AI Prompt: Describing different sorting algorithms - insertion, selection, bubble, merge, quick sorts. For each algorithm, describe the algorithm and show the code to generate a visualization of sorting step by step, using balls of different sizes. You can also talk about how long it takes for each algorithm to run, their time complexity, advantages and disadvantages in the language of a 15 year old. You have initial setup code near the introduction. This should be in the style of a python notebook, so I can convert it into python notebook.
AI Prompt: Convert all of this into one big md file in the style of python notebook - giving the introduction followed by setup code, then describe each algorithm, followed by it's code for visualization.

Sorting Algorithms: The Race of the Balls

Welcome to the sorting arena! This notebook explores five classic sorting algorithms. Think of sorting algorithms as different ways to line up a messy pile of balls, where the height of the bar (or "ball") represents its size.

We'll explore how they work, how fast they are (time complexity), and when to use them.

1. Initial Setup and Visualization Utility

This code block sets up all necessary libraries (`matplotlib` for visualization) and defines the core functions needed to run and animate each sorting demo.

You must run this cell first!

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
import copy
from IPython.display import HTML # Needed to display the animation in a notebook

# --- Setup Code for Visualization ---

def visualize_sort(data, title, generator):
    """
    Sets up and runs the visualization for a sorting algorithm using Matplotlib.
    The 'data' array represents the ball sizes (bar heights).
    The 'generator' is the sorting algorithm logic, which MUST be a generator function
    that yields the array state after each significant step (swap or comparison).
    """
    # Create a fresh copy of the data for the visualization function
    A = copy.deepcopy(data)

    fig, ax = plt.subplots(figsize=(10, 6))
```

```
    ax.set_title(f'Sorting Visualization: {title}', fontsize=16, color='#1A202C')
    ax.set_xticks([])
    ax.set_facecolor('#f7fafc')

    # Initialize the bars (representing ball sizes)
    bar_rects = ax.bar(range(len(A)), A, align='edge', color='#4C51BF')

    # Set axis limits
    max_val = max(A)
    ax.set_ylim(0, max_val * 1.1)

    # Function to update the bar heights for each frame
    def update_fig(state):
        """Updates the bar heights and colors based on the state yielded by the generat
        A, color_indices = state # state is the array and a list of indices to highligh

        # Reset colors (default blue)
        for rect in bar_rects:
            rect.set_color('#4C51BF')

        # Update the height and color of the bars
        for i, (rect, val) in enumerate(zip(bar_rects, A)):
            rect.set_height(val)
            # Highlight elements (e.g., red for comparison/swap)
            if i in color_indices:
                rect.set_color('#E53E3E') # Red highlight

        return bar_rects

    # Create the animation
    anim = animation.FuncAnimation(
        fig,
        func=update_fig,
        frames=generator(A),  # The generator provides the array state for each frame
        interval=100, # Speed of animation in ms
        repeat=False,
        blit=False
    )

    # Close the plot to prevent it from displaying prematurely
    plt.close(fig)

    # Return the HTML representation for display in the notebook
    return HTML(anim.to_jshtml())


# Helper function to generate initial random data
def generate_data(n=40):
    """Generates a list of unique random integers."""
    return random.sample(range(1, 150), n)

# Global data array for consistency
GLOBAL_DATA = generate_data(40)

print("Setup Complete. Data generation and visualization utilities are ready!")
print(f"Sample Size: {len(GLOBAL_DATA)} balls.")
```

2. Insertion Sort

The Algorithm: How it Works

Imagine you are sorting a deck of playing cards in your hand. You pick up one card at a time and **insert** it into the correct spot within the cards you've already sorted. Insertion Sort works the same way: it builds the final sorted array one item at a time by repeatedly comparing the current element with the elements to its left and shifting them over until it finds its correct position.

Time Complexity (How long it takes to run)

Case	Time Complexity	Description (15-year-old language)
Best	$\mathcal{O}(n)$	**Super fast!** If the balls are ALMOST sorted already, it just takes a single quick check for each ball.
Average	$\mathcal{O}(n^2)$	**Slow-ish.** You have to move and check a lot of balls, but not the worst.
Worst	$\mathcal{O}(n^2)$	**Slow.** If the balls are sorted perfectly backward, you check *every* ball against *every* other ball.

Pros & Cons

Advantages	Disadvantages
Simple Code: Easy to write and understand.	**Bad for large lists:** The $\mathcal{O}(n^2)$ means it gets *really* slow if you have thousands of balls.
Fast for Small Lists/Nearly Sorted: It's the best choice if your data is already mostly in order.	**Inefficient for random data:** Lots of shifting (moving elements one spot at a time) wastes time.

Visualization Code

```
# Generator function for Insertion Sort
def insertion_sort(A):
    n = len(A)
    for j in range(1, n):
        key = A[j]
        i = j - 1

        # Highlight the key element being considered (index j)
        yield A, [j]

        # Compare key with elements to its left (i)
        while i >= 0 and A[i] > key:
            # Shift elements to the right
            A[i + 1] = A[i]
```

```
            # Highlight the swap/shift movement (indices i and i+1)
            yield A, [i, i + 1]
            i -= 1

        A[i + 1] = key
        # Highlight the final insertion point (index i+1)
        yield A, [i + 1]

# Run the visualization
print("--- Insertion Sort Demo ---")
# UNCOMMENT the line below to run the visualization:
# visualize_sort(GLOBAL_DATA, "Insertion Sort", insertion_sort)
```

3. Selection Sort

The Algorithm: How it Works

Selection Sort is like picking the gold medal winner out of a crowd. It repeatedly searches the *unsorted* portion of the list for the **smallest** ball. Once the smallest ball is found, it swaps it with the first element of that unsorted section, moving it into its final, correct position.

Time Complexity (How long it takes to run)

Case	Time Complexity	Description (15-year-old language)
Best	$\mathcal{O}(n^2)$	**Always slow.** Even if the balls are already sorted, it still has to check every single ball to confirm it is the smallest.
Average	$\mathcal{O}(n^2)$	**Slow.** It doesn't matter how the balls start, the number of checks is always huge.
Worst	$\mathcal{O}(n^2)$	**Still slow.** The number of checks stays the same.

Pros & Cons

Advantages	Disadvantages
Minimizes Swaps: It performs the absolute minimum number of swaps (at most n swaps). Good if swaps are very expensive (time-consuming).	**Always Slow:** The $\mathcal{O}(n^2)$ is unavoidable. It spends too much time searching.
Easy to Understand: The process is very straightforward and easy to code.	**Inefficient for large lists:** Terrible performance compared to faster algorithms.

Visualization Code

```
# Generator function for Selection Sort
def selection_sort(A):
    n = len(A)
    for i in range(n - 1):
        min_idx = i

        # Highlight the starting position (i)
        yield A, [i]

        # Find the smallest element in the remaining unsorted array
        for j in range(i + 1, n):
            # Highlight the element currently being compared (j)
            yield A, [i, j, min_idx]

            if A[j] < A[min_idx]:
                min_idx = j

        # Swap the smallest element found with the element at position i
        A[i], A[min_idx] = A[min_idx], A[i]

        # Highlight the two elements that were just swapped (i and min_idx)
        yield A, [i, min_idx]

# Run the visualization
print("--- Selection Sort Demo ---")
# UNCOMMENT the line below to run the visualization:
# visualize_sort(GLOBAL_DATA, "Selection Sort", selection_sort)
```

4. Bubble Sort

The Algorithm: How it Works

Bubble Sort is the most basic. It repeatedly steps through the list, compares adjacent balls, and swaps them if they are in the wrong order. The larger balls "**bubble up**" to their correct final positions with each full pass through the list.

Time Complexity (How long it takes to run)

Case	Time Complexity	Description (15-year-old language)
Best	$\mathcal{O}(n)$	**Fast!** If the balls are already sorted, it can do one quick check pass and quit.
Average	$\mathcal{O}(n^2)$	**Very Slow.** You do a massive number of swaps and comparisons.
Worst	$\mathcal{O}(n^2)$	**Worst time.** If sorted backward, every ball has to travel all the way to the end, causing a maximum number of swaps.

Pros & Cons

Advantages	Disadvantages
Trivial to Implement: So simple, even a baby coder can write it.	**Terrible Performance:** It's almost always slower than Insertion and Selection Sorts.
Can detect if a list is sorted quickly: Good for a quick check, thanks to the $\mathcal{O}(n)$ best-case.	**Excessive Swapping:** Creates huge overhead because of too many unnecessary swaps.

Visualization Code

```
# Generator function for Bubble Sort
def bubble_sort(A):
    n = len(A)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):

            # Highlight the two elements being compared (j and j+1)
            yield A, [j, j + 1]

            if A[j] > A[j + 1]:
                # Swap if the element found is greater than the next element
                A[j], A[j + 1] = A[j + 1], A[j]
                swapped = True

                # Highlight the swapped elements
                yield A, [j, j + 1]

        # Optimization: If no two elements were swapped by inner loop, then break
        if not swapped:
            break

# Run the visualization
print("--- Bubble Sort Demo ---")
# UNCOMMENT the line below to run the visualization:
# visualize_sort(GLOBAL_DATA, "Bubble Sort", bubble_sort)
```

5. Merge Sort

The Algorithm: How it Works

This is the first of the fast algorithms, using the **"Divide and Conquer"** strategy. It's like a production line:

1. **Divide:** Keep cutting the list of balls in half until you have lists with only one ball (which are sorted).
2. **Conquer (Merge):** Start combining these small, sorted lists back together into larger sorted lists until the whole array is sorted.

Time Complexity (How long it takes to run)

Case	Time Complexity	Description (15-year-old language)
Best	$\mathcal{O}(n \log n)$	**Super Fast!** Cutting the problem in half is very efficient.
Average	$\mathcal{O}(n \log n)$	**Super Fast!** The performance is consistently amazing no matter the starting order.
Worst	$\mathcal{O}(n \log n)$	**Still Super Fast!** This is one of the best parts about Merge Sort—it never has a slow day.

Pros & Cons

Advantages	Disadvantages
Incredibly Consistent & Fast: $\mathcal{O}(n \log n)$ holds true for all cases.	**Requires Extra Memory:** It needs a temporary list (the "factory floor") to do the merging, which uses up extra RAM.
Stable: It maintains the relative order of equal elements (important for complex data).	**More Complex to Implement:** The recursive (self-calling) structure is harder to code than $\mathcal{O}(n^2)$ sorts.

Visualization Code

```
# Generator function for Merge Sort
def merge_sort(A, start=0, end=None):
    if end is None:
        end = len(A)

    if end - start > 1:
        mid = (start + end) // 2

        # Recursively call merge_sort on the two halves
        yield from merge_sort(A, start, mid)
        yield from merge_sort(A, mid, end)

        # Merge the two sorted halves
        p1 = start
        p2 = mid
        temp = []

        # Highlight the section currently being merged
        yield A, list(range(start, end))

        while p1 < mid and p2 < end:
            if A[p1] <= A[p2]:
                temp.append(A[p1])
                p1 += 1
            else:
                temp.append(A[p2])
```

```
                p2 += 1

        temp.extend(A[p1:mid])
        temp.extend(A[p2:end])

        # Copy sorted temporary array back to original array (A)
        for k in range(len(temp)):
            A[start + k] = temp[k]
            # Highlight the elements being placed back in the final array
            yield A, [start + k]

# Wrapper function to call the generator
def merge_sort_wrapper(A):
    yield from merge_sort(A)

# Run the visualization
print("--- Merge Sort Demo ---")
# UNCOMMENT the line below to run the visualization:
# visualize_sort(GLOBAL_DATA, "Merge Sort", merge_sort_wrapper)
```

6. Quick Sort

The Algorithm: How it Works

Quick Sort is another famous fast algorithm that uses **"Divide and Conquer."**

1. **Choose a Pivot:** Pick one ball (usually the last one) as the central **pivot**.
2. **Partition:** Rearrange the list so that all balls smaller than the pivot are on its left, and all balls larger are on its right. The pivot is now in its final, correct spot.
3. **Recurse:** Apply the same two steps to the sub-list of small balls and the sub-list of large balls.

Time Complexity (How long it takes to run)

Case	Time Complexity	Description (15-year-old language)
Best	$\mathcal{O}(n \log n)$	**The Fastest!** If you pick a perfect pivot every time (one that splits the list exactly in half).
Average	$\mathcal{O}(n \log n)$	**Still Super Fast!** The performance is typically amazing in the real world.
Worst	$\mathcal{O}(n^2)$	**Super Slow!** If you are unlucky and always pick the smallest or largest ball as the pivot. This rarely happens in practice with good pivot selection.

Pros & Cons

Advantages	Disadvantages

Fastest on Average: In almost all real-world scenarios, Quick Sort is slightly faster than Merge Sort due to fewer data movements.

Worst-Case $\mathcal{O}(n^2)$: If the pivot is always badly chosen, it can become as slow as Bubble Sort.

In-Place Sorting: It sorts the array without needing large amounts of extra memory (unlike Merge Sort).

Not Stable: It doesn't guarantee the relative order of equal elements.

Visualization Code

```
# Helper function for Partitioning in Quick Sort (part of the generator)
def partition(A, low, high):
    pivot = A[high]
    i = low - 1  # Index of smaller element

    # Highlight the pivot (high) and the partition range
    yield A, [high] + list(range(low, high))

    for j in range(low, high):
        # Highlight current comparison (j) and the potential swap position (i)
        yield A, [high, j, i + 1]

        if A[j] <= pivot:
            i = i + 1
            A[i], A[j] = A[j], A[i]
            # Highlight the elements after swap
            yield A, [i, j]

    A[i + 1], A[high] = A[high], A[i + 1]
    # Highlight the pivot's final position
    yield A, [i + 1]
    return i + 1

# Generator function for Quick Sort
def quick_sort_recursive(A, low, high):
    if low < high:
        # pi is partitioning index, A[pi] is now at right place
        pi = yield from partition(A, low, high)

        # Separately sort elements before partition and after partition
        yield from quick_sort_recursive(A, low, pi - 1)
        yield from quick_sort_recursive(A, pi + 1, high)

# Wrapper function to call the generator
def quick_sort_wrapper(A):
    yield from quick_sort_recursive(A, 0, len(A) - 1)

# Run the visualization
print("--- Quick Sort Demo ---")
# UNCOMMENT the line below to run the visualization:
# visualize_sort(GLOBAL_DATA, "Quick Sort", quick_sort_wrapper)
```