# Final Submission for Assignment

GitHub Repository: [Insert your link here]

## Task 1: Maze Solver

The goal of this task is to solve a maze using two algorithms: Backtracking and Las Vegas.

The maze is loaded and displayed using Matplotlib, and the user can choose which algorithm to run.

The Backtracking algorithm uses a systematic approach to find the exit by trying every possible direction

and backtracking when a dead-end is encountered. The Las Vegas algorithm follows a random walk,

limited to 400 steps. Success is achieved if the exit is found within the step limit.

We compare the algorithms by running them 10,000 times and calculating the success rate for each.

The Backtracking algorithm uses a stack-based data structure, while Las Vegas records visited nodes using a set.

Below is the code for both algorithms:

```
import numpy as np
import matplotlib.pyplot as plt
import random

def load_maze(image_path):
    maze = plt.imread(image_path)
    plt.imshow(maze)
    plt.show()
    return maze

def backtracking_solver(maze, start, end):
    def backtrack(x, y, visited):
```

```
        if (x, y) == end:
            return True
        visited[x][y] = True
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and not visited[nx][ny]:
                if backtrack(nx, ny, visited):
                    return True
        return False

def las_vegas_solver(maze, start, end):
    x, y = start
    steps = 0
    visited = set()
    while steps < 400:
        direction = random.choice([(0, 1), (0, -1), (1, 0), (-1, 0)])
        nx, ny = x + direction[0], y + direction[1]
        if (nx, ny) == end:
            return True
        if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and (nx, ny) not in visited:
            x, y = nx, ny
            visited.add((x, y))
            steps += 1
    return False
```

## Task 2: Minimum Spanning Tree

For this task, we implemented Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a graph.

Kruskal's algorithm selects the edges in increasing order of weights and ensures no cycles are formed.

The process is depicted step by step, starting with the original graph and ending with the final MST.

Here is the code for the Kruskal's algorithm implementation:

```
import networkx as nx
import matplotlib.pyplot as plt

def plot_graph(graph):
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, with_labels=True)
    labels = nx.get_edge_attributes(graph, 'weight')
    nx.draw_networkx_edge_labels(graph, pos, edge_labels=labels)
    plt.show()

def kruskal_mst(graph):
    mst = nx.Graph()
    edges = sorted(graph.edges(data=True), key=lambda x: x[2]['weight'])
    for edge in edges:
        u, v, weight = edge
        if not nx.has_path(mst, u, v):
            mst.add_edge(u, v, weight=weight['weight'])
    return mst
```

## Task 3: Quicksort Algorithm

This task required the implementation of an in-place quicksort algorithm that sorts words alphabetically,

with the pivot being the middle element. Below is the Python code implementing the algorithm.

```
def quicksort(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 quicksort(left) + middle + quicksort(right)
```

## Task 4: Palindrome Check

A recursive function is used to check if a word is a palindrome. The function `isPalindrome()` checks

if the word is the same when read forwards and backwards. Below is the Python code:

```
def isPalindrome(word):
    if len(word) <= 1:
        return True
    if word[0] != word[-1]:
        return False
    return isPalindrome(word[1:-1])
```

## Task 5: Needleman-Wunsch Algorithm

This task required solving the string alignment problem using the Needleman-Wunsch algorithm for the

strings CATGATAA and ATACATA. The alignment matrix, traceback, and final alignment are computed manually.

No code is required for this question.