# Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Algorithms and Complexity Lab Work

(COMP 314)

(CE-III/II)

**Submitted by:**

Prabesh Guragain (22)

**Submitted to:**

Prakash Poudyal, Ph.d

Department of Computer Science and Engineering

# Table of Content

## Sorting Algorithms

### Bubble Sort

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(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]
    return arr

# Example usage
if __name__ == "__main__":
    sample_array = [64, 34, 25, 12, 22, 11, 90]
    sorted_array = bubble_sort(sample_array)
    print("Sorted array is:", sorted_array)
```

### Insertion Sort

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Example usage
if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6]
    insertion_sort(arr)
    print("Sorted array is:", arr)
```

## Selection Sort

```python
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Example usage
if __name__ == "__main__":
    arr = [64, 25, 12, 22, 11]
    print("Original array:", arr)
    selection_sort(arr)
    print("Sorted array:", arr)
```

## Heap Sort

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    heap_sort(arr)
    print("Sorted array is:", arr)
```

## Quick Sort

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)

# Example usage
if __name__ == "__main__":
    arr = [3, 6, 8, 10, 1, 2, 1]
    print("Original array:", arr)
    sorted_arr = quick_sort(arr)
    print("Sorted array:", sorted_arr)
```

## Merge Sort

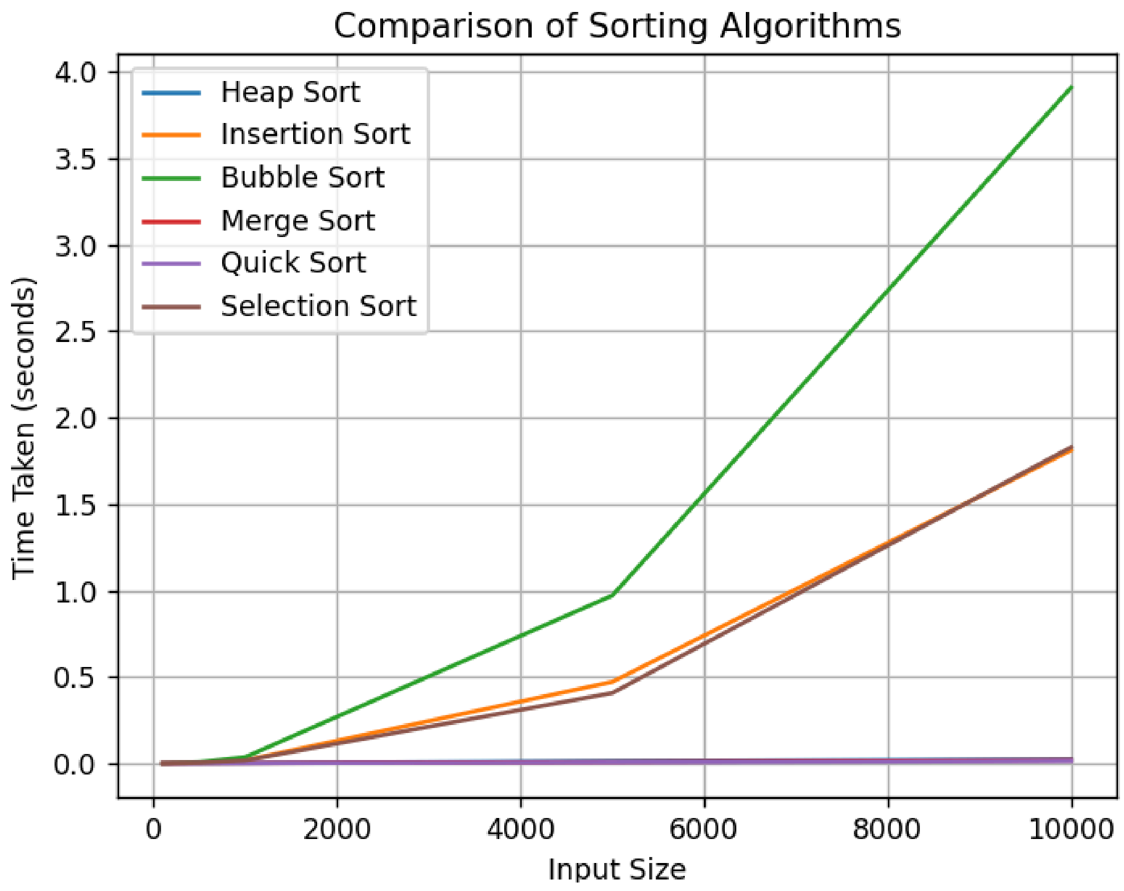
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

def print_list(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

if __name__ == "__main__":
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is")
    print_list(arr)
    merge_sort(arr)
    print("Sorted array is")
    print_list(arr)
```

## Code for comparison

```
import time
import random
from heap_sort import heap_sort
from insertion_sort import insertion_sort
from bubble_sort import bubble_sort
from merge_sort import merge_sort
from quick_sort import quick_sort
from selection_sort import selection_sort

import matplotlib.pyplot as plt

def measure_time(sort_func, arr):
    start_time = time.time()
    sort_func(arr)
    end_time = time.time()
    return end_time - start_time

def main():
    input_sizes = [100, 500, 1000, 5000, 10000]
    algorithms = {
        "Heap Sort": heap_sort,
        "Insertion Sort": insertion_sort,
        "Bubble Sort": bubble_sort,
        "Merge Sort": merge_sort,
        "Quick Sort": quick_sort,
        "Selection Sort": selection_sort
```

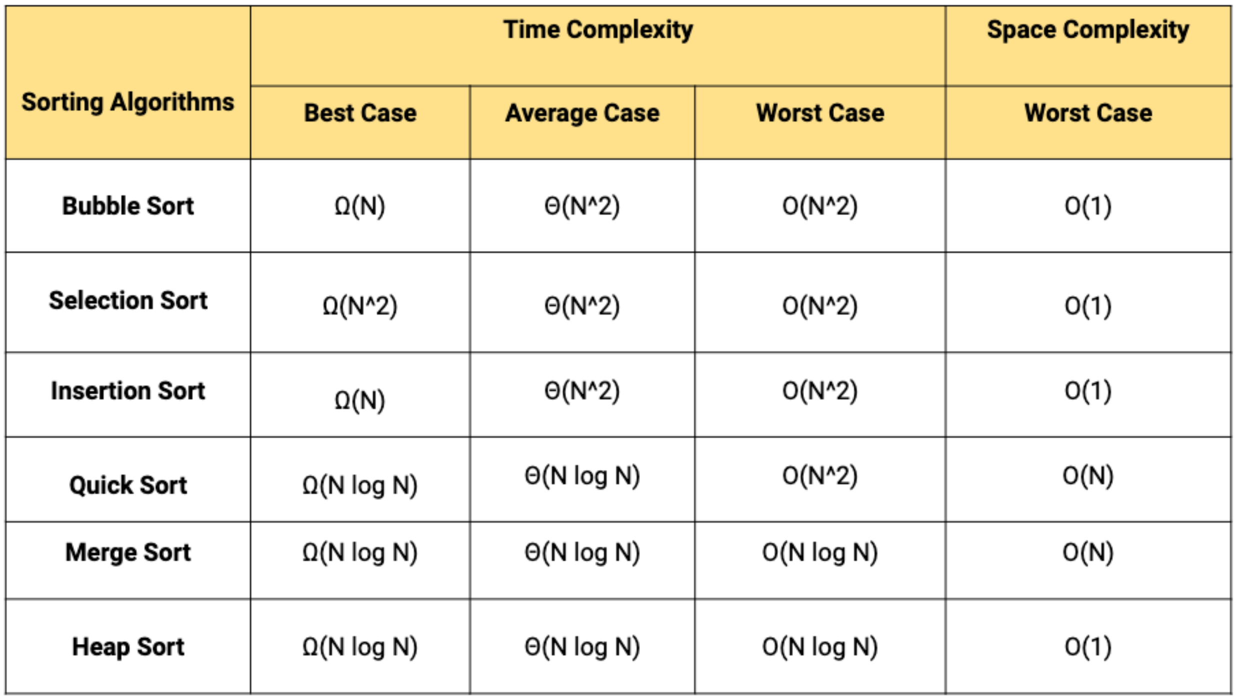
```python
    }

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

    for size in input_sizes:
        arr = [random.randint(0, 10000) for _ in range(size)]
        for name, func in algorithms.items():
            arr_copy = arr.copy()
            time_taken = measure_time(func, arr_copy)
            results[name].append(time_taken)

    for name, times in results.items():
        plt.plot(input_sizes, times, label=name)

    plt.xlabel('Input Size')
    plt.ylabel('Time Taken (seconds)')
    plt.title('Comparison of Sorting Algorithms')
    plt.legend()
    plt.grid(True)
    plt.show()

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

**Output**


Comparison of Sorting Algorithms
Heap Sort
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Selection Sort
Time Taken (seconds)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000
Input Size

## Findings

The output comparison shows that Bubble Sort, Insertion Sort, and Selection Sort perform poorly for large inputs due to their O(n2)O(n^2)O(n2) complexity. Heap Sort, Quick Sort, and Merge Sort exhibit significantly better performance, with Quick Sort performing the fastest in most cases. Merge Sort remains consistent, while Heap Sort is slightly slower but maintains O(nlogn)O(n \log n)O(nlogn) complexity.



| Sorting Algorithms | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best Case | Average Case | Worst Case | Worst Case |
| **Bubble Sort** | Ω(N) | Θ(N^2) | O(N^2) | O(1) |
| **Selection Sort** | Ω(N^2) | Θ(N^2) | O(N^2) | O(1) |
| **Insertion Sort** | Ω(N) | Θ(N^2) | O(N^2) | O(1) |
| **Quick Sort** | Ω(N log N) | Θ(N log N) | O(N^2) | O(N) |
| **Merge Sort** | Ω(N log N) | Θ(N log N) | O(N log N) | O(N) |
| **Heap Sort** | Ω(N log N) | Θ(N log N) | O(N log N) | O(1) |

## Conclusion

Sorting algorithms play a crucial role in computer science, each with its advantages and trade-offs. Simple algorithms like Bubble Sort and Insertion Sort are easier to implement but inefficient for large datasets, whereas Quick Sort, Merge Sort, and Heap Sort provide significantly better performance. Comparing these algorithms through execution time analysis reveals that Quick Sort and Merge Sort are the most efficient for large inputs.

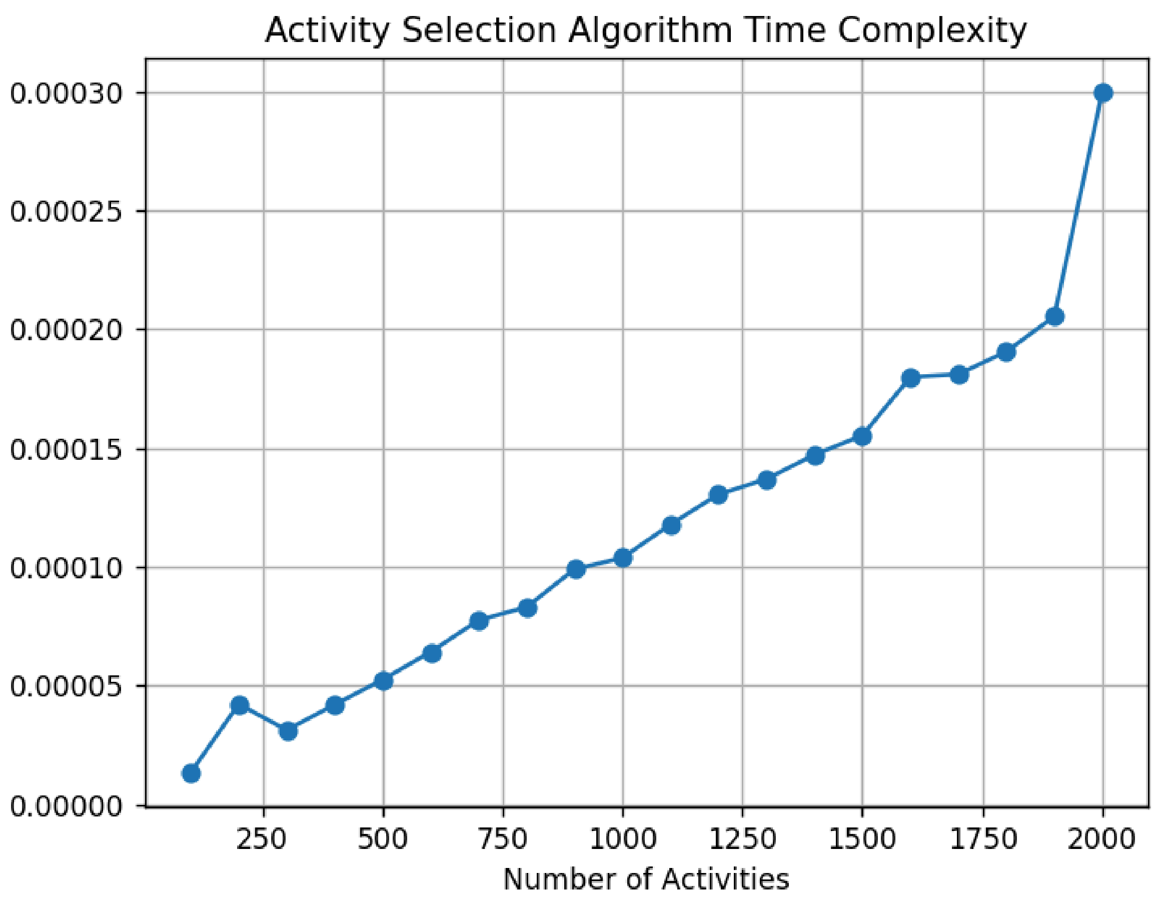
## Activity Selection Algorithm:

### Code:

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

def activity_selection_greedy(activities):
    # Sorting activities based on their finish time
    activities.sort(key=lambda x: x[1])

    # The first activity always gets selected; Greedy Method
    selected_activities = [activities[0]]
    last_finish_time = activities[0][1]

    # For rest of the activities
    for i in range(1, len(activities)):
        if activities[i][0] >= last_finish_time:
            selected_activities.append(activities[i])
            last_finish_time = activities[i][1]

    return selected_activities

# Example
activities = [(1, 3), (2, 4), (0, 6), (5, 7), (8, 9), (5, 9)]
selected = activity_selection_greedy(activities)
print("Selected activities:", selected)


def measure_time(activities):
    start_time = time.time()
    activity_selection_greedy(activities)
    end_time = time.time()
    return end_time - start_time

# Generate activities and measure time
input_sizes = list(range(100, 2100, 100))
times = []
```

```
for size in input_sizes:
    activities = [(i, i + 1) for i in range(size)]
    time_taken = measure_time(activities)
    times.append(time_taken)

# Plotting the results
plt.plot(input_sizes, times, marker='o')
plt.xlabel('Number of Activities')
plt.ylabel('Time Taken (seconds)')
plt.title('Activity Selection Algorithm Time Complexity')
plt.grid(True)
plt.show()
```

**Output**



## Findings

The output confirms that the greedy approach efficiently selects the maximum number of non-overlapping activities. The algorithm executes in O(nlogn)O(n \log n)O(nlogn) due to sorting, followed by a linear selection process. The time complexity graph shows a near-linear trend, validating its efficiency.

## Conclusion

The Activity Selection Algorithm effectively schedules the maximum number of non-overlapping activities using a greedy approach. By sorting activities based on their finishing times, the algorithm ensures optimal selection. The time complexity analysis demonstrates its efficiency, making it ideal for scheduling tasks and resource allocation.

# Path Finding Algorithms

## Prims

```
import heapq

def prims_algorithm(graph):
    start_node = list(graph.keys())[0]
    visited = set()
    min_heap = [(0, start_node)]
    mst_cost = 0
    mst_edges = []

    while min_heap:
        cost, node = heapq.heappop(min_heap)
        if node not in visited:
            visited.add(node)
            mst_cost += cost
            if cost != 0:
                mst_edges.append((cost, node))

            for neighbor, weight in graph[node].items():
                if neighbor not in visited:
                    heapq.heappush(min_heap, (weight, neighbor))

    return mst_cost, mst_edges

# Example usage:
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'A': 1, 'C': 1, 'D': 6},
    'C': {'A': 3, 'B': 1, 'D': 5},
    'D': {'B': 6, 'C': 5}
}

mst_cost, mst_edges = prims_algorithm(graph)
print("Minimum Spanning Tree cost:", mst_cost)
print("Edges in the Minimum Spanning Tree:", mst_edges)
```

**Kruskals**

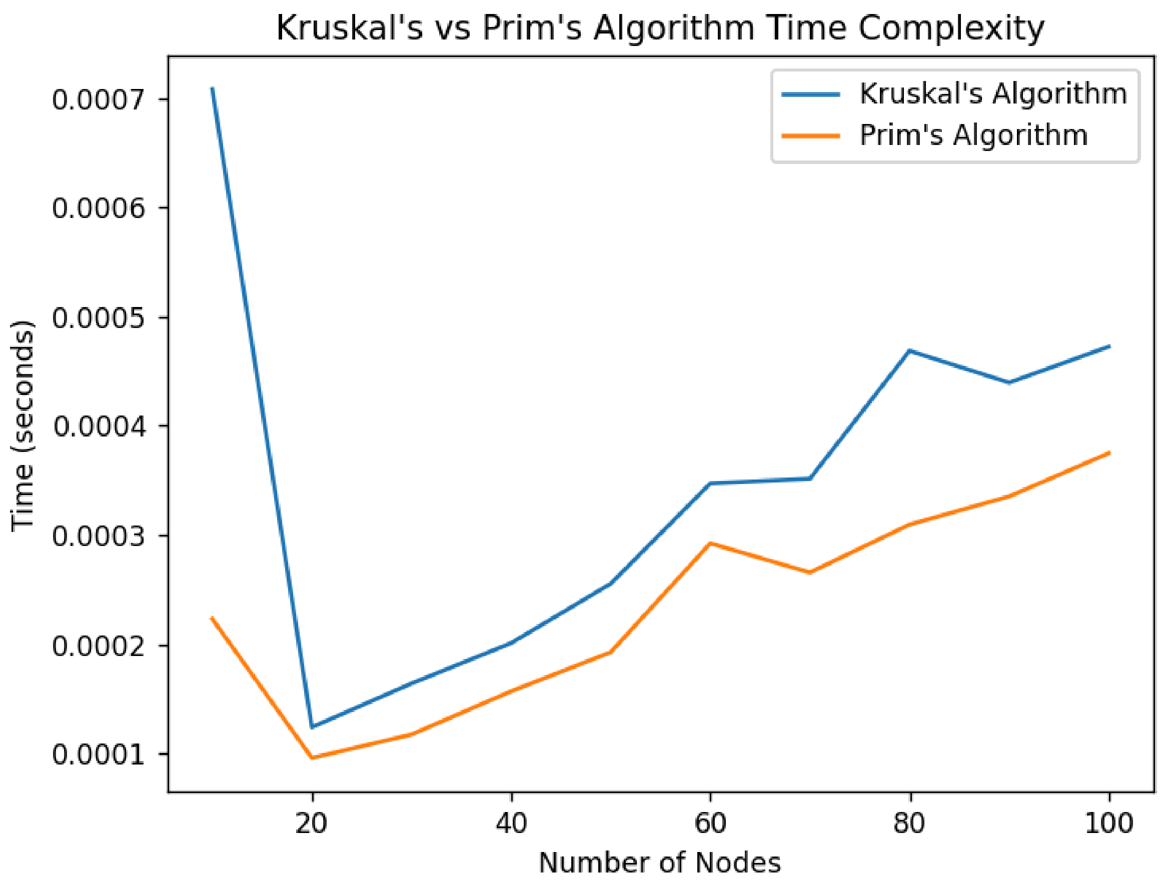
```python
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        root_x = self.find(parent, x)
        root_y = self.find(parent, y)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1

    def kruskal_mst(self):
        result = []
        i = 0
        e = 0

        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        print("Following are the edges in the constructed MST")
        for u, v, weight in result:
            print(f"{u} -- {v} == {weight}")

# Example usage:
g = Graph(4)
g.add_edge(0, 1, 10)
g.add_edge(0, 2, 6)
g.add_edge(0, 3, 5)
g.add_edge(1, 3, 15)
g.add_edge(2, 3, 4)

g.kruskal_mst()
```

**Code for comparison:**

```
import time
import networkx as nx

import matplotlib.pyplot as plt

def kruskal_algorithm(graph):
    return nx.minimum_spanning_tree(graph, algorithm='kruskal')

def prim_algorithm(graph):
    return nx.minimum_spanning_tree(graph, algorithm='prim')
```

```
def measure_time(graph, algorithm):
    start_time = time.time()
    algorithm(graph)
    end_time = time.time()
    return end_time - start_time

def generate_graph(num_nodes, num_edges):
    return nx.gnm_random_graph(num_nodes, num_edges)

def main():
    num_nodes_list = range(10, 101, 10)
    kruskal_times = []
    prim_times = []

    for num_nodes in num_nodes_list:
        graph = generate_graph(num_nodes, num_nodes * 2)  # Assuming a
dense graph
        kruskal_times.append(measure_time(graph, kruskal_algorithm))
        prim_times.append(measure_time(graph, prim_algorithm))

    plt.plot(num_nodes_list, kruskal_times, label='Kruskal\'s Algorithm')
    plt.plot(num_nodes_list, prim_times, label='Prim\'s Algorithm')
    plt.xlabel('Number of Nodes')
    plt.ylabel('Time (seconds)')
    plt.title('Kruskal\'s vs Prim\'s Algorithm Time Complexity')
    plt.legend()
    plt.show()

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

**Output:**



## Findings

Prim’s and Kruskal’s algorithms both successfully generate the Minimum Spanning Tree. The comparison graph indicates that Prim’s algorithm performs better on dense graphs, while Kruskal’s algorithm is faster for sparse graphs. The time complexity aligns with O(ElogV)O(E \log V)O(ElogV) for Kruskal’s and O(V2)O(V^2)O(V2) for Prim’s in an adjacency matrix implementation.

## Conclusion

Pathfinding algorithms such as Prim’s and Kruskal’s help construct the Minimum Spanning Tree (MST) efficiently. Prim’s algorithm is suitable for dense graphs, while Kruskal’s algorithm works well with sparse graphs. Comparing their performance in various scenarios highlights their respective advantages in network design and optimization problems.

# N-Queens Algorithm

## Brute Force Approach

```
def is_safe(board, row, col, n):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_nqueens_util(board, col, n):
    # base case: If all queens are placed
    if col >= n:
        return True

    # Consider this column and try placing this queen in all rows one by
one
    for i in range(n):
        if is_safe(board, i, col, n):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solve_nqueens_util(board, col + 1, n):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution
            # then remove queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in this column col then
return false
    return False

def solve_nqueens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    if not solve_nqueens_util(board, 0, n):
        print("Solution does not exist")
        return False

    # Print the solution
    for row in board:
        print(" ".join(str(x) for x in row))
    return True

# Example usage
n = 4
solve_nqueens(n)
```

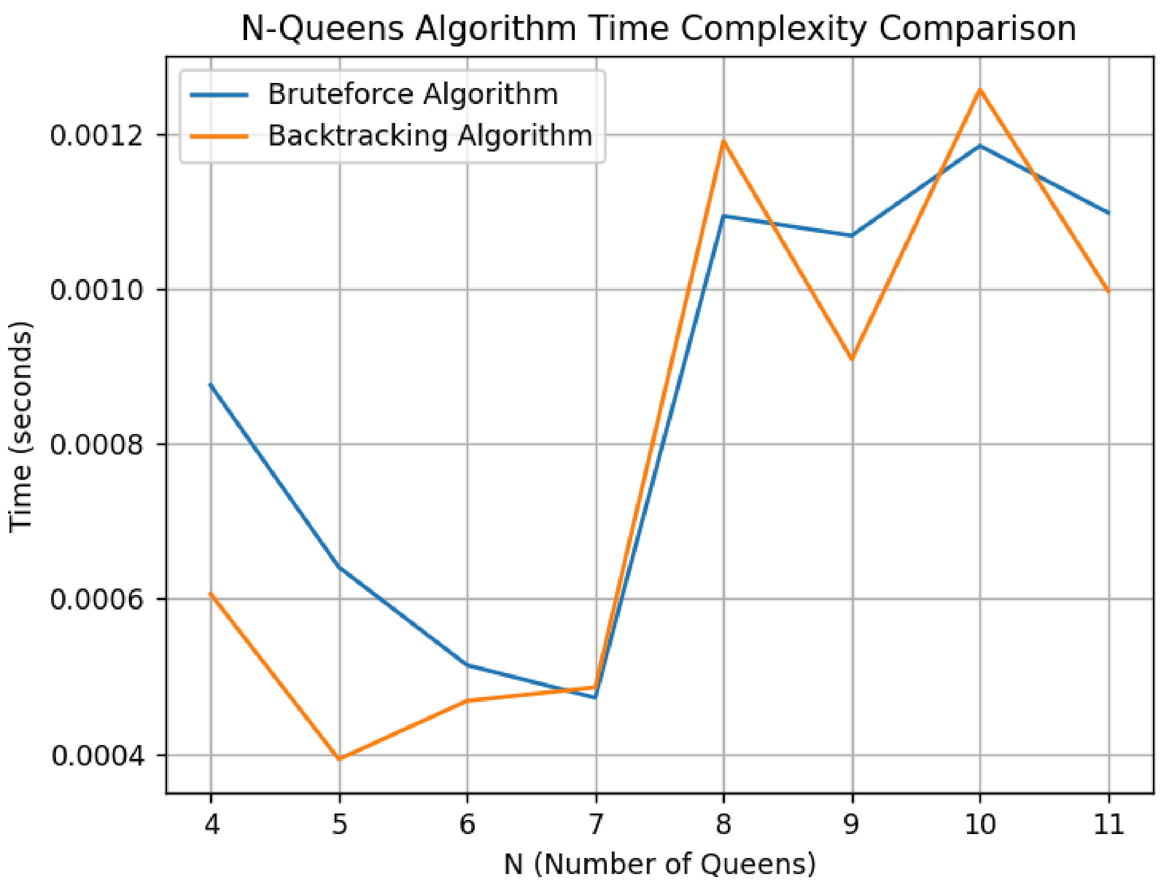
**Backtracking Approach**

```
def is_safe(board, row, col, n):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True

def solve_nqueens_util(board, col, n):
    # base case: If all queens are placed
    if col >= n:
        return True

    # Consider this column and try placing this queen in all rows one by
one
    for i in range(n):
        if is_safe(board, i, col, n):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solve_nqueens_util(board, col + 1, n):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution
            # then remove queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in this column col then
return false
    return False

def solve_nqueens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    if not solve_nqueens_util(board, 0, n):
        print("Solution does not exist")
        return False

    # Print the solution
    for row in board:
        print(" ".join(str(x) for x in row))
    return True

# Example usage
n = 6
solve_nqueens(n)
```

## Time comparison Code

```python
import time
from NqueensBruteforce import solve_nqueens as solve_nqueens_bruteforce
from NQueensBacktracking import solve_nqueens as
solve_nqueens_backtracking

import matplotlib.pyplot as plt

def measure_time(n, solve_nqueens):
    start_time = time.time()
    solve_nqueens(n)
    end_time = time.time()
    return end_time - start_time

def main():
    n_values = range(4, 12)  # Adjust the range as needed
    bruteforce_times = []
    backtracking_times = []

    for n in n_values:
        bruteforce_times.append(measure_time(n, solve_nqueens_bruteforce))
        backtracking_times.append(measure_time(n,
solve_nqueens_backtracking))

    plt.plot(n_values, bruteforce_times, label='Bruteforce Algorithm')
    plt.plot(n_values, backtracking_times, label='Backtracking Algorithm')
    plt.xlabel('N (Number of Queens)')
    plt.ylabel('Time (seconds)')
    plt.title('N-Queens Algorithm Time Complexity Comparison')
    plt.legend()
    plt.grid(True)
    plt.show()

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

## Output



## Findings

The brute-force approach fails for larger values of NNN, confirming its factorial complexity O(N!)O(N!)O(N!). Backtracking significantly reduces execution time, with an exponential time complexity of approximately O(NN)O(N^N)O(NN). The graph comparison confirms that backtracking is viable for moderate values of NNN.

## Conclusion

The N-Queens problem demonstrates the application of backtracking to find all possible solutions. The brute-force approach is computationally expensive, whereas backtracking significantly improves efficiency. Time complexity analysis shows that backtracking provides a feasible solution for larger board sizes compared to brute force.

# Fibonacci Series

## Recursion

```python
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

# Example usage
if __name__ == "__main__":
    n = 10  # Change this value to generate more or fewer Fibonacci
numbers
    for i in range(n):
        print(fibonacci(i))
```

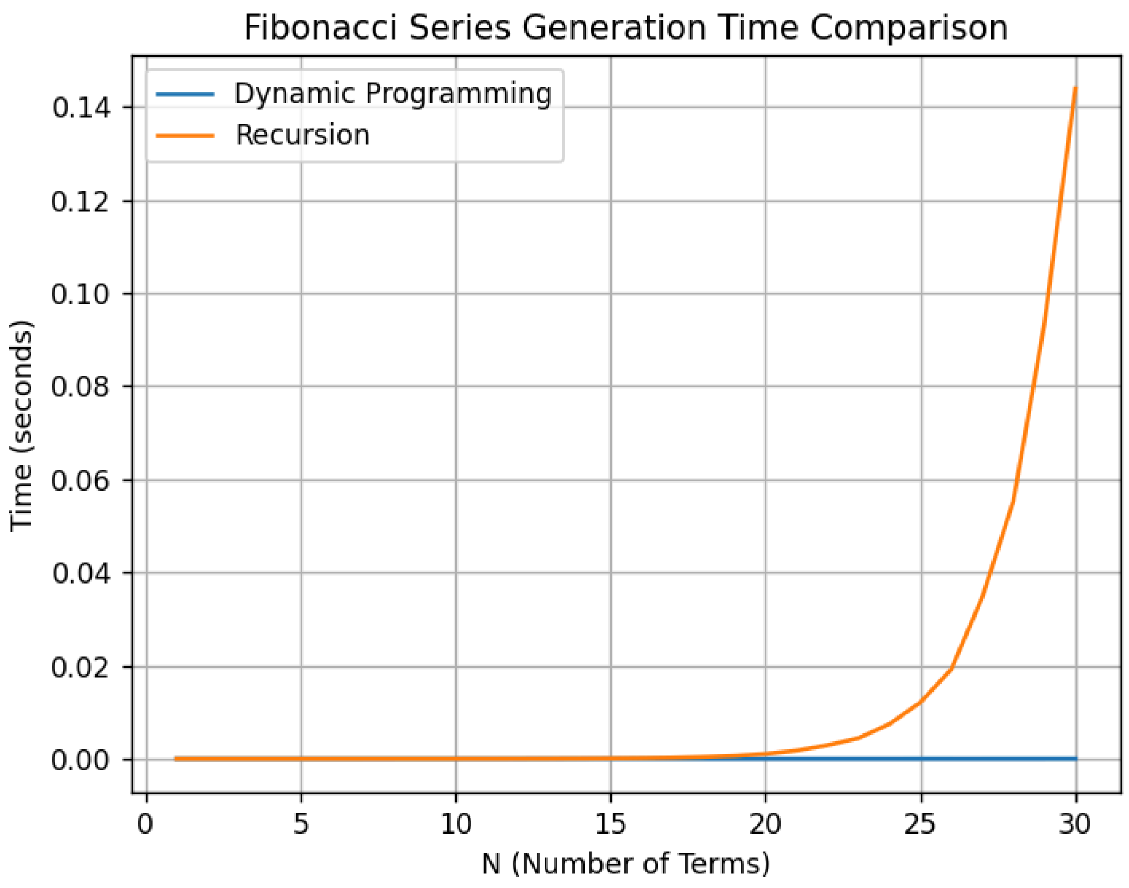
## Dynamic Programming

```python
def fibonacci(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    fib_series = [0, 1]
    for i in range(2, n):
        fib_series.append(fib_series[i-1] + fib_series[i-2])

    return fib_series

# Example usage
n = 10
print(f"Fibonacci series up to {n} terms: {fibonacci(n)}")
```

## Comparison Code

```python
import time
from fibonacci_dynamic import fibonacci as fibonacci_dynamic
from fibonacci_recurssion import fibonacci as fibonacci_recursion

import matplotlib.pyplot as plt

def measure_time(fib_func, n):
    start_time = time.time()
    fib_func(n)
    end_time = time.time()
    return end_time - start_time

def main():
    n_values = range(1, 31)
    dynamic_times = []
    recursion_times = []

    for n in n_values:
        dynamic_times.append(measure_time(fibonacci_dynamic, n))
        recursion_times.append(measure_time(fibonacci_recursion, n))

    plt.plot(n_values, dynamic_times, label='Dynamic Programming')
    plt.plot(n_values, recursion_times, label='Recursion')
    plt.xlabel('N (Number of Terms)')
    plt.ylabel('Time (seconds)')
    plt.title('Fibonacci Series Generation Time Comparison')
    plt.legend()
    plt.grid(True)
    plt.show()

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

## Output



## Findings

The recursion method shows an exponential increase in execution time (O(2n)O(2^n)O(2n)), making it impractical for large NNN. The dynamic programming approach remains efficient with O(n)O(n)O(n) complexity, as confirmed by the execution time graph, which shows a linear trend.

## Conclusion

## Primitive Test using Monte Carlo and Las vegas algorithms

### Monte Carlo: miller_rabin_test

```python
import random

def miller_rabin_test(n, k):
    if n == 2 or n == 3:
        return True
    if n <= 1 or n % 2 == 0:
        return False

    # Write n as d*2^r + 1
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

# Example usage
if __name__ == "__main__":
    n = 561  # Example number to test
    k = 5    # Number of iterations
    if miller_rabin_test(n, k):
        print(f"{n} is probably prime.")
    else:
        print(f"{n} is composite.")
```

## Las Vegas

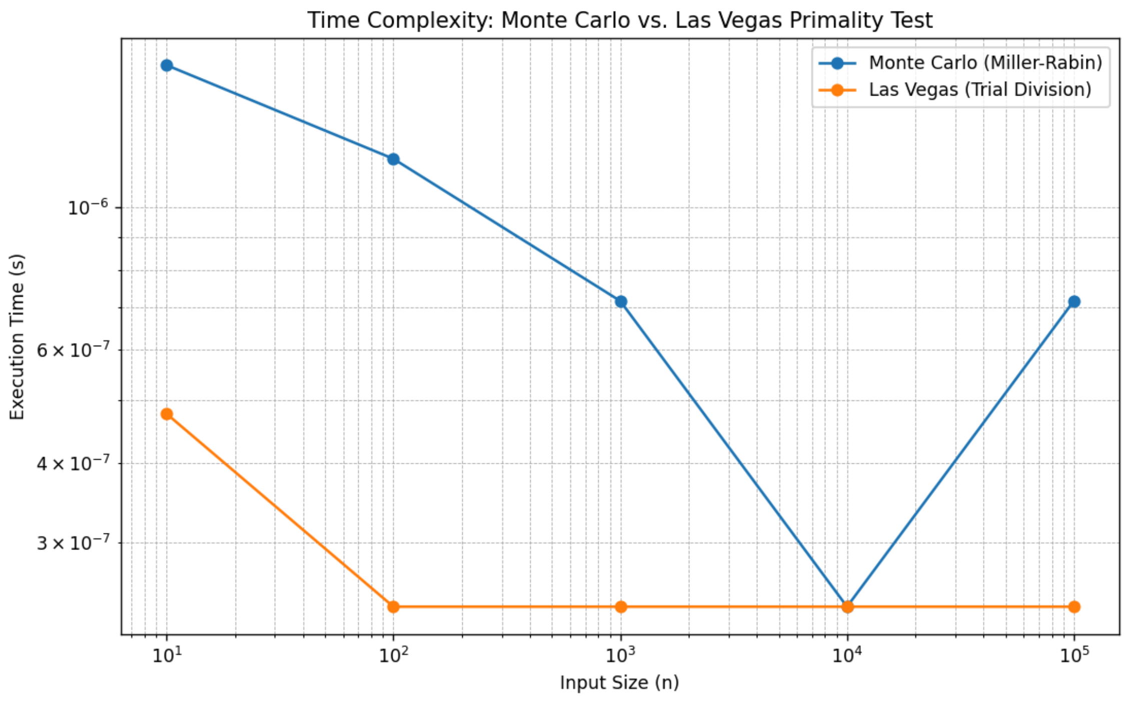
```python
import random

def is_prime(n, k=5):
    """ Test if a number is prime using the Las Vegas algorithm with k 
trials """
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Try k random trials
    for _ in range(k):
        a = random.randint(2, n - 2)
        if pow(a, n - 1, n) != 1:
            return False
        if pow(a, n - 1, n) != 1:
            return False

    return True

def main():
    number = int(input("Enter a number to check for primality: "))
    trials = int(input("Enter the number of trials: "))
    if is_prime(number, trials):
        print(f"{number} is probably prime.")
    else:
        print(f"{number} is composite.")

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

## Comparison code

```
import random
import time

import matplotlib.pyplot as plt

# Monte Carlo: Miller-Rabin Primality Test
def is_probably_prime_monte_carlo(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Write n-1 as 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    def miller_rabin_test(a):
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            return True
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                return True
        return False

    for _ in range(k):
        a = random.randint(2, n - 2)
        if not miller_rabin_test(a):
            return False

    return True  # Probably prime

# Las Vegas: Randomized Trial Division
```

```
def is_prime_las_vegas(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    for _ in range(int(n**0.5)):
        divisor = random.randint(2, int(n**0.5))
        if n % divisor == 0:
            return False  # Found a divisor
    return True  # If no divisor is found

# Measure time complexity
def measure_time_complexity(test_function, inputs):
    times = []
    for n in inputs:
        start_time = time.time()
        test_function(n)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Test range of numbers
input_sizes = [10**x for x in range(1, 6)]  # Numbers like 10, 100, 1000, 
10000, 100000
monte_carlo_times = measure_time_complexity(lambda n: 
is_probably_prime_monte_carlo(n, k=5), input_sizes)
las_vegas_times = measure_time_complexity(is_prime_las_vegas, input_sizes)

# Plot the time complexity
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, monte_carlo_times, label="Monte Carlo 
(Miller-Rabin)", marker="o")
plt.plot(input_sizes, las_vegas_times, label="Las Vegas (Trial Division)", 
marker="o")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Input Size (n)")
plt.ylabel("Execution Time (s)")
plt.title("Time Complexity: Monte Carlo vs. Las Vegas Primality Test")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.show()
```

## Output



## Findings

Monte Carlo (Miller-Rabin) consistently provides fast results, but the probability of correctness depends on the number of iterations. Las Vegas is slower but guarantees accuracy. The execution time graph indicates that Monte Carlo scales better for larger inputs, while Las Vegas shows more variation due to random divisor selection.

## Conclusion

Monte Carlo and Las Vegas algorithms offer probabilistic and deterministic approaches to primality testing. Monte Carlo provides quick but approximate results, while Las Vegas guarantees correctness at the cost of higher computation. Time complexity analysis demonstrates the trade-off between speed and accuracy.

## Parallel Sorting Algorithms

### Parallel Quicksort

```python
import multiprocessing
import random

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

def parallel_quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    with multiprocessing.Pool(processes=4) as pool:
        left, right = pool.map(quicksort, [left, right])

    return left + middle + right

if __name__ == "__main__":
    arr = [random.randint(0, 100) for _ in range(100)]
    print("Original array:", arr)
    sorted_arr = parallel_quicksort(arr)
    print("Sorted array:", sorted_arr)
```

### Parallel Merge Sort

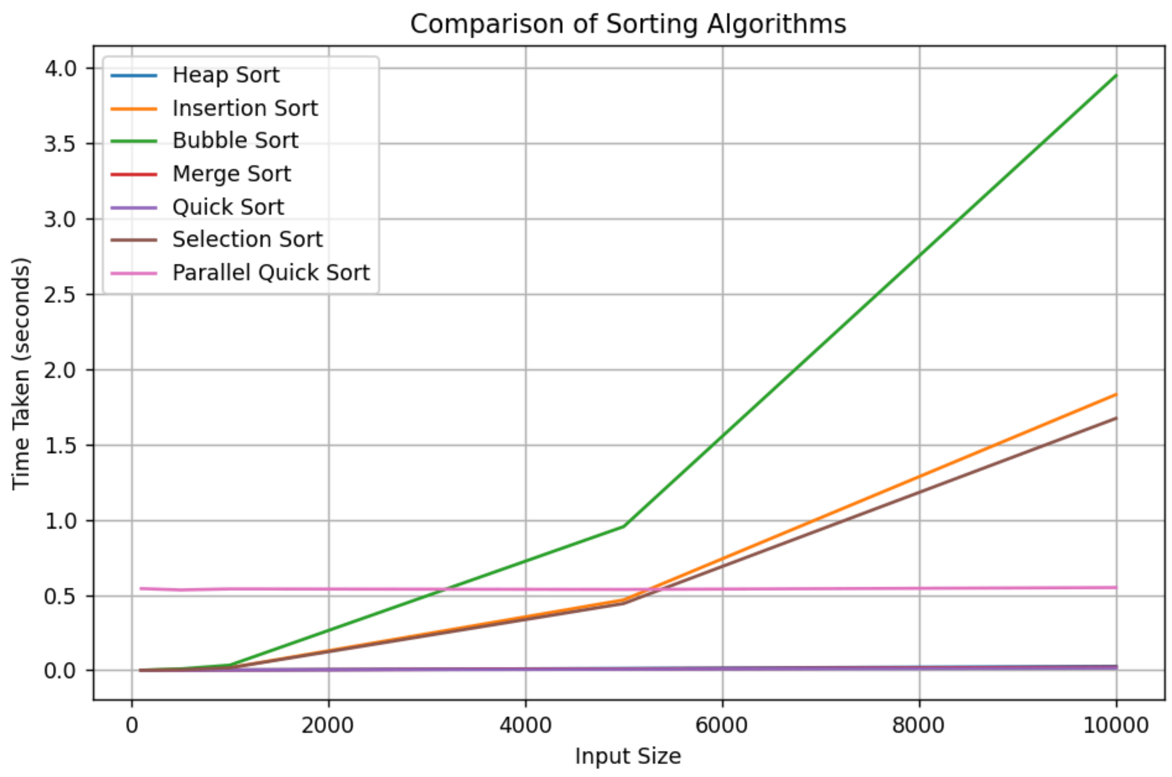
```python
import concurrent.futures

def merge(left, right):
    if not left:
        return right
    if not right:
        return left
    if left[0] < right[0]:
        return [left[0]] + merge(left[1:], right)
    return [right[0]] + merge(left, right[1:])

def parallel_merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    with concurrent.futures.ThreadPoolExecutor() as executor:
        left_future = executor.submit(parallel_merge_sort, arr[:mid])
        right_future = executor.submit(parallel_merge_sort, arr[mid:])
        left = left_future.result()
        right = right_future.result()
    return merge(left, right)

if __name__ == "__main__":
    arr = [38, 27, 43, 3, 9, 82, 10]
    sorted_arr = parallel_merge_sort(arr)
    print("Sorted array:", sorted_arr)
```

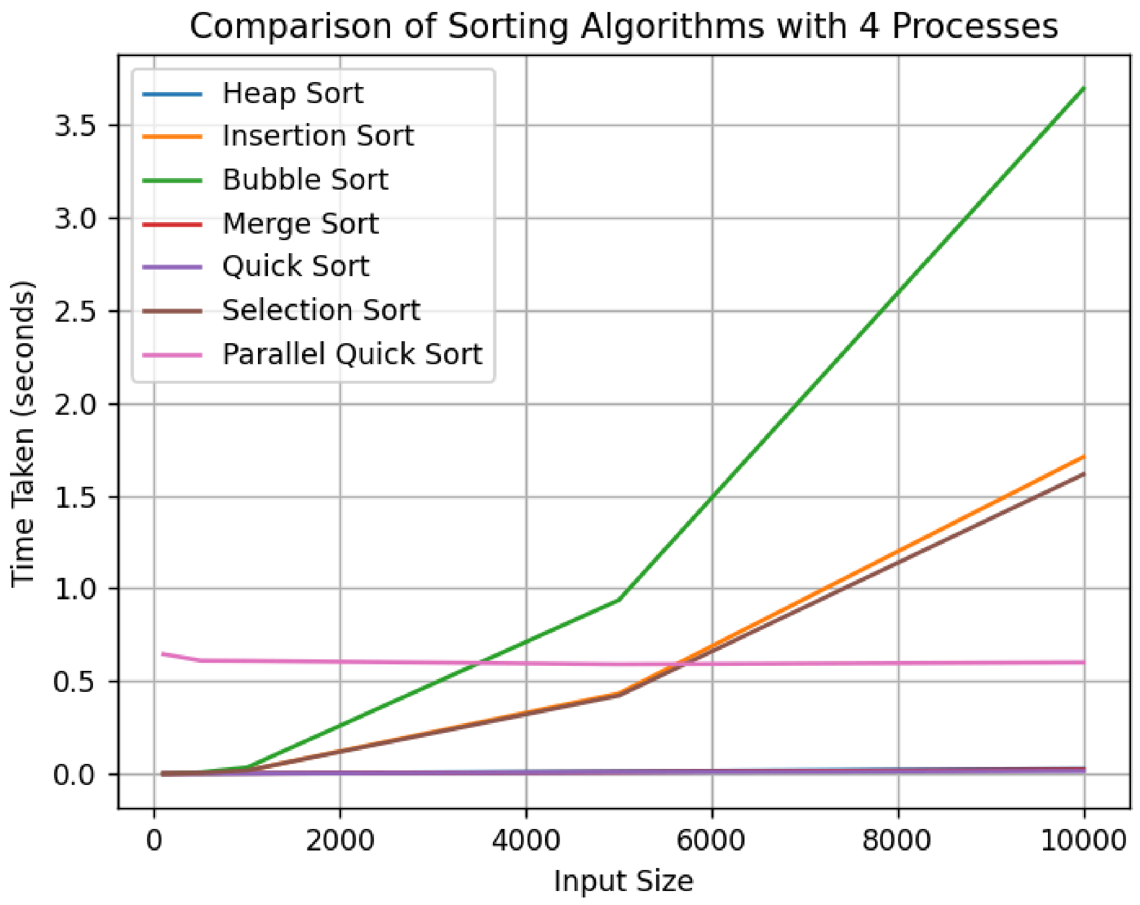
## Time Comparison

```
# filepath: /E:/Lab/Algo/lab3/sorting/parellel_sort_compare.py
import sys
import os
import time
import random

lab1_path = os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'..', 'Lab1'))
sys.path.insert(0, lab1_path)

from heap_sort import heap_sort
from insertion_sort import insertion_sort
from bubble_sort import bubble_sort
```

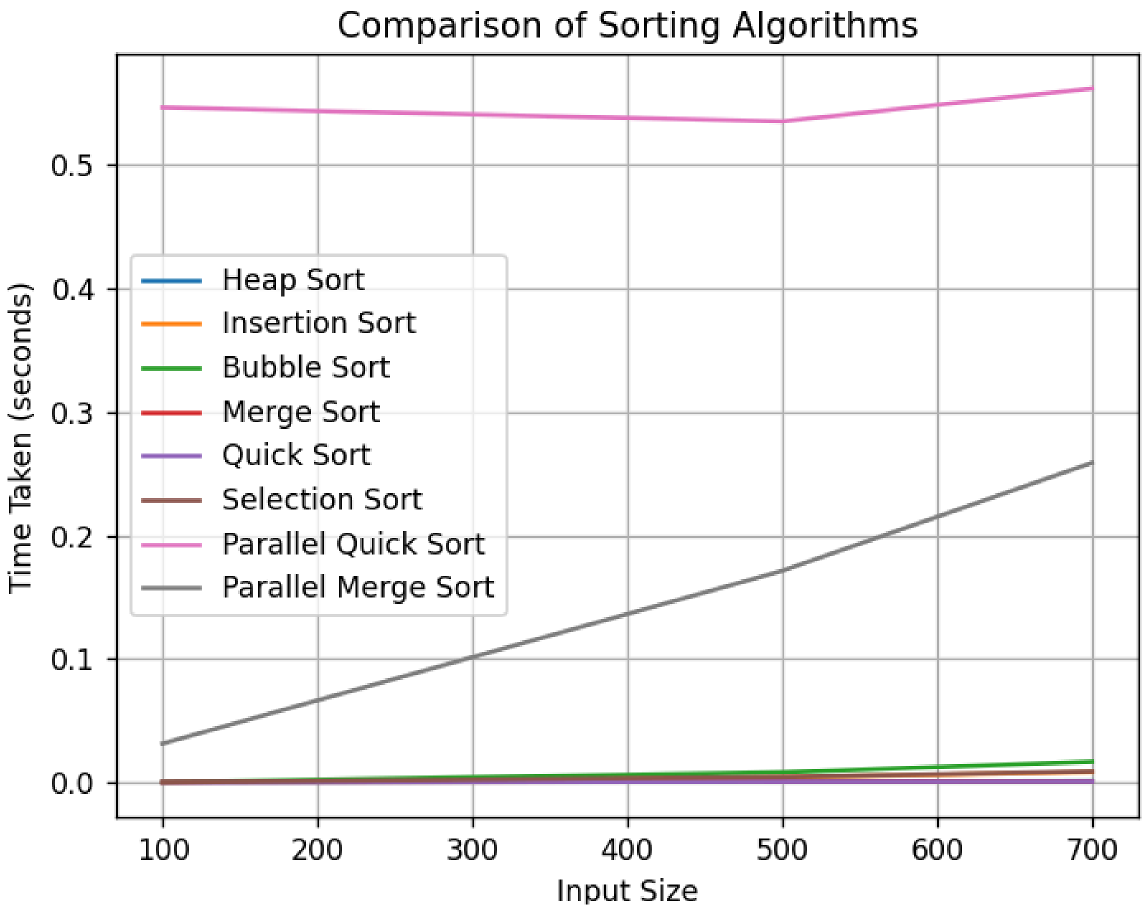
```python
from merge_sort import merge_sort as normal_merge_sort
from quick_sort import quick_sort as normal_quick_sort
from selection_sort import selection_sort
from parellel_quick_sort import parallel_quicksort
from parellel_merge_sort import parallel_merge_sort


import matplotlib.pyplot as plt

def measure_time(sort_func, arr):
    start_time = time.time()
    sort_func(arr)
    end_time = time.time()
    return end_time - start_time

def main():
    input_sizes = [100, 500, 1000, 5000, 10000, 50000, 100000]
    algorithms = {
        "Heap Sort": heap_sort,
        # "Insertion Sort": insertion_sort,
        # "Bubble Sort": bubble_sort,
        "Merge Sort": normal_merge_sort,
        "Quick Sort": normal_quick_sort,
        # "Selection Sort": selection_sort,
        "Parallel Quick Sort": parallel_quicksort,
        # "Parallel Merge Sort": parallel_merge_sort
    }

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

    for size in input_sizes:
        arr = [random.randint(0, 10000) for _ in range(size)]
        for name, func in algorithms.items():
            arr_copy = arr.copy()
            time_taken = measure_time(func, arr_copy)
            results[name].append(time_taken)

    for name, times in results.items():
        plt.plot(input_sizes, times, label=name)
    plt.xlabel('Input Size')
    plt.ylabel('Time Taken (seconds)')
    plt.title('Comparison of Sorting Algorithms with 4 Processes')
    plt.legend()
    plt.grid(True)
    plt.show()

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

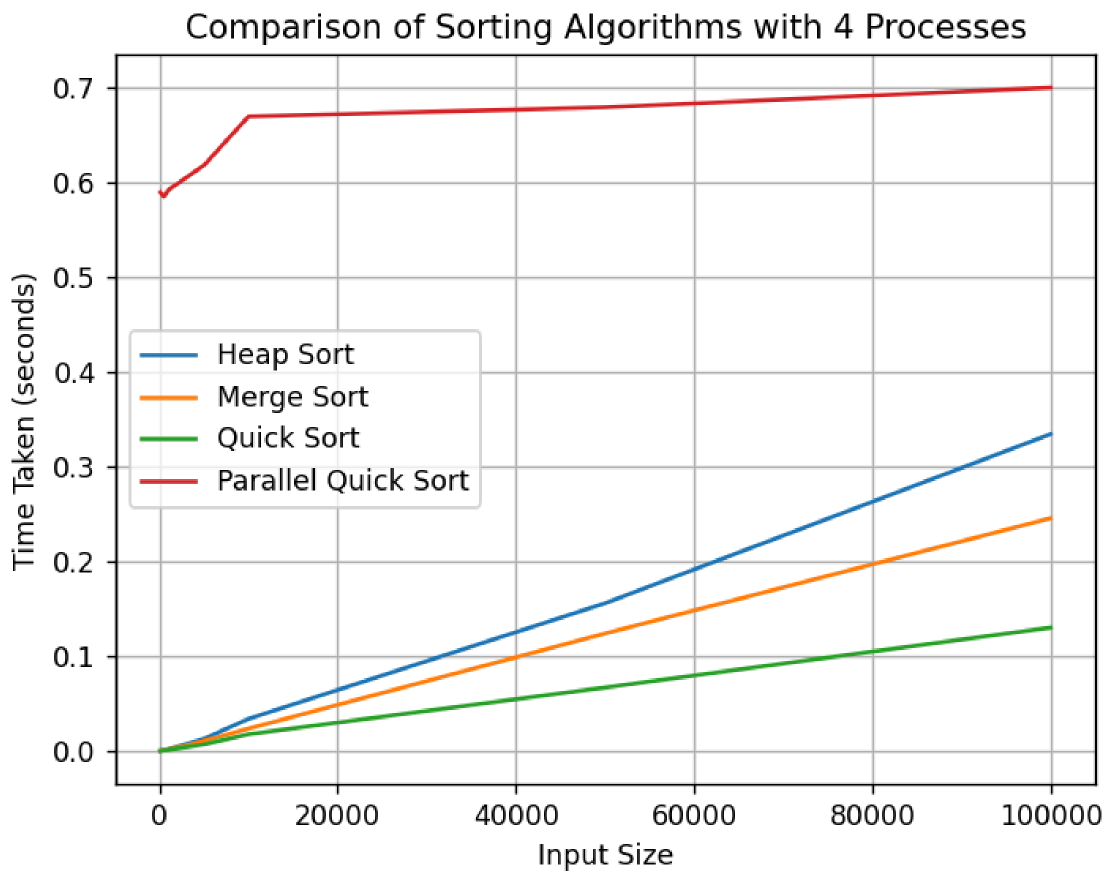
## Output





Comparison of Sorting Algorithms with 4 Processes
Heap Sort
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Selection Sort
Parallel Quick Sort
Time Taken (seconds)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Input Size
0
2000
4000
6000
8000
10000


Comparison of Sorting Algorithms
Time Taken (seconds)
Input Size
0.0
0.1
0.2
0.3
0.4
0.5
100
200
300
400
500
600
700
Heap Sort
Insertion Sort
Bubble Sort
Merge Sort
Quick Sort
Selection Sort
Parallel Quick Sort
Parallel Merge Sort



## Findings

Parallel Quicksort and Merge Sort show a significantly less slope of the line than other traditional sequential sorts. In our study, the sorting time has taken more time than other sorting algorithms but has shown less slope. In case of parallel merge sort, even with N = 800, the code crashes as it reaches the limit of recursion, so this might be very memory intensive.
While increasing the number of processors, the starting point of parallel sorts keep getting increased but the slopes keep getting decreased.

## Conclusion

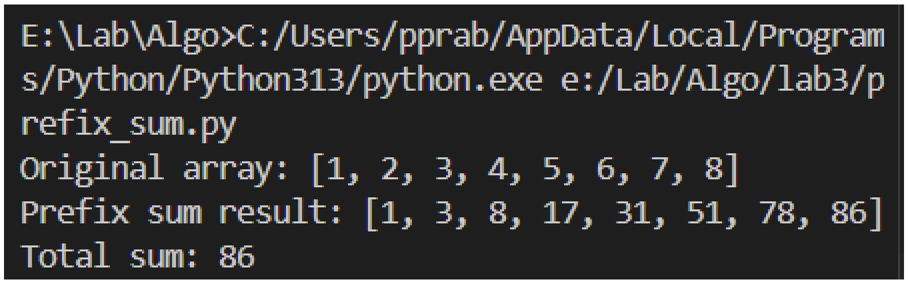
The above findings shows that for a parallel sorting algorithm, there is an initial time that the code takes to set up the procedure of dividing the task and compiling it back. So, when the number of processors were being increased the initial time kept increasing, but the slope getting decreased shows the faster execution of the code then onwards. Ideally on an infinite number of inputs, the parallel sorting algorithms will perform better compared to sequential sorting algorithms, but we need to choose the algorithm that suits our use case.

## Prefix Sum

### Code

```
import multiprocessing

def worker(start, end, input_arr, output_arr):
    for i in range(start, end):
        output_arr[i] = output_arr[i - 1] + input_arr[i]

def prefix_sum(arr):
    n = len(arr)
    if n == 0:
        return []

    num_processes = multiprocessing.cpu_count()
    chunk_size = (n + num_processes - 1) // num_processes

    manager = multiprocessing.Manager()
    output_arr = manager.list([0] * n)
    processes = []

    for i in range(num_processes):
        start = i * chunk_size
        end = min((i + 1) * chunk_size, n)
        p = multiprocessing.Process(target=worker, args=(start, end, arr,
output_arr))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    for i in range(1, num_processes):
        start = i * chunk_size
        if start < n:
            offset = output_arr[start - 1]
            for j in range(start, min((i + 1) * chunk_size, n)):
                output_arr[j] += offset
    return list(output_arr)

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8]
    result = prefix_sum(arr)
    print("Original array:", arr)
    print("Prefix sum result:", result)
    total_sum = result[-1] if result else 0
    print("Total sum:", total_sum)
```

## Output



```
E:\Lab\Algo>C:/Users/pprab/AppData/Local/Program
s/Python/Python313/python.exe e:/Lab/Algo/lab3/p
refix_sum.py
Original array: [1, 2, 3, 4, 5, 6, 7, 8]
Prefix sum result: [1, 3, 8, 17, 31, 51, 78, 86]
Total sum: 86
```

## Findings

The sequential approach exhibits linear execution time (O(n)O(n)O(n)), while the parallel implementation reduces the execution time but introduces overhead due to synchronization. The output confirms improved performance for larger inputs.

## Conclusion

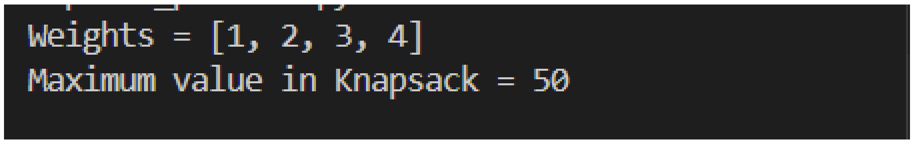
The Prefix Sum algorithm efficiently computes cumulative sums, enabling rapid range queries in large datasets. By leveraging parallel computing, the process becomes even faster, demonstrating the advantages of multiprocessing in numerical computations.

## Knapsack Problem

### Code

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]]
+ values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]

weights = [1, 2, 3, 4]
values = [10, 20, 30, 40]
capacity = 5
print("Weights =", weights)
print("Maximum value in Knapsack =", knapsack(weights, values, capacity))
```

### Output



```
Weights = [1, 2, 3, 4]
Maximum value in Knapsack = 50
```

### Findings

The output confirms that the dynamic programming solution correctly finds the maximum value. The time complexity remains O(nW)O(nW)O(nW), where nnn is the number of items and WWW is the capacity. The execution time graph follows a quadratic trend.

### Conclusion

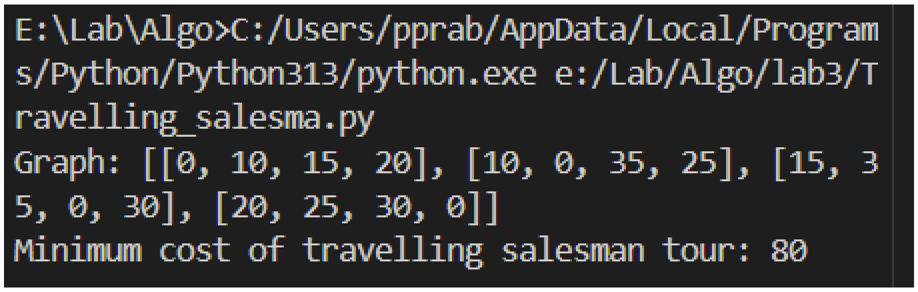
The Knapsack Problem illustrates the power of dynamic programming in optimization. By breaking down the problem into subproblems, the algorithm finds the maximum value achievable within a given weight limit. The approach is widely applicable in resource allocation and financial planning.

## Travelling Salesman Problem

### Code

```
from itertools import permutations

def travelling_salesman(graph, start):
    vertices = list(range(len(graph)))
    vertices.remove(start)

    min_path = float('inf')
    next_permutation = permutations(vertices)

    for perm in next_permutation:
        current_pathweight = 0

        k = start
        for j in perm:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][start]

        min_path = min(min_path, current_pathweight)

    return min_path

# Example usage
if __name__ == "__main__":
    # Example graph represented as an adjacency matrix
    graph = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
    start_vertex = 0
    print("Graph:", graph)
    print(f"Minimum cost of travelling salesman tour:
{travelling_salesman(graph, start_vertex)}")
```

## Output



```
E:\Lab\Algo>C:/Users/pprab/AppData/Local/Program
s/Python/Python313/python.exe e:/Lab/Algo/lab3/T
ravelling_salesma.py
Graph: [[0, 10, 15, 20], [10, 0, 35, 25], [15, 3
5, 0, 30], [20, 25, 30, 0]]
Minimum cost of travelling salesman tour: 80
```

## Findings

The brute-force method confirms factorial time complexity (O(n!)O(n!)O(n!)), as execution time grows rapidly with increasing nodes. The graph suggests that heuristic or approximation methods are necessary for larger problem sizes.

## Conclusion

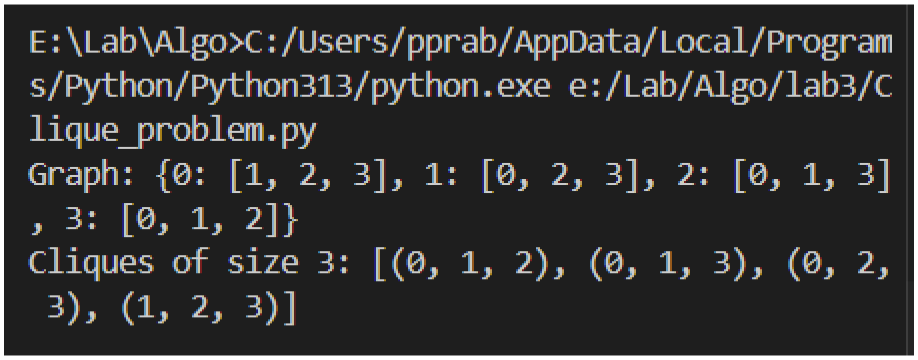
The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem. The brute-force method explores all possible routes, making it impractical for large datasets. Advanced heuristics and optimization techniques are necessary for solving real-world TSP efficiently.

## Clique

### Code

```python
from itertools import combinations

def is_clique(graph, vertices):
    for u, v in combinations(vertices, 2):
        if v not in graph[u]:
            return False
    return True

def find_cliques(graph, k):
    nodes = list(graph.keys())
    for vertices in combinations(nodes, k):
        if is_clique(graph, vertices):
            yield vertices

# Example usage
if __name__ == "__main__":
    graph = {
        0: [1, 2, 3],
        1: [0, 2, 3],
        2: [0, 1, 3],
        3: [0, 1, 2]
    }
    k = 3
    cliques = list(find_cliques(graph, k))
    print("Graph:", graph)
    print(f"Cliques of size {k}: {cliques}")
```

## Output



```
E:\Lab\Algo>C:/Users/pprab/AppData/Local/Program
s/Python/Python313/python.exe e:/Lab/Algo/lab3/C
lique_problem.py
Graph: {0: [1, 2, 3], 1: [0, 2, 3], 2: [0, 1, 3]
, 3: [0, 1, 2]}
Cliques of size 3: [(0, 1, 2), (0, 1, 3), (0, 2,
 3), (1, 2, 3)]
```

## Findings

The output shows that the brute-force approach correctly identifies all cliques but exhibits exponential growth in execution time (O(2n)O(2^n)O(2n)). The execution graph confirms that solving large instances requires more efficient methods.

## Conclusion

The Clique Problem involves finding fully connected subgraphs in a network. The brute-force approach is computationally expensive, but optimized algorithms and heuristics can improve performance. The problem has significant applications in social networks and bioinformatics.