# DSA Assessment – 1

**Jagtap Mahesh Vilas**

**24MCS1017**

1. **You are given an ordered list of integers: [4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 58, 65, 80, 98]. The task is to locate the number 20 using the binary search algorithm. The image provided shows the steps involved in performing a binary search to find the element 20.**

**Analyze the process of the binary search algorithm as shown in the image and answer the following questions:**

a) **Describe each step involved in the binary search process to locate the element 20. How does the algorithm decide which part of the list to search next?**

**Answer**:

Step 1: Initialization:

The lower pointer is set to the beginning of the list (index 0).

The higher pointer is set to the end of the list (index 14).

Step 2: First Iteration:

Calculate the middle index: middle= (lower+higher)/2=(0+14)/2=7;

Check if the value of the middle element is equal to 20 or not.

Here arr [middle]=35, is not equal to 20.

Since 20 is less than 35, the algorithm decides to search the element in the left half of the array.

Update the higher pointer as, higher=middle-1 i.e. higher=7-1=6.

Step 3: Second iteration:

Calculate the new middle index: middle=(0+6)/2=3

The value at middle index (3) is 15.

Since 20 is greater than 15(middle element), the algo decides to search in the right half of the list.

Update the lower pointer to middle+1=3+1=4.

Step 4: Third iteration:

Calculate the new middle index: Middle=(4+6)/2=5.

The value at middle index (5) is 25.

Since 20 is less than 25, the algorithm decides to search the element in the left half of the array.

Update the higher pointer as, higher=middle-1 i.e. higher=5-1=4.

<u>Step 5: Fourth iteration:</u>

Calculate the new middle index: Middle= (4+4)/2=4.

The value at middle index (4) is 20.

Since 20 is equal to 20, the Search is successful and the target value is found at index 4 .

**b) Explain why binary search is more efficient than a linear search for locating an element in a sorted list. Provide an example of the number of comparisons needed in a worst-case scenario for both binary and linear searches in this list.**

**Answer:**

Binary search is more efficient than linear search for locating an element in a sorted list because in binary search we repeatedly divide the search space in half with each iteration, whereas in linear search we check and compare each and every element of the array.

In Linear search, in the worst case, if the target element is at the last position or not in the list, we will have to make n comparisons. Hence the worst case complexity of linear search is O (n)

While in Binary search, we are not comparing every element of the list, instead we are only checking the middle element iteratively by dividing the search space into half. Here in the worst case we will have to make log n comparisons i.e. O(log n).

<u>Example:</u>

To find an element using both linear search and binary search in the worst case scenario, let us assume that element is not present in the list.

Then in linear search we will have to compare every single element until the end of the list.So for 15 elements, total comparisons will be 15.

Whereas in Binary search we will have log n comparisons. i.e. ($\log_2 15$)=4.

**c) Suppose the list had an additional element, 23, inserted in the correct order, making the list [4, 5, 10, 15, 20, 23, 25, 30, 35, 40, 45, 50, 58, 65, 80, 98]. How would the binary search algorithm change to locate 20 in this new list? Would the steps change significantly, and why?**

**Answer:**

The binary search algorithm for locating the value 20 in this new list will proceed similarly to how it would have with the previous list, because the structure and principles of binary search remain the same. Here's how the steps will be carried out:

Initial Setup:

Target: 20 lower: 0 higher: 15

First Iteration:

- Calculate middle: (0+15)/2=7
- Compare arr[7] (which is 30) with 20:
    - 30 is greater than 20, so adjust the higher index:
    - new higher: 7−1=6

Second Iteration:

- Calculate new middle: (0+6)/2=3
- Compare arr[3] (which is 15) with 20:
    - 15 is less than 20, so adjust the lower index:
    - new lower: 3+1=4

Third Iteration:

- Calculate new mid: (4+6)/2=5
- Compare arr[5] (which is 23) with 20:
    - 23 is greater than 20, so adjust the higher index:
    - new higher: 5−1=4

Fourth Iteration:

- Calculate new mid: (4+4)/2=4
- Compare list[4] (which is 20) with 20:
    - 20 is equal to 20. Target found.

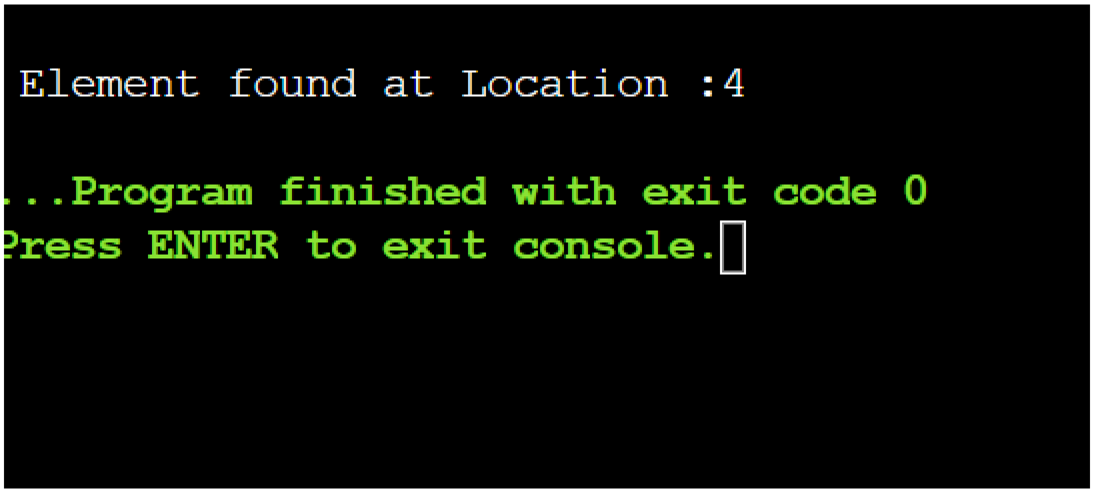
The steps do not change significantly because binary search always involves halving the search range and comparing the target with the middle element of the current range, regardless of the specific values or minor adjustments in the list's composition. The presence of the additional element (23) only slightly shifts the indices but does not alter the fundamental process of the binary search algorithm.

d) **Implement the binary search algorithm in a programming language of your choice to search for the element 20 in the original list. Provide your code and explain how it mirrors the steps shown in the image.**

**Code :**

```
#include <iostream>
using namespace std;
int binarySearch(int arr[],int size,int num){
   int lower=0,higher=size-1;
   int middle;
   while(lower<=higher){
      middle=(lower+higher)/2;
      if (arr[middle]==num){
         return middle;
      }
      if (num<arr[middle]){
         higher=middle-1;
      }
      else{
         lower=middle+1;
      }
   }
   return -1;

}
int main()
{
   int arr[]={4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 58, 65, 80, 98};
   int size=sizeof(arr)/sizeof(arr[0]);
   int num =20;
   int result =binarySearch(arr,size,num);
   if (result==-1){
      cout<<"\n Element not found in the array ";
   }
   else{
      cout<<"\n Element found at Location :"<<result;
   }
   return 0;
}
```

**Output:**



```
 Element found at Location :4

...Program finished with exit code 0
ress ENTER to exit console.
```

**e) Consider a situation where binary search might not be the best option. Identify such a scenario and explain why a different search algorithm would be preferable.**

**Answer:**

Binary search is very efficient for searching in sorted arrays, but it may not be the best option in the following scenarios:

<u>1. Unsorted  Data:</u>

Binary search requires the data to be sorted. If the data is not sorted or is poorly sorted, binary search cannot be used directly.

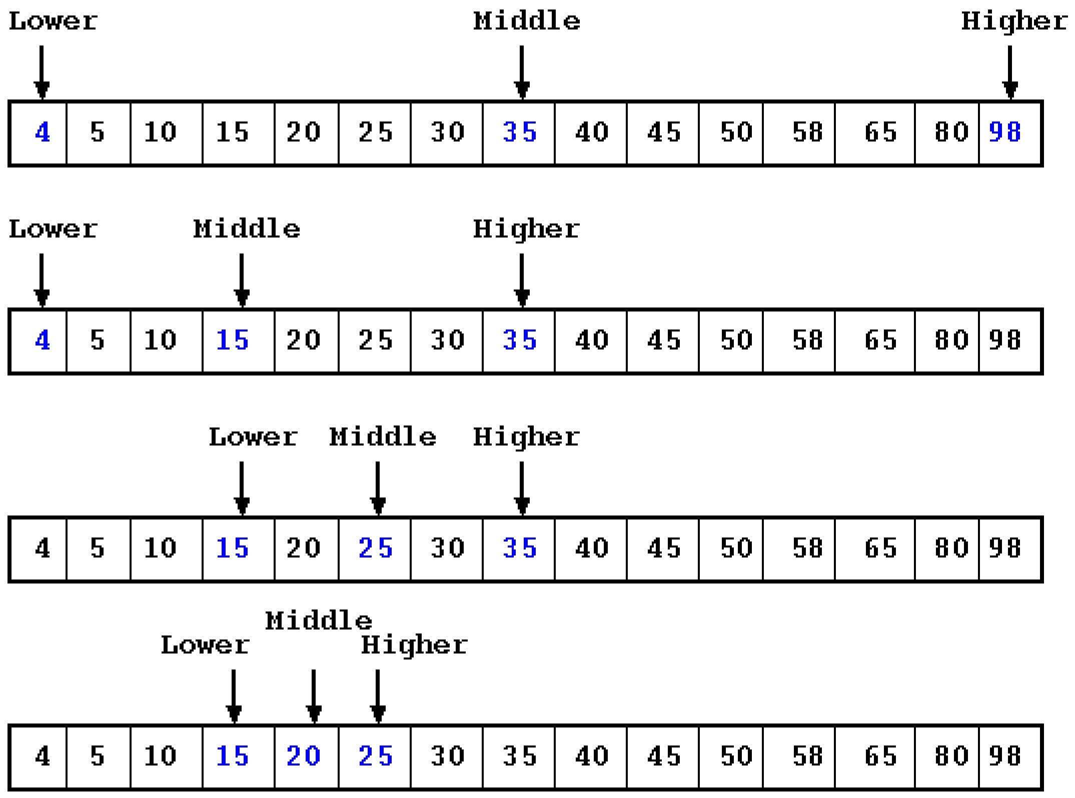
In cases where the data is unsorted, a linear search would be preferable. Linear search simply checks each element in the list sequentially until it finds the target value or reaches the end of the list. While its time complexity is O(n), it is straightforward and works without needing the data to be sorted.

<u>2. Dynamic Data with Frequent Insertions/Deletions:</u>

If the data set is dynamic, with frequent insertions and deletions, maintaining a sorted order can be costly in terms of time and computational resources.

*<u>Hash Tables</u>*: Hash tables allow for average O(1) time complexity for both search and insertion operations. This is much faster than maintaining a sorted list and then performing a binary search.

*<u>Balanced Trees (e.g., AVL Trees, Red-Black Trees)</u>*: These data structures allow for O(logn) time complexity for insertions, deletions, and searches. They maintain a sorted order while allowing for efficient dynamic operations.



---

2. **The Tower of Hanoi is a classic problem in computer science and mathematics, involving three rods and a number of disks of different sizes. The objective is to move the entire stack to another rod, obeying the following rules:**
   a. **Only one disk can be moved at a time.**
   b. **Each move involves taking the top disk from one of the stacks and placing it on top of another stack or on an empty rod.**
   c. **No disk may be placed on top of a smaller disk.**

   **For this exercise, we will use a recursive approach to solve the Tower of Hanoi problem for n disks.**

   **Analyze and solve the Tower of Hanoi problem for n disks using recursion. Answer the following questions to demonstrate your understanding and application of the recursive solution:**

a) **Describe the recursive algorithm used to solve the Tower of Hanoi problem. How does the base case differ from the recursive case**?

**Answer:** The Tower of Hanoi problem involves moving n disks from a source rod (S) to a destination rod (D) using an auxiliary rod (A). The objective is to move all disks from the source rod to the destination rod following the rules of the game.

Base Case

The base case of the recursive algorithm handles the simplest scenario, where there is only one disk to move. This is the direct action that terminates the recursion.

If n=1:

a. Move the single disk from the source rod (S) to the destination rod (D).
b. This step is straightforward and does not require further recursive calls.

Recursive Case

The recursive case handles the situation where there are more than one disk (n>1). The algorithm breaks down the problem into smaller subproblems by using recursion. The key idea is to use the auxiliary rod to temporarily hold disks while moving the larger disks directly to their final positions.

If n>1:

Move n−1 disks from the source rod (S) to the auxiliary rod (A):

i. This step is performed by recursively calling the function to move n−1n-1n−1 disks from the source rod (S) to the auxiliary rod (A), using the destination rod (D) as the auxiliary rod in this subproblem.

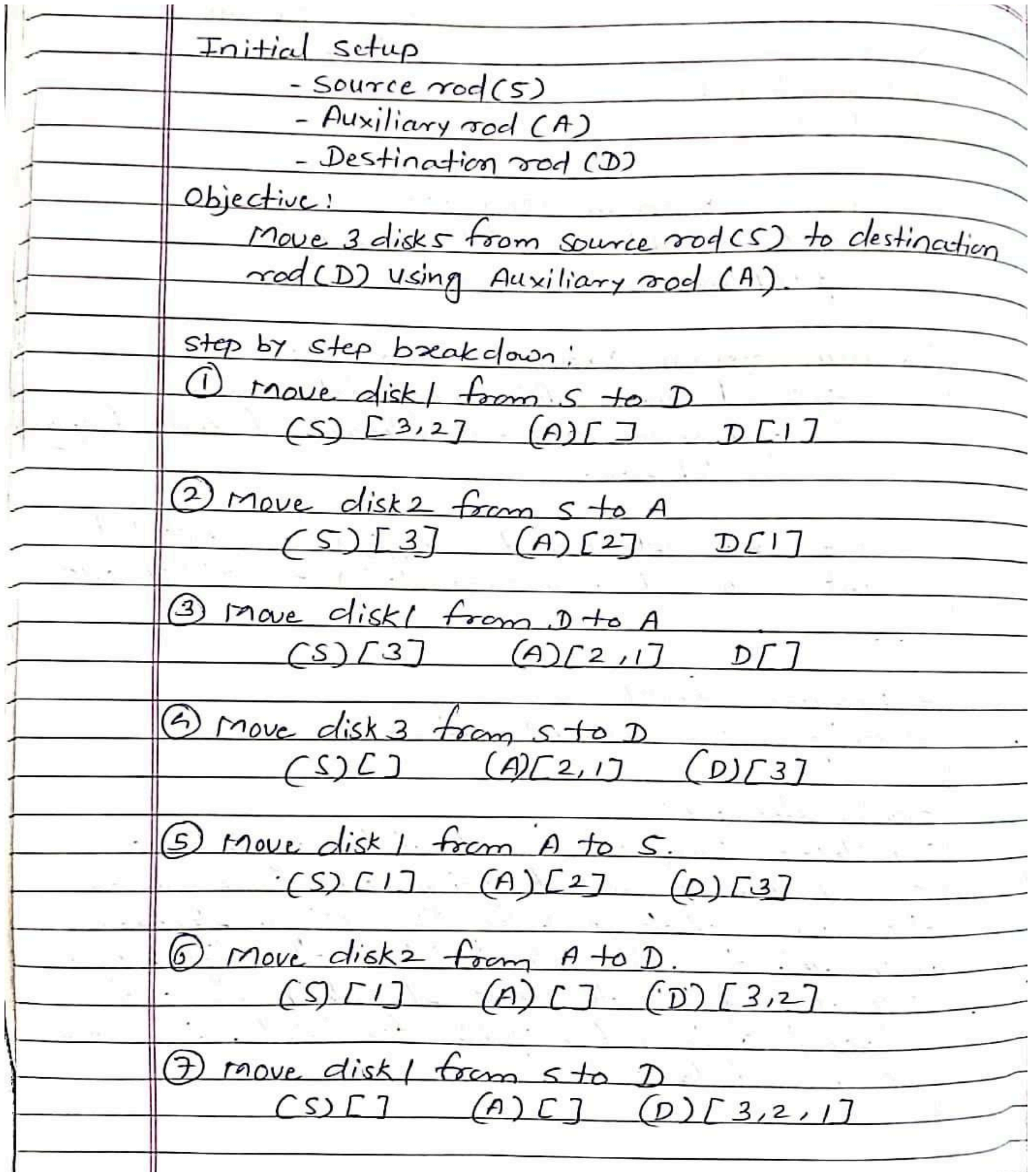
Move the n-th (largest) disk from the source rod (S) to the destination rod (D):

ii. This is a direct move and is executed after the n−1 disks have been moved to the auxiliary rod (A).

Move the n−1 disks from the auxiliary rod (A) to the destination rod (D):

iii. This step is performed by recursively calling the function to move n−1 disks from the auxiliary rod (A) to the destination rod (D), using the source rod (S) as the auxiliary rod in this subproblem.

b) **Explain the process of solving the Tower of Hanoi problem for 3 disks. Provide a step-by-step breakdown of the moves involved and how the recursive calls work together to achieve the final solution.**



Initial setup
- Source rod (S)
- Auxiliary rod (A)
- Destination rod (D)

Objective:
Move 3 disks from source rod (S) to destination rod (D) using Auxiliary rod (A).

Step by step breakdown:

① move disk 1 from S to D
(S) [3,2] (A) [ ] D [1]

② Move disk 2 from S to A
(S) [3] (A) [2] D [1]

③ Move disk 1 from D to A
(S) [3] (A) [2,1] D [ ]

④ Move disk 3 from S to D
(S) [ ] (A) [2,1] (D) [3]

⑤ Move disk 1 from A to S.
(S) [1] (A) [2] (D) [3]

⑥ Move disk 2 from A to D.
(S) [1] (A) [ ] (D) [3,2]

⑦ move disk 1 from S to D
(S) [ ] (A) [ ] (D) [3,2,1]

c) **Suppose you have 5 disks. Predict how the number of moves required changes as the number of disks increases. Use the recursive formula to calculate the number of moves needed for n disks, and validate your prediction for n = 5.**

**Answer:**

The Tower of Hanoi problem's number of moves required to solve for n disks follows a recursive formula: $T(n)=2T(n-1)+1$

where:

$T(n)$ is the number of moves required for n disks,

$T(n-1)$ is the number of moves required for $n-1$ disks.

We can calculate the number of moves required for different numbers of disks using the recursive formula.

1. Base Case: $T(1)=1$
2. For n=2: $T(2)=2T(1)+1=2(1)+1=3$
3. For n=3:

   $T(3)=2T(2)+1=2(3)+1=7$

4. For n=4 :

   $T(4)=2T(3)+1=2(7)+1=15$

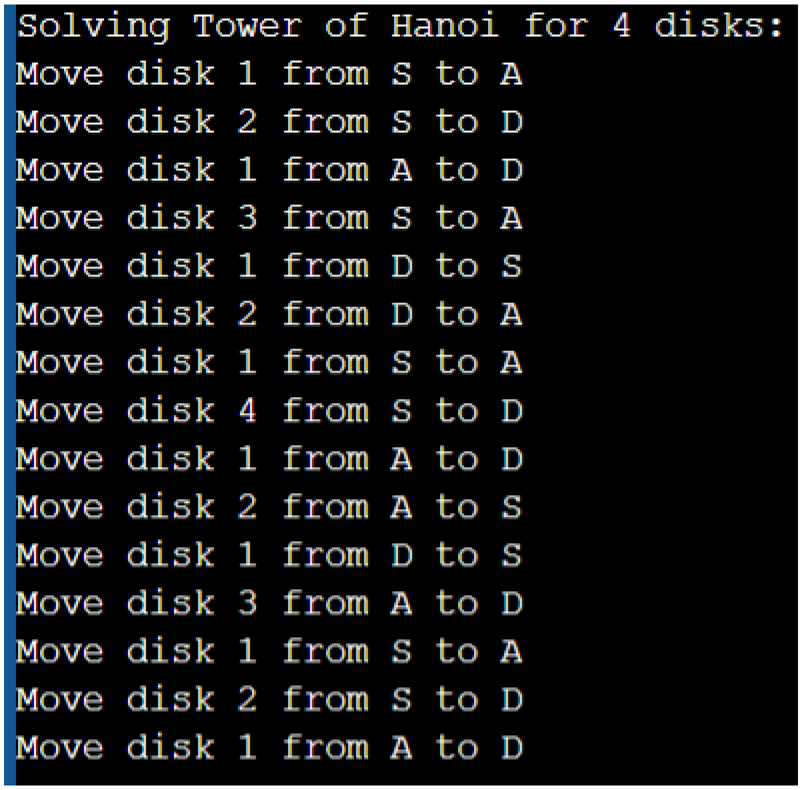
5. For n=5:

   $T(5)=2T(4)+1=2(15)+1=31$

d) **Implement the recursive algorithm for solving the Tower of Hanoi problem in a programming language of your choice. Provide the code and explain how each part of the code corresponds to the recursive steps you described earlier.**

```
#include <iostream>
using namespace std;

// Function to move n disks from source to destination using auxiliary
void move(int n, char source, char destination, char auxiliary) {
   if (n == 1) {
      // Base case: Move the single disk from source to destination
      cout << "Move disk 1 from " << source << " to " << destination << endl;
      return;
```

```
    }

    // Recursive case: Move n-1 disks from source to auxiliary using destination
    move(n - 1, source, auxiliary, destination);
    // Move the nth disk from source to destination
     cout << "Move disk " << n << " from " << source << " to " << destination <<
endl;

    // Move the n-1 disks from auxiliary to destination using source
    move(n - 1, auxiliary, destination, source);
}

int main() {
    int n = 4; // Number of disks
    cout << "Solving Tower of Hanoi for " << n << " disks:" << endl;
    move(n, 'S', 'D', 'A'); // S: Source, D: Destination, A: Auxiliary
    return 0;
}
```



```
Solving Tower of Hanoi for 4 disks:
Move disk 1 from S to A
Move disk 2 from S to D
Move disk 1 from A to D
Move disk 3 from S to A
Move disk 1 from D to S
Move disk 2 from D to A
Move disk 1 from S to A
Move disk 4 from S to D
Move disk 1 from A to D
Move disk 2 from A to S
Move disk 1 from D to S
Move disk 3 from A to D
Move disk 1 from S to A
Move disk 2 from S to D
Move disk 1 from A to D
```

e) **Evaluate the efficiency of the recursive solution. Are there any potential limitations or drawbacks of using recursion for the Tower of Hanoi problem, particularly as the number of disks increases? Suggest an alternative approach or optimization, if applicable.**

**Answer:**

<u>Efficiency Analysis:</u>

1. Time Complexity:
   - The time complexity of the Tower of Hanoi recursive solution is $O(2^n)$.
   - This is because each move of n disks involves making 2 recursive calls for n−1 disks and 1 additional move.
   - As n increases, the number of moves grows exponentially. Specifically, the number of moves required is $2^n-1$.
2. Space Complexity:
   - The space complexity is O(n) due to the recursion stack.
   - For each disk, a recursive call is made, so the maximum depth of the recursion stack is n.

<u>Limitations and Drawbacks</u>

1. Exponential Growth:
   - The exponential growth in the number of moves means that for a large number of disks, the number of required operations becomes impractically high.
   - For example, solving the problem for 64 disks would require $2^{64} - 1$ moves, which is a very large number.
2. Stack Overflow:
   - Recursive solutions are limited by the maximum stack size. If n is very large, the recursion depth can exceed the stack limit, leading to a stack overflow error.

<u>Alternative Approach or Optimization</u>

1. Iterative Solution:
   - An iterative solution can avoid the limitations of recursion, such as stack overflow.
   - The iterative solution uses an explicit stack or loop to simulate the recursive process.
2. Using an Explicit Stack:
   - We can use an explicit stack data structure to simulate the recursion, allowing the algorithm to handle larger values of n.