Artificial Intelligence
Assignment - 1

Name: Tanish Pagaria
Roll No.: B21AI040

Sorting as Searching

- **state:** a comma-separated list of numbers
- **successor function (action):** swap any two neighboring numbers
- **cost:** 1 unit per action
- **goal test / state:** a comma-separated list of numbers sorted in ascending order

Part - A

Code implementation details:

- Programming language used: **Python**
- Helper functions
  Certain helper functions were defined in order to aid in the process of finding next (children or neighboring) states, checking if an array is sorted or not, and checking if an array (state) is present inside an array of arrays (states).

- Breadth First Search (BFS)
  - Fringe data structure used: **stack**
  - A simple breadth first search algorithm was implemented, keeping in check if a state has already been visited or not

- Depth First Search (DFS)
  - Fringe data structure used: **queue**
  - Similar to BFS, a simple depth first search algorithm was implemented

- Iterative Deepening
  - Fringe data structure used: **stack**
  - A depth first search was applied, keeping in mind the depth covered, iteratively
  - The depth for the search was increased with each iteration, and the search was started again until the solution was found

- Uniform Cost Search (UCS)
  - Fringe data structure used: **priority queue** (
    priority-1 : minimum cumulative cost,
    priority-2: state array elements
)

- Greedy (Best First) Search
  - Heuristic function used: The count of all the elements on the right that are greater than the element at the current index was calculated for each index. The sum of these counts was used as the heuristic.
  - Fringe data structure: **priority queue** (priority : minimum heuristic value)

- A* Search
  - Same heuristic function as Greedy Search
  - Fringe data structure: **priority queue** (
    priority-1: minimum sum of heuristic value and cumulative cost,
    priority-2: state array elements
)

- Hill Climbing Search
    - Heuristic function used: The count of all elements that are less than the element at the index just before them was used as the heuristic.
    - The nearest neighbor or child was selected based on the best heuristic value
    - The algorithm would continue recursively for the selected node until none of the neighbors or children were better than the current one
    - The current state would be returned as the local optimum (final state)

Example run for the given example: `{4, 6.3, 9, -3}`

- **Breadth First Search**
    - Path: `[4, 6.3, 9, -3] => [6.3, 4, 9, -3] => [4, 9, 6.3, -3] => [4, 6.3, -3, 9] => [6.3, 9, 4, -3] => [6.3, 4, -3, 9] => [9, 4, 6.3, -3] => [4, 9, -3, 6.3] => [4, -3, 6.3, 9] => [9, 6.3, 4, -3] => [6.3, 9, -3, 4] => [6.3, -3, 4, 9] => [9, 4, -3, 6.3] => [4, -3, 9, 6.3] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 15

- **Depth First Search**
    - Path: `[4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, -3, 6.3, 9] => [4, -3, 9, 6.3] => [4, 9, -3, 6.3] => [4, 9, 6.3, -3] => [9, 4, 6.3, -3] => [9, 4, -3, 6.3] => [9, -3, 4, 6.3] => [9, -3, 6.3, 4] => [9, 6.3, -3, 4] => [9, 6.3, 4, -3] => [6.3, 9, 4, -3] => [6.3, 9, -3, 4] => [6.3, -3, 9, 4] => [6.3, -3, 4, 9] => [6.3, 4, -3, 9] => [6.3, 4, 9, -3] => [-3, 6.3, 4, 9] => [-3, 6.3, 9, 4] => [-3, 9, 6.3, 4] => [-3, 9, 4, 6.3] => [-3, 4, 9, 6.3] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 24

- **Iterative Deepening**
    - Path: `[4, 6.3, 9, -3] => (no solution, depth increased) => [4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, 9, 6.3, -3] => [6.3, 4, 9, -3] => (no solution, depth increased) => [4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, -3, 6.3, 9] => [6.3, 4, -3, 9] => [4, 9, 6.3, -3] => [4, 9, -3, 6.3] => [9, 4, 6.3, -3] => [6.3, 4, 9, -3] => [6.3, 9, 4, -3] => (no solution, depth increased) => [4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, -3, 6.3, 9] => [4, -3, 9, 6.3] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 19

- **Uniform Cost Search**
    - Path: `[4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, 9, 6.3, -3] => [6.3, 4, 9, -3] => [4, -3, 6.3, 9] => [4, 9, -3, 6.3] => [6.3, 4, -3, 9] => [6.3, 9, 4, -3] => [9, 4, 6.3, -3] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 10

- **Greedy Search (Best First Search)**
    - Path: `[4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, -3, 6.3, 9] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 4

- **A* Search**
    - Path: `[4, 6.3, 9, -3] => [4, 6.3, -3, 9] => [4, -3, 6.3, 9] => [-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 4

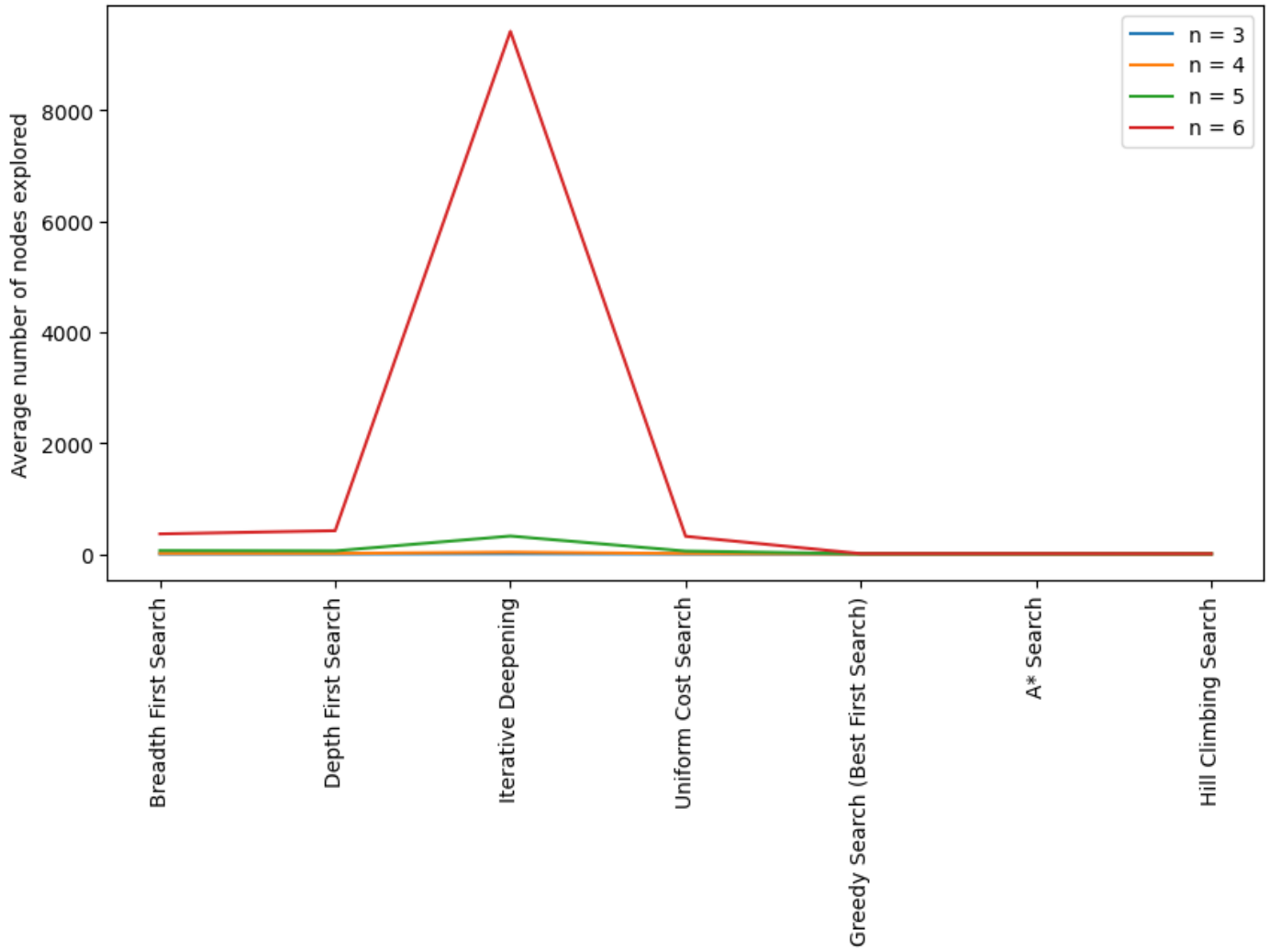
- **Hill Climbing Search**
    - Final State: `[-3, 4, 6.3, 9]`
    - No. of steps taken (nodes explored): 4

Part - B

All the algorithms were run 20 times for the following lengths of array: 3, 4, 5, and 6, containing random float values (up to decimal place = 1) in the range (-10, 10).

n → Algorithms ↓	n = 3	n = 4	n = 5	n = 6
Breadth First Search	3.4	11.9	63.55	363.7
Depth First Search	3.1	12.5	58.3	421.85
Iterative Deepening	7.1	34.9	325.05	9515.95
Uniform Cost Search	3.0	10.3	54.65	320.1
Greedy Search	2.4	3.9	5.9	8.45
A* Search	2.4	3.9	5.9	8.45
Hill Climbing Search	3.0	4.0	5.0	6.0

Table showing the average number of nodes explored by each algorithm for different values of n



Plot showing the average number of nodes visited for each algorithm for each value of n

Observations:

- Greedy Search and A* Search show the same results.
- Iterative Deepening takes the maximum number of steps for each value of n.
- Hill Climbing Search gives only the locally optimal solution based on the heuristic and the resultant final state may or may not be sorted.
- Greedy Search and A* Search perform better than Uniform Cost Search in case of Informed Search methods.

Analyses:

- I have tried to optimize the search by giving second priority to the arrays after giving first priority to cumulative cost in case of Uniform Cost Search, heuristic in case of Greedy Search, and (cumulative cost + heuristic) in case of A* Search, hoping for faster results in this manner since they would be closer to the sorted state.
- The cost for each action is 1. Therefore, the cumulative cost is same for each depth. Hence, Uniform Cost Search works similar to Breadth First Search, depth by depth.
  However, Uniform Cost Search works slightly better because of the priorities defined in the previous point.
- Using the logic mentioned in the previous point, the algorithms Greedy Search (Best First Search) and A* Search also give the same result. This happens because the fringe (priority queue) in both cases uses heuristic cost, with A* using (cumulative cost + heuristic cost).
- Hill Climbing Search does not ensure sorted array as the final state because, when going for the locally optimal state in a greedy approach, we always get a good solution close to the optimal state but not necessarily the optimal state.