Computational Thinking

Lecture 09: Searching & Sorting Algorithms

University of Engineering and Technology

VIETNAM NATIONAL UNIVERSITY HANOI

Outline

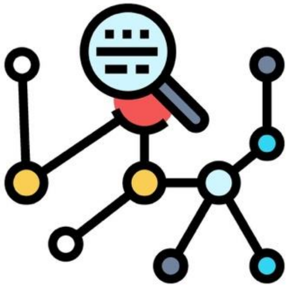
- Motivation and Real-life Scenarios
- Searching Algorithms
  - Linear Search
  - Binary Search
- Sorting Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort

Motivation and Real-life Scenarios

Searching in Daily Life

From waking up to entering this classroom, how many times have you searched?

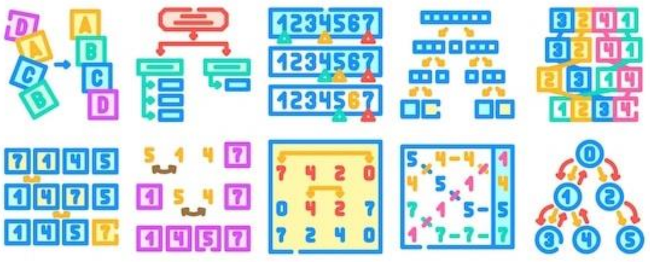
- **Looking** for your **phone** in your room
- **Finding** a **homework file** on your computer
- **Searching** for a **teacher's email**
- **Scrolling** a list to locate a course on LMS



Sorting in Daily Activities

What have you sorted recently?

- **Emails** by **time** or **sender**
- **Photos** by **date**
- **Music playlists** by **artist**
- **Grade lists** from **highest** to **lowest**
- **Messages** by **priority**



The Real Challenge

Q1. Search your name in the list of 5000 student names?

Q2. What differences once searching on an unsorted and a sorted list?

Searching Algorithms

Searching

- **Searching** is one of the common tasks that computers perform.
- **Searching Algorithms** are designed to check for an element or retrieve and element from any source of data.
- Two parameters that affect search algorithm selection:
  1) Whether the list is **sorted**
  2) Whether all the elements in the list are **unique** or have **duplicate** values.
- Two types of searches:
  1) Sequential Search: Ex. **Linear Search**
  2) Interval Search: Ex. **Binary Search**

Searching: Linear Search

- The simplest way to find an element in a list is to check if it **matches** the required value
- **Linear Search** starts at the beginning of the list and checks every element in the list.
  1) If the value is matched it returns the current element's index, else it returns -1.
  2) Worst case: the entire list must be linearly searched.
  3) This occurs when the value is in the last element or not found.

Searching: Linear Search

Let's take a look at how the linear search algorithm operates!

Example 1:

Let's take an unsorted array as input.

Let the elements of array are:

0	1	2	3	4	5	6	7	8
30	11	70	40	41	14	57	52	25

Let the element to be search is **V = 40**

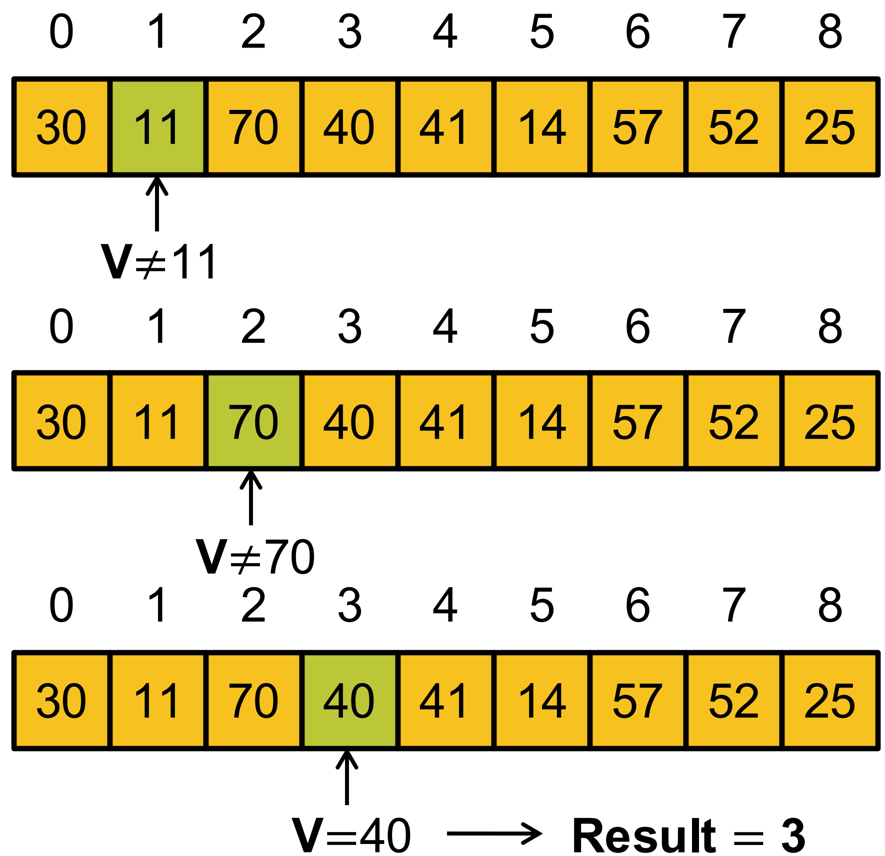
Start from the first element and compare **V** with each element of the array.

0	1	2	3	4	5	6	7	8
30	11	70	40	41	14	57	52	25

V$\neq$30

If **V** doesn't match the first element, we move to the next and repeat the process until we find the target or reach the end of the array.

Searching: Linear Search



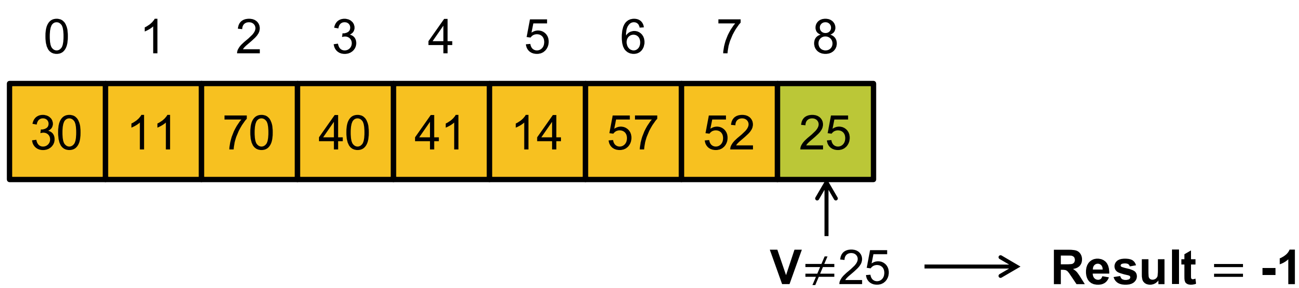
Now, the element to be searched is found. So, the algorithm will return the index of the element matched.

What if the element isn't found in the list?

Example 2:

Let **V = 50**

We start to search from the first element …



If the element you're searching for isn't in the list, the algorithm will typically return a special value to indicate that the search failed.

In this example the return results is -1 if the value to be searched isn't there.

Implementation

```
# Linear search for a single occurrence – Complexity O(n)
def linear_search(arr, value):
    for i in range(len(arr)):
        if arr[i] == value:
            return i
    return -1
```

Found the value, return its index

Value not found

What if ?

In **Example 1 & 2**, we assume that the required value appears only once.

What if we use this algorithm on a list that has **duplicate elements**?

Example 3: Let **V = 70** and the elements of the array are:

0	1	2	3	4	5	6	7	8
30	11	70	40	41	70	57	52	25

The algorithm return with **Result = 2** after it first saw the value 70 in the array.

0	1	2	3	4	5	6	7	8
30	11	70	40	41	70	57	52	25

V=70 ⟶ **Result** = **2**

We cannot assume that once one element is found, the search is done

⇒ Thus, in this case, we need to continue searching through the **entire array.**

What if ?

We need to modify the algorithm to store the results in an array and ensure it searches through the entire list.

0	1	2	3	4	5	6	7	8
30	11	70	40	41	70	57	52	25

V=70 ⟶ **Results** = [2]

0	1	2	3	4	5	6	7	8
30	11	70	40	41	70	57	52	25

V=70 ⟶ **Results** = [2, 5]

Implementation

Linear search for multiple occurrence – Complexity O(n)

```
def linear_search_all(arr, value):
    indices = []
    for i in range(len(arr)):
        if arr[i] == value:
            indices.append(i)
    return indices
```

Found the value, add its index to the list

Return list of indices (empty if not found)

Searching: Binary Search

- **Binary Search** is designed to searching for an element in a sorted array.
- It searches the given element in the array by dividing the array into two halves

$\Rightarrow$ Hence, the name "binary".

1) Searching begin at the **middle** off the list.
2) If **value** < **middle**, check the **middle** element between the **first element** and the **middle**.
3) If **value** ≥ **middle**, check the **middle** element between the **middle** and the **last element**.
4) The process stops when the value is found or there isn't a valid range to check.

Searching: Binary Search

Let's take a look at how the binary search algorithm operates!

Example 4:

Let's take a sorted array as input.

Let the elements of array are:

0	1	2	3	4	5	6	7	8
11	14	25	30	40	41	52	57	70

Let the element to be search is **V = 57**

0	1	2	3	4	5	6	7	8
11	14	25	30	40	41	52	57	70

↑

Arr[mid] = 40 < V
left = mid + 1 = 5
right = 8
mid = (left + right)/2 = 6

Searching: Binary Search

0	1	2	3	4	5	6	7	8
11	14	25	30	40	41	52	57	70

Arr[mid] = 52 < V
left = mid + 1 = 7
right = 8
mid = (left + right)/2 = 7

0	1	2	3	4	5	6	7	8
11	14	25	30	40	41	52	57	70

Arr[mid] = 40 = V
Result = 7

Now, the element to be searched is found. So, the algorithm will return the index of the element matched.

Implementation

```
# Binary search for a single occurrence – Complexity O(log n)
def binary_search(arr, value):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == value:
            return mid
        elif arr[mid] < value:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Ensures the search continues **only while there's a valid range** to check

Found the value, return its index

Value not found

Linear Search versus Binary Search

Criteria	**Linear Search**	**Binary Search**
Search strategy	Check each element one by one from start to end	Repeatedly divide the search space in half
Data requirement	Works on unsorted data	Requires sorted data
Time Complexity	- Best: O(1) - Average: O(n) - Worst: O(n)	- Best: O(1) - Average: O(log n) - Worst: O(log n)
Implementation difficulty	✅ Very easy	⚠️ More complex
Flexibility	✅ Works for any data	❌ Only for sorted data
Typical use cases	- Small datasets - Quick & dirty search - Data not sorted	- Large datasets - High-performance systems - Databases, indexes

Sorting Algorithms

Sorting

- Computers spend a tremendous amount of time **sorting**.
- **A Sorting Algorithm** is used to rearrange a given array or list of elements in an order.
- Types of sorting algorithms:
  1) Comparison Based: **Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort**
  2) Non-Comparison Based: **Counting Sort, Radix Sort**
  3) Hybrid Sorting Algorithm: **Intro Sort = Quick + Heap + Insert**

Sorting: Bubble Sort

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent element if they are in the wrong order.

  ⇒ Worst-case time complexity is high.

  ⇒ Not suitable for large data sets.

- **Bubble Sort** algorithm:

  Step 1:

  Loop through all element of the list.

  Step 2:

  For each element, compare it to all successive elements.

  Swap them if they are out of order.

Sorting: Bubble Sort

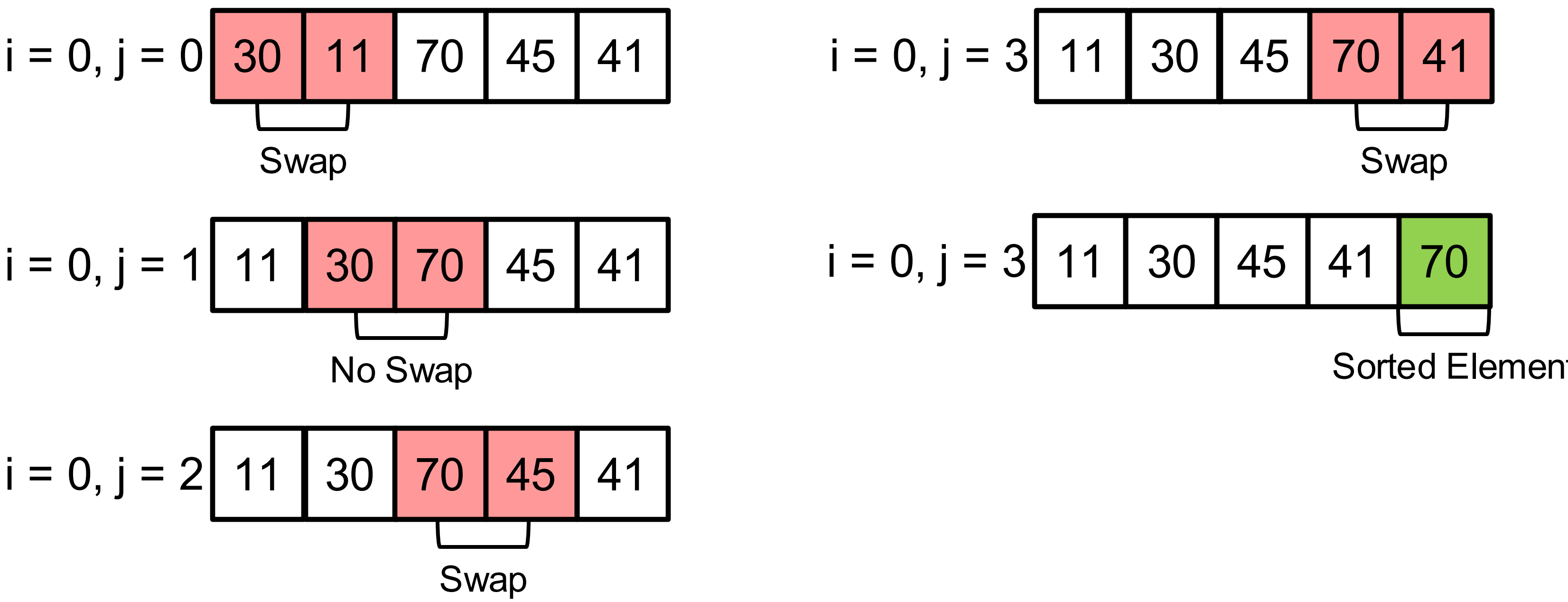
Let's take a look at how the bubble sort algorithm operates!

Example 5:

Let's take an unsorted array as input.

30	11	70	45	41

1st iteration through the array:



Sorting: Bubble Sort

2nd iteration through the array:

i = 1, j = 0	11	30	45	41	70

No Swap

i = 1, j = 0	11	30	45	41	70

No Swap

i = 1, j = 2	11	30	45	41	70

Swap

i = 1, j = 2	11	30	41	45	70

Sorted Element

3rd iteration through the array:

i = 2, j = 0	11	30	41	45	70

No Swap

i = 2, j = 1	11	30	45	41	70

No Swap

i = 2, j = 2	11	30	41	45	70

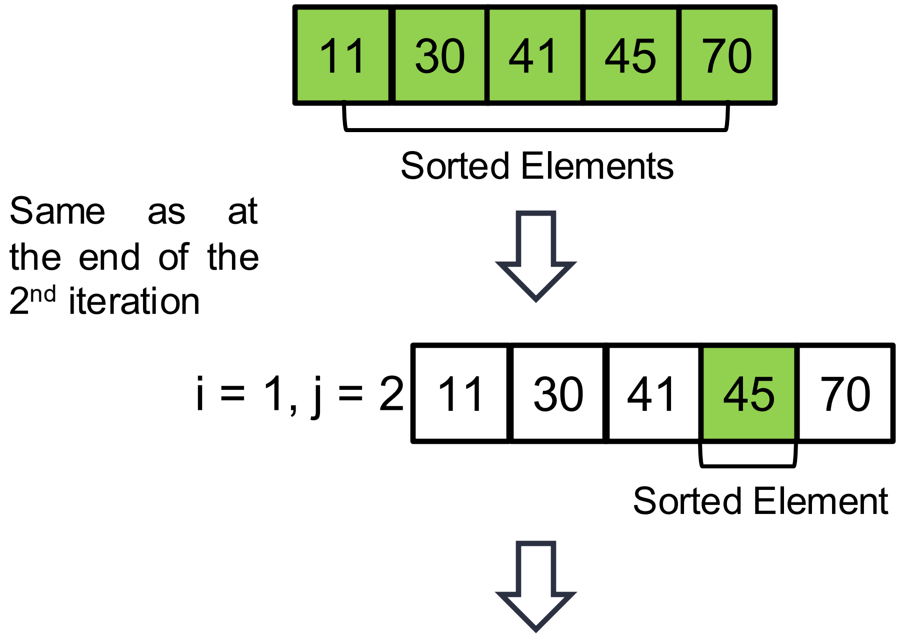
No Swap

i = 2, j = 3	11	30	41	45	70

No Swap

Sorting: Bubble Sort

The 4th iteration is the same as the 3rd



Without early stopping condition, the algorithm is costly.

Implementation

```python
# Bubble sort with early stopping – Complexity O(n²)
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j] #Swap
                swapped = True
        if not swapped:
            break
```

No swaps means the array is sorted

Sorting: Selection Sort

- **Selection Sort** works by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element.
- **Selection Sort** algorithm:

Step 1:

Find the **smallest element** and swap it with the **first element**.

Step 2:

Find the **smallest among remaining elements** and swap it with the **second element**.

Step 3:

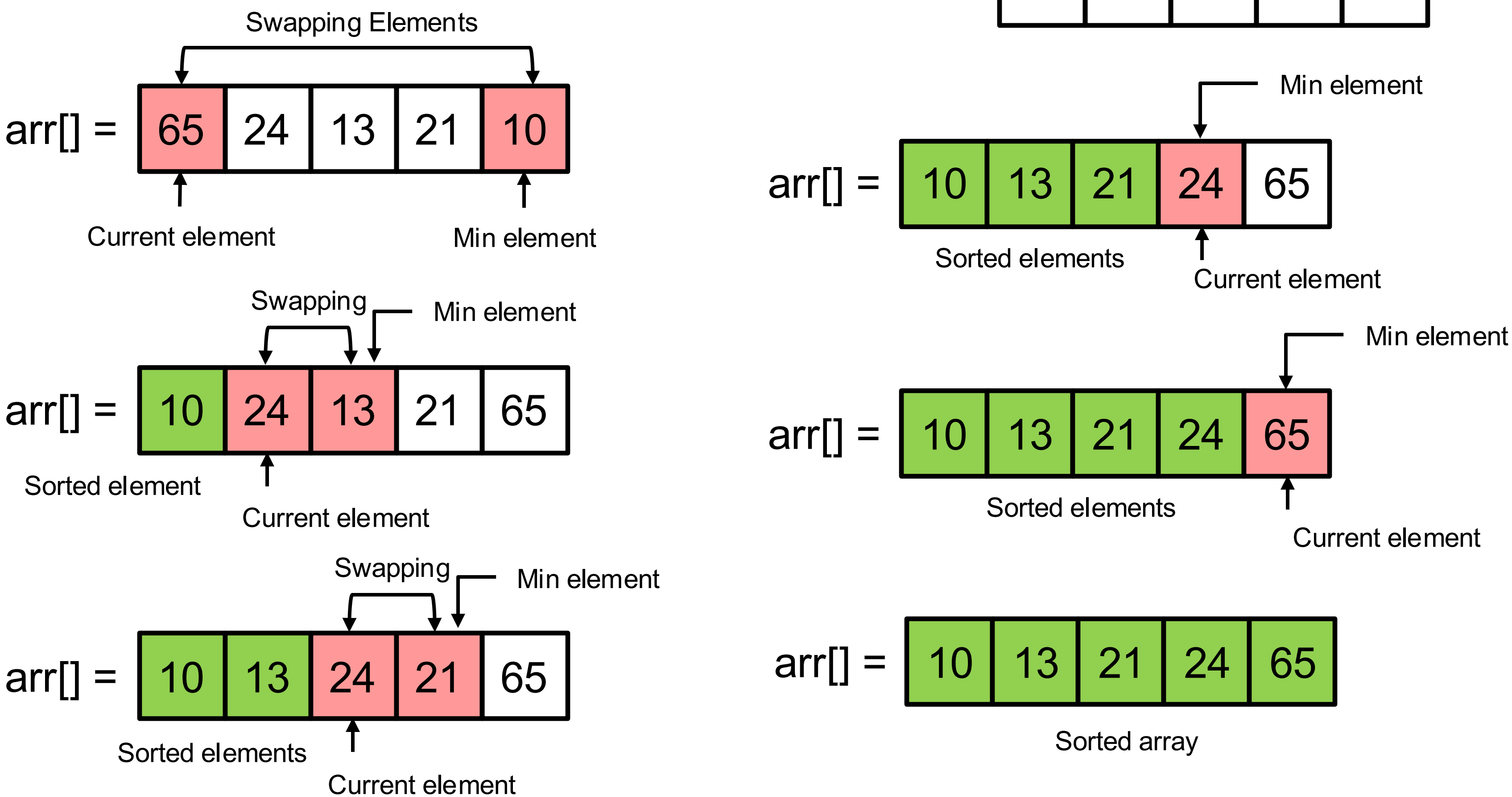
The process is repeated until the array is sorted.

Sorting: Selection Sort

Let's take a look at how the selection sort algorithm operates!

Example 6: Let's take an unsorted array as input:

65	24	13	21	10



Implementation

```python
# Selection sort – Complexity O(n²)
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j # Update min_idx
        arr[i], arr[min_idx] = arr[min_idx], arr[i] # Swap
```

Sorting: Insertion Sort

- **Insertion Sort** works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.
- **Insertion Sort** algorithm:

  Consider only the **first element**, and thus, our list is sorted.

  Step 1:

  Compare the **second element** with the **first element.** If the **second element** is smaller then swap them.

  Step 2:

  Move to the **next element**, compare it with the **first two**, and put it in its correct position.

  Step 3:

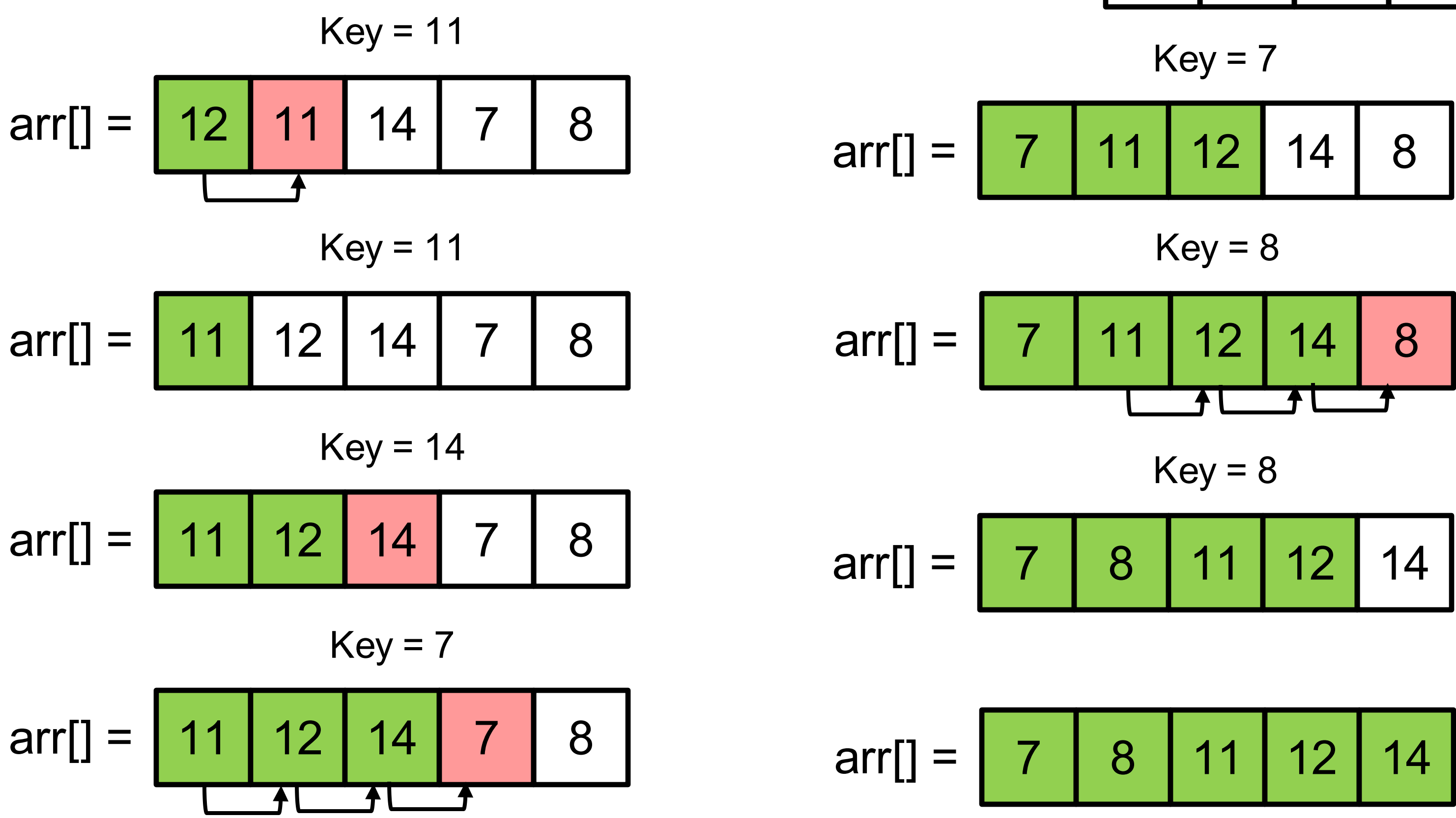
  Repeat until the array is sorted.

Sorting: Insertion Sort

Let's take a look at how the insertion sort algorithm operates!

Example 7: Let's take an unsorted array as input:

12	11	14	7	8



Implementation

```python
# Insertion sort – Complexity O(n²)
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i] # Current element to be sorted
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j] # Shift element > key to the right
            j -= 1
        arr[j+1] = key # Insert key at the correct position
```

Sorting: Merge Sort

- **Merge Sort** works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.
- **Merge Sort** algorithm:

Step 1: Divide

Divide the list or array recursively into two halves until it can no more be divided.

Step 2: Conquer

Each subarray is sorted individually using the merge sort algorithm.

Step 3: Merge

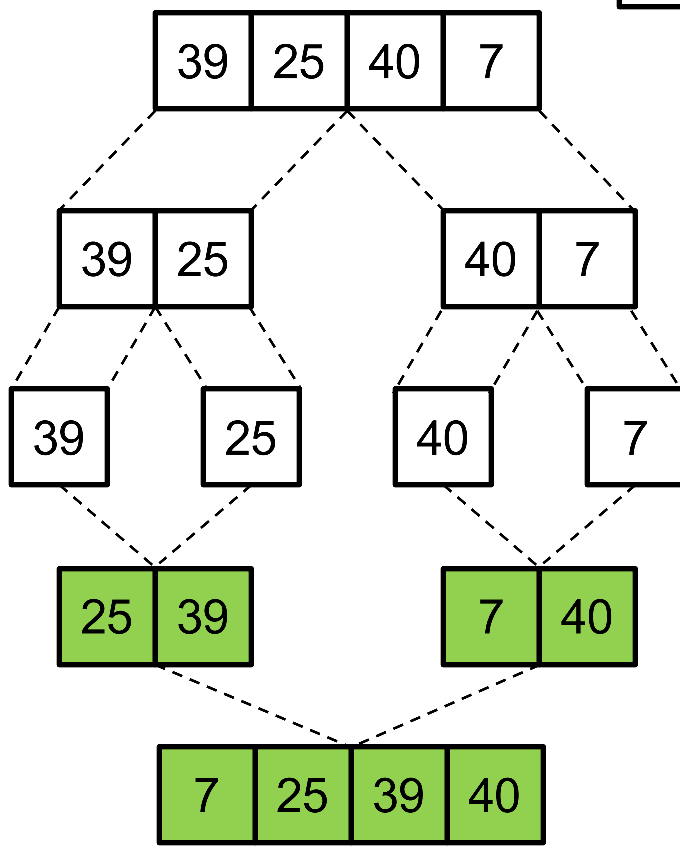
Sorted subarrays are merged back together in sorted order until all elements are combined.

Sorting: Merge Sort

Let's take a look at how the merge sort algorithm operates!

Example 8: Let's take an unsorted array as input:

39	25	40	7



Implementation

```
# Merge sort - Complexity O(n log n)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    sorted_left = merge_sort(left)
    sorted_right = merge_sort(right)

    return merge(sorted_left,
  sorted_right)
```

```
def merge(left, right):
    res = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            res.append(left[i])
            i += 1
        else:
            res.append(right[j])
            j += 1

    res.extend(left[i:])
    res.extend(right[j:])

    return res
```

Sorting: Quick Sort

- **Quick Sort** picks an element as a pivot and partitions the array around the pivot by placing the pivot in its correct position in the sorted array.
- It also works on the principle of **divide and conquer:**

Step 1: Choose a Pivot

Select an element from the array as pivot (first, last, median, random, etc.)

Step 2: Partition the Array

Re arrange the array around the pivot

Ex of partition algorithm: Naive Partition, Lomuto Partition, Hoare's Partition

Step 3: Recursively call

Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

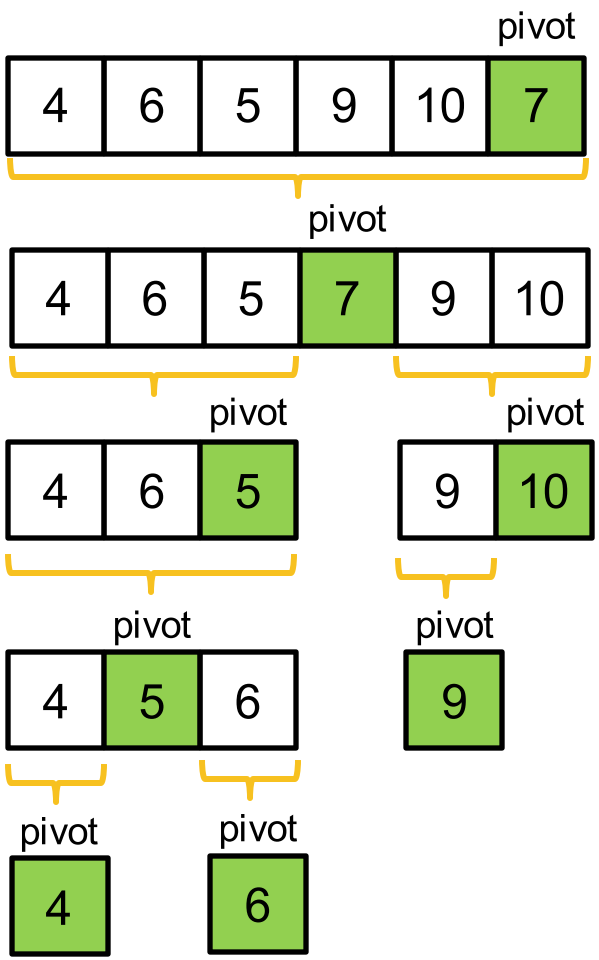
Base case:

The recursion stops when there is only one element left in the sub-array

Sorting: Quick Sort

Let's take a look at how the quick sort algorithm operates!

Example 9:



Implementation

```python
# Quick sort

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

Sorting Algorithm Comparison

Criteria	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Sorting strategy	Repeatedly swap adjacent elements	Find minimum and place at correct position	Insert each element into sorted prefix	Divide & merge sorted halves	Partition around pivot
Worst-case time	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Space complexity	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
Stable?	✅ Yes	❌ No	✅ Yes	✅ Yes	❌ No
Used in practice	❌ Rarely	❌ Rarely	⚠️ Internally used	✅ Yes	✅ Yes (as base for IntroSort)

Summary

Summary

- **Searching & Sorting**: Core building blocks for many algorithms and data structures
- **Searching**: Linear Search, Binary Search
- **Sorting**:
  - Simple comparison-based algorithms: Bubble, Selection, Insertion Sort
  - Efficient divide-and-conquer algorithms: Merge Sort, Quick Sort
- **Key takeaway for computational thinking**
  - Choose the right algorithm for the right context
  - Trade-off between simplicity vs. efficiency, clarity vs. performance
  - Understanding time complexity helps us design scalable solutions

Hands-On Exercises

Exercise I: Sorting Race

Implement multiple sorting algorithms and measure execution time using Python **time** module.

Exercise II: Median Matters

If we use the median element of the array as pivot in quicksort, how can we implement the algorithm? What are the pros compared to other choices of pivot?

Exercise III: Unsorted vs Sorted Search

Can we implement binary search with unsorted array? Compare the performance with linear search.