# 🔹 Sorting Algorithms – Coding Exercises

## 1. **Selection Sort Exercises**

**Goal:** Practice finding the minimum and swapping.

**Exercise 1.1 — Manual Minimum Swap**

* Write a function that takes a list and finds the smallest number in the list.
* Swap it with the first element.
* Print the modified list.

**Exercise 1.2 — Full Selection Sort**

* Extend your function so it keeps repeating the process to fully sort the list.
* Test it with [64, 25, 12, 22, 11].

**Exercise 1.3 — Step Tracker**

* Modify your function so that after each pass, it prints the list (so you can see how the list is sorted step by step).

## 2. **Insertion Sort Exercises**

**Goal:** Practice shifting elements into the right position.

**Exercise 2.1 — Insert One Card**

* Imagine a list that is partly sorted: [3, 7, 9, | 5] (the vertical bar separates sorted and unsorted).
* Write code to insert “5” into the correct place among the sorted numbers.

**Exercise 2.2 — Full Insertion Sort**

* Create a function that sorts an entire list using the insertion sort technique.
* Test it with [12, 11, 13, 5, 6].

**Exercise 2.3 — Counting Shifts**

* Modify your function so that it counts how many “shifts” (moving items one step right) occur during the sorting process.
* Print the total shifts used to sort the list.

## 3. **Bubble Sort Exercises**

**Goal:** Demonstrate repeated pairwise comparison and swapping.

**Exercise 3.1 — One Bubble Pass**

* Write a function that performs **just one pass** of bubble sort on [5, 1, 4, 2, 8].
* Print the result.

**Exercise 3.2 — Full Bubble Sort**

* Extend the code to keep repeating passes until the entire list is sorted.
* Print the list after each pass.

**Exercise 3.3 — Optimized Bubble Sort**

* Modify your bubble sort so it stops early if *no swaps* are made in a pass.
* Compare how many passes are needed on:
  + [1, 2, 3, 4, 5] (already sorted)
  + [5, 4, 3, 2, 1] (reverse order)

## 4. **Quicksort Exercises (Higher Level)**

**Goal:** Understand recursion and partitioning.

**Exercise 4.1 — Partition Step Only**

* Write a function that picks the **last element as pivot** and rearranges the list so smaller numbers are on the left, and larger numbers are on the right.
* Test with [10, 80, 30, 90, 40, 50, 70].

**Exercise 4.2 — Full Recursive Quicksort**

* Implement Quicksort using recursion.
* Test with [10, 7, 8, 9, 1, 5].

**Exercise 4.3 — Quicksort Variation**

* Change your implementation so the pivot is chosen at **random** instead of the last element.
* Compare the performance on different datasets.

# 🔹 **Extension Exercises – For Stronger Students**

These require deeper thinking about efficiency, complexity, and edge cases.

**Extension 1 — Timing Comparisons**

* Use Python’s time module to measure how long each algorithm takes to sort:
  + A list of 100 random numbers
  + A list of 100 numbers already sorted
  + A list of 100 numbers in reverse order
* Record and compare results.

**Extension 2 — Hybrid Sort**

* Write a function that uses **Insertion Sort** for small lists (length < 10) and **Quicksort** for larger lists.
* Test it on various list sizes.

**Extension 3 — Visualization Challenge**

* Using matplotlib, create a bar chart that updates at each step of a sorting algorithm (e.g., Bubble Sort).
* The bars should move as the sort progresses, visually showing the sorting process.

**Extension 4 — Compare and Reflect**

* Create a table showing for each algorithm:
  + Best case comparisons
  + Worst case comparisons
  + Memory usage
* Write a short reflection: Which one makes sense to use in real-world programming, and why?

If you are not sure where to start with the examples above then I have provided a starting template for each of the exercises to get you started.

# 🔹 Sorting Algorithms – Starter Code Templates (Python)

## 1. **Selection Sort**

**Exercise 1.1 — Manual Minimum Swap**

def selection\_one\_step(arr):  
 # TODO: Find the index of the smallest element in the list  
 min\_index = 0  
   
 # Loop through arr to find smallest item  
 for i in range(1, len(arr)):  
 # TODO: Compare element with arr[min\_index] and update if smaller  
 pass  
   
 # TODO: Swap the first element with smallest element found  
 pass  
  
 return arr  
  
print(selection\_one\_step([64, 25, 12, 22, 11]))

**Exercise 1.2 — Full Selection Sort**

def selection\_sort(arr):  
 # Repeat for each position in list  
 for i in range(len(arr)):  
 # TODO: find index of the smallest element in remaining list  
 min\_index = i  
 for j in range(i + 1, len(arr)):  
 pass # compare and update min\_index  
  
 # TODO: Swap arr[i] with smallest element  
 pass  
   
 return arr  
  
print(selection\_sort([64, 25, 12, 22, 11]))

## 2. **Insertion Sort**

**Exercise 2.1 — Insert One Card**

def insert\_one(sorted\_part, new\_item):  
 # Assume sorted\_part is already sorted.  
 i = len(sorted\_part) - 1  
   
 # TODO: Shift elements to the right until you find place for new\_item  
 while i >= 0 and sorted\_part[i] > new\_item:  
 pass  
  
 # TODO: Insert new\_item into correct position  
 pass  
  
 return sorted\_part  
  
print(insert\_one([3, 7, 9], 5))

**Exercise 2.2 — Full Insertion Sort**

def insertion\_sort(arr):  
 # Start with second element (first is "sorted")  
 for i in range(1, len(arr)):  
 current = arr[i]  
   
 # TODO: shift larger elements to the right  
 j = i - 1  
 while j >= 0 and arr[j] > current:  
 pass  
   
 # TODO: insert current in correct place  
 pass  
 return arr  
  
print(insertion\_sort([12, 11, 13, 5, 6]))

**Exercise 2.3 — Counting Shifts**

def insertion\_sort\_with\_count(arr):  
 shifts = 0  
   
 for i in range(1, len(arr)):  
 current = arr[i]  
 j = i - 1  
 while j >= 0 and arr[j] > current:  
 # Each move = one shift  
 shifts += 1  
 pass  
 pass  
   
 print("Shifts:", shifts)  
 return arr

## 3. **Bubble Sort**

**Exercise 3.1 — One Pass**

def bubble\_pass(arr):  
 # One full scan comparing pairs  
 for i in range(len(arr)-1):  
 # TODO: Compare arr[i] and arr[i+1], swap if needed  
 pass  
 return arr  
  
print(bubble\_pass([5, 1, 4, 2, 8]))

**Exercise 3.2 — Full Bubble Sort**

def bubble\_sort(arr):  
 n = len(arr)  
 for pass\_num in range(n - 1):  
 # TODO: One bubble pass through list  
 for i in range(n - 1):  
 pass # swap if necessary  
 print("After pass", pass\_num+1, arr)  
 return arr

**Exercise 3.3 — Optimized Bubble Sort**

def bubble\_sort\_optimized(arr):  
 n = len(arr)  
 for pass\_num in range(n - 1):  
 swapped = False  
 for i in range(n - 1):  
 pass # swap if needed and update swapped = True  
 print("After pass", pass\_num+1, arr)  
 if not swapped:  
 break  
 return arr

## 4. **Quicksort (Higher Level)**

**Exercise 4.1 — Partition Step**

def partition(arr, low, high):  
 pivot = arr[high] # last element as pivot  
 i = low - 1 # tracks smaller elements  
   
 for j in range(low, high):  
 # TODO: if arr[j] <= pivot, move it to left partition  
 pass  
   
 # TODO: Place pivot into correct position  
 pass  
   
 return i + 1  
  
print(partition([10, 80, 30, 90, 40, 50, 70], 0, 6))

**Exercise 4.2 — Full Recursive Quicksort**

def quicksort(arr, low, high):  
 if low < high:  
 # Partition  
 pi = partition(arr, low, high)  
   
 # TODO: Recursively sort partitions  
 pass  
   
 return arr  
  
print(quicksort([10, 7, 8, 9, 1, 5], 0, 5))

# 🔹 Extension Starters

**Extension 1 — Timing Comparisons**

import time, random  
  
def measure\_time(sort\_func, arr):  
 start = time.time()  
 sort\_func(arr[:]) # copy so original isn’t affected  
 end = time.time()  
 return end - start  
  
data = [random.randint(1, 1000) for \_ in range(100)]  
print("Selection sort time:", measure\_time(selection\_sort, data))

**Extension 2 — Hybrid Sort**

def hybrid\_sort(arr):  
 if len(arr) < 10:  
 # TODO use insertion\_sort  
 pass  
 else:  
 # TODO use quicksort  
 pass  
 return arr

**Extension 3 — Visualization (Bubble Sort Idea)**

import matplotlib.pyplot as plt  
import time  
  
def bubble\_sort\_visual(arr):  
 n = len(arr)  
 for \_ in range(n - 1):  
 for i in range(n - 1):  
 if arr[i] > arr[i+1]:  
 arr[i], arr[i+1] = arr[i+1], arr[i]  
 # draw array  
 plt.bar(range(len(arr)), arr)  
 plt.pause(0.1)  
 plt.clf()  
 plt.show()  
  
bubble\_sort\_visual([5, 3, 8, 2, 1])

👉 This way, students get **guided templates** to fill in, rather than having to start from a blank screen — ideal for learning progression.

Jim, do you want me to also design a **student worksheet version** (without code, just pseudocode prompts and guiding questions), so you could differentiate between programming vs conceptual learners?