

Slot 08 - Recursion

Presenter:

Dr. LE Thanh Tung

- 1 Introduction
- 2 Recursion Function
- 3 Examples

- Recursion is an extremely powerful problem-solving technique
- Recursion is encountered not only in mathematics, but also in daily life
- An object is said to be recursive if it is defined in terms of a smaller version of itself



- This technique provides a way to break complicated problems down into simple problems which are easier to solve
- For example, we can define the operation "find your way home" as:
 - If you are at home, stop moving
 - Take one step toward home
 - "find your way home"

- All recursive algorithms must have the following:
 - Base Case (i.e., when to stop) - Halting Condition
 - Work toward Base Case
 - Recursive Call (i.e., call ourselves)

- Natural numbers
 - 0 is a natural number
 - The successor of a natural number is a natural number
- Fractional:
 - $0! = 1$
 - $n! = n * (n - 1)!$
- Fibonacci numbers:
 - $F(0) = 0, F(1) = 1$
 - $F(n) = F(n-1) + F(n-2)$

- Fibonacci numbers:
 - Base case:
 - $F(0) = 0, F(1) = 1$
 - Work toward base case:
 - $F(n) = F(n-1) + F(n-2)$
 - Recursive Call: ??

- So far, we have learned about control structures that allow C++ to iterate a set of statements a number of times
- In addition to iteration, C++ can repeat an action by having a **function call itself**.
 - This is called recursion
 - In some case it is more suitable than iteration

```
void message() {  
    cout << "Hello" << endl;  
    message();    // recursive function call  
}
```


- If the halting condition is not well-defined, the recursion function will be looped infinitely.

```
void message() {  
    cout << "Hello" << endl;  
    message();    // recursive function call  
}
```

- We should determine the base case carefully

```
void message(int n) {  
    if (n == 0)    return;  
    cout << "Hello" << endl;  
    message(n - 1); // recursive function call  
}  
message(5);        // function call
```

- The structure of recursive functions is typically like the following

```
RecursiveFunction(){  
    if (test for simple case){  
        Compute the solution without recursion  
    }  
    else{  
        Break the problem into subproblems of the same form  
        Call RecursiveFunction() on each subproblem  
        Reassamble the results of the subproblems  
    }  
}
```

- 3 “must have” of a recursive algorithm
 - Your code must have a case for **all valid inputs**
 - You must have a **base case** with **no recursive calls**.
 - When you make a recursive case, it should be to a **simpler instance** and make forward progress towards the base case

- Example: Calculating $n!$
 - Definition 1:
 - If $n = 0$: $n! = 1$
 - If $n > 0$: $n! = 1 \times 2 \times 3 \times \cdots \times n$

→ Can not use recursion

- Definition 2:
 - If $n = 0$: $n! = 1$
 - If $n > 0$: $n! = (n - 1)! \times n$

→ Can use recursion

- Example: Calculating $n!$
 - Definition 2:
 - If $n = 0$: $n! = 1$
 - If $n > 0$: $n! = (n - 1)! \times n$

```
int calcFactorial(int n){  
    if (n == 0) return 1;  
    else return n*calcFactorial(n - 1);  
}
```

```
int func1(int a1)
{
    return a1*2;
}
```

```
int func2(int a2)
{
    int x, z;
    x = 5;
    z = func1(a2)+x;
    return z;
}
```

- An activation record is stored into a function **call stack (run-time stack)**
 - The computer stops executing **func2** and starts executing **func1**
 - Since it needs to come back to **func2** later, it needs to store everything about **func2** that is going to need (a2, x, z and the place to start executing upon return)
 - Then, **a2** from func2 is bounded to **a1** from **func1**
 - Control is transferred to **func1**

```
int func1(int a1)
{
    return a1*2;
}
```

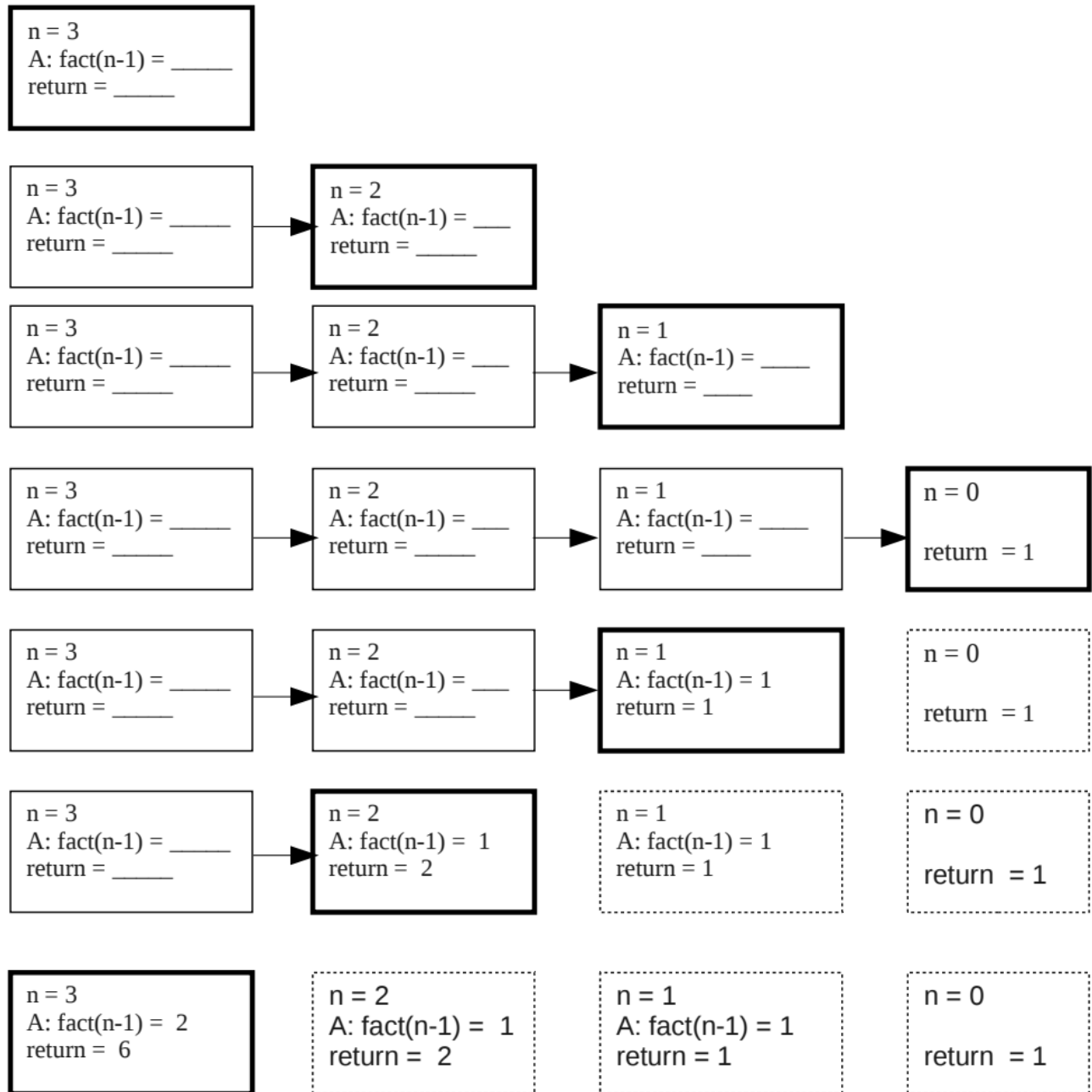
```
int func2(int a2)
{
    int x, z;
    x = 5;
    z = func1(a2)+x;
    return z;
}
```

- An activation record is stored into a function **call stack (run-time stack)**
 - After **func1** is executed, the activation record is popped out of the run-time stack
 - All the old values of the parameters and variables in **func2** are restored and the return value of **func1** replaces **func1(a2)** in the assignment statement

- When a recursive call is encountered, execution of the current function is **temporarily stopped**
 - This is because the **result** of the recursive call **must be known** before it can proceed
- So, it **saves** all of the information it needs in order to continue executing that function later
 - i.e., all current values of all local variables and the location where it stopped
- Then, when the recursive call is completed, the computer **returns** and **completes** execution of the function.

Tracing Recursion

- Read more:
<https://codeahoy.com/learn/recursion/ch8/>



- A function is called directly recursive if it calls itself
- A function that calls another function and eventually results in the original function call is said to be indirectly recursive
- A recursive function in which the last statement executed is a recursive call is called a **tail recursive function**

```
int calcPower(int x, int n) {  
    if (n == 0)  
        return 1;  
    return calcPower(x, n - 1) * x;  
}
```

- Print out from numbers from n to 1
 - **Input: n = 5**
 - **Output: 5 4 3 2 1**

- Print out from numbers from n to 1
 - **Input: n = 5**
 - **Output: 5 4 3 2 1**

```
void printNum(int n){  
    if (n == 0){  
        cout << endl;  
        return;  
    }  
    cout << n << " ";  
    printNum(n - 1);  
}
```

- Calculating the sum of all elements in an array:

$$a_0 + a_1 + \cdots + a_{n-1}$$

- **Recursion definition:**

$$S_n = a_0 + a_1 + \cdots + a_{n-1}$$

$$S_n = S_{n-1} + a_{n-1}$$

- **Base case:** $S_1 = a_0$

- Calculating the sum of all elements in an array:

```
int sum(int arr[], int n) {  
    if (n == 0) { // base case  
        return 0;  
    } else {  
        return arr[n-1] + sum(arr, n-1); // recursive case  
    }  
}
```

- Verifying if the elements in an array are in ascending order

$$a_0 \leq a_1 \leq \dots \leq a_{n-1}?$$

- The above array is in ascending order if it satisfies two conditions
 - The first $n - 1$ elements are in ascending order, and
 - $a_{n-2} \leq a_{n-1}$
- If the array contains only one element (a_0), it must be in ascending order

- Verifying if the elements in an array are in ascending order

```
bool isAscending(int arr[], int size) {  
    // base case  
    if (size <= 1) {  
        return true;  
    }  
  
    // recursive case  
    if (arr[size - 1] < arr[size - 2]) {  
        return false;  
    } else {  
        return isAscending(arr, size - 1);  
    }  
}
```


- Stack Overflow: means that you've tried to make too many function calls recursively and the memory in stack is full
- If you get this error, one clue would be to look to see if you have infinite recursion
 - This situation will cause you to exceed the size of your stack -- no matter how large your stack is

- While recursion is very powerful
 - It should not be used if iteration can be used to solve the problem in a maintainable way (i.e., if it isn't too difficult to solve using iteration)
 - So, think about the problem. Can loops do the trick instead of recursion?

- Why select iteration versus recursion
 - Every time we call a function a stack frame is pushed onto the program stack and a jump is made to the corresponding function
 - This is done in addition to evaluating a control structure (such as the conditional expression for an if statement to determine when to stop the recursive calls
 - With iteration all we need is to check the control structure (such as the conditional expression for the while, do-while, or for) → efficiency

- Iteration can be used in place of recursion
 - An iterative algorithm uses a **looping construct**
 - A recursive algorithm uses a **branching structure**
- Recursive solutions are often less efficient, in terms of both **time** and **space**, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in **shorter**, more easily understood source code

- Verifying if a string is a palindrome
- Formally, a palindrome can be defined as follows:
 - If a string is a palindrome, it must begin and end with the same letter. Further, when the first and last letters are removed, the resulting string must also be a palindrome
 - A string of length 1 is a palindrome
 - The empty string is a palindrome

- Verifying if a string is a palindrome

```
bool isPalindrome(string str, int start, int end) {  
    // Base case: If there is only one character  
    if (start == end) {  
        return true;  
    }  
    // If the first and last characters don't match  
    if (str[start] != str[end]) {  
        return false;  
    }  
    // Recursively check if the remaining substring is a palindrome  
    if (start < end + 1) {  
        return isPalindrome(str, start + 1, end - 1);  
    }  
    return true;  
}
```

- Verifying if a string is a palindrome

```
bool isPalindrome(string str, int start, int end) {  
    // Base case: If there is only one character  
    if (start == end) {  
        return true;  
    }  
    // If the first and last characters don't match  
    if (str[start] != str[end]) {  
        return false;  
    }  
    // Recursively check if the remaining substring is a palindrome  
    if (start < end + 1) {  
        return isPalindrome(str, start + 1, end - 1);  
    }  
    return true;  
}
```

- Verifying if a string is a palindrome

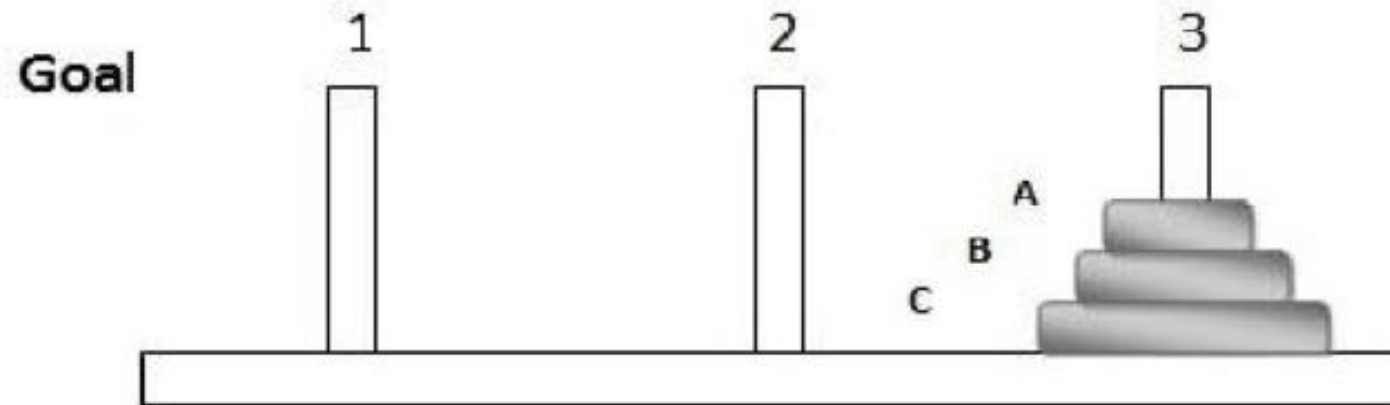
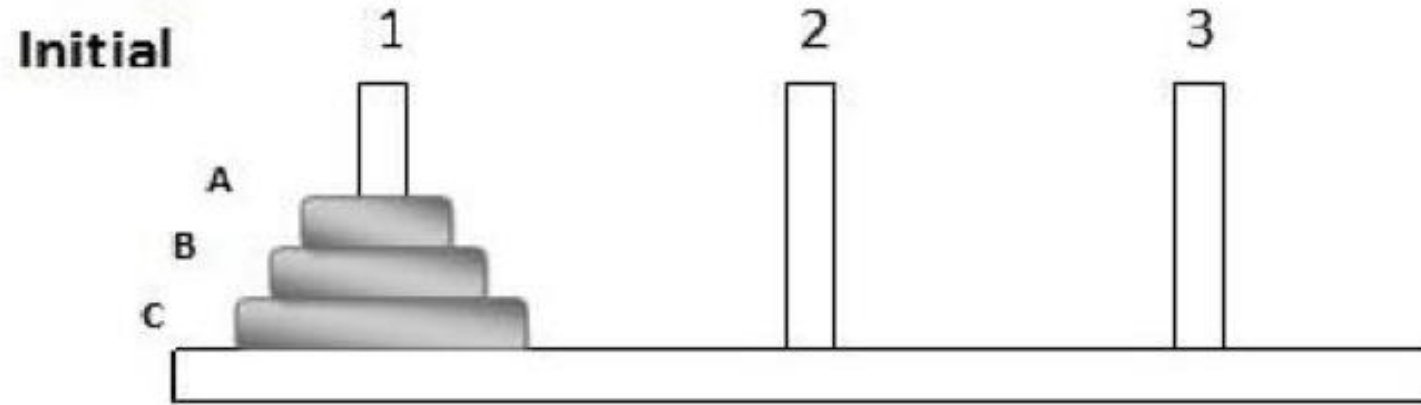
```
bool isPalindrome(string str) {  
    int len = str.length();  
    if (len <= 1) {  
        return true;  
    } else if (str[0] != str[len - 1]) {  
        return false;  
    } else {  
        // Eliminate the first and last character in str  
        return isPalindrome(str.substr(1, len - 2));  
    }  
}
```


- Implement binary search with recursion

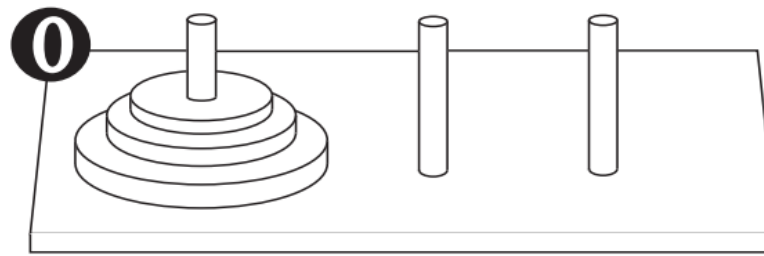
Exercise

- Implement binary search with recursion

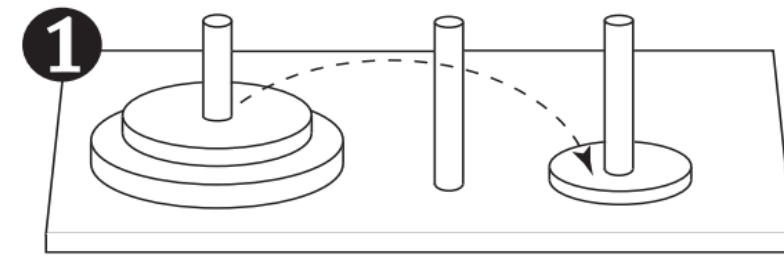
```
int binarySearch(int arr[], int l, int r, int x) {  
    if (r >= l) {  
        int mid = l + (r - l) / 2;  
  
        // If the element is present at the middle itself  
        if (arr[mid] == x)  
            return mid;  
  
        // If element is smaller than mid, then it can only  
        // be present in left subarray  
        if (arr[mid] > x)  
            return binarySearch(arr, l, mid - 1, x);  
  
        // Else the element can only be present in right subarray  
        return binarySearch(arr, mid + 1, r, x);  
    }  
  
    // We reach here when element is not present in array  
    return -1;  
}
```



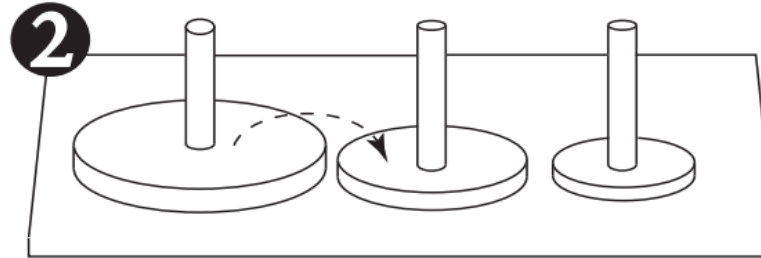
Hanoi Tower Puzzle



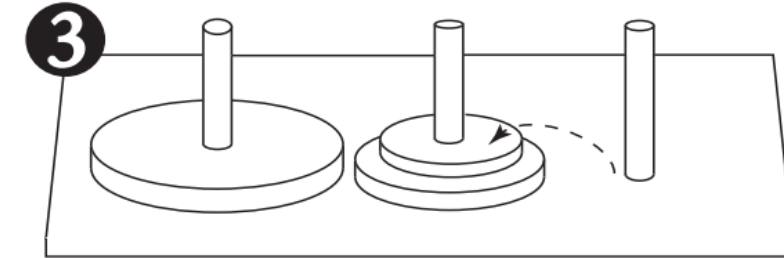
Original setup.



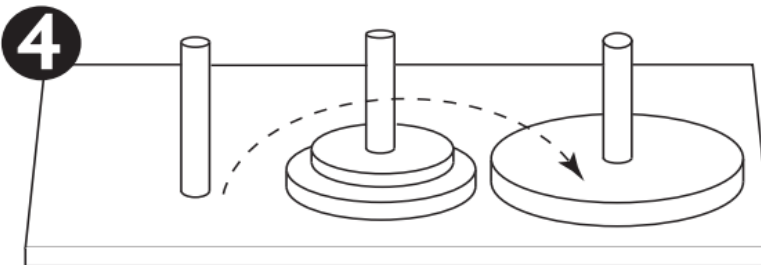
First move: Move disc 1 to peg 3.



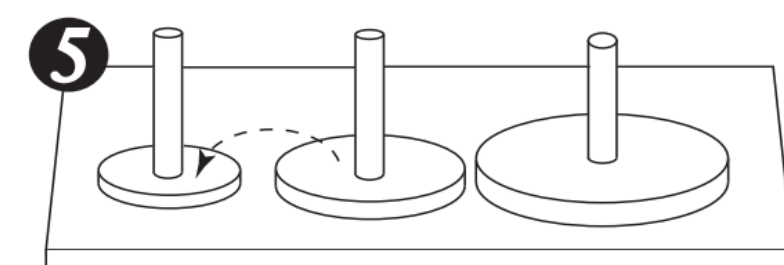
Second move: Move disc 2 to peg 2.



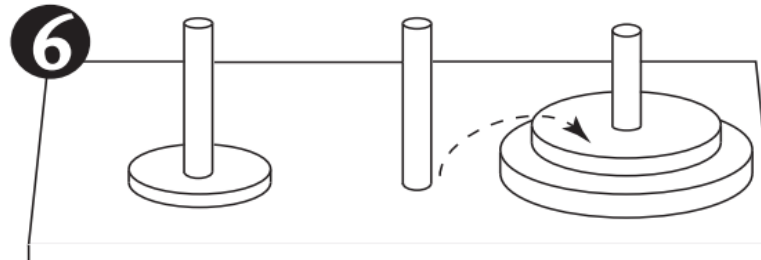
Third move: Move disc 1 to peg 2.



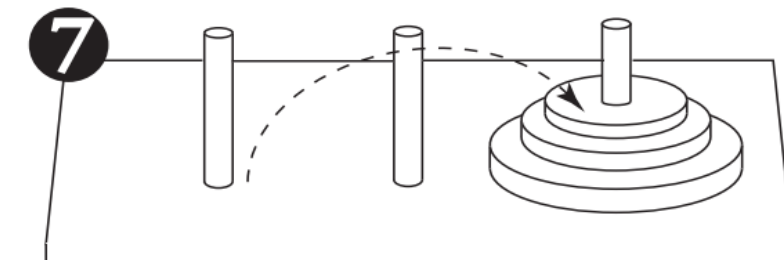
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.

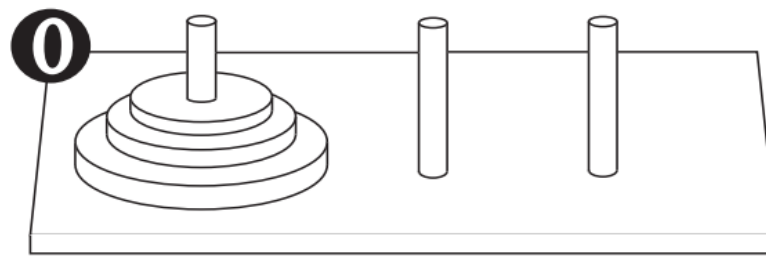


Sixth move: Move disc 2 to peg 3.

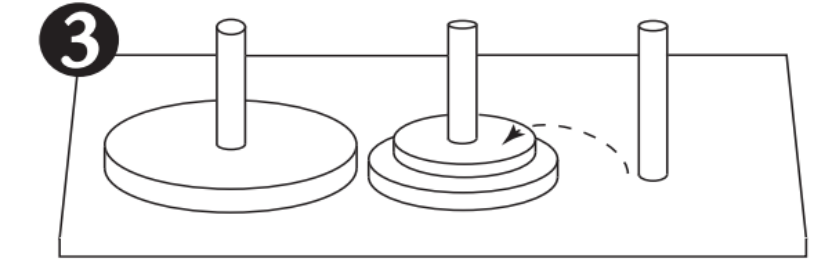
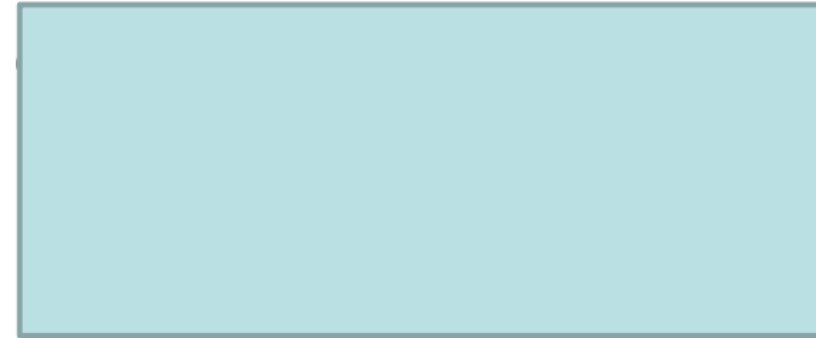


Seventh move: Move disc 1 to peg 3.

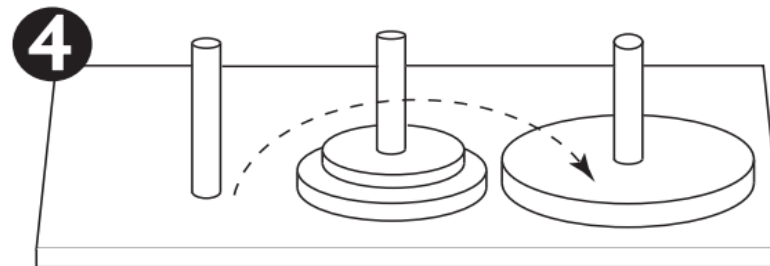
Hanoi Tower Puzzle



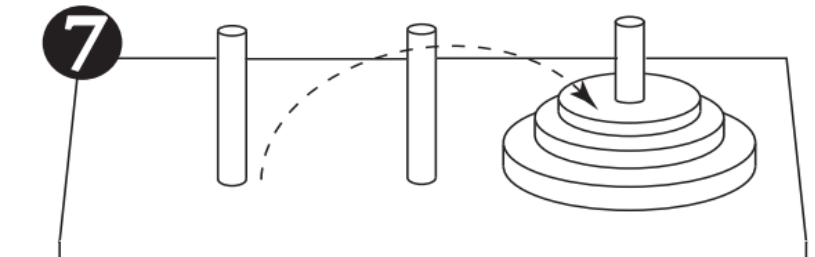
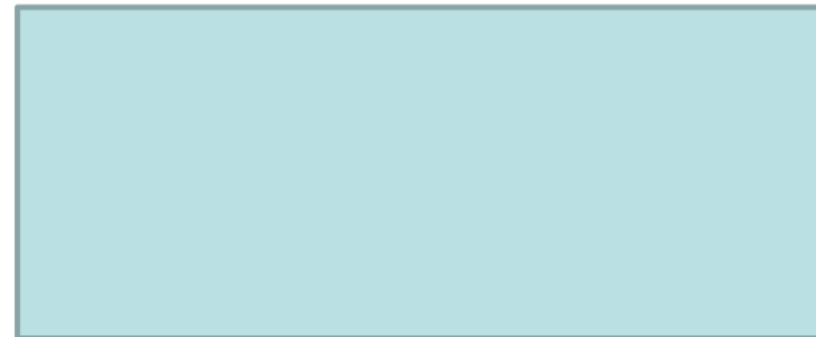
Original setup.



Third move: Move disc 1 to peg 2.



Fourth move: Move disc 3 to peg 3.



Seventh move: Move disc 1 to peg 3.

Move n discs from peg 1 to peg 3 using peg 2 as an auxiliary peg

If $n > 0$ **then**

 Move $n - 1$ discs from peg 1 to peg 2, using peg 3 as an auxiliary peg

 Move the remaining disc from the peg 1 to peg 3

 Move $n - 1$ discs from peg 2 to peg 3, using peg 1 as an auxiliary peg

End If

```
void hanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        cout << "Move disk 1 from " << from << " to " << to << endl;
        return;
    }
    hanoi(n-1, from, aux, to);
    cout << "Move disk " << n << " from " << from << " to " << to << endl;
    hanoi(n-1, aux, to, from);
}
```

- Rewrite the insert (addLast) and remove (remove data) functions with linked lists using recursion

- Rewrite the insert (addLast) and remove (remove data) functions with linked lists using recursion

```
void insert(Node*& head, int data) {  
    if (head == nullptr) {  
        head = new Node{ data, nullptr };  
    } else {  
        insert(head->next, data);  
    }  
}
```

```
void remove(Node*& head, int data) {  
    if (head == nullptr) {  
        return;  
    } else if (head->data == data) {  
        Node* temp = head;  
        head = head->next;  
        delete temp;  
    } else {  
        remove(head->next, data);  
    }  
}
```

THANK YOU
for YOUR ATTENTION