APPLICATIONS OF RECURSIVE FUNCTIONS

RESEARCH :

Recursion is a programming technique where a function calls itself repeatedly until a specific base condition is met. A function that performs such self-calling behavior is known as a **recursive function**, and each instance of the function calling itself is called a **recursive call**.

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Imagine standing between two mirrors facing each other. You see endless reflections of yourself, each one smaller than the last. Similarly, in recursion, the function keeps calling itself until it reaches the final reflection—the base case.

How Does C Recursion Work?

When a recursive function is called in C, here's what happens behind the scenes in simple terms:

- Each function call is added to the call stack – a special memory structure that keeps track of active function calls.
- The function keeps calling itself, and each new call is pushed on top of the stack.
- This continues until the base case is reached. No more recursive calls are made now.

Types of Recursion in C

The following are the types of recursion in C language with examples:

1. Direct Recursion

When the recursion functions in C call themselves directly, it is known as direct recursion.

Explanation:

The function directRecursion() is directly calling itself with a smaller value.

2. Indirect Recursion

When a function calls another function, and that function again calls the first function, it's called indirect recursion.

Explanation:

Here, functionA() and functionB() call each other recursively, forming an indirect recursion loop.

3. Tail Recursion

A recursion is called tail recursion if the recursive call is the last statement executed by the function.

Explanation:

Since nothing is done after the recursive call, this is a tail-recursive function. Tail recursion can be optimized by the compiler.

4. Head Recursion

If the recursive call happens before any other processing in the function, it is known as head recursion.

Explanation:

Here, the function first calls itself, and only after the recursion ends, it processes the printf().

5. Tree Recursion

When a function calls itself more than once in each invocation, it results in multiple branches, forming a tree-like structure.

SOURCE :

1. www.geeksforgeeks.org
2. https://www.researchgate.net/publication/389235736_Recursion_in_c_on_Integers
3. Width formula reference : https://www.geeksforgeeks.org/dsa/program-to-print-full-pyramid-pattern-star-pattern/

ANALYSE :

To implement recursive programming type in following patterns :

1. Recursive Triangle Pattern
2. Recursive Diamond Pattern
3. Recursive Number Pyramid Pattern
4. Recursive Hourglass Pattern

   Etc.

I have made an **HOLLOW HOURGLASS PATTERN** by using recursive functions .

ALGORITHM :

- Step 1:
  Start the program

- Step 2:

  Declare function printspaces to print the number of spaces using recursion

- Step 3:

  Declare function upperhalf(i, n) to print the upper inverted hollow triangle using recursion

- Step 4:

  Declare function lowerhalf(i, n) to print the lower upright hollow triangle using recursion

- Step 5:

  In main function, set n = 10 (or any size of hourglass)

- Step 6:

  Call upperhalf(1, n) to print the upper hollow half of the hourglass

- Step 7:

  Call lowerhalf(1, n) to print the lower hollow half of the hourglass

- Step 8:

  In printspaces, if n equals 0, return. Else print one space and call printspaces(n - 1)

- Step 9:

  In upperhalf(i, n), if i > n, return

  Call printspaces(i - 1)

  Calculate width = 2 × (n − i) + 1

  For j = 1 to width, do:

  If i is 1, print *

  Else if j is 1 or j is width, print *

  Else print space

  Print newline

  Call upperhalf(i + 1, n)

- Step 10:

  In lowerhalf(i, n), if i > n, return

  Call printspaces(n − i)

  Calculate width = 2 × i − 1

  For j = 1 to width, do:

  If i is n, print *

  Else if j is 1 or j is width, print *

  Else print space

  Print newline

  Call lowerhalf(i + 1, n)

- Step 11:

  End the program

BUILD :

```
#include <stdio.h>



void printspaces(int n) {
    if (n == 0) return;
    printf(" ");
    printspaces(n - 1);
}



void upperhalf(int i, int n) {
    if (i > n) return;

    printspaces(i - 1);
    int width = 2 * (n - i) + 1;
```

```
    for (int j = 1; j <= width; j++) {
        if (i == 1)
            printf("*");
        else if (j == 1 || j == width)
            printf("*");
        else
            printf(" ");
    }
    printf("\n");

    upperhalf(i + 1, n);
}


void lowerhalf(int i, int n) {
    if (i > n) return;

    printspaces(n - i);
    int width = 2 * i - 1;
```

```
    for (int j = 1; j <= width; j++) {
        if (i == n)
            printf("*");
        else if (j == 1 || j == width)
            printf("*");
        else
            printf(" ");
    }
    printf("\n");

    lowerhalf(i + 1, n);
}

int main() {
    int n = 10;

    printf("\nHOLLOW HOURGLASS PATTERN \n\n");
```

```
    upperhalf(1, n);
    lowerhalf(1, n);

    return 0;
}
```

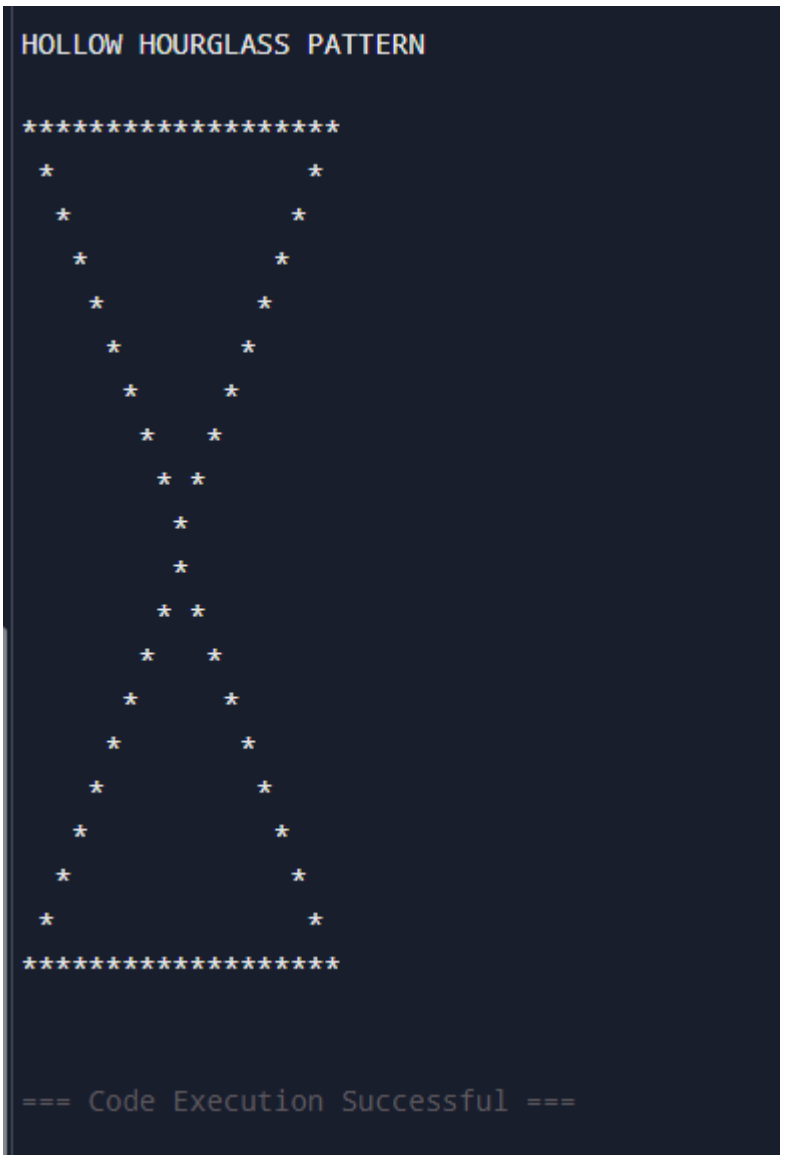
TESTING :

HOLLOW HOURGLASS PATTERN

```
**********
 *      *
  *    *
   *  *
    **
     *
```

```
      *
     * *
    *   *
   *     *
  *       *
 *         *
*           *
*******************
```

=== Code Execution Successful ===



IMPLEMENTATION :