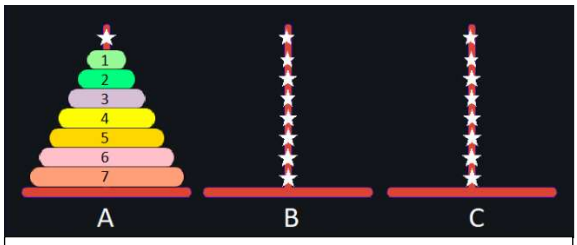
Definition of the Problem

1. State

```
typedef struct State
{
    char tower_matrix[9][3];
    int disk_num;
    float h_n;    // Heuristic function

}State;
```

Prototype of Tower Matrix

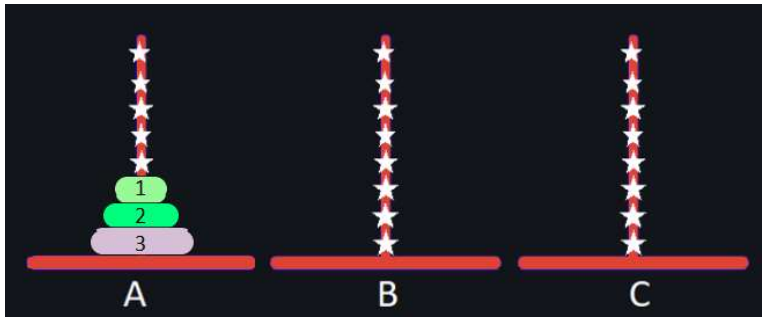


The asterisk (*) in the tower matrix represents spaces.

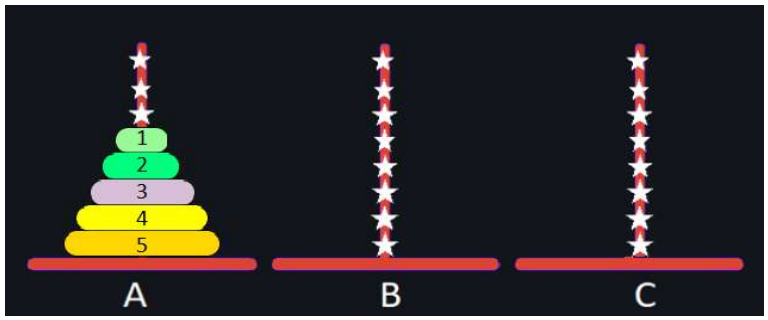
The letters (A,B,C) at the bottom of the tower matrix represent which tower we are in.

2. Initial State

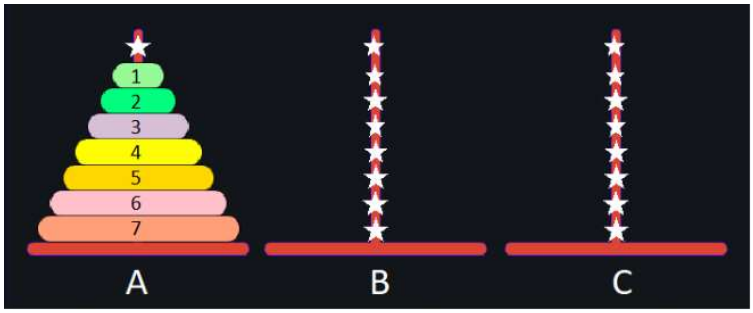
a) Three Disks



b) Five Disks



c) Seven Disks



3. Actions

```
enum ACTIONS // All possible actions
{
    TakeA_PutB, TakeA_PutC,
    TakeB_PutA, TakeB_PutC,
    TakeC_PutA, TakeC_PutB
};
```

- ✓ TakeA_PutB means take the disk at the top of tower A and put it in tower B.
- ✓ TakeA_PutC means take the disk at the top of tower A and put it in tower C.
- ✓ TakeB_PutA means take the disk at the top of tower B and put it in tower A.
- ✓ TakeB_PutC means take the disk at the top of tower B and put it in tower C.
- ✓ TakeC_PutA means take the disk at the top of tower C and put it in tower A.
- ✓ TakeC_PutB means take the disk at the top of tower C and put it in tower B.

4. Transition Model

```
// This struct is used to determine a new state in transition model
typedef struct Transition_Model
{
    State new_state;
    float step_cost;
}Transition_Model;
```

5. Node

```
typedef struct Node
{
    State state;
    float path_cost;
    enum ACTIONS action; //The action applied to the parent to generate this node
    struct Node *parent;
    int Number_of_Child; // required for depth-first search algorithms
}Node;
```

6. Queue

```
typedef struct Queue  // Used for frontier
{
    Node *node;
    struct Queue *next;
}Queue;
```

Definition of Heuristic Function

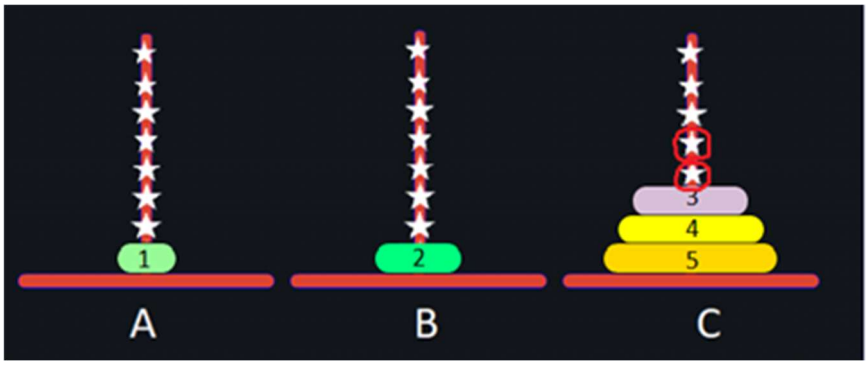
```
float Compute_Heuristic_Function(const State *const state, const State *const goal)
{
    int i = 7;
    int count = 0;
    while (ft_is_numeric(goal->tower_matrix[i][2]))
    {
        if (state->tower_matrix[i][2] != goal->tower_matrix[i][2])
            count++;
        i--;
    }

    return count;
}
```

✓ Returns the number of disks in the current state that differ in location from the disks in the goal state.

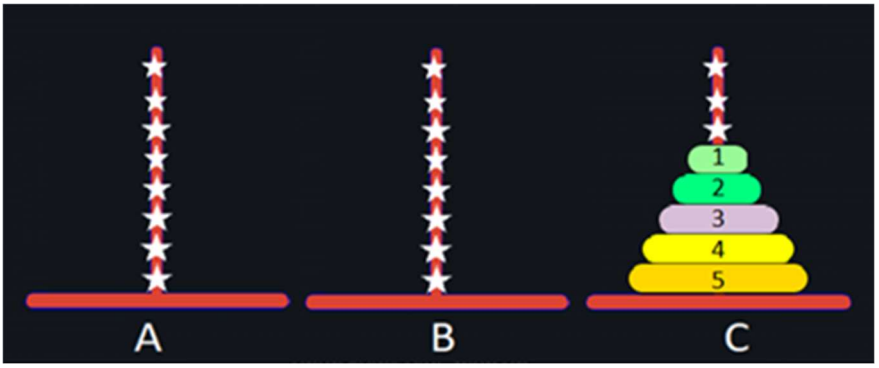
Example 2: In this example, our function will return the number 2.

Current State



Current State

Goal State



Goal State

Result of Several Simulations

a) For three disks using Breadth-First Search

```
The number of searched nodes is : 25

The number of generated nodes is : 53

The number of generated nodes in memory is : 53

THE COST PATH IS 7.00.

Process exited after 6.194 seconds with return value 0
```

b) For five disks using A* Search

```
The number of searched nodes is : 152

The number of generated nodes is : 453

The number of generated nodes in memory is : 453

THE COST PATH IS 36.00.

Process exited after 45.58 seconds with return value 0
```

c) For five disks and maximum level is 500 using Depth-Limited Search

```
The number of searched nodes is : 123

The number of generated nodes is : 242

The number of generated nodes in memory is : 123

THE COST PATH IS 81.00.

Process exited after 15.42 seconds with return value 0
```

d) For seven disks using Uniform-Cost Search

```
The number of searched nodes is : 2145

The number of generated nodes is : 6431

The number of generated nodes in memory is : 6431

THE COST PATH IS 127.00.

Process exited after 204.2 seconds with return value 0
```

For three disks:

	Breadth-first search	Uniform-Cost Search	Depth-First Search	Depth-Limited Search	Iterative Deepening Search	Greedy Search	A* Search
The number of searched nodes	25	25	15	15	71	16	14
The number of generated nodes	53	71	26	26	131	35	39
The number of generated nodes in memory	53	71	15	15	11	35	39
The cost path	7	7	9	9	7	7	8
Time (second)	3.53	5.036	3.201	4	6.876	6.873	3.193

- ✓ The depth-limited search algorithm generated and searched the same number of nodes, and had the lowest memory usage among all the algorithms tested. This indicates that it may be a good choice for situations where memory is a concern.
- ✓ The A* search algorithm generated fewer nodes than the uniform-cost search algorithm and the greedy search algorithm, but more nodes than the breadth-first search algorithm. This algorithm may be a good choice for situations where finding an optimal solution quickly is a priority.
- ✓ The greedy search algorithm generated and searched fewer nodes than the uniform-cost search and breadth-first search algorithms, but had higher memory usage. This algorithm may be a good choice for situations where memory is not a concern, but finding a solution quickly is a priority.
- ✓ The uniform-cost search algorithm had the highest time among all the algorithms tested. This suggests that this algorithm may not be the best choice for situations where time is a concern.
- ✓ The breadth-first search algorithm generated and searched fewer nodes than the uniform-cost search algorithm, but had higher memory usage. This suggests that this algorithm may not be the best choice for memory-intensive tasks, but may be useful in situations where finding a solution quickly is a priority.

For five disks:

	Breadth-first search	Uniform-Cost Search	Depth-First Search	Depth-Limited Search	Iterative Deepening Search	Greedy Search	A* Search
The number of searched nodes	233	233	123	123	3566	125	152
The number of generated nodes	617	695	242	242	9402	344	453
The number of generated nodes in memory	617	695	123	123	67	344	453
The cost path	31	31	81	81	45	31	36
Time (second)	16.22	17.77	16.09	18.07	202.12	11	14.44

- ✓ Greedy Search generated the least number of searched nodes. This is because greedy search algorithm selects the node that appears to be the closest to the goal, without considering the distance from the starting point. As a result, it may not explore all the possible paths, leading to fewer searched nodes.
- ✓ A* Search generated the most generated nodes. This is because the A* algorithm uses a heuristic function that estimates the distance from a node to the goal, and combines this with the actual cost of reaching that node. As a result, it generates more nodes than other algorithms that do not use heuristic functions.
- ✓ The cost path time was the longest for Iterative Deepening Search. This is because IDS algorithm performs a DFS search with a depth limit that increases iteratively, leading to a higher computational cost.
- ✓ Breadth-First Search generated the most nodes in memory. This is not surprising as BFS explores all the neighbors of a node before proceeding to the next level, thus requiring more memory to store all the nodes generated at each level.
- ✓ All variants of Depth-First Search generated the same number of nodes in memory. This is because DFS algorithm follows one path until it reaches the end or encounters a dead end. As a result, the algorithm does not require a lot of memory to store the nodes it generates.

For seven disks:

	Breadth-first search	Uniform-Cost Search	Depth-First Search	Depth-Limited Search	Iterative Deepening Search	Greedy Search	A* Search
The number of searched nodes	2145	2145	1092	1009	?	1098	1573
The number of generated nodes	6113	6431	2186	2372	?	3197	4716
The number of generated nodes in memory	6113	6431	1095	611	?	3197	4716
The cost path	127	127	729	407	?	127	148
Time (second)	158.1	170.1	356.1	263	2400+	85.11	147.4

Note: Each simulation was created using different