# 1)A* Algorithm:

## Code

```python
import numpy as np

class Node:

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

# This function return the path of the search


def return_path(current_node, maze):
    path = []
    no_rows, no_columns = np.shape(maze)
    # here we create the initialized result maze with -1 in every position
    result = [[-1 for i in range(no_columns)] for j in range(no_rows)]
    current = current_node
    while current is not None:
        path.append(current.position)
        current = current.parent
    # Return reversed path as we need to show from start to end path
    path = path[::-1]
    start_value = 0
    # we update the path of start to end found by A-star serch with every step incremented by 1
    for i in range(len(path)):
        result[path[i][0]][path[i][1]] = start_value
        start_value += 1
    return result


def search(maze, cost, start, end):
    # Create start and end node with initized values for g, h and f
    start_node = Node(None, tuple(start))
    start_node.g = start_node.h = start_node.f = 0
```

```
    end_node = Node(None, tuple(end))
    end_node.g = end_node.h = end_node.f = 0
    yet_to_visit_list = []
    visited_list = []

    # Add the start node
    yet_to_visit_list.append(start_node)

    # Adding a stop condition. This is to avoid any infinite loop and stop
    # execution after some reasonable number of steps
    outer_iterations = 0
    max_iterations = (len(maze) // 2) ** 10
    #(4 movements) from every positon
    move = [[-1, 0],  # go up
            [0, -1],  # go left
            [1, 0],  # go down
            [0, 1]]  # go right
    no_rows, no_columns = np.shape(maze)

    while len(yet_to_visit_list) > 0:

        outer_iterations += 1
        current_node = yet_to_visit_list[0]

        current_index = 0
        for index, item in enumerate(yet_to_visit_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        if outer_iterations > max_iterations:
            print("giving up on pathfinding too many iterations")
            return return_path(current_node, maze)

        yet_to_visit_list.pop(current_index)
        visited_list.append(current_node)

        if current_node == end_node:
            return return_path(current_node, maze)

        children = []

        for new_position in move:
            node_position = (
                current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            if (node_position[0] > (no_rows - 1) or
                node_position[0] < 0 or
                node_position[1] > (no_columns - 1) or
                    node_position[1] < 0):
                continue

            if maze[node_position[0]][node_position[1]] != 0:
                continue

            new_node = Node(current_node, node_position)

            children.append(new_node)

        # Loop through children
        for child in children:
            if len([visited_child for visited_child in visited_list if visited_child == child]) > 0:
                continue

            # Create the f, g, and h values
            child.g = current_node.g + cost
            # Heuristic costs calculated here, this is using eucledian distance
```

```
            child.h = (((child.position[0] - end_node.position[0]) ** 2) +
                       ((child.position[1] - end_node.position[1]) ** 2))

            child.f = child.g + child.h

            # Child is already in the yet_to_visit list and g cost is already lower
            if len([i for i in yet_to_visit_list if child == i and child.g > i.g]) > 0:
                continue

            # Add the child to the yet_to_visit list
            yet_to_visit_list.append(child)


if __name__ == '__main__':

    maze = [[0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0],
            [0, 1, 0, 1, 0, 0],
            [0, 1, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0]]

    print("The maze is ('1' represents hinderence), \n")
    print('\n'.join([' |'.join(["{:" ">3d}".format(item) for item in row])
                     for row in maze]))
    print("_" * 30, end='\n\n')
    start = [0, 0]  # starting position
    end = [4, 5]  # ending position
    cost = 1  # cost per movement

    path = search(maze, cost, start, end)
    print("The path is ('-1' represents no path),\n")
    print('\n'.join([' |'.join(["{:" ">3d}".format(item) for item in row])
                     for row in path]))

```

## Output

```
The maze is ('1' represents hinderence),

  0 |  1 |  0 |  0 |  0 |  0
  0 |  0 |  0 |  0 |  0 |  0
  0 |  1 |  0 |  1 |  0 |  0
  0 |  1 |  0 |  0 |  1 |  0
  0 |  0 |  0 |  0 |  1 |  0
------------------------------

The path is ('-1' represents no path),

  0 | -1 | -1 | -1 | -1 | -1
  1 |  2 |  3 |  4 |  5 | -1
 -1 | -1 | -1 | -1 |  6 |  7
 -1 | -1 | -1 | -1 | -1 |  8
 -1 | -1 | -1 | -1 | -1 |  9
PS C:\Users\Prashanth>
```

# 2)Randomized Quick-Sort Algorithm:

## Code

```python
from random import randint

def inPlaceQuickSort(A, start, end):
    if start < end:
        pivot = randint(start, end)
        temp = A[end]
        A[end] = A[pivot]
        A[pivot] = temp

        p = inPlacePartition(A, start, end)
        inPlaceQuickSort(A, start, p - 1)
        inPlaceQuickSort(A, p + 1, end)

def inPlacePartition(A, start, end):
    pivot = randint(start, end)
    temp = A[end]
    A[end] = A[pivot]
    A[pivot] = temp
    newPivotIndex = start - 1
    for index in range(start, end):
        if A[index] < A[end]:  # check if current val is less than pivot value
            newPivotIndex = newPivotIndex + 1

            temp = A[newPivotIndex]
            A[newPivotIndex] = A[index]
            A[index] = temp
    temp = A[newPivotIndex + 1]
    A[newPivotIndex + 1] = A[end]
    A[end] = temp
    return newPivotIndex + 1

X = [4, 5, 7, 4, 3, 6, 0, 4, 22, 45, 82]
print(f"Before sorting X = {X}")
inPlaceQuickSort(X, 0, len(X) - 1)
print(f"After sorting X = {X}")
```

## Output

```
PS C:\Users\Prashanth> python -u "c:\Users\Prashanth\Desktop\randomized_quicksort.py"
Before sorting X = [4, 5, 7, 4, 3, 6, 0, 4, 22, 45, 82]
After sorting X = [0, 3, 4, 4, 4, 5, 6, 7, 22, 45, 82]
PS C:\Users\Prashanth>
```