VNUHCM - University of Science
fit@hcmus

CSC10004 – Data Structures and Algorithms

Session 01
Recursion

Instructors:
Dr. Lê Thanh Tùng

Content

- 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

Introduction

- 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"

Introduction

- 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)

Recursion Example

- 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:
  - $F(0) = 0, F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

Recursion Example

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

Recursion Function

- 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
    }
}
```

Recursion Function

- 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: $\mathrm{n}! = 1 \times 2 \times 3 \times \cdots \times n$

  → Can not use recursion
  - Definition 2:
    - If n = 0: $n! = 1$
    - If n > 0: $\mathrm{n}! = (\mathrm{n} - 1)! \times n$

  → Can use recursion

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

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

when a function is called


```
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**

when a function is called

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

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

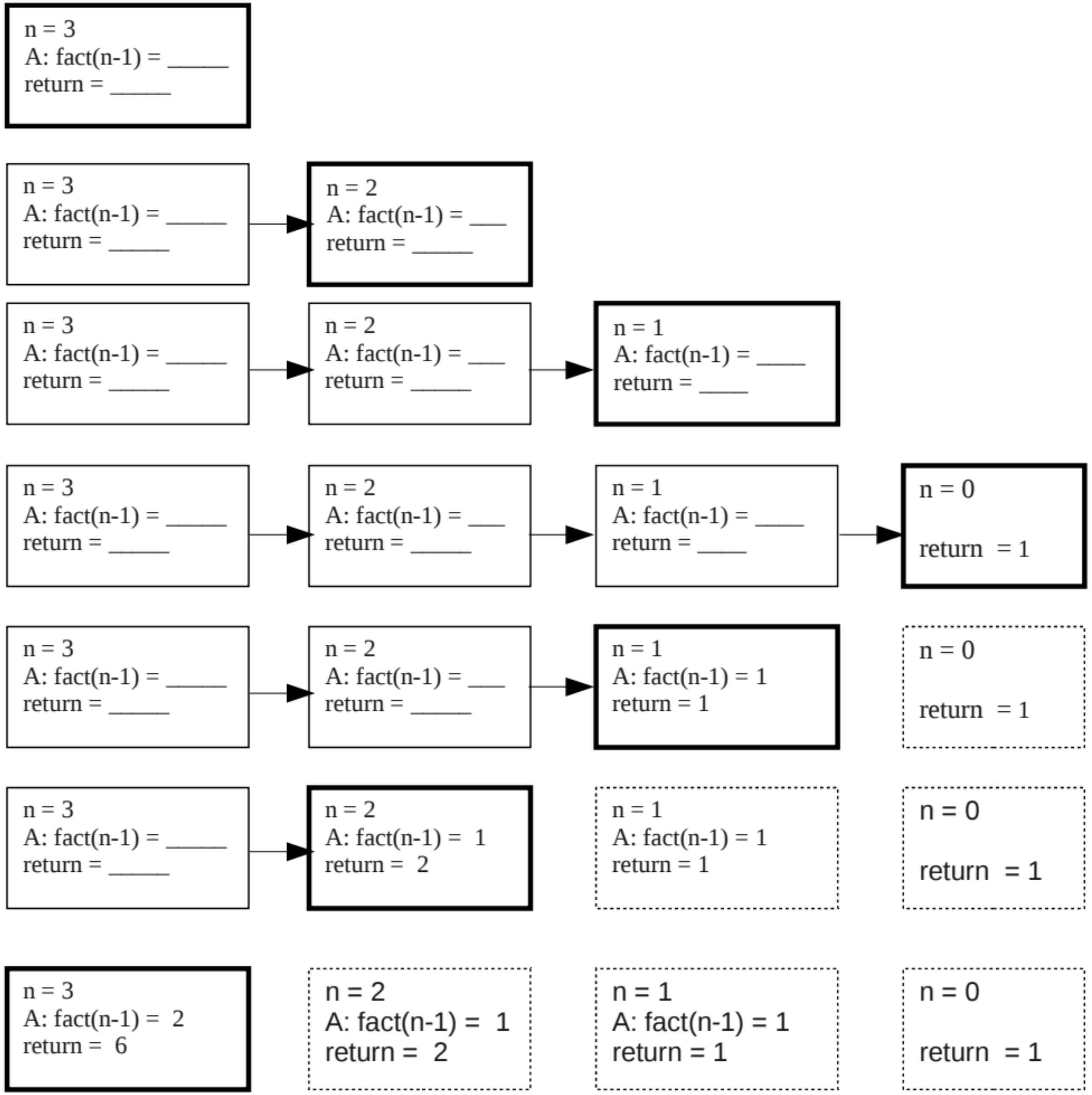
- An activation record is stored into a function **call stack (run-time stack)**
  - After **func1** is executed, the activation record is popped out of the run-time stack
  - All the old values of the parameters and variables in **func2** are restored and the return value of **func1** replaces **func1**(**a2**)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

when a function is called

- 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/



Direct and Indirect Recursion

- 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;
}
```

Recursion Example

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

Recursion Example

- 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);
}
```

Recursion Example

- 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:** $S_1 = a_0$

Recursion Example

- Calculating the sum of all elements in an array:

```
int recursiveSum(int arr[], int size) {
    // Base case:
    // if the array size is zero, return 0
    if (size == 0) {
        return 0;
    }
    // Recursive case:
    // return the last element and sum the rest
    return arr[size - 1] + recursiveSum(arr, size - 1);
}
```

Recursion Example

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

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

- 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$$), it must be in ascending order

Recursion Example

- 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);
}
```

The most common error

- 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) $\rightarrow$ effiency

Recursion and Iteration

- 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

Exercise

- 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

Exercise

- 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;
}
```

Exercise

- Verifying if a string is a palindrome

```cpp
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;
}
```

Exercise

- Verifying if a string is a palindrome

```cpp
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));
    }
}
```

Exercise

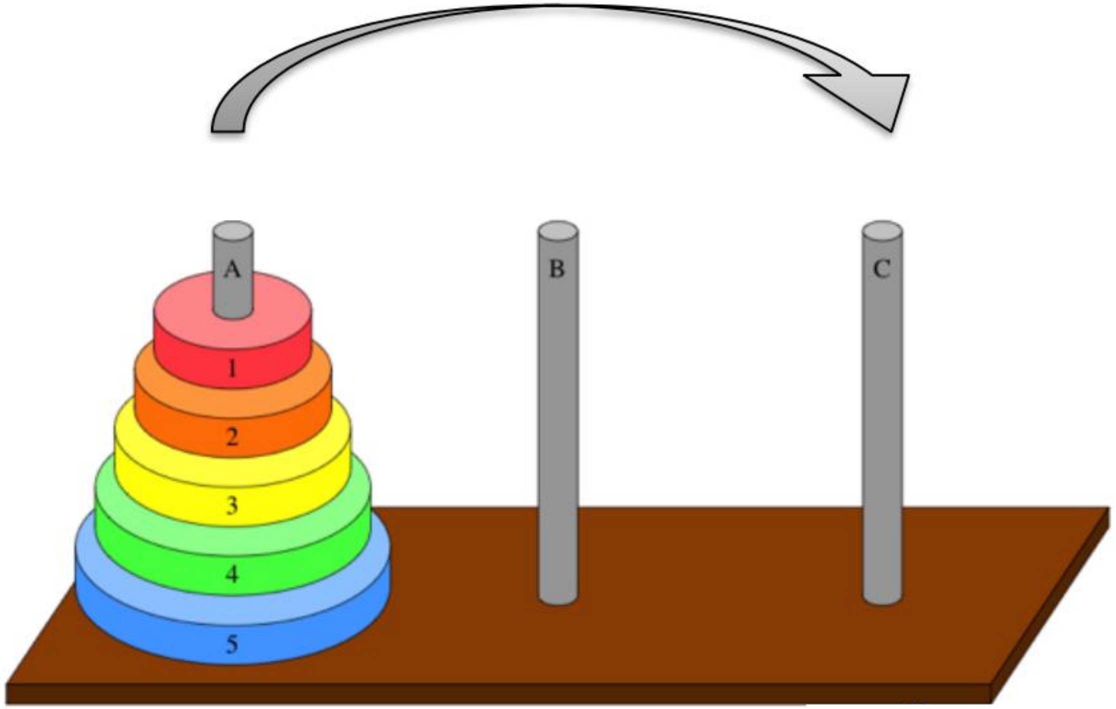
- Implement binary search with recursion

Exercise

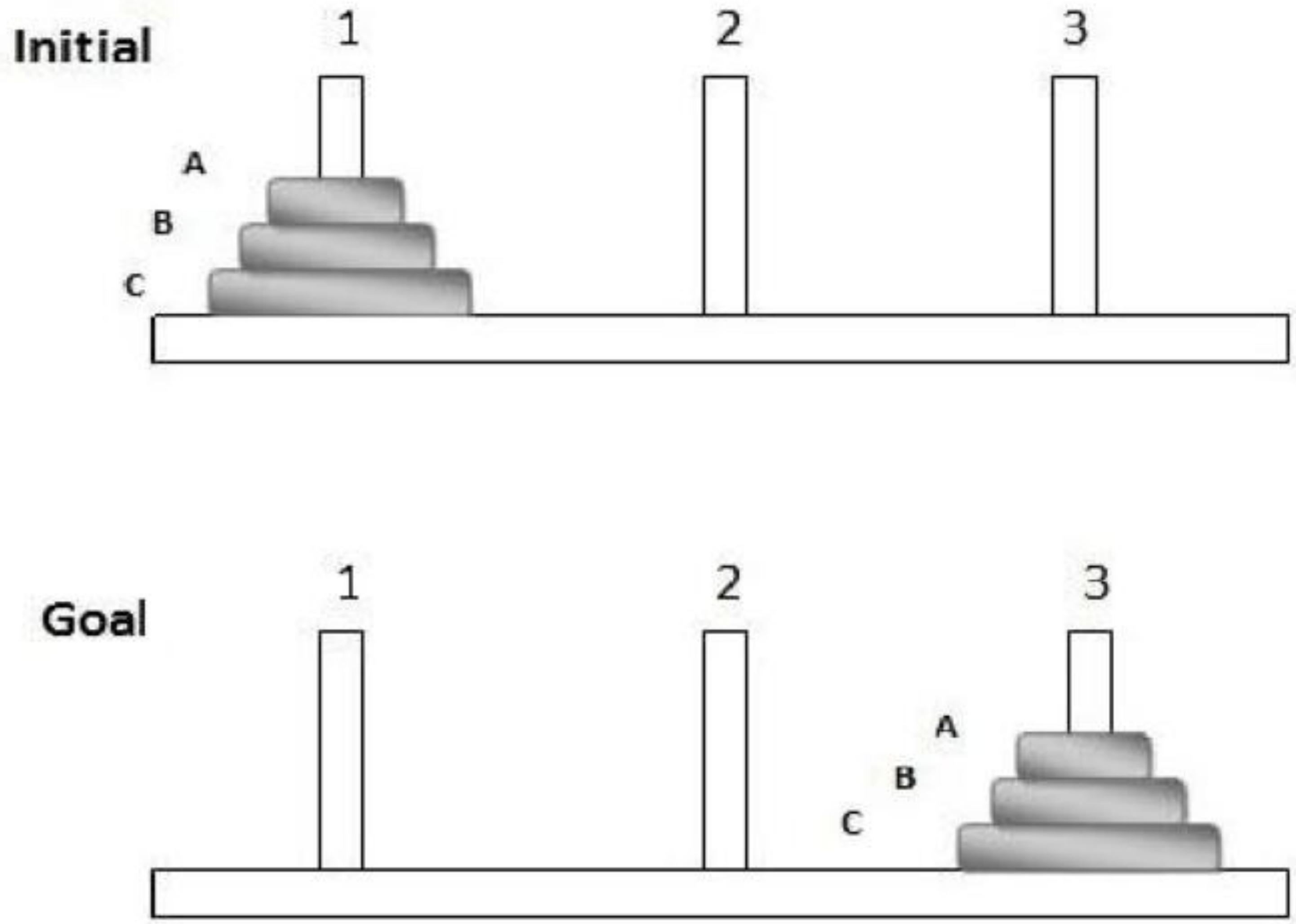
- Implement binary search with recursion

```
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;
}
```

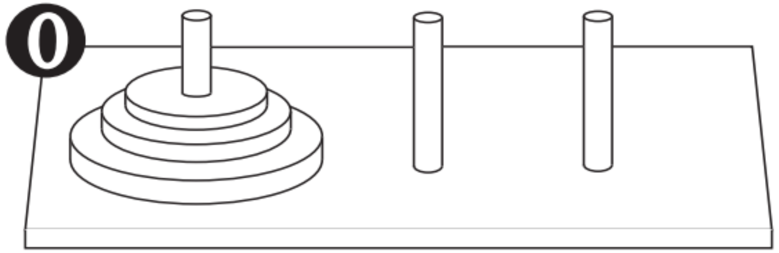
- Given a set of three pegs A, B, C, and n disks, with each disk a different size (disk 1 is the smallest, disk n is the largest)
- Initially, n disks are on peg A, in order of decreasing size from bottom to top.
- The goal is to move all n disks from peg A to peg C
- 2 rules:
  - You can move 1 disk at a time.
  - Smaller disk must be above larger disks



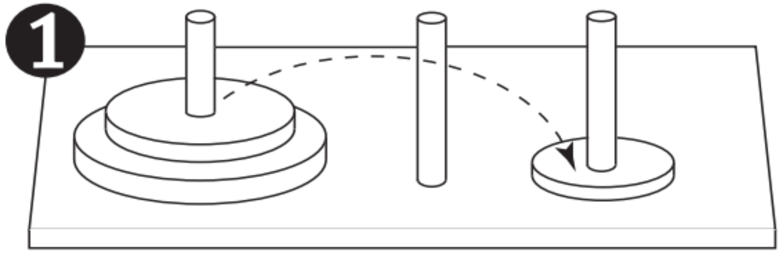
The Towers of Hanoi Puzzle



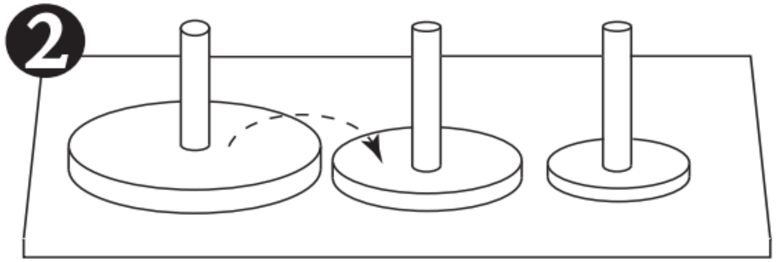
Hanoi Tower Puzzle



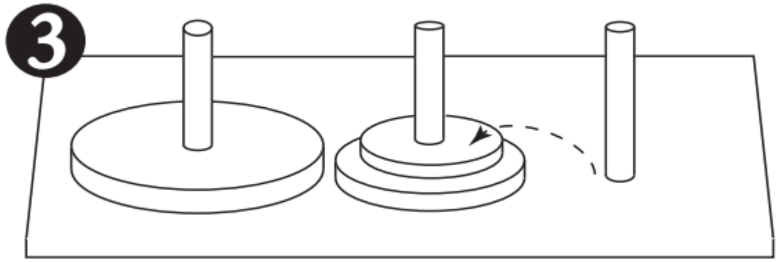
Original setup.



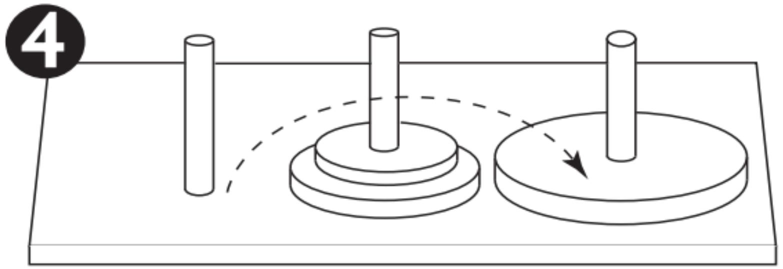
First move: Move disc 1 to peg 3.



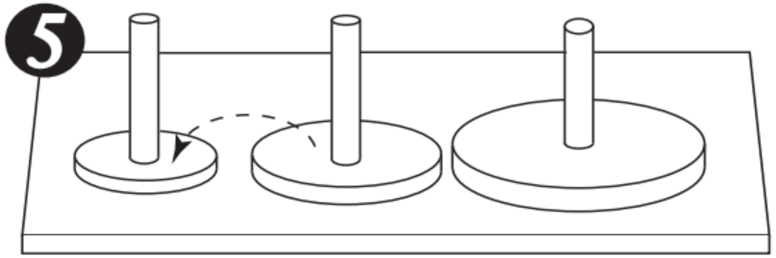
Second move: Move disc 2 to peg 2.



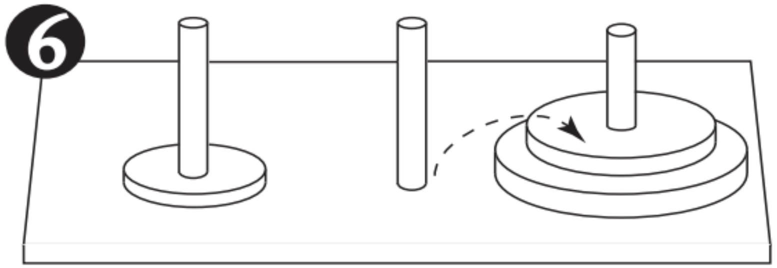
Third move: Move disc 1 to peg 2.



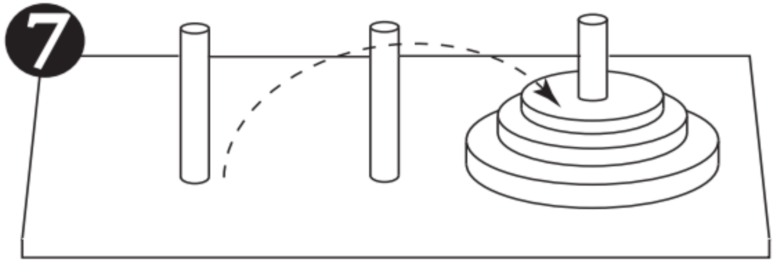
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.

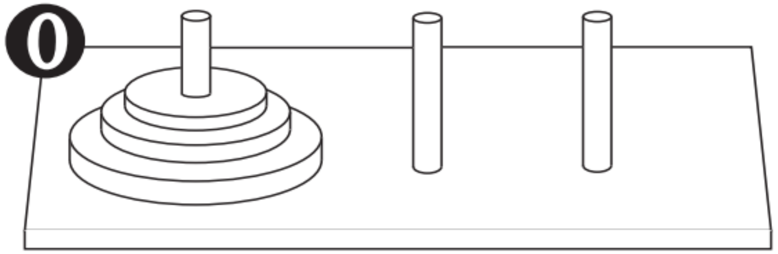


Sixth move: Move disc 2 to peg 3.

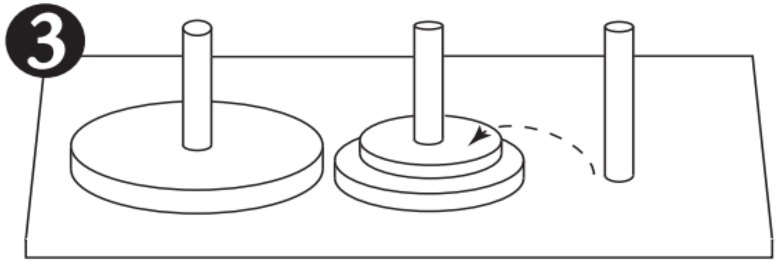


Seventh move: Move disc 1 to peg 3.

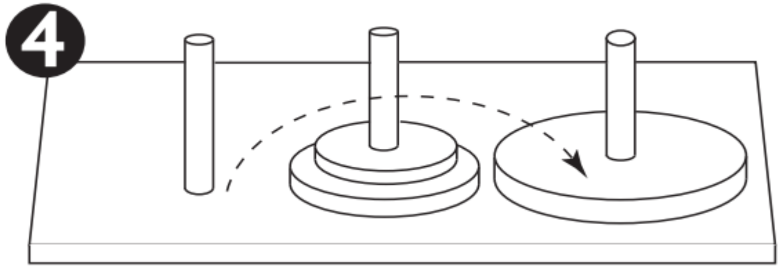
Hanoi Tower Puzzle



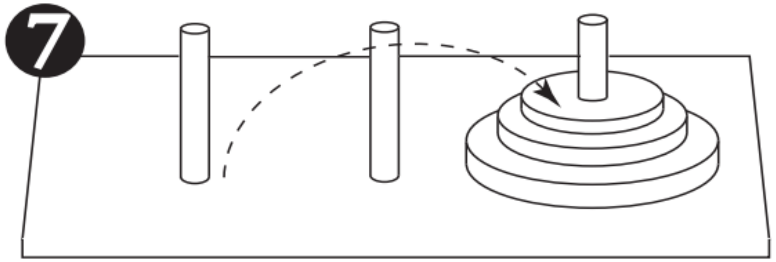
Original setup.



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

```cpp
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);
}
```

Exercise

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

Exercise

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

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

```cpp
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);
    }
}
```

Type of Recursion

- Direct Recursion & Indirect Recursion
- Linear Recursion, Binary Recursion & Multiple Recursion
- Tail Recursion vs Non-tail Recursion
- Nested Recursion

Type of Recursion


```
// direct recursion
int fact(int x){
    if (x == 0)
        return 1;
    else
        return x*fact(x-1);
}
```

```
// indirect recursion
bool isOdd(int x){
    return !isEven(x);
}

bool isEven(int x){
    if (x == 0)
        return true;
    else
        return isOdd(x - 1);
}
```

Type of Recursion

```
// linear recursion
int fact(int x){
    if (x == 0)
        return 1;
    else
        return x*fact(x-1);
}
```

```
// binary recursion
int fibo(int x){
    if (x < 2)
        return x;
    else
        return fibo(x - 1) + fibo(x - 2);
}
```

```cpp
// Tail Recursion
void printNum_tail(int n){
    cout << n << endl;
    if(n > 0)
        printNum_tail(n-1);
}
```

```cpp
// Non-tail Recursion
void printNum_nontail(int n){
    if(n > 0){
        printNum_nontail(n-1);
        cout << n << endl;
    }
}
```

- Nested Recursion $$h(n) = \begin{cases} 0 & if\ n = 0 \\ n & if\ n > 4 \\ h(2 + h(2n)) & if\ n \leq 4 \end{cases}$$

```
// nested recursion
int func(int n){
    if(n ==0) return 0;
    if(n > 4) return n;
    return
        func(2+ func(2*n));
}
```

Examples

- Write a program in C to count the digits of a given number using recursion
- E.g: n = 5413 $\rightarrow$ output: 4

Examples

- Write a program in C to count the digits of a given number using recursion

```c
int countDigits(int n) {
    if (n == 0) {
        return 0;
    } else {
        return 1 + countDigits(n / 10);
    }
}
```

Examples

- Write a program in C to convert a decimal number to binary using recursion

Examples

- Write a program in C to convert a decimal number to binary using recursion

```
void decimalToBinary(int decimal) {
    if (decimal == 0) {
        return;
    } else {
        decimalToBinary(decimal / 2);
        cout << decimal % 2;
    }
}
```

Examples

- Count the number of items in a linear linked list recursively

Examples

- Count the number of items in a linear linked list

```
int countItems(Node* head) {
    if (head == NULL) {
        return 0;
    } else {
        return 1 + countItems(head->next);
    }
}
```

Examples

- Delete all nodes in a linear linked list by recursion

Examples

- Delete all nodes in a linear linked list by recursion

```
void removeAll(Node*& head) {
    if (head == NULL) {
        return;
    }
    Node* nextNode = head->next;
    delete head;
    head = NULL;
    removeAll(nextNode);
}
```

Examples

- What is the output of the following program as calling **`watch(-7)`**

```
int watch(int n) {
    if (n > 0)
        return n;
    cout << n <<endl;
    return watch(n+2)*2;
}
```

Removing Recursion

- “Never hire a developer who computes the factorial using Recursion”
- Complex Recursion that is hard to understand should probably be considered a "bad smell" in the code and a good candidate to be replaced with Iteration
- There are 2 techniques to remove recursion
  - **Iteration**
  - Stacking

- The Simple Method:
  - Convert all recursive calls into tail calls
  - Introduce a loop around the function body
  - Convert tail calls into continue statements
  - Tidy up
- Mechanics:
  - Determine the base case of the Recursion
  - Implement a loop that will iterate until the base case is reached
  - Make a progress towards the base case

```
int Fact(int n){
  if(n < 2)
    return 1;
  return n * Fact(n-1);
}
```

Non-tail recursion

```
int Fact(int n, int acc=1){
  if(n < 2)
    return 1 * acc;
  return Fact(n-1, acc * n);
}
```

Convert to Tail recursion

Replacing Recursion with Iteration

```
int Fact(int n, int acc=1){
  while(true)
  {
    if(n < 2)
      return 1 * acc;
    (n, acc) = (n-1, acc * n);
    continue;
  }
}
```

Introduce a loop

Replace recursive call by the original function's agurment list

Replacing Recursion with Iteration

```
int Fact(int n, int acc=1){
  while(n > 1)
  {
    n = n - 1;
    acc = acc * n;
  }
  return acc;
}
```

Tidy up!

Example

- Merge two sorted linear linked lists, keeping the result sorted
  - With recursion
  - Without recursion

- Merge two sorted linear linked lists, keeping the result sorted, recursively

```
Node* mergeTwoLists(Node* head1, Node* head2) {
    if (head1 == NULL) {
        return head2;
    }
    if (head2 == NULL) {
        return head1;
    }

    Node* merged;
    if (head1->data < head2->data) {
        merged = head1;
        merged->next = mergeTwoLists(head1->next, head2);
    } else {
        merged = head2;
        merged->next = mergeTwoLists(head1, head2->next);
    }
    return merged;
}
```

Example

- Merge two sorted linear linked lists, keeping the result sorted without recursion

```
Node* mergedHead = NULL;
if (head1->data <= head2->data) {
    mergedHead = head1;
    head1 = head1->next;
} else {
    mergedHead = head2;
    head2 = head2->next;
}
```

```
Node* current = mergedHead;
while (head1 != NULL && head2 != NULL) {
    if (head1->data <= head2->data) {
        current->next = head1;
        head1 = head1->next;
    } else {
        current->next = head2;
        head2 = head2->next;
    }
    current = current->next;
}

if (head1 != NULL) {
    current->next = head1;
} else {
    current->next = head2;
}
```

THANK YOU

for YOUR ATTENTION