

Python Programming - Sorting & Searching (Sắp Xếp và Tìm Kiếm)

Mục tiêu học tập: Làm chủ các thuật toán tìm kiếm và sắp xếp - từ cơ bản đến nâng cao, từ thuật toán thủ công đến built-in methods của Python.

1. Giới Thiệu

1.1. Tại Sao Cần Searching & Sorting?

Đặt vấn đề

Bạn có danh sách 1000 học sinh, làm sao để:

- **Tìm** một học sinh cụ thể? (Searching)
- **Sắp xếp** theo điểm từ cao xuống thấp? (Sorting)

```
students = [  
    {"name": "An", "score": 85},  
    {"name": "Bình", "score": 92},  
    {"name": "Chi", "score": 78},  
    # ... 997 học sinh khác  
]  
  
# Làm sao tìm "An" nhanh?  
# Làm sao sắp xếp theo điểm?
```

Giải pháp

Searching (Tìm kiếm): Tìm vị trí hoặc kiểm tra sự tồn tại của phần tử

- Linear Search - tìm tuần tự $O(n)$
- Binary Search - tìm nhị phân $O(\log n)$

Sorting (Sắp xếp): Sắp xếp các phần tử theo thứ tự

- Bubble Sort, Selection Sort, Insertion Sort
- Python built-in: `sorted()`, `.sort()`

1.2. Big O Notation - Độ Phức Tạp

Tại sao cần quan tâm?

```
# Cùng một nhiệm vụ, performance khác xa:
```

```
# Linear Search: O(n) - 1000 operations
for item in big_list: # 1000 items
    if item == target:
        break

# Binary Search: O(log n) - chỉ 10 operations!
# (với list đã sorted)
```

Bảng so sánh:

| Complexity | n=10 | n=100 | n=1000 | n=10000 | Example |
|------------|------|-------|---------|-----------|-------------------|
| O(1) | 1 | 1 | 1 | 1 | Dictionary lookup |
| O(log n) | 3 | 7 | 10 | 13 | Binary Search |
| O(n) | 10 | 100 | 1000 | 10000 | Linear Search |
| O(n²) | 100 | 10000 | 1000000 | 100000000 | Bubble Sort |

2. Searching (Tìm Kiếm)

2.1. Linear Search - Tìm Kiếm Tuần Tự

Ý tưởng

Duyệt từng phần tử từ đầu đến cuối cho đến khi tìm thấy.

Ví dụ: Tìm số 42 trong list

```
[10, 25, 42, 67, 88]
  ↑  ↑  ↑ Found!
Check từng phần tử cho đến khi gặp 42
```

Implementation

```
def linear_search(arr, target):
    """
    Tìm kiếm tuần tự - duyệt từng phần tử

    Args:
        arr: Danh sách cần tìm
        target: Giá trị cần tìm

    Returns:
        Index của phần tử nếu tìm thấy, -1 nếu không tìm thấy
    """
    for i in range(len(arr)):
        if arr[i] == target:
```

```

        return i # Tìm thấy, trả về index
    return -1 # Không tìm thấy

# Test
numbers = [10, 25, 42, 67, 88]

result = linear_search(numbers, 42)
print(f"Tìm thấy 42 ở index: {result}") # 2

result = linear_search(numbers, 100)
print(f"Tìm 100: {result}") # -1 (không có)

```

Ưu và Nhược điểm

☑ Ưu điểm:

- Đơn giản, dễ implement
- Hoạt động với mọi loại list (sorted hay không)
- Tốt cho list nhỏ

✗ Nhược điểm:

- Chậm với list lớn - $O(n)$
- Worst case: phải duyệt hết list

Use Cases

```

# 1. Tìm học sinh theo tên
students = ["An", "Bình", "Chi", "Dũng"]
index = linear_search(students, "Chi")
print(f"Chi ở vị trí: {index}") # 2

# 2. Kiểm tra sự tồn tại
def exists(arr, target):
    return linear_search(arr, target) != -1

print(exists(students, "An")) # True
print(exists(students, "Hoa")) # False

# 3. Tìm tất cả vị trí xuất hiện
def find_all(arr, target):
    """Tìm tất cả index của target"""
    indices = []
    for i in range(len(arr)):
        if arr[i] == target:
            indices.append(i)
    return indices

numbers = [5, 3, 7, 3, 9, 3]
result = find_all(numbers, 3)
print(f"Số 3 xuất hiện ở: {result}") # [1, 3, 5]

```

2.2. Binary Search - Tìm Kiếm Nhị Phân

Ý tưởng

Điều kiện: List phải được **sorted** (đã sắp xếp)

Chiến lược: Chia đôi search space mỗi lần - giống trò chơi đoán số!

Tìm 42 trong [10, 25, 42, 67, 88, 95]

Bước 1: Check giữa (67)

[10, 25, 42, 67, 88, 95]

↑

42 < 67 → Tìm bên trái

Bước 2: Check giữa (25)

[10, 25, 42]

↑

42 > 25 → Tìm bên phải

Bước 3: Check giữa (42)

[42]

↑

Found! Index = 2

Implementation - Iterative

```
def binary_search(arr, target):  
    """  
    Tìm kiếm nhị phân - chia đôi mỗi lần  
  
    ⚠ Điều kiện: arr PHẢI được sorted!  
  
    Args:  
        arr: Danh sách đã sorted  
        target: Giá trị cần tìm  
  
    Returns:  
        Index của phần tử nếu tìm thấy, -1 nếu không  
    """  
    left = 0  
    right = len(arr) - 1  
  
    while left <= right:  
        # Tìm middle index  
        mid = (left + right) // 2
```

```

    if arr[mid] == target:
        return mid # Tìm thấy!
    elif arr[mid] < target:
        # Target ở bên phải
        left = mid + 1
    else:
        # Target ở bên trái
        right = mid - 1

    return -1 # Không tìm thấy

# Test
numbers = [10, 25, 42, 67, 88, 95] # ⚠ Phải sorted!

result = binary_search(numbers, 42)
print(f"Tìm thấy 42 ở index: {result}") # 2

result = binary_search(numbers, 100)
print(f"Tìm 100: {result}") # -1

```

Implementation - Recursive

```

def binary_search_recursive(arr, target, left=0, right=None):
    """Binary search - phiên bản đệ quy"""
    if right is None:
        right = len(arr) - 1

    # Base case: không còn gì để tìm
    if left > right:
        return -1

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        # Tìm bên phải
        return binary_search_recursive(arr, target, mid + 1, right)
    else:
        # Tìm bên trái
        return binary_search_recursive(arr, target, left, mid - 1)

# Test
numbers = [10, 25, 42, 67, 88, 95]
result = binary_search_recursive(numbers, 67)
print(f"Tìm thấy 67 ở index: {result}") # 3

```

Visualization - Step by Step

```
def binary_search_verbose(arr, target):
    """Binary search với visualization"""
    left, right = 0, len(arr) - 1
    step = 1

    print(f"Tìm {target} trong {arr}")

    while left <= right:
        mid = (left + right) // 2

        print(f"\nBước {step}:")
        print(f"  Left={left}, Right={right}, Mid={mid}")
        print(f"  Checking arr[{mid}] = {arr[mid]}")

        if arr[mid] == target:
            print(f"  ✓ Tìm thấy tại index {mid}!")
            return mid
        elif arr[mid] < target:
            print(f"  {arr[mid]} < {target} → Tìm bên phải")
            left = mid + 1
        else:
            print(f"  {arr[mid]} > {target} → Tìm bên trái")
            right = mid - 1

        step += 1

    print(f"\nX Không tìm thấy {target}")
    return -1

# Demo
numbers = [10, 25, 42, 67, 88, 95]
binary_search_verbose(numbers, 67)

# Output:
# Tìm 67 trong [10, 25, 42, 67, 88, 95]
#
# Bước 1:
#   Left=0, Right=5, Mid=2
#   Checking arr[2] = 42
#   42 < 67 → Tìm bên phải
#
# Bước 2:
#   Left=3, Right=5, Mid=4
#   Checking arr[4] = 88
#   88 > 67 → Tìm bên trái
#
# Bước 3:
#   Left=3, Right=3, Mid=3
#   Checking arr[3] = 67
#   ✓ Tìm thấy tại index 3!
```

So Sánh: Linear vs Binary Search

```
import time

def measure_search_time(search_func, arr, target):
    """Đo thời gian tìm kiếm"""
    start = time.perf_counter()
    result = search_func(arr, target)
    end = time.perf_counter()
    return result, (end - start) * 1000 # milliseconds

# Test với list lớn
big_list = list(range(100000)) # 100,000 numbers
target = 99999 # Worst case - ở cuối

# Linear Search
result, time_linear = measure_search_time(linear_search, big_list, target)
print(f"Linear Search: {time_linear:.4f} ms")

# Binary Search
result, time_binary = measure_search_time(binary_search, big_list, target)
print(f"Binary Search: {time_binary:.4f} ms")

print(f"\nBinary nhanh gấp: {time_linear / time_binary:.0f} lần!")

# Output (example):
# Linear Search: 2.1500 ms
# Binary Search: 0.0020 ms
# Binary nhanh gấp: 1075 lần!
```

2.3. Python Built-in Search Methods

in Operator - Check Membership

```
# Đơn giản nhất - check có tồn tại không
numbers = [10, 25, 42, 67, 88]

if 42 in numbers:
    print("42 có trong list") # ✓

if 100 not in numbers:
    print("100 không có trong list") # ✓

# Với strings
students = ["An", "Bình", "Chi"]
print("An" in students) # True
print("Dũng" not in students) # True

# Performance: O(n) - tương đương Linear Search
```

`.index()` Method - Find Index

```
# Tìm index của phần tử
numbers = [10, 25, 42, 67, 88]

index = numbers.index(42)
print(f"42 ở index: {index}") # 2

# ⚠ Nếu không tìm thấy → ValueError
try:
    index = numbers.index(100)
except ValueError:
    print("Không tìm thấy 100")

# Safe version
def safe_index(arr, target):
    """Trả về index hoặc -1"""
    try:
        return arr.index(target)
    except ValueError:
        return -1

print(safe_index(numbers, 42)) # 2
print(safe_index(numbers, 100)) # -1
```

`.index()` với Start và End

```
# Tìm trong range cụ thể
numbers = [5, 3, 7, 3, 9, 3]

# Tìm 3 đầu tiên
first = numbers.index(3)
print(f"3 đầu tiên ở: {first}") # 1

# Tìm 3 thứ hai (bắt đầu sau index 1)
second = numbers.index(3, first + 1)
print(f"3 thứ hai ở: {second}") # 3

# Tìm 3 thứ ba
third = numbers.index(3, second + 1)
print(f"3 thứ ba ở: {third}") # 5

# Tìm tất cả index của một giá trị
def find_all_indices(arr, target):
    """Tìm tất cả vị trí của target"""
    indices = []
    start = 0
    while True:
```

```

    try:
        index = arr.index(target, start)
        indices.append(index)
        start = index + 1
    except ValueError:
        break
    return indices

result = find_all_indices(numbers, 3)
print(f"Tất cả vị trí của 3: {result}") # [1, 3, 5]

```

.count() Method - Count Occurrences

```

# Đếm số lần xuất hiện
numbers = [5, 3, 7, 3, 9, 3]

count = numbers.count(3)
print(f"Số 3 xuất hiện {count} lần") # 3

# Kết hợp với các methods khác
def most_frequent(arr):
    """Tìm phần tử xuất hiện nhiều nhất"""
    if not arr:
        return None

    unique_items = set(arr)
    max_count = 0
    most_freq = None

    for item in unique_items:
        count = arr.count(item)
        if count > max_count:
            max_count = count
            most_freq = item

    return most_freq, max_count

numbers = [1, 2, 3, 2, 4, 2, 5]
item, count = most_frequent(numbers)
print(f"Phần tử {item} xuất hiện nhiều nhất ({count} lần)")
# Phần tử 2 xuất hiện nhiều nhất (3 lần)

```

3. Sorting (Sắp Xếp)

3.1. Bubble Sort - Sắp Xếp Nổi Bọt

Ý tưởng

So sánh và swap các cặp phần tử liên tiếp. Phần tử lớn nhất "nổi" lên cuối sau mỗi vòng lặp.

Visualization:

```

Pass 1: [5, 3, 8, 4, 2]
      ↑ ↑
      swap → [3, 5, 8, 4, 2]
            ↑ ↑
            OK → [3, 5, 8, 4, 2]
              ↑ ↑
              swap → [3, 5, 4, 8, 2]
                    ↑ ↑
                    swap → [3, 5, 4, 2, 8]
                          ↑
                          8 đã đúng vị trí!

Pass 2: [3, 5, 4, 2, 8]
      ... tiếp tục ...

```

Implementation

```

def bubble_sort(arr):
    """
    Bubble Sort - sắp xếp nổi bọt

    Time Complexity: O(n²)
    Space Complexity: O(1)

    Args:
        arr: List cần sắp xếp
    """
    n = len(arr)

    # n-1 passes
    for i in range(n - 1):
        # Mỗi pass: so sánh và swap các cặp
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                # Swap
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Test
numbers = [64, 34, 25, 12, 22, 11, 90]
print("Original:", numbers)
bubble_sort(numbers)
print("Sorted:", numbers)
# Sorted: [11, 12, 22, 25, 34, 64, 90]

```

Optimized Version

```
def bubble_sort_optimized(arr):
    """
    Bubble Sort với optimization:
    - Dừng sớm nếu không có swap (đã sorted)
    """
    n = len(arr)

    for i in range(n - 1):
        swapped = False # Flag để track swap

        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Nếu không có swap nào → đã sorted
        if not swapped:
            print(f"Stopped early at pass {i + 1}")
            break

# Test với list gần như đã sorted
numbers = [1, 2, 4, 3, 5, 6]
print("Original:", numbers)
bubble_sort_optimized(numbers)
print("Sorted:", numbers)
# Stopped early at pass 2
```

Visualization với Print

```
def bubble_sort_verbose(arr):
    """Bubble sort với visualization"""
    n = len(arr)
    print(f"Original: {arr}\n")

    for i in range(n - 1):
        print(f"Pass {i + 1}:")
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                print(f"  Swap {arr[j]} and {arr[j + 1]}")
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        print(f"  Result: {arr}\n")

numbers = [5, 3, 8, 4, 2]
bubble_sort_verbose(numbers)

# Output:
# Original: [5, 3, 8, 4, 2]
#
# Pass 1:
#   Swap 5 and 3
```

```
# Swap 8 and 4
# Swap 8 and 2
# Result: [3, 5, 4, 2, 8]
#
# Pass 2:
# Swap 5 and 4
# Swap 5 and 2
# Result: [3, 4, 2, 5, 8]
# ...
```

3.2. Selection Sort - Sắp Xếp Chọn

Ý tưởng

Tìm phần tử nhỏ nhất, đưa về đầu. Lặp lại với phần còn lại.

```
[64, 25, 12, 22, 11]

Pass 1: Tìm min (11), swap với vị trí 0
[11, 25, 12, 22, 64]
✓   ↑ unsorted part

Pass 2: Tìm min (12), swap với vị trí 1
[11, 12, 25, 22, 64]
✓   ✓   ↑ unsorted

Pass 3: Tìm min (22), swap với vị trí 2
[11, 12, 22, 25, 64]
✓   ✓   ✓   ↑

...
```

Implementation

```
def selection_sort(arr):
    """
    Selection Sort - tìm min và đưa về đầu

    Time Complexity: O(n²)
    Space Complexity: O(1)
    """
    n = len(arr)

    for i in range(n - 1):
        # Tìm index của phần tử nhỏ nhất trong phần chưa sort
        min_index = i

        for j in range(i + 1, n):
```

```

        if arr[j] < arr[min_index]:
            min_index = j

    # Swap phần tử nhỏ nhất về vị trí i
    if min_index != i:
        arr[i], arr[min_index] = arr[min_index], arr[i]

# Test
numbers = [64, 25, 12, 22, 11]
print("Original:", numbers)
selection_sort(numbers)
print("Sorted:", numbers)
# Sorted: [11, 12, 22, 25, 64]

```

Verbose Version

```

def selection_sort_verbose(arr):
    """Selection sort với visualization"""
    n = len(arr)
    print(f"Original: {arr}\n")

    for i in range(n - 1):
        min_index = i

        # Tìm min
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        print(f"Pass {i + 1}:")
        print(f"  Min trong {arr[i:]} là {arr[min_index]} ở index {min_index}")

        # Swap
        if min_index != i:
            print(f"  Swap {arr[i]} với {arr[min_index]}")
            arr[i], arr[min_index] = arr[min_index], arr[i]

        print(f"  Result: {arr}")
        print(f"  Sorted part: {arr[:i+1]}\n")

    numbers = [64, 25, 12, 22, 11]
    selection_sort_verbose(numbers)

```

3.3. Insertion Sort - Sắp Xếp Chèn

Ý tưởng

Giống như sắp xếp bài khi chơi bài: lấy từng lá, chèn vào đúng vị trí trong phần đã sort.

```
[12, 11, 13, 5, 6]

[12]          ← Sorted
[11, 13, 5, 6] ← Unsorted

Lấy 11, chèn vào [12]:
[11, 12]      ← Sorted
[13, 5, 6]    ← Unsorted

Lấy 13, chèn vào [11, 12]:
[11, 12, 13]  ← Sorted
[5, 6]        ← Unsorted

...
```

Implementation

```
def insertion_sort(arr):
    """
    Insertion Sort - chèn từng phần tử vào đúng vị trí

    Time Complexity:  $O(n^2)$  worst case,  $O(n)$  best case (đã sorted)
    Space Complexity:  $O(1)$ 

    ✓ Tốt cho list nhỏ hoặc gần như đã sorted
    """
    n = len(arr)

    # Bắt đầu từ phần tử thứ 2 (index 1)
    for i in range(1, n):
        key = arr[i] # Phần tử cần chèn
        j = i - 1    # Vị trí so sánh

        # Dịch các phần tử lớn hơn key sang phải
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Chèn key vào đúng vị trí
        arr[j + 1] = key

    # Test
    numbers = [12, 11, 13, 5, 6]
    print("Original:", numbers)
    insertion_sort(numbers)
    print("Sorted:", numbers)
    # Sorted: [5, 6, 11, 12, 13]
```

Verbose Version

```
def insertion_sort_verbose(arr):
    """Insertion sort với visualization"""
    print(f"Original: {arr}\n")

    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        print(f"Pass {i}:")
        print(f"  Key = {key}")
        print(f"  Sorted part: {arr[:i]}")
        print(f"  Inserting {key}...")

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
        print(f"  Result: {arr}")
        print(f"  Sorted part now: {arr[:i+1]}\n")

numbers = [12, 11, 13, 5, 6]
insertion_sort_verbose(numbers)
```

3.4. Comparison of Sorting Algorithms

```
import time
import random

def measure_sort_time(sort_func, arr):
    """Đo thời gian sort"""
    arr_copy = arr.copy() # Copy để không ảnh hưởng arr gốc
    start = time.perf_counter()
    sort_func(arr_copy)
    end = time.perf_counter()
    return (end - start) * 1000 # milliseconds

# Test với các kích thước khác nhau
sizes = [100, 500, 1000]

for size in sizes:
    # Random array
    arr = [random.randint(1, 1000) for _ in range(size)]

    print(f"\n=== Array size: {size} ===")

    time_bubble = measure_sort_time(bubble_sort, arr)
    print(f"Bubble Sort:    {time_bubble:.2f} ms")
```

```
time_selection = measure_sort_time(selection_sort, arr)
print(f"Selection Sort: {time_selection:.2f} ms")

time_insertion = measure_sort_time(insertion_sort, arr)
print(f"Insertion Sort: {time_insertion:.2f} ms")

# Output (example):
# === Array size: 100 ===
# Bubble Sort:      2.50 ms
# Selection Sort:   1.80 ms
# Insertion Sort:   1.20 ms
#
# === Array size: 500 ===
# Bubble Sort:     58.30 ms
# Selection Sort:  42.10 ms
# Insertion Sort:  28.50 ms
#
# === Array size: 1000 ===
# Bubble Sort:    230.20 ms
# Selection Sort: 167.50 ms
# Insertion Sort: 110.80 ms
```

So sánh:

| Algorithm | Time (Avg) | Time (Worst) | Space | Stable | Best For |
|----------------|---------------|---------------|--------|--------|---------------|
| Bubble | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✓ | Educational |
| Selection | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✗ | Few writes |
| Insertion | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✓ | Nearly sorted |
| Python .sort() | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | ✓ | Production |

4. Python Built-in Sorting

4.1. sorted() Function - Return New List

```
# sorted() tạo list mới, không thay đổi list gốc
numbers = [5, 2, 8, 1, 9]

sorted_numbers = sorted(numbers)
print("Original:", numbers)      # [5, 2, 8, 1, 9]
print("Sorted:", sorted_numbers) # [1, 2, 5, 8, 9]

# Descending order
desc = sorted(numbers, reverse=True)
print("Descending:", desc)      # [9, 8, 5, 2, 1]

# Works with any iterable
text = "python"
```

```
sorted_chars = sorted(text)
print(sorted_chars) # ['h', 'n', 'o', 'p', 't', 'y']

# Tuple
tuple_nums = (5, 2, 8, 1, 9)
sorted_tuple = sorted(tuple_nums)
print(sorted_tuple) # [1, 2, 5, 8, 9] - returns list!
```

4.2. `.sort()` Method - In-place Sorting

```
# .sort() thay đổi list gốc, không return gì (None)
numbers = [5, 2, 8, 1, 9]

numbers.sort()
print(numbers) # [1, 2, 5, 8, 9] - đã thay đổi!

# Descending
numbers.sort(reverse=True)
print(numbers) # [9, 8, 5, 2, 1]

# ⚠ Common mistake
numbers = [5, 2, 8, 1, 9]
result = numbers.sort() # ✗ Sai!
print(result) # None - không return gì

# ☑ Đúng
numbers.sort()
print(numbers) # Dùng numbers trực tiếp
```

4.3. Sorting với `key` Parameter

Sort Strings by Length

```
# Sort theo độ dài string
words = ["python", "is", "awesome", "language"]

# Default: alphabetical
print(sorted(words))
# ['awesome', 'is', 'language', 'python']

# By length
sorted_by_length = sorted(words, key=len)
print(sorted_by_length)
# ['is', 'python', 'awesome', 'language']

# By length descending
sorted_desc = sorted(words, key=len, reverse=True)
print(sorted_desc)
# ['language', 'awesome', 'python', 'is']
```

Sort với Lambda Functions

```
# Sort tuples theo phần tử thứ 2
pairs = [(1, 5), (3, 2), (2, 8), (4, 1)]

sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)
# [(4, 1), (3, 2), (1, 5), (2, 8)]

# Sort strings case-insensitive
names = ["Alice", "bob", "Charlie", "david"]

sorted_names = sorted(names, key=str.lower)
print(sorted_names)
# ['Alice', 'bob', 'Charlie', 'david']

# Sort by absolute value
numbers = [-5, 2, -8, 1, -3, 9]

sorted_abs = sorted(numbers, key=abs)
print(sorted_abs)
# [1, 2, -3, -5, -8, 9]
```

Sort Complex Objects

```
# Sort dictionaries
students = [
    {"name": "An", "score": 85},
    {"name": "Bình", "score": 92},
    {"name": "Chi", "score": 78},
    {"name": "Dũng", "score": 92}
]

# Sort by score
sorted_by_score = sorted(students, key=lambda s: s["score"])
print("By score:")
for s in sorted_by_score:
    print(f"  {s['name']}: {s['score']}")

# Output:
# By score:
#   Chi: 78
#   An: 85
#   Bình: 92
#   Dũng: 92

# Sort by name
```

```

sorted_by_name = sorted(students, key=lambda s: s["name"])
print("\nBy name:")
for s in sorted_by_name:
    print(f"  {s['name']}: {s['score']}")

# Sort by multiple criteria (score desc, then name asc)
sorted_multi = sorted(students,
                      key=lambda s: (-s["score"], s["name"]))
print("\nBy score (desc) then name (asc):")
for s in sorted_multi:
    print(f"  {s['name']}: {s['score']}")

# Output:
#   Bình: 92
#   Dũng: 92
#   An: 85
#   Chi: 78

```

Using operator Module

```

from operator import itemgetter, attrgetter

# itemgetter - cho dict/tuple
students = [
    {"name": "An", "score": 85},
    {"name": "Bình", "score": 92},
    {"name": "Chi", "score": 78}
]

# Lambda version
sorted_lambda = sorted(students, key=lambda s: s["score"])

# itemgetter version - nhanh hơn!
sorted_itemgetter = sorted(students, key=itemgetter("score"))

print(sorted_itemgetter)

# Sort by multiple keys
sorted_multi = sorted(students,
                     key=itemgetter("score", "name"))

# attrgetter - cho objects
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def __repr__(self):
        return f"Student({self.name}, {self.score})"

students_obj = [

```

```

    Student("An", 85),
    Student("Bình", 92),
    Student("Chi", 78)
]

sorted_obj = sorted(students_obj, key=attrgetter("score"))
print(sorted_obj)
# [Student(Chi, 78), Student(An, 85), Student(Bình, 92)]

```

5. Advanced Sorting Techniques

5.1. Multi-Level Sorting

```

# Sort by nhiều tiêu chí
students = [
    {"name": "An", "grade": "A", "score": 92},
    {"name": "Bình", "grade": "B", "score": 85},
    {"name": "Chi", "grade": "A", "score": 95},
    {"name": "Dũng", "grade": "B", "score": 88},
]

# Sort by grade, then score (descending)
sorted_students = sorted(students,
                        key=lambda s: (s["grade"], -s["score"]))

print("Sorted by grade, then score (desc):")
for s in sorted_students:
    print(f"  {s['grade']} - {s['name']}: {s['score']}")

# Output:
# A - Chi: 95
# A - An: 92
# B - Dũng: 88
# B - Bình: 85

```

5.2. Custom Comparison Functions

```

from functools import cmp_to_key

# Custom comparison function
def compare_students(s1, s2):
    """
    Compare 2 students:
    1. Higher score comes first
    2. If equal score, alphabetically by name

    Returns:
        negative: s1 < s2
    """

```

```

        zero: s1 == s2
        positive: s1 > s2
    """
    # Compare by score (descending)
    if s1["score"] != s2["score"]:
        return s2["score"] - s1["score"] # Descending

    # If scores equal, compare by name (ascending)
    if s1["name"] < s2["name"]:
        return -1
    elif s1["name"] > s2["name"]:
        return 1
    else:
        return 0

students = [
    {"name": "An", "score": 85},
    {"name": "Bình", "score": 92},
    {"name": "Chi", "score": 85},
    {"name": "Dũng", "score": 92}
]

sorted_students = sorted(students, key=cmp_to_key(compare_students))

for s in sorted_students:
    print(f"{s['name']}: {s['score']}")

# Output:
# Bình: 92
# Dũng: 92
# An: 85
# Chi: 85

```

5.3. Stable vs Unstable Sorting

```

# Python's sort is STABLE - preserves relative order of equal elements

students = [
    {"name": "An", "score": 85, "id": 1},
    {"name": "Bình", "score": 85, "id": 2},
    {"name": "Chi", "score": 85, "id": 3},
]

# Sort by score only
sorted_students = sorted(students, key=lambda s: s["score"])

print("After sorting by score:")
for s in sorted_students:
    print(f"ID {s['id']}: {s['name']} - {s['score']}")

# Output (original order preserved for equal scores):

```

```
# ID 1: An - 85
# ID 2: Bình - 85
# ID 3: Chi - 85
```

6. Real-World Applications

6.1. Top K Elements

```
def top_k_scores(students, k):
    """
    Tìm top K học sinh có điểm cao nhất
    """
    sorted_students = sorted(students,
                             key=lambda s: s["score"],
                             reverse=True)
    return sorted_students[:k]

students = [
    {"name": "An", "score": 85},
    {"name": "Bình", "score": 92},
    {"name": "Chi", "score": 78},
    {"name": "Dũng", "score": 95},
    {"name": "Hoa", "score": 88}
]

top_3 = top_k_scores(students, 3)
print("Top 3 students:")
for i, s in enumerate(top_3, 1):
    print(f"{i}. {s['name']}: {s['score']}")

# Output:
# 1. Dũng: 95
# 2. Bình: 92
# 3. Hoa: 88
```

6.2. Leaderboard System

```
class Leaderboard:
    def __init__(self):
        self.scores = {}

    def add_score(self, player, score):
        """Thêm hoặc update điểm"""
        if player not in self.scores:
            self.scores[player] = []
        self.scores[player].append(score)

    def get_top_scores(self, k=10):
```

```

    """Lấy top K điểm cao nhất"""
    all_scores = []
    for player, scores in self.scores.items():
        for score in scores:
            all_scores.append({"player": player, "score": score})

    # Sort by score descending
    sorted_scores = sorted(all_scores,
                           key=lambda x: x["score"],
                           reverse=True)
    return sorted_scores[:k]

def get_player_rank(self, player):
    """Tìm thứ hạng của player"""
    top_scores = self.get_top_scores(k=float('inf'))

    for rank, entry in enumerate(top_scores, 1):
        if entry["player"] == player:
            return rank, entry["score"]

    return None, None

# Demo
board = Leaderboard()
board.add_score("Alice", 100)
board.add_score("Bob", 150)
board.add_score("Alice", 200)
board.add_score("Charlie", 180)
board.add_score("Bob", 120)

print("=== TOP 3 SCORES ===")
for i, entry in enumerate(board.get_top_scores(3), 1):
    print(f"{i}. {entry['player']}: {entry['score']}")

rank, score = board.get_player_rank("Alice")
print(f"\nAlice's best rank: #{rank} ({score} points)")

```

6.3. Merge Sorted Lists

```

def merge_sorted_lists(list1, list2):
    """
    Merge 2 sorted lists thành 1 sorted list
    Time: O(n + m)
    """
    result = []
    i, j = 0, 0

    # Merge while both lists have elements
    while i < len(list1) and j < len(list2):
        if list1[i] <= list2[j]:
            result.append(list1[i])

```

```

        i += 1
    else:
        result.append(list2[j])
        j += 1

    # Add remaining elements
    result.extend(list1[i:])
    result.extend(list2[j:])

    return result

# Test
list1 = [1, 3, 5, 7]
list2 = [2, 4, 6, 8, 10]

merged = merge_sorted_lists(list1, list2)
print(f"Merged: {merged}")
# [1, 2, 3, 4, 5, 6, 7, 8, 10]

# Merge multiple sorted lists
def merge_k_sorted_lists(lists):
    """Merge K sorted lists"""
    if not lists:
        return []

    result = lists[0]
    for i in range(1, len(lists)):
        result = merge_sorted_lists(result, lists[i])

    return result

lists = [
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9]
]

merged = merge_k_sorted_lists(lists)
print(f"Merged K lists: {merged}")
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

```

7. Performance Tips & Best Practices

7.1. When to Use What

```

# ☒ Use built-in sorted() or .sort()
numbers = [5, 2, 8, 1, 9]
sorted_nums = sorted(numbers) # ALWAYS use this in production!

# ☐ Don't implement your own sort for production

```

```
# bubble_sort(numbers) # Only for learning!

# ☒ Use binary search on sorted data
sorted_nums = sorted(numbers)
index = binary_search(sorted_nums, 8)

# ☒ Don't use linear search on large sorted data
# index = linear_search(numbers, 8) # Slow!

# ☒ Use 'in' for membership check on small lists
if 5 in numbers: # OK for small lists
    pass

# ☒ Use set for membership check on large lists
large_list = list(range(100000))
large_set = set(large_list) # Convert to set

# Slow
if 99999 in large_list: # O(n)
    pass

# Fast
if 99999 in large_set: # O(1)
    pass
```

7.2. Sort Optimization

```
# ☒ Sorting multiple times
students = [...]

by_score = sorted(students, key=lambda s: s["score"])
by_name = sorted(students, key=lambda s: s["name"])
by_age = sorted(students, key=lambda s: s["age"])

# ☒ Pre-compute keys if sorting multiple times
students_with_keys = [
    (s["score"], s["name"], s["age"], s)
    for s in students
]

by_score = sorted(students_with_keys, key=lambda x: x[0])
by_name = sorted(students_with_keys, key=lambda x: x[1])

# ☒ Use itemgetter for better performance
from operator import itemgetter

sorted_students = sorted(students, key=itemgetter("score"))
```

7.3. Memory Considerations

```
# sorted() creates new list - uses more memory
big_list = list(range(1000000))
sorted_list = sorted(big_list) # 2x memory!

# .sort() sorts in-place - saves memory
big_list.sort() # 1x memory only

# For very large data, consider:
# 1. Sort in-place with .sort()
# 2. Use generators when possible
# 3. Sort in chunks
```

8. Bài Tập Thực Hành

Bài 1: Student Grade System

```
"""
Cho danh sách students:
[
    {"name": "An", "scores": [85, 90, 88]},
    {"name": "Bình", "scores": [92, 88, 95]},
    {"name": "Chi", "scores": [78, 82, 80]}
]

Viết functions:
1. calculate_average(student) - tính điểm TB
2. find_top_student(students) - tìm học sinh điểm TB cao nhất
3. sort_by_average(students) - sắp xếp theo điểm TB giảm dần
4. find_student(students, name) - tìm học sinh theo tên
"""

# Solution
def calculate_average(student):
    """Tính điểm trung bình"""
    return sum(student["scores"]) / len(student["scores"])

def find_top_student(students):
    """Tìm học sinh có điểm TB cao nhất"""
    if not students:
        return None

    top_student = max(students, key=calculate_average)
    return top_student

def sort_by_average(students):
    """Sắp xếp theo điểm TB giảm dần"""
    return sorted(students, key=calculate_average, reverse=True)

def find_student(students, name):
```

```

    """Tìm học sinh theo tên"""
    for student in students:
        if student["name"].lower() == name.lower():
            return student
    return None

# Test
students = [
    {"name": "An", "scores": [85, 90, 88]},
    {"name": "Bình", "scores": [92, 88, 95]},
    {"name": "Chi", "scores": [78, 82, 80]}
]

top = find_top_student(students)
print(f"Top student: {top['name']} (avg: {calculate_average(top):.2f})")

sorted_students = sort_by_average(students)
print("\nRanking:")
for i, s in enumerate(sorted_students, 1):
    print(f"{i}. {s['name']}: {calculate_average(s):.2f}")

found = find_student(students, "chi")
print(f"\nFound: {found}")

```

Bài 2: Product Inventory

```

"""
Hệ thống quản lý kho hàng:

products = [
    {"id": 1, "name": "Laptop", "price": 1200, "stock": 5},
    {"id": 2, "name": "Mouse", "price": 25, "stock": 50},
    {"id": 3, "name": "Keyboard", "price": 75, "stock": 30}
]

Implement:
1. find_product_by_id(products, product_id)
2. sort_by_price(products, ascending=True)
3. low_stock_alert(products, threshold=10)
4. most_expensive(products, k=3)
"""

# Solution
def find_product_by_id(products, product_id):
    """Tìm sản phẩm theo ID - binary search nếu sorted"""
    # Linear search (for unsorted)
    for product in products:
        if product["id"] == product_id:
            return product
    return None

```

```
def sort_by_price(products, ascending=True):
    """Sắp xếp theo giá"""
    return sorted(products,
                   key=lambda p: p["price"],
                   reverse=not ascending)

def low_stock_alert(products, threshold=10):
    """Tìm sản phẩm sắp hết hàng"""
    low_stock = [p for p in products if p["stock"] < threshold]
    return sorted(low_stock, key=lambda p: p["stock"])

def most_expensive(products, k=3):
    """Top K sản phẩm đắt nhất"""
    sorted_products = sorted(products,
                              key=lambda p: p["price"],
                              reverse=True)
    return sorted_products[:k]

# Test
products = [
    {"id": 1, "name": "Laptop", "price": 1200, "stock": 5},
    {"id": 2, "name": "Mouse", "price": 25, "stock": 50},
    {"id": 3, "name": "Keyboard", "price": 75, "stock": 30},
    {"id": 4, "name": "Monitor", "price": 300, "stock": 8},
    {"id": 5, "name": "Webcam", "price": 80, "stock": 3}
]

print("=== LOW STOCK ALERT ===")
for p in low_stock_alert(products):
    print(f"{p['name']}: {p['stock']} left")

print("\n=== TOP 3 MOST EXPENSIVE ===")
for p in most_expensive(products, 3):
    print(f"{p['name']}: ${p['price']}")
```

9. Tổng Kết

☒ Kiến thức cần nắm vững

Searching:

- ☐ Linear Search - $O(n)$
- ☐ Binary Search - $O(\log n)$, requires sorted
- ☐ Python built-in: `in`, `.index()`, `.count()`

Sorting:

- ☐ Bubble Sort - $O(n^2)$, educational
- ☐ Selection Sort - $O(n^2)$, few writes
- ☐ Insertion Sort - $O(n^2)$, good for nearly sorted

- ☐ Python built-in: `sorted()`, `.sort()`
- ☐ Custom sorting with `key` parameter
- ☐ Lambda functions for sorting

Best Practices:

- ☐ Always use built-in `sorted()` or `.sort()` in production
- ☐ Use binary search on sorted data
- ☐ Use `key` parameter for custom sorting
- ☐ Consider memory: `sorted()` vs `.sort()`
- ☐ Profile performance for large datasets

🔗 Khi nào dùng gì?

| Task | Method | Note |
|--------------------|-------------------------|-------------------------|
| Check exists | <code>in</code> | O(n), simple |
| Find index | <code>.index()</code> | O(n), raises ValueError |
| Sort small list | <code>.sort()</code> | In-place, O(n log n) |
| Sort need original | <code>sorted()</code> | Returns new list |
| Sort custom | <code>key=lambda</code> | Most flexible |
| Search sorted | Binary search | O(log n) |
| Search unsorted | Linear search | O(n) |