**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**


# COMP 314

# Algorithms and Complexity

# Lab Report 3

**Submitted By:**

**Aayush Man Shakya(48)**

**Submitted To:**

**Dr. Prakash Poudyal**

**Department of Computer Science and Engineering**

**Date of Submission:**

**Jan 30, 2024**

# 1. Dynamic Programming (Fibonacci numbers)

## a. Traditional method (no dynamic programming)

Each time the fib() function is called, it gets broken down into two smaller subproblems. When it reaches the base case of either F(0) or F(1), it can finally return a result back to its caller. Most of these calls are redundant because you've already calculated their results. The required time grows exponentially because the function calculates many identical subproblems over and over again.

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

## b. Memoization approach

Instead of a new call every time, we can store the results of previous calls in something like a Python list. This technique is called memoization.

Memoization speeds up the execution of expensive recursive functions by storing previously calculated results in a cache. This way, when the same input occurs again, the function just has to look up the corresponding result and return it without having to run the computation again.

```
import sys
sys.setrecursionlimit(10000)
cache = {0: 0, 1: 1}

def fib(n):
    if n in cache:
        return cache[n]
    cache[n] = fib(n-1) + fib(n-2)
    return cache[n]

def reset_cache():
    global cache
    cache = {0: 0, 1: 1}
```

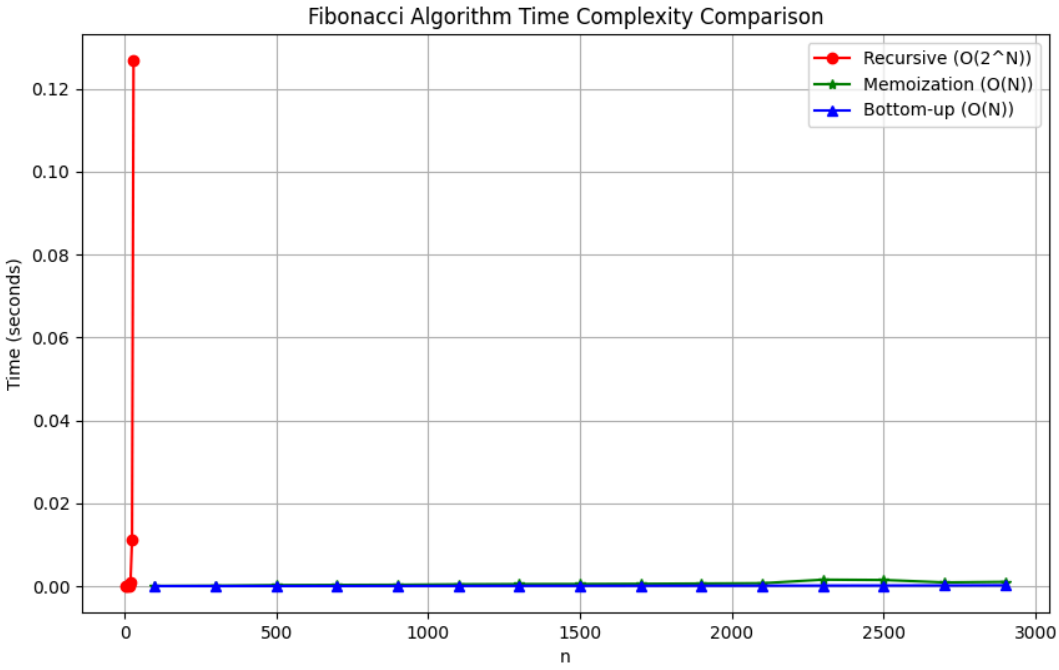
## c. Bottom Up Approach

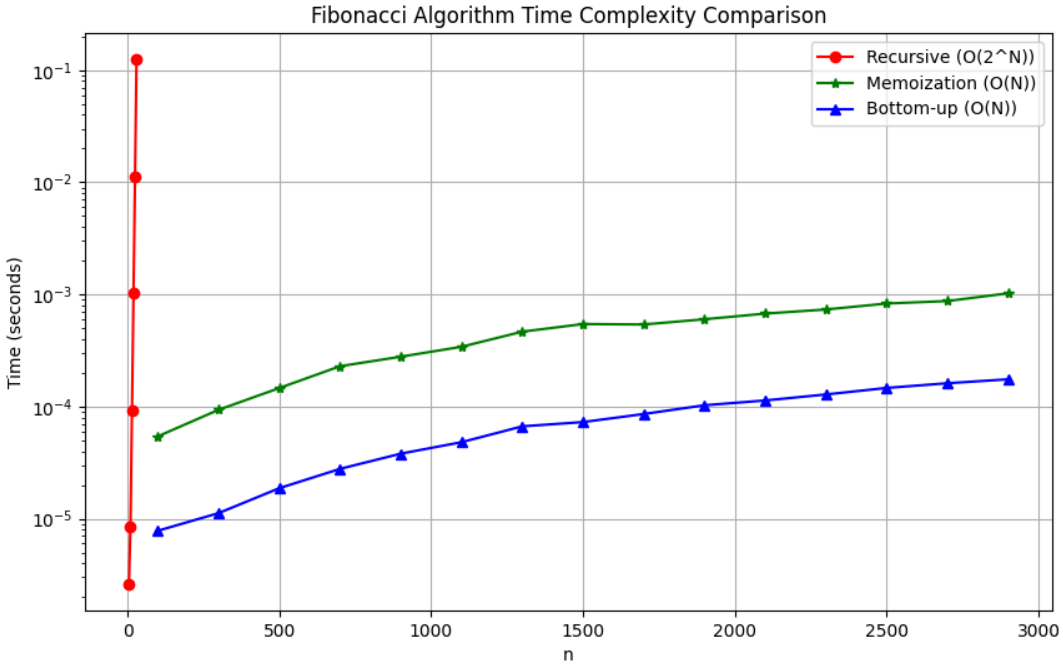
The bottom up approach starts with 0 and 1, then repeatedly updates the values by adding the previous two numbers until it reaches the $n^{th}$ term. This avoids the inefficiency of recursion by using a loop and constant space.

**The time complexity is O(n)**

```python
def fib(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n+1):
        a, b = b, a + b
    return b
```

**Comparison Graph:**






The first graph is plotted on a linear scale and shows that the $O(n^2)$ recursive approach is very inefficient even for small inputs. In the second graph, we use a logarithmic scale to better show the Y-axis and we see that the bottom up approach is faster than memoization (both take O(n)).

## 2. Probabilistic Algorithms

N-Queens algorithm is implemented to show probabilistic algorithms.

### a. Las Vegas Algorithm

The Las Vegas algorithm guarantees a valid solution (if one exists) but has variable runtime. It places queens column by column, using randomization and backtracking to avoid conflicts. It is efficient for small to moderate N.

```
import random

def lasvegas_nqueens(n, max_attempts=1000):
    def is_safe(board, row, col):
        for prev_col in range(col):
            if board[prev_col] == row or \
               abs(board[prev_col] - row) == abs(prev_col - col):
```

```
                return False
        return True

    for _ in range(max_attempts):
        board = [-1] * n
        for col in range(n):
            valid_rows = [row for row in range(n) if is_safe(board,
row, col)]
            if not valid_rows:
                break  # Conflict found, restart
            board[col] = random.choice(valid_rows)
        if -1 not in board:
            return board  # Success
    return None  # Failure after max_attempts
```

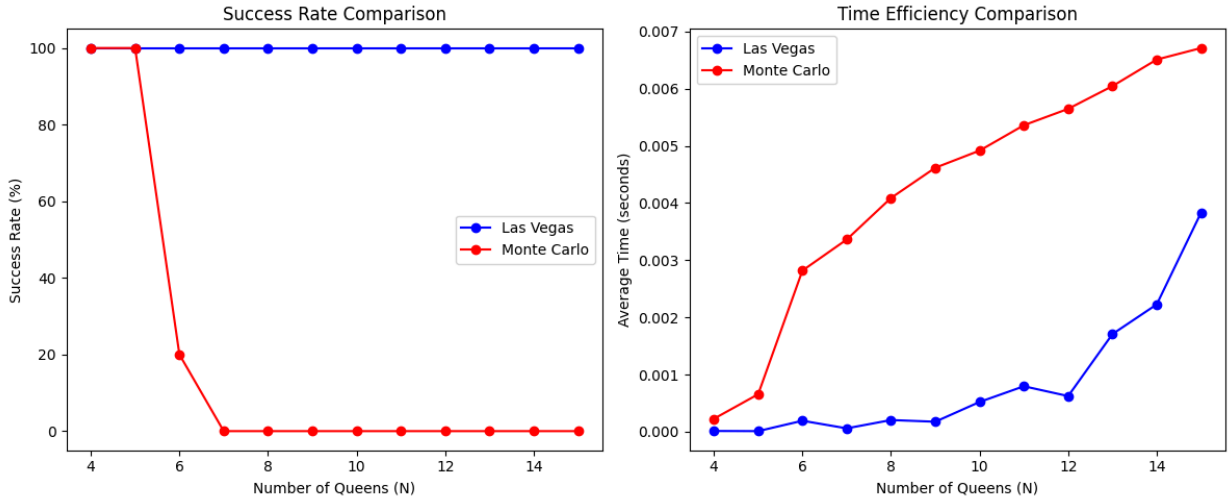
## b. Monte Carlo Algorithm

The Monte Carlo algorithm prioritizes speed over guaranteed correctness. It randomly generates complete board configurations and checks for validity. It returns the valid board if found within max_trials. The success probability drops exponentially as n increases.

```
import random

def montecarlo_nqueens(n, max_trials=1000):
    def is_valid(board):
        for col in range(n):
            for prev_col in range(col):
                if board[prev_col] == board[col] or \
                   abs(board[prev_col] - board[col]) == abs(prev_col
- col):
                    return False
        return True

    for _ in range(max_trials):
        board = [random.randint(0, n-1) for _ in range(n)]
```

```
        if is_valid(board):
            return board  # Success
    return None  # Failure after max_trials
```

**Comparison Graphs:**




From the left graph, we can see that the las vegas algorithm always has a 100% success rate while monte carlo has a low success rate for bigger board sizes.

## 3. Parallel Algorithms

### a. Parallel Quicksort

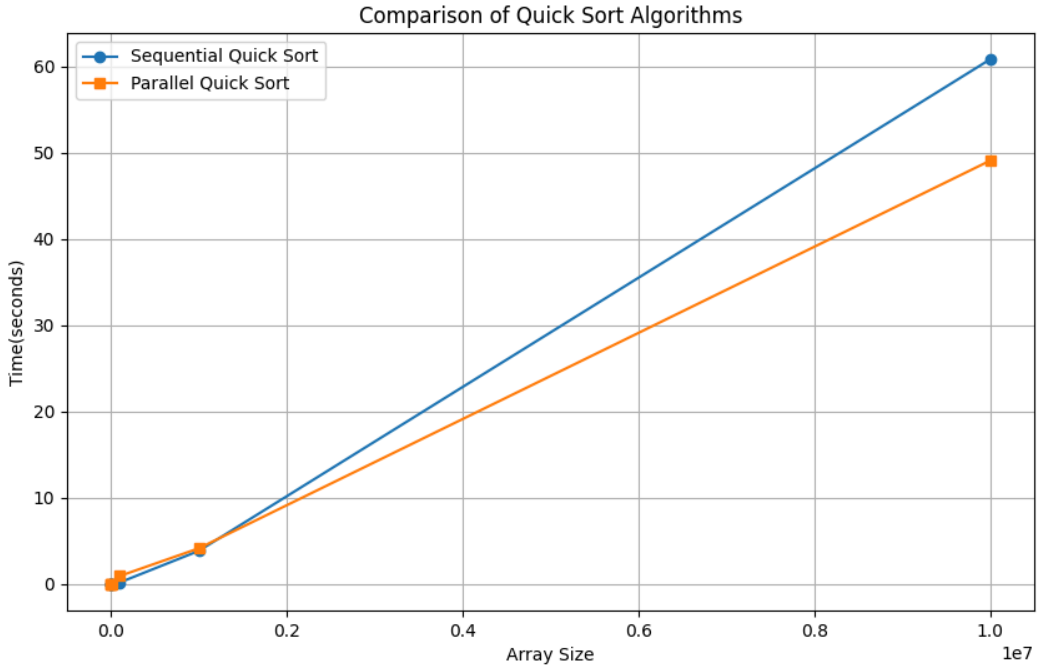
Parallel Quick Sort algorithm is designed to enhance performance for large datasets by leveraging multiprocessing. The algorithm follows the standard Quick Sort approach: selecting a pivot (middle element), partitioning the array into elements less than, equal to, and greater than the pivot, and recursively sorting the partitions. However, it introduces parallelism, where the left and right partitions are sorted concurrently using two separate processes.

**Code:**

```python
import multiprocessing
from quick_sort import quick_sort

def parallel_quick_sort(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]

    if len(arr) > 10000:
        with multiprocessing.Pool(2) as pool:
            left, right = pool.map(quick_sort, [left, right])
    else:
        left = quick_sort(left)
        right = quick_sort(right)

    return left + middle + right
```

**Comparison with Traditional Quick Sort:**

```
Size: 10, Sequential: 0.00s, Parallel: 0.00s
Size: 100, Sequential: 0.00s, Parallel: 0.00s
Size: 1000, Sequential: 0.00s, Parallel: 0.00s
Size: 10000, Sequential: 0.02s, Parallel: 0.02s
Size: 100000, Sequential: 0.20s, Parallel: 0.78s
Size: 1000000, Sequential: 3.16s, Parallel: 3.15s
Size: 10000000, Sequential: 50.99s, Parallel: 39.14s
Size: 100000000, Sequential: 763.06s, Parallel: 498.18s
```




After comparing parallel quick sort with traditional sequential quick sort on input arrays of size $10^1$ to $10^8$, we can see that for smaller input sizes, sequential approach is faster. But, for bigger inputs, parallel computing is much better because of multiple simultaneous computations.

## b. Parallel Merge Sort

Like traditional Merge Sort, the algorithm divides the input array into left and right halves, recursively sorts them, and merges the sorted subarrays. However, it introduces parallelism, and the left and right partitions are sorted concurrently using two processes via Python's multiprocessing.Pool, distributing the workload across CPU cores to reduce sorting time.

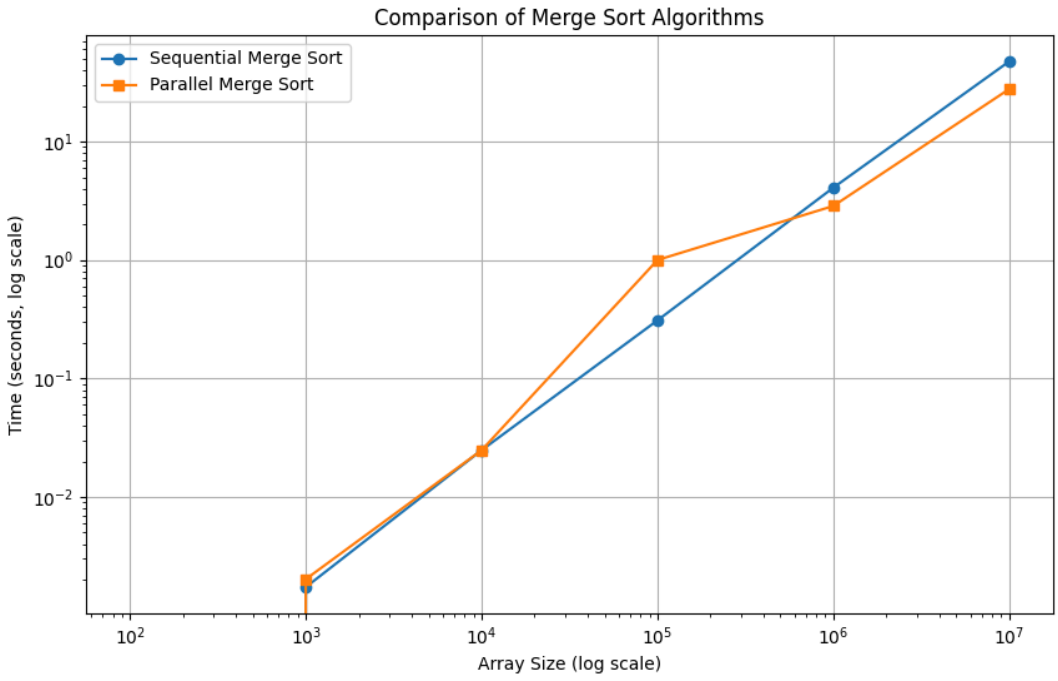
**Code:**

```
import multiprocessing
from merge_sort import merge, merge_sort

def parallel_merge_sort(arr):
    if len(arr) <= 1:
        return arr
```

```
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    if len(arr) > 10000:  # Threshold for parallel processing
        with multiprocessing.Pool(2) as pool:
            left, right = pool.map(merge_sort, [left, right])
    else:
        left = merge_sort(left)
        right = merge_sort(right)

    return merge(left, right)
```

**Comparison with Traditional Merge Sort:**

```
Size: 100, Sequential: 0.00s, Parallel: 0.00s
Size: 1000, Sequential: 0.00s, Parallel: 0.00s
Size: 10000, Sequential: 0.02s, Parallel: 0.02s
Size: 100000, Sequential: 0.31s, Parallel: 1.00s
Size: 1000000, Sequential: 4.08s, Parallel: 2.86s
Size: 10000000, Sequential: 47.44s, Parallel: 27.73s
```



After comparing parallel merge sort with traditional merge sort on input arrays of size $10^1$ to $10^7$, we can see that for smaller input sizes, sequential approach is faster. But, for bigger inputs, parallel computing is much better because of multiple simultaneous computations.

## 4. Comparing with bruteforce

### a. Maximum Clique Problem

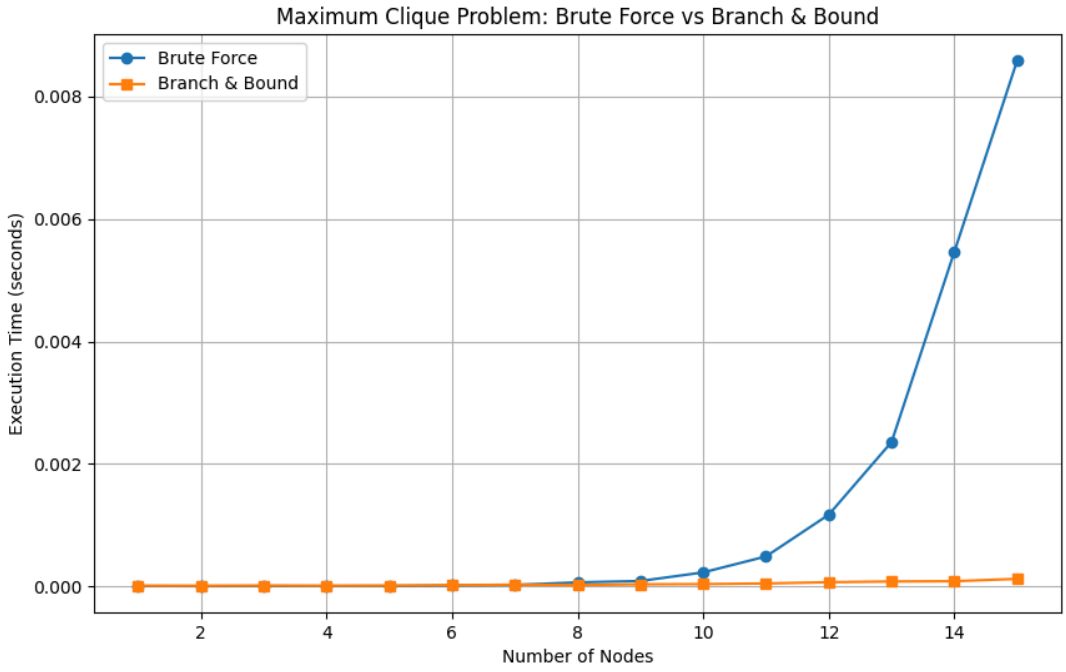
Given a small graph with N nodes and E edges, the task is to find the maximum clique in the given graph. A clique is a complete subgraph of a given graph. This means that all nodes in the said subgraph are directly connected to each other. The maximal clique is the complete subgraph of a given graph which contains the maximum number of nodes.

For the maximum clique problem, we implement it using a **branch and bound** approach.

```
def max_clique_branch_and_bound(graph):
    n = len(graph)
    if n == 0:
        return 0
    best_size = [0]  # Using list to allow modification in nested
function

    def backtrack(current_clique, candidates):
        # Update best size if current clique is larger
        if len(current_clique) > best_size[0]:
            best_size[0] = len(current_clique)

        # Prune if remaining candidates can't improve the solution
        if len(current_clique) + len(candidates) <= best_size[0]:
            return

        for i, node in enumerate(candidates):
```

```
            # Check if node is connected to all nodes in current
clique
            if all(graph[node][v] for v in current_clique):
                # Generate new candidates: nodes after current one
connected to 'node'
                new_candidates = [v for v in candidates[i+1:] if
graph[v][node]]
                backtrack(current_clique + [node], new_candidates)

    backtrack([], list(range(n)))
    return best_size[0] if best_size[0] > 0 else 1
```

**Comparison Graph:**




## b. 0/1 Knapsack Problem

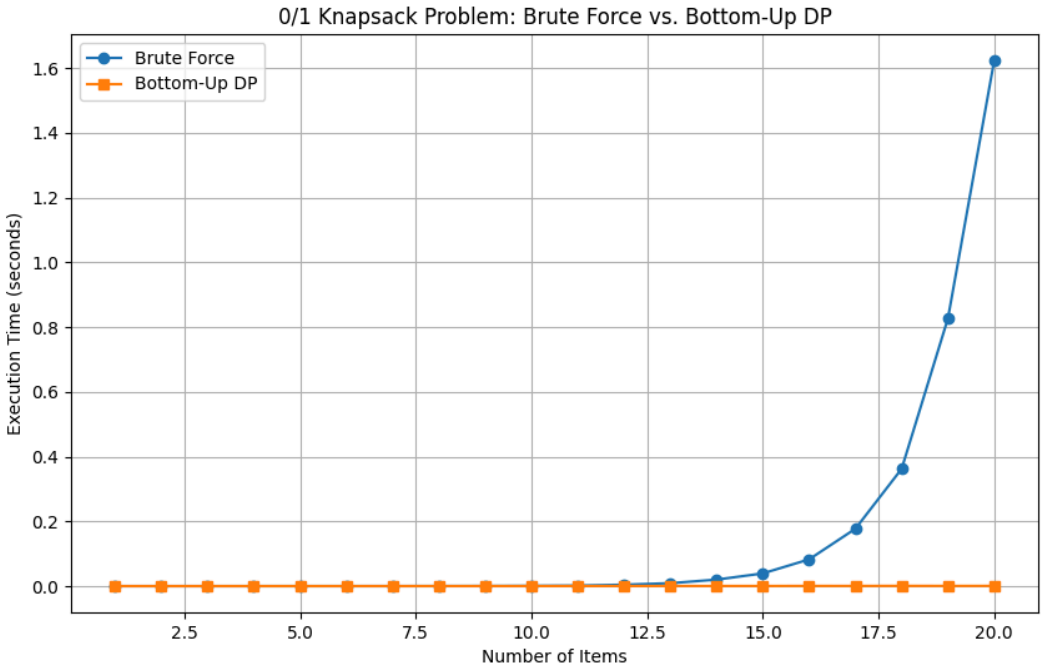
The Knapsack Problem says that: Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

We use a bottom up dynamic programming approach to implement it

```python
def knapsack_bottom_up(values, weights, capacity):
    dp = [0] * (capacity + 1)
    for i in range(len(values)):
        for w in range(capacity, weights[i] - 1, -1):
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    return dp[capacity]
```

**Comparison Graph:**



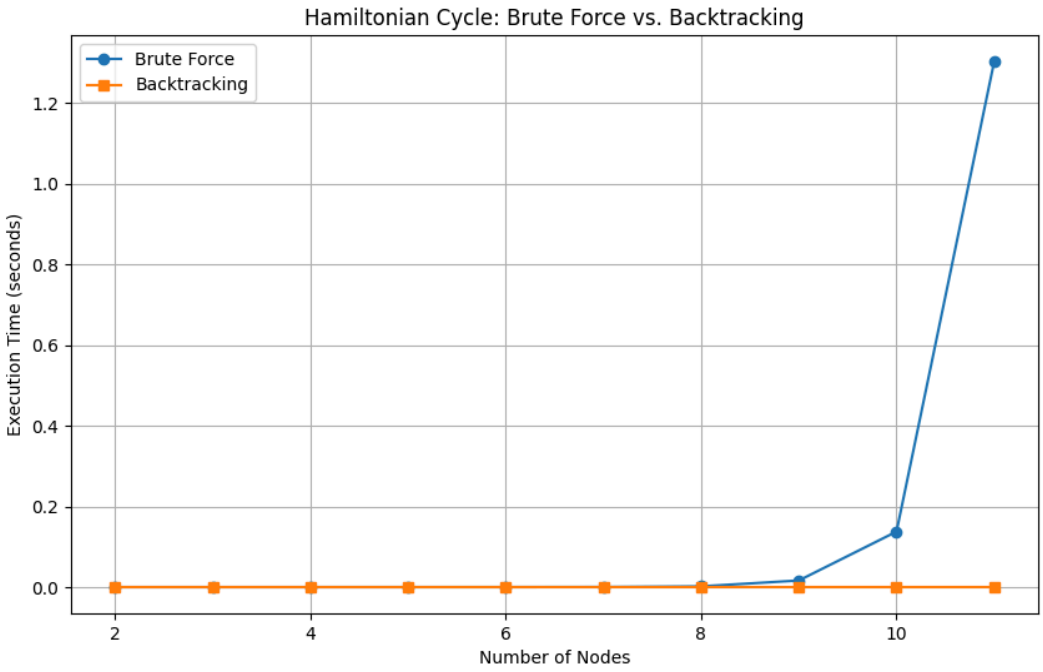
## c. Hamilton Cycle Problem

Hamiltonian Cycle or Circuit in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex. Finding a Hamiltonian Cycle in a graph is a well-known NP-complete problem, which means that there's no known efficient algorithm to solve it for all types of graphs.

We implement backtracking-based algorithm to determine if a Hamiltonian cycle exists in an undirected graph.

```python
def hamiltonian_backtracking(graph):
    n = len(graph)
    if n <= 1:
        return False
    path = [0]
    visited = [False] * n
    visited[0] = True

    def backtrack(current):
        if len(path) == n:
            return graph[current][0] == 1
        for v in range(n):
            if graph[current][v] and not visited[v]:
                visited[v] = True
                path.append(v)
                if backtrack(v):
                    return True
                # Backtrack
                path.pop()
                visited[v] = False
        return False

    return backtrack(0)
```

**Comparison Graph:**




We can see that for all three problems, brute force algorithm performs worse for big inputs and is impractical.

## Conclusion

In this lab, we implemented dynamic programming and parallel computing and compared them with traditional approaches. We also implemented and compared probabilistic algorithms like Monte Carlo and Las Vegas. And we compared recommended implementations of Maximum Clique Problem, Hamilton Cycle Problem and 0/1 Knapsack Problem with their backtracking approach.