Lecture Notes > Sorting Algorithms

Students Guide to Selection Sort & Bubble Sort

---

1. Why Sorting Matters in Real Life 🎯

Imagine you're organizing your closet:

- **Sorted clothes** = Easy to find your favorite shirt.
- **Unsorted mess** = Wasting time searching every day.

In the digital world, sorting helps with:

1. **Searching quickly:** Like finding a contact in your phone.
2. **Analyzing data:** Spotify uses sorting to show "Top 50" songs.
3. **Efficiency:** Delivery apps sort routes to save time and fuel.

Fun Fact: Every time you filter products by price on Amazon, sorting algorithms are working behind the scenes!

---

2. Selection Sort: The "Treasure Hunt" Method 🔍

What is Selection Sort?

- **Simple idea:** Find the biggest (or smallest) item in your list, put it in the correct spot, and repeat.
- **Analogy:** Imagine picking the reddest apple from a basket, placing it aside, then repeating until all apples are sorted.

How It Works (Step-by-Step)

Let's sort `[6, 2, 9, 4]` **ascending**:

1. **Step 1 (Find Largest):**
    - Compare all items: 6 vs 2 vs 9 vs 4.
    - **Largest = 9** (at position 2).
    - Swap 9 with the last item (4): `[6, 2, 4, 9]`.
2. **Step 2 (Repeat for Remaining):**
    - Now ignore 9 (already sorted).
    - Find the largest in `[6, 2, 4]` → **6** (position 0).
    - Swap 6 with the last unsorted item (4): `[4, 2, 6, 9]`.
3. **Step 3:**
    - Find the largest in `[4, 2]` → **4** (position 0).
    - Swap 4 with 2: `[2, 4, 6, 9]`.
4. **Done!** Only 1 item left (2).

Code Walkthrough

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n-1, 0, -1):  # Start from the end
        max_index = 0  # Assume first item is the largest
        for j in range(1, i+1):  # Check remaining items
            if arr[j] > arr[max_index]:
                max_index = j  # Update if bigger item found
        # Swap the largest item to its correct position
```

```
        arr[i], arr[max_index] = arr[max_index], arr[i]
    return arr
```

Line-by-Line Explanation:

- `for i in range(n-1, 0, -1):` → Work backward from the end of the list.
- `max_index = 0` → Start by assuming the first item is the largest.
- The inner loop (`for j in range(...)`) finds the true largest item.
- `arr[i], arr[max_index] = ...` → Swap the largest item into its correct spot.

Key Points to Remember ✓

✅ **Time Complexity:** Always $O(n^2)$ → Inefficient for large lists (e.g., sorting 1000 items takes ~500k operations).
✅ **Use When:** The list is small (e.g., less than 20 items).
✅ **Stability:** Not stable → Equal items might swap places.

---

3. Bubble Sort: The "Swap Party" Technique 🛁

What is Bubble Sort?

- **Simple idea:** Compare neighboring items. If they're in the wrong order, swap them. Repeat until no swaps are needed.
- **Analogy:** Like bubbles rising in soda – bigger numbers "float" to the top.

How It Works (Step-by-Step)

Let's sort `[5, 1, 4, 2, 8]` **ascending**:

Pass 1:

- Compare 5 & 1 → Swap → `[1, 5, 4, 2, 8]`

- Compare 5 & 4 → Swap → `[1, 4, 5, 2, 8]`
- Compare 5 & 2 → Swap → `[1, 4, 2, 5, 8]`
- Compare 5 & 8 → No swap.
  Result: Largest item (8) is at the end.

Pass 2:

- Compare 1 & 4 → No swap.
- Compare 4 & 2 → Swap → `[1, 2, 4, 5, 8]`
- Compare 4 & 5 → No swap.
  Result: Second largest (5) is in place.

Pass 3:

- No swaps are needed → Sorting stops early.

Optimized Code

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1, 0, -1):  # Reduce unsorted region each pass
        swapped = False
        for j in range(i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swap
                swapped = True
        if not swapped:  # Early exit if no swaps
            break
    return arr
```

Why the `swapped` Flag?

- If no swaps occur in a pass, the list is already sorted.
- Saves time! For `[1, 2, 3, 4]`, Bubble Sort finishes in **1 pass**.

Key Points to Remember ✓

✅ **Best Case:** O(n) → Fast for nearly sorted lists.
✅ **Worst Case:** $O(n^2)$ → Inefficient for reverse-sorted lists.
✅ **Stability:** Stable → Duplicate items stay in order.

---

4. Selection Sort vs. Bubble Sort: Face-Off ⚔️

Feature	Selection Sort	Bubble Sort
Speed (Best Case)	$O(n^2)$ → Always slow	O(n) → Fast if nearly sorted
Swaps	1 swap per pass → Efficient	Many swaps → Can be slow
Memory Usage	Low → Only needs 1 extra variable	Low → Similar to Selection Sort
Real-World Use	Small lists, limited memory	Simple apps, educational purposes

When to Use Which?

- **Selection Sort:** When swaps are costly, and the list is small.
- **Bubble Sort:** For nearly sorted lists or quick fixes.

---

5. Trick Questions (Test Yourself!)

Q1: *If you sort* `[3, 1, 4, 1, 5]` *with Selection Sort, will the two "1"s stay in the same order?*
A: ❌ No! Selection Sort is *not stable*. The first "1" might swap places with the second "1".

Q2: *Which algorithm is better for sorting* `[10, 9, 8, 7, 6]`*?*
A: Selection Sort! Both take $O(n^2)$ time, but Selection Sort makes fewer swaps.

---

6. At a Glance: Summary Cheat Sheet

Selection Sort

- Hunts for the largest/smallest item → Swaps it to the end/start.
- Good for: Small lists, minimal swaps.
- Code tip: Use `max_index` to track the largest item.

Bubble Sort

- Swaps neighbors → Bubbles up the largest item.
- Good for: Nearly sorted lists, simple code.
- Code tip: Use `swapped` flag for early exit.

---

7. Let's Practice! 🧑 💻

Exercise 1: Sort `[18, 5, 3, 22, 9]` using Selection Sort. Show all steps.
Exercise 2: Apply Bubble Sort to `[7, 3, 8, 2, 1]`. How many passes are needed?

Starter Code:

```
# Selection Sort
def selection_sort(arr):
```

```python
    # Your code here

# Bubble Sort
def bubble_sort(arr):
    # Your code here

# Test your code
arr1 = [18, 5, 3, 22, 9]
arr2 = [7, 3, 8, 2, 1]
print("Selection Sort Result:", selection_sort(arr1))
print("Bubble Sort Result:", bubble_sort(arr2))
```

Hints for Practice:

- For Exercise 1: Track the `max_index` for each pass.
- For Exercise 2: Count the passes and swaps.
