SORTING ALGORITHMS

Mahsa Ziraksima
ACIBADEM UNIVERSITY
Fall 2025

Run Time

```
for i in range(1, 6):
    print(i)
```

```
f(n) = 5
```

```
count = 0
for i in range(1, 6):
    print(i)
    count += i
```

```
f(n) = 11
```

```
count = 0
for i in range(1, n):
    print(i)
    count += i
```

```
f(n) = 1 + 2n
```

```
def print_it(n):
    # loop 1
    for i in range(n):
        print(i)
    # loop 2
    for i in range(n):
        print(i)
        for j in range(n):
            print(j)
            for h in range(n):
                print(h)
```

```
T(n) = n + n**3
```

```python
customers = [.....]

# 1
print(customers[0])


# log n
i = 1
while i <= (int(len(customers))):
    print(customers[i])
    i = i * 2

# n
for customer in customers:
    if customer[0] == 'B':
        print(customer)


# n log n
for customer in customers:
    var = ""
    i = 1
    while i <= (int(len(customers))):
        var += (customers[i])
        i = i * 2
    print(var)
```

```python
# n**2
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        x = i * j
        print(x)

# n**3
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    for j in numbers:
        for h in numbers:
            x = i + j + h
            print(x)


# c**n
pin = 931
n = len(str(pin))
for i in range(10**n):
    if i == pin:
        print(i)
```

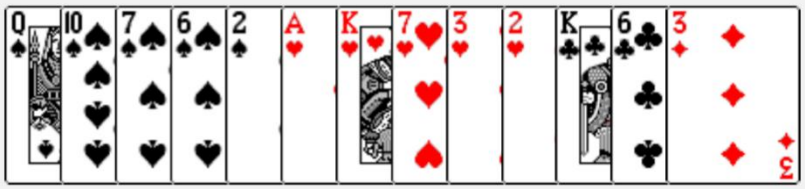
SUMMARY

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Quick sort

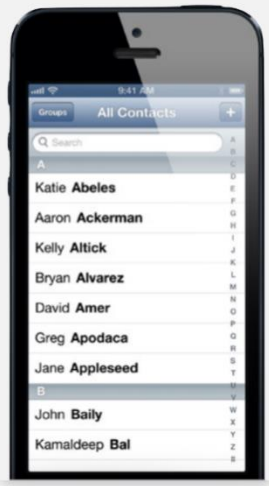
SORTING DATA

- It means arranging it in a meaningful way.
  - *Numbers*
    - from the smallest to the largest
  - *Books*
    - from the shortest book to the longest
    - oldest to newest

FedEx packages



playing cards



contacts

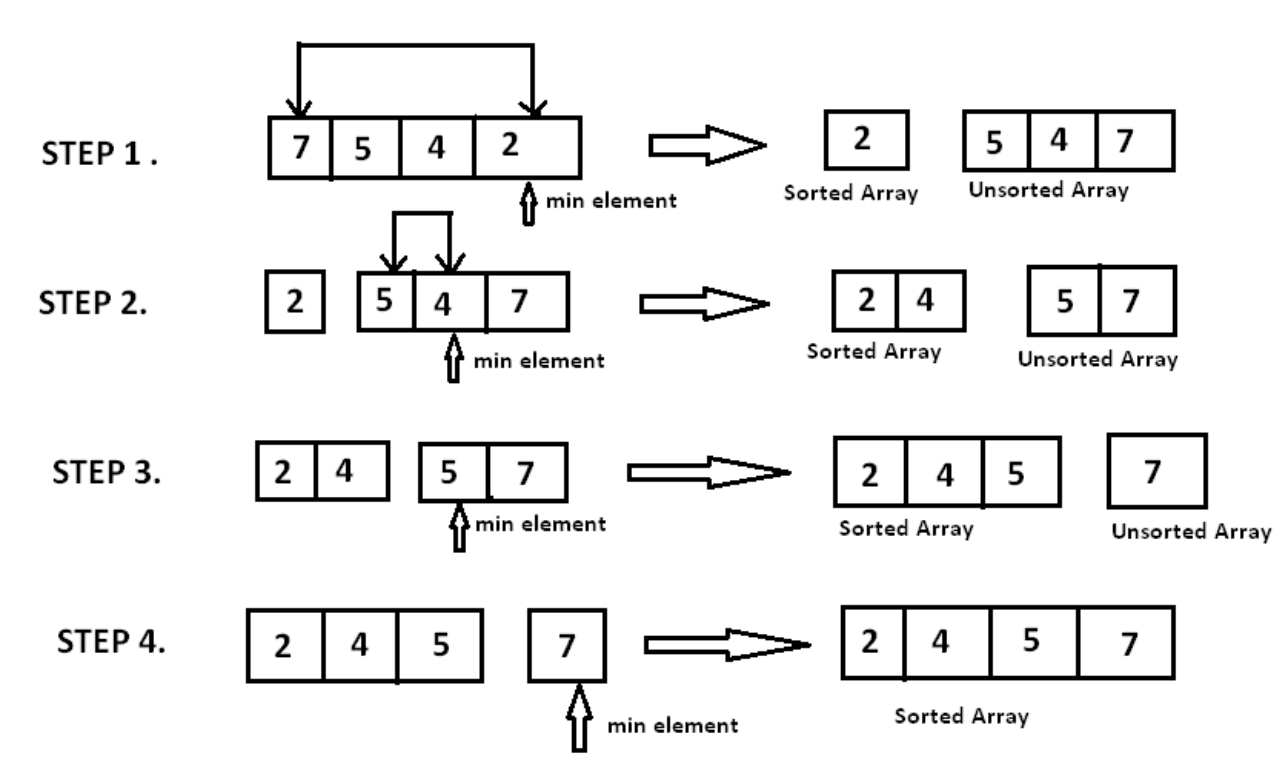
SELECTION SORT

- In computer science, selection sort is an in-place comparison sorting algorithm. The algorithm works by repeatedly finding the minimum element from the unsorted part and putting it at the end of the sorted part.

2.1 SELECTION SORT DEMO

SELECTION SORT

- In computer science, selection sort is an in-place comparison sorting algorithm. The algorithm works by repeatedly finding the minimum element from the unsorted part and putting it at the end of the sorted part.



SELECTION SORT

- Pseudocode:

```
function selectionSort(array):
    n = length(array)
    for i from 0 to n - 1:
        // Assume the minimum is the first element of the unsorted part
        minIndex = i
        // Find the index of the smallest element in the unsorted part
        for j from i + 1 to n - 1:
            if array[j] < array[minIndex]:
                minIndex = j
        // Swap the found minimum element with the first element of the unsorted part
        swap(array[i], array[minIndex])
    return array
```

- Complexity: $O(n^2)$

BUBBLE SORT

- Computer scientists call it a bubble sort because the numbers with the highest values "bubble up" to the end of the list, and the numbers with the smallest values move to the beginning of the list as the algorithm progresses.

```
[32, 1, 9, 6]
[32, 1, 9, 6]
[1, 32, 9, 6]
[1, 32, 9, 6]
[1, 9, 32, 6]
[1, 9, 32, 6]
```

```
[1, 9, 6, 32]
[1, 9, 6, 32]
[1, 9, 6, 32]
[1, 6, 9, 32]
```

```
[1, 6, 9, 32]
```

BUBBLE SORT

- Pseudocode:

```
function bubbleSort(array):
    n = length(array)
    for i from 0 to n - 1:
        // Track if a swap was made
        swapped = false
        for j from 0 to n - i - 2:
            if array[j] > array[j + 1]:
                // Swap the elements
                swap(array[j], array[j + 1])
                swapped = true
        // If no swaps were made, the array is sorted
        if not swapped:
            break
    return array
```

- Complexity: $O(n^2)$

BUBBLE SORT

- Bubble sort is considered a stable sorting method because it preserves the relative order of equal elements in the sorted output.

```
[(1, 'B'), (2, 'C'), (3, 'A'), (3, 'D')]
```

```
[(3, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]
```

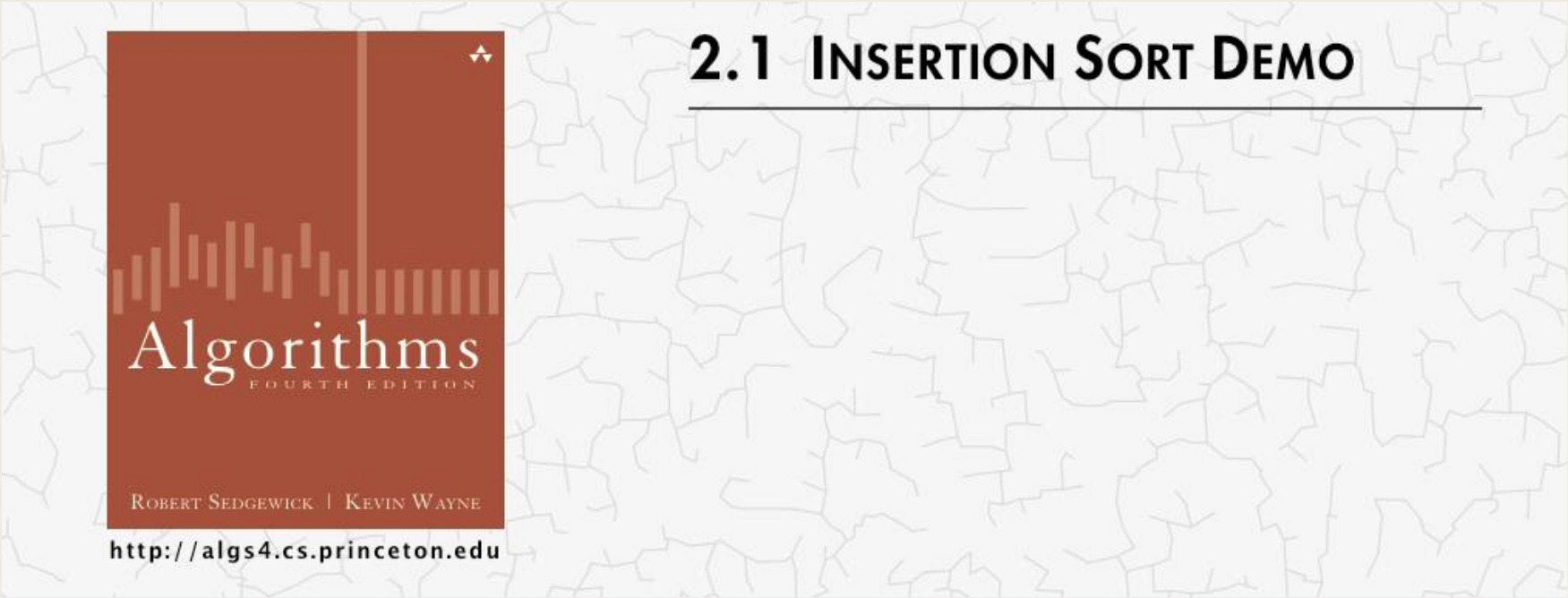
- How Bubble Sort Maintains Stability:

  During the sorting process, bubble sort compares adjacent elements and only swaps them if the earlier element is greater than the later one, and if they are equal, bubble sort does not swap them; thus, their relative positions remain unchanged.

- In Multi-level sorting stable sorting ensures that the secondary sort doesn't disrupt the order established by the primary sort. For example, if you sort a list of names first by last name and then by first name, a stable sort will keep individuals with the same last name in their original first-name order.

INSERTION SORT

- First, you split a list of numbers into two: a sorted left half and an unsorted right half. Then, you sort the left half the same way you sort a hand of playing cards.



INSERTION SORT

- First, you split a list of numbers into two: a sorted left half and an unsorted right half. Then, you sort the left half the same way you sort a hand of playing cards.

```
[6, 5, 8, 2]
[5, 6, 8, 2]
[5, 6, 8, 2]
[5, 6, 8, 2]
[5, 6, 8, 2]
[5, 6, 8, 2]
[2, 5, 6, 8]
```

INSERTION SORT

```
function insertionSort(array):
    n = length(array)
    for i from 1 to n - 1:  // Start from the second element
        key = array[i]      // The element to be inserted
        j = i - 1           // Index of the last sorted element

        // Move elements of array[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while j >= 0 and array[j] > key:
            array[j + 1] = array[j]
            j = j - 1

        // Insert the key at its correct position
        array[j + 1] = key

    return array
```

- Complexity: $O(n^2)$

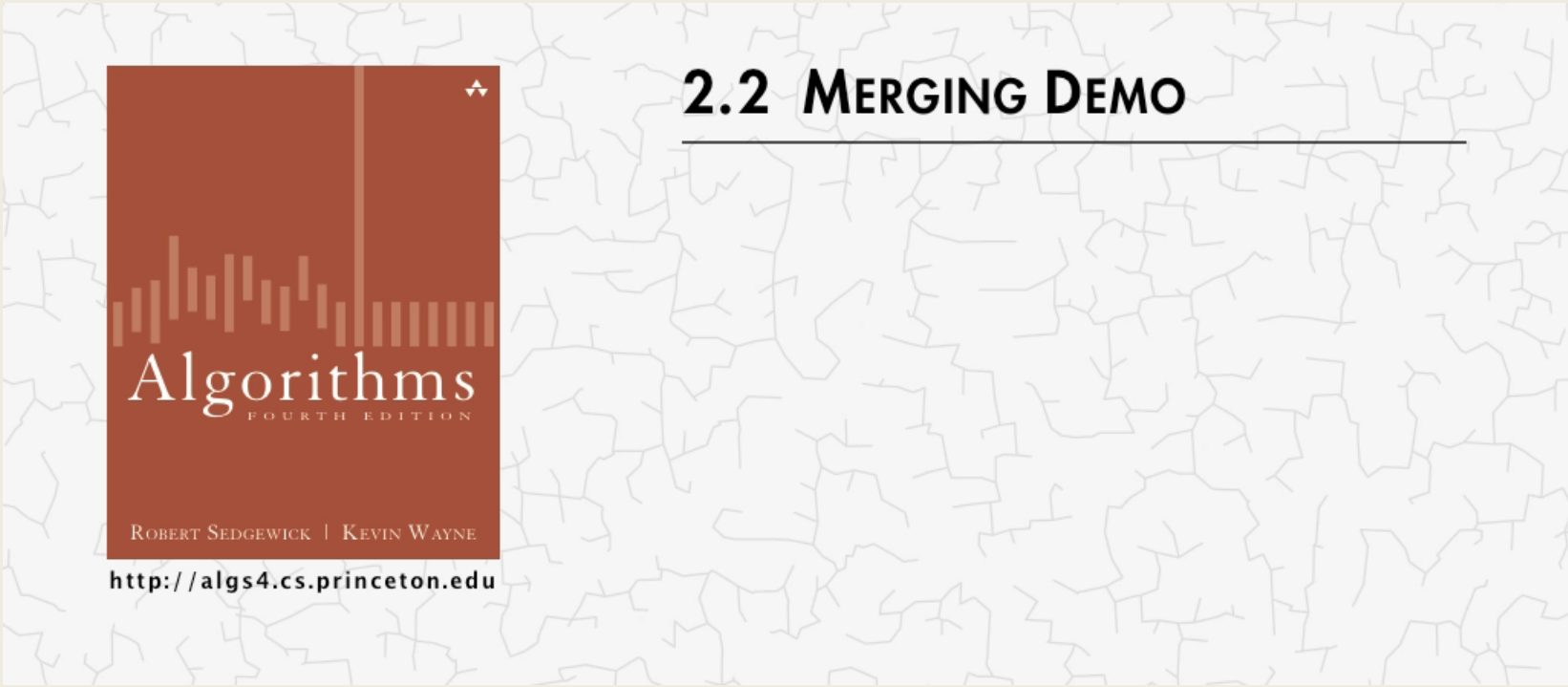
INSERTION SORT

- It is an **stable** sort algorithm.
- It is an efficient choice if you have nearly sorted data, with time complexity of O(*n)*
- It is linear because the second loop has to run only once.

```
[1, 2, 3, 4, 5, 7, 6]
```

MERGE SORT

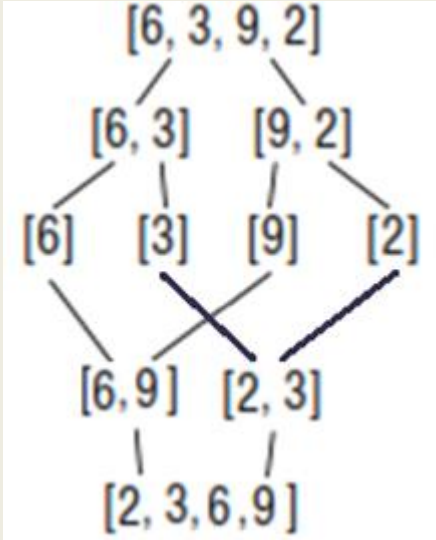
- It is a recursive sorting algorithm that continually splits a list in half until there are one or more lists containing one item and then puts them back together in the correct order.



MERGE SORT

- It is an example of a **divide- and- conquer** algorithm.

recursively breaks a problem into two or more related sub problems until they are simple enough to solve easily.



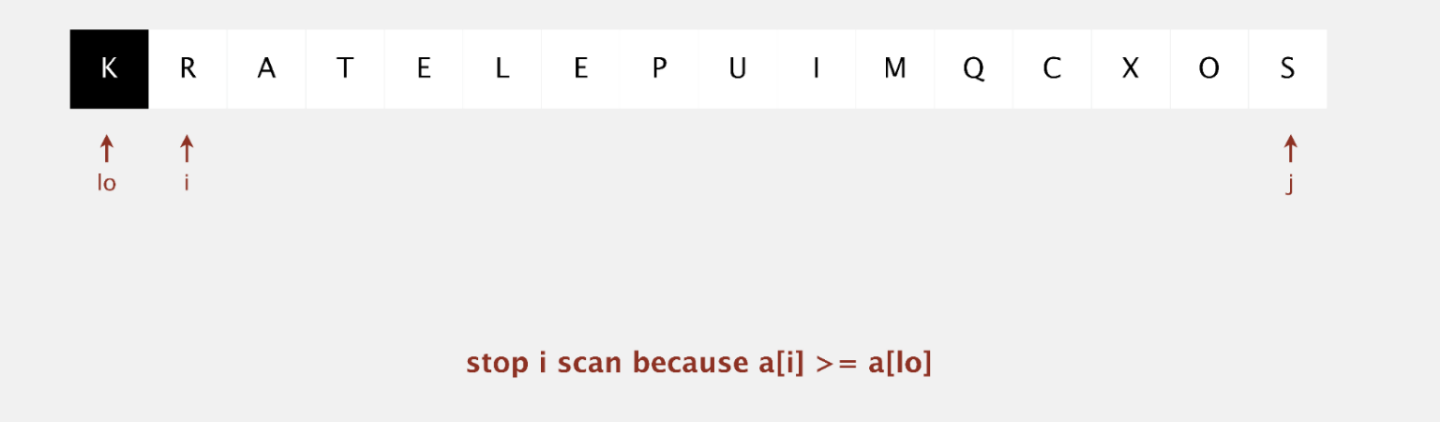
- Complexity: O(n * log n)
  - *The number of times you can divide the array into halves is O(logn).*
  - *Merging two sorted arrays takes linear time, O(n), because you have to compare each element and arrange them in order.*
- ***Merge concept** is one of the most efficient sorting algorithms and widely used by computer scientists.*

MERGE SORT

```
def merge_sort(a_list):
    if len(a_list) > 1:
        mid = len(a_list) // 2
        left_half = a_list[:mid]
        right_half = a_list[mid:]
        merge_sort(left_half)
        merge_sort(right_half)

        left_ind = 0
        right_ind = 0
        alist_ind = 0
        while left_ind < len(left_half) and right_ind < len(right_half):
            if left_half[left_ind] <= right_half[right_ind]:
                a_list[alist_ind] = left_half[left_ind]
                left_ind += 1
            else:
                a_list[alist_ind] = right_half[right_ind]
                right_ind += 1
            alist_ind += 1

        while left_ind < len(left_half):
            a_list[alist_ind]=left_half[left_ind]
            left_ind += 1
            alist_ind += 1

        while right_ind < len(right_half):
            a_list[alist_ind]= right_half[right_ind]
            right_ind += 1
            alist_ind += 1
```

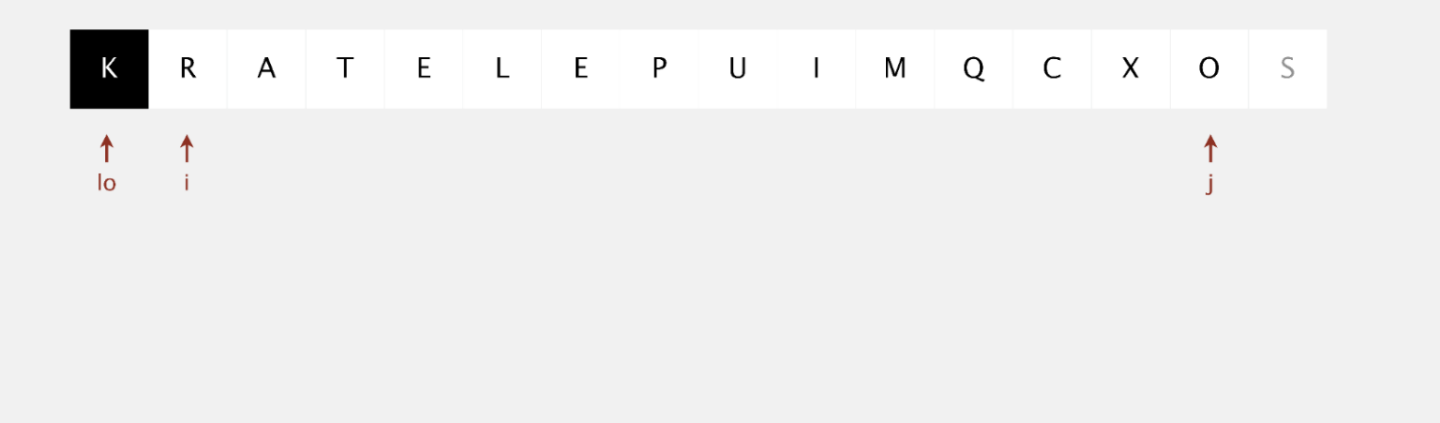
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



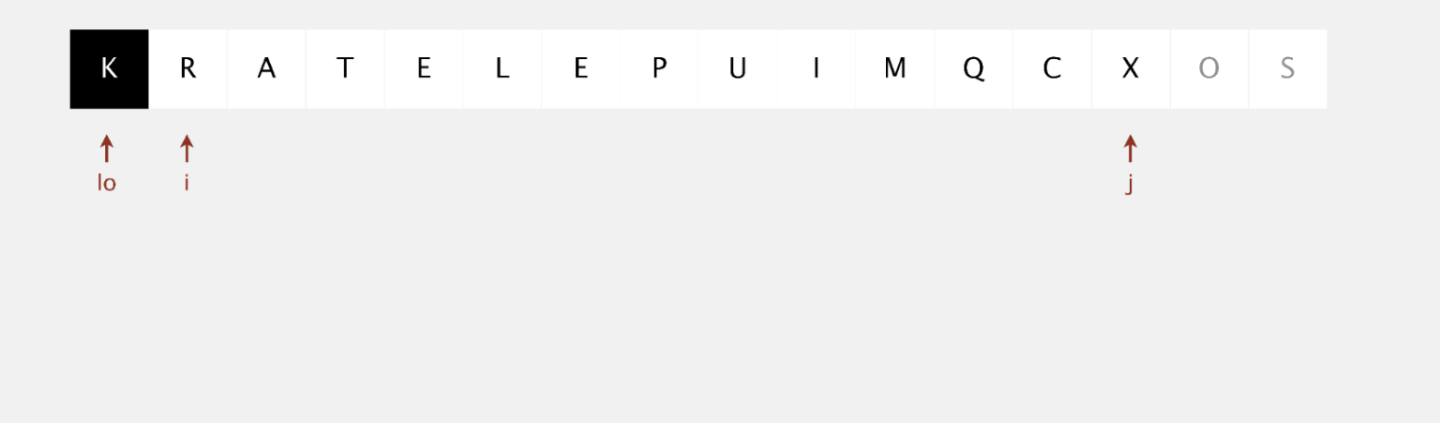
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



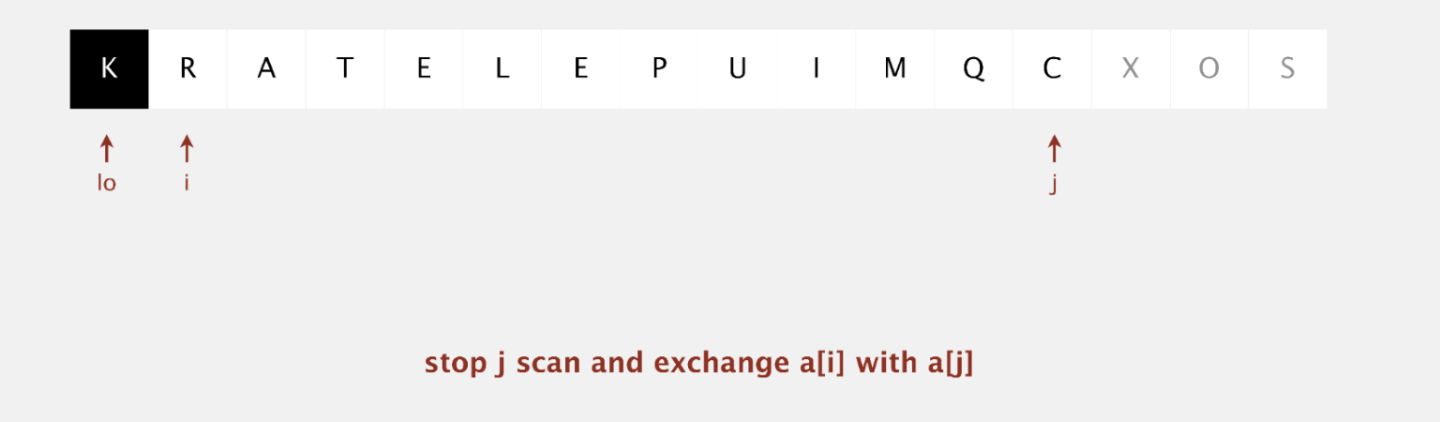
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



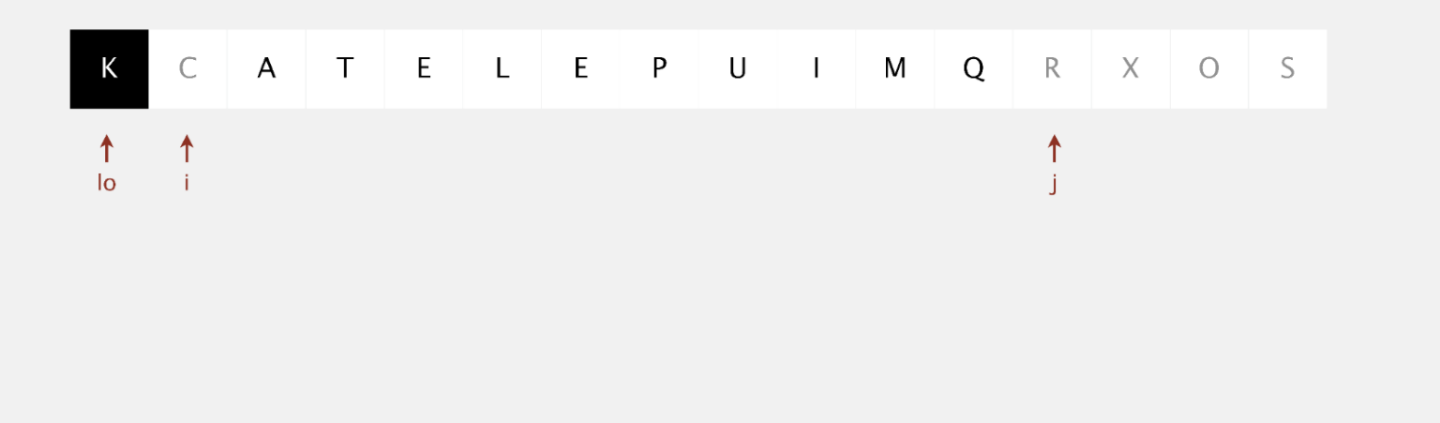
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



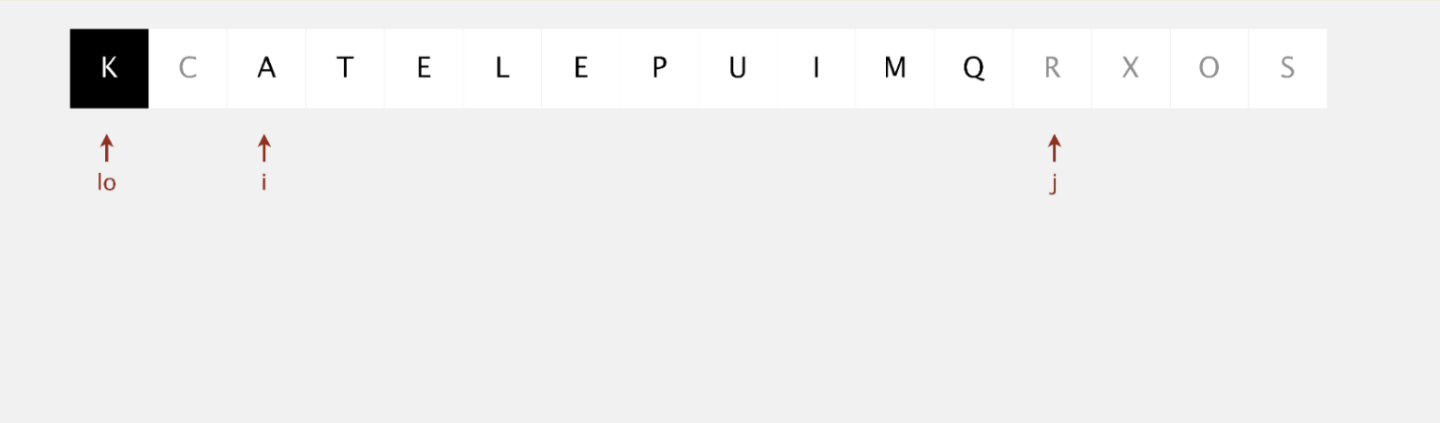
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



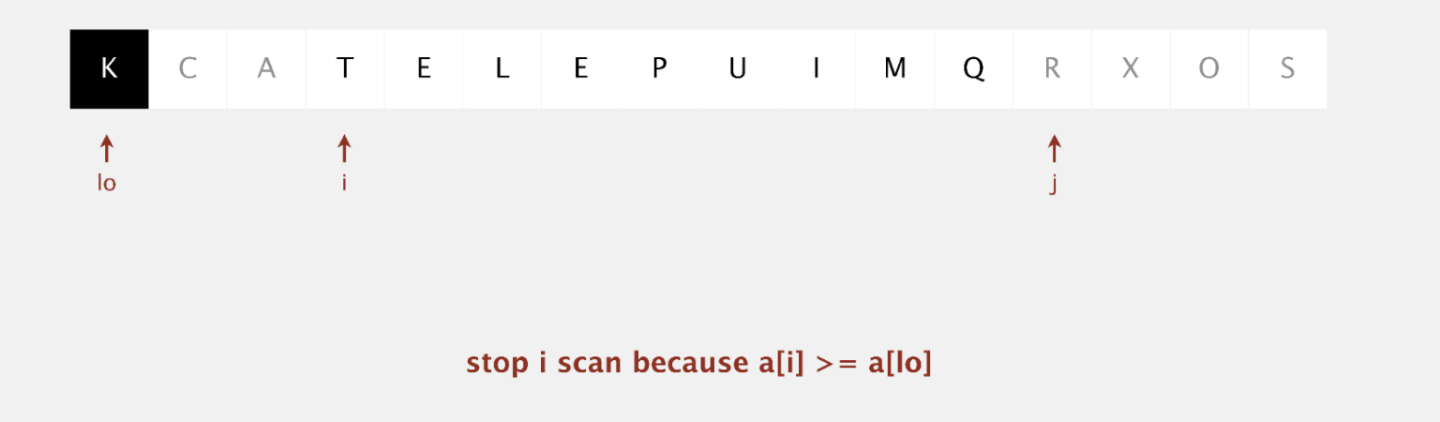
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



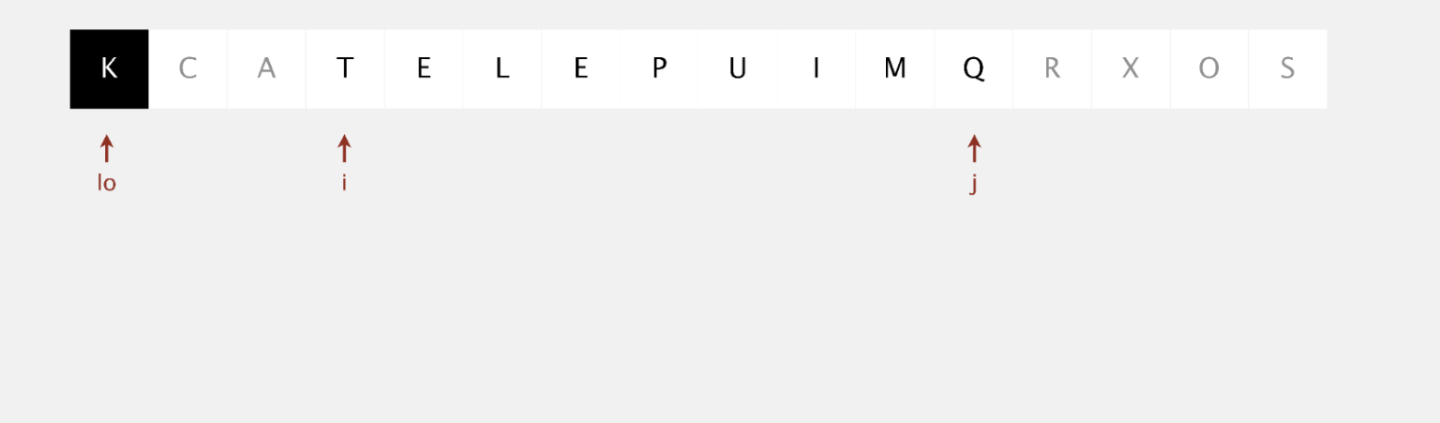
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.

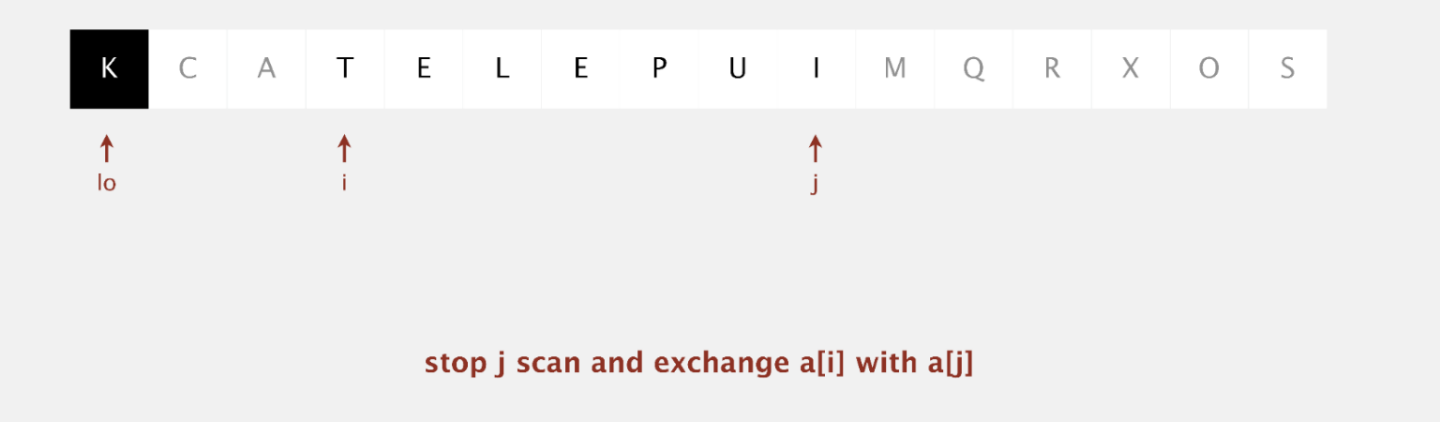


QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.

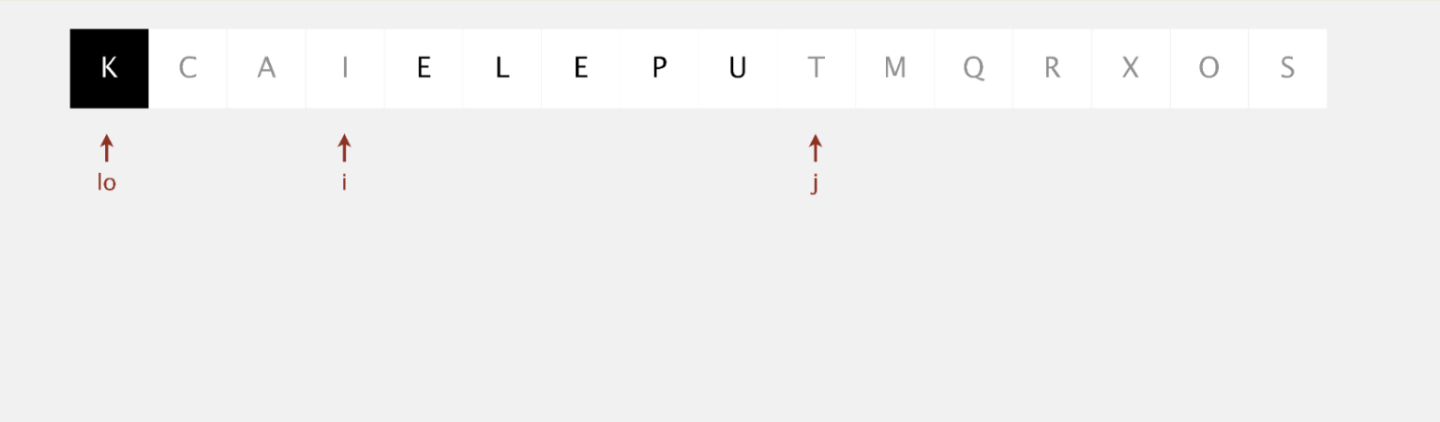
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



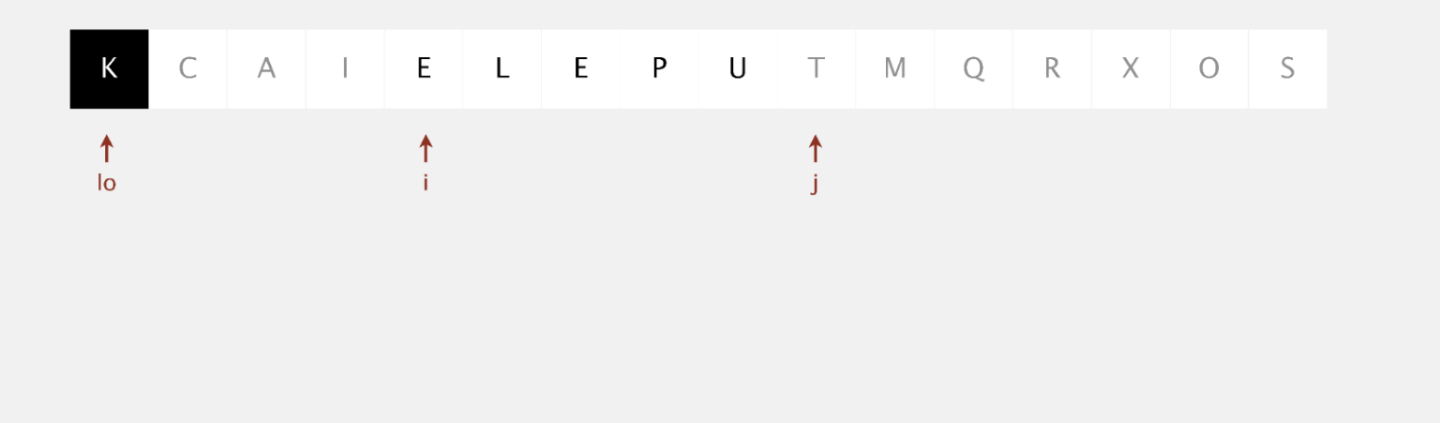
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



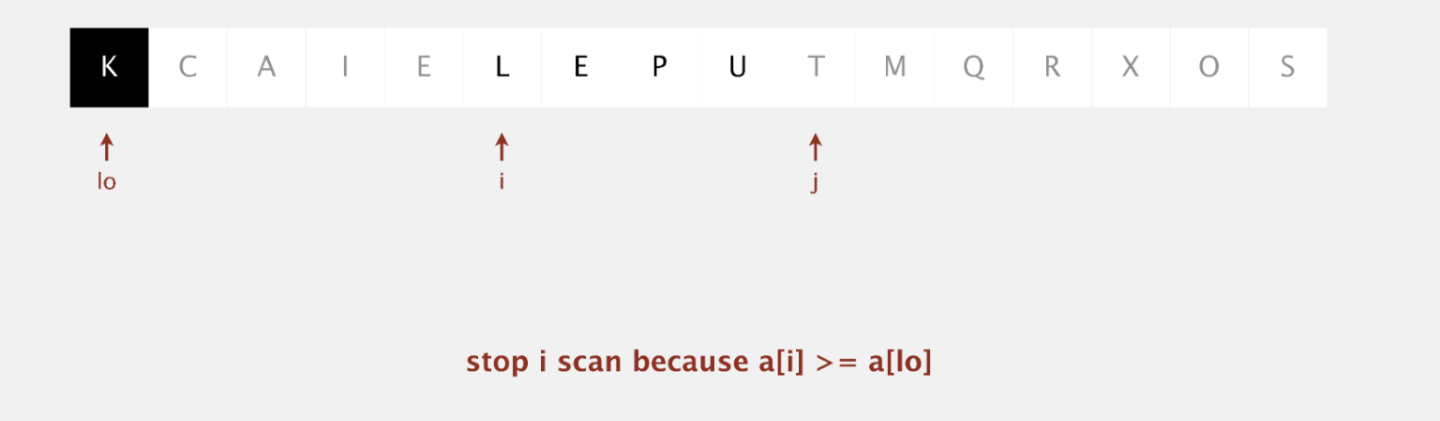
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



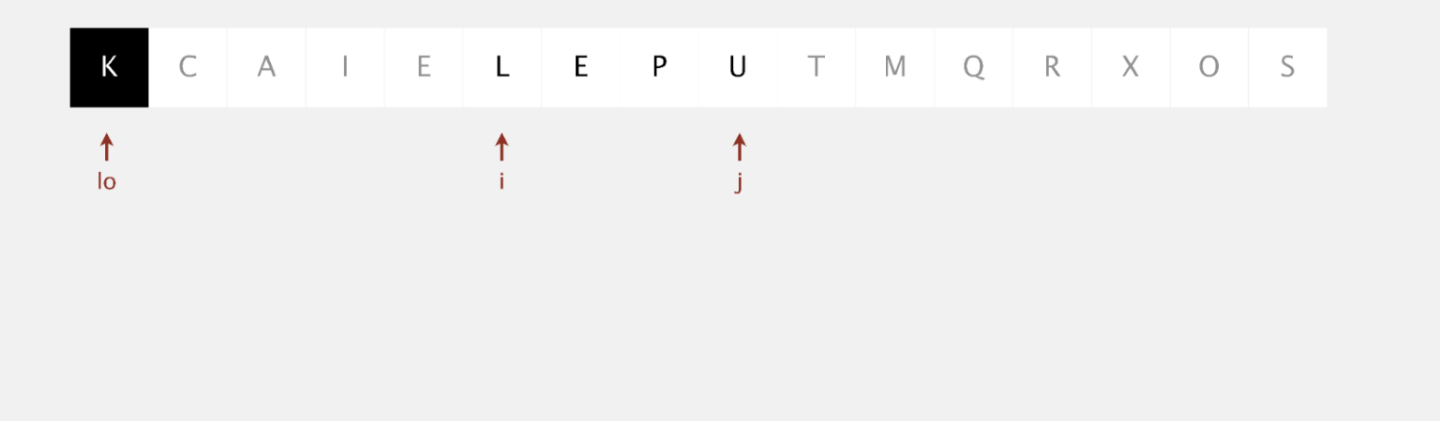
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



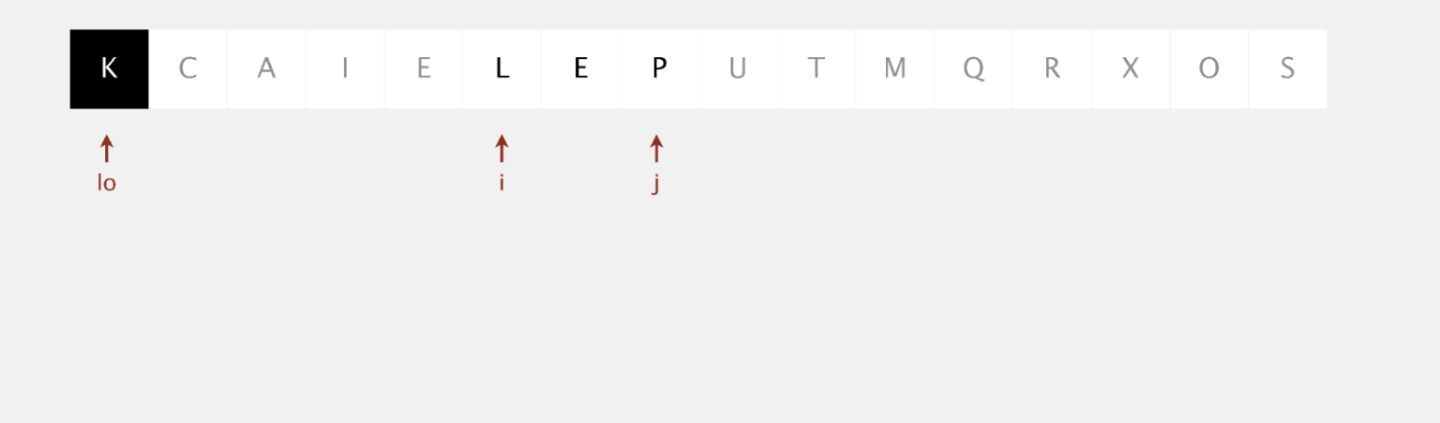
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



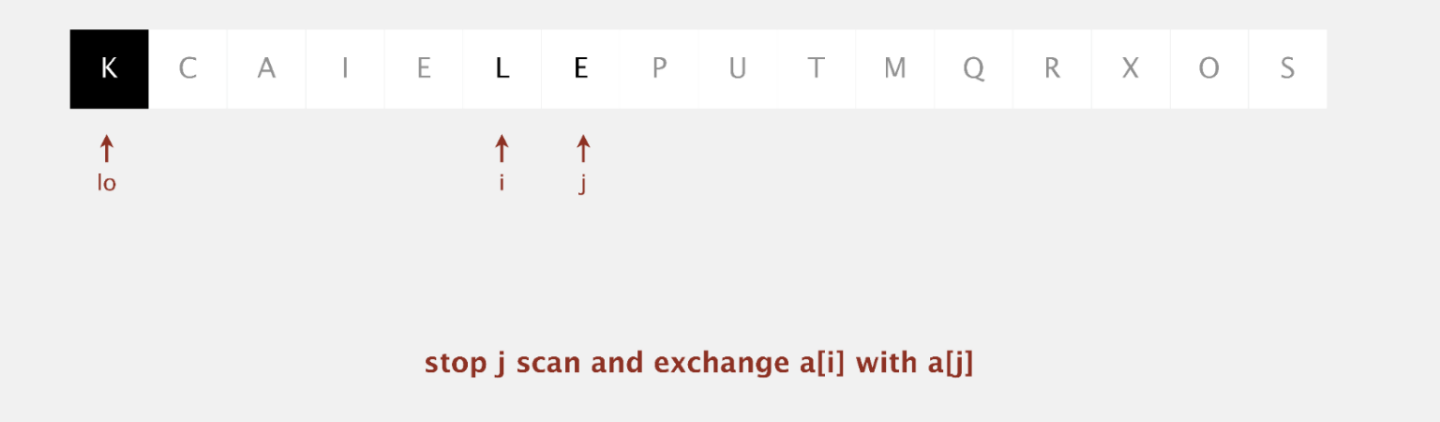
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



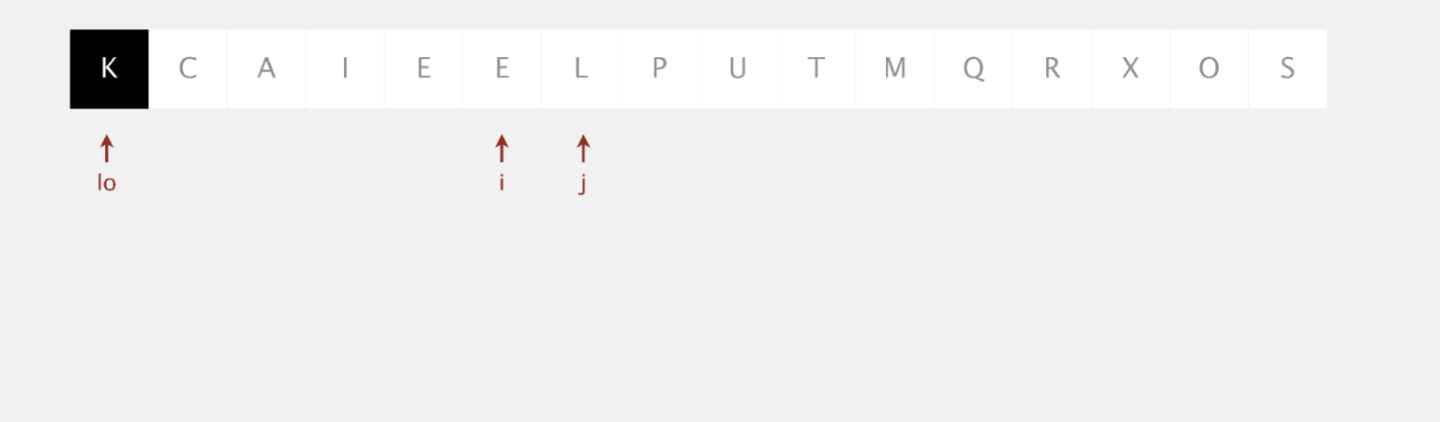
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



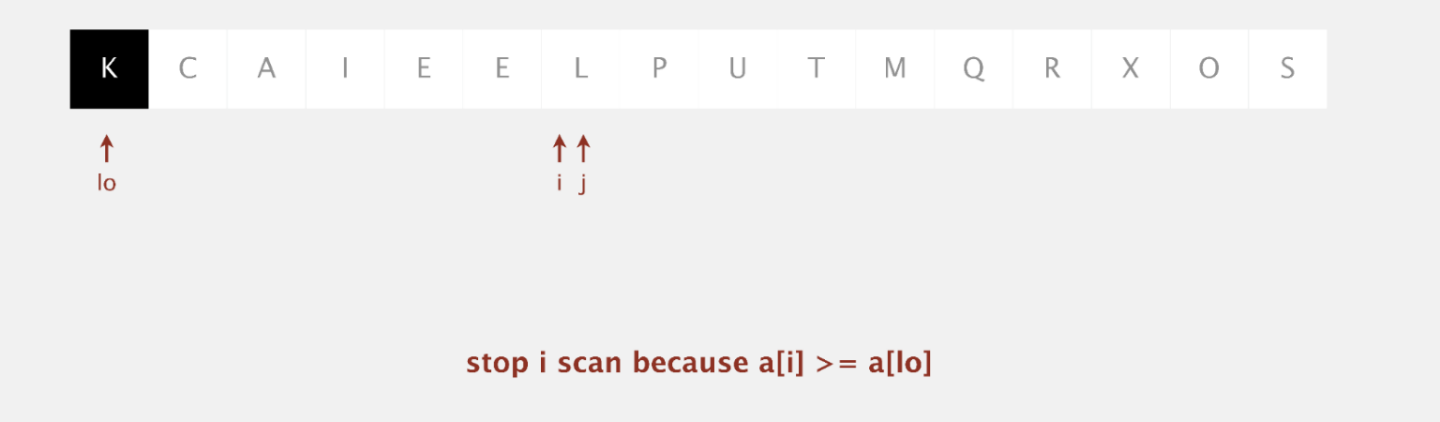
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



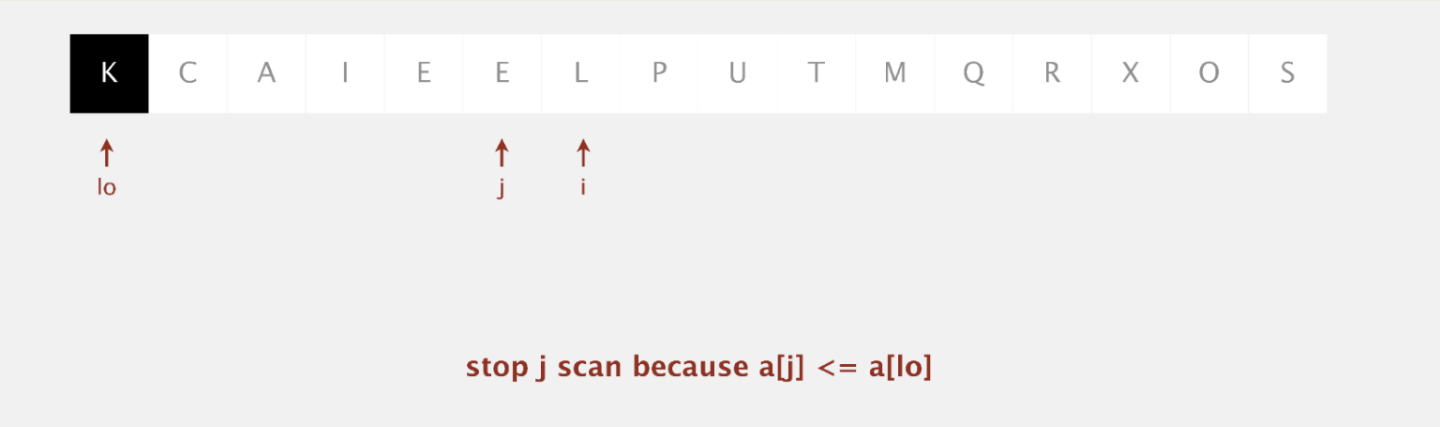
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



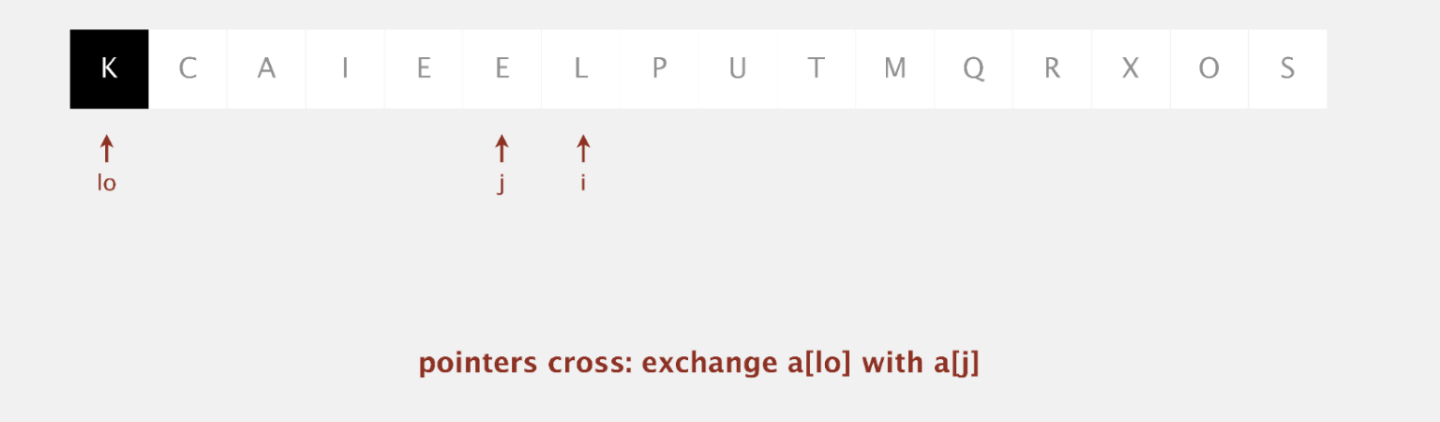
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



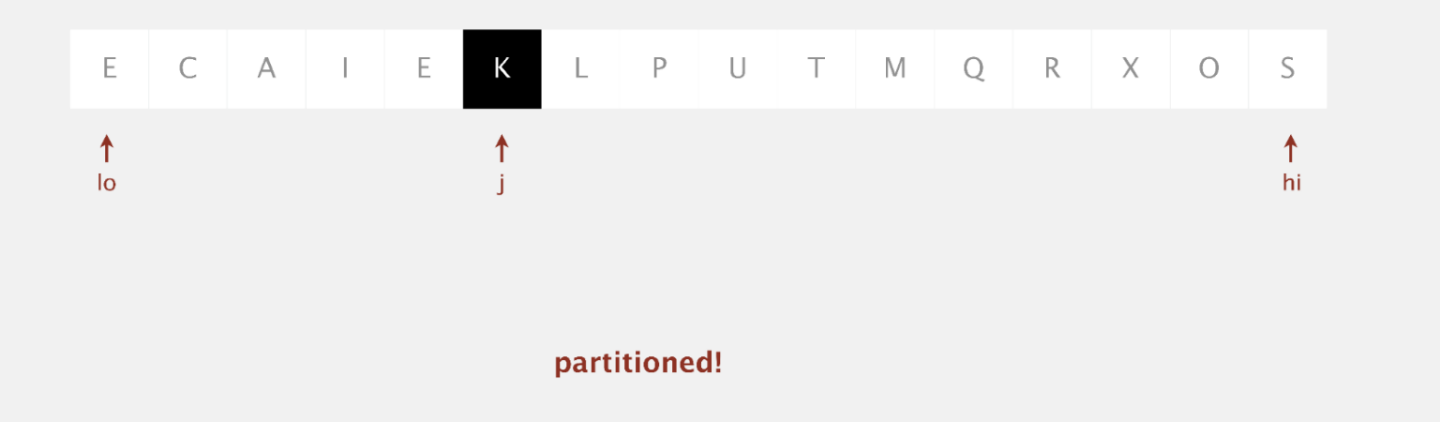
QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



QUICK SORT

- Quicksort 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.
- For this reason, it is sometimes called partition-exchange sort.



QUICK SORT

- The time complexity of the quicksort algorithm can vary based on the choice of the pivot and the distribution of the input data.
  - *Average Case: O(nlogn)*
  - *Worst Case: O(n^2)*

    The worst case occurs when the pivot selection consistently results in unbalanced partitions, such as when the smallest or largest element is chosen as the pivot. For example, this can happen if the array is already sorted or nearly sorted. In this case, quicksort would make n recursive calls, each handling n−1, n−2, ..., down to 1 element, leading to quadratic time complexity.

$1, 2, 3, 4, 5$

Step-by-Step Breakdown:

1. **Choose a Pivot:**
   - Let's say we always choose the last element as the pivot. In this case, the pivot is `5`.
2. **Partitioning:**
   - During partitioning, all elements are less than `5`, so no swaps are made. The array remains the same: $1, 2, 3, 4, 5$
   - The pivot `5` is in its correct position (index 4).
3. **Recursively Sort Subarrays:**
   - Left subarray: $1, 2, 3, 4$ (elements less than `5`).
   - Right subarray: [] (no elements greater than `5`).
4. **Next Level of Recursion:**
   - Choose `4` as the pivot for the left subarray $1, 2, 3, 4$.
   - Again, no swaps are needed since all elements are less than `4`. The array remains: $1, 2, 3, 4$
   - The pivot `4` is in its correct position (index 3).
5. **Continue Recursion:**
   - This process continues:
     - Choose `3` as the pivot for $1, 2, 3$.
     - Choose `2` as the pivot for $1, 2$.
     - Choose `1` as the pivot for 1.

```python
a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list))



a_list = ["Guido van Rossum", "James Gosling", "Brendan Eich", "Yukihiro Matsumoto"]
print(sorted(a_list))



a_list = [1, 8, 10, 33, 4, 103]
print(sorted(a_list, reverse=True))



a_list = ["onehundred", "five", "seventy", "two"]
print(sorted(a_list, key=len))


a_list = [1, 8, 10, 33, 4, 103]
a_list.sort()
print(a_list)
```

```
[1, 4, 8, 10, 33, 103]
['Brendan Eich', 'Guido van Rossum', 'James Gosling', 'Yukihiro Matsumoto']
[103, 33, 10, 8, 4, 1]
['two', 'five', 'seventy', 'onehundred']
[1, 4, 8, 10, 33, 103]
```

Feature	`list.sort()`	`sorted()`
Works on	Lists only	Any iterable
Returns	`None`	New sorted list
Modifies original?	✅ Yes	❌ No
Common usage	`my_list.sort()`	`sorted(my_list)`