Sorting Algorithms Project

main comparison implementation:

```
import random
import time
import sys
from quick_sort import shuffled_quick_sort
from merge_sort import merge_sort
from insertion_sort import insertion_sort
from test_cases import test_arrays
from quick_sort_no_shuffle import quick_sort as quick_sort_no_shuffle

sys.setrecursionlimit(2000)

def main():
    print("Comparing sorting algorithms on predefined test cases:\n")
    for case_name, arr in test_arrays.items():
        print(f"Test case: {case_name} (size: {len(arr)})")

        # Quick Sort (shuffled)
        arr_copy = arr.copy()
        start_time = time.time()
        shuffled_quick_sort(arr_copy)
        quick_time = time.time() - start_time

        # Merge Sort
        arr_copy = arr.copy()
        start_time = time.time()
        merge_sort(arr_copy)
        merge_time = time.time() - start_time
```

```
        # Insertion Sort
        arr_copy = arr.copy()
        start_time = time.time()
        insertion_sort(arr_copy)
        insertion_time = time.time() - start_time

        # Quick Sort (no shuffle) only for sorted and reverse_sorted
        if case_name in ["sorted", "reverse_sorted"]:
            arr_copy = arr.copy()
            start_time = time.time()
            quick_sort_no_shuffle(arr_copy)
            no_shuffle_time = time.time() - start_time
            print(f"  Quick Sort (no shuffle) time: {no_shuffle_time:.6f} seconds")

        # Print results
        print(f"  Quick Sort (shuffled) time: {quick_time:.6f} seconds")
        print(f"  Merge Sort time: {merge_time:.6f} seconds")
        print(f"  Insertion Sort time: {insertion_time:.6f} seconds\n")

if __name__ == "__main__":
    main()
```

Quick Sort :

with shuffle:

```
import random

def quick_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr
```

```
    else:
        # Choose the first element as pivot
        pivot = arr[0]
        #Subarray of elements less than or equal to pivot
        less = [x for x in arr[1:] if x <= pivot]
        #Subarray of elements greater than pivot
        greater = [x for x in arr[1:] if x > pivot]
        # Recursively sort and concatenate
        return quick_sort(less) + [pivot] + quick_sort(greater)

def shuffled_quick_sort(arr):
    # Create a copy to avoid modifying the original list
    arr_copy = arr[:]
    random.shuffle(arr_copy)
    return quick_sort(arr_copy)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = shuffled_quick_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

with no shuffle:

```
def quick_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr
    else:
        # Choose the first element as pivot
        pivot = arr[0]
        # Subarray of elements less than or equal to pivot
        less = [x for x in arr[1:] if x <= pivot]
        # Subarray of elements greater than pivot
```

```
        greater = [x for x in arr[1:] if x > pivot]
        # Recursively sort and concatenate
        return quick_sort(less) + [pivot] + quick_sort(greater)

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

Merge Sort :

```
def merge_sort(arr):
    # Base case: if array has 1 or 0 elements, it's already sorted
    if len(arr) <= 1:
        return arr

    # Find middle point to divide array into two halves
    mid = len(arr) // 2  # Fixed: divide by 2 to get the middle point

    # Recursively sort first and second halves
    left = merge_sort(arr[:mid])   # Sort left half
    right = merge_sort(arr[mid:])  # Sort right half

    # Merge the sorted halves
    return merge(left, right)

# Helper function to merge two sorted arrays
def merge(left, right):
    result = []  # Initialize empty result array
    i = j = 0    # Initialize pointers for both arrays

    # Compare elements from both arrays and merge them in sorted order
    while i < len(left) and j < len(right):
```

```
        if left[i] <= right[j]:
            result.append(left[i])  # Add element from left array
            i += 1                  # Move left array pointer
        else:
            result.append(right[j]) # Add element from right array
            j += 1                  # Move right array pointer

    # Add remaining elements from left array, if any
    result.extend(left[i:])
    # Add remaining elements from right array, if any
    result.extend(right[j:])

    return result  # Return merged sorted array

# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = merge_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

insertion sort :

```
def insertion_sort(arr):
    # Traverse from the second element to the end
    for i in range(1, len(arr)):
        key = arr[i]  # Element to be placed at correct position
        j = i - 1
        # Shift elements that are greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # Place the key in its correct location
    return arr
```

```
# Example usage:
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = insertion_sort(arr)
print(sorted_arr)  # Output: [1, 1, 2, 3, 6, 8, 10]
```

test cases:

```
import random

test_arrays = {
    "sorted": [i for i in range(1, 1001)],  # Sorted 1 to 1000
    "reverse_sorted": [i for i in range(1000, 0, -1)],  # Reverse sorted 1000 to 1
    "mixed": [random.randint(1, 1000) for _ in range(1000)],  # 1000 mixed values
    "duplicates": [random.choice([4, 7, 1, 9, 3, 2]) for _ in range(1000)],  # 1000 iter
    "empty": [],  # Empty list
    "single_element": [42],  # One element
    "large_random": [random.randint(1, 1000) for _ in range(1000)],  # 1000 randon
    "nearly_sorted": [i if i % 10 != 0 else i - 3 for i in range(1, 1001)],  # Small pertur
}
```

visualization code:

```
import matplotlib.pyplot as plt
import numpy as np
import time
from quick_sort import shuffled_quick_sort
from merge_sort import merge_sort
from insertion_sort import insertion_sort
from test_cases import test_arrays
from quick_sort_no_shuffle import quick_sort as quick_sort_no_shuffle
```

```
def measure_sorting_time(sort_func, arr):
    arr_copy = arr.copy()
    start_time = time.time()
    sort_func(arr_copy)
    return time.time() - start_time

def create_performance_visualization():
    # Prepare data
    algorithms = ['Quick Sort', 'Merge Sort', 'Insertion Sort']
    test_cases = list(test_arrays.keys())

    # Initialize results matrix
    results = np.zeros((len(algorithms), len(test_cases)))

    # Measure performance for each algorithm and test case
    for i, (case_name, arr) in enumerate(test_arrays.items()):
        results[0, i] = measure_sorting_time(shuffled_quick_sort, arr)
        results[1, i] = measure_sorting_time(merge_sort, arr)
        results[2, i] = measure_sorting_time(insertion_sort, arr)

    # Create bar chart
    plt.figure(figsize=(15, 8))
    x = np.arange(len(test_cases))
    width = 0.25

    for i, algorithm in enumerate(algorithms):
        plt.bar(x + i*width, results[i], width, label=algorithm)

    plt.xlabel('Test Cases')
    plt.ylabel('Time (seconds)')
    plt.title('Sorting Algorithms Performance Comparison')
    plt.xticks(x + width, test_cases, rotation=45)
    plt.legend()
    plt.tight_layout()

    # Save the plot
```

```
    plt.savefig('sorting_performance.png')
    plt.close()

def create_time_comparison_plot():
    # Test with different array sizes
    sizes = [100, 500, 1000, 2000, 5000]
    algorithms = {
        'Quick Sort': shuffled_quick_sort,
        'Merge Sort': merge_sort,
        'Insertion Sort': insertion_sort
    }

    results = {name: [] for name in algorithms.keys()}

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

        for name, sort_func in algorithms.items():
            time_taken = measure_sorting_time(sort_func, arr)
            results[name].append(time_taken)

    # Create line plot
    plt.figure(figsize=(10, 6))
    for name, times in results.items():
        plt.plot(sizes, times, marker='o', label=name)

    plt.xlabel('Array Size')
    plt.ylabel('Time (seconds)')
    plt.title('Sorting Algorithms Performance vs Array Size')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()

    # Save the plot
    plt.savefig('sorting_time_comparison.png')
```

```
    plt.close()

if __name__ == "__main__":
    create_performance_visualization()
    create_time_comparison_plot()
    print("Visualizations have been created and saved as 'sorting_performance.pn
```