

MODULE-1

INTRODUCTION TO DATA STRUCTURES

Definition: Types of data structures - Primitive & Non primitive, Linear and Non-linear; Operations on data structures.

Dynamic memory allocation: Static & Dynamic memory allocation; Memory allocation and de-allocation functions - malloc, calloc, realloc and free. Algorithm Specification, Performance Analysis, Performance Measurement.

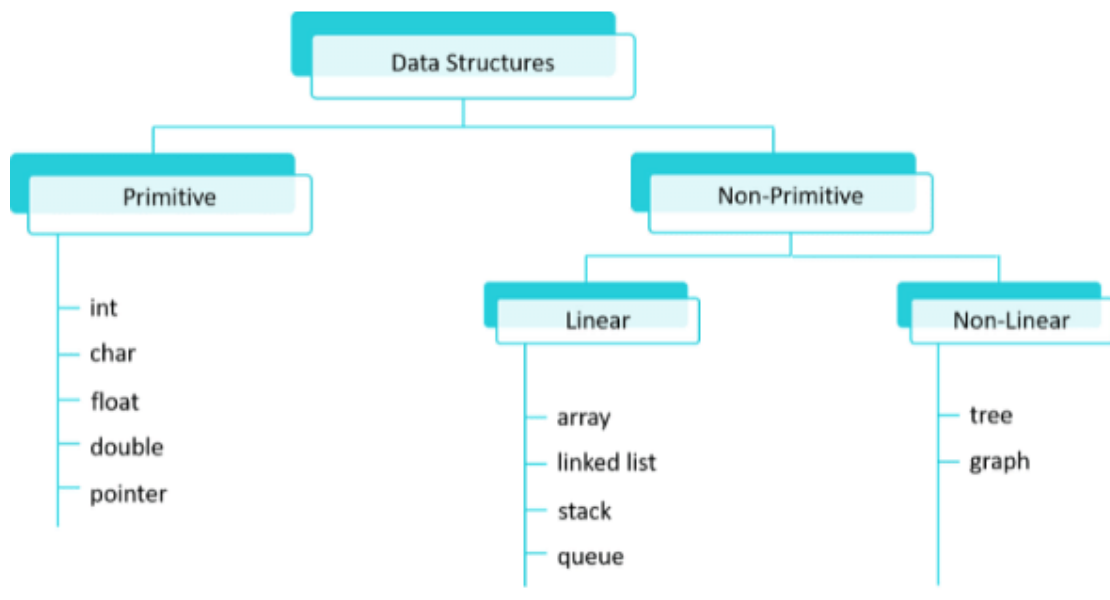
Recursion: Definition; Types of recursions; Recursion Technique Examples - GCD, Factorial, Binomial coefficient nCr , Towers of Hanoi; Comparison between iterative and recursive functions.

Data Structures: A data structure is a specialized format for organizing, storing, and managing data efficiently. It defines the relationship between data elements and the operations that can be performed on them.

A well-designed data structure enhances performance, reduces complexity, and optimizes resource utilization.

Data Structure = Organized Data + Allowed Operations

Classification of Data Structure:



- **Primitive Data Structures**

- The primitive data structures in C are those basic data structures that are already defined in the C language. These data structures can be used to store only a single value. They are the foundation of data manipulation.
- The primitive data structures in C (also known as primitive data types) include int, char, float, double, and pointers.

- **Non-Primitive Data Structures**

- The non-primitive data structures in C are also known as the derived data structures as they are derived from primitive ones. A large number of values can be stored using the non-primitive data structures. The data stored can also be manipulated using various operations like insertion, deletion, searching, sorting, etc.
- The data structures like arrays, trees, stacks, and many more come under this category.

The non-primitive data structures in C can further be divided into two categories:

1. Linear Data Structures

- Linear data structures in C store the data in a sequential or linear fashion.
- The memory location of each element stored can be accessed sequentially. The elements may not be present adjacently in the memory, however, each element is attached to the next element in some way.

Example - arrays, linked lists, stacks, etc.

2. Non-Linear Data Structures

- Non-linear data structures in C store the data in a non-sequential manner.
- The data is stored in multiple levels. The implementation of non-linear data structures is more complex than linear data structures.

Example - graphs, trees.

Operations on Data Structures:

Operations on data structures are fundamental actions that help manage, modify, and utilize the data stored in the structure.

These operations vary slightly depending on the type of data structure, but the **basic operations** are common across most structures.

Here are the main operations:

1. Traversal

- **Definition:**

Traversal means **visiting each element** of a data structure exactly once to process it (e.g., display it, search for an element, or perform a computation).

- **Example:**

- Traversing an array: visiting elements one by one.
- Traversing a linked list: starting from the head and visiting each node.

- **Importance:**

Needed for operations like searching, updating, or displaying the contents.

- **Example Program (Array Traversal in C):**

```
#include<stdio.h>

int main() {

    int arr[] = {10, 20, 30, 40, 50};

    for(int i = 0; i < 5; i++) {

        printf("%d ", arr[i]);

    }

    return 0;

}
```

Output: 10 20 30 40 50

2. Insertion:

- **Definition:**
Insertion refers to **adding a new element** to a data structure at a specific position.
- **Where?**
 - At the beginning, end, or any random position in arrays, linked lists, etc.
 - Pushing an element onto a stack.
- **Importance:**
Used to update data dynamically.
- **Example Program (Array Insertion: in C):**

```
#include <stdio.h>

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int n = 5; // Current number of elements
    int element = 25, position = 2; // Insert 25 at index 2 (3rd position)

    // Shift elements to right
    for (int i = n; i > position; i--) {
        arr[i] = arr[i-1];
    }
    arr[position] = element;
    n++; // Increase size

    printf("Array after insertion:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output:

javascript

Array after insertion:

10 20 25 30 40 50

3. Deletion

- **Definition:**
Deletion means **removing an existing element** from a data structure.
- **Where?**
 - Deleting an element at a specific index in an array.
 - Deleting a node in a linked list.
 - Popping from a stack.
- **Importance:**
Frees memory and manages space efficiently.
- **Example Program (Array Deletion in C):**

```
#include <stdio.h>

int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int n = 5;
    int position = 2; // Delete element at index 2 (30)

    // Shift elements to left
    for (int i = position; i < n-1; i++) {
        arr[i] = arr[i+1];
    }
    n--; // Reduce size
```

```
printf("Array after deletion:\n");
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}

return 0;
}
```

Output:

javascript

```
Array after deletion:
10 20 40 50
```

4. Searching

- **Definition:**
Searching is **finding the location** or **existence** of a particular element in a data structure.
- **Types:**
 - **Linear Search:** Check each element one by one.
 - **Binary Search:** Divide and conquer (only on sorted arrays).
- **Importance:**
Used in applications like finding users, records, or files.

- **Example Program (Array Searching in C):**

```
#include <stdio.h>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;
    int key = 30;
    int found = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Element %d found at index %d\n", key, i);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Element not found\n");
    }

    return 0;
}
```

-

Output:

```
pgsql
```

```
Element 30 found at index 2
```

5. Sorting

- **Definition:**
Sorting arranges elements of a data structure in a **particular order** (ascending or descending).
- **Types of Sorting Algorithms:**
 - **Bubble Sort**

- Selection Sort
 - Insertion Sort
 - Quick Sort
 - Merge Sort
- **Importance:**
Sorting improves the efficiency of other operations like searching (especially binary search).
- **Example Program (Array Searching in C):**

```
#include <stdio.h>

int main() {
    int arr[5] = {40, 10, 30, 20, 50};
    int n = 5;
    int temp;

    // Bubble sort
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
```

```
                // Swap
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output:

php

Sorted array:
10 20 30 40 50

6. Merging

- **Definition:**
Merging means **combining two or more** data structures into one.
- **Example:**
Merging two sorted arrays into one sorted array.
- **Importance:**
Important in data processing, merging sorted files, etc.

- **Example Program(Array Merging)**

```
#include <stdio.h>

int main() {
    int arr1[5] = {10, 30, 50};
    int arr2[5] = {20, 40, 60};
    int merged[10];
    int n1 = 3, n2 = 3, i, j, k;

    // Merging two arrays
    i = j = k = 0;
    while (i < n1 && j < n2) {
        if (arr1[i] < arr2[j])
            merged[k++] = arr1[i++];
        else
            merged[k++] = arr2[j++];
    }
    while (i < n1)
        merged[k++] = arr1[i++];
    while (j < n2)
        merged[k++] = arr2[j++];

    printf("Merged array:\n");
    for (i = 0; i < k; i++) {
        printf("%d ", merged[i]);
    }

    return 0;
}
```

Output:

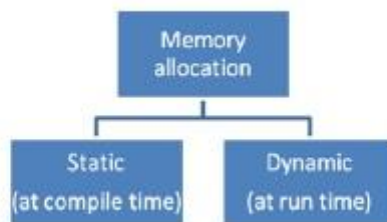
```
php
```

```
Merged array:
```

```
10 20 30 40 50 60
```

Chapter 2:

Dynamic memory allocation: Static & Dynamic memory allocation; Memory allocation and de-allocation functions - malloc, calloc, realloc and free. Algorithm Specification, Performance Analysis, Performance Measurement.



Static Memory Allocation:

Definition: Static memory allocation in C involves allocating memory for variables at compile time, meaning the memory size is fixed before program execution. This approach is straightforward for data structures with known, fixed sizes, such as arrays.

Static variable defines in one block of allocated space, of a fixed size. Once it is allocated, it can never be freed.

Memory is allocated for the declared variable in the program.

- The address can be obtained by using '&' operator and can be assigned to a pointer.
- The memory is allocated during compile time.
- It uses stack for maintaining the static allocation of memory.
- In this allocation, once the memory is allocated, the memory size cannot change.
- It is less efficient.

The final size of a variable is decided before running the program, it will be called as static memory allocation. It is also called compile-time memory allocation.

We can't change the size of a variable which is allocated at compile-time.

Example

Static memory allocation is generally used for an array. Let's take an example program on arrays –

```
#include<stdio.h>
main (){
    int a[5] = {10,20,30,40,50};
    int i;
    printf ("Elements of the array are");
    for ( i=0; i<5; i++)
        printf ("%d, a[i]);
}
```

Output

Elements of the array are

10 20 30 40 50

Advantages of Static Allocation:

- **Simplicity:** Easy to understand and implement, especially for small programs.
- **Speed:** Memory is allocated once at the beginning, so there's no overhead of allocating or deallocating memory during runtime.
- **Efficiency:** No need for dynamic memory management, which can introduce fragmentation and slower performance.

Disadvantages of Static Allocation:

- **Inflexibility:**
The size of the data structure is fixed at compile time, so it can't be changed during program execution.
- **Memory Waste:**
If the allocated memory is larger than needed, it can lead to memory waste.
- **Limited Applicability:**
Not suitable for data structures with dynamically changing sizes or when the size isn't known at compile time.

When to Use Static Allocation:

- **Simple programs:** When the size of data structures is known and fixed, static allocation is a good choice.
- **Arrays and other data structures with fixed sizes:** When the size is known during compilation, static allocation is suitable.
- **Situations where speed and simplicity are prioritized:** Static allocation can be faster and easier to manage compared to dynamic allocation in some cases.

Dynamic memory allocation:

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example : malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    // deallocating the memory
    free(ptr);

    return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

Here, we have dynamically allocated the memory for `n` number of `int`.

C `calloc()`

The name "**calloc**" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of `calloc()`:

```
ptr = (castType*)calloc(n, size);
```

Example 2: calloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

Example 3: realloc()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr, i, n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; ++i)
        printf("%pc\n", ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory:\n");
    for(i = 0; i < n2; ++i)
        printf("%pc\n", ptr + i);

    free(ptr);

    return 0;
}
```


Output

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855476
26855480
26855484
```

Concept of Algorithm Specification

An **algorithm** is a step-by-step procedure to solve a problem. **Algorithm Specification** involves:

1. **Defining the problem**
2. **Input/Output format**
3. **Writing step-by-step logic**
4. **Ensuring clarity, correctness, and efficiency**

Example: Algorithm Specification for Linear Search

Let's specify and implement the **Linear Search** algorithm.

1. Problem Statement:

Given an array of n elements, find the index of a given element (key) using linear search. If the element is not found, return -1.

2. Input:

- An array A of n integers
- An integer key to search for

3. Output:

- Index of the key if found
- -1 if not found

4. Algorithm Steps (Pseudocode):

```
Step 1: Start
Step 2: Read array A and key
Step 3: Repeat for i = 0 to n-1
        If A[i] == key
            Return i
Step 4: If not found, return -1
Step 5: Stop
```

C Program Implementation of the Algorithm:

```
c

#include <stdio.h>

int linearSearch(int A[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (A[i] == key) {
            return i; // return index if found
        }
    }
}

int main() {
    int A[100], n, key, result;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &A[i]);
    }
}
```

```
printf("Enter the key to search: ");
scanf("%d", &key);

result = linearSearch(A, n, key);

if (result == -1) {
    printf("Element not found in the array.\n");
} else {
    printf("Element found at index %d.\n", result);
}

return 0;
}
```

Sample Output:

```
mathematica

Enter number of elements: 5
Enter 5 elements:
10 20 30 40 50
Enter the key to search: 30
Element found at index 2.
```

1. Performance Analysis

Performance Analysis is the **theoretical evaluation** of an algorithm's resource requirements (time and space) as a function of the input size.

Types of Performance Analysis:

a. Time Complexity

- Measures the **amount of time** an algorithm takes to run as a function of input size n .

b. Space Complexity

- Measures the **amount of memory** used by the algorithm during execution.
-

Example 1: Time Complexity of Linear Search

```
#include <stdio.h>
#include <time.h>

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int key = n - 1; // Worst-case scenario: Last element


clock_t start = clock(); // Start timer

int index = linearSearch(arr, n, key);

clock_t end = clock(); // End timer

double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

if (index != -1)
    printf("Key found at index %d\n", index);
else
    printf("Key not found\n");
```



```
    printf("Time taken: %f seconds\n", time_taken);

    return 0;
}
```

◆ Output Example (May vary by system):

```
pgsql

Key found at index 99999
Time taken: 0.003210 seconds
```

Time Complexity Analysis:

- **Best Case:** $O(1)$ → Key is at the first position.
- **Worst Case:** $O(n)$ → Key is at the last position or not present.
- **Average Case:** $O(n/2)$ → Still written as $O(n)$ in Big-O notation.

2. Space Complexity Example (No Extra Space)

We'll use an array and a loop, but no extra space — constant space usage.

```
#include <stdio.h>

int sumArray(int arr[], int n) {
    int sum = 0; // Only one variable used →  $O(1)$ 
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;

    int result = sumArray(arr, n);
}
```

```
int main() {  
    int arr[5] = {10, 20, 30, 40, 50};  
    int n = 5;  
  
    int result = sumArray(arr, n);  
  
    printf("Sum of array elements = %d\n", result);  
  
    return 0;  
}
```

◆ Output:

```
java
```

```
Sum of array elements = 150
```

◆ This program uses:

- Input array: provided by the user.
- Only one extra variable (`sum`) → **O(1) space complexity**.

2. Performance Measurement

Performance Measurement is the **practical observation** of how long an algorithm actually takes during execution on a real machine. It's measured using system functions or benchmarking tools.

In C: Measuring Time Using <time.h>

```
#include <stdio.h>
#include <time.h>

void linearSearch(int A[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (A[i] == key) {
            printf("Found at index %d\n", i);
            return;
        }
    }
    printf("Not found\n");
}
```

```
int main() {
    int A[10000], key = 9999;
    for (int i = 0; i < 10000; i++) A[i] = i;

    clock_t start = clock(); // Start time

    linearSearch(A, 10000, key);

    clock_t end = clock(); // End time

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", time_taken);
    return 0;
}
```

Output Example:

pgsql

Found at index 9999

Time taken: 0.001200 seconds

NOTE:

Performance Analysis: Understand how algorithm behaves as n grows.

Performance Measurement: Evaluate how fast it really runs on a machine.

Both are important for writing efficient programs and choosing the right data structure.

Chapter 3

Recursion: Definition; Types of recursions; Recursion Technique Examples - GCD, Factorial, Binomial coefficient nCr , Towers of Hanoi; Comparison between iterative and recursive functions.

Definition: Recursion is the process by which a function calls itself. C language allows writing of such functions, which call itself to solve complicated problems by breaking them down into simple and easy problems. These functions are known as recursive functions.

What is a Recursive Function in C?

A recursive function in C is a function that calls itself. A recursive function is used when a certain problem is defined in terms of itself. Although it involves iteration, using iterative approach to solve such problems can be tedious. Recursive approach provides a very concise solution to seemingly complex problems.

Syntax

This is how a general recursive function looks like –

```
void recursive_function(){
    recursion(); // function calls itself
}
|
int main(){
    recursive_function();
}
```

Note : While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an [infinite loop](#).

Why Recursion is Used in C?

Recursion is used to perform complex tasks such as [tree](#) and [graph](#) structure traversals. Popular recursive programming solutions include factorial, binary search, tree traversal, tower of Hanoi, eight queens problem in chess, etc.

A recursive program becomes concise, it is not easily comprehensible. Even if the size of the code may reduce, it needs more resources of the processor, as it involves multiple IO calls to the function.

Types of recursions

1. **Direct Recursion:** A function calls itself directly within its definition.
2. **Indirect Recursion:** A function calls another function, which in turn calls the original function, creating a cycle of calls.

Direct Recursion:

C

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1); // Direct recursive call
    }
}
```

Indirect Recursion:

```
#include <stdio.h>

void functionB(int n); // Forward declaration

void functionA(int n) {
    if (n > 0) {
        printf("From A: %d\n", n);
        functionB(n - 1);
    }
}

void functionB(int n) {
    if (n > 0) {
        printf("From B: %d\n", n);
        functionA(n - 1); // Indirect recursive call
    }
}
```

- **Tail Recursion:** The recursive call is the last operation performed in the function. This type of recursion can be optimized by compilers to avoid stack overflow errors.

```
#include <stdio.h>

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

- **Head Recursion:** The recursive call is made at the beginning of the function, before any other operations.

```
#include <stdio.h>

void head_recursive_function(int n) {
    if (n > 0) {
        head_recursive_function(n - 1); // Head recursive call
        printf("%d ", n);
    }
}
```

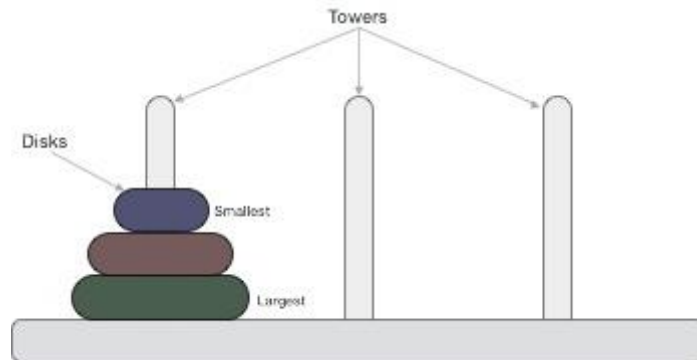
Tree Recursion: The function makes multiple recursive calls within its body.

```
#include <stdio.h>

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2); // Two recursive calls
    }
}
```

Tower of Hanoi

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –



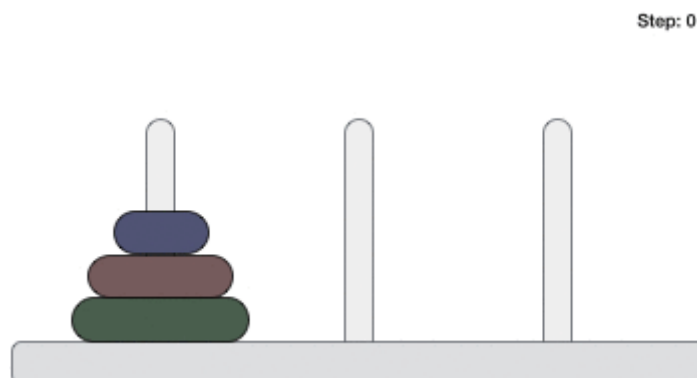
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



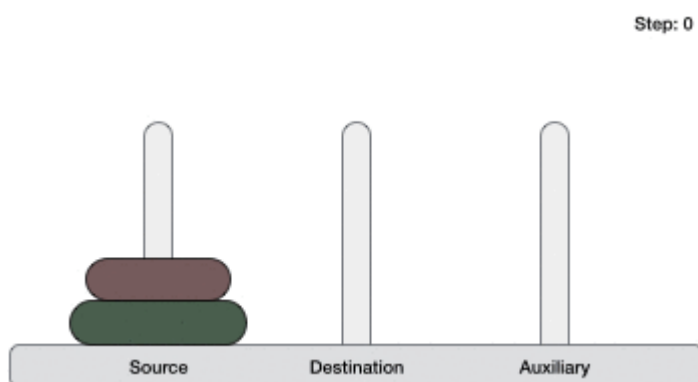
Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps. This presentation shows that a puzzle with 3 disks has taken $2^3 - 1 = 7$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say \rightarrow 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other $(n-1)$ disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other $(n-1)$ disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

The steps to follow are –

```
Step 1 - Move n-1 disks from source to aux
Step 2 - Move nth disk from source to dest
Step 3 - Move n-1 disks from aux to dest
```

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)

    IF disk == 1, THEN
        move disk from source to dest
    ELSE
        Hanoi(disk - 1, source, aux, dest)    // Step 1
        move disk from source to dest        // Step 2
        Hanoi(disk - 1, aux, dest, source)    // Step 3
    END IF

END Procedure
STOP
```

```
#include <stdio.h>
void hanoi(int n, char from, char to, char via) {
    if(n == 1){
        printf("Move disk 1 from %c to %c\n", from, to);
    }
    else{
        hanoi(n-1, from, via, to);
        printf("Move disk %d from %c to %c\n", n, from, to);
        hanoi(n-1, via, to, from);
    }
}
int main() {
    int n = 3;
    char from = 'A';
    char to = 'B';
    char via = 'C';
    //calling hanoi() method
    hanoi(n, from, via, to);
}
```

Output

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

Example 1: GCD (Greatest Common Divisor)

```
#include <stdio.h>

int gcd(int a, int b) {
    if (b == 0)
        return a; // base case
    return gcd(b, a % b); // recursive call
}

int main() {
    printf("GCD of 48 and 18 is %d\n", gcd(48, 18));
    return 0;
}
```

Output:

csharp


```
GCD of 48 and 18 is 6
```

Example 2: Factorial

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0 || n == 1)
        return 1; // base case
    return n * factorial(n - 1); // recursive call
}

int main() {
    printf("Factorial of 5 is %d\n", factorial(5));
    return 0;
}
```



Output:

csharp


Factorial of 5 is 120

Example 3: Binomial Coefficient (nCr)

```
#include <stdio.h>

int nCr(int n, int r) {
    if (r == 0 || r == n)
        return 1; // base case
    return nCr(n - 1, r - 1) + nCr(n - 1, r); // recursive case
}

int main() {
    printf("Value of 5C2 is %d\n", nCr(5, 2));
    return 0;
}
```



Output:

```
csharp
```

```
Value of 5C2 is 10
```

Difference between Recursion and Iteration

The following table highlights all the important differences between recursion and iteration ?

Recursion	Iteration
Recursion uses the selection structure.	Iteration uses the repetition structure.
Infinite recursion occurs if the step in recursion doesn't reduce the problem to a smaller problem. It also becomes infinite recursion if it doesn't convert on a specific condition. This specific condition is known as the base case.	An infinite loop occurs when the condition in the loop doesn't become False ever.
The system crashes when infinite recursion is encountered.	Iteration uses the CPU cycles again and again when an infinite loop occurs.
Recursion terminates when the base case is met.	Iteration terminates when the condition in the loop fails.
Recursion is slower than iteration since it has the overhead of maintaining and updating the stack.	Iteration is quick in comparison to recursion. It doesn't utilize the stack.
Recursion uses more memory in comparison to iteration.	Iteration uses less memory in comparison to recursion.
Recursion reduces the size of the code.	Iteration increases the size of the code.

Conclusion

Recursion uses selection structure and reduces the size of the code. Iteration, on the other hand, the iteration uses repetition structure and increases the size of the code. However, iteration uses less memory as compared to recursion.

