Binary Search vs Linear Search

```
import matplotlib.pyplot as plt
import random
import time

def linear_search(arr: list, key: int) -> int:
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1

def binary_search(arr: list, key: int) -> int:
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = int((low + high)/2)
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            high = mid - 1
        else:
            low = mid + 1
    return -1

def mean_time(arr: list, key: int, search_func, index: int):
    total_time = 0.0
    for i in range(index):
        start_time = time.perf_counter()
        search_func(arr, key)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/index

def populate_array(n: int):
    arr = []
    for i in range(n):
        arr.append(random.randint(0,n))
    return arr

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    arr = []
```

```python
    binary_search_time = []
    linear_search_time = []

    n_vals = list(range(50,1000,50))
    for n in n_vals:
        index = n * 10
        arr = populate_array(n)
        key = n + 20
        linear_search_time.append(mean_time(arr, key, linear_search, index))
        binary_search_time.append(mean_time(arr, key, binary_search, index))

    plot_comparison(n_vals, ('Binary Search', binary_search_time),
    ('Linear Search', linear_search_time), 'bsearch_vs_lsearch')

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

Recursive Factorial vs Iterative Factorial

```
import matplotlib.pyplot as plt
import time

def recursive_factorial(n: int) -> int:
    if(n < 0):
        return 1
    elif(n <= 1):
        return n
    else:
        return (n * recursive_factorial(n-1))

def iterative_factorial(n: int) -> int:
    fact = 1
    for i in range(1,n+1):
        fact *= i
    return fact

def mean_time(fact_function, iterations: int, input: int):
    mean_time = 0.0
    for i in range(iterations):
        start_time = time.perf_counter()
        fact_function(input)
        end_time = time.perf_counter()
        mean_time += (end_time - start_time)
    mean_time /= iterations
    return mean_time

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    recursive_factorial_times = []
    iterative_factorial_times = []
    n_vals = list(range(50,1000,50))

    for n in n_vals:
        recursive_factorial_times.append(mean_time(recursive_factorial,1000,n))
        iterative_factorial_times.append(mean_time(iterative_factorial,1000,n))

    plot_comparison(n_vals, ('Recursive Factorial',recursive_factorial_times),
    ('Iterative Factorial',iterative_factorial_times), 'factorial_recursive_vs_iterative')

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

Iterative Fibonacci vs Recursive Fibonacci

```
import matplotlib.pyplot as plt
import time

def fibonacci_iterative(n: int) -> int:
    n0 = 0
    n1 = 1
    if (n == 0) or (n == 1):
        return n
    elif n > 1:
        while (n - 1) > 0:
            num = n1 + n0
            n0 = n1
            n1 = num
            n -= 1
        return num

def fibonacci_recursive(n: int) -> int:
    if (n == 0) or (n == 1):
        return n
    elif n > 1:
        return (fibonacci_recursive(n-1) + fibonacci_recursive(n-2))

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def mean_time(fibonacci_function, iterations: int, input: int):
    mean_time = 0.0
    for i in range(iterations):
        start_time = time.perf_counter()
        fibonacci_function(input)
        end_time = time.perf_counter()
        mean_time += (end_time - start_time)
    mean_time /= iterations
    return mean_time

def main():
    recursive_fibonacci_times = []
    iterative_fibonacci_times = []
    n_vals = list(range(2,40,2))

    for n in n_vals:
        recursive_fibonacci_times.append(mean_time(fibonacci_recursive,5,n))
        iterative_fibonacci_times.append(mean_time(fibonacci_iterative,5,n))
```

```
    plot_comparison(n_vals, ('Recursive Fibonacci',recursive_fibonacci_times), ('Iterative Fibonacci',iterative_

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

Matrix Multiplication Analysis

```
import time
import matplotlib.pyplot as plt
import random

def create_matrix(order) -> list:
    mat = []
    for i in range(order):
        row = []
        for j in range(order):
            row.append(0)
        mat.append(row)
    return mat

def populate_matrix(mat, lim) -> list:
    for i in range(len(mat)):
        row = mat[i]
        for j in range(len(row)):
            row[j] = random.randint(0,lim)
    return mat

def matrix_multiply(mat1, mat2) -> list:
    res = create_matrix(len(mat1))
    for i in range(len(mat1)):
        for j in range(len(mat1[i])):
            for k in range(len(mat1)):
                res[i][j] += mat1[i][k] * mat2[k][j]
    return res

def mean_time(matmul_function, iterations: int, order: int) -> float:
    mean_time = 0.0
    for i in range(iterations):
        mat1 = populate_matrix(create_matrix(order), order * 10)
        mat2 = populate_matrix(create_matrix(order), order * 10)
        start = time.perf_counter()
        matmul_function(mat1,mat2)
        end = time.perf_counter()
        mean_time += (end - start)
    return mean_time/iterations

def plot_algo(x_axis: list, data_algo: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])


    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    matmul_times = []
    n_vals = list(range(50,500,50))
```

```python
    for n in n_vals:
        matmul_times.append(mean_time(matrix_multiply, 5, n))

    plot_algo(n_vals, ('Matrix Multiplication', matmul_times), 'matrixmul_analysis')


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

Unique checking Analysis

```
import time
import matplotlib.pyplot as plt
import random

def unique_check(a: list) -> bool:
    length = len(a)
    for i in range(length-1):
        for j in range(i+1, length):
            if a[j] == a[i]:
                return False
    return True

def unique_populate(n: int, lim: int) -> list:
    arr = []
    for i in range(n):
        randgen = random.randint(0,lim)
        while randgen in arr:
            randgen = random.randint(0,lim)
        arr.append(randgen)
    return arr

def mean_time(unique_check_func, n: int, iterations: int, a: list) -> float:
    mean_time = 0.0
    for i in range(iterations):
        start = time.perf_counter()
        unique_check_func(a)
        end = time.perf_counter()
        mean_time += (end - start)
    return mean_time/iterations

def plot_algo(x_axis: list, data_algo: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    ucheck_times = []
    n_vals = list(range(50,1000,50))
    for n in n_vals:
        ucheck_times.append(mean_time(unique_check, n, 1000, unique_populate(n, n*10)))

    plot_algo(n_vals, ('Array Unique Checking', ucheck_times), 'uniquecheck_analysis')

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

Merge Sort Analysis

```
import time
import matplotlib.pyplot as plt

def merge_sort(a: list, low: int, high: int):
    if low < high:
        mid = (low + high) // 2
        merge_sort(a, low, mid)
        merge_sort(a, mid + 1, high)
        merge(a, low, mid, high)

def merge(a: list, low: int, mid: int, high: int):
    temp = []
    i = low
    j = mid + 1
    k = 0

    while i <= mid and j <= high:
        if a[i] <= a[j]:
            temp.append(a[i])
            i += 1
        else:
            temp.append(a[j])
            j += 1
        k += 1

    while i <= mid:
        temp.append(a[i])
        i += 1
        k += 1

    while j <= high:
        temp.append(a[j])
        j += 1
        k += 1

    for idx in range(len(temp)):
        a[low + idx] = temp[idx]

def mean_time(arr: list, sort_func, index: int):
    total_time = 0.0
    for i in range(index):
        start_time = time.perf_counter()
        sort_func(arr, 0, len(arr) - 1)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/index

def populate_array_descending(n: int):
    arr = list(range(n,0,-1))
    return arr

def plot_algo(x_axis: list, data_algo: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
```

```
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    mergesort_times = []
    n_vals = list(range(50,1000,50))
    for n in n_vals:
        iterations = 100
        arr = populate_array_descending(n)
        mergesort_times.append(mean_time(arr, merge_sort, iterations))

    plot_algo(n_vals, ('Merge Sort', mergesort_times), 'mergesort_analysis')

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

Quick Sort Analysis

```
import time
import matplotlib.pyplot as plt

def partition(a: list, low: int, high: int) -> int:
    i = low - 1
    j = high + 1
    pivot = a[low]

    while(True):
        while(True):
            i += 1
            if not (a[i] < pivot):
                break

        while(True):
            j -= 1
            if not (a[j] > pivot):
                break

        if i >= j:
            return j

        a[i], a[j] = a[j], a[i]

def quick_sort(a: list, low: int, high: int):
    if low < high:
        pi = partition(a, low, high)
        quick_sort(a, low, pi)
        quick_sort(a, pi + 1, high)

def mean_time(arr: list, sort_func, index: int):
    total_time = 0.0
    for i in range(index):
        start_time = time.perf_counter()
        sort_func(arr, 0, len(arr) - 1)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/index

def populate_array_descending(n: int):
    arr = list(range(n,0,-1))
    return arr

def plot_algo(x_axis: list, data_algo: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()
```

```
def main():
    quicksort_times = []
    n_vals = list(range(50,1000,50))
    for n in n_vals:
        iterations = 100
        arr = populate_array_descending(n)
        quicksort_times.append(mean_time(arr, quick_sort, iterations))

    plot_algo(n_vals, ('Quick Sort', quicksort_times), 'quicksort_analysis')

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

Selection Sort vs Bubble Sort

```
import matplotlib.pyplot as plt
import random
import time

def bubble_sort(arr):
    n = len(arr)
    for i in range(0,n):
        for j in range(0,n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

def selection_sort(arr):
    n = len(arr)
    for i in range(0,n):
        min = i
        for j in range(i+1,n):
            if arr[min] > arr[j]:
                min = j
        if min != i:
            arr[i], arr[min] = arr[min], arr[i]

def mean_time(arr: list, sort_func, index: int):
    total_time = 0.0
    for i in range(index):
        start_time = time.perf_counter()
        sort_func(arr)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/index

def populate_array_descending(n: int):
    arr = list(range(n,0,-1))
    return arr

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()


def main():
    arr = []
    bubble_sort_time = []
    selection_sort_time = []
```

```
    n_vals = list(range(50,1000,50))
    for n in n_vals:
        index = 25
        arr = populate_array_descending(n)
        bubble_sort_time.append(mean_time(arr, bubble_sort, index))
        selection_sort_time.append(mean_time(arr, selection_sort, index))

    plot_comparison(n_vals, ('Bubble Sort', bubble_sort_time), ('Selection Sort',
    selection_sort_time), 'bsort_vs_ssort')


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

Binary Digit counting Recursive vs Iterative

```
import matplotlib.pyplot as plt
import time

def bindigitcounting_recursive(n: int) -> int:
    if n == 0:
        return n
    else:
        return (1 + n//2)

def bindigitcounting_iterative(n: int) -> int:
    count = 1
    while(n > 1):
        n //= 2
        count += 1
    return count

def mean_time(bindigitcounting_function, iterations: int, input: int) -> int:
    mean_time = 0.0
    for i in range(iterations):
        start = time.perf_counter()
        bindigitcounting_function(input)
        end = time.perf_counter()
        mean_time += (end - start)
    return (mean_time/iterations)

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    bindigitcounting_iterative_times = []
    bindigitcounting_recursive_times = []

    n_vals = list(range(50,1000,50))
    for n in n_vals:
        bindigitcounting_recursive_times.append(mean_time(bindigitcounting_recursive, 10000, n))
        bindigitcounting_iterative_times.append(mean_time(bindigitcounting_iterative, 10000, n))

    plot_comparison(n_vals, ('Recursive Binary Digit Counting', bindigitcounting_recursive_times),
    ('Iterative Binary Digit Counting', bindigitcounting_iterative_times), 'binarydigitcounting_recursive_vs_ite

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

Naive vs Horspool String matching

```
import string
import random
import time
import matplotlib.pyplot as plt

def naive_string_match(string: str, pattern: str, table=None):
    m = len(pattern)
    n = len(string)
    for i in range(n - m + 1):
        j = 0
        while j < m and string[i + j] == pattern[j]:
            j += 1
        if j == m:
            return i
    return -1


def shift_table(pattern: str):
    m = len(pattern)
    table = []
    for i in range(128):
        table.append(m)
    for j in range(m - 1):
        table[ord(pattern[j])] = m - 1 - j
    return table

def horspool_string_match(string: str, pattern: str, table: list):
    m = len(pattern)
    n = len(string)
    i = m - 1
    for i in range(n):
        j = 0
        while j < m and string[i - j] == pattern[m - 1 - j]:
            j += 1
        if j == m:
            return i - m + 1
        else:
            i += table[ord(string[i])]
    return -1

def generate_random_string(length, exclude_letters: str):
    alphabets = list(string.ascii_letters)

    for letter in exclude_letters:
        if letter in alphabets:
            alphabets.remove(letter)

    random_string = ''.join(random.choices(alphabets, k=length))
    return random_string

def mean_time(match_func, iterations: int, length: int):
    total_time = 0.0
    pattern = 'AEIOUaeiou'
    table = shift_table(pattern)
    for i in range(iterations):
        string = generate_random_string(length, pattern)
        start_time = time.perf_counter()
        match_func(string, pattern, table)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
```

```
        total_time += time_elapsed
    return total_time/iterations

def plot_comparison(x_axis: list, data_algo2: tuple[str, list], data_algo1: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    naive_match_time = []
    horspool_match_time = []

    n_vals = list(range(50,1000,50))
    for n in n_vals:
        iterations = n * 10
        naive_match_time.append(mean_time(naive_string_match, iterations, n))
        horspool_match_time.append(mean_time(horspool_string_match, iterations, n))

    plot_comparison(n_vals, ('Naive String Match', naive_match_time),
    ('Horspool String Match', horspool_match_time), 'naive_vs_horspool')

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

Insertion Sort Analysis

```
import matplotlib.pyplot as plt
import time

def insertion_sort(arr: list[int]):
    for i in range(1, len(arr)):
        key = arr[i]

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

        arr[j + 1] = key
    return arr

def mean_time(arr: list, sort_func, index: int):
    total_time = 0.0
    for i in range(index):
        start_time = time.perf_counter()
        sort_func(arr)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/index

def populate_array_descending(n: int):
    arr = list(range(n,0,-1))
    return arr

def plot_algo(x_axis: list, data_algo: tuple[str, list], filename: str):
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    insertionsort_times = []
    n_vals = list(range(50,1000,50))
    for n in n_vals:
        iterations = 10000
        arr = populate_array_descending(n)
        insertionsort_times.append(mean_time(arr, insertion_sort, iterations))

    plot_algo(n_vals, ('Insertion Sort', insertionsort_times), 'insertionsort_analysis')

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

Warshall's Algorithm

```
import random
import time
import matplotlib.pyplot as plt

def warshall(graph):
    n = len(graph)
    reach = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            reach[i][j] = graph[i][j]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])
    return reach

def generate_random_adj_matrix(num_nodes, edge_prob):
    matrix = [[0] * num_nodes for _ in range(num_nodes)]
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j and random.random() < edge_prob:
                matrix[i][j] = 1
    return matrix

def mean_time(num_nodes, edge_prob, algo_func, iterations):
    total_time = 0.0
    for _ in range(iterations):
        matrix = generate_random_adj_matrix(num_nodes, edge_prob)
        start_time = time.perf_counter()
        algo_func(matrix)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def plot_algo(x_axis, data_algo, filename):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()

def main():
    warshall_times = []
    n_vals = list(range(10, 200, 20))
    edge_prob = 0.5
    for n in n_vals:
        iterations = 10
        warshall_times.append(mean_time(n, edge_prob, warshall, iterations))

    plot_algo(n_vals, ('Warshall Algorithm', warshall_times), 'warshall_analysis')

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

Floyd-Warshall Algorithm

```
import random
import time
import matplotlib.pyplot as plt

def floyd_warshall(graph):
    dist = [[float('inf')] * len(graph) for _ in range(len(graph))]
    for u in range(len(graph)):
        dist[u][u] = 0
        for v, weight in graph[u]:
            dist[u][v] = weight

    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
    return dist

def generate_random_graph(num_nodes, edge_prob, max_weight=10):
    graph = [[] for _ in range(num_nodes)]
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j and random.random() < edge_prob:
                weight = random.randint(1, max_weight)
                graph[i].append((j, weight))
    return graph

def mean_time(graph_gen_func, num_nodes, edge_prob, algo_func, iterations):
    total_time = 0.0
    for _ in range(iterations):
        graph = graph_gen_func(num_nodes, edge_prob)
        start_time = time.perf_counter()
        algo_func(graph)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def plot_algo(x_axis, data_algo, filename):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()

def main():
    floyd_warshall_times = []
    n_vals = list(range(2, 200, 20))
    edge_prob = 0.5
    for n in n_vals:
        iterations = 10
        floyd_warshall_times.append(mean_time(generate_random_graph, n, edge_prob, floyd_warshall, iterations))

    plot_algo(n_vals, ('Floyd-Warshall Algorithm', floyd_warshall_times),
    'floyd_warshall_analysis')
```

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

Knapsack 0/1 Algorithm

```
import random
import time
import matplotlib.pyplot as plt

def knapsack(weights, values, W):
    n = len(values)
    K = [[0 for w in range(W + 1)] for i in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif weights[i - 1] <= w:
                K[i][w] = max(values[i - 1] + K[i - 1][w - weights[i - 1]], K[i - 1][w])
            else:
                K[i][w] = K[i - 1][w]
    return K[n][W]

def generate_knapsack_problem(n, max_weight):
    weights = [random.randint(1, max_weight) for _ in range(n)]
    values = [random.randint(1, 100) for _ in range(n)]
    W = random.randint(max_weight, max_weight * n // 2)
    return weights, values, W

def mean_time(num_items, max_weight, algo_func, iterations):
    total_time = 0.0
    for _ in range(iterations):
        weights, values, W = generate_knapsack_problem(num_items, max_weight)
        start_time = time.perf_counter()
        algo_func(weights, values, W)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def plot_algo(x_axis, data_algo, filename):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.plot(x_axis, data_algo[1], marker='o', label=data_algo[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo[0]} Analysis', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()

def main():
    knapsack_times = []
    n_vals = list(range(10, 200, 20))
    max_weight = 50
    for n in n_vals:
        iterations = 10
        knapsack_times.append(mean_time(n, max_weight, knapsack, iterations))

    plot_algo(n_vals, ('Knapsack Algorithm', knapsack_times), 'knapsack_analysis')

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

Dijkstra Algorithm

```
import time
import random
import heapq
import matplotlib.pyplot as plt

def dijkstra_algorithm(graph, start):
    n = len(graph)
    dist = [float('inf')] * n
    dist[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_dist, u = heapq.heappop(priority_queue)

        if current_dist > dist[u]:
            continue

        for v, weight in graph[u]:
            distance = current_dist + weight

            if distance < dist[v]:
                dist[v] = distance
                heapq.heappush(priority_queue, (distance, v))

    return dist

def mean_time(num_nodes, edge_prob, iterations):
    total_time = 0.0
    for _ in range(iterations):
        graph = generate_random_graph(num_nodes, edge_prob)
        start_node = random.choice(range(num_nodes))
        start_time = time.perf_counter()
        dijkstra_algorithm(graph, start_node)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def plot_algo(x_axis, data_algo, filename):
    fig, ax = plt.subplots(figsize=(12,8))
    ax.plot(x_axis, data_algo, marker='o', label='Dijkstra Algorithm')
    ax.set_facecolor('lightgreen')
    ax.set_title('Dijkstra Algorithm Analysis', fontsize=24)
    ax.set_xlabel('Number of Nodes', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()

def generate_random_graph(num_nodes, edge_prob):
    graph = [[] for _ in range(num_nodes)]
    for u in range(num_nodes):
        for v in range(u + 1, num_nodes):
            if random.random() < edge_prob:
                weight = random.randint(1, 10)
                graph[u].append((v, weight))
                graph[v].append((u, weight))
    return graph
```

```
def main():
    dijkstra_times = []
    n_vals = list(range(10, 200, 20))
    edge_prob = 0.5
    iterations = 10
    for n in n_vals:
        dijkstra_times.append(mean_time(n, edge_prob, iterations))

    plot_algo(n_vals, dijkstra_times, 'dijkstra_analysis.png')

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

Kruskal Algorithm

```
import time
import random
import matplotlib.pyplot as plt

def find(parent, u):
    if parent[u] != u:
        parent[u] = find(parent, parent[u])
    return parent[u]

def union(parent, rank, u, v):
    root_u = find(parent, u)
    root_v = find(parent, v)
    if root_u != root_v:
        if rank[root_u] > rank[root_v]:
            parent[root_v] = root_u
        elif rank[root_u] < rank[root_v]:
            parent[root_u] = root_v
        else:
            parent[root_v] = root_u
            rank[root_u] += 1

def kruskal_algorithm(graph, num_nodes):
    edges = []
    for u in range(num_nodes):
        for v, weight in graph[u]:
            edges.append((weight, u, v))
    edges.sort()

    parent = list(range(num_nodes))
    rank = [0] * num_nodes
    mst = []

    for weight, u, v in edges:
        if find(parent, u) != find(parent, v):
            union(parent, rank, u, v)
            mst.append((u, v, weight))

    return mst

def mean_time(num_nodes, edge_prob, iterations):
    total_time = 0.0
    for _ in range(iterations):
        graph = generate_random_graph(num_nodes, edge_prob)
        start_time = time.perf_counter()
        kruskal_algorithm(graph, num_nodes)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def generate_random_graph(num_nodes, edge_prob):
    graph = [[] for _ in range(num_nodes)]
    for u in range(num_nodes):
        for v in range(u + 1, num_nodes):
            if random.random() < edge_prob:
                weight = random.randint(1, 10)
                graph[u].append((v, weight))
                graph[v].append((u, weight))
    return graph
```

```
def plot_algo(x_axis, data_algo, filename):
    fig, ax = plt.subplots(figsize=(12,8))
    ax.plot(x_axis, data_algo, marker='o', label='Kruskal Algorithm')
    ax.set_facecolor('lightgreen')
    ax.set_title('Kruskal Algorithm Analysis', fontsize=24)
    ax.set_xlabel('Number of Nodes', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()

def main():
    kruskal_times = []
    n_vals = list(range(10, 200, 20))
    edge_prob = 0.5
    iterations = 10
    for n in n_vals:
        kruskal_times.append(mean_time(n, edge_prob, iterations))

    plot_algo(n_vals, kruskal_times, 'kruskal_analysis.png')

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

Tree Traversal analysis - Preorder, Postorder and Inorder

```
import matplotlib.pyplot as plt
import time
import random

class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def insert(root: Node, key):
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

def inorder(root):
    result = []
    if root:
        result = inorder(root.left)
        result.append(root.val)
        result += inorder(root.right)
    return result

def preorder(root):
    result = []
    if root:
        result.append(root.val)
        result += preorder(root.left)
        result += preorder(root.right)
    return result

def postorder(root):
    result = []
    if root:
        result = postorder(root.left)
        result += postorder(root.right)
        result.append(root.val)
    return result

def populate_bst(n: int):
    root = Node(random.randint(0,100))
    for _ in range(n):
        insert(root, random.randint(0,100))
    return root

def mean_time(traversal_function, iterations: int):
    total_time = 0.0
    for i in range(iterations):
        root = populate_bst(iterations)
        start_time = time.perf_counter()
        traversal_function(root)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time/iterations
```

```
def plot_comparison(x_axis: list, data_algo1: tuple[str, list], data_algo2: tuple[str, list], data_algo3: tuple[
    fig, ax = plt.subplots(figsize=(12,8))

    ax.plot(x_axis, data_algo1[1], marker='o', color='red', label=data_algo1[0])

    ax.plot(x_axis, data_algo2[1], marker='o', color='blue', label=data_algo2[0])

    ax.plot(x_axis, data_algo3[1], marker='o', color='orange', label=data_algo3[0])

    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]} vs {data_algo3[0]}', fontsize=24)
    ax.set_xlabel('Input Size (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)

    ax.legend()
    plt.savefig(filename)

    plt.show()

def main():
    postorder_time = []
    inorder_time = []
    preorder_time = []

    n_vals = list(range(50,1000,50))
    for n in n_vals:
        index = 100
        postorder_time.append(mean_time(postorder, index))
        preorder_time.append(mean_time(preorder, index))
        inorder_time.append(mean_time(inorder, index))

    plot_comparison(n_vals, ('Inorder', inorder_time), ('Preorder', preorder_time),
    ('Postorder', postorder_time), 'traversal_analysis.png')


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

Breadth-first search vs Depth-first search

```
import random
import time
import matplotlib.pyplot as plt
from collections import deque, defaultdict

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

def dfs_stack(graph, start):
    visited = set()
    stack = [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

def generate_random_graph(num_nodes, edge_prob):
    graph = defaultdict(set)
    for i in range(num_nodes):
        for j in range(num_nodes):
            if i != j and random.random() < edge_prob:
                graph[i].add(j)
                graph[j].add(i)
    return graph

def mean_time(search_func, iterations, num_nodes, edge_prob):
    total_time = 0.0
    for _ in range(iterations):
        graph = generate_random_graph(num_nodes, edge_prob)
        start_node = random.choice(list(graph.keys()))
        start_time = time.perf_counter()
        search_func(graph, start_node)
        end_time = time.perf_counter()
        time_elapsed = end_time - start_time
        total_time += time_elapsed
    return total_time / iterations

def plot_comparison(x_axis, data_algo1, data_algo2, filename):
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.plot(x_axis, data_algo1[1], marker='o', label=data_algo1[0])
    ax.plot(x_axis, data_algo2[1], marker='o', label=data_algo2[0])
    ax.set_facecolor('lightgreen')
    ax.set_title(f'{data_algo1[0]} vs {data_algo2[0]}', fontsize=24)
    ax.set_xlabel('Number of Nodes (n)', fontsize=18)
    ax.set_ylabel('Average Time (seconds)', fontsize=18)
    ax.tick_params(axis='x', labelsize=12)
    ax.tick_params(axis='y', labelsize=12)
    ax.legend()
    plt.savefig(filename)
    plt.show()
```

```
def main():
    bfs_time = []
    dfs_time = []
    n_vals = list(range(20, 200, 20))
    edge_prob = 0.1
    for n in n_vals:
        iterations = n * 10
        bfs_time.append(mean_time(bfs, iterations, n, edge_prob))
        dfs_time.append(mean_time(dfs_stack, iterations, n, edge_prob))
    plot_comparison(n_vals, ('BFS', bfs_time), ('DFS (Stack)', dfs_time), 'bfs_vs_dfs_stack')

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