

# Sorting(2/2)

**Data Structures**  
**C++ for C Coders**

한동대학교 김영섭 교수  
idebtor@gmail.com

Merge Sort  
Quick Sort

# Objectives & Agenda

---

- **Objectives:**
  - Understand advanced algorithms of sorting.
- **Agenda**
  - Merge Sort
  - Quick Sort

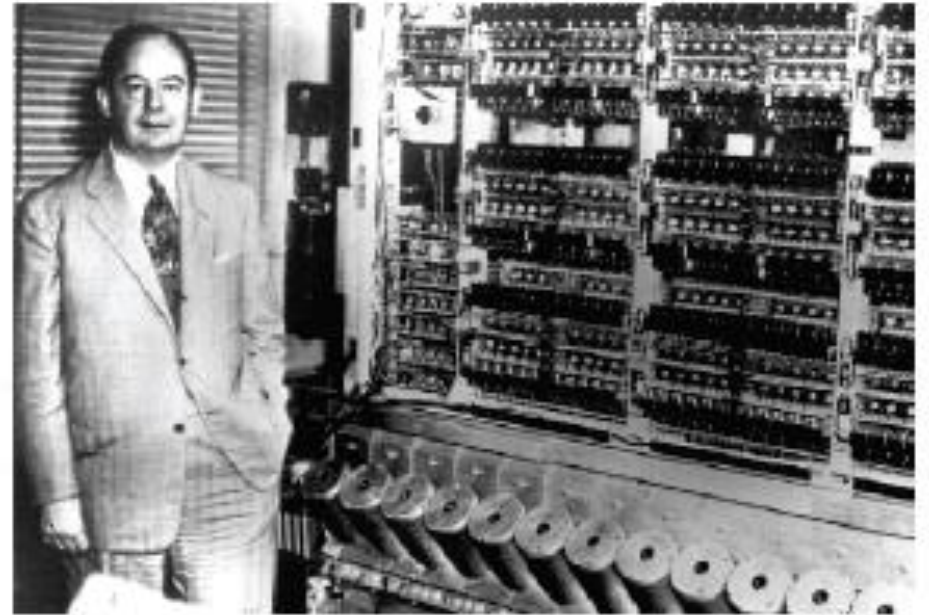
# Mergesort

- Divide and conquer algorithm
- 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,

## First Draft of a Report on the EDVAC

John von Neumann



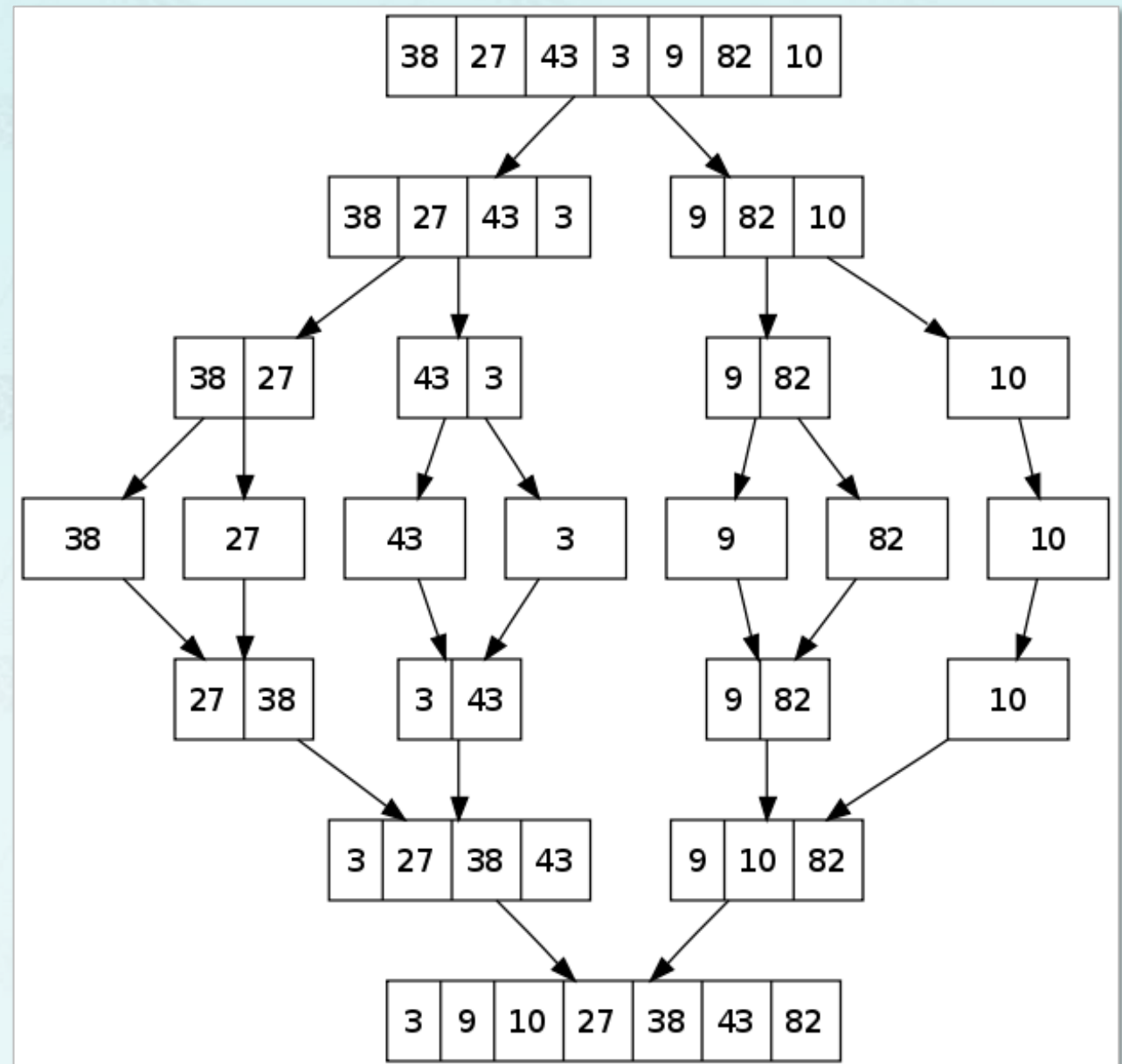
# Mergesort: Algorithm

---

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

# Mergesort: Algorithm

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

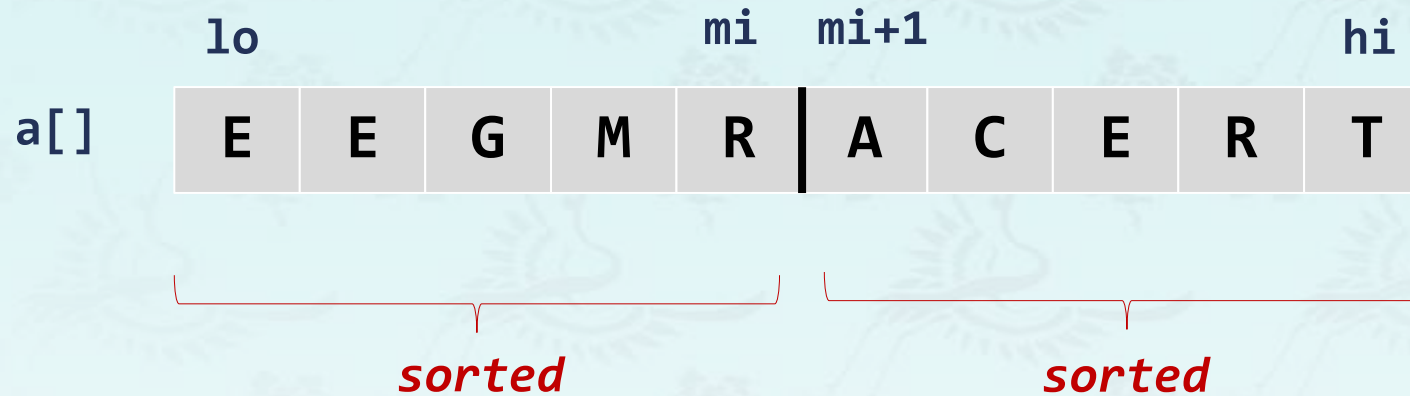


input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

# Mergesort: 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  $a[hi]$ .



# Mergesort: 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  $a[hi]$ .

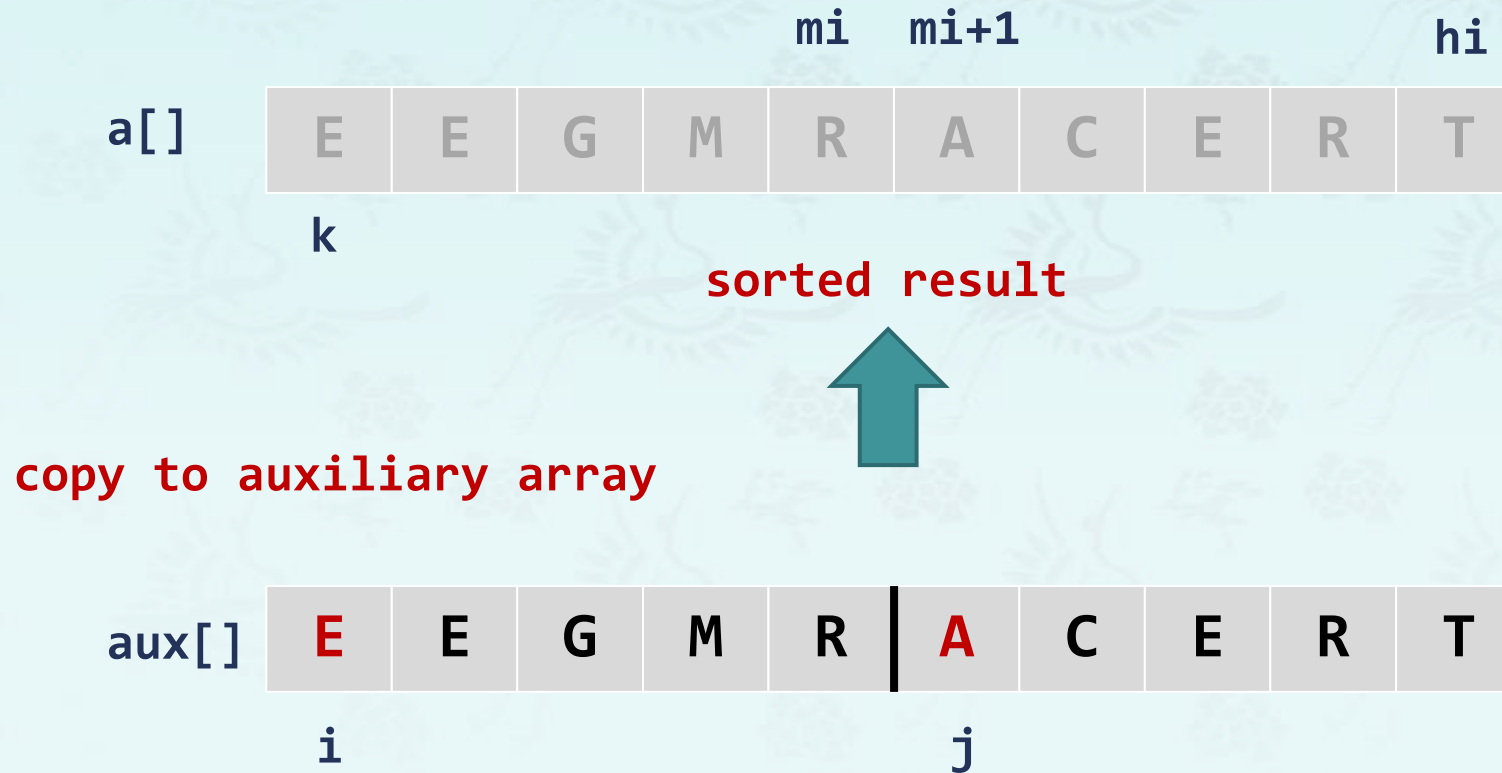


copy to auxiliary array



# Mergesort: merge

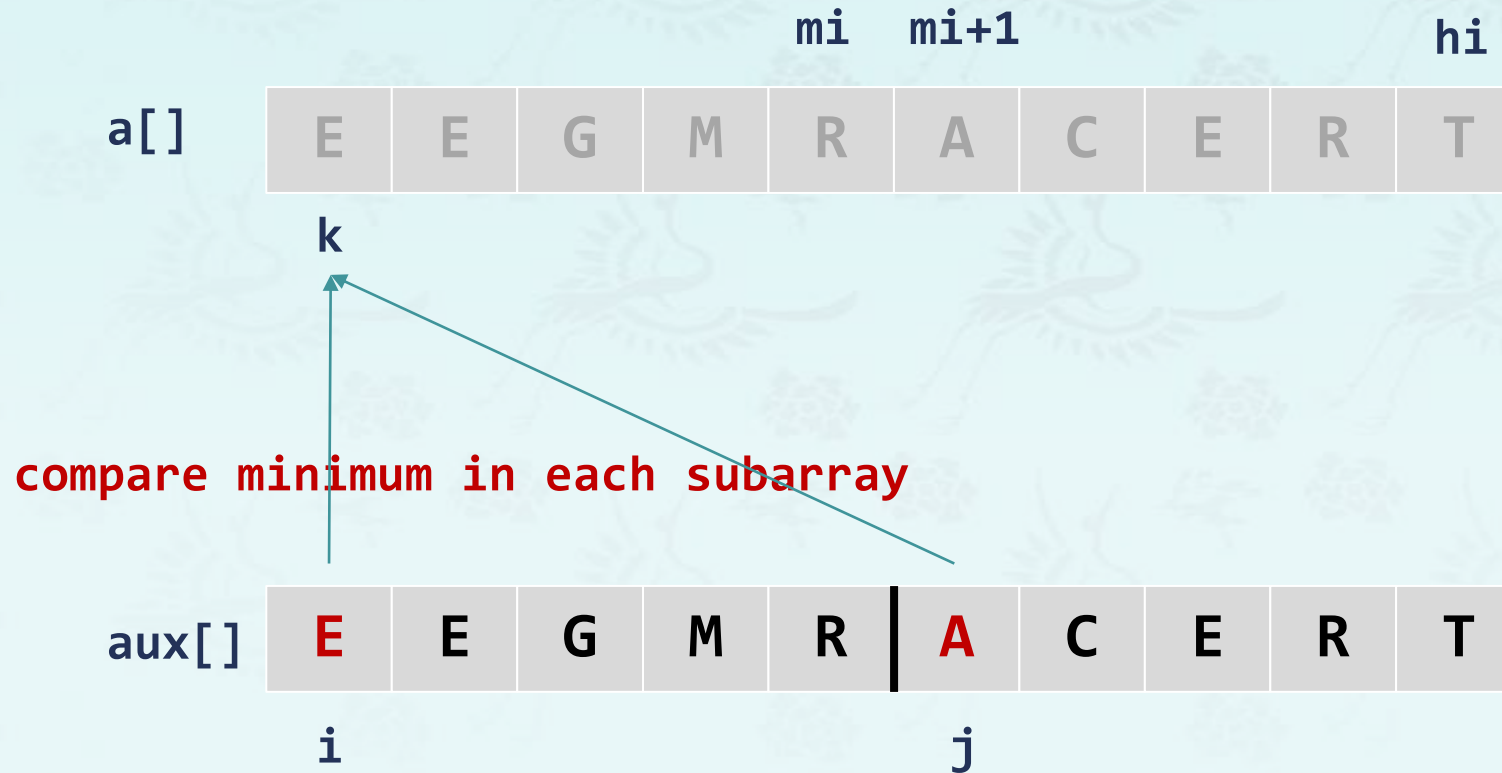
- **Goal:** Given two sorted subarrays  $a[lo]$  to  $a[mi]$  and  $a[mi+1]$  to  $a[hi]$ , replace with sorted subarray  $a[lo]$  to  $a[hi]$ .





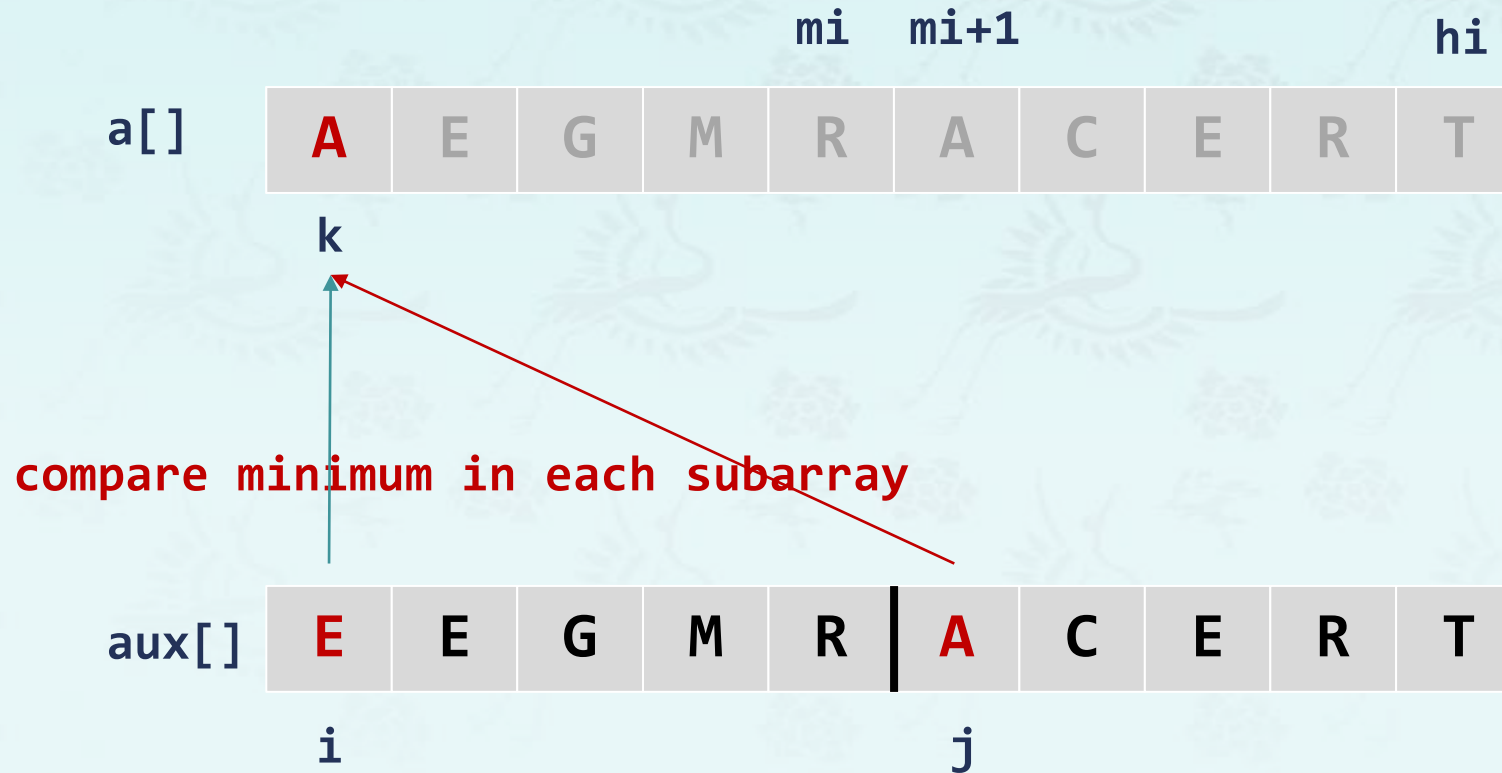
# Mergesort: merge

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



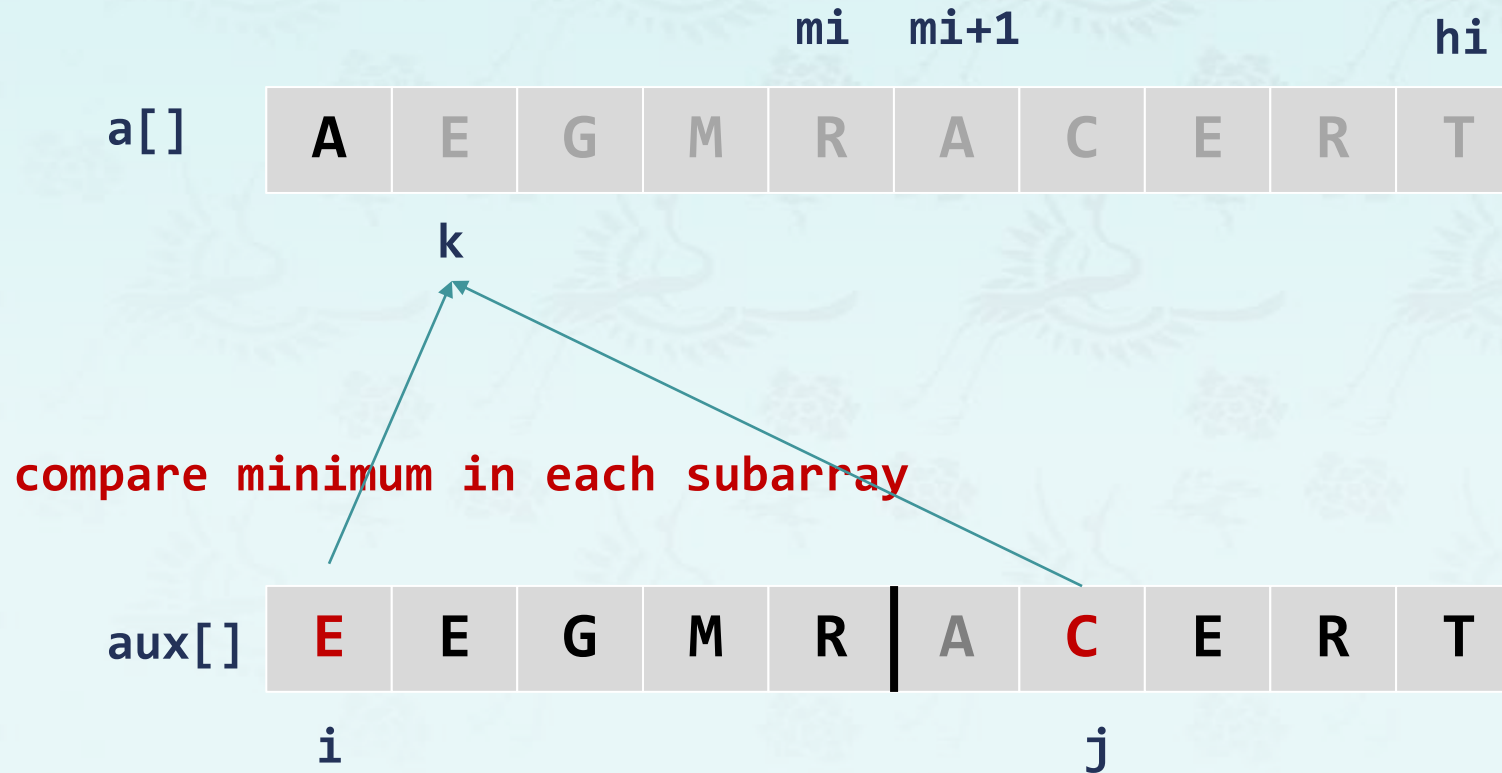
# Mergesort: merge

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



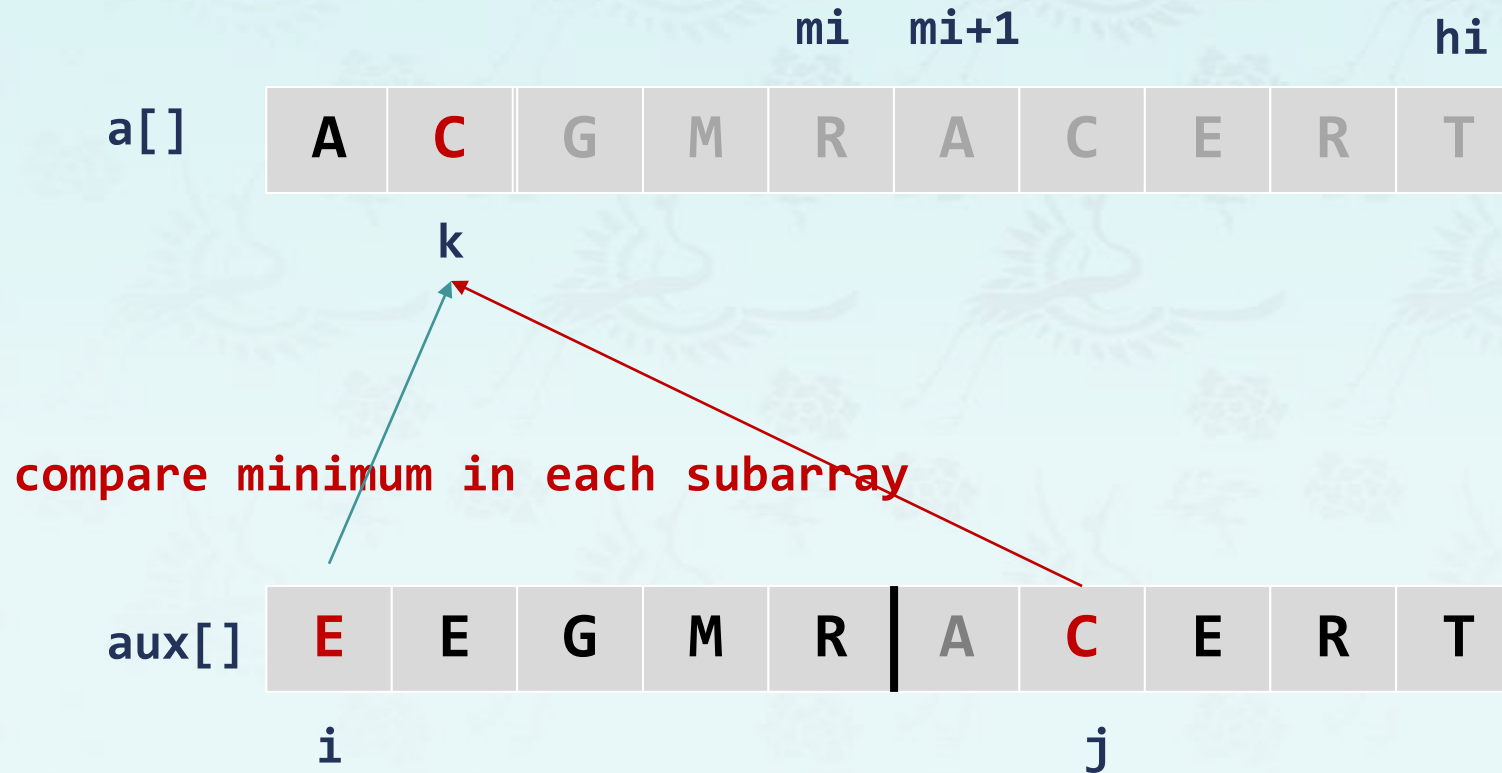
# Mergesort: merge

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



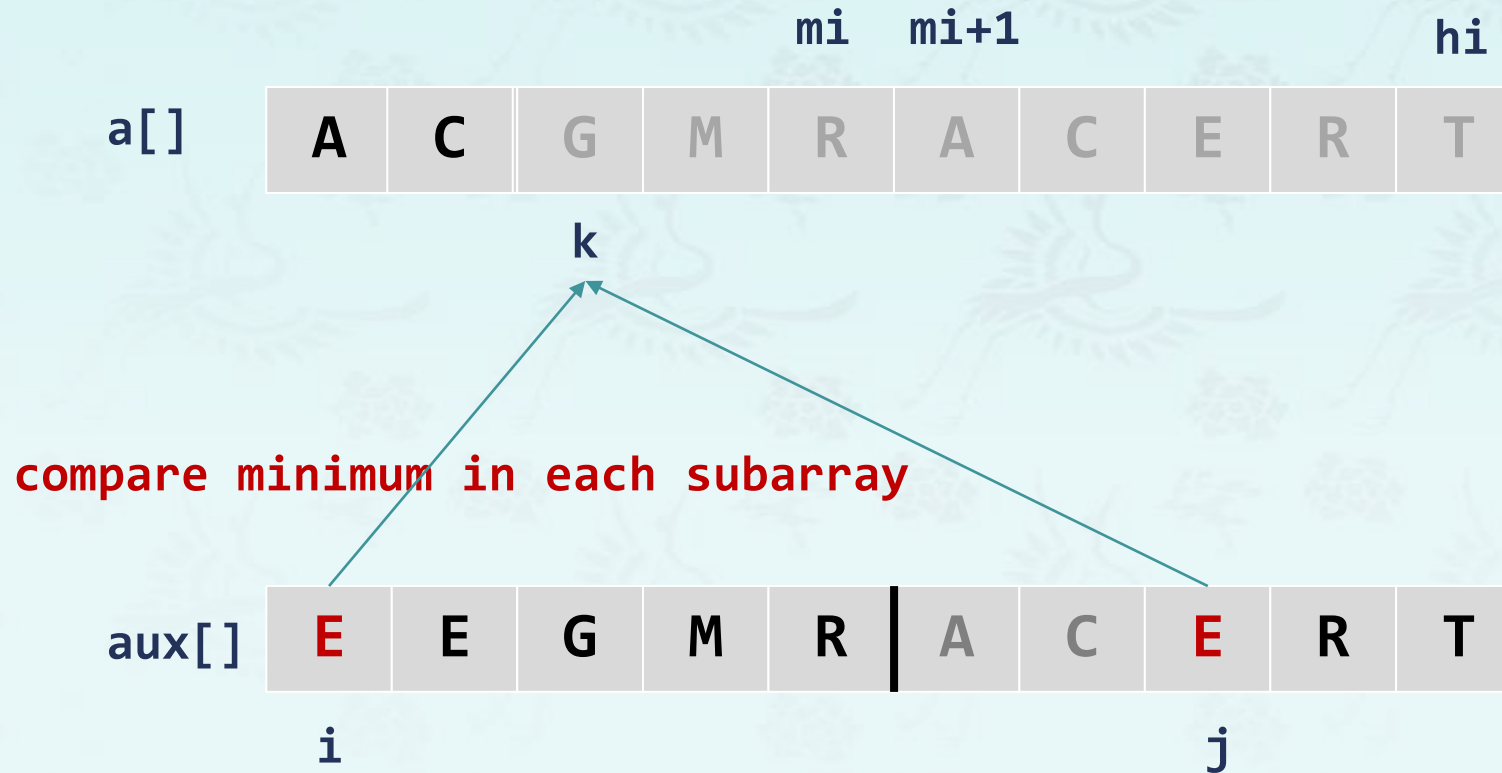
# Mergesort: merge

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



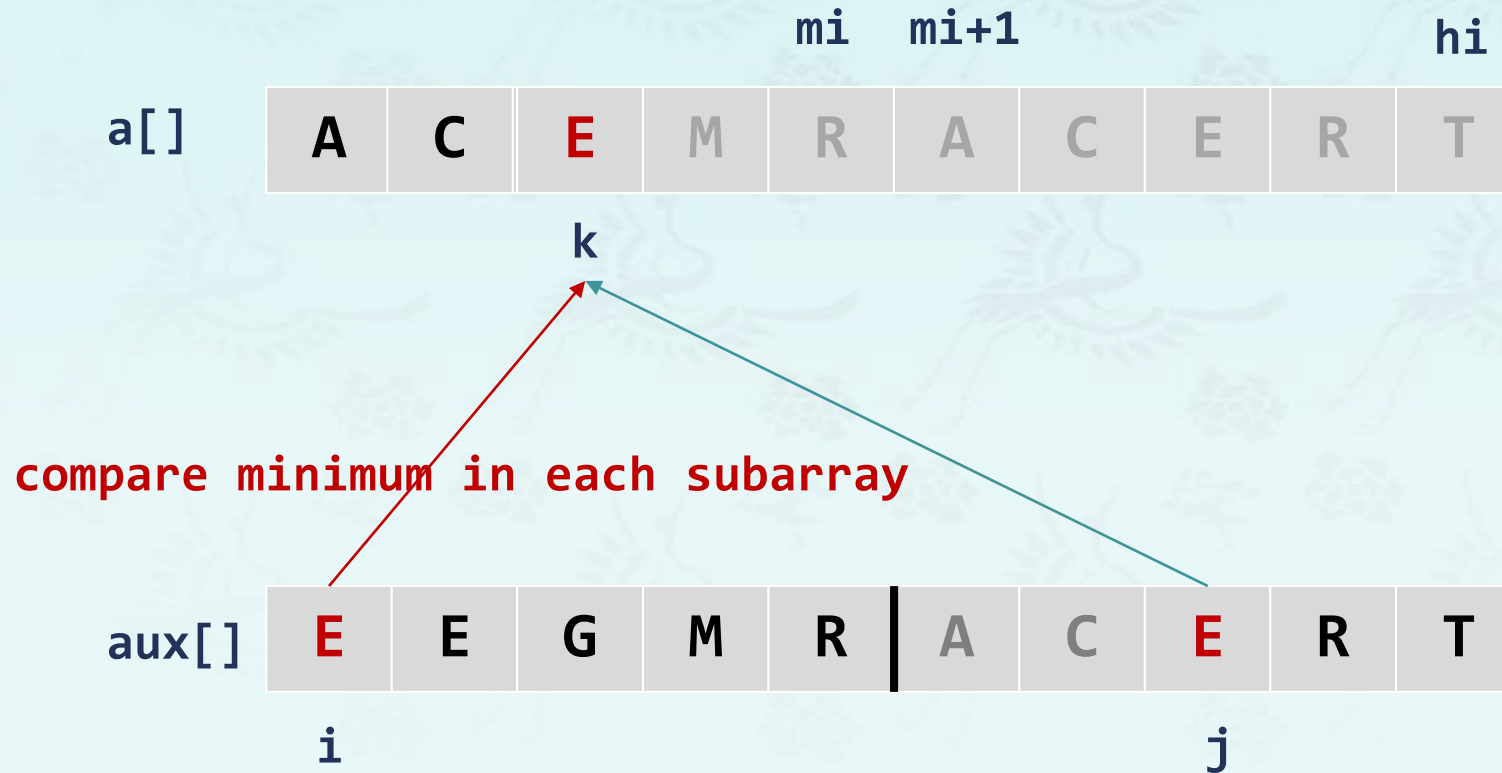
# Mergesort: merge

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



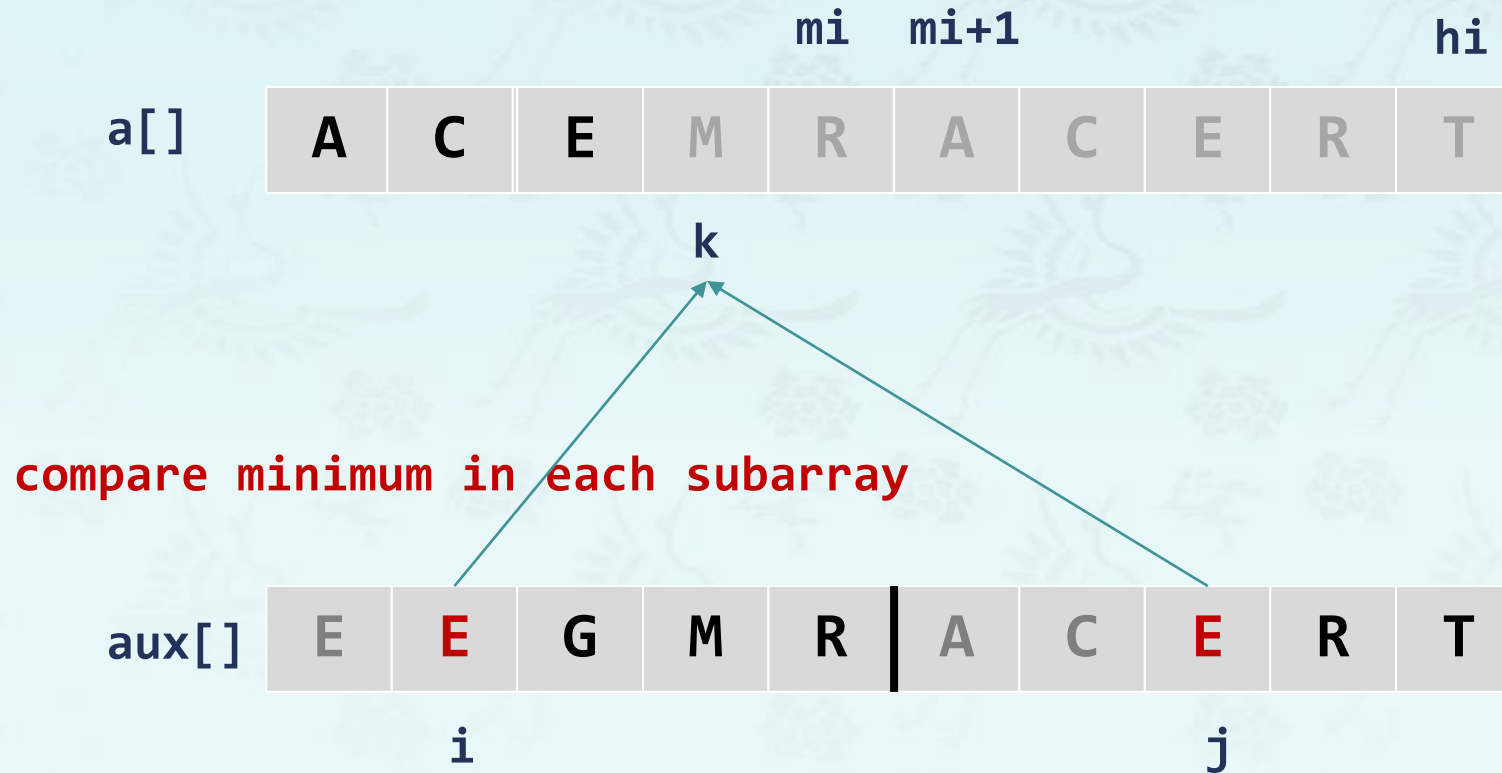
# Mergesort: merge

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



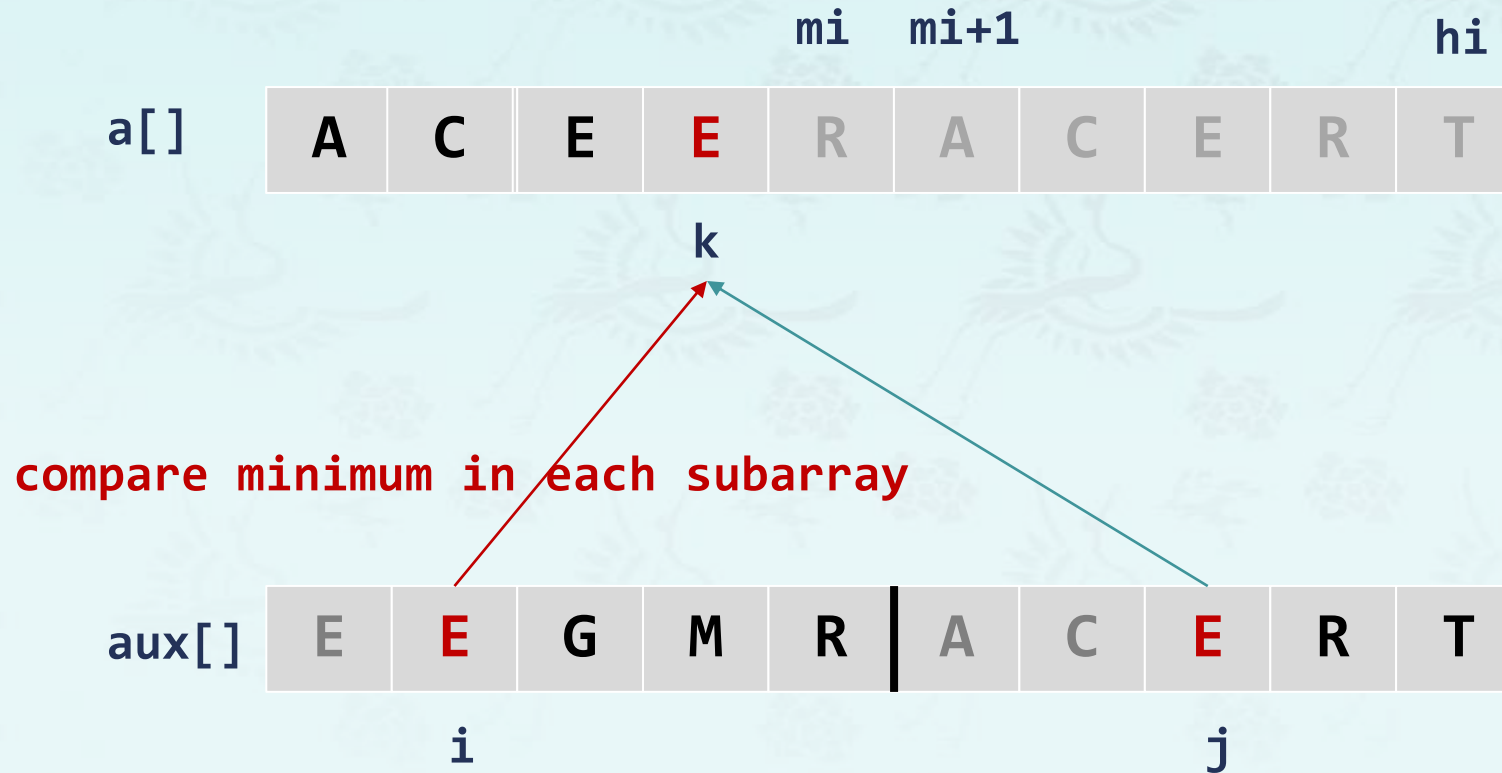
# Mergesort: merge

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



# Mergesort: merge

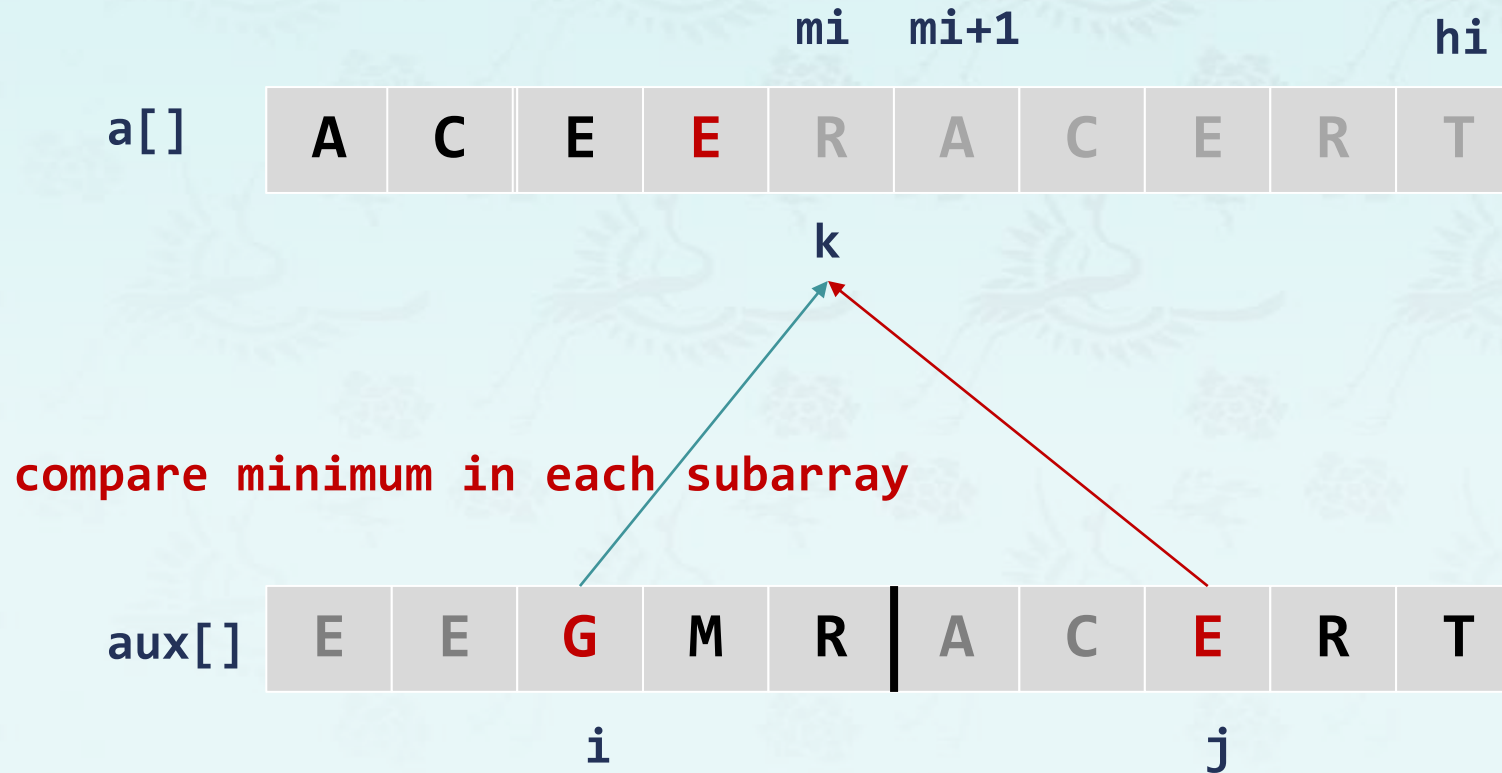
- **Goal:** Given two sorted subarrays  $a[lo]$  to  $a[mi]$  and  $a[mi+1]$  to  $a[hi]$ , replace with sorted subarray  $a[lo]$  to  $a[hi]$ .





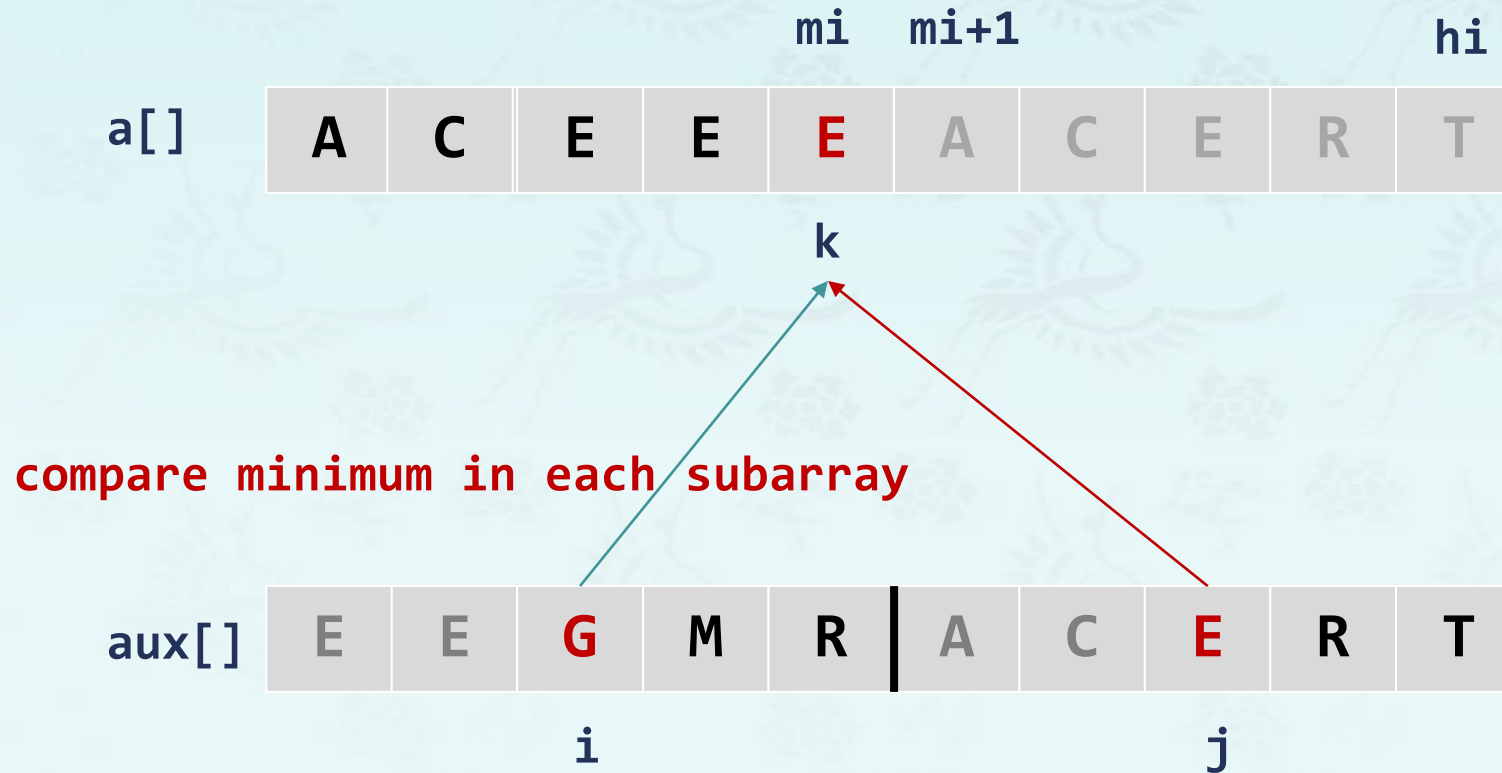
# Mergesort: merge

- **Goal:** Given two sorted subarrays  $a[\text{lo}]$  to  $a[\text{mi}]$  and  $a[\text{mi}+1]$  to  $a[\text{hi}]$ , replace with sorted subarray  $a[\text{lo}]$  to  $a[\text{hi}]$ .



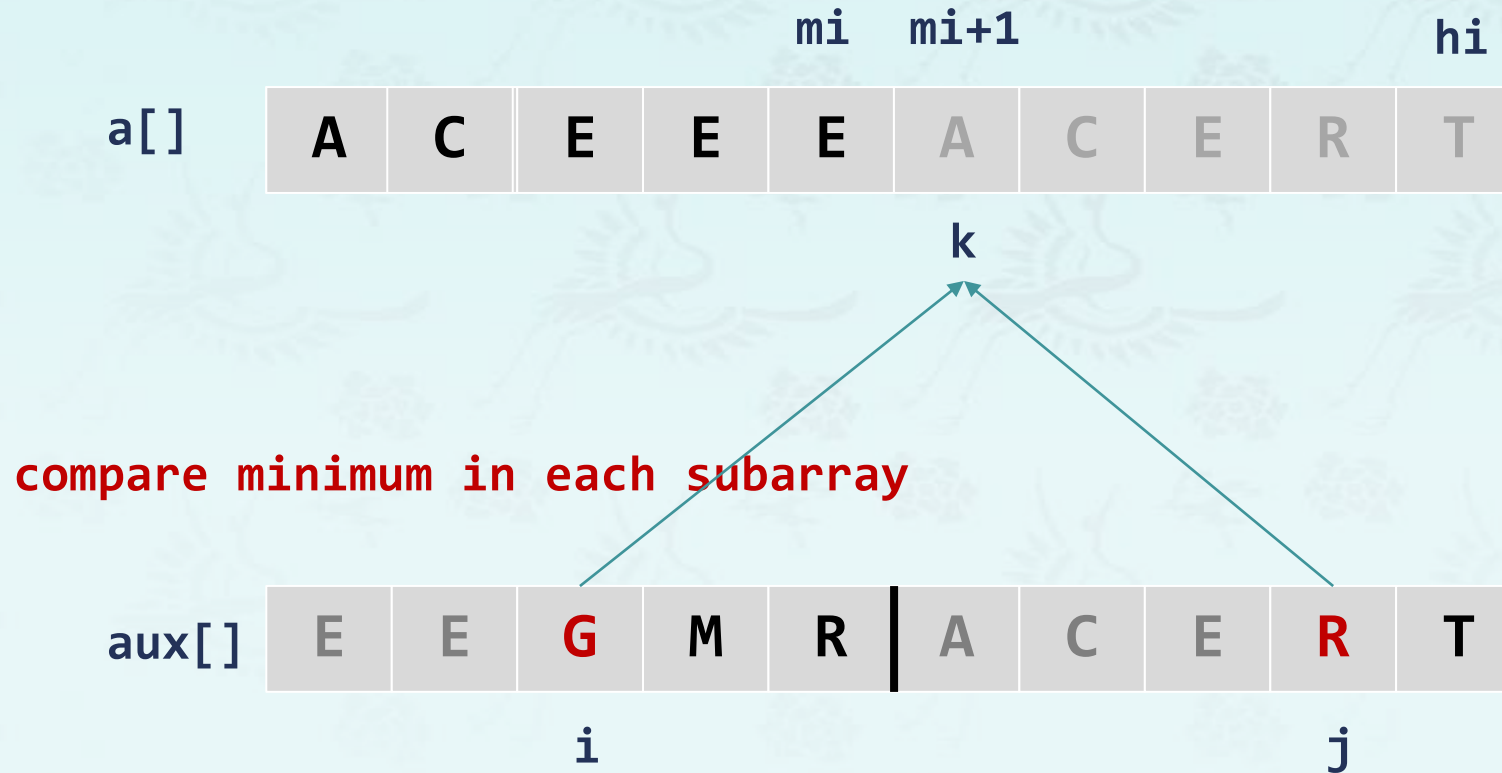
# Mergesort: merge

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



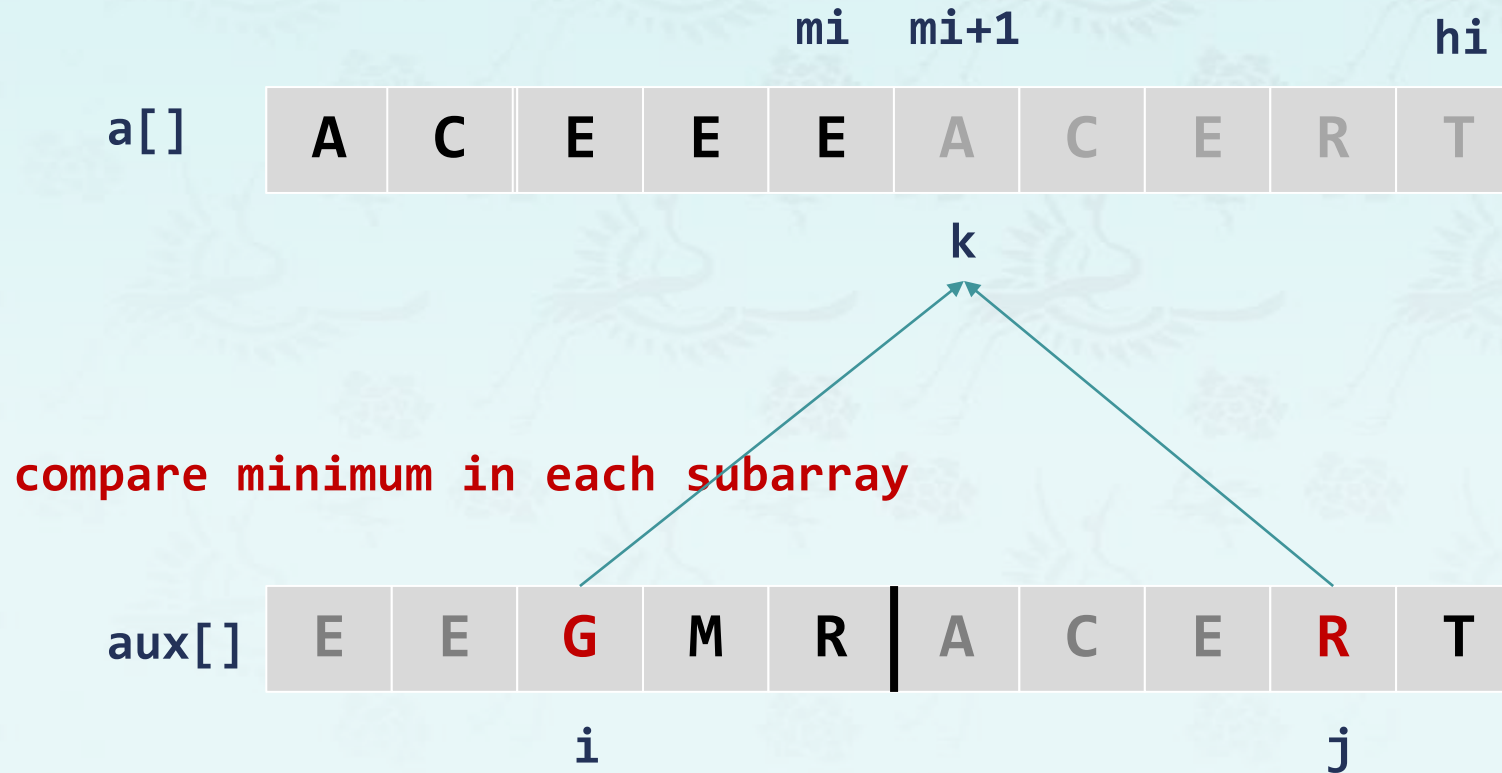
# Mergesort: merge

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



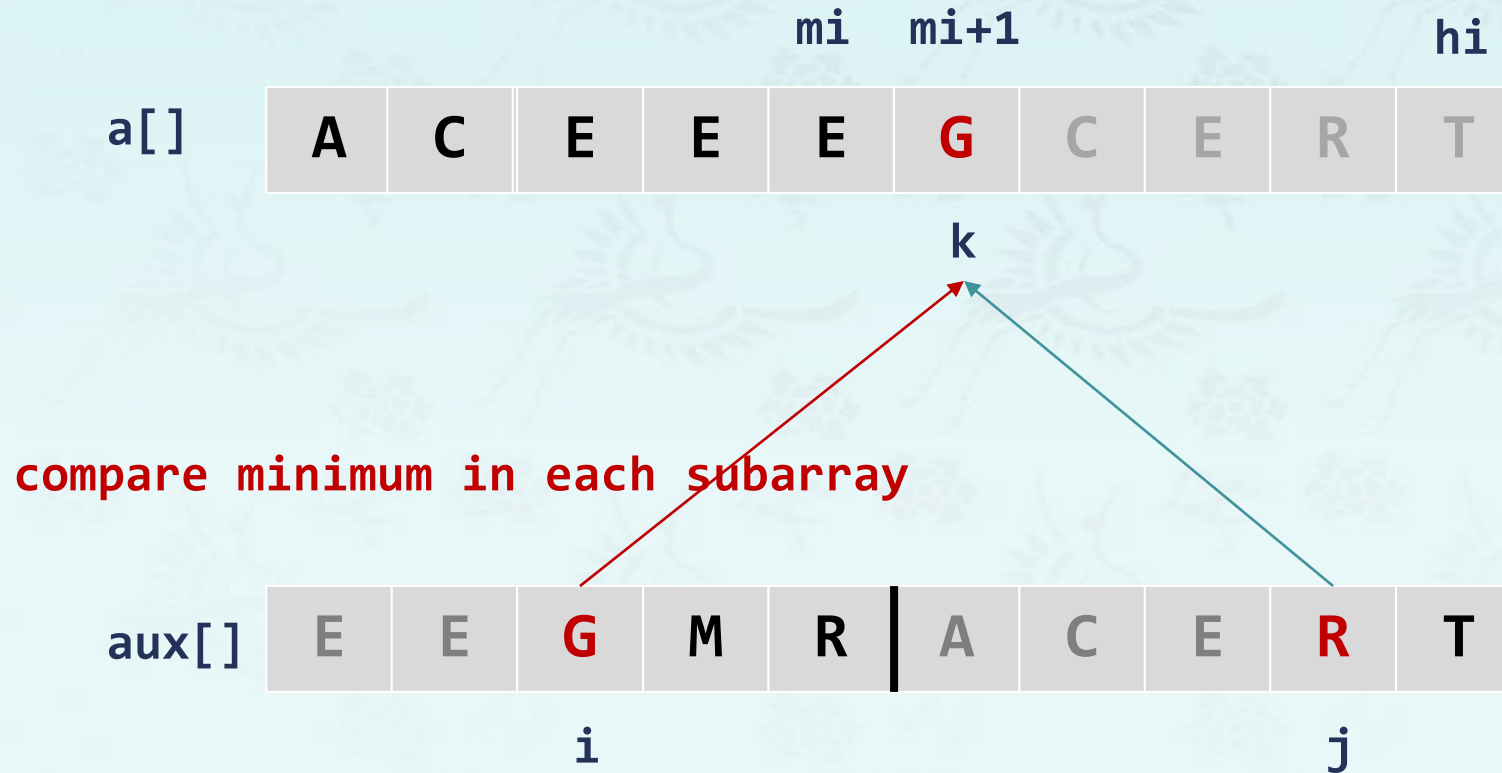
# Mergesort: merge

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



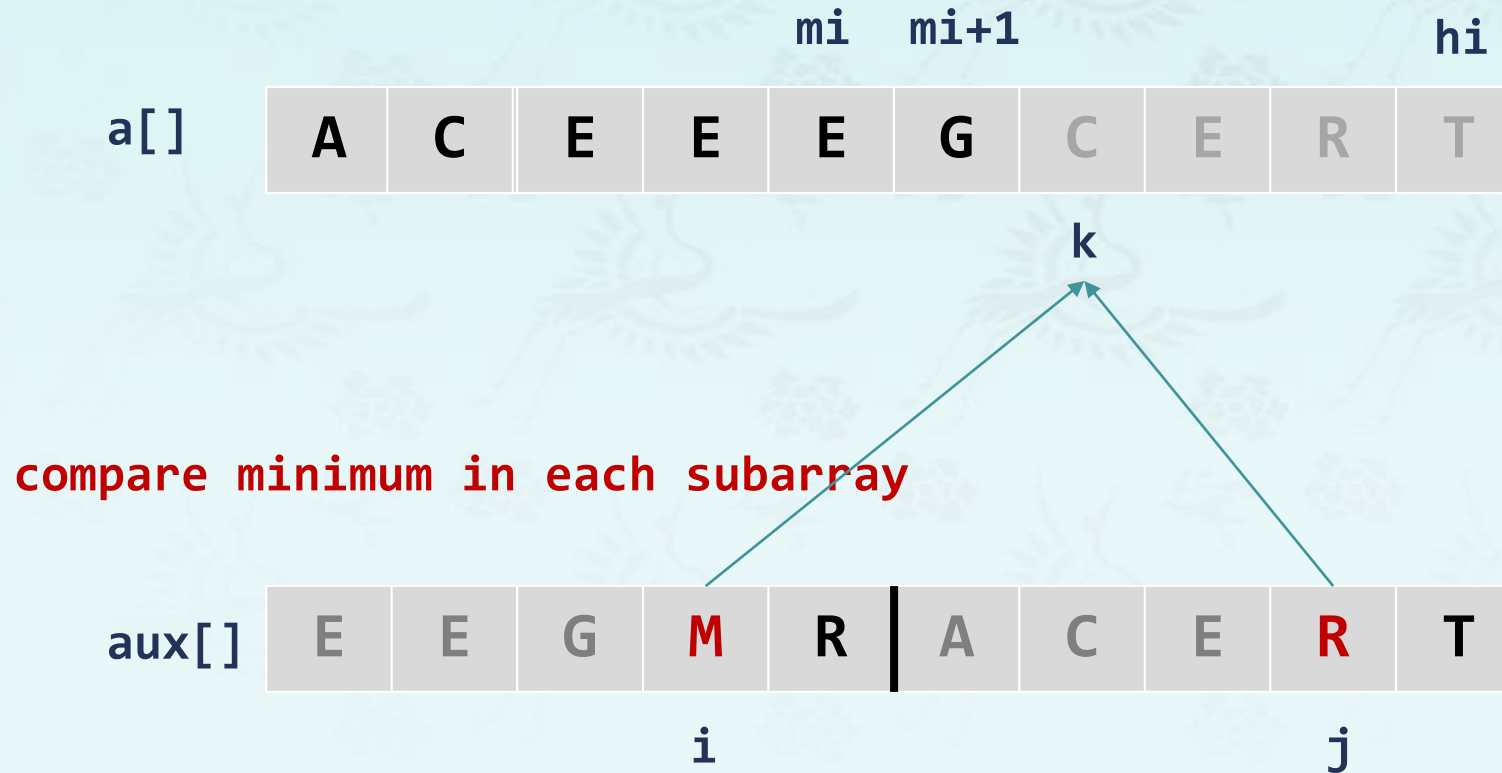
# Mergesort: merge

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



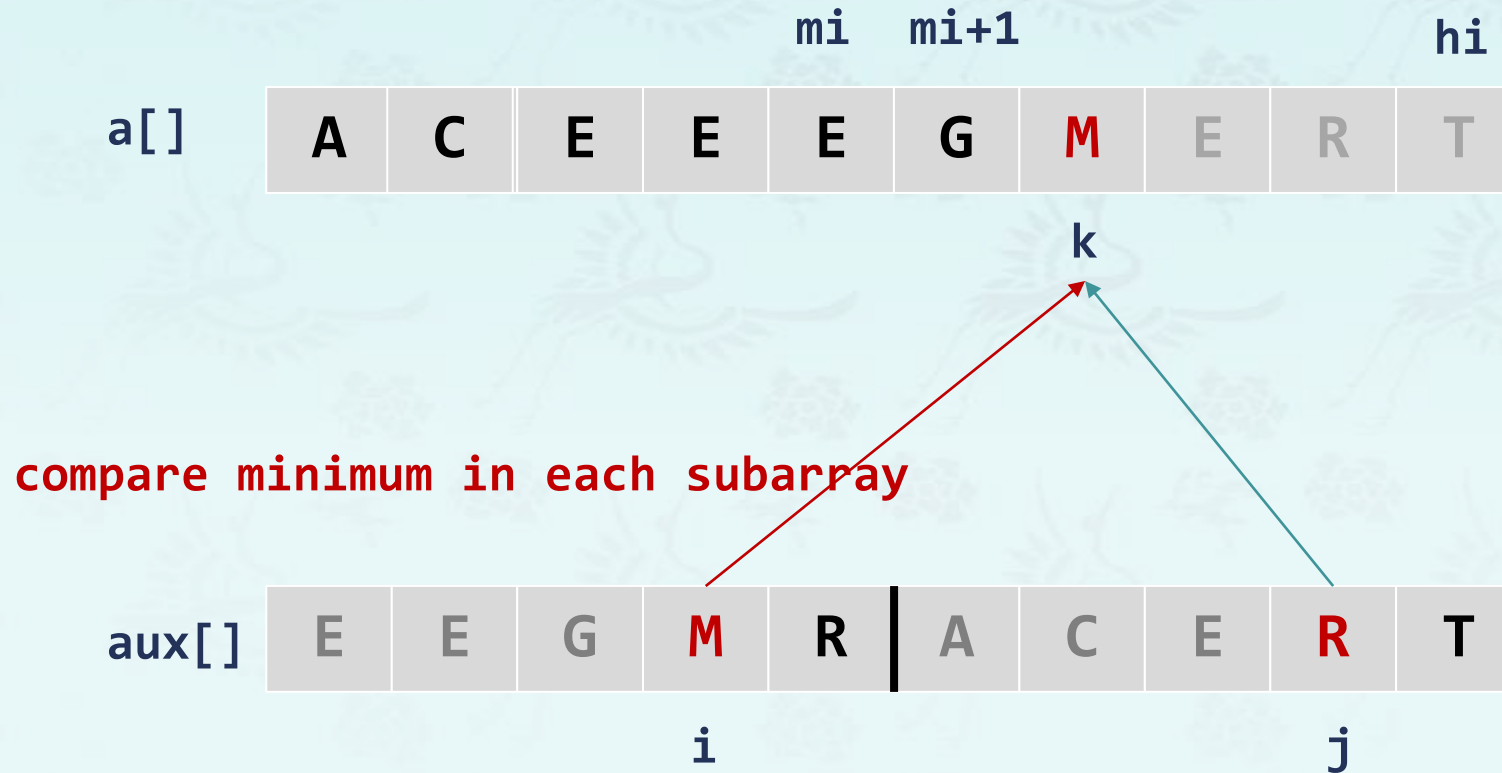
# Mergesort: merge

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



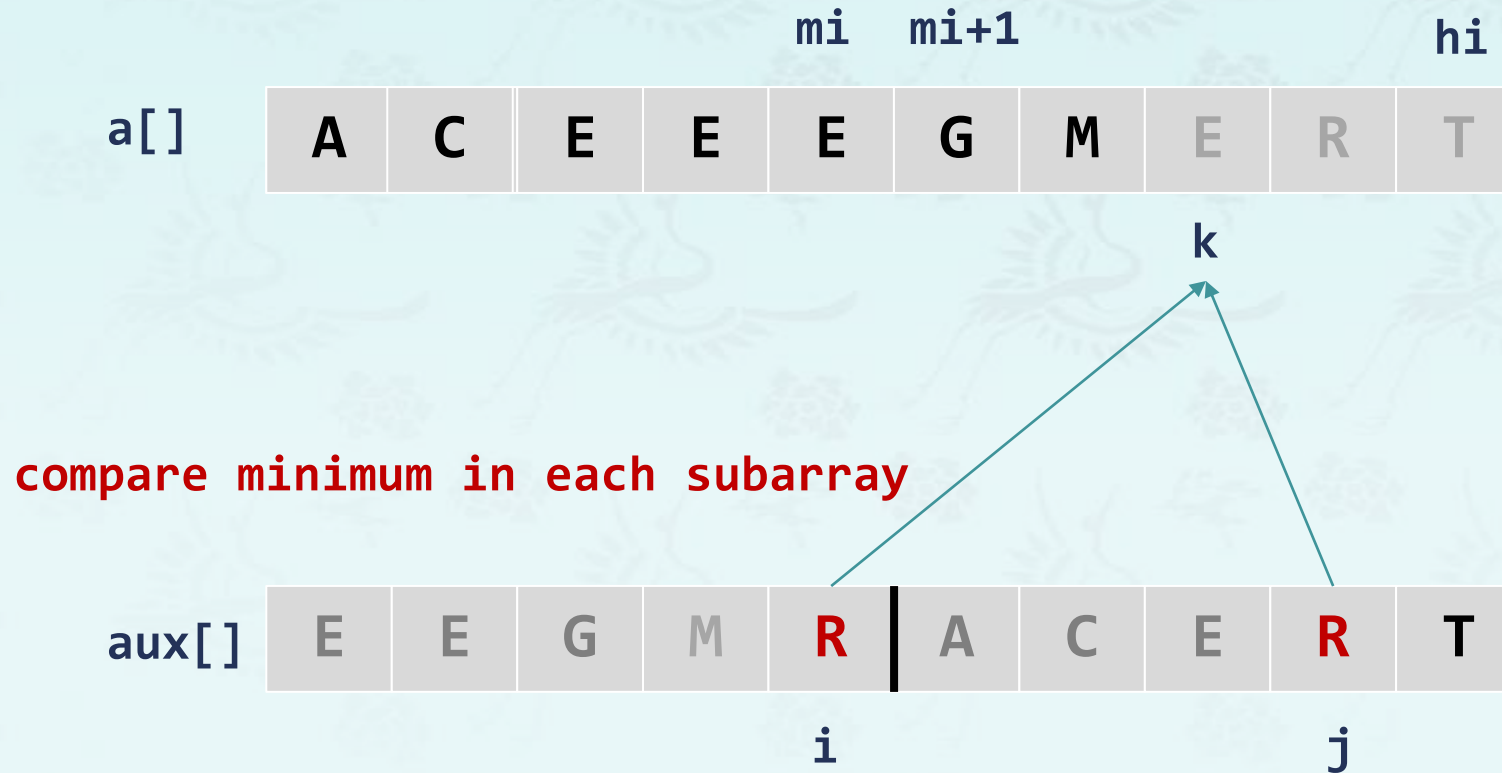
# Mergesort: merge

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



# Mergesort: merge

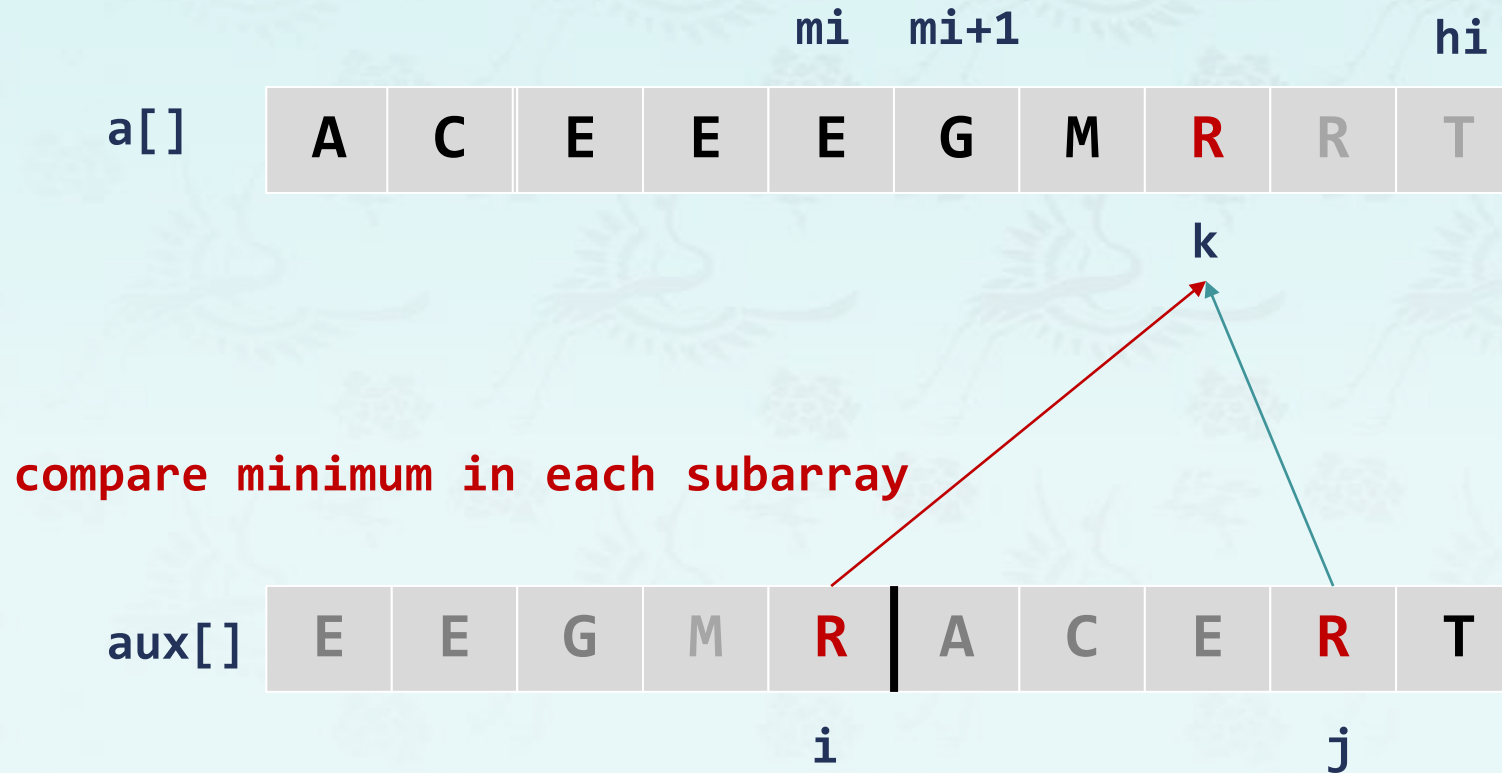
- **Goal:** Given two sorted subarrays  $a[lo]$  to  $a[mi]$  and  $a[mi+1]$  to  $a[hi]$ , replace with sorted subarray  $a[lo]$  to  $a[hi]$ .





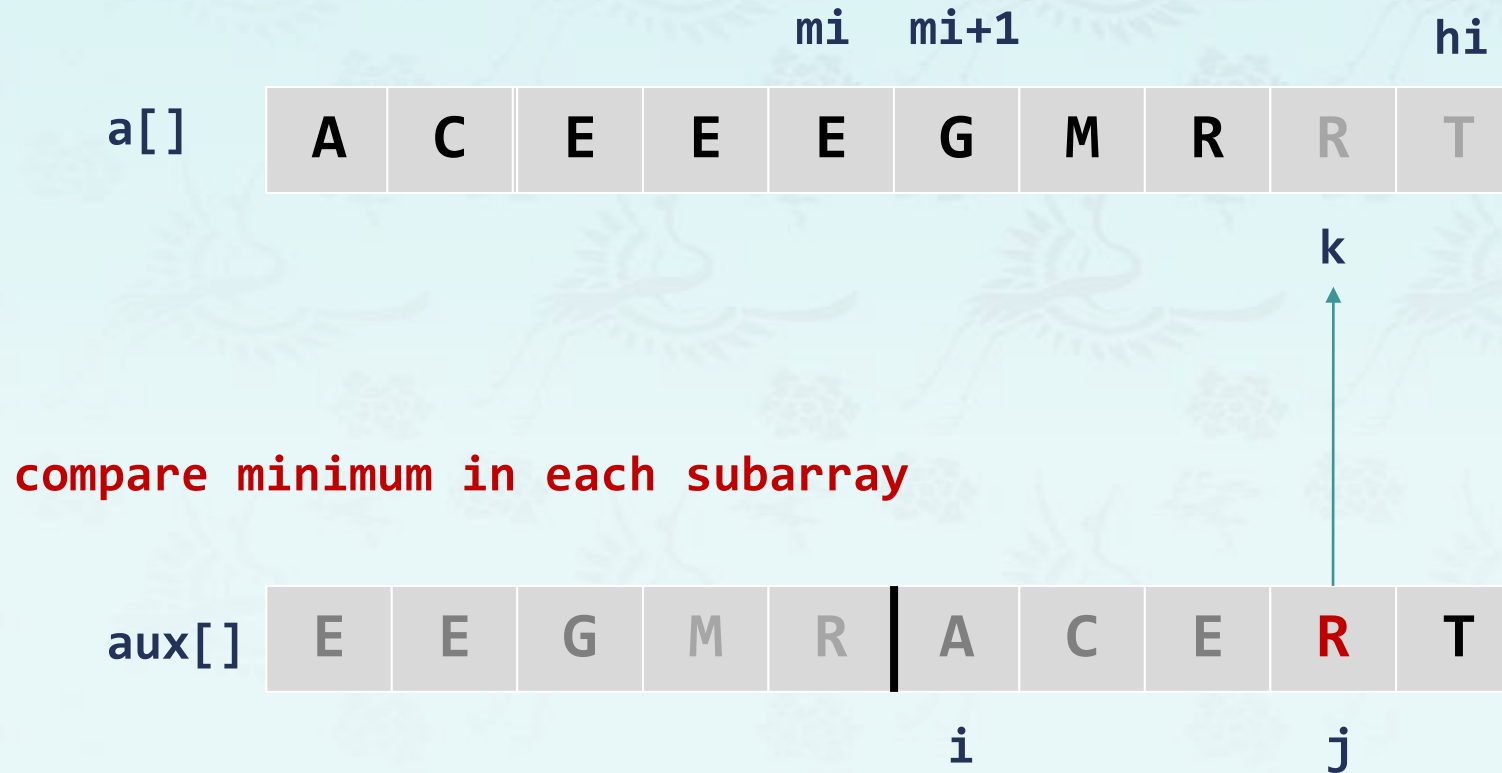
# Mergesort: merge

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



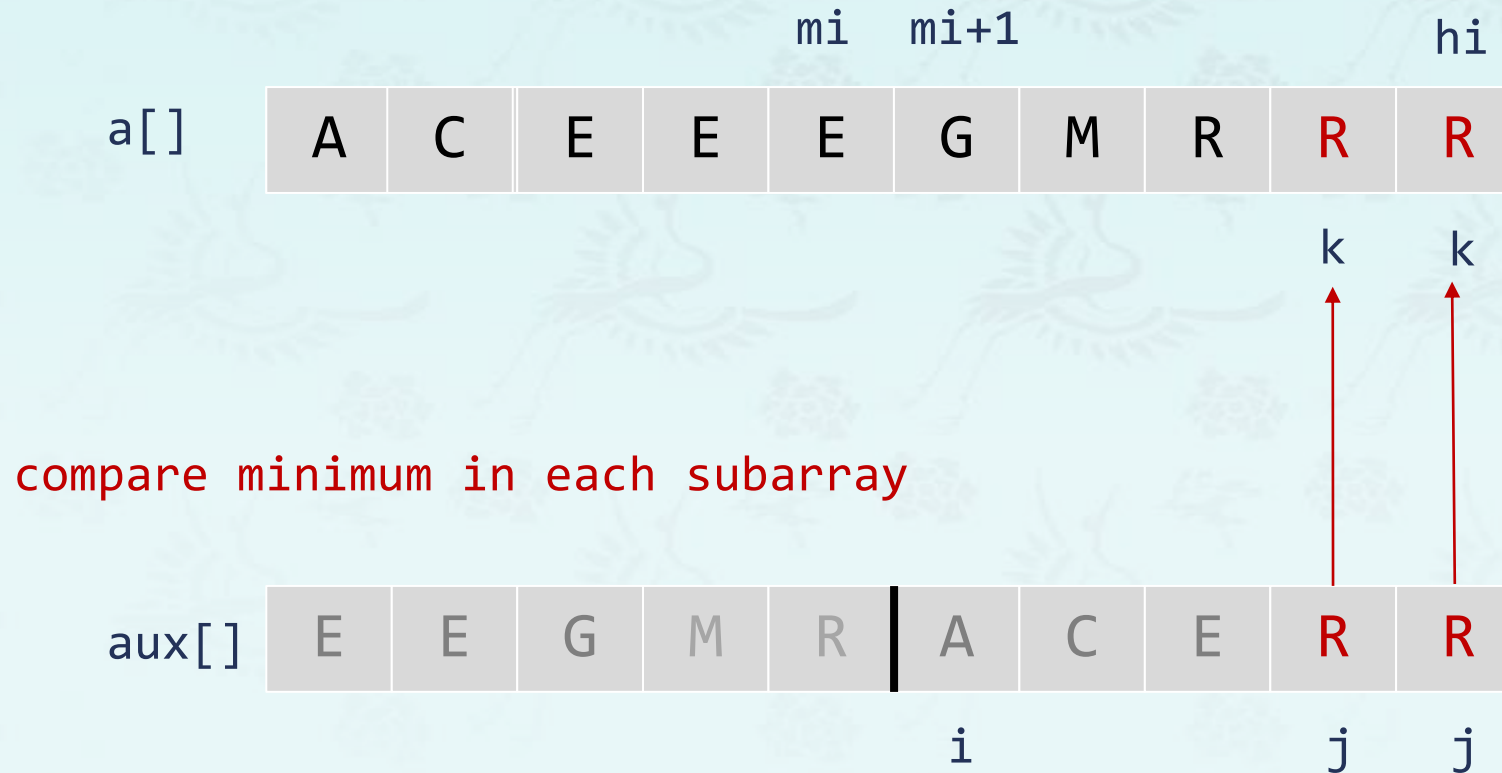
# Mergesort: merge

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



# Mergesort: 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  $a[hi]$ .



# Mergesort: merge

- **Goal:** Given two sorted subarrays  $a[\text{lo}]$  to  $a[\text{mi}]$  and  $a[\text{mi}+1]$  to  $a[\text{hi}]$ , replace with sorted subarray  $a[\text{lo}]$  to  $a[\text{hi}]$ .

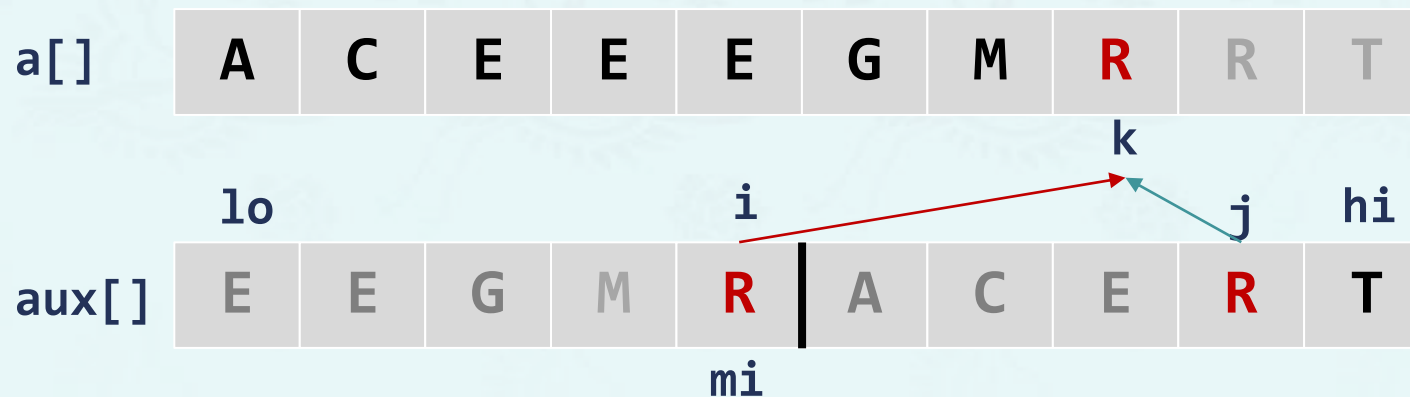
				$\text{mi}$	$\text{mi}+1$				$\text{hi}$
$a[]$	A	C	E	E	E	G	M	R	R

mergeSort complete using auxiliary array

$\text{aux}[]$	E	E	G	M	R		A	C	E	R	R
----------------	---	---	---	---	---	--	---	---	---	---	---

# Mergesort: 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**;
  - **merge** on the left half and **merge** on the right half;
    - **merge** the arrays by picking the smallest head element from the two sub-arrays until they are exhausted.



# Example 5: Recursive binary search

- For instance, we want to search "23" from the array. If we find it, we return its array index; otherwise, -1 or something else.

0	1	2	3	4	5	6	7	8	9
2	5	8	9	16	23	31	56	62	71

lo=0	1	2	3	mi=4	5	6	7	8	hi=9
2	5	8	9	16	23	31	56	62	71

0	1	2	3	4	lo=5	6	mi=7	8	hi=9
2	5	8	9	16	23	31	56	62	71

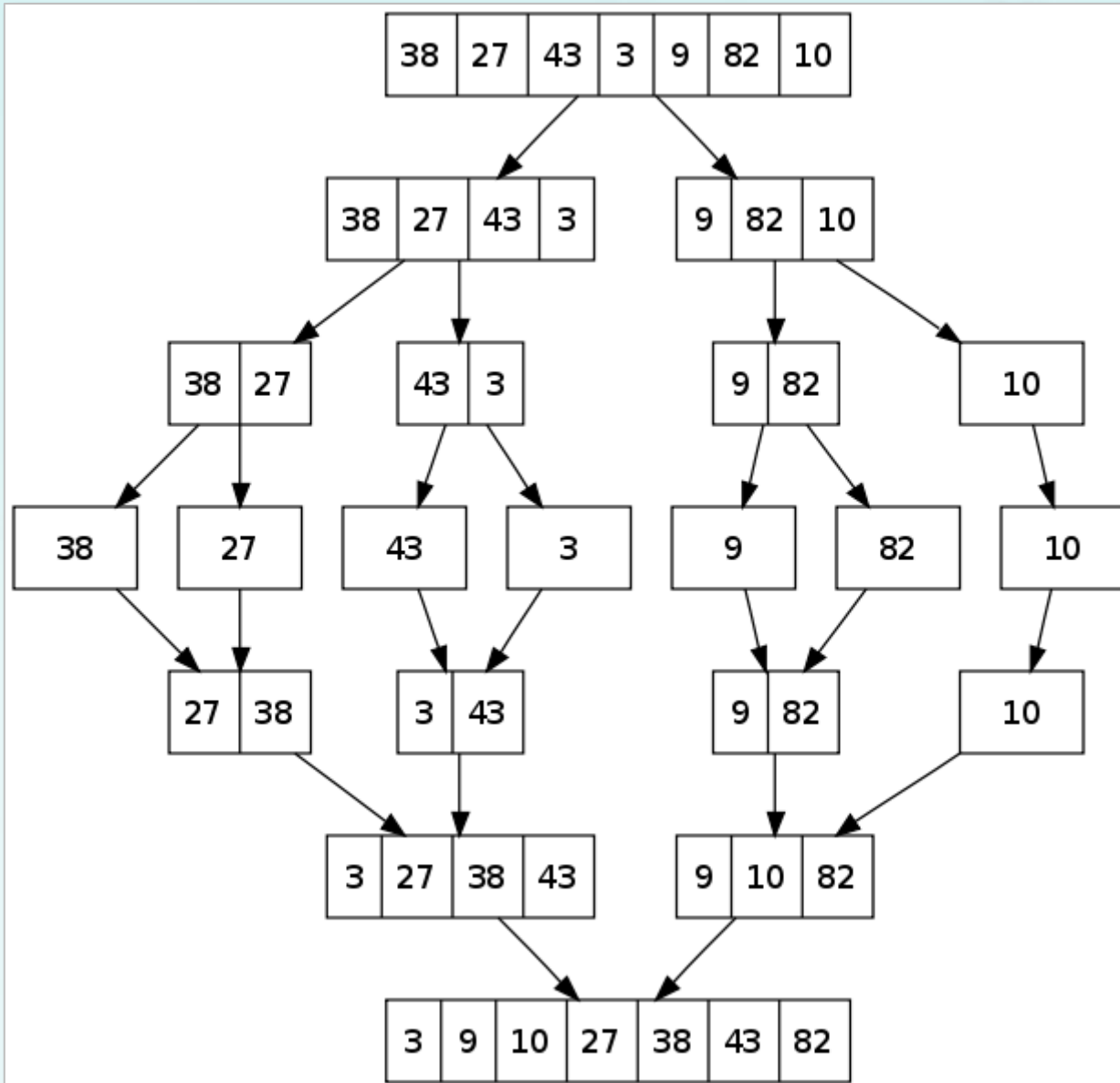
0	1	2	3	4	mi=5 lo=5	hi=6	7	8	9
2	5	8	9	16	23	31	56	62	71

```
int binarySearch(int list[], int key,
                 int lo, int hi) {
    if (lo > hi) return -1;

    mi = (lo + hi)/2;
    if (key == list[mi]) return mi;
    if (key < list[mi])
        return binarySearch(list, key, lo, mi - 1);
    else
        return binarySearch(list, key, mi + 1, hi);
}
```



# Mergesort: Coding



5 1 7 3 2 8 6 4

merge each sorted list together with its neighbor — maintaining sorted order

1 5 3 7 2 8 4 6

1 3 5 7 2 4 6 8

continue merging sublists until...

1 2 3 4 5 6 7 8

There is only one part left:  
The sorted list, merged together!

# Mergesort: Coding

```
mergeSort(a[], aux[], N, lo, hi)
```

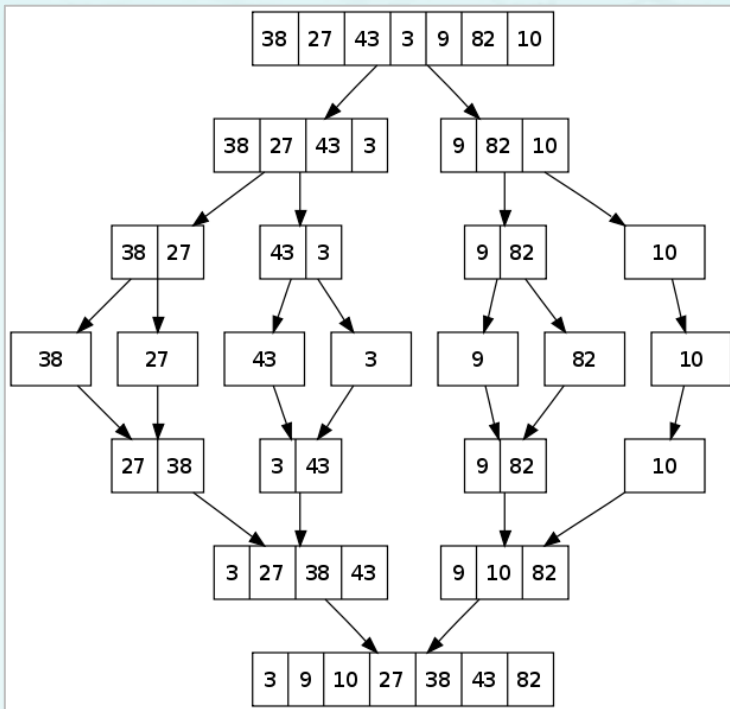
If  $hi > lo$

Find the middle to divide the array into two:  $mi = (lo+hi)/2$

Split 1st half: `mergeSort(a, aux, N, lo, mi)`

Split 2nd half: `mergeSort(a, aux, N, mi+1, hi)`

Merge the two halves sorted: `merge(a, aux, lo, mi, hi)`



```
mergeSort(a[], n) {
    ...
    mergeSort(a, aux, N, 0, N-1);
}
```



# Mergesort: Coding

```
void mergeSort(char *a, char *aux, int N, int lo, int hi) {
    if (hi <= lo) return;
    int mi = lo + (hi - lo) / 2;                // mi=(lo+hi)/2
    mergeSort (a, aux, N, lo,      mi);
    mergeSort (a, aux, N, mi + 1, hi);
    merge(a, aux, lo, mi, hi);
}

int main() {
    char a[]={'M','E','R','G','E','S','O','R','T','E','X','A','M','P','L','E'};
    cout << "UNSORTED: "; for (auto x: a) cout << x; cout << endl;
    int N = sizeof(a) / sizeof(a[0]);
    char *aux = new char[N];
    mergeSort(a, aux, N, 0, N - 1);
    cout << "    SORTED: "; for (auto x: a) cout << x; cout << endl;
}
```

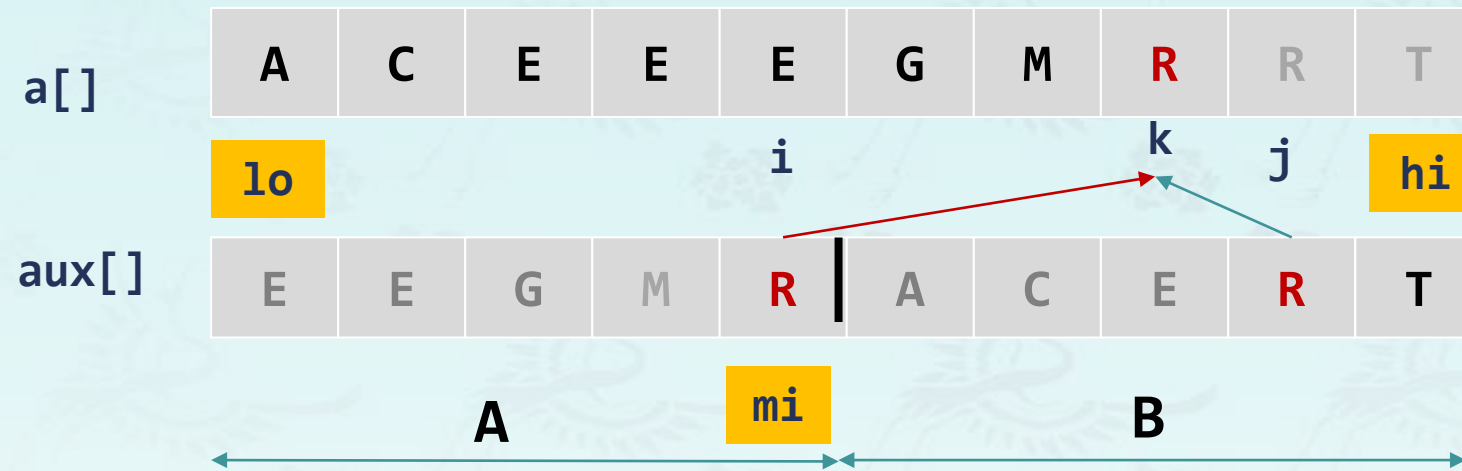
# Mergesort: Coding

```
int isSorted(char *a, int i, int j){return a[i] <= a[j];}

void merge(char *a, char *aux, int lo, int mi, int hi) {
    assert(isSorted(a, lo,  mi));    // precondition: a[lo..mi]    sorted
    assert(isSorted(a, mi+1, hi));    // precondition: a[mi+1..hi] sorted
    for (int k = lo; k <= hi; k++) aux[k] = a[k];

    int i = lo;
    int j = mi + 1;
    for (int k = lo; k <= hi; k++) {
        if      (i > mi)          a[k] = aux[j++];        // A is exhausted, take B[j]
        else if (j > hi)          a[k] = aux[i++];        // B is exhausted, take A[i]
        else if (aux[j] < aux[i]) a[k] = aux[j++];        // B[j] < A[i], take B[j]
        else                     a[k] = aux[i++];        // A[i] <= B[j], take A[i]
    }
    assert(isSorted(a, lo, hi));    // postcondition: a[lo..hi] sorted
}
```

# Mergesort: Coding

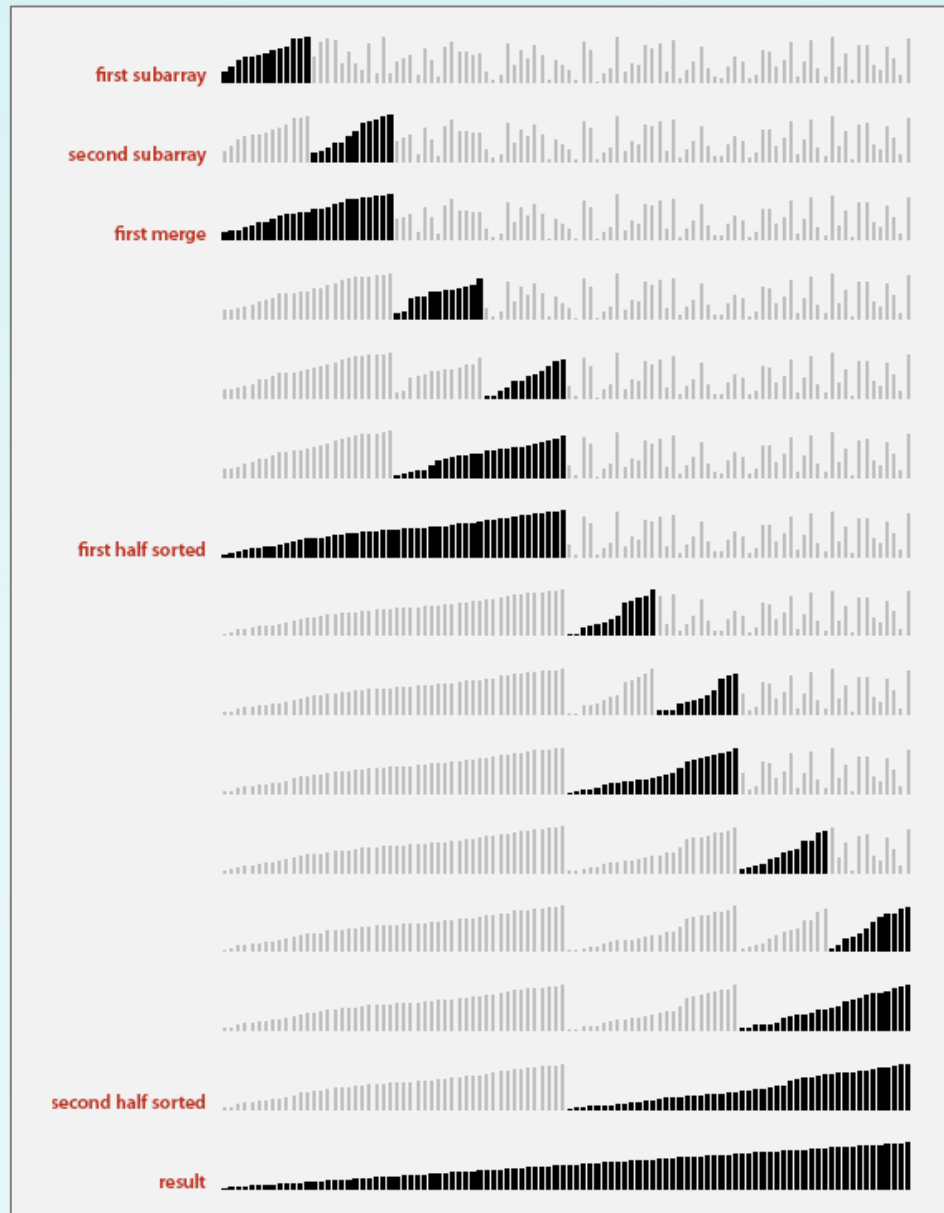


# Mergesort: Coding

	lo	hi	a[]															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)			E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)			E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)			E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)			E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)			E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)			E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)			E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)			E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)			E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)			E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)			E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)			E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)			E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)			E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)			A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

result after recursive call

# Mergesort: Coding



<https://algs4.cs.princeton.edu/22mergeSort/>

# Assertion in C/C++

---

- **Assertion:** Statement to test assumptions about your program in Java.
  - Helps detect logic bugs.
  - Documents code.
- **Assert statement:** abort the program and print an error message (the function name and a line number) unless Boolean condition is true.

```
#include <cassert>
assert( isSorted(a, lo, hi) );
```

- **Best practices:** Use assertions to check internal invariants;
  - Assume assertions will be disabled in production code.
  - Do not use for external argument checking.

# Mergesort: Quiz 1

---

1. Improvement by reducing the number of `merge()` function call.  
Some hints for this problem are provided in the following pages.
2. How many times did you spare `merge()` calls for "MERGESORTEXAMPLE" case?
  - Total number of **merge()** calls without your improvement: \_\_\_\_\_
  - The number of **merge()** calls spared with your improvement: \_\_\_\_\_
3. Identify those sets of char array groups that `merge()` call was unnecessary.

# Mergesort: Quiz 1

- **Hint:** Do not invoke "merge()" function **if two halves are already sorted..**
  - Is the biggest item in first half  $\leq$  the smallest item in second half?
  - For example, the following case **should not call merge()** since  $J \leq M$ .

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

정렬들에 관한 좋은 자료를 읽어 보길 적극 추천합니다.

영어: <https://medium.com/basics/making-sense-of-merge-sort-part-1-49649a143478>

한글: <https://gmlwjd9405.github.io/2018/05/08/algorithm-merge-sort.html>



# Mergesort: Quiz 1

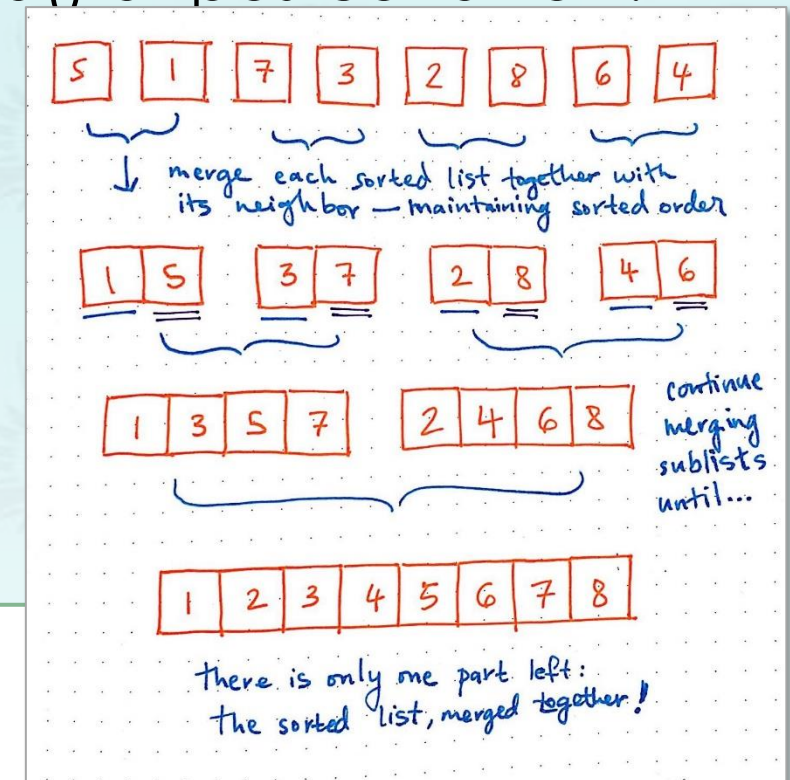
- **Hint:** Do not invoke "merge()" function **if two halves are already sorted..**
  - Is the biggest item in first half  $\leq$  the smallest item in second half?
  - For example, the following case **should not call merge()** since  $J \leq M$ .

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

```
void mergeSort(char *a, char *aux, int N, int lo, int hi) {  
    if (hi <= lo) return;  
    int mi = lo + (hi - lo) / 2;  
    mergeSort(a, aux, N, lo, mi);  
    mergeSort(a, aux, N, mi + 1, hi);  
    // your code here // already sorted  
    merge(a, aux, lo, mi, hi);  
}
```

# Mergesort: Quiz 2

- In the figure, which elements are compared in `isSorted()` at postcondition?
- Why `isSorted()` checks only two elements?  
Is this enough?

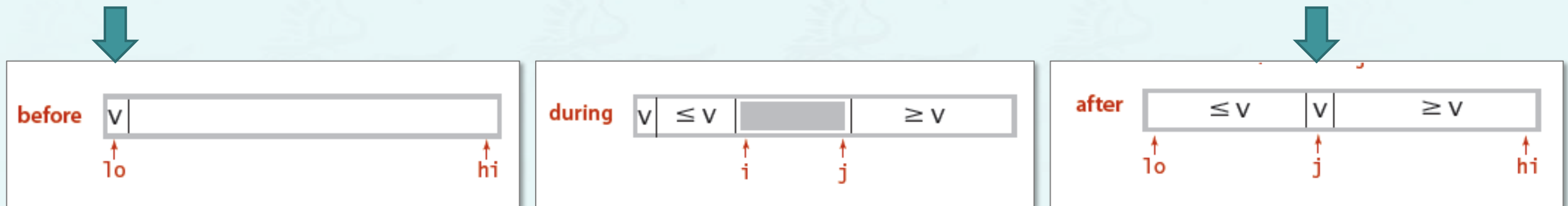


```
int isSorted(int *a, int i, int j){return a[i] <= a[j];}
```

```
void merge(int *a, char *aux, int lo, int mi, int hi) {  
    assert(isSorted(a, lo, mi));    // precondition: a[lo..mi] sorted  
    assert(isSorted(a, mi+1, hi));  // precondition: a[mi+1..hi] sorted  
    for (int k = lo; k <= hi; k++) aux[k] = a[k];  
    .....  
    assert(isSorted(a, lo, hi));    // postcondition: a[lo..hi] sorted  
}
```

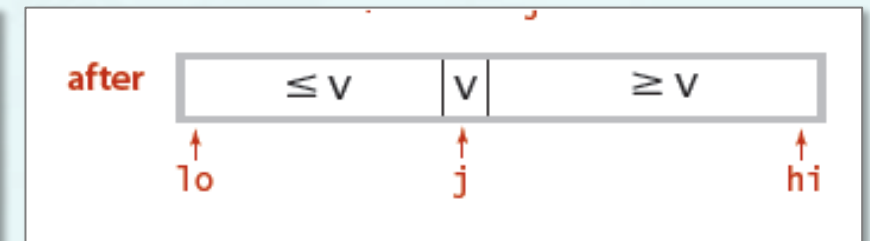
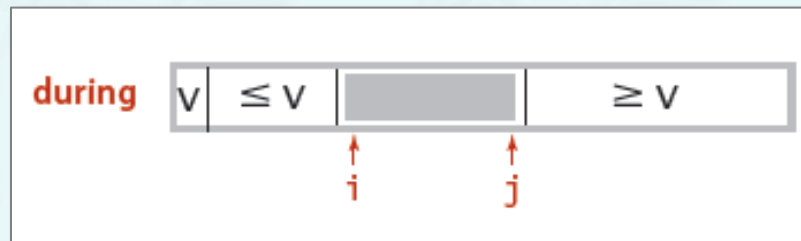
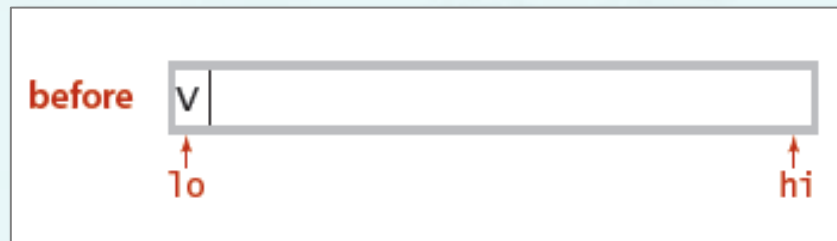
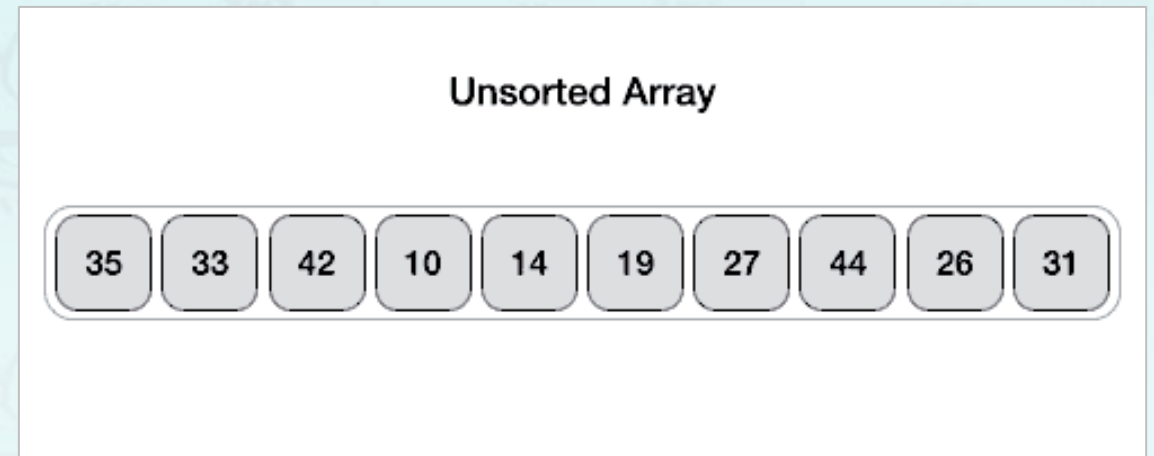
# Quick Sort(퀵 정렬)

- Quick Sort is a divide-and-conquer algorithm.
- It works by selecting a '**pivot**' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively.



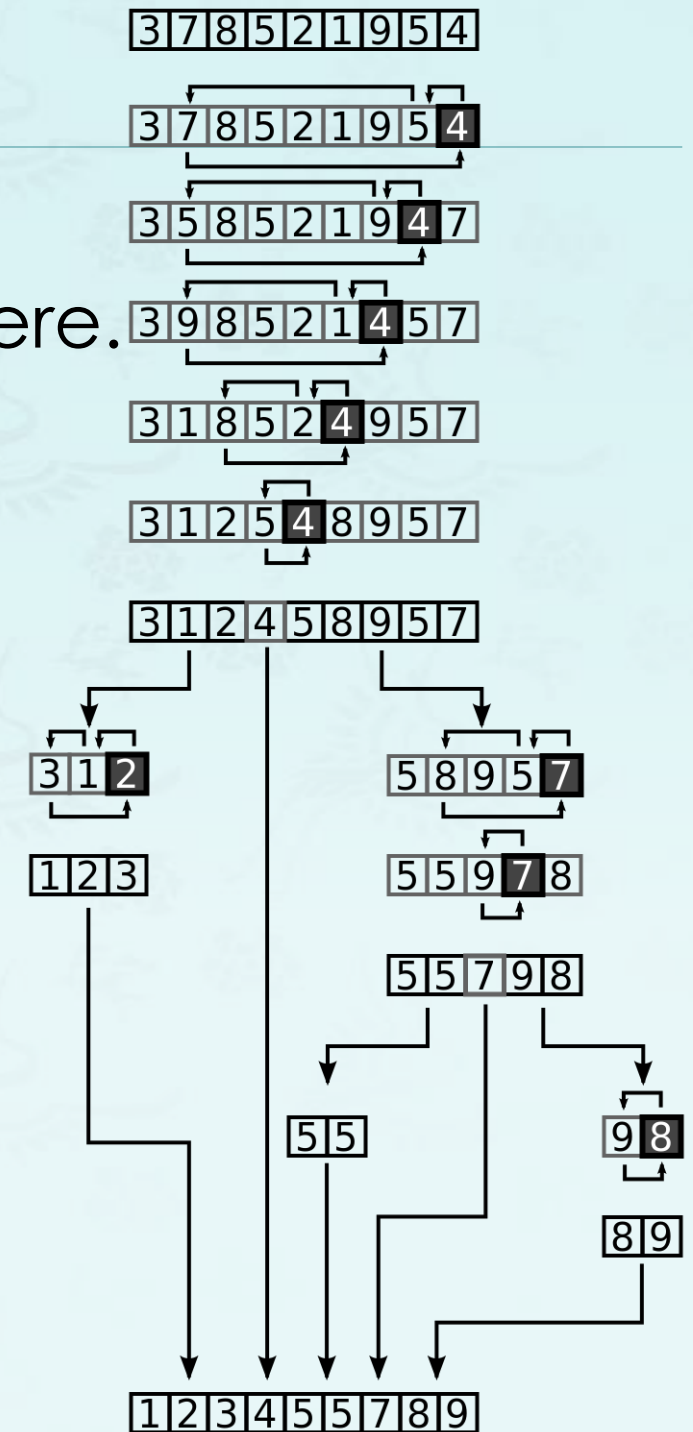
# Quick Sort(퀵 정렬)

- Quick Sort is a divide-and-conquer algorithm.
- It works by selecting a '**pivot**' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively.



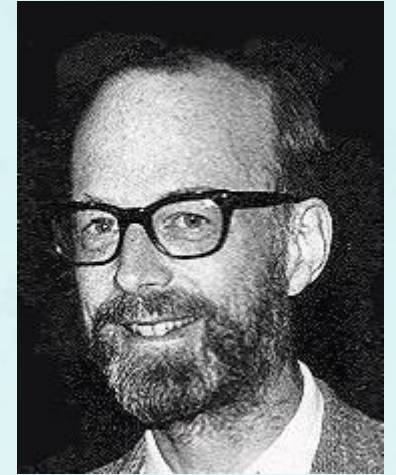
## Quick Sort(퀵 정렬)

- The shaded element is the pivot.
- It is chosen as the last element of the partition here.



## 4. Quick Sort - by Hoare in 1961

- Algorithm:
  - Shuffle the array.
  - Partition so that, for some  $j$ 
    - entry  $a[j]$  is in place
    - no larger entry to the left of  $j$
    - no smaller entry to the right of  $j$
  - Sort each piece recursively.

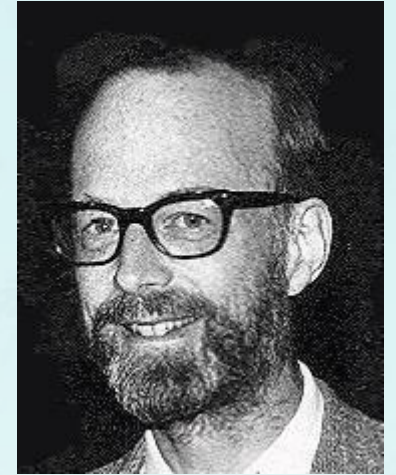


Sir Charles Antony Richard Hoare  
1980 Turing award



## 4. Quick Sort - by Hoare in 1961

- Algorithm:
  - Shuffle the array.
  - Partition so that, for some  $j$ 
    - entry  $a[j]$  is in place
    - no larger entry to the left of  $j$
    - no smaller entry to the right of  $j$
  - Sort each piece recursively.



Sir Charles Antony Richard Hoare  
1980 Turing award

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

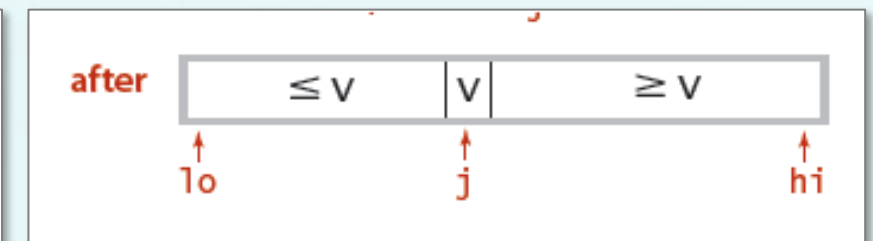
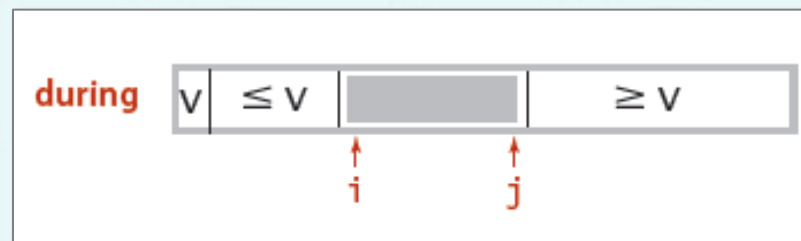
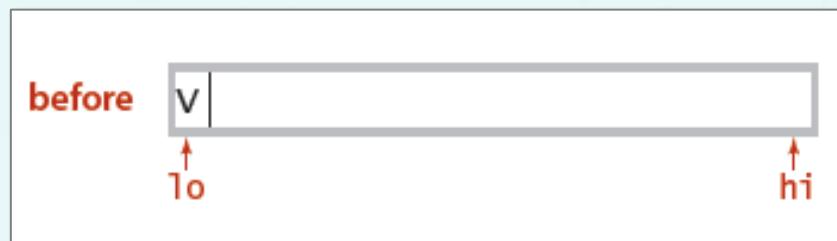
<https://algs4.cs.princeton.edu/23quicksort/>

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .

pivot partitioning element

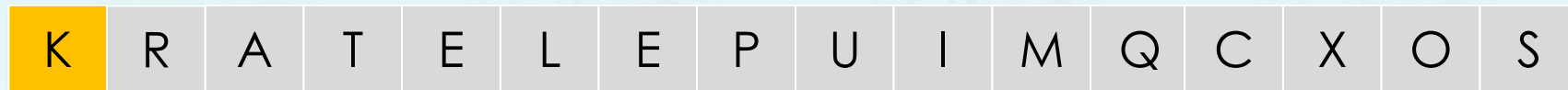
A diagram showing a sequence of 15 letters in a row: K, R, A, T, E, L, E, P, U, I, M, Q, C, X, O, S. The letter 'K' is highlighted in a yellow box. Below the letters, there is a red 'lo' at the beginning and a red 'j' at the end, indicating a range from index 0 to index 14.





# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



lo    i     $\longrightarrow$

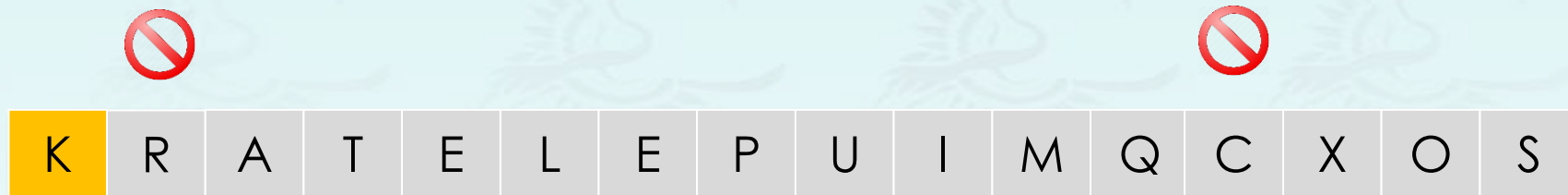
$i$  pointer moves from left to right  
as long as it is less than the pivot

$\longleftarrow$     j

moves from right to left

- Phase I. Repeat until  $i$  and  $j$  pointers cross:

- Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
- Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
- Exchange  $a[i]$  with  $a[j]$ .



i

i pointer moves from left to right  
as long as it is less than the pivot

i pointer stops immediately since ....

i

i

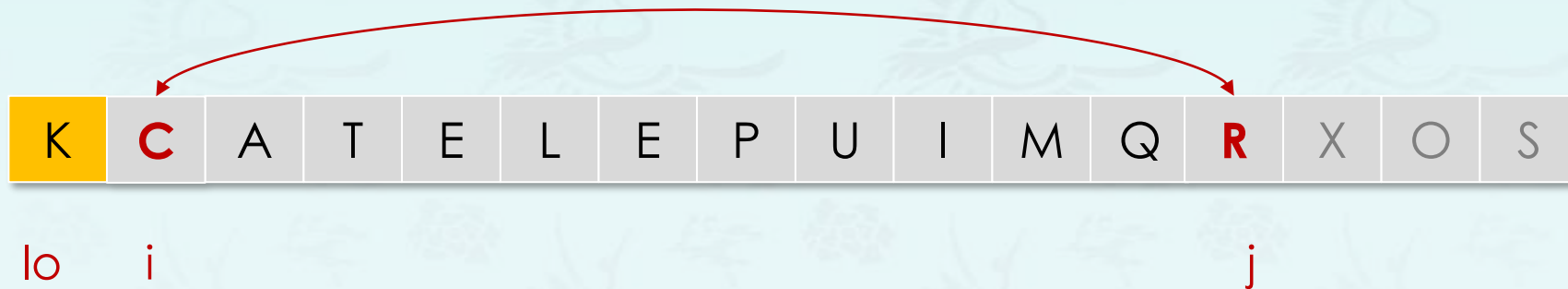
moves from right to left

now decrement  $j$  until ....

j pointer moves and stops at "C"

# Quick Sort partitioning demo

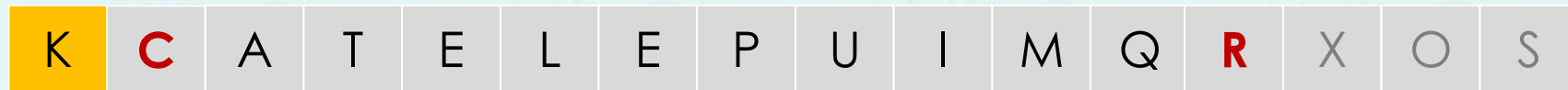
- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



stop scan and exchange  $a[i]$  with  $a[j]$

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



lo    i →

← j

now increment  $i$  until ....

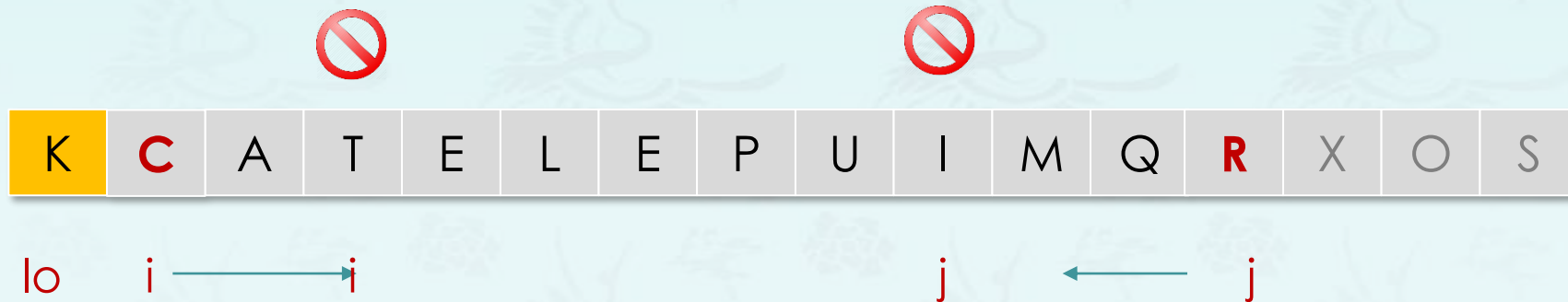
now decrement  $j$  until ....

stop  $i$  scan because  $a[i] \geq a[lo]$

pivot

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



now increment  $i$  until ....

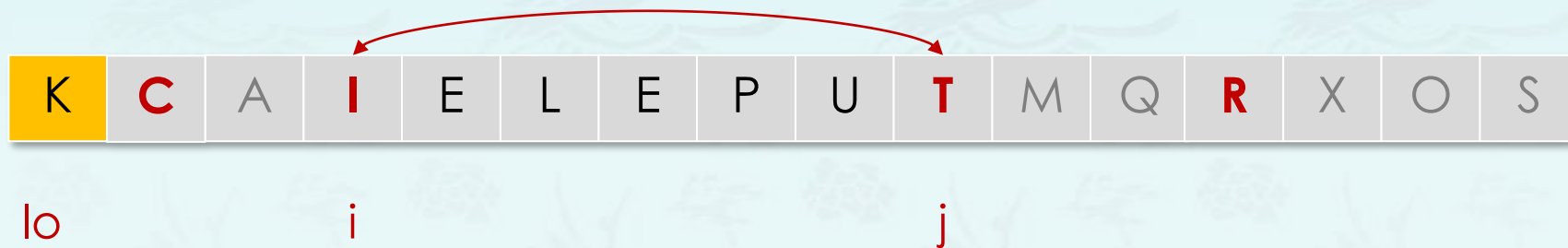
stop  $i$  scan because  $a[i] \geq a[lo]$

pivot

now decrement  $j$  until ....

# Quick Sort partitioning demo

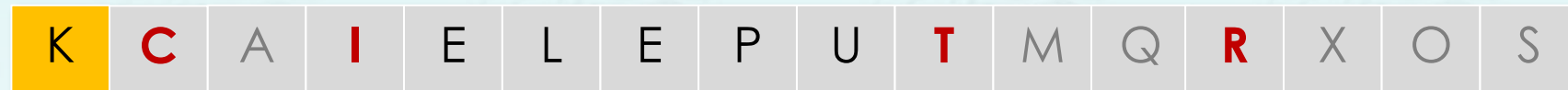
- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



stop scan and exchange  $a[i]$  with  $a[j]$

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



lo                      i                      j

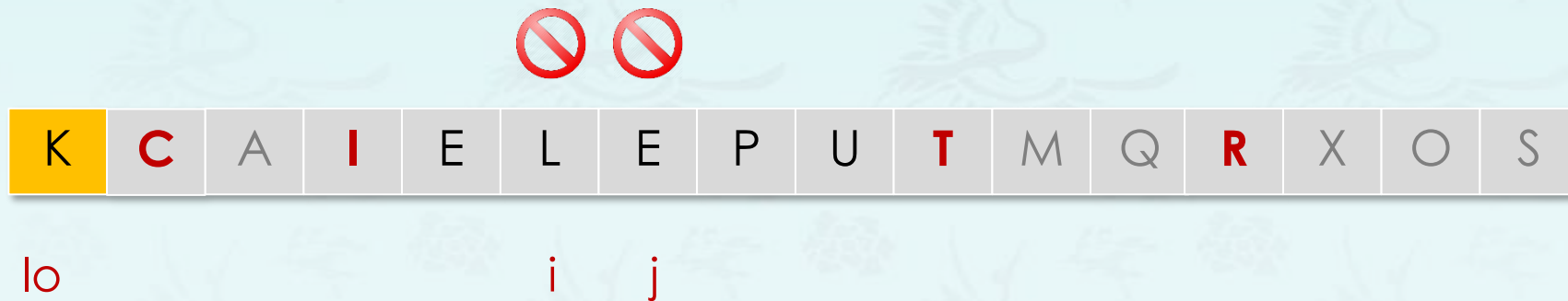
now increment i until ....

now decrement j until ....



# Quick Sort partitioning demo

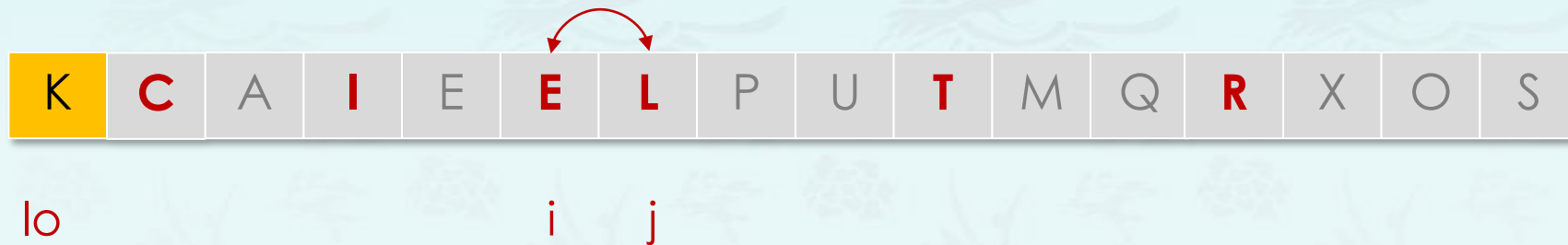
- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .





# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



stop scan and exchange  $a[i]$  with  $a[j]$

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



lo

i

j

now increment i until ....

now decrement j until ....

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .



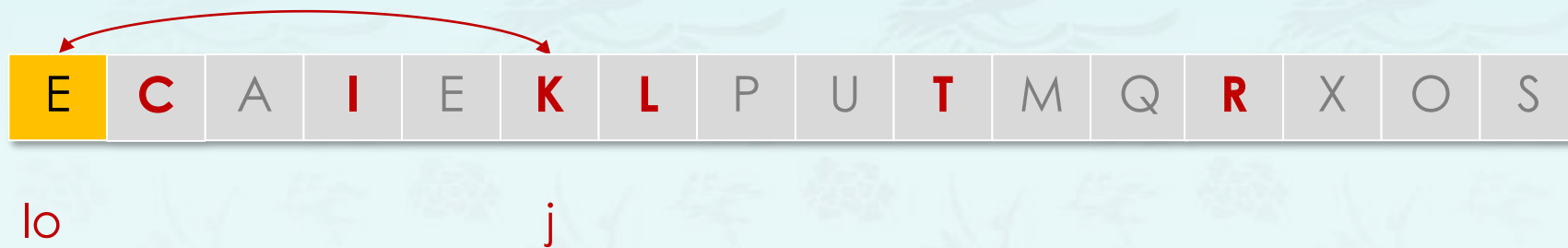
stop  $j$  scan because  $a[j] \leq a[lo]$

now  $j$  points the last element of left subarray

at this point, partitioning process is **complete!**

# Quick Sort partitioning demo

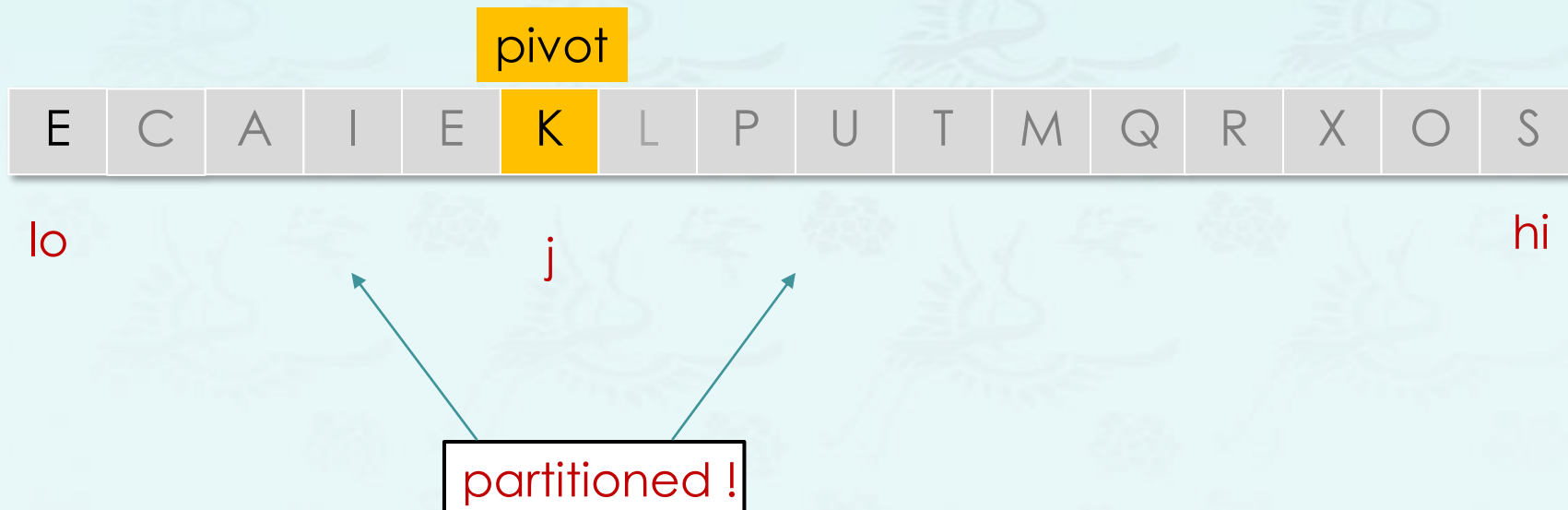
- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .
- **Phase II. When pointers cross.**
  - Exchange  $a[lo]$  with  $a[j]$ .



at this point, partitioning process is complete!

# Quick Sort partitioning demo

- Phase I. Repeat until  $i$  and  $j$  pointers cross:
  - Scan  $i$  from left to right so long as  $(a[i] < a[lo])$
  - Scan  $j$  from right to left so long as  $(a[j] > a[lo])$ .
  - Exchange  $a[i]$  with  $a[j]$ .
- **Phase II. When pointers cross.**
  - Exchange  $a[lo]$  with  $a[j]$ .



at this point, partitioning process is complete!

# Quick Sort implementation

```
bool less(char a, char b) { return a < b; }
```

```
void swap(char *a, int i, int j) { char t = a[i]; a[i] = a[j]; a[j] = t; }
```

```
int partition(char *a, int lo, int hi) {
```

```
    int i = lo; int j = hi + 1;
```

```
    while (1) {
```

```
        while (less(a[++i], a[lo]))
```

```
            if (i == hi) break;
```

```
        while (less(a[lo], a[--j]))
```

```
            if (j == lo) break;
```

```
        if (i >= j) break;
```

```
        swap(a, i, j);
```

```
    }
```

```
    swap(a, lo, j);
```

```
    return j;
```

```
}
```

pivot

find item on left to swap

find item on right to swap

check if pointers cross  
swap

swap with pivot  
return index of item now sorted

before



during



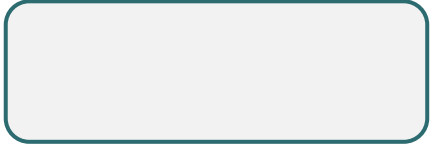
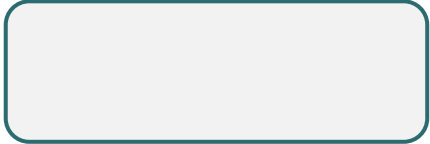
after



# Quick Sort implementation

```
void quicksort(char *a, int lo, int hi) {
    if (hi <= lo) return;

    int j = partition(a, lo, hi);

    quicksort(a, 
    quicksort(a, 
}

void main() {
    char a[] = {'Q', 'U', 'I', 'C', 'K', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
    int N = sizeof(a) / sizeof(a[0]);

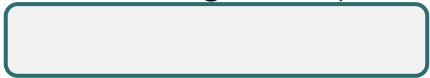
    cout << "UNSORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
    // shuffle(a, N);

    quicksort(a, 0, N-1);
    cout << "SORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
}
```

# Quick Sort implementation

```
void quicksort(char *a, int lo, int hi) {
    if (hi <= lo) return;

    int j = partition(a, lo, hi);

    quicksort(a, lo, j - 1);
    quicksort(a, 
}

void main() {
    char a[] = {'Q', 'U', 'I', 'C', 'K', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
    int N = sizeof(a) / sizeof(a[0]);

    cout << "UNSORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
    // shuffle(a, N);

    quicksort(a, 0, N-1);
    cout << "SORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
}
```



# Quick Sort implementation

```
void quicksort(char *a, int lo, int hi) {
    if (hi <= lo) return;

    int j = partition(a, lo, hi);

    quicksort(a, lo,    j - 1);
    quicksort(a, j + 1, hi    );
}

void main() {
    char a[] = {'Q', 'U', 'I', 'C', 'K', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
    int N = sizeof(a) / sizeof(a[0]);

    cout << "UNSORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
    //  shuffle(a, N);

    quicksort(a, 0, N-1);
    cout << "SORTED: \n";
    for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
}
```

# Quick Sort implementation

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

<https://algs4.cs.princeton.edu/23quicksort/>

# Quick Sort: best-case analysis

- **Best case:** Number of compares is  $\sim N \lg N$ .

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

# Quick Sort: worst-case analysis

- **Worst case:** Number of compares is  $\sim \frac{1}{2} N^2$ .

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

# Quick Sort: average-case analysis

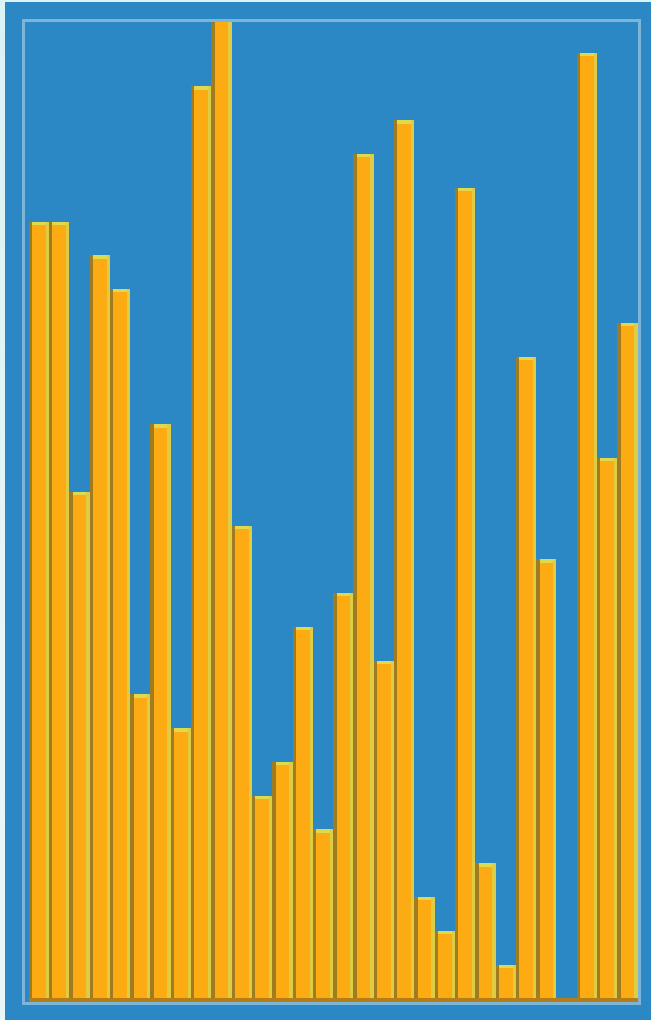
---

- Worst case: Number of compares is quadratic.
  - $N + (N - 1) + (N - 2) + \dots + 1 \sim \frac{1}{2} N^2$ .
  - More likely that your computer is struck by lightning bolt
- **Average case:** Number of compares is  $\sim 1.39 N \lg N$ .
  - 39% more compares than mergesort.
  - But faster than mergesort in practice because of less data movement.
- Random shuffle:
  - Probabilistic guarantee against worst case.
  - Basis for math model that can be validated with experiments.

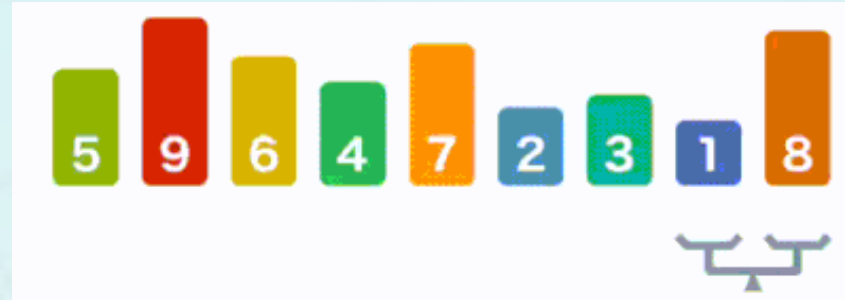
Age Group	Number of Deaths
0-4	75
5-9	75
10-14	65
15-19	85
20-24	80
25-29	100
30-34	85
35-39	75
40-44	75
45-49	95
50-54	95
55-59	100
60-64	85
65-69	75
70-74	65
75+	95

# Sorting Algorithm Animation

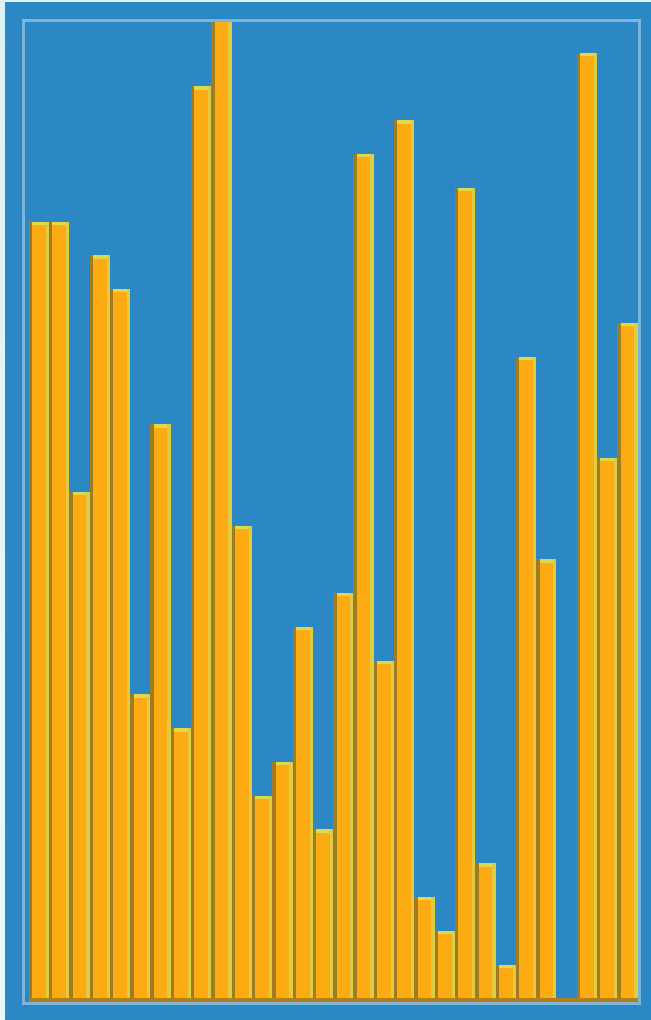
---



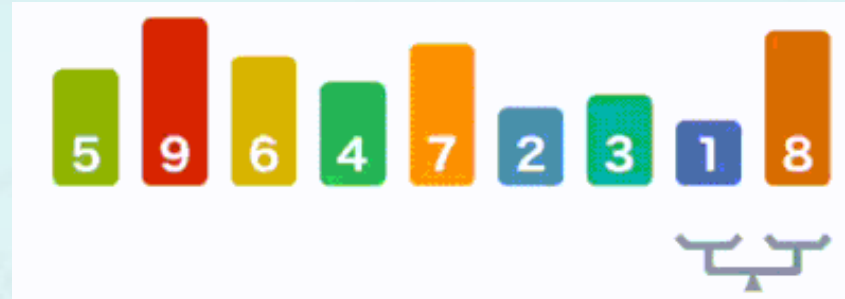
Insertion Sort



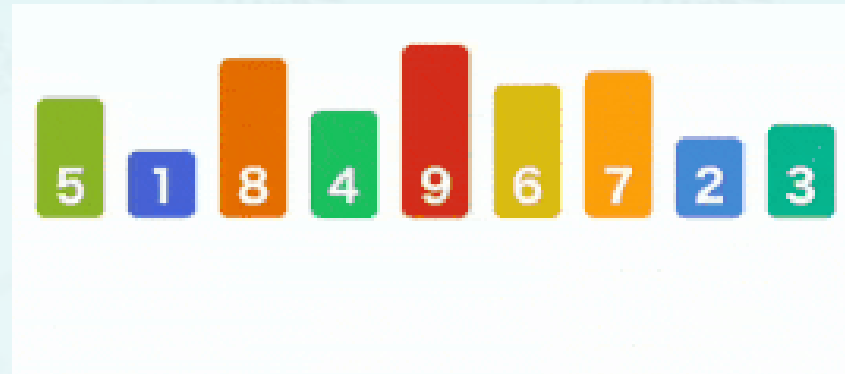
# Sorting Algorithm Animation



Insertion Sort

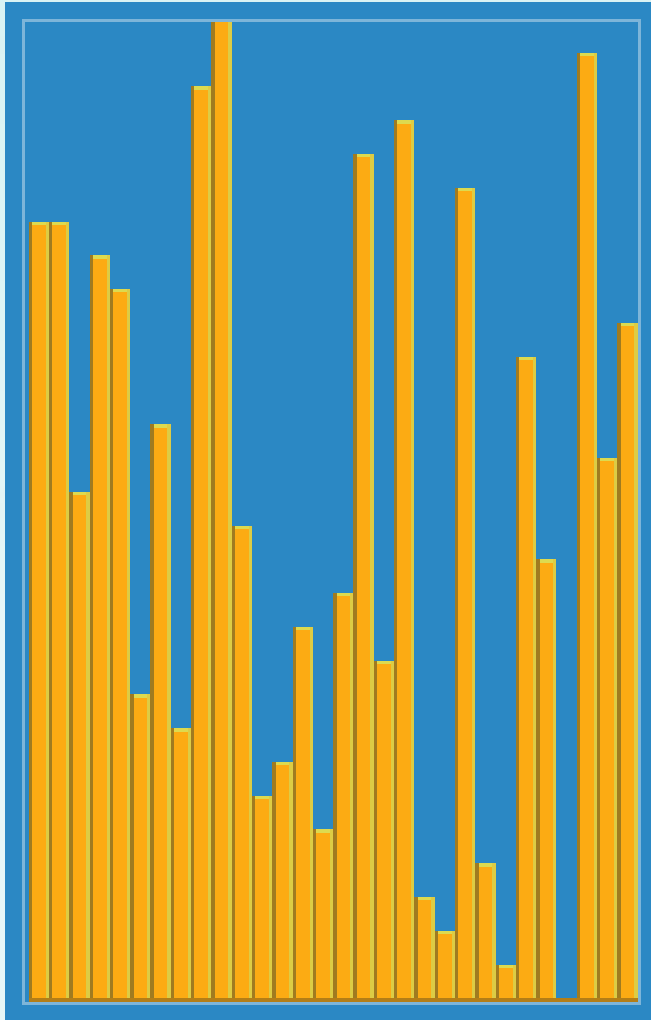


Bubble Sort

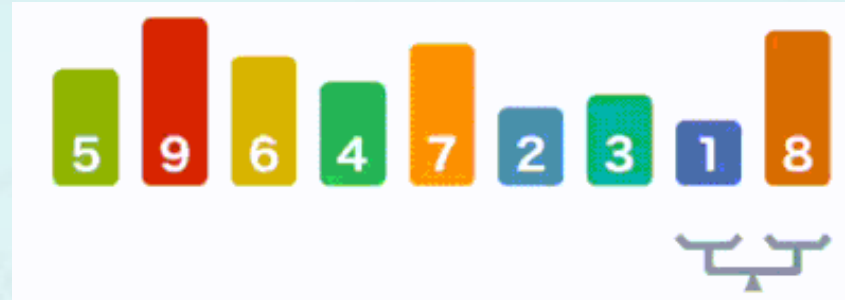




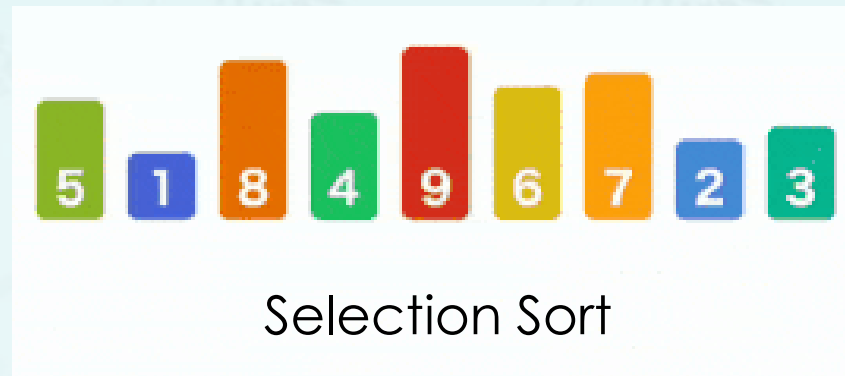
# Sorting Algorithm Animation



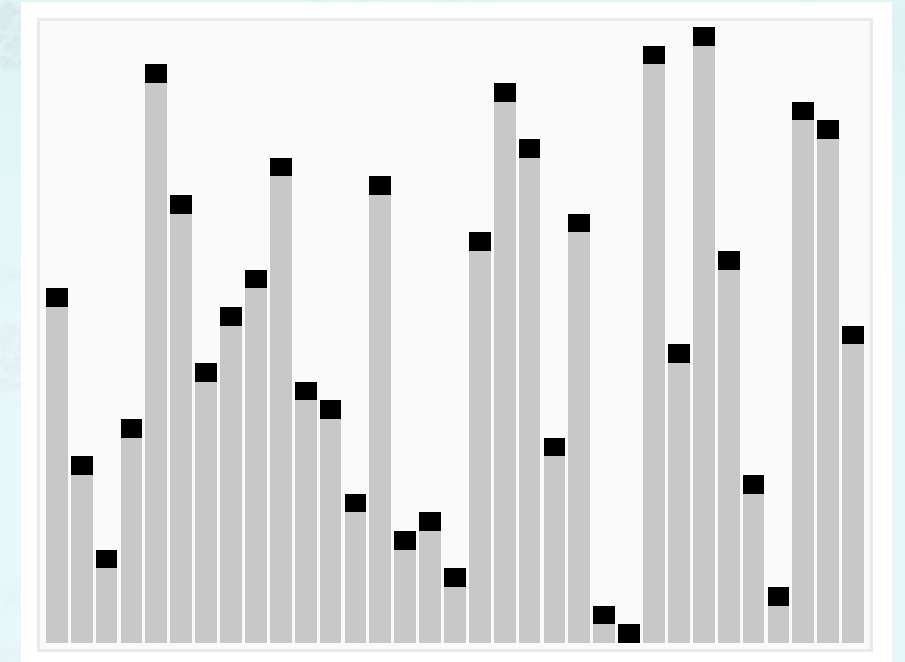
Insertion Sort



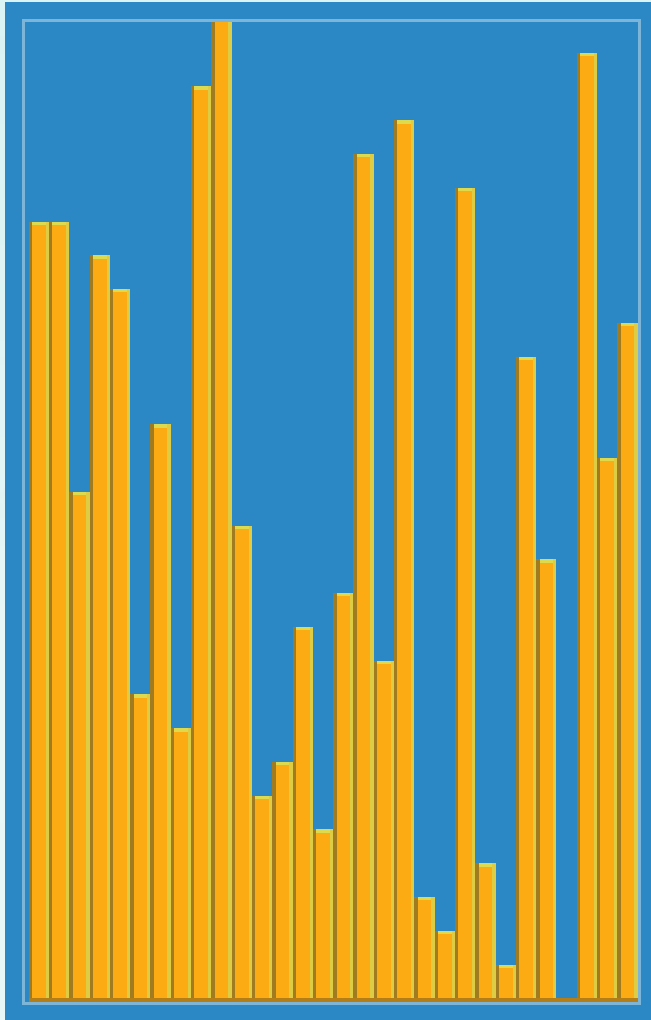
Bubble Sort



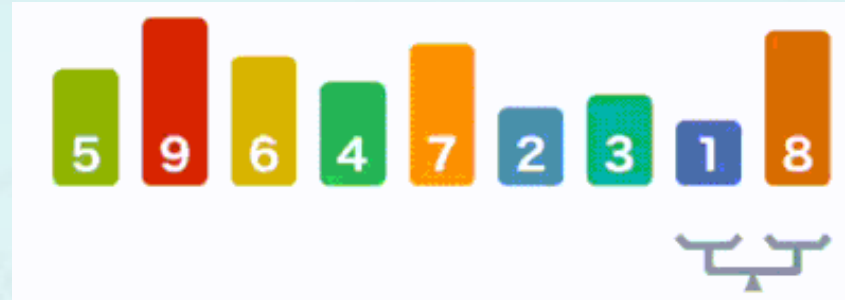
Selection Sort



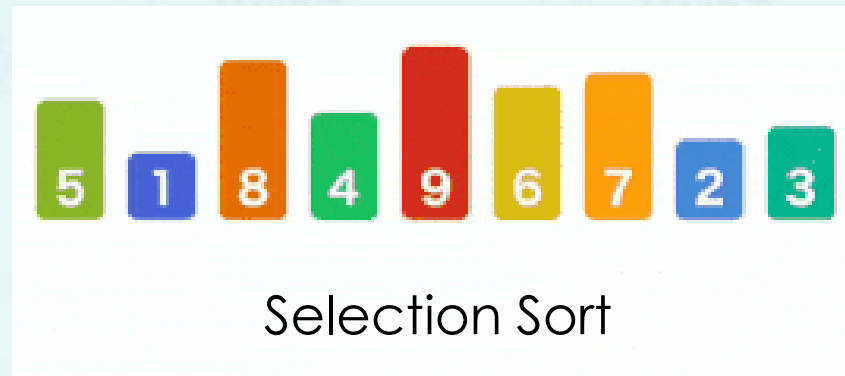
# Sorting Algorithm Animation



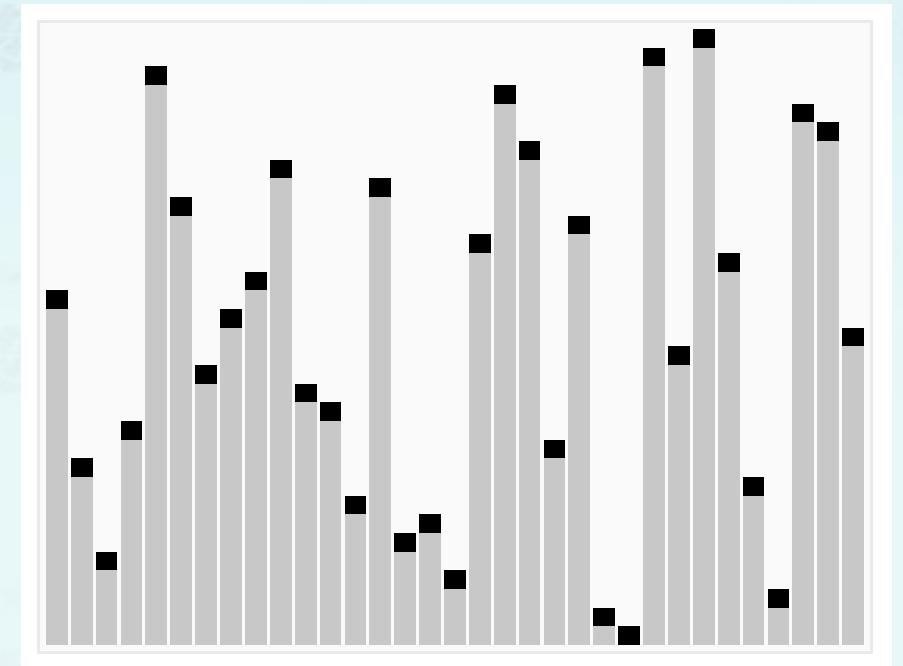
Insertion Sort



Bubble Sort



Selection Sort



quicksort

<https://commons.wikimedia.org/wiki/File:Insertion-sort-example.gif>  
[https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)

# Sorting(2/2)

Data Structures  
C++ for C Coders

한동대학교 김영섭 교수  
idebtor@gmail.com

Merge Sort  
Quick Sort