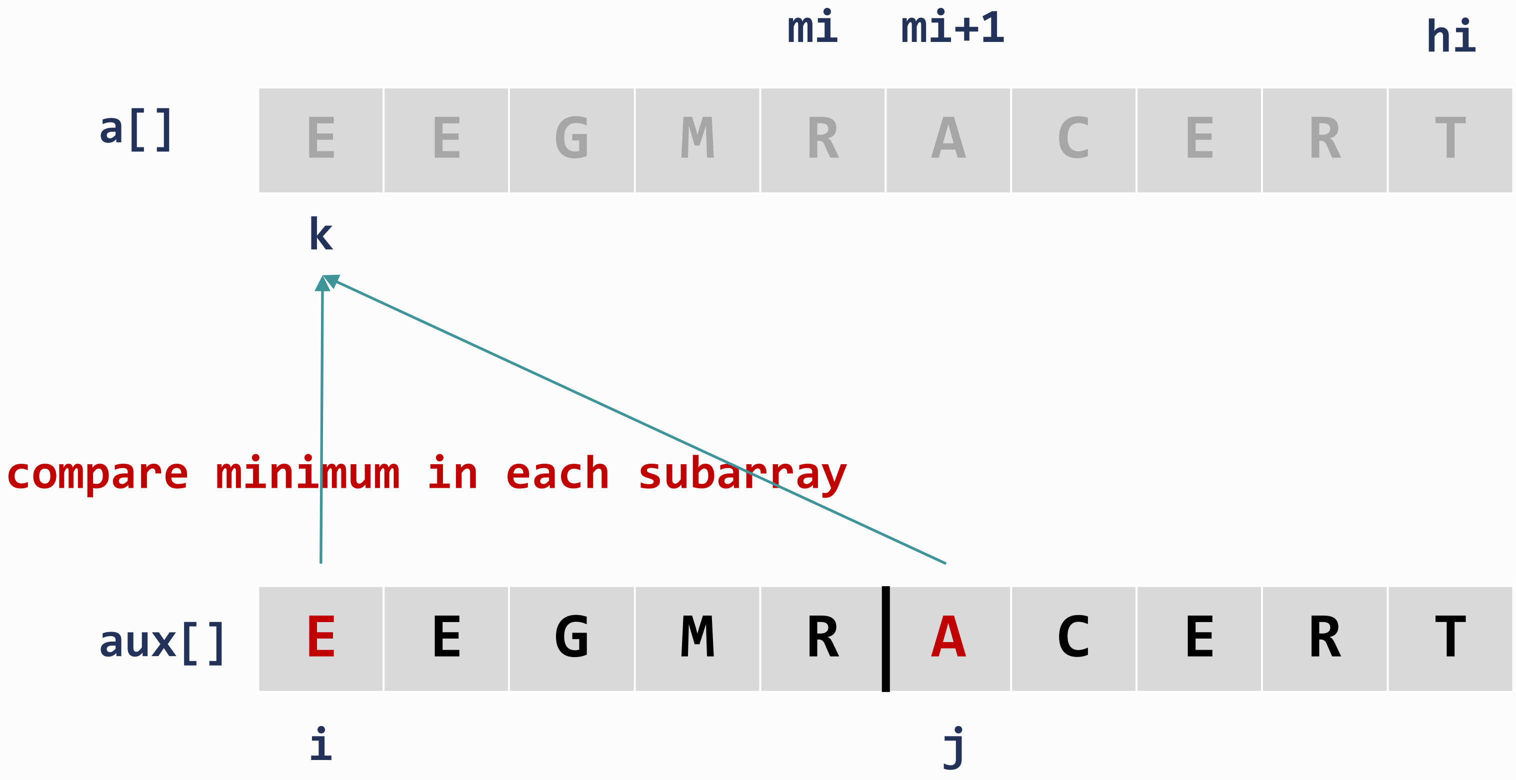
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



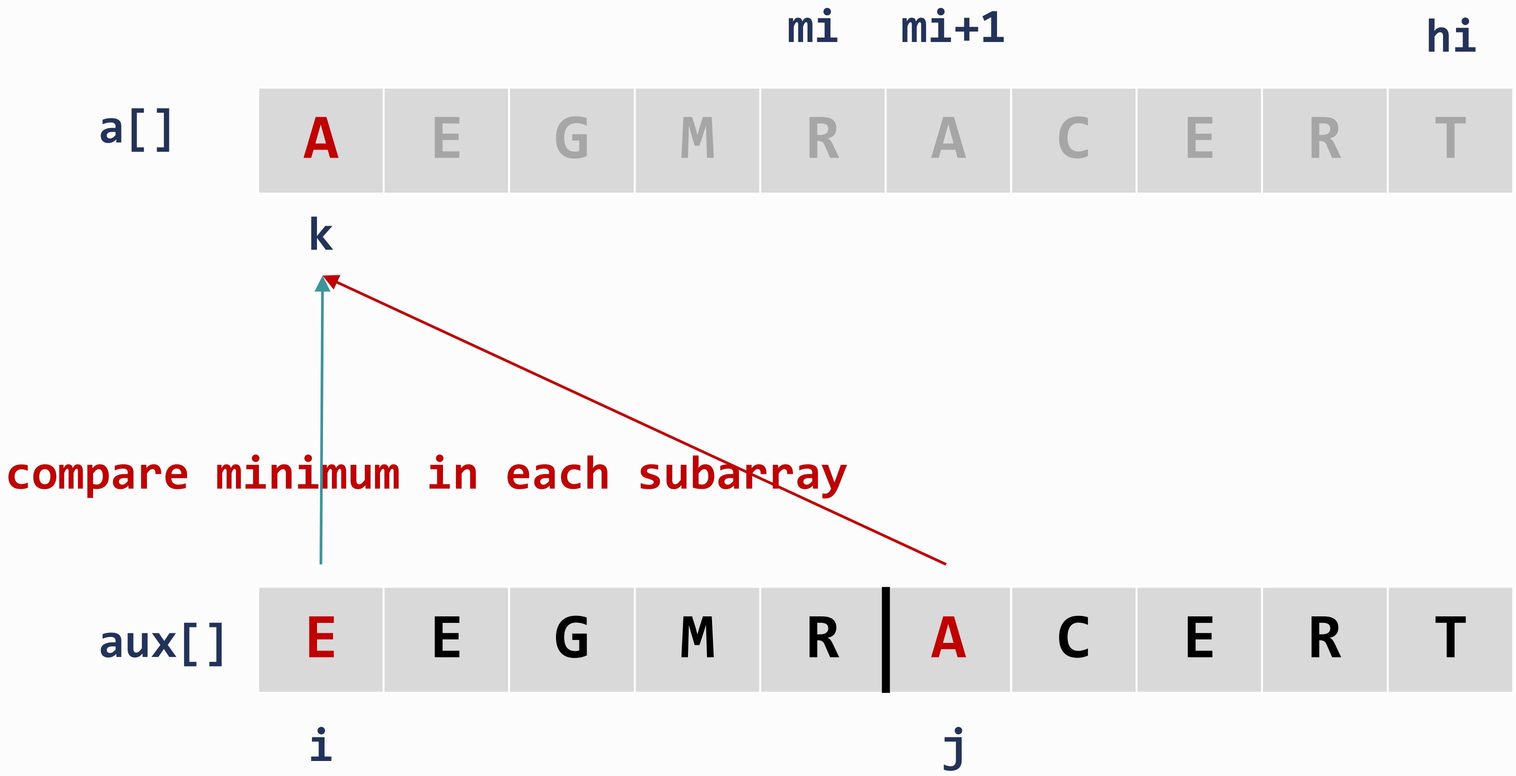
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



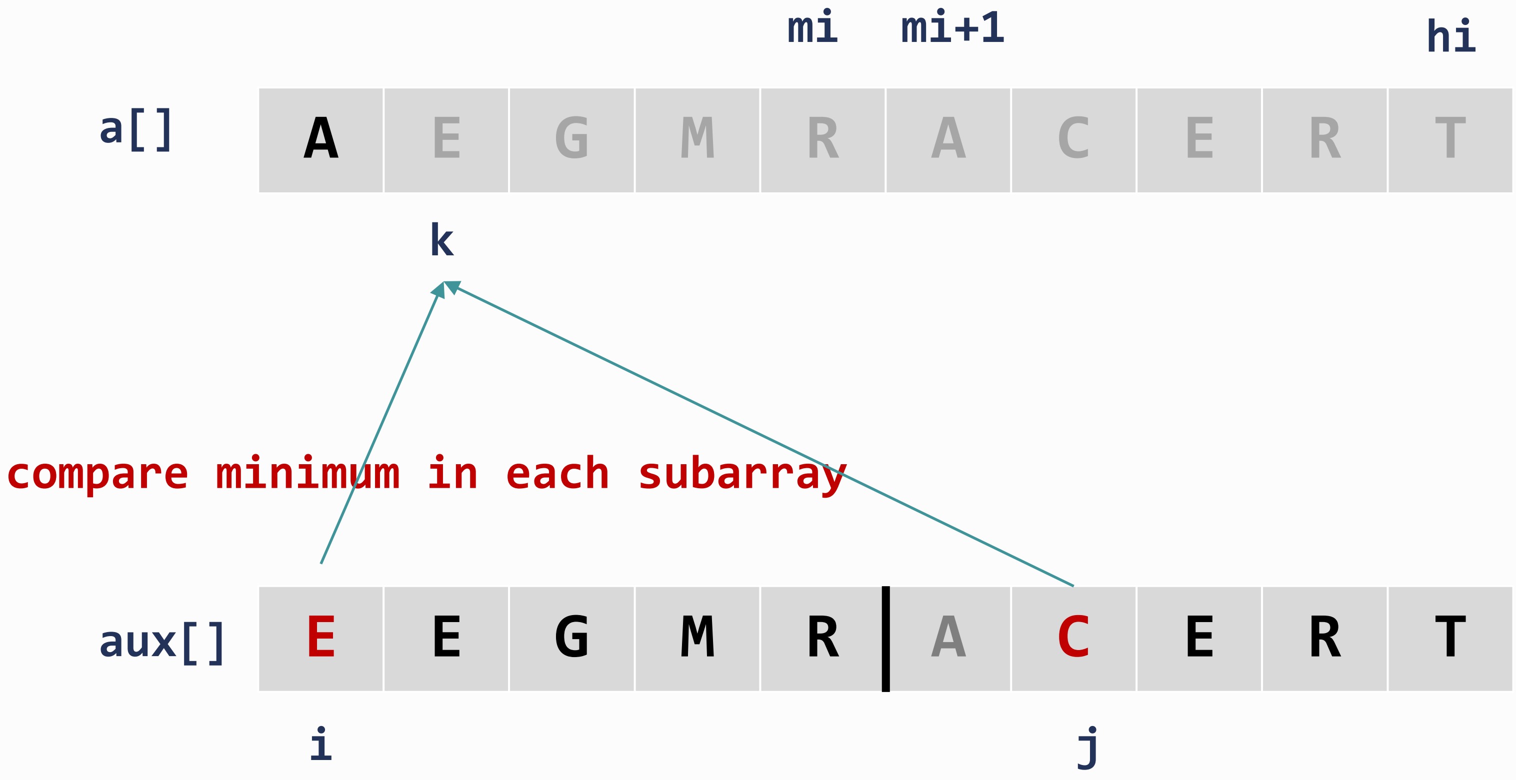
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



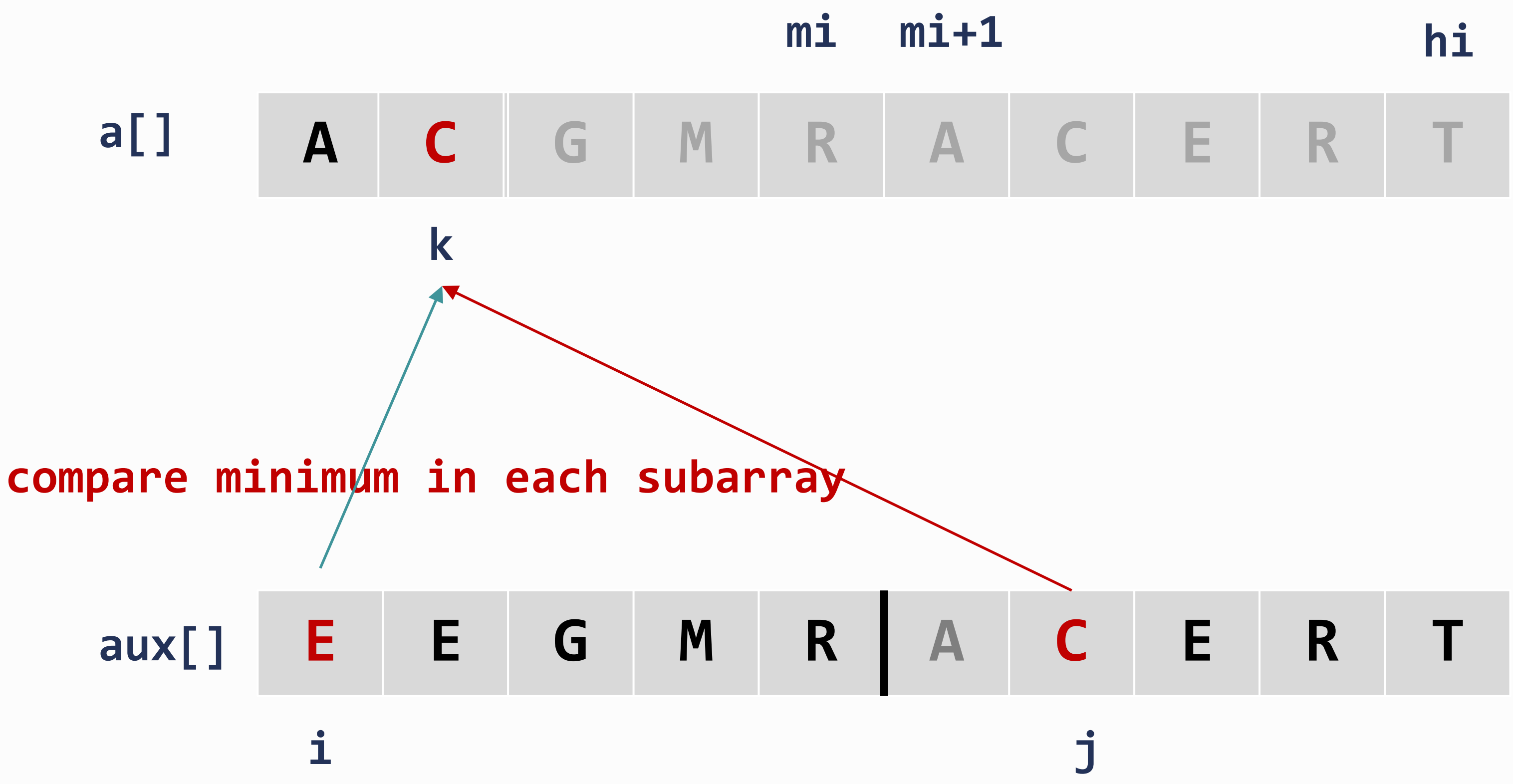
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



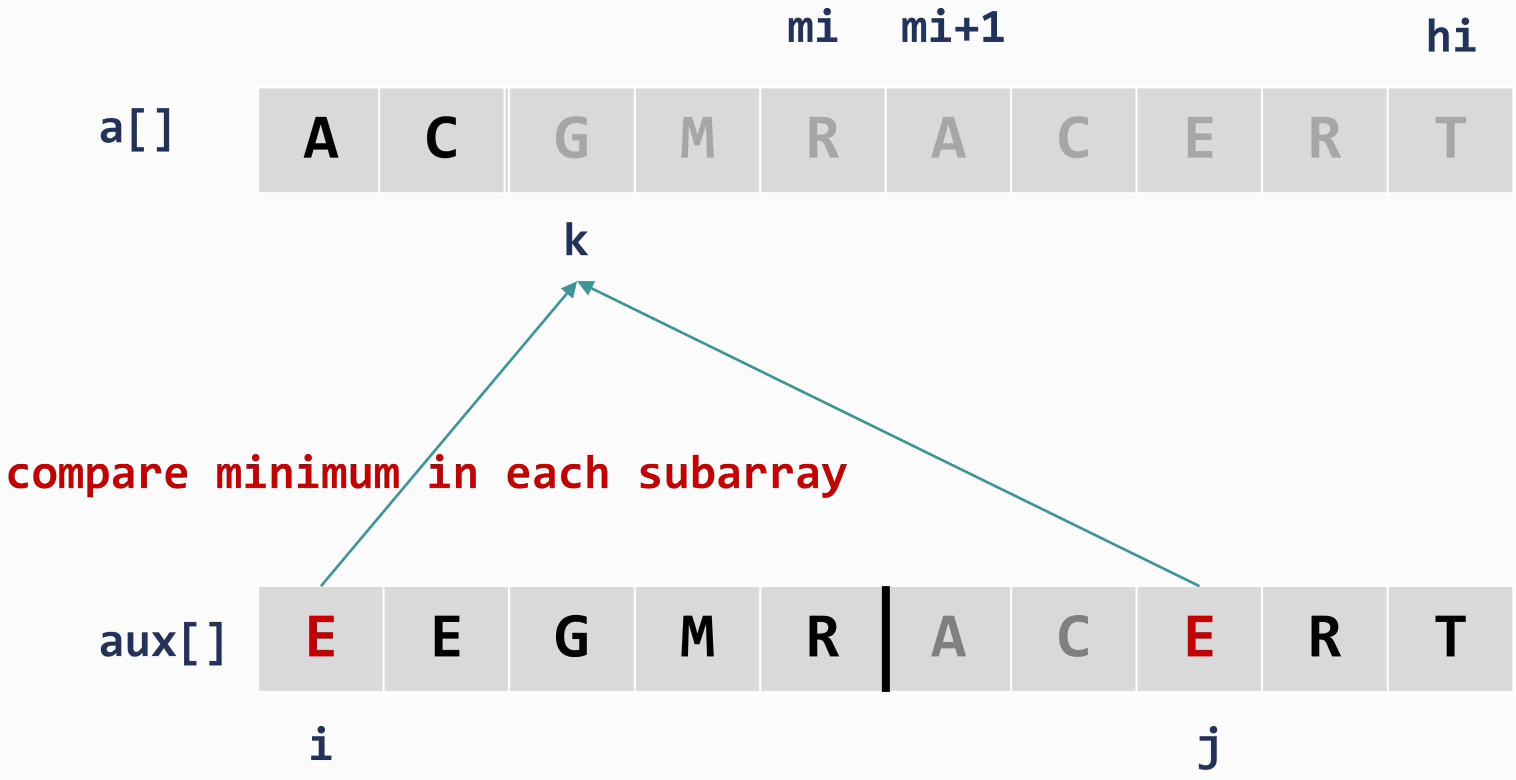
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



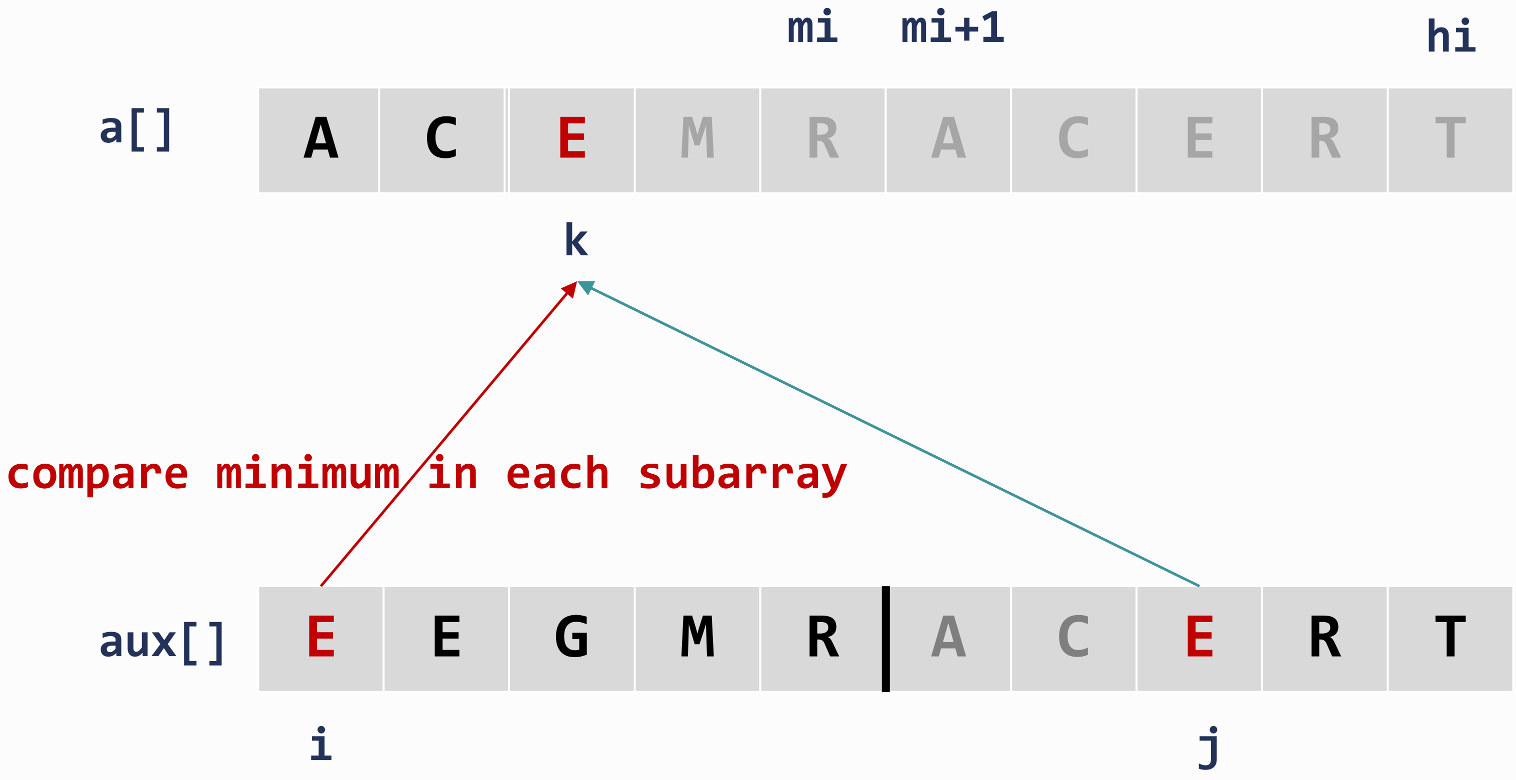
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



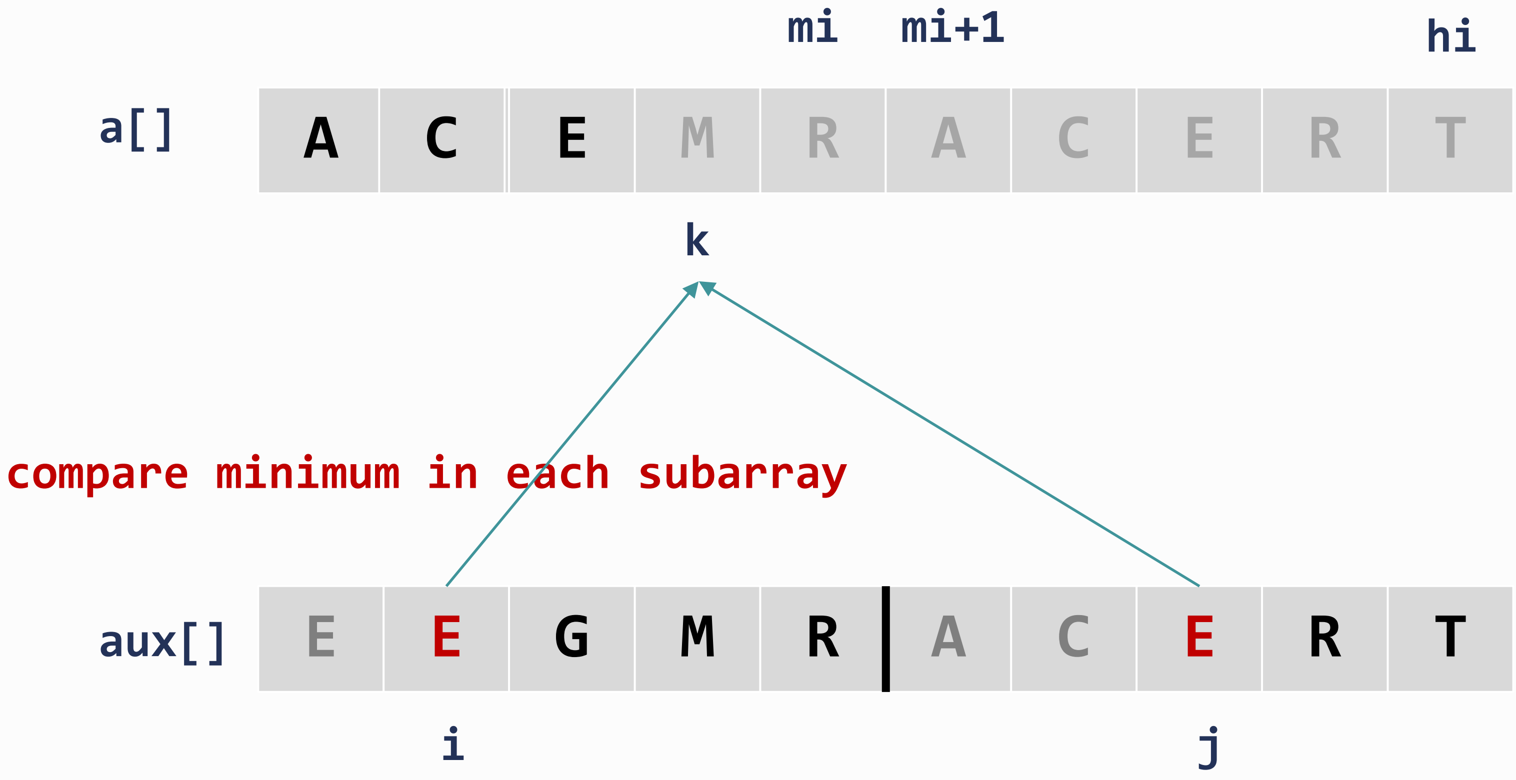
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



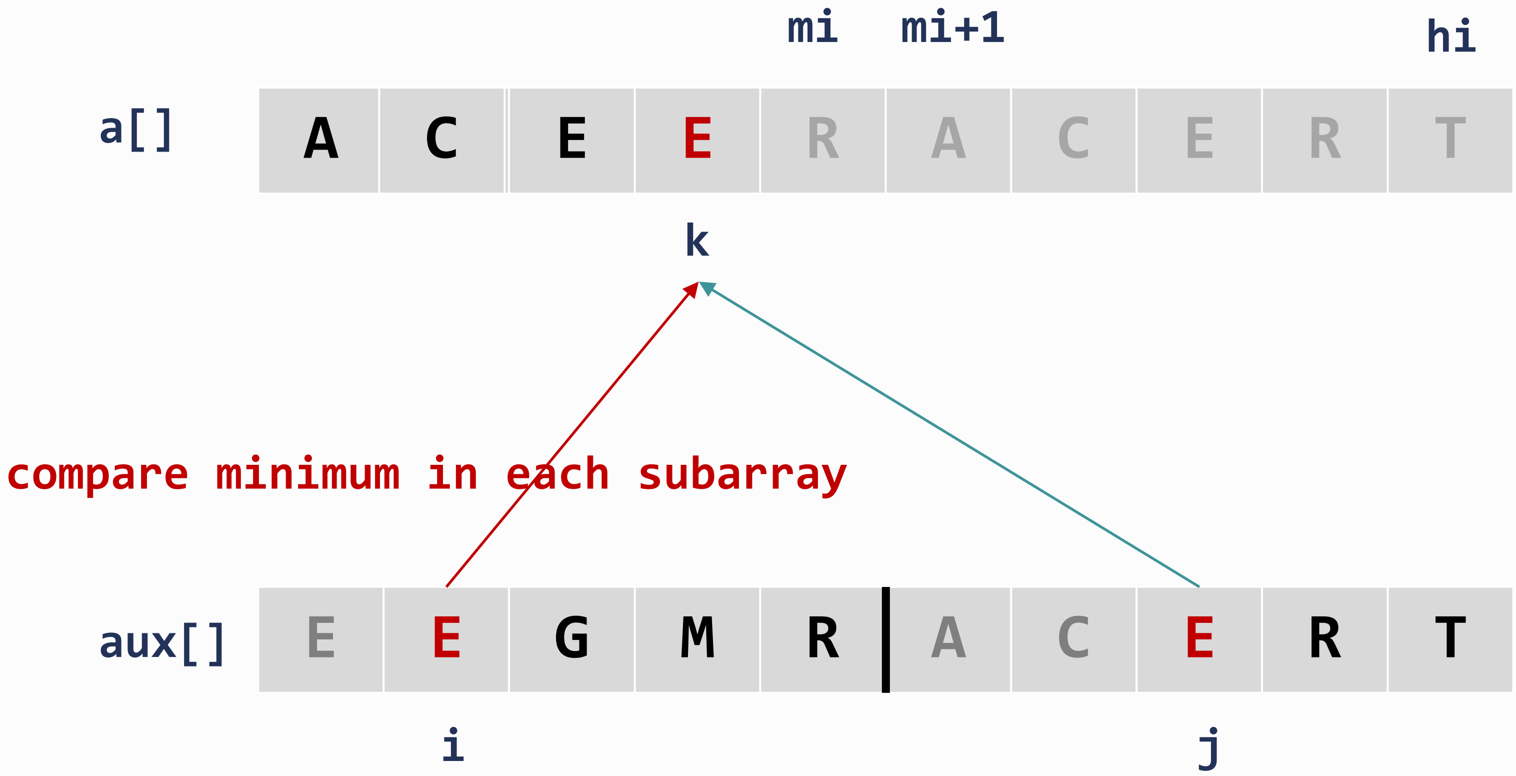
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



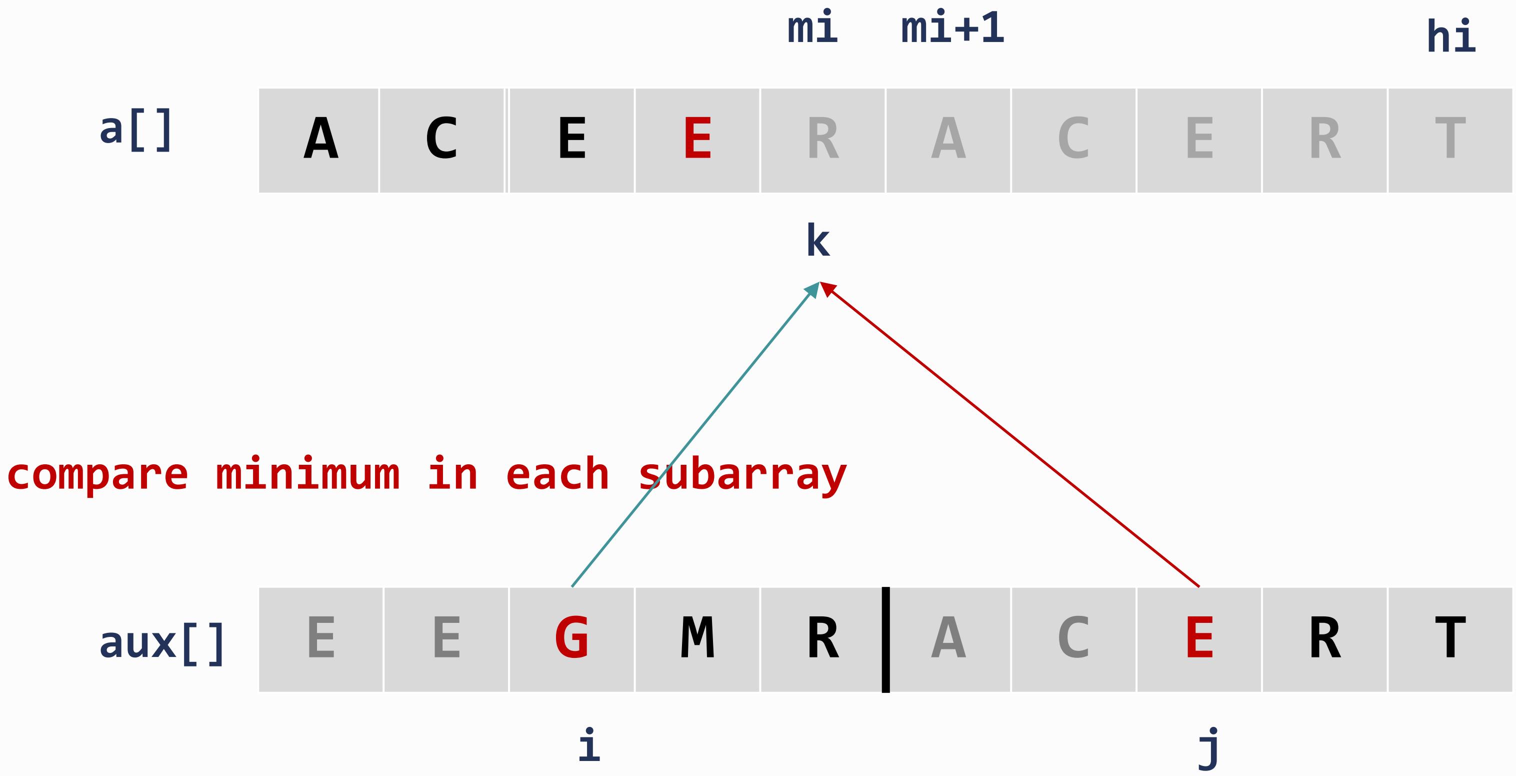
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



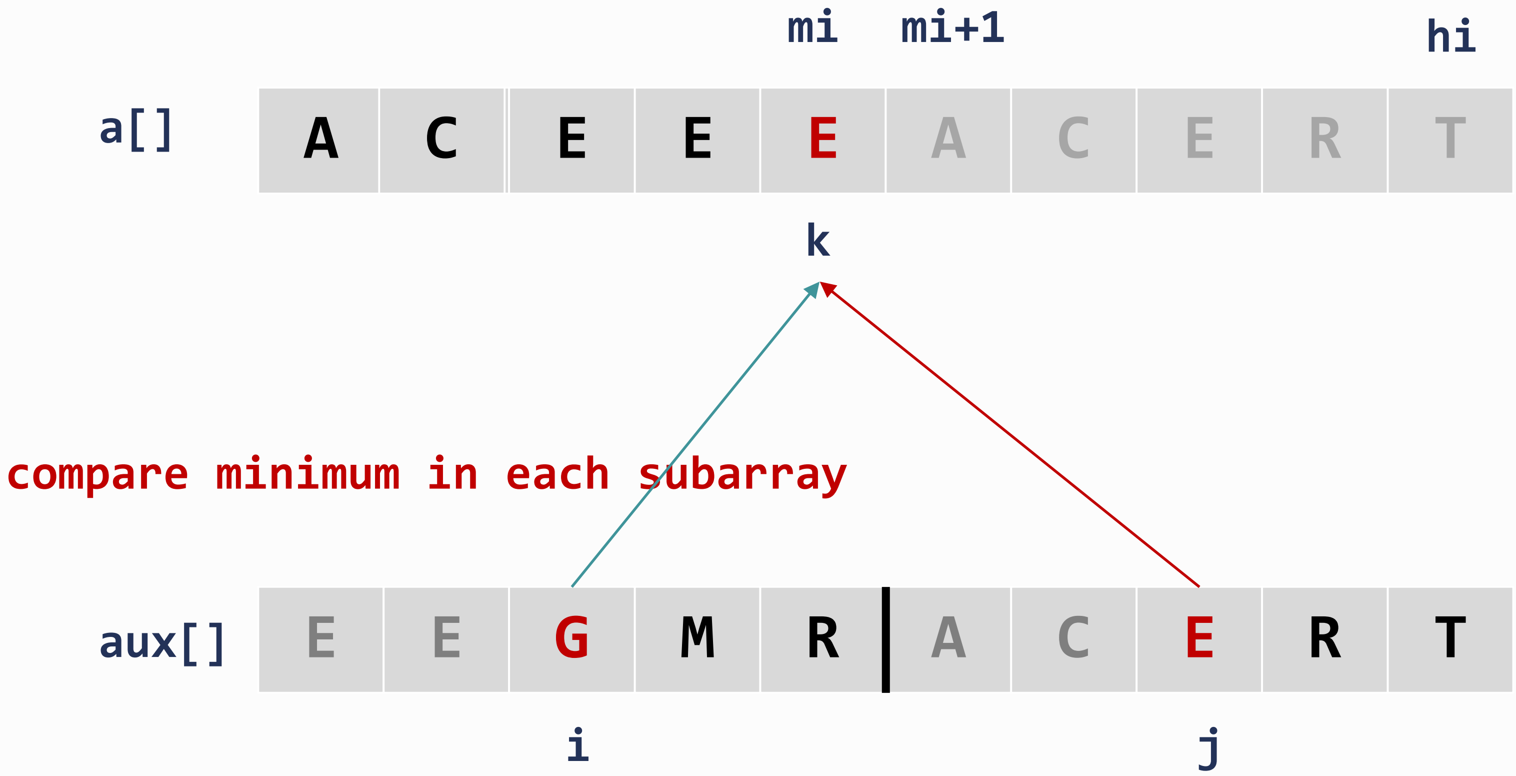
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



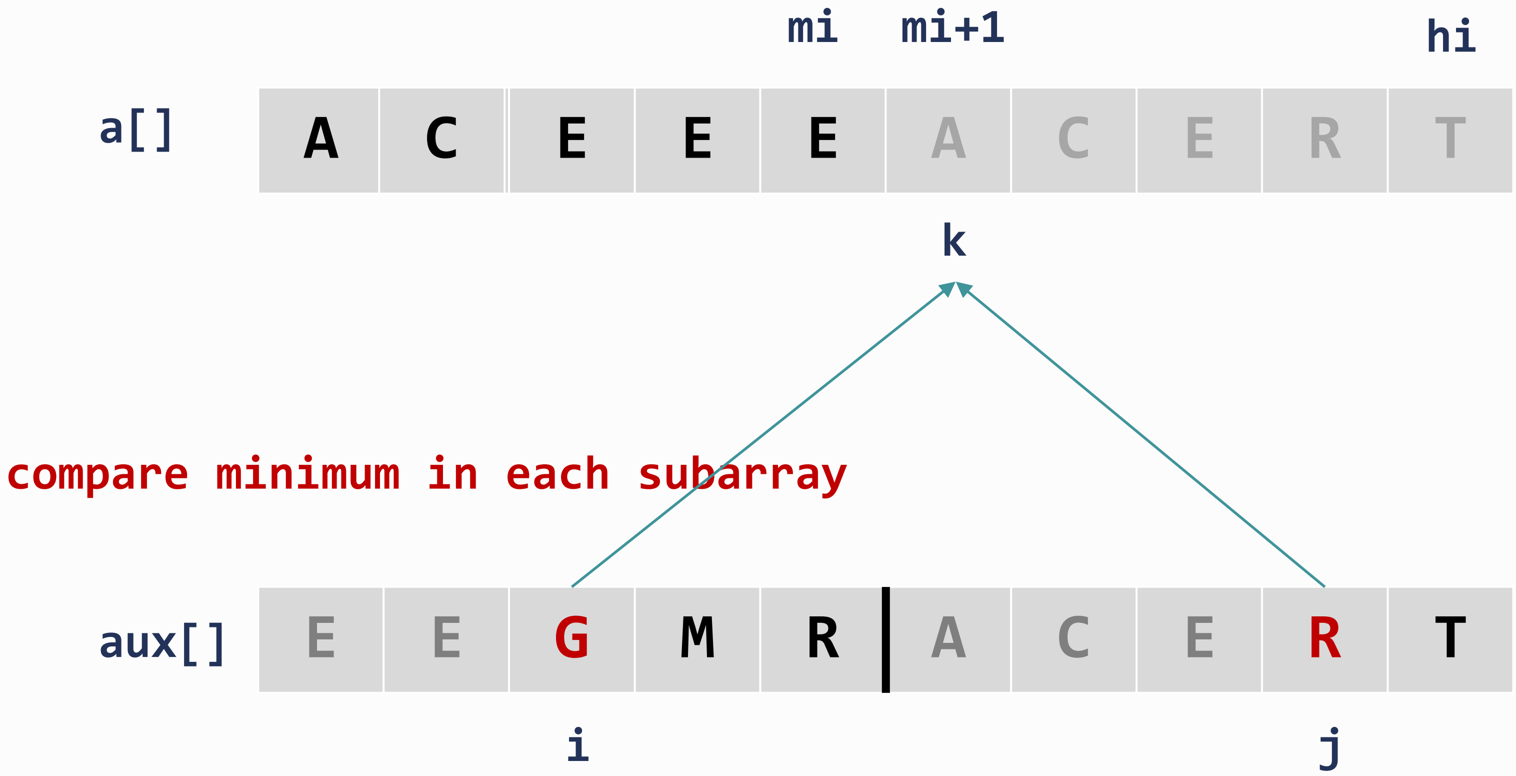
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



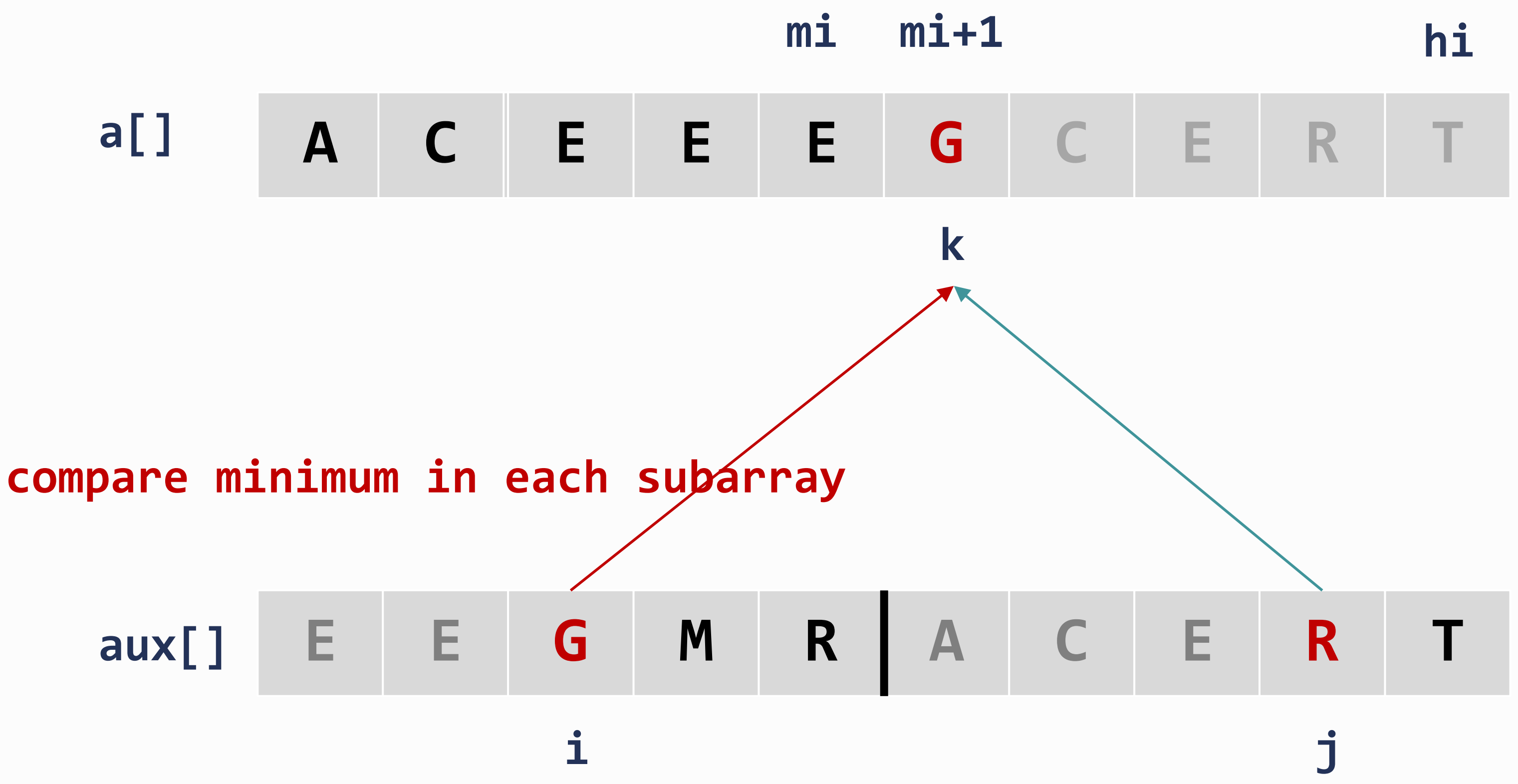
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



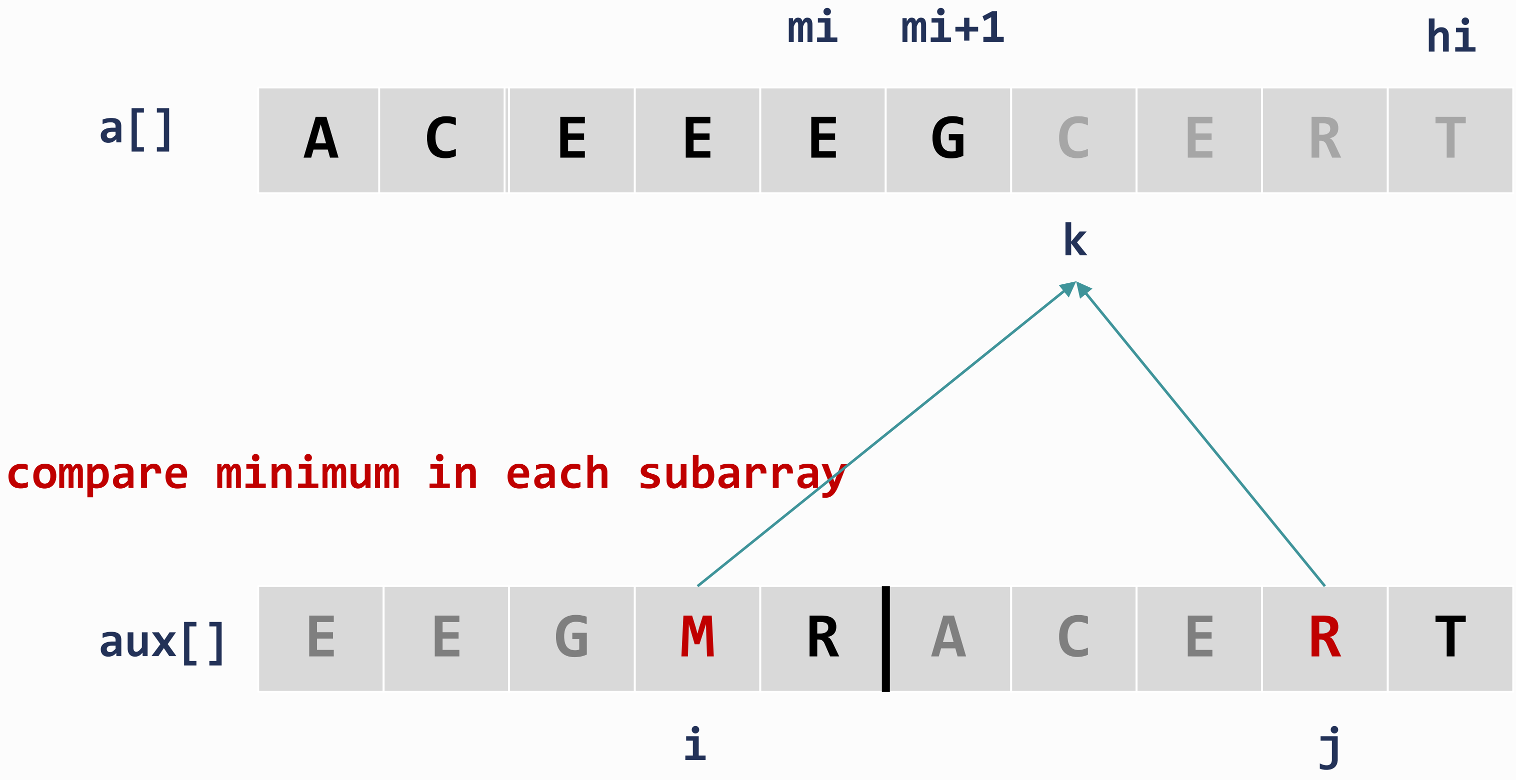
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



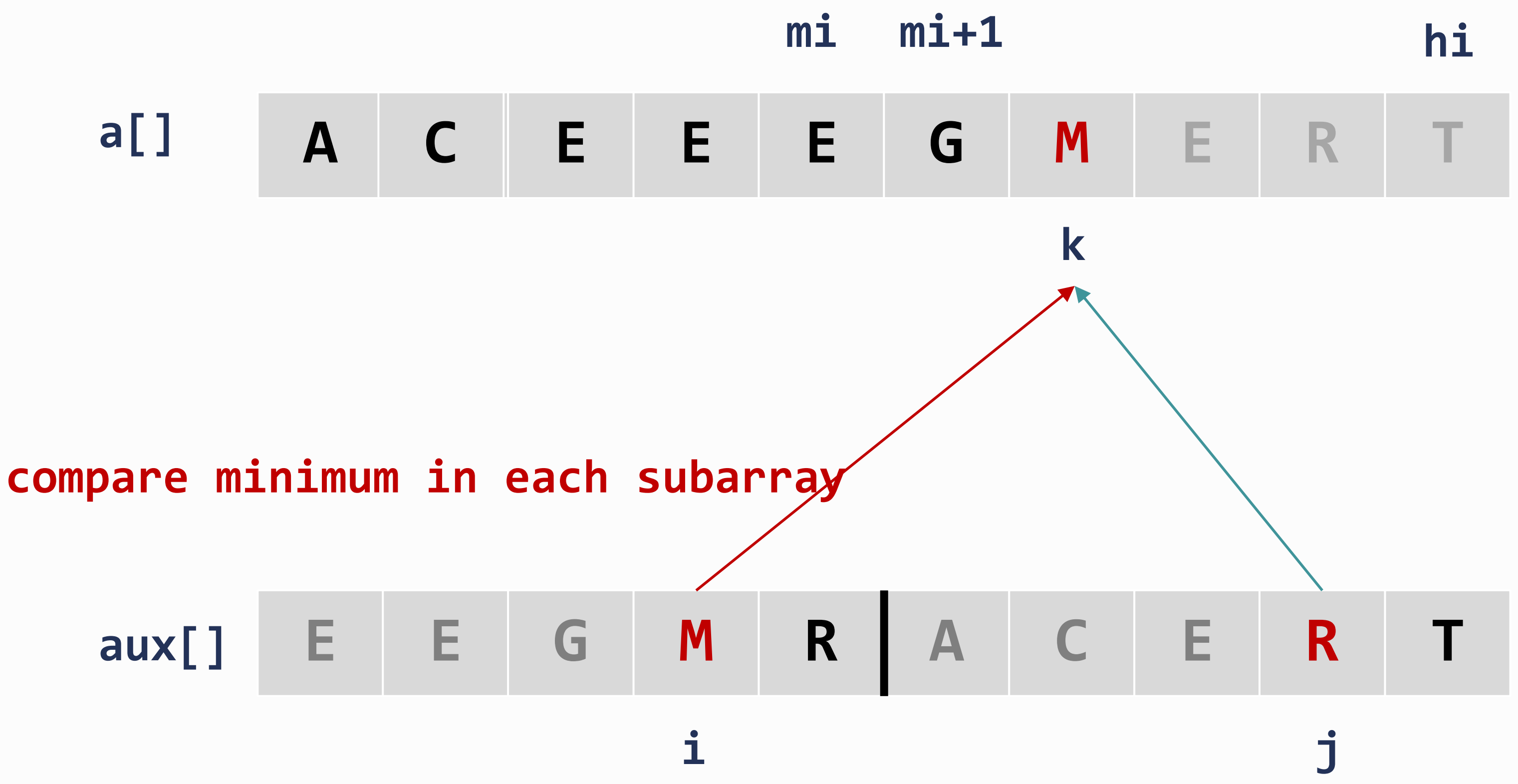
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



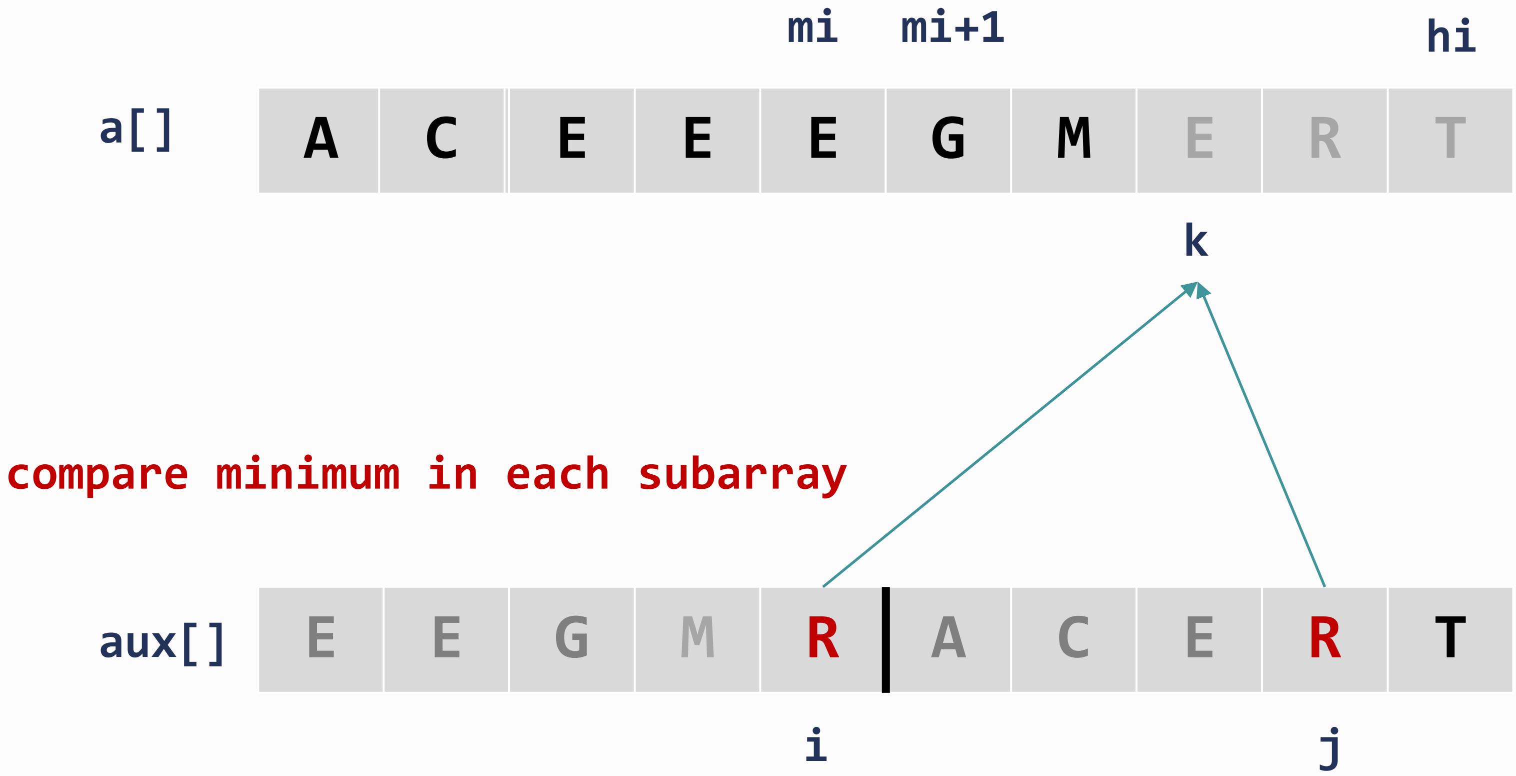
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



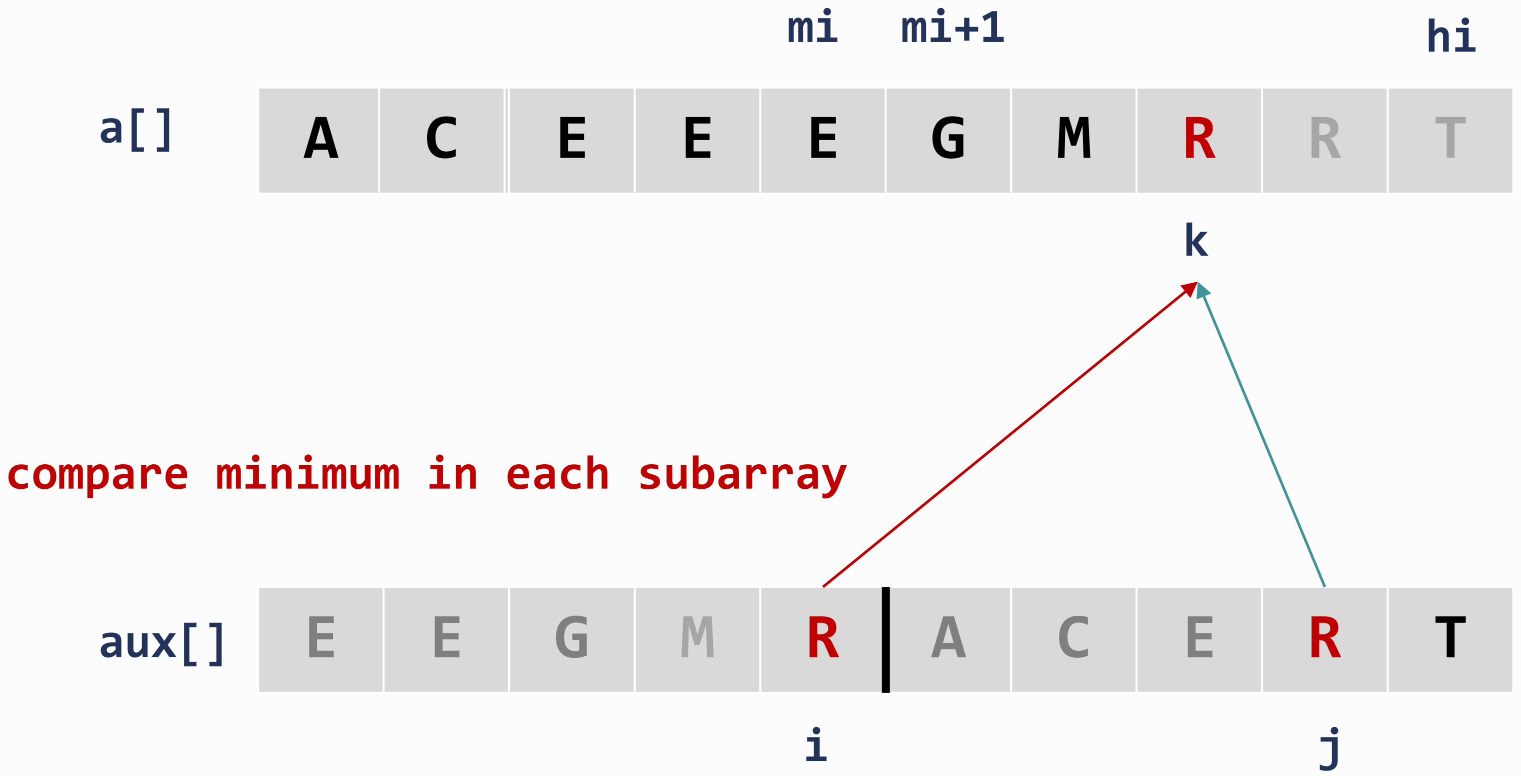
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



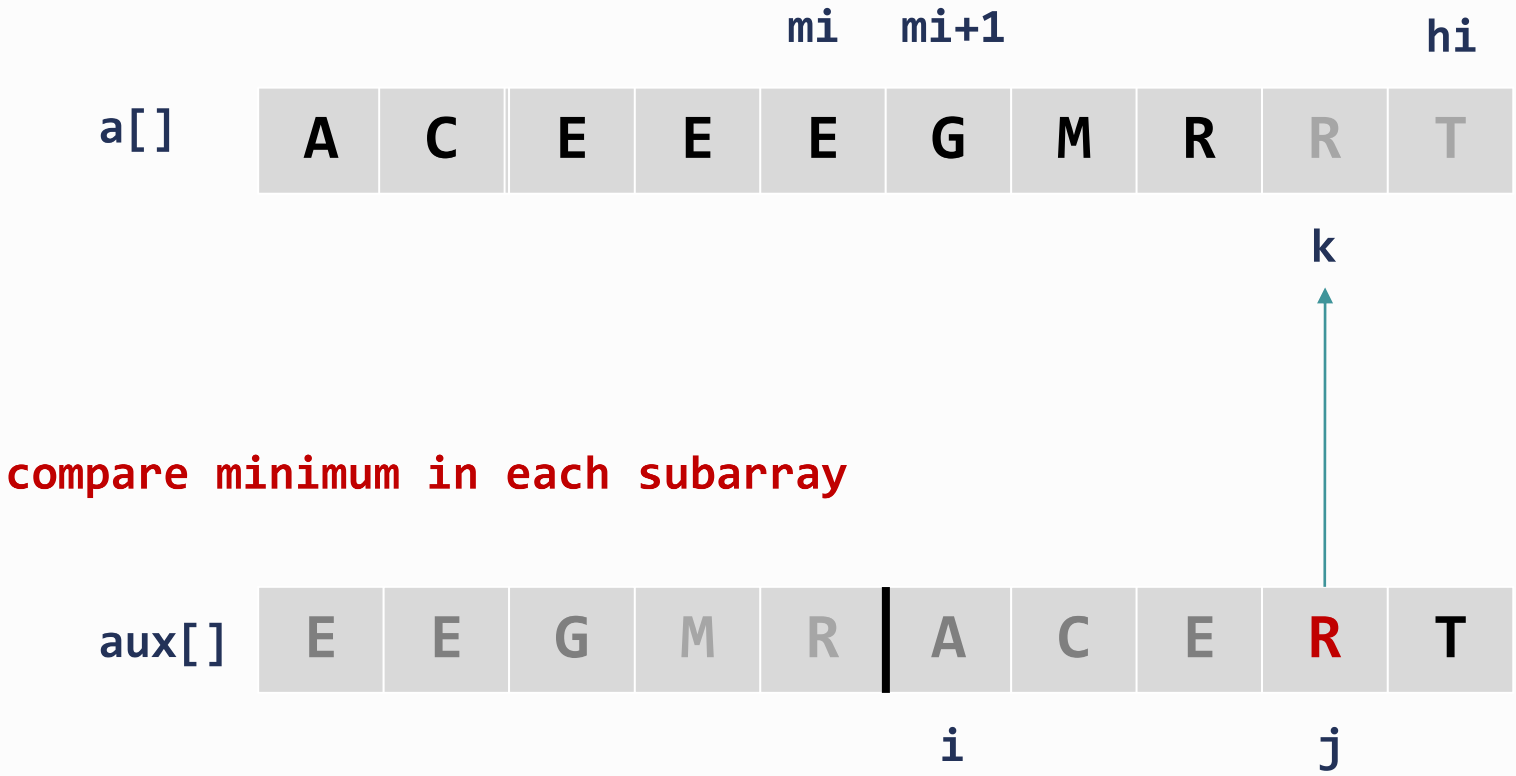
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



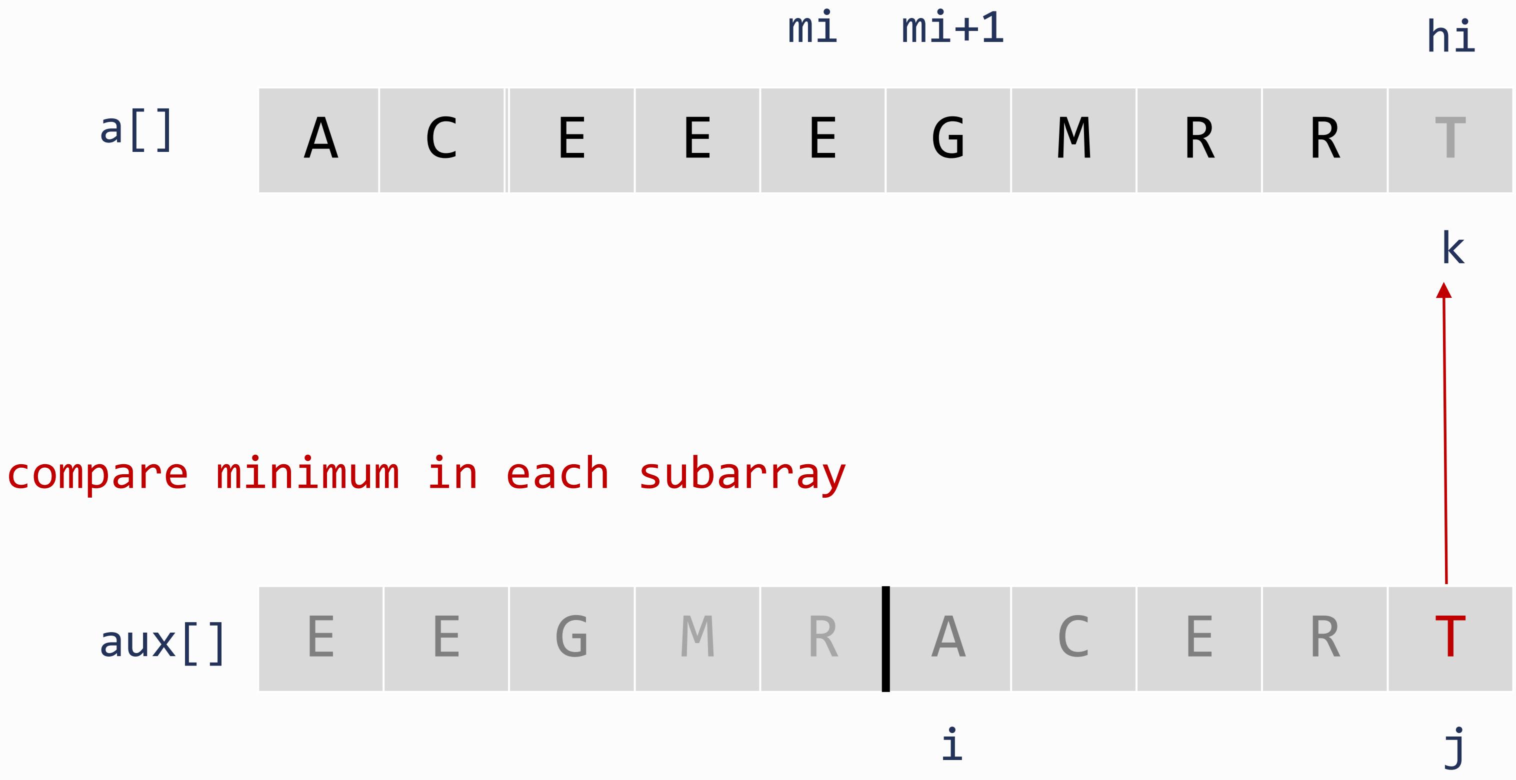
Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.



Merge sort: merge

- Goal: Given two sorted subarrays `a[lo]` to `a[mi]` and `a[mi+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

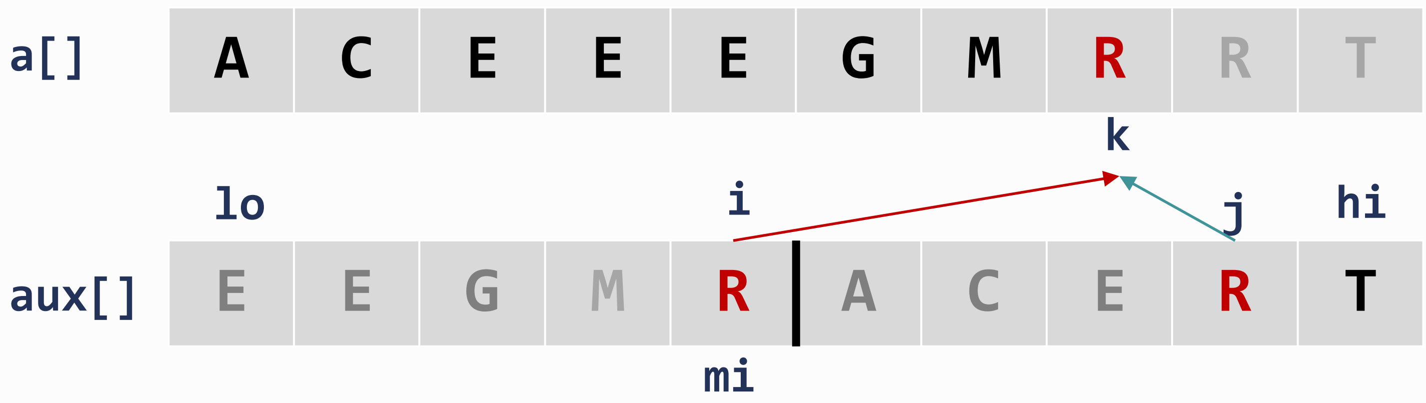
					mi	mi+1				hi
a[]	A	C	E	E	E	G	M	R	R	T

mergeSort complete using auxiliary array

aux[]	E	E	G	M	R	A	C	E	R	T

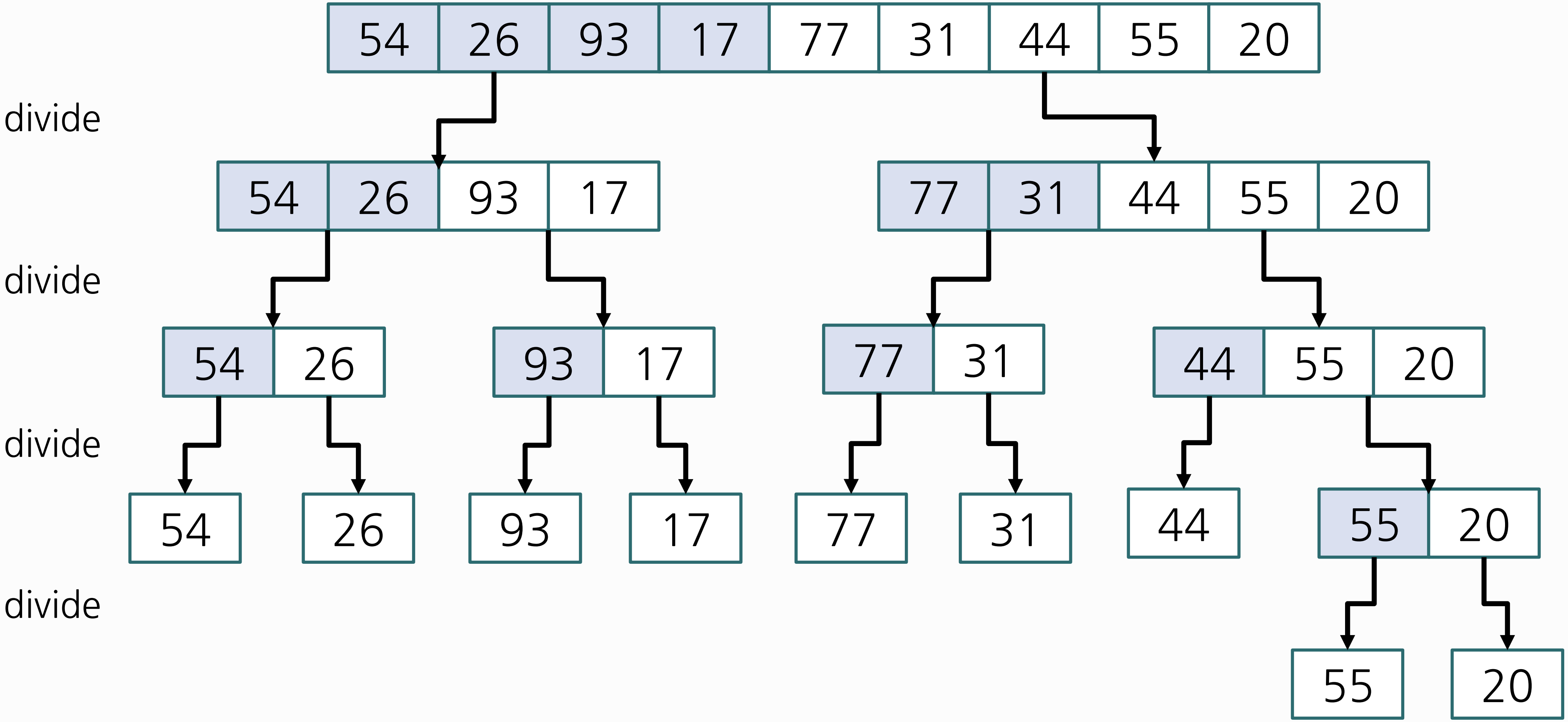
Merge sort: merge

- If your array is empty or has one element, it is sorted.
- If it has two elements, sort it by swapping as appropriate.
- If it has more than two elements, do this:
  - split the array in half at the midpoint **mi**;
  - call **mergesort()** on the left half;
  - call **mergesort()** on the right half;
  - **merge()** the arrays by picking the smallest head element from the two sub-arrays until they are exhausted.



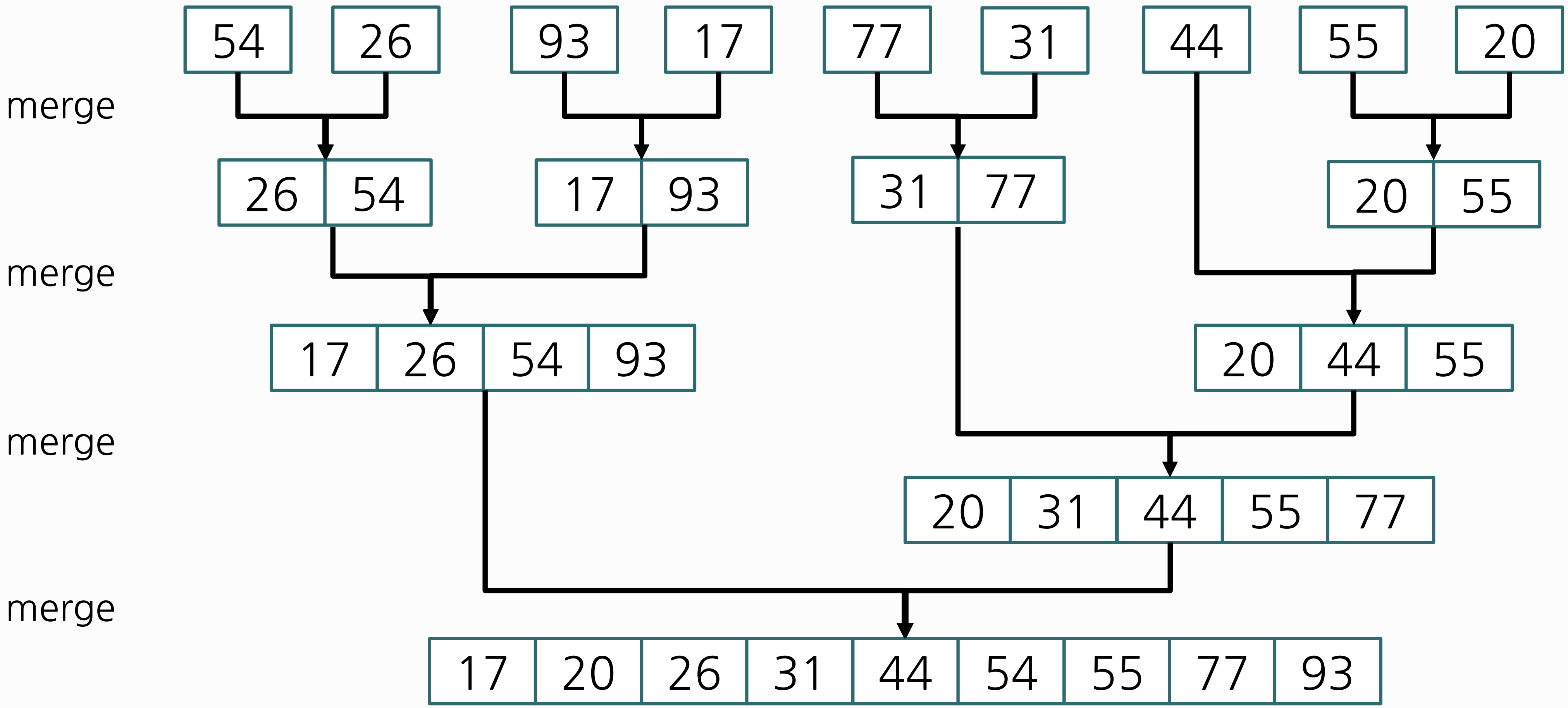
Merge sort

- Below is the call tree for the merge sort algorithm:



Merge sort

- Below is the tree of the merged parts returned by the merge sort algorithm:



Merge sort - Splitting list

- Slicing will be useful when halving the list in the merge sort code:

```
def split_list():
    a = [54, 26, 93, 17, 20]
    mi = len(a) // 2
    le = a[:mi]      # copy left half
    ri = a[mi:]      # copy right half
    print(a, le, ri)

if __name__ == "__main__":
    split_list()
```

```
[54, 26, 93, 17, 20] [54, 26] [93, 17, 20]
```

Merge sort - Merging the two halves of the list

```
def merge(a, le, ri):
    i = j = k = 0
    while i < len(le) and j < len(ri):
        if le[i] < ri[j]:
            a[k] = le[i]
            i = i + 1
        else:
            a[k] = ri[j]
            j = j + 1
        k = k + 1
    while i < len(le):
        a[k] = le[i]
        i = i + 1
        k = k + 1
    while j < len(ri):
        a[k] = ri[j]
        j = j + 1
        k = k + 1
```

```
[54, 26, 93, 17, 20] [54, 26] [93, 17, 20]
```

Merge sort - Merging the two halves of the list

```
if __name__ == "__main__":
    le, ri = [1, 2, 5], [3, 4, 6, 8, 10]
    a = [0] * (len(le) + len(ri))
    le, ri = [1, 2, 5], [3, 4, 6, 8, 10]
    print(le, ri)
    merge(a, le, ri)
    print(a)
```

```
[1, 2, 5] [3, 4, 6, 8, 10]
[1, 2, 3, 4, 5, 6, 8, 10]
```

Merge sort Code

- Use the function merge() defined previously to merge two halves.

```
def merge_sort(a):
    if len(a) > 1:
        mi = len(a) // 2
        le = a[:mi]
        ri = a[mi:]
        merge_sort(le)
        merge_sort(ri)
        merge(a, le, ri)

if __name__ == "__main__":
    a = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print("before:", a)
    merge_sort(a)
    print(" after:", a)
```

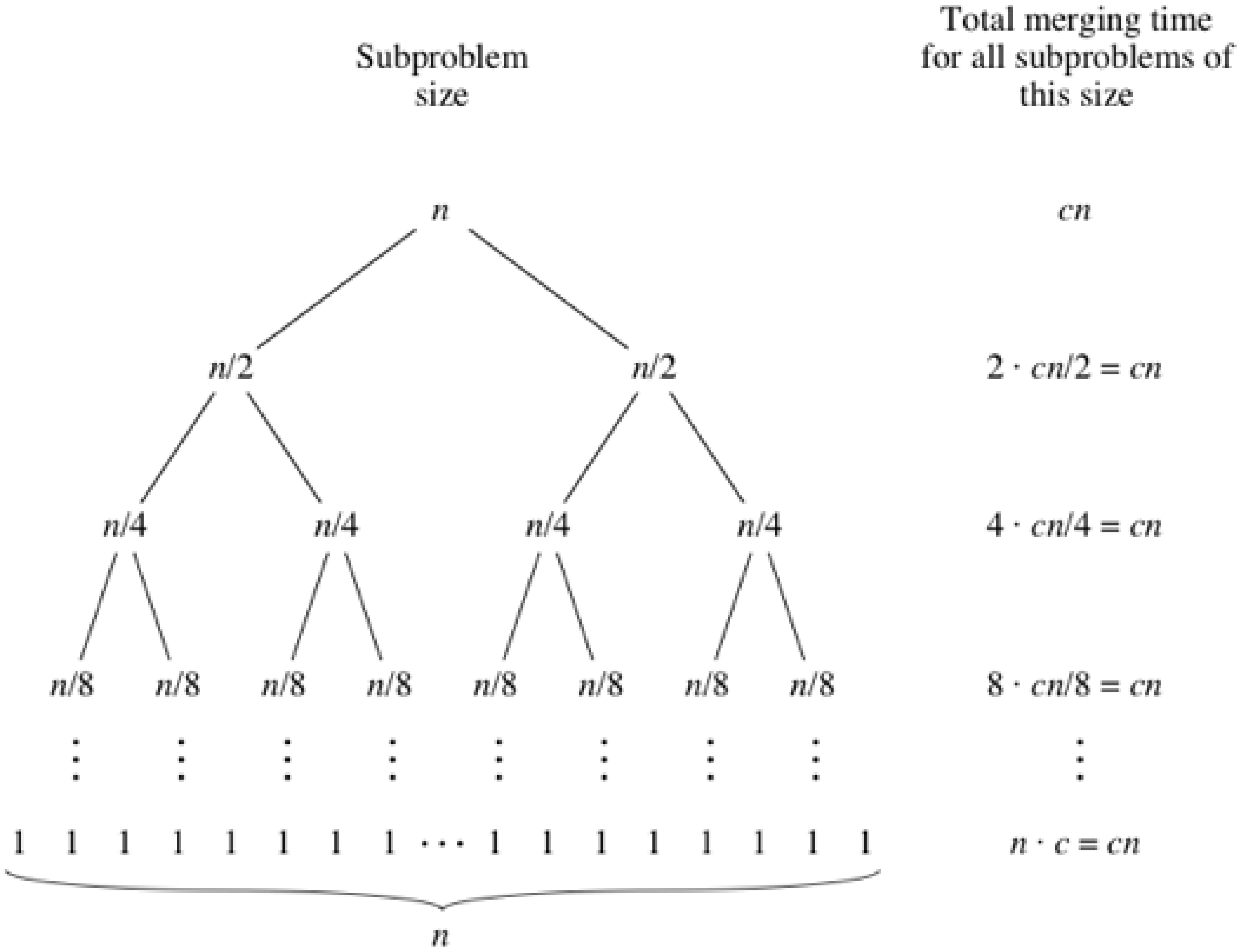
```
before: [54, 26, 93, 17, 77, 31, 44, 55, 20]
 after: [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Merge sort - Big O

- To analyze the complexity of merge sort, you can look at its two steps separately:
  - **Split Step: (Divide)**
    Since the array is halved until a single element remains, the total number of halving operations performed ($\log_2 n$) times. Since there are no comparisons**, however,** swap nor shift operations during this step, we may consider this step takes constant time $O(1)$ regardless of the subarray size.
  - **Merge Step: (Conquer and Combine)**
    It receives two arrays whose combined length is at most n (the length of the original input array), and **it combines** both arrays by looking at each element at most once for the comparison. This leads to a runtime complexity of $\boldsymbol{O(n)}$**.**
    Since we have split the input array $\log_2 n$ times, we also must merge $\log_2 n$ times as well. Then we get a total time complexity of $\boldsymbol{O(n\ log_2 n)}$.
- Therefore, The time complexity of the merge sort becomes $\boldsymbol{O(n\ log_2 n)}$.

Merge sort - Big O

- To analyze the time complexity of merge sort:



Compare two subarrays and merge them into one: cn → $O(n)$

The merging time at each level: cn
The number of levels: $level = \log_2 n + 1$
The total merging time: $level * cn$

- Therefore, the time complexity of the merge sort becomes $\boldsymbol{O(n \log_2 n)}$.

Merge sort - Big O

- Let $T(n)$ be the total time taken by the Merge sort algorithm.
  - Sorting two halves will take at the most $2\,T(n/2)$ time.
  - When we merge the sorted lists, we come up with a total $n-1$ comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

  - Thus, the recurrence relation will be:

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n - 1 &\quad (1)\\
&= 2T(\frac{n}{2}) + n &\quad (2)
\end{aligned}
$$

Merge sort - Big O

- The timing a list of size 1 is constant, i.e., $T(1) = 1$.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n && (1)\\
&= 2(2T(\frac{n}{2^2})) + n + n && (2)\\
&= 2(2(2T(\frac{n}{2^3}))) + n + n + n && (3)\\
&= 2^4T(\frac{n}{2^4}) + 4n && (4)\\
&= \dots && (5)\\
&= 2^kT(\frac{n}{2^k}) + kn && (6)
\end{aligned}
$$

- Since the base case, $T(1) = T(\frac{n}{2^k})$, occurs when $n = 2^k$. That is, $k = \log n$.

$$T(n) = n \cdot T(\frac{n}{n}) + n \cdot \log_2 n = n + n \cdot \log_2 n \qquad (1)$$

- Therefore, Big O of Merge sort is $O(n \log_2 n)$.

Summary

Algorithm	Best	Worst	Average	Extra Memory	
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	slow
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Good if often almost sorted
Shell	$O(n)$	$O(n\,(\log n)^2)$	$O(n\,(\log n)^2)$	$O(1)$	
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	**Good for very large datasets**
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n)$	Faster than merge sort in general
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Best if $O(n \log n)$ required
Tim	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	used in Python, hybrid of merge sort and nsertion sort

- Note: A comparison-based sorting algorithm cannot be better than $O(n \log n)$ in the average and worst case.

Data Structures in Python
Chapter 5 - 2

- **Merge sort**
- Quick sort Algorithm
- Quick sort Analysis
- Empirical Analysis

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University