Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab 3

COMP 314

(CE-III/II)

Submitted by

Puspa Hari Ghimire (19)

Submitted to

Dr. Prakash Poudyal

Department of Computer Science and Engineering

Submission Date:

January 27, 2025

Lab 3: Implementation of Dynamic Programming, Parallel and Probabilistic Algorithms and NP-Complete Problems

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. However, it applies when the subproblems overlap, i.e. when subproblems share sub-subproblems. It solves each sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub-subproblem.

There are two ways to implement dynamic programming:

1. Memoization - Top down approach which maintain a map of already solved sub problems.
2. Tabulation - Bottom up approach that solves all related sub-problems first. Based on the results in the table, we compute the solution to the "top" / original problem.

Fibonacci numbers:

- F(n) = F(n-1) + F(n-2)
- F(0) = 0
- F(1) = 1

The beginning of the sequence is thus:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Fibonacci numbers: Recursion

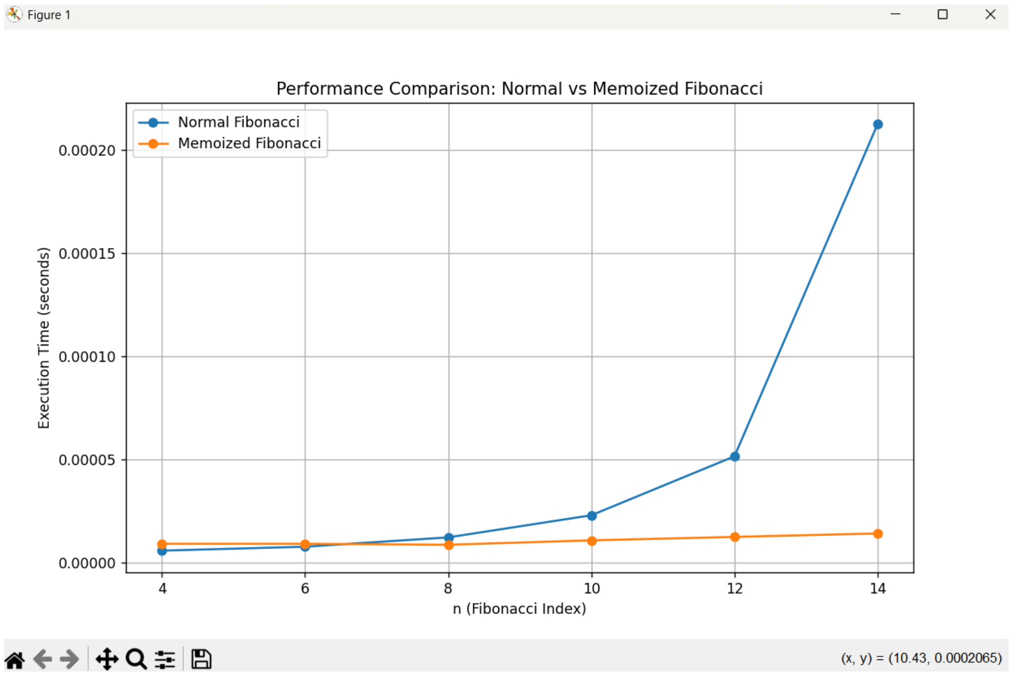
```
def Fib(n):
    if n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1) + Fib(n-2)
    return result
```

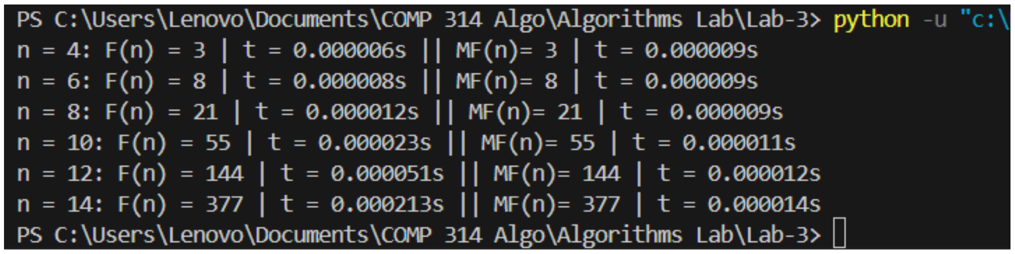
Fibonacci numbers: Memoization

Memoization in the Fibonacci sequence is a technique that stores the results of function calls to avoid redundant calculations. This speeds up the process of computing Fibonacci numbers.

```
def Fib(n, memo):
    if memo[n] != null:
        return memo[n]
    elif n==0:
        result = 0
    elif n==1:
        result = 1
    else:
        result = Fib(n-1, memo) + Fib(n-2, memo)
    memo[n] = result
    return result
```

Output:





```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> python -u "c:\
n = 4: F(n) = 3 | t = 0.000006s || MF(n)= 3 | t = 0.000009s
n = 6: F(n) = 8 | t = 0.000008s || MF(n)= 8 | t = 0.000009s
n = 8: F(n) = 21 | t = 0.000012s || MF(n)= 21 | t = 0.000009s
n = 10: F(n) = 55 | t = 0.000023s || MF(n)= 55 | t = 0.000011s
n = 12: F(n) = 144 | t = 0.000051s || MF(n)= 144 | t = 0.000012s
n = 14: F(n) = 377 | t = 0.000213s || MF(n)= 377 | t = 0.000014s
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

Analysis:

The time complexity of the recursive Fibonacci algorithm is $O(2^n)$, which is exponential. This is because the algorithm performs redundant calculations as n increases. We can see from the above graph clearly that the time complexity starts to increase exponentially for recursive algorithm. With memoization, the time complexity reduces to O(n) because each Fibonacci number is computed only once and stored for reuse. This eliminates redundant computations and ensures a linear traversal from F(0) and F(n), significantly improving performance.

Parallel Computing

Parallelism or parallel computing refers to the ability to carry out multiple calculations or the execution of processes simultaneously. Parallel computing is commonly used to solve computationally large and data-intensive tasks. Today parallelism is available in all computer systems, multicore chips are used in essentially all computing devices. At the larger scale, many computers can be connected by a network and used together to solve large problems. Parallel computing requires design of efficient parallel algorithms.

Parallel Quick Sort

In parallel quick sort, the independent subarrays are processed concurrently using multiple threads or processes.

Steps:

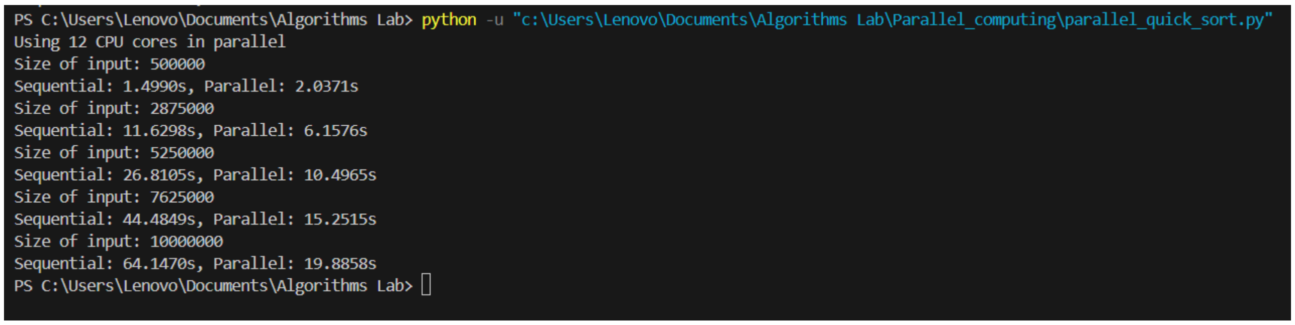
1. Start p processes which will partition the list and sort it using selected pivot element.
2. p processes will work on all partitions from the start of the algorithm till the list is sorted.
3. Each processes finds a pivot and partitions the list based on selected pivot.
4. Finally the list is merged forming a sorted list.

Implementation Considerations:

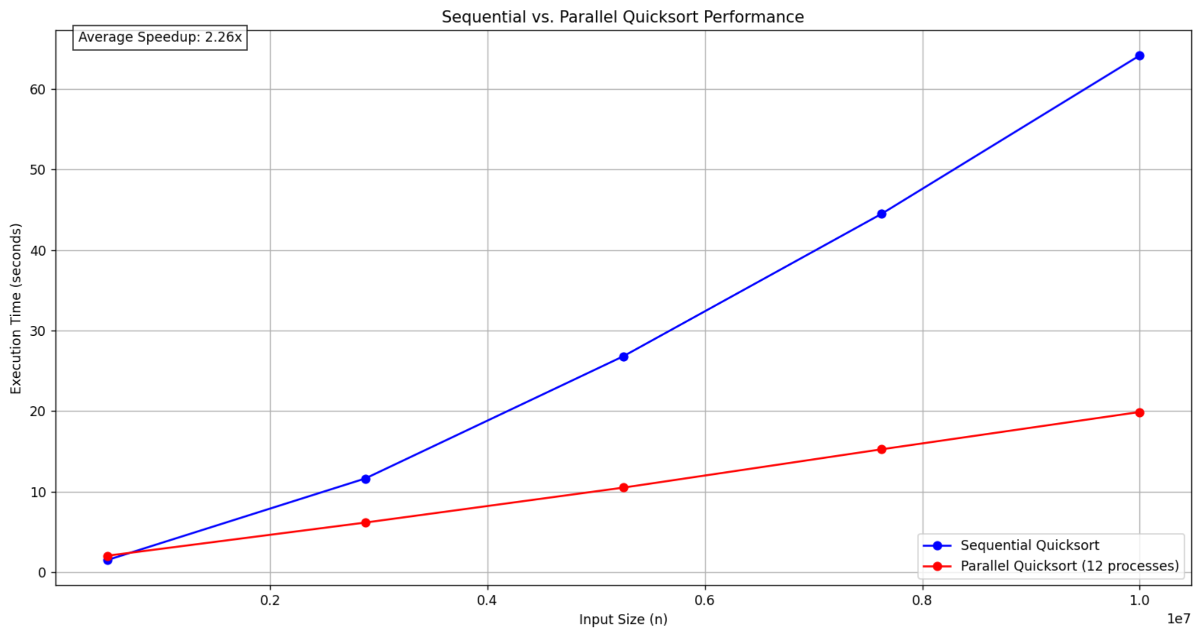
- **Thread/Process Management:** Efficiently manage thread or process creation to avoid excessive overhead. In below code, 12 process are created to handle sorting.

- **Divide Until Threshold:** Only parallelize until subarrays reach a certain size to avoid overhead dominating the benefits. (if input size is <100000, sequential quicksort is used)

Output:



```
PS C:\Users\Lenovo\Documents\Algorithms Lab> python -u "c:\Users\Lenovo\Documents\Algorithms Lab\Parallel_computing\parallel_quick_sort.py"
Using 12 CPU cores in parallel
Size of input: 500000
Sequential: 1.4990s, Parallel: 2.0371s
Size of input: 2875000
Sequential: 11.6298s, Parallel: 6.1576s
Size of input: 5250000
Sequential: 26.8105s, Parallel: 10.4965s
Size of input: 7625000
Sequential: 44.4849s, Parallel: 15.2515s
Size of input: 10000000
Sequential: 64.1470s, Parallel: 19.8858s
PS C:\Users\Lenovo\Documents\Algorithms Lab>
```



Analysis:

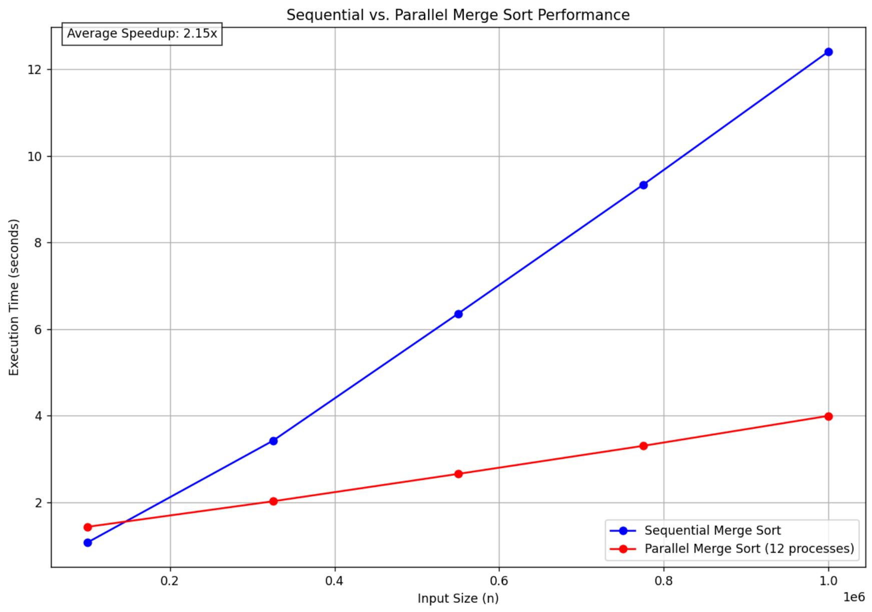
Here, we can clearly see that for 500,000 input size, the sequential quicksort is performing better than the parallel quicksort. But as the input size increases, parallel quicksort outperforms the sequential quicksort. This graph shows that parallel quicksort is much better when the input size is huge. In above approach, p number of processes work on n number of elements of a list, each p process works on n/p sub-list elements for (logn) steps. Total time complexity is O((n/p)logn).

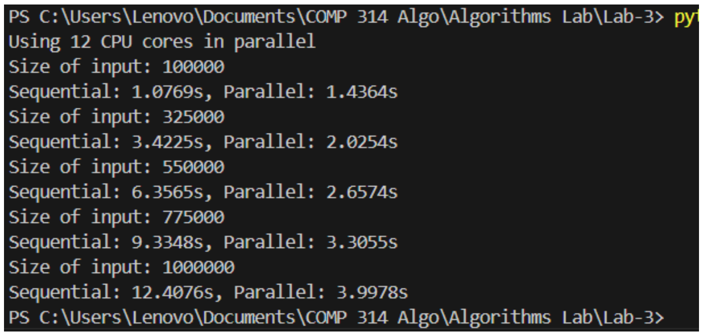
Parallel Merge Sort:

Steps:

1. Divide: Split the array into two roughly equal halves.
2. Conquer (in parallel): Sort each half concurrently using parallel processes.
3. Combine: Merge the sorted halves into a single sorted array.

Output:





```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> pyt
Using 12 CPU cores in parallel
Size of input: 100000
Sequential: 1.0769s, Parallel: 1.4364s
Size of input: 325000
Sequential: 3.4225s, Parallel: 2.0254s
Size of input: 550000
Sequential: 6.3565s, Parallel: 2.6574s
Size of input: 775000
Sequential: 9.3348s, Parallel: 3.3055s
Size of input: 1000000
Sequential: 12.4076s, Parallel: 3.9978s
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

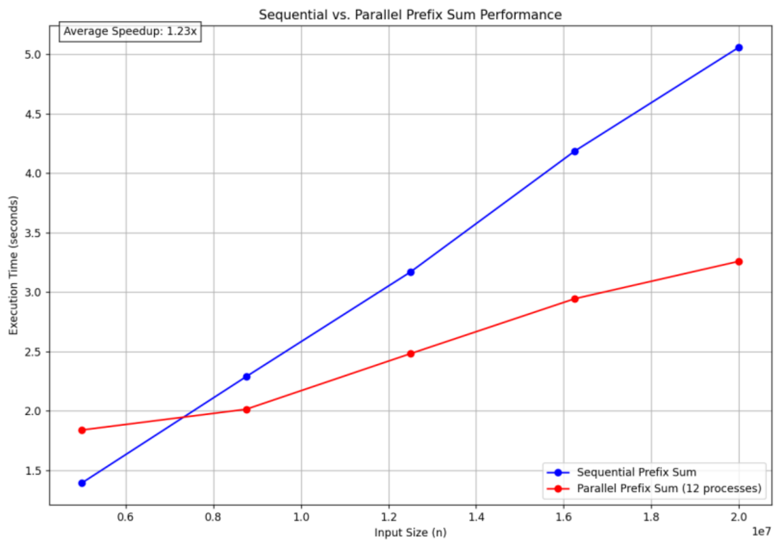
Analysis:

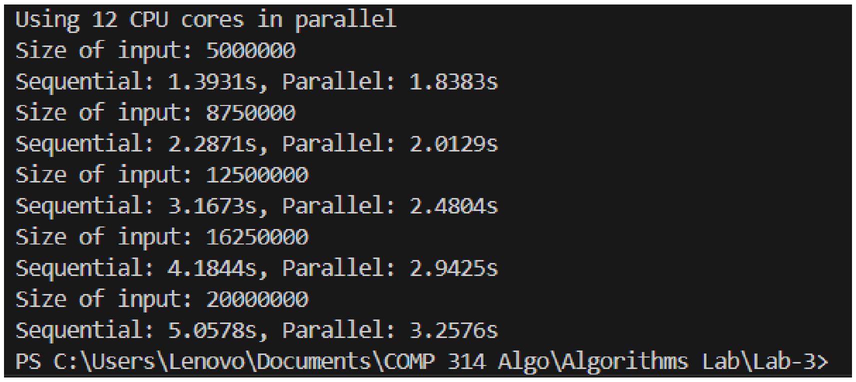
The speedup in parallel merge sort depends on the number of processes (p). In ideal case, the depth of recursion can be reduced to O(log(n/p)), but the total work across all processes still sums to O(nlogn). In the graph, we can see that sequential merge sort performs better for small input sizes but as the size increases, parallel merge sort outperforms sequential merge sort.

Parallel Prefix Sum:

A parallel prefix sum (also called a parallel scan) is a technique used to compute prefix sums efficiently in parallel computing environments. The goal of a prefix sum is to compute an array of partial sums where each element in the result array is the sum of all previous elements in the input array, including the element at that position.

Output:





```
Using 12 CPU cores in parallel
Size of input: 5000000
Sequential: 1.3931s, Parallel: 1.8383s
Size of input: 8750000
Sequential: 2.2871s, Parallel: 2.0129s
Size of input: 12500000
Sequential: 3.1673s, Parallel: 2.4804s
Size of input: 16250000
Sequential: 4.1844s, Parallel: 2.9425s
Size of input: 20000000
Sequential: 5.0578s, Parallel: 3.2576s
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

Analysis:

Sequential sum has time complexity of O(n). If we use p processes in parallel to compute sum of n numbers then, the time complexity is ideally reduced to O(n/p) for each process but the overall total time complexity is still O(n). We can see in the graph that sequential sum is better for input size of 5 million random numbers but after that the performance of parallel sum is better.

Probabilistic/Randomized Algorithms:

Probabilistic or randomized algorithms employ some form of random element in an attempt to obtain improved performance. Sometimes, improvement can be dramatic, from intractable to tractable. Some loss in reliability of results can occur. Different runs may produce different results for the same input, reliability may be improved by running several times.

Monte Carlo Algorithms:

These are randomized algorithms which may produce incorrect results with some small probability, but whose execution time is deterministic.

Primality Test:

Prime number is a natural number greater than 1 number whose only factors are 1 and itself. Primality test is an algorithm for determining whether an input number is prime.

Fermat Primality Test

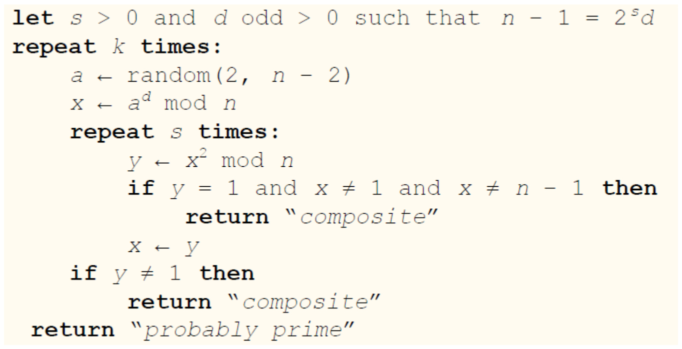
Fermat primality test is based on the Fermat's Little Theorem, which states that, if n is a prime number, then for any integer a, $1 < a < n-1$, $a^{n-1} \equiv 1 \pmod{n}$ [i.e., $a^{n-1} \% n = 1$].

Steps:

1. Repeat k times:
    a. Pick a randomly in the range $[2, n - 2]$
    b. If (a^{n-1} % n) != 1, then return composite
2. Return probably prime

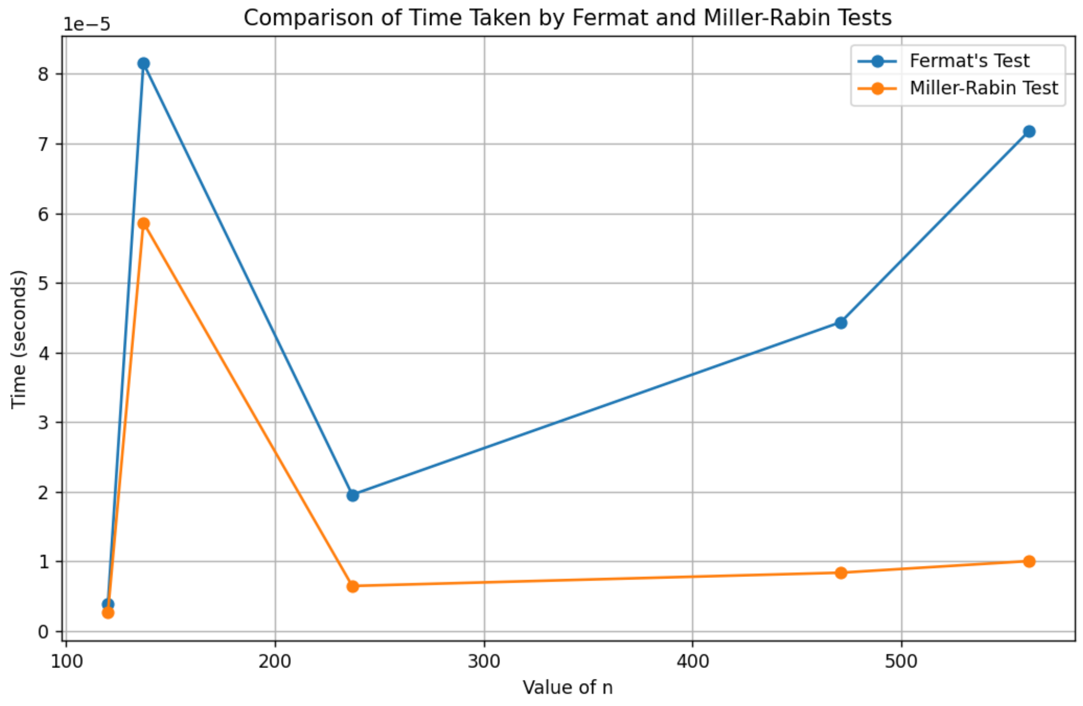
Miller-Rabin Primality Test

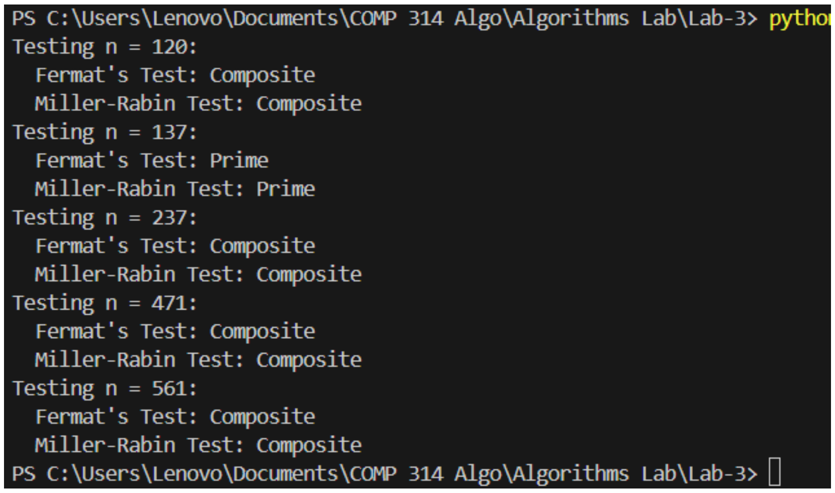
It is also based on the Fermat's Little Theorem. The result of this test will be whether the given number is a composite number or a probable prime number. The probable prime numbers in the Miller-Rabin Primality Test are called strong probable numbers, as they have a higher chance of being a prime number than in the Fermat's Primality Test.



```
let $s > 0$ and $d$ odd $> 0$ such that $n - 1 = 2^s d$
repeat $k$ times:
    $a \leftarrow$ random(2, $n - 2$)
    $x \leftarrow a^d$ mod $n$
    repeat $s$ times:
        $y \leftarrow x^2$ mod $n$
        if $y = 1$ and $x \neq 1$ and $x \neq n - 1$ then
            return "composite"
        $x \leftarrow y$
    if $y \neq 1$ then
        return "composite"
 return "probably prime"
```

Output:





```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> pytho
Testing n = 120:
  Fermat's Test: Composite
  Miller-Rabin Test: Composite
Testing n = 137:
  Fermat's Test: Prime
  Miller-Rabin Test: Prime
Testing n = 237:
  Fermat's Test: Composite
  Miller-Rabin Test: Composite
Testing n = 471:
  Fermat's Test: Composite
  Miller-Rabin Test: Composite
Testing n = 561:
  Fermat's Test: Composite
  Miller-Rabin Test: Composite
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

Analysis:

The value for k is taken as 20. From the above graph and output we see that the time taken to check if the number n is prime or not is lower for Miller-Rabin test. 137 is the only prime number provided which took the highest time to compute.

Las Vegas algorithms

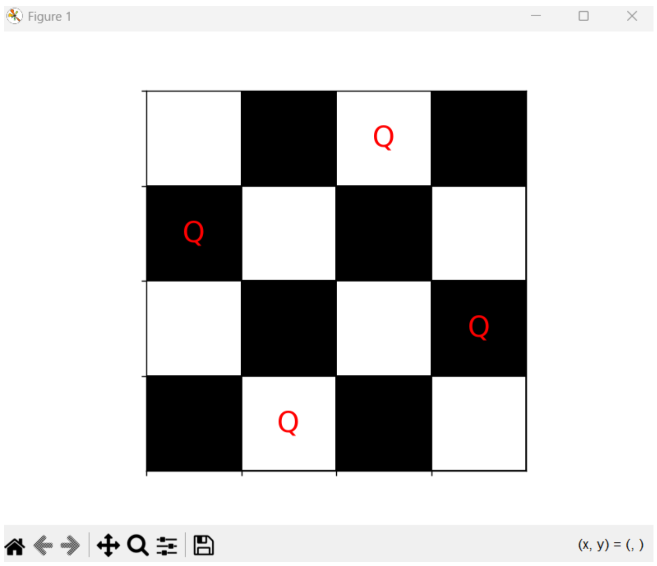
Las Vegas algorithms make probabilistic choices to help guide them more quickly to a correct solution. Unlike Monte Carlo algorithms, they never return a wrong answer.

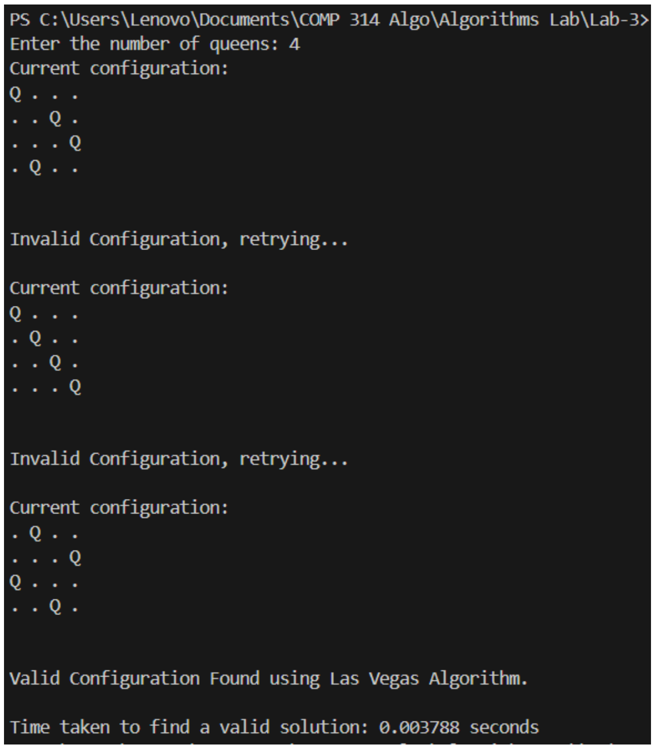
Two main categories are:

- Those which guarantee a correct solution though they may take longer if unfortunate choices are made.
- Those that may fail to produce a result, but if they return a result, it is always correct.

A Las Vegas approach – non-backtracking (N-Queens Problem)

- Place queens randomly on successive rows.
- No attempt is made to relocate previous queens when there is no possibility left for the next queen.
- The algorithm either ends successfully or fails if there is no square in which the next queen can be placed.
- The process can be repeated if a failure is detected, and will consider a probably different placement.





```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
Enter the number of queens: 4
Current configuration:
Q . . .
. . Q .
. . . Q
. Q . .


Invalid Configuration, retrying...

Current configuration:
Q . . .
. Q . .
. . Q .
. . . Q


Invalid Configuration, retrying...

Current configuration:
. Q . .
. . . Q
Q . . .
. . Q .


Valid Configuration Found using Las Vegas Algorithm.

Time taken to find a valid solution: 0.003788 seconds
```

Analysis:

The Las Vegas algorithm randomly generates configurations of queens on the board and checks if it's valid. If the configuration is valid, it visualizes and prints the solution along with the time taken to find it. If invalid, it retries by generating a new random configuration until a valid solution is found. Our solution for N = 4 is found in 3 random configurations.

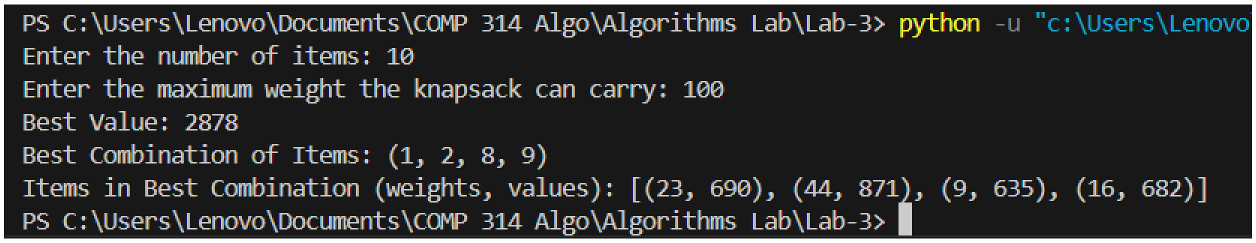
NP-Complete Problems:

0/1 Knapsack Problem:

We are given n objects and a knapsack or bag. Object i has a weight w_i and value p_i. The knapsack has a capacity m. If an object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

The 0/1 Knapsack problem can be solved using brute force by checking all possible subsets of items to find the one that maximizes the total value without exceeding the weight limit. Since there are 2^n possible subsets for n items, this approach has an exponential time complexity of $O(2^n)$, which makes it inefficient for large inputs.

Output:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> python -u "c:\Users\Lenovo
Enter the number of items: 10
Enter the maximum weight the knapsack can carry: 100
Best Value: 2878
Best Combination of Items: (1, 2, 8, 9)
Items in Best Combination (weights, values): [(23, 690), (44, 871), (9, 635), (16, 682)]
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

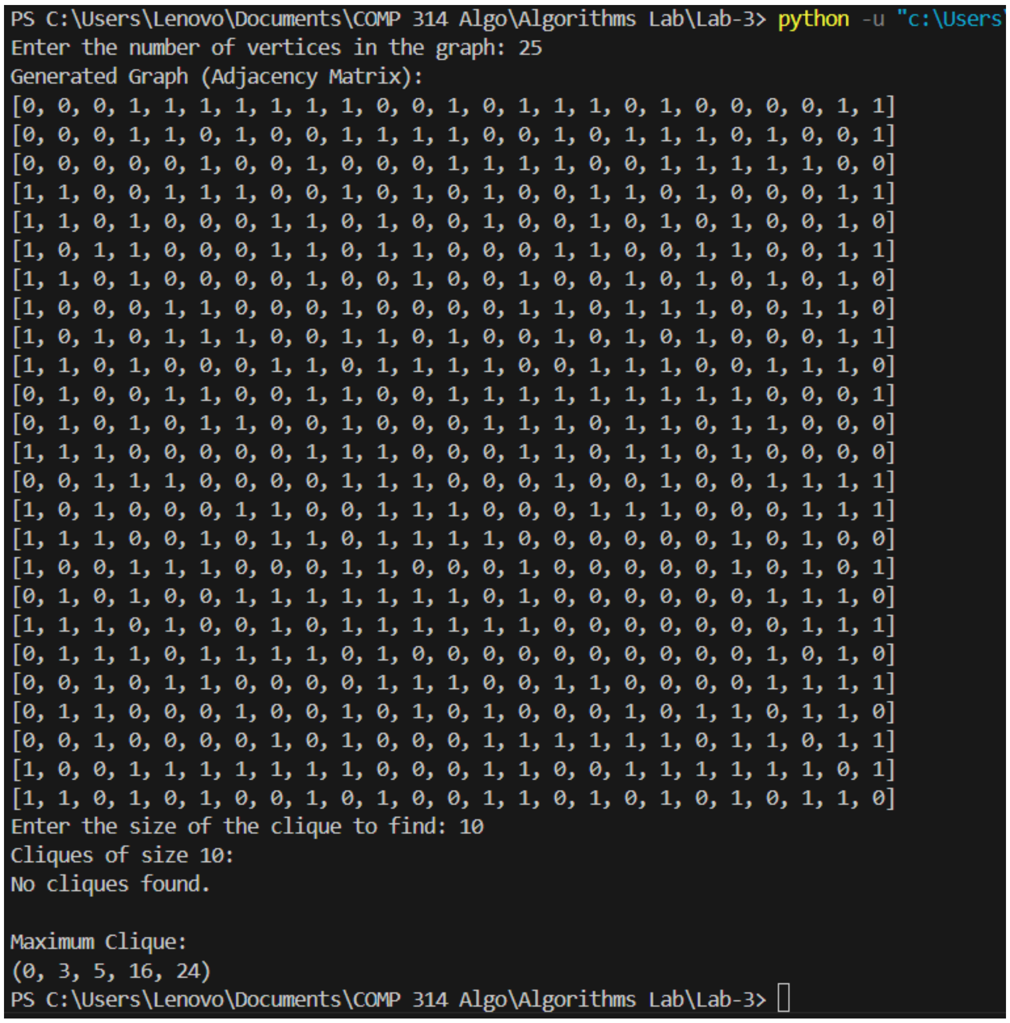
Analysis:

We have taken 10 items at random. The values are taken randomly at a range (50, 1000) and weights at range (5, 50). The maximum weight knapsack can carry is 100. The code gives the best value as 2878 where combination of items is (1, 2, 8, 9). While this approach works for small inputs, it is not practical for large problems because the number of subsets grows exponentially.

The Clique Problem:

A clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. CLIQUE = {(G, k): G is a graph containing a clique of size k}.

Output:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> python -u "c:\Users
Enter the number of vertices in the graph: 25
Generated Graph (Adjacency Matrix):
[0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1]
[0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0]
[1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1]
[1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0]
[1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
[1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0]
[1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0]
[1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1]
[1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0]
[0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1]
[0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0]
[1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1]
[1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1]
[1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0]
[1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1]
[0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0]
[1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
[0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
[0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0]
[0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1]
[1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1]
[1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0]
Enter the size of the clique to find: 10
Cliques of size 10:
No cliques found.

Maximum Clique:
(0, 3, 5, 16, 24)
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

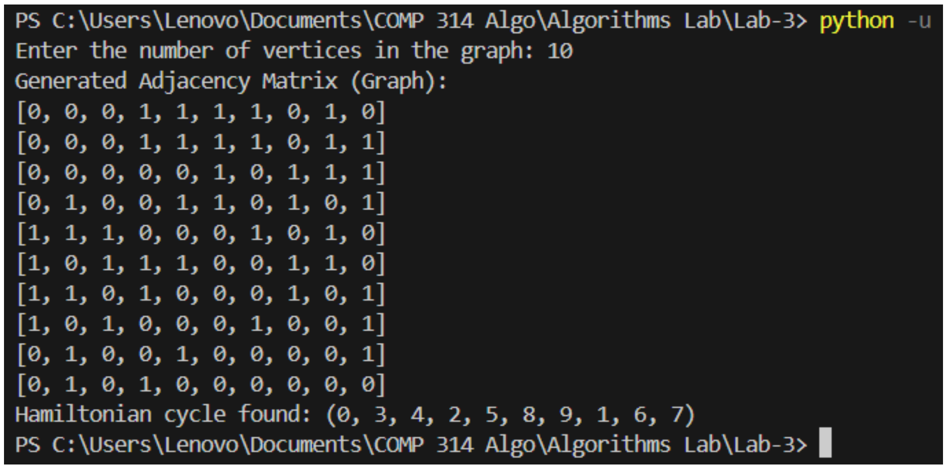
Analysis:

In the above output, the maximum size of clique is 5 which contains the vertices (0, 3, 5, 16, 24). It calculates the maximum clique size for smaller graphs like the above but struggles for huge graphs. For already generated graph, we asked if clique of size 10 exists or not and it returned no cliques found.

Hamiltonian 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. If graph contains a Hamiltonian cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian. 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. However, it can be solved for small or specific types of graphs.

Output:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> python -u
Enter the number of vertices in the graph: 10
Generated Adjacency Matrix (Graph):
[0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
[0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
[0, 0, 0, 0, 0, 1, 0, 1, 1, 1]
[0, 1, 0, 0, 1, 1, 0, 1, 0, 1]
[1, 1, 1, 0, 0, 0, 1, 0, 1, 0]
[1, 0, 1, 1, 1, 0, 0, 1, 1, 0]
[1, 1, 0, 1, 0, 0, 0, 1, 0, 1]
[1, 0, 1, 0, 0, 0, 1, 0, 0, 1]
[0, 1, 0, 0, 1, 0, 0, 0, 0, 1]
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0]
Hamiltonian cycle found: (0, 3, 4, 2, 5, 8, 9, 1, 6, 7)
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

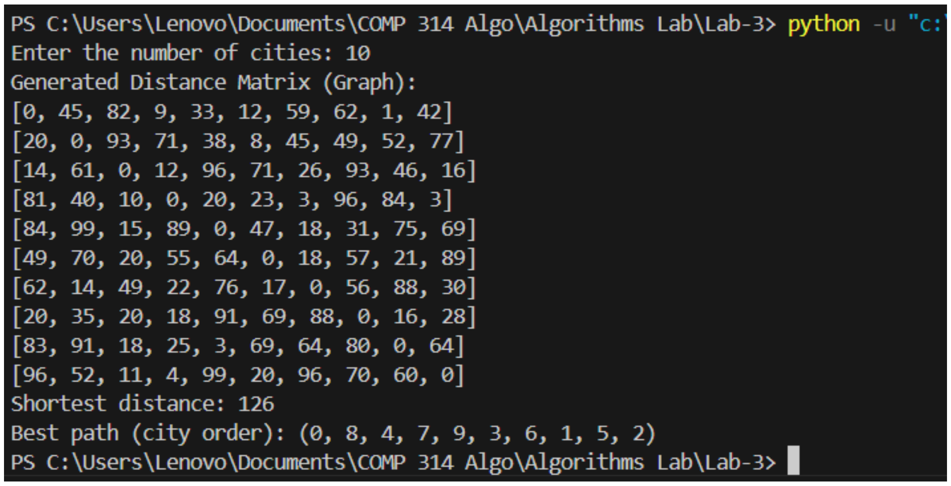
Analysis:

The brute-force method generates all possible permutations of the vertices and checks if each permutation forms a valid Hamiltonian cycle. This output validates that the graph contains a Hamiltonian cycle (0, 3, 4, 2, 5, 8, 9, 1, 6, 7). The brute-force approach used to find the Hamiltonian cycle is computationally very expensive, which is why it doesn't work efficiently for graphs with many vertices.

The Traveling-Salesman Problem:

We are given a complete undirected graph G = (V, E) that has a non-negative integer cost or distance associated with each edge (u, v) ∈ E, and we must find a Hamiltonian cycle of G with minimum cost or distance.

Output:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3> python -u "c:
Enter the number of cities: 10
Generated Distance Matrix (Graph):
[0, 45, 82, 9, 33, 12, 59, 62, 1, 42]
[20, 0, 93, 71, 38, 8, 45, 49, 52, 77]
[14, 61, 0, 12, 96, 71, 26, 93, 46, 16]
[81, 40, 10, 0, 20, 23, 3, 96, 84, 3]
[84, 99, 15, 89, 0, 47, 18, 31, 75, 69]
[49, 70, 20, 55, 64, 0, 18, 57, 21, 89]
[62, 14, 49, 22, 76, 17, 0, 56, 88, 30]
[20, 35, 20, 18, 91, 69, 88, 0, 16, 28]
[83, 91, 18, 25, 3, 69, 64, 80, 0, 64]
[96, 52, 11, 4, 99, 20, 96, 70, 60, 0]
Shortest distance: 126
Best path (city order): (0, 8, 4, 7, 9, 3, 6, 1, 5, 2)
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-3>
```

Analysis:

The output above is the result of applying the Traveling Salesman Problem using brute-force approach. The brute-force approach checks all possible permutations of city visits and calculates the total distance for each permutation. It then selects the permutation with the smallest total distance. The shortest distance that visits all cities exactly once and returns to the starting city is 126 and the cities are visited in the following order: 0, 8, 4, 7, 9, 3, 6, 1, 5, 2, 0.