Sure! Let’s break this down clearly. Python has **built-in support for some sorting/searching**, but there are also classic algorithms you can implement manually. I’ll categorize them for clarity.

**1️⃣ Sorting Algorithms**

Sorting is arranging data in a particular order (ascending/descending).

**A. Built-in Python methods**

* sorted(list) → returns a new sorted list
* list.sort() → sorts the list **in place**

Example:

numbers = [5, 2, 9, 1]

sorted\_numbers = sorted(numbers) # [1, 2, 5, 9]

numbers.sort() # numbers is now [1, 2, 5, 9]

**B. Classic sorting algorithms**

| **Algorithm** | **Time Complexity (avg)** | **Stable** | **Notes** |
| --- | --- | --- | --- |
| **Bubble Sort** | O(n²) | Yes | Simple, compare & swap adjacent elements repeatedly |
| **Selection Sort** | O(n²) | No | Select min element & swap with current position |
| **Insertion Sort** | O(n²) | Yes | Build sorted part by inserting elements in correct place |
| **Merge Sort** | O(n log n) | Yes | Divide & conquer, recursive splitting |
| **Quick Sort** | O(n log n) avg | No | Divide & conquer, choose pivot & partition |
| **Heap Sort** | O(n log n) | No | Uses heap data structure |
| **Counting Sort** | O(n + k) | Yes | For integers in a small range |
| **Radix Sort** | O(nk) | Yes | For integers or strings, sort digit by digit |
| **Python Timsort** | O(n log n) avg | Yes | Python’s sorted() & .sort() use this internally |

**2️⃣ Search Algorithms**

Searching is **finding an element** in a data structure.

**A. Linear Search (Sequential Search)**

* Go through elements **one by one** until the target is found.
* Works on **unsorted or sorted** data.
* Time Complexity: O(n)

def linear\_search(arr, target):

for i, val in enumerate(arr):

if val == target:

return i

return -1

**B. Binary Search**

* Works on **sorted arrays only**.
* Check the **middle element**, discard half of the list each time.
* Time Complexity: O(log n)

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

**C. Other search algorithms**

| **Algorithm** | **Notes** |
| --- | --- |
| **Jump Search** | For sorted arrays, jumps fixed steps |
| **Exponential Search** | Useful for unbounded lists |
| **Fibonacci Search** | Based on Fibonacci numbers |
| **Hashing** | O(1) average lookup if hash table used |
| **Depth-First Search (DFS)** | For trees/graphs |
| **Breadth-First Search (BFS)** | For trees/graphs |

💡 **Summary Table:**

| **Type** | **Python Built-in** | **Classic / Manual** |
| --- | --- | --- |
| Sort | sorted(), list.sort() | Bubble, Merge, Quick, Heap, Insertion, Selection |
| Search | in, index() | Linear, Binary, Jump, Exponential, DFS, BFS |

If you want, I can **create a single Python file that implements all common sorting and searching algorithms** with examples, so you can **practice them in one place**.

Do you want me to do that?

Perfect! I’ll create a **comprehensive Python script** that implements the **most common sorting and searching algorithms** with examples, so you can run and practice them all in one file.

Here’s a clean version you can copy and run:

# =========================

# SORTING ALGORITHMS

# =========================

# 1. Bubble Sort

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

# 2. Selection Sort

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

arr[i], arr[min\_idx] = arr[min\_idx], arr[i]

return arr

# 3. Insertion Sort

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i-1

while j >=0 and key < arr[j]:

arr[j+1] = arr[j]

j -= 1

arr[j+1] = key

return arr

# 4. Merge Sort

def merge\_sort(arr):

if len(arr) > 1:

mid = len(arr)//2

L = arr[:mid]

R = arr[mid:]

merge\_sort(L)

merge\_sort(R)

i = j = k = 0

while i < len(L) and j < len(R):

if L[i] < R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

while i < len(L):

arr[k] = L[i]

i += 1

k += 1

while j < len(R):

arr[k] = R[j]

j += 1

k += 1

return arr

# 5. Quick Sort

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr)//2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

# =========================

# SEARCHING ALGORITHMS

# =========================

# 1. Linear Search

def linear\_search(arr, target):

for i, val in enumerate(arr):

if val == target:

return i

return -1

# 2. Binary Search (array must be sorted)

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right)//2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

# =========================

# EXAMPLES

# =========================

if \_\_name\_\_ == "\_\_main\_\_":

arr = [64, 34, 25, 12, 22, 11, 90]

target = 22

print("Original array:", arr)

print("\n--- Sorting ---")

print("Bubble Sort:", bubble\_sort(arr.copy()))

print("Selection Sort:", selection\_sort(arr.copy()))

print("Insertion Sort:", insertion\_sort(arr.copy()))

print("Merge Sort:", merge\_sort(arr.copy()))

print("Quick Sort:", quick\_sort(arr.copy()))

print("\n--- Searching ---")

print(f"Linear Search for {target}: Index", linear\_search(arr, target))

sorted\_arr = sorted(arr)

print(f"Binary Search for {target} in sorted array {sorted\_arr}: Index", binary\_search(sorted\_arr, target))

**✅ What this script does**

1. Implements **5 sorting algorithms**: Bubble, Selection, Insertion, Merge, Quick.
2. Implements **2 search algorithms**: Linear and Binary Search.
3. Provides **example usage** with the same array so you can see results immediately.
4. Uses .copy() to make sure each sort works independently without affecting the original array.

If you want, I can **also add DFS and BFS for tree/graph traversal** into this same file so you have **all the basic algorithms in one place** for practice.

Do you want me to do that?

Perfect! Let’s visualize **Merge Sort** and **Binary Search** step by step with diagrams so it’s easy to understand.

**1️⃣ Merge Sort Visualization**

**Original list:**

[8, 3, 7, 4, 9, 2]

**Step 1: Divide the list into halves repeatedly**

[8, 3, 7] [4, 9, 2]

[8] [3, 7] [4] [9, 2]

[8] [3] [7] [4] [9] [2]

**Step 2: Merge and sort each half**

[3, 7, 8] [2, 4, 9]

**Step 3: Merge the two sorted halves**

[2, 3, 4, 7, 8, 9]

✅ **Result:** Sorted list [2, 3, 4, 7, 8, 9]

**Key idea:** Split → Sort → Merge

**2️⃣ Binary Search Visualization**

**Sorted list:**

[2, 3, 4, 7, 8, 9]

**Target:** 7

**Step 1:** Find the middle element

Left: 0, Right: 5

Middle index: (0+5)//2 = 2

Middle element: 4

* 7 > 4 → search **right half** [7, 8, 9]

**Step 2:** Find middle of new sublist

Left: 3, Right: 5

Middle index: (3+5)//2 = 4

Middle element: 8

* 7 < 8 → search **left half** [7]

**Step 3:** Check the last element

Middle index: 3

Middle element: 7 → Found!

✅ **Result:** Target 7 found at index 3

**Key idea:** Repeatedly divide the list in half until you find the target.

If you want, I can **draw a linked diagram showing both Merge Sort and Binary Search visually**, almost like a “tree diagram,” which makes it **super easy to remember for interviews**.

Do you want me to do that?