Artificial Intelligence
Assignment 1
Nitish Bhardwaj (B21AI056)

PART- A:

1. Sorting using BFS:
    a. **Algorithm**: The code uses BFS to sort an input array by swapping adjacent elements.
    b. **Data Structures**: It employs a queue (q), a path list, and a parent_map dictionary.
    c. **BFS Implementation**:
        i. Starts with the initial array in the queue.
        ii. Iteratively explores states (arrays) and generates child states by swapping elements.
        iii. Tracks visited states and parent-child relationships.
        iv. Continues until the sorting goal (sorted array) is reached.
    d. **Path Retrieval**:
        i. Extends the path with unexplored states in the queue.
        ii. Reconstructs the sorting path using the parent_map.
    e. **Output**: Returns original array, sorted array, sorting path, and nodes explored.
    f. **Printing Results**: The code prints the start state, goal state, sorting path, and nodes explored.
    g. **Purpose**: Demonstrates how BFS systematically explores states to achieve a goal (sorting an array).

```
Start State [4.0, 6.3, 9.0, -3.0]
Goal State [-3.0, 4.0, 6.3, 9.0]
Path taken from start state to goal state [(4.0, 6.3, 9.0, -3.0), (4.0, 6.3, -3.0, 9.0), (4.0, -3.0, 6.3, 9.0), (-3.0, 4.0, 6.3, 9.0)]
Number of nodes explored 9
```

2. Sorting using DFS:
    a. **Algorithm**: The code uses Depth-First Search (DFS) to sort an input array by swapping adjacent elements.
    b. **DFS Implementation**:
        i. Begins with the initial array.
        ii. Recursively explores states (arrays) and generates child states by swapping elements.
        iii. Maintains a visited list to track visited states.
        iv. Stops when the sorting goal (a sorted array) is achieved or all possible states are explored.
    c. **Path Retrieval**:

i. Reconstructs the sorting path from the initial state to the sorted state.
ii. Uses a parent_map dictionary to track parent-child relationships.

d. **Output**: Returns the original array, the sorted array, the sorting path, and the number of nodes (states) explored during DFS.
e. **Printing Results**: The code prints:
    i. The start state (original array).
    ii. The goal state (sorted array).
    iii. The path taken from the start state to the goal state.
    iv. The number of nodes explored during DFS traversal.
f. **Purpose**: Demonstrates how DFS explores states in a depth-first manner to achieve a goal (sorting an array).

```
Start State [4.0, 6.3, 9.0, -3.0]
Goal State [-3.0, 4.0, 6.3, 9.0]
Path taken from start state to goal state [(4.0, 6.3, 9.0, -3.0), (6.3, 4.0, 9.0, -3.0), (6.3, 9.0, 4.0, -3.0), (9.0, 6.3, 4.0, -3.0), (9.0, 4.
Number of nodes explored 14
```

3. Sorting using Iterative Deepening
    a. **Algorithm**: The code implements the IDDFS algorithm to sort an input array by swapping adjacent elements.
    b. **IDDFS Implementation**:
        i. Starts with a depth of 1 and iteratively increases the depth limit until the sorting goal is achieved.
        ii. Utilizes a depth-limited depth-first search (DFS) approach within each iteration.
        iii. Maintains a visited set to track visited states and a stack for DFS traversal.
        iv. DFS Implementation:
            1. Begins with the initial array and adds it to the stack.
            2. Continues DFS traversal until the current depth limit is reached.
            3. Explores child states by swapping adjacent elements.
            4. Keeps track of the explored path, the current node, and the depth of the current search.
            5. Stops when the sorting goal (a sorted array) is achieved or all states within the depth limit are explored.
    c. **Output**: Returns the original array, the sorted array, the sorting path, the number of nodes (states) explored during DFS, and the depth of the search.
    d. **Printing Results**: The code prints:
        i. The start state (original array).

ii. The goal state (sorted array).
iii. The path taken from the start state to the goal state.
iv. The number of nodes explored during DFS traversal.
v. The depth of the search.

e. **Purpose**: Demonstrates how the IDDFS algorithm explores states at increasing depths to achieve a goal (sorting an array) while ensuring a controlled exploration depth.

```
Start State [4.0, 6.3, 9.0, -3.0]
Goal State [-3.0, 4.0, 6.3, 9.0]
Path taken from start state to goal state [[4.0, 6.3, 9.0, -3.0], [4.0, 6.3, -3.0, 9.0], [4.0, -3.0, 6.3, 9.0], [-3.0, 4.0, 6.3, 9.0]]
Number of nodes explored 18
The depth is 4
```

4. Sorting using UCS:
    a. **Algorithm**: The code implements the Uniform Cost Search (UCS) algorithm to sort an input array by swapping adjacent elements.
    b. **UCS Implementation**:
        i. Utilizes a priority queue (PriorityQueue) to manage states (arrays) based on their path cost.
        ii. Maintains a visited set to track visited states.
    c. **Priority Queue**: The priority queue ensures that states with lower path costs are explored first, making UCS optimal.
    d. **UCS Process**:
        i. Starts with the initial array and its path cost of 0.
        ii. Continues to explore states with the lowest path cost.
        iii. Stops when the sorting goal (a sorted array) is achieved.
    e. **Output**: Returns the original array, the sorted array, the sorting path, the number of nodes (states) explored, and the cost of the solution.
    f. **Printing Results**: The code prints:
        i. The start state (original array).
        ii. The goal state (sorted array).
        iii. The path taken from the start state to the goal state.
        iv. The number of nodes explored during UCS traversal.
        v. The cost of the solution (total path cost).
    g. **Purpose**: Demonstrates how UCS systematically explores states with the lowest path cost to find an optimal solution (sorting an array).
    Note: For the task of sorting, all edge costs are the same.

```
Start State [4.0, 6.3, 9.0, -3.0]
Goal State [-3.0, 4.0, 6.3, 9.0]
Path taken from start state to goal state [[4.0, 6.3, 9.0, -3.0], [4.0, 6.3, -3.0, 9.0], [4.0, -3.0, 6.3, 9.0], [-3.0, 4.0, 6.3, 9.0]]
Number of nodes explored 7
The cost is 3
```

5. Sorting using Greedy Search:
    a. **Algorithm**: The code implements the Greedy Search (Best First Search) algorithm to sort an input array by swapping adjacent elements.
    b. **Greedy Search Implementation**:
        i. Utilizes a priority queue (priority_queue) to manage states (arrays) based on their heuristic values.
        ii. Keeps track of visited states and an exploration path.
    c. **Priority Queue**: The priority queue ensures that states with lower heuristic values (closer to the goal) are explored first.
    d. **Heuristic Function**: It employs a custom heuristic function that counts the number of out-of-place elements in the array, providing an estimate of the distance from the current state to the goal state.
    e. Greedy Search Process:
        i. Begins with the initial state and its heuristic value.
        ii. Continues to explore states with the lowest heuristic values.
        iii. Stops when the sorting goal (a sorted array) is achieved.
    f. **Output**: Returns the original array, the sorted array, the sorting path, and the number of nodes (states) explored.
    g. **Printing Results**: The code prints:
        i. The start state (original array).
        ii. The goal state (sorted array).
        iii. The path taken from the start state to the goal state.
        iv. The number of nodes explored during Greedy Search.
    h. **Purpose**: Demonstrates how Greedy Search uses a custom heuristic function to estimate the "closeness" to the solution, guiding the search efficiently towards the goal state. It's suitable for problems where a heuristic can provide a meaningful estimate of the distance to the solution.

```
Start State [4, 6.3, 9, -3]
Goal State [-3, 4, 6.3, 9]
Path taken from start state to goal state [[4, 6.3, 9, -3], [4, 6.3, -3, 9], [4, -3, 6.3, 9], [-3, 4, 6.3, 9]]
Number of nodes explored 4
```

6. Sorting using A* Search:
    a. **Algorithm**: The code implements the A* Search algorithm to sort an input array by swapping adjacent elements. It uses a heuristic function to guide the search efficiently.
    b. A* Search Implementation:
        i. Utilizes a priority queue (pqueue) to manage states (arrays) based on their estimated cost (fcost) which combines the cumulative cost and the heuristic value.

ii. Keeps track of visited states, an exploration path, and the total cost of each state.

c. **Priority Queue**: The priority queue ensures that states with lower estimated costs (fcost) are explored first, considering both the current cumulative cost and the heuristic value.

d. **Heuristic Function**: It employs a heuristic function (heuristic) to estimate the distance or cost from the current state to the goal state, which is added to the cumulative cost to calculate fcost.

e. **A* Search Process**:
   i. Begins with the initial state and its cumulative cost and fcost.
   ii. Continues to explore states with the lowest estimated costs (fcost=h_cost + g_cost).
   iii. Stops when the sorting goal (a sorted array) is achieved.

f. **Output**: Returns the original array, the sorted array, the sorting path, and the number of nodes (states) explored.

g. **Printing Results**: The code prints:
   i. The start state (original array).
   ii. The goal state (sorted array).
   iii. The path taken from the start state to the goal state.
   iv. The number of nodes explored during A* Search.

h. **Purpose**: Demonstrates how A* Search uses a heuristic function to estimate the "closeness" to the solution and combines it with the cumulative cost to prioritize states for exploration. It is particularly useful for problems where both cost and heuristic information are available to guide the search efficiently.

```
Start State [4, 6.3, 9, -3]
Goal State [-3, 4, 6.3, 9]
Path taken from start state to goal state [[4, 6.3, 9, -3], [4, 6.3, -3, 9], [4, -3, 6.3, 9], [-3, 4, 6.3, 9]]
Number of nodes explored 4
```

7. Sorting using Hill Climbing:

a. **Algorithm**: The code implements the Hill Climbing algorithm with a heuristic function to sort an input array by swapping adjacent elements.

b. **Hill Climbing Implementation**:
   i. Utilizes a recursive function rec_hc for the Hill Climbing search.
   ii. Keeps track of visited states, explored nodes, and the current state.

c. Recursive Hill Climbing:
   i. The rec_hc function recursively explores the state space by generating and evaluating neighboring states.

ii. It evaluates each neighboring state's heuristic value, and if a better state is found, it progresses to that state.
iii. The process continues until a local maximum (the best state) is reached or a goal state (sorted array) is found.

d. **Output**: Returns the goal state (sorted array) and the number of nodes (states) explored during the search.
e. **Printing Results**: The code prints:
   i. The start state (original array).
   ii. The goal state (sorted array).
   iii. The number of nodes explored during the Hill Climbing search.
f. **Purpose**: Demonstrates how Hill Climbing, guided by a heuristic function, attempts to reach the best state (local maximum) in the state space. It may not always find the global optimum but efficiently converges to local maxima. This approach is suitable for sorting problems where a heuristic can assess the quality of the current state.

```
Start State [4, 6.3, 9, -3]
Goal State [-3, 4, 6.3, 9]
Number of nodes explored 4
```

PART -B:

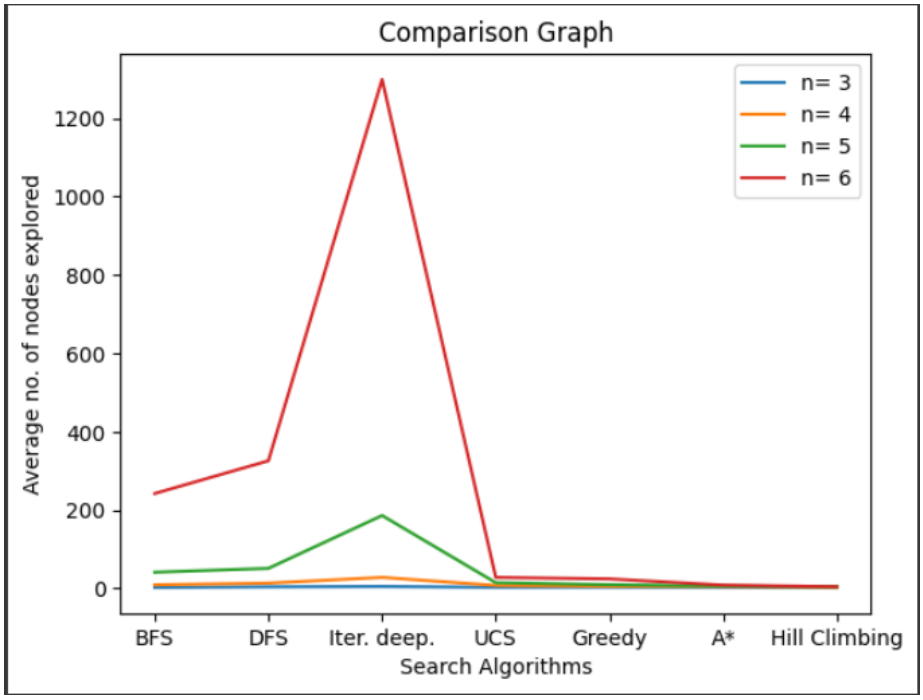
1. Run the search algorithms for at least 20 iterations with different number of elements in the array (n=3,4,5,6)

```
For n = 3
Average nodes explored using bfs : 2.1
Average nodes explored using dfs : 3.6
Average nodes explored using iterative deepening : 4.5
Average nodes explored using ucs : 2.4
Average nodes explored using greedy search : 2.8
Average nodes explored using A* search : 2.6
Average nodes explored using hill climbing search : 1.6
```

```
For n = 4
Average nodes explored using bfs : 8.8
Average nodes explored using dfs : 13.05
Average nodes explored using iterative deepening : 27.75
Average nodes explored using ucs : 7.2
Average nodes explored using greedy search : 4.9
Average nodes explored using A* search : 4.35
Average nodes explored using hill climbing search : 2.7
```

```
For n = 5
Average nodes explored using bfs : 40.9
Average nodes explored using dfs : 50.8
Average nodes explored using iterative deepening : 185.8
Average nodes explored using ucs : 14.2
Average nodes explored using greedy search : 9.1
Average nodes explored using A* search : 5.7
Average nodes explored using hill climbing search : 4.0
```

```
For n = 6
Average nodes explored using bfs : 241.95
Average nodes explored using dfs : 325.55
Average nodes explored using iterative deepening : 1299.65
Average nodes explored using ucs : 27.75
Average nodes explored using greedy search : 23.9
Average nodes explored using A* search : 8.1
Average nodes explored using hill climbing search : 4.0
```



Observation and Analysis: For this experiment, we observed that:

1. With increasing ‘n’, the number of nodes explored increases (drastically increases for some search algorithms).
2. Iterative deepening was the slowest.
3. A* search and hill climbing were the fastest.
4. Among the general search algorithms i.e. BFS and DFS, DFS comes out to be slower than BFS.