CS 2124: DATA STRUCTURES
Spring 2024

4th Lecture (Part – II)

Topics: **Recursion**

Topics

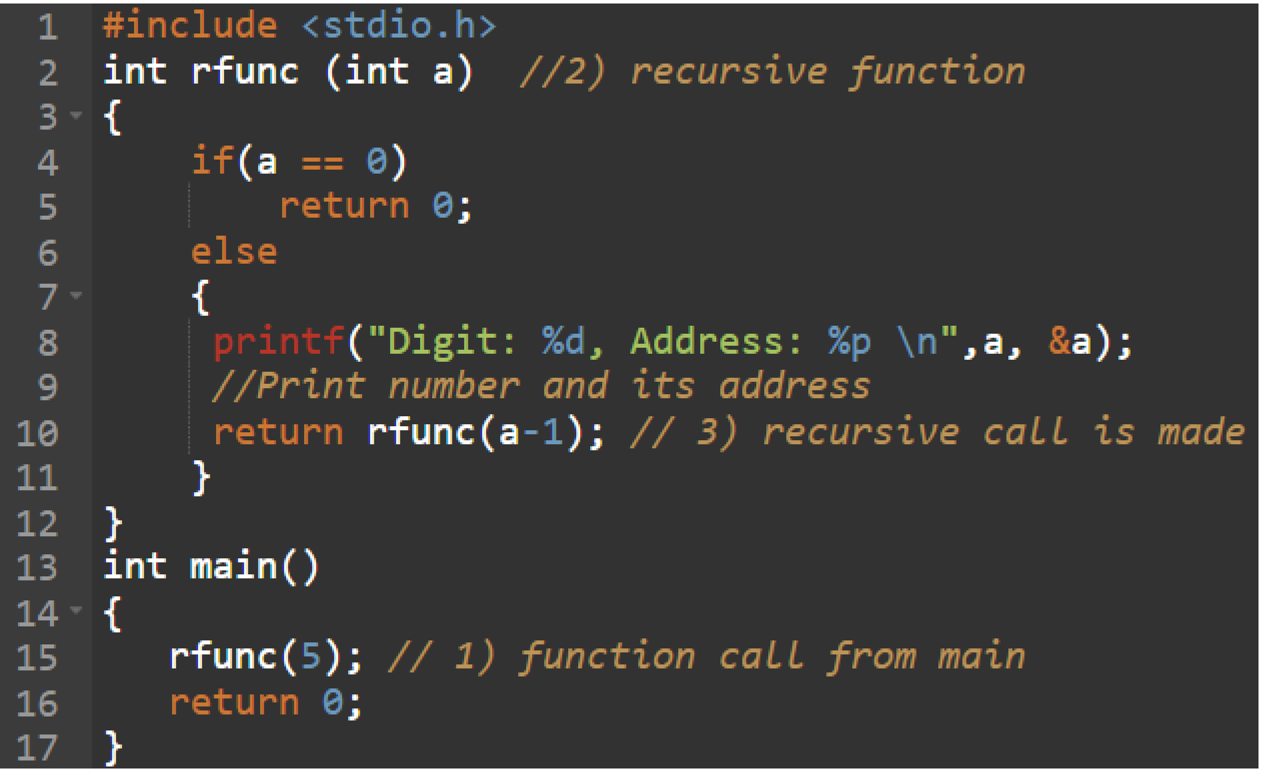
- Assignment – 2 (Any Questions)
- Mid-Term Exam (Discussion)
- Recursion
    - Recursion (Properties)
    - Recursion (Types)
- Recursion vs Iteration
    - Example using Factorial
    - Example using Fibonacci Sequence
- Recursion (Memory)
- Recursion (Advantages and Disadvantages)
- Recursion (Real world examples)
- Binary Search (Using Recursion and Iteration)
- Towers of Hanoi

Midterm Exams

- Data Structure (Midterm Exam – In person) – Thursday, 29th Feb
  - Exam will be on Canvas
  - Attendance is compulsory
  - Location: NPB 1.226
  - Timing:

Section			**Time/ NPB 1.226**	**Students**
CS 2124 - 0C1	CS 2124-0CA	36734	10:00 – 10:30	30
	CS 2124-0CB	36736	10:40 – 11:10	29
CS 2124 - 0D4	CS 2124-0DA	36738	11:30 – 12:00	30
	CS 2124-0DB	36739	12:10 – 12:40	30
CS 2124 - 0E1	CS 2124-0EA	42879	01:30 – 02:00	30
	CS 2124-0EB	42880	02:10 – 02:40	30

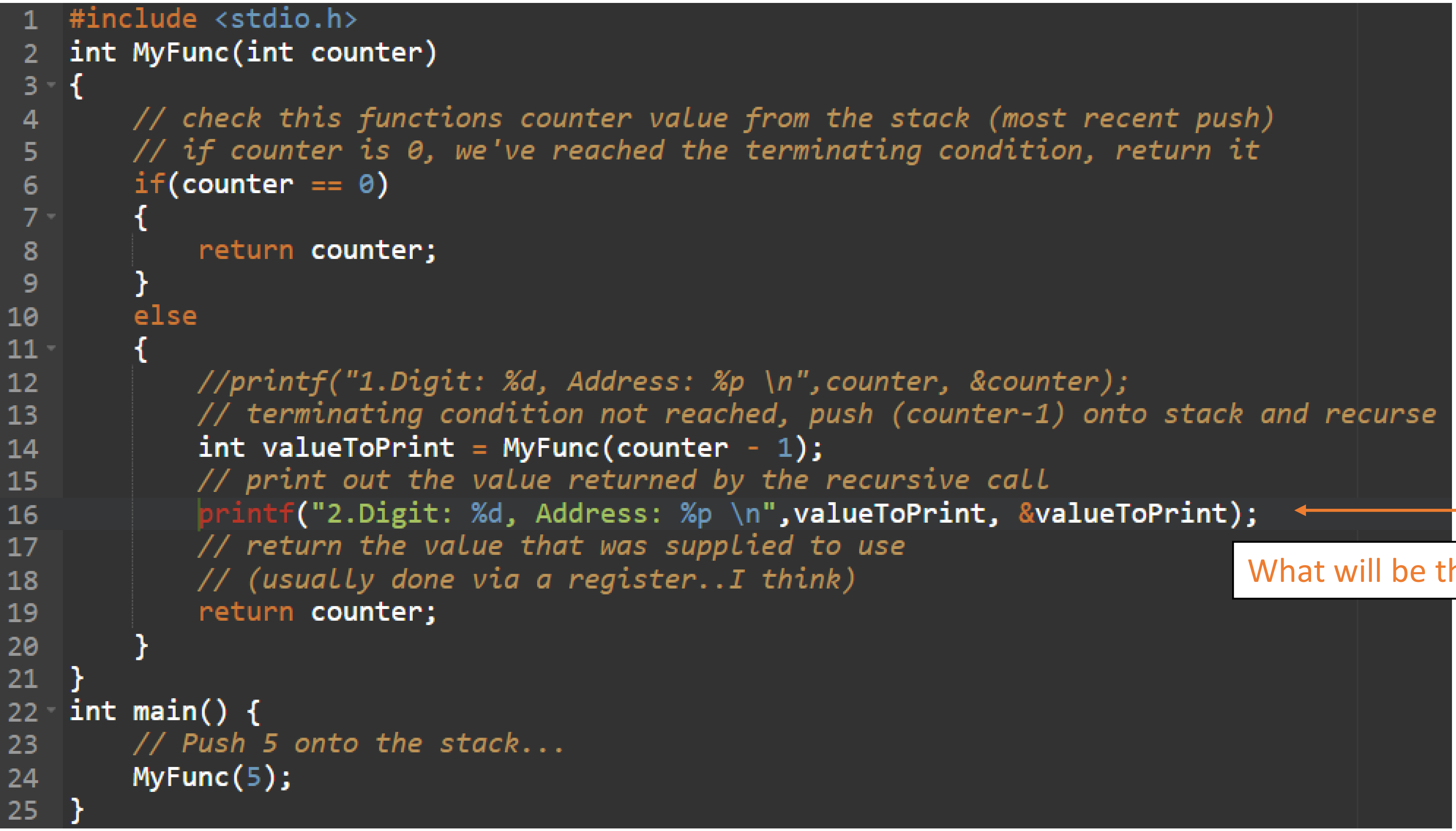
Recursion (Memory)



```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
     printf("Digit: %d, Address: %p \n",a, &a);
     //Print number and its address
     return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main
   return 0;
}
```

1. The first call to the function rfunc() having value a=5 will be a copy on the bottom of the stack, and it is also the copy that will return at the end.
2. Meanwhile, the rfunc() will call another occurrence of the same function but with 1 subtracted, i.e., a=4.
3. Each time a new occurrence is called, it is stored at the top of the stack, which goes on until the condition is satisfied.
4. As the condition is unsatisfied, i.e., a=0, there will be no further calls, and each function copy stored in the stack will start to return its respected values, and the function will now terminate.

Recursion (Memory)

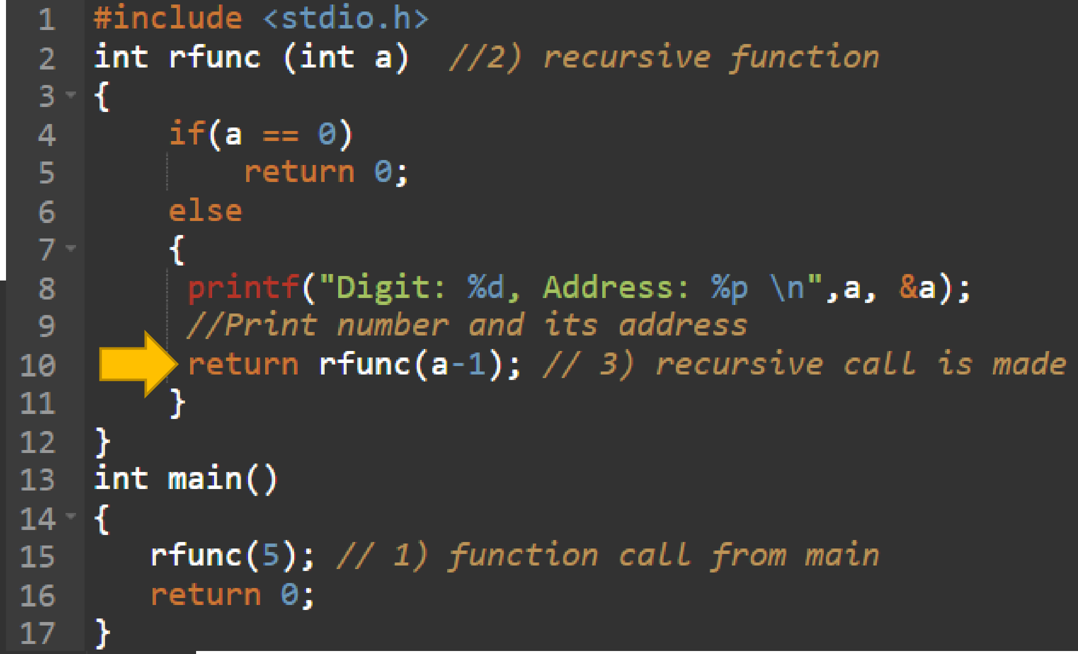


```c
#include <stdio.h>
int MyFunc(int counter)
{
    // check this functions counter value from the stack (most recent push)
    // if counter is 0, we've reached the terminating condition, return it
    if(counter == 0)
    {
        return counter;
    }
    else
    {
        //printf("1.Digit: %d, Address: %p \n",counter, &counter);
        // terminating condition not reached, push (counter-1) onto stack and recurse
        int valueToPrint = MyFunc(counter - 1);
        // print out the value returned by the recursive call
        printf("2.Digit: %d, Address: %p \n",valueToPrint, &valueToPrint);
        // return the value that was supplied to use
        // (usually done via a register..I think)
        return counter;
    }
}
int main() {
    // Push 5 onto the stack...
    MyFunc(5);
}
```

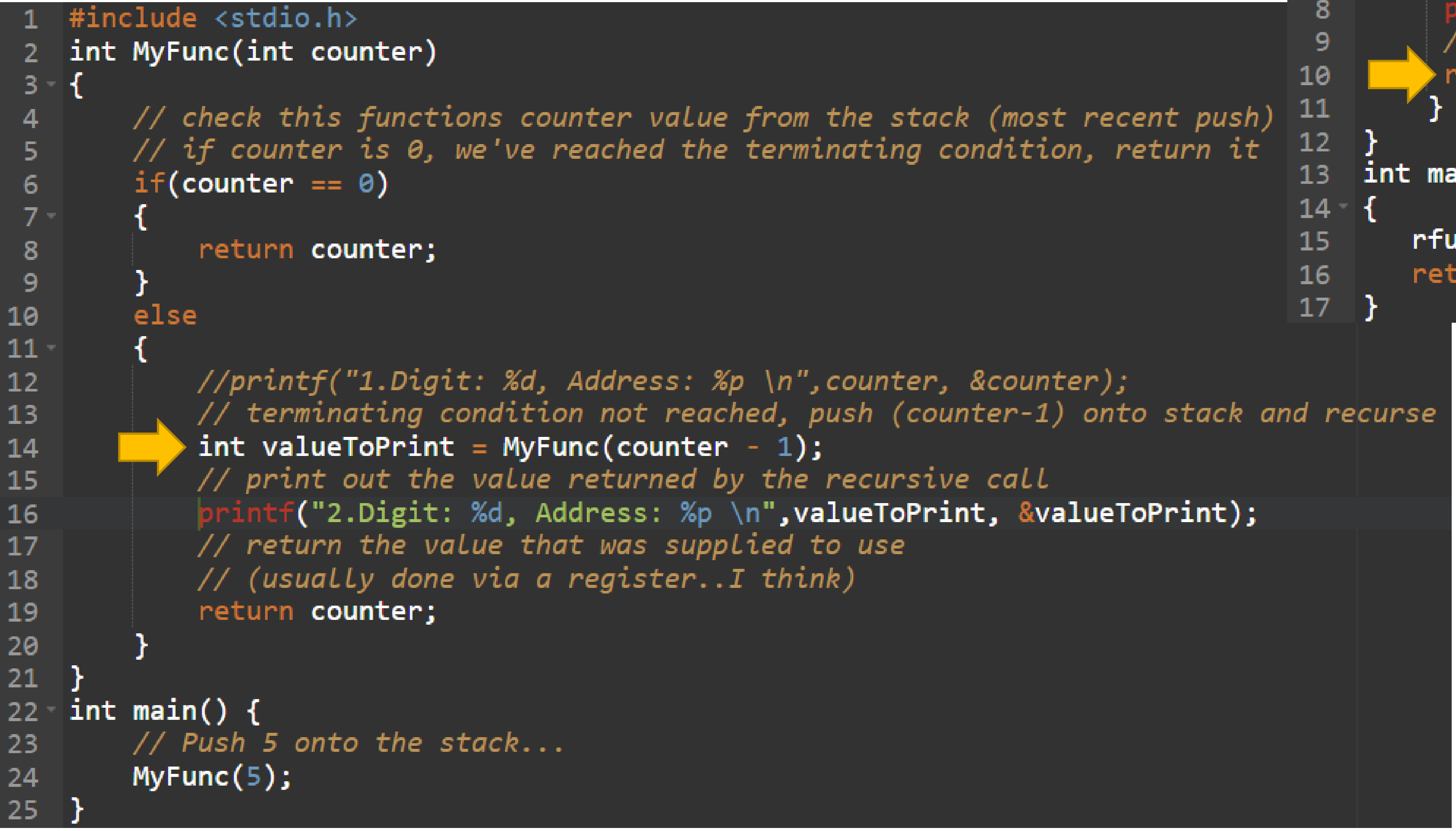
What will be the Sequence of Digit output

Note: This code will not be part of any quiz or exam. It is only for implementation / understanding

Recursion (Memory)

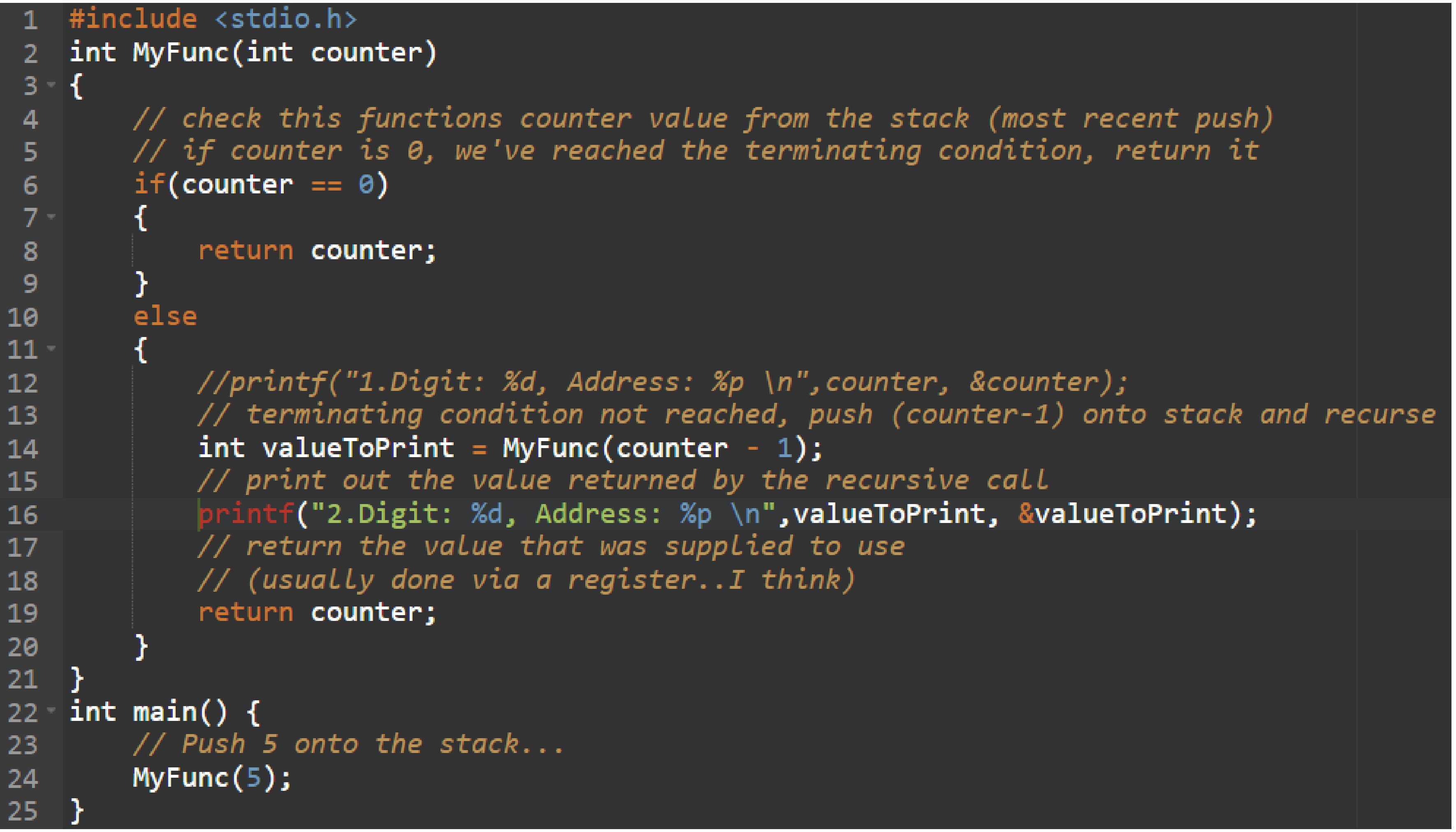


```c
#include <stdio.h>
int rfunc (int a)  //2) recursive function
{
    if(a == 0)
        return 0;
    else
    {
     printf("Digit: %d, Address: %p \n",a, &a);
     //Print number and its address
     return rfunc(a-1); // 3) recursive call is made
    }
}
int main()
{
   rfunc(5); // 1) function call from main
   return 0;
}
```



```c
#include <stdio.h>
int MyFunc(int counter)
{
    // check this functions counter value from the stack (most recent push)
    // if counter is 0, we've reached the terminating condition, return it
    if(counter == 0)
    {
        return counter;
    }
    else
    {
        //printf("1.Digit: %d, Address: %p \n",counter, &counter);
        // terminating condition not reached, push (counter-1) onto stack and recurse
        int valueToPrint = MyFunc(counter - 1);
        // print out the value returned by the recursive call
        printf("2.Digit: %d, Address: %p \n",valueToPrint, &valueToPrint);
        // return the value that was supplied to use
        // (usually done via a register..I think)
        return counter;
    }
}
int main() {
    // Push 5 onto the stack...
    MyFunc(5);
}
```

Recursion (Memory)



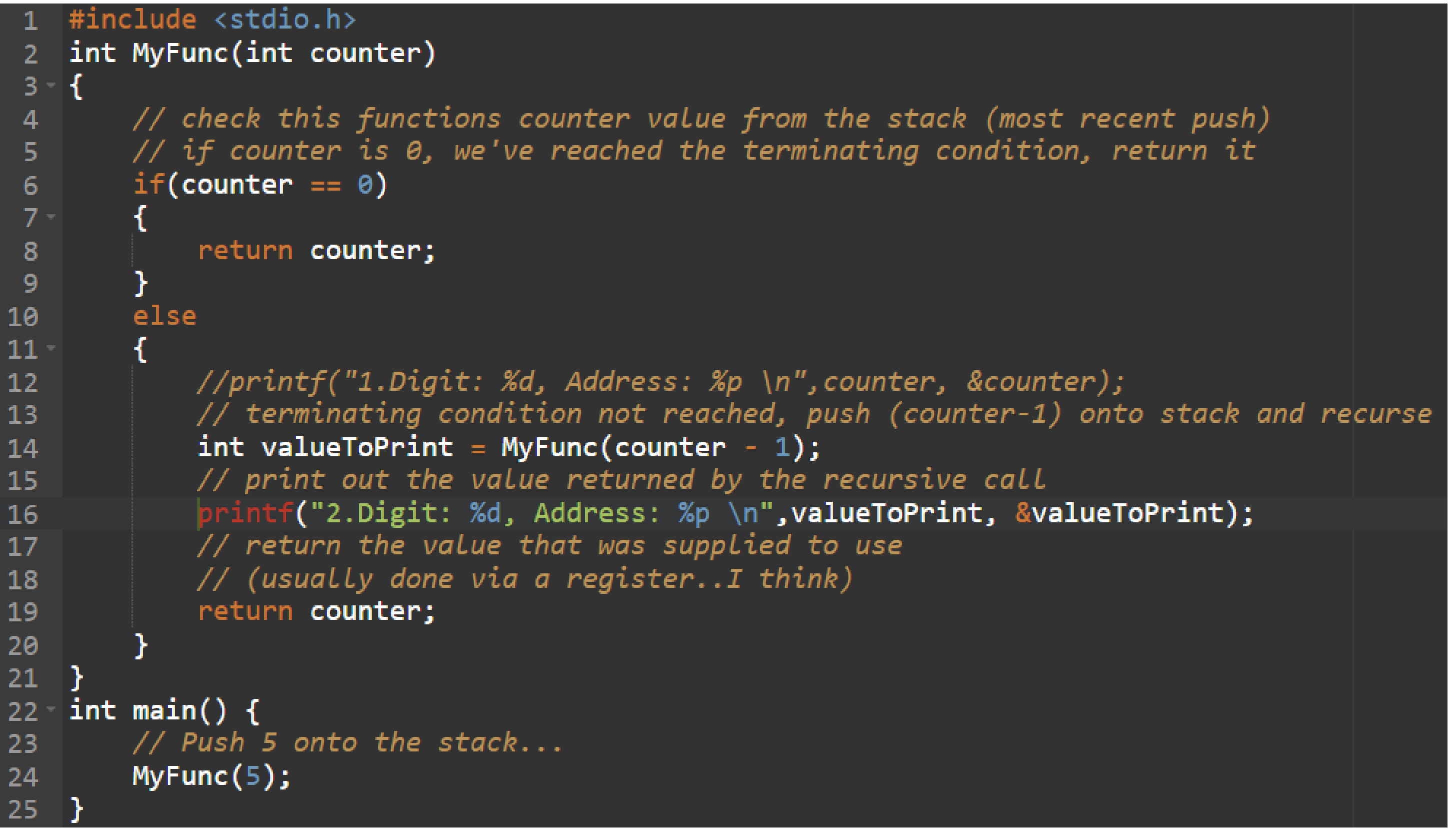
```c
#include <stdio.h>
int MyFunc(int counter)
{
    // check this functions counter value from the stack (most recent push)
    // if counter is 0, we've reached the terminating condition, return it
    if(counter == 0)
    {
        return counter;
    }
    else
    {
        //printf("1.Digit: %d, Address: %p \n",counter, &counter);
        // terminating condition not reached, push (counter-1) onto stack and recurse
        int valueToPrint = MyFunc(counter - 1);
        // print out the value returned by the recursive call
        printf("2.Digit: %d, Address: %p \n",valueToPrint, &valueToPrint);
        // return the value that was supplied to use
        // (usually done via a register..I think)
        return counter;
    }
}
int main() {
    // Push 5 onto the stack...
    MyFunc(5);
}
```

The first time through MyFunc, count is 5. It fails the terminating check (it is not 0), so the recursive call is invoked, with (counter -1), 4.
This repeats, decrementing the value pushed onto the stack each time until counter == 0.
At this point, the terminating clause fires and the function simply returns the value of counter (0), usually in a register.

Note: This code will not be part of any quiz or exam. It is only for implementation / understanding

Continue >>>

Recursion (Memory)



```c
#include <stdio.h>
int MyFunc(int counter)
{
    // check this functions counter value from the stack (most recent push)
    // if counter is 0, we've reached the terminating condition, return it
    if(counter == 0)
    {
        return counter;
    }
    else
    {
        //printf("1.Digit: %d, Address: %p \n",counter, &counter);
        // terminating condition not reached, push (counter-1) onto stack and recurse
        int valueToPrint = MyFunc(counter - 1);
        // print out the value returned by the recursive call
        printf("2.Digit: %d, Address: %p \n",valueToPrint, &valueToPrint);
        // return the value that was supplied to use
        // (usually done via a register..I think)
        return counter;
    }
}
int main() {
    // Push 5 onto the stack...
    MyFunc(5);
}
```

The next call up the stack, uses the returned value to print (0), then returns the value that was supplied into it when it was called (1).
This repeats:

- The next call up the stack, uses the returned value to print (1), then returns the value that was supplied into it when it was called (2). etc, till you get to the top of the stack.

Note: This code will not be part of any quiz or exam. It is only for implementation / understanding

Continue >>>

Recursion (Memory)

So, if MyFunc was **invoked with 3 (Code on next slide)**, you'd get the equivalent of (ignoring return addresses etc from the stack):

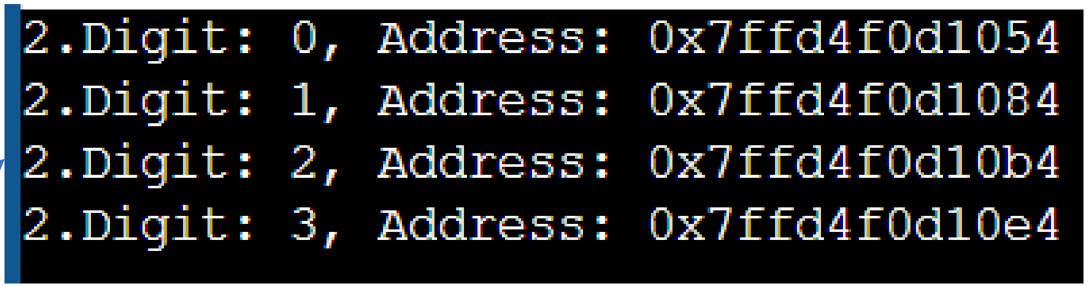
- Call MyFunc(3) Stack: [3]
- Call MyFunc(2) Stack: [2,3]
- Call MyFunc(1) Stack: [1,2,3]
- Call MyFunc(0) Stack: [0,1,2,3]
- Termination fires (top of stack == 0), return top of stack(0).

// Flow returns to:
MyFunc(1) Stack: [1,2,3]
Print returned value (0) and address
return current top of stack (1)

// Flow returns to:
MyFunc(2) Stack: [2,3]
Print returned value (1) and address
return current top of stack (2)

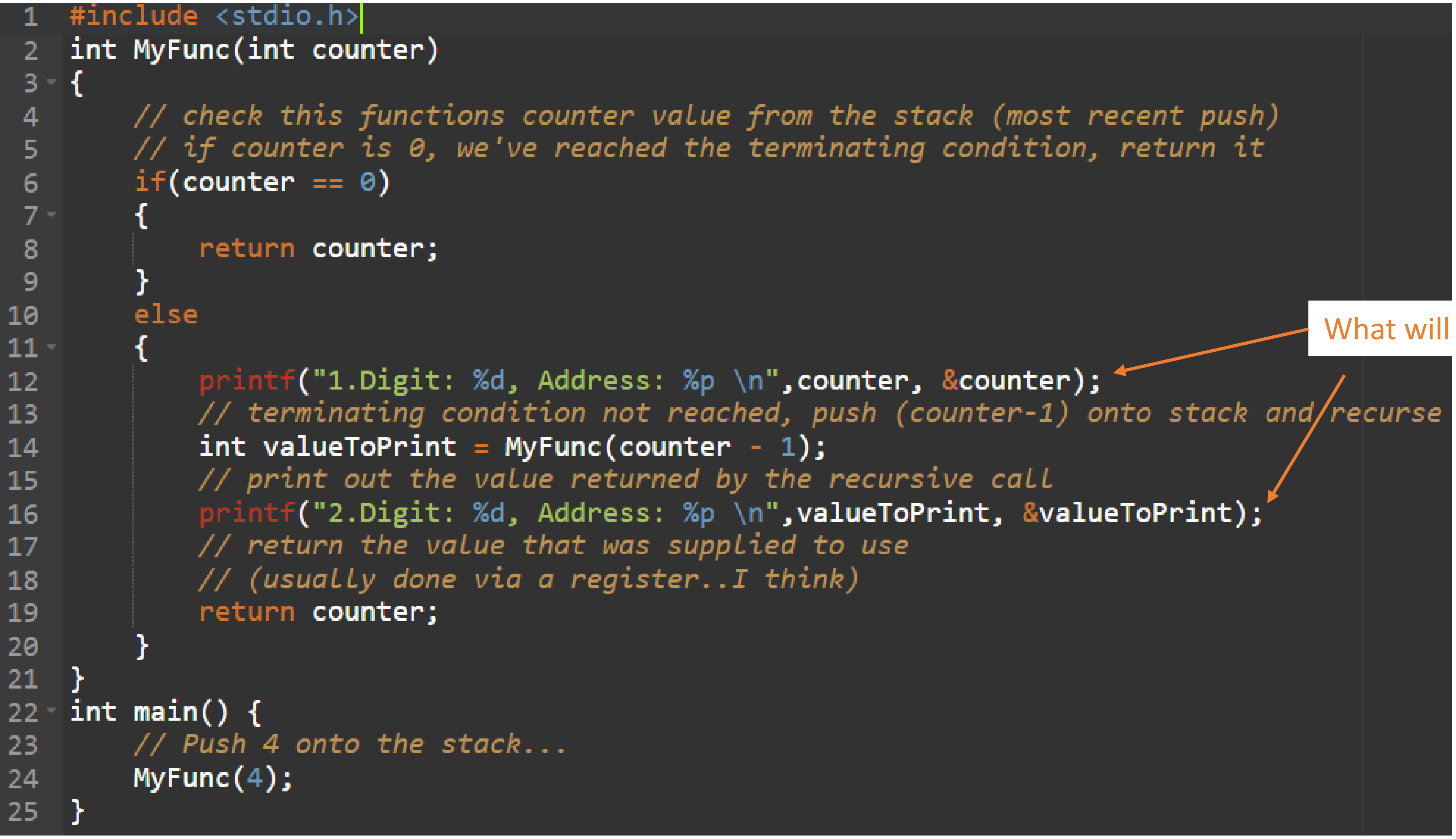
// Flow returns to:
MyFunc(3) Stack: [3]
Print returned value (2) and address
return current top of stack (3)

// Flow returns to:
MyFunc(4) Stack: [3]
Print returned value (3) and address
return current top of stack (-1)



```
2.Digit: 0, Address: 0x7ffd4f0d1054
2.Digit: 1, Address: 0x7ffd4f0d1084
2.Digit: 2, Address: 0x7ffd4f0d10b4
2.Digit: 3, Address: 0x7ffd4f0d10e4
```

Recursion (Memory)



```c
#include <stdio.h>
int MyFunc(int counter)
{
    // check this functions counter value from the stack (most recent push)
    // if counter is 0, we've reached the terminating condition, return it
    if(counter == 0)
    {
        return counter;
    }
    else
    {
        printf("1.Digit: %d, Address: %p \n",counter, &counter);
        // terminating condition not reached, push (counter-1) onto stack and recurse
        int valueToPrint = MyFunc(counter - 1);
        // print out the value returned by the recursive call
        printf("2.Digit: %d, Address: %p \n",valueToPrint, &valueToPrint);
        // return the value that was supplied to use
        // (usually done via a register..I think)
        return counter;
    }
}
int main() {
    // Push 4 onto the stack...
    MyFunc(4);
}
```

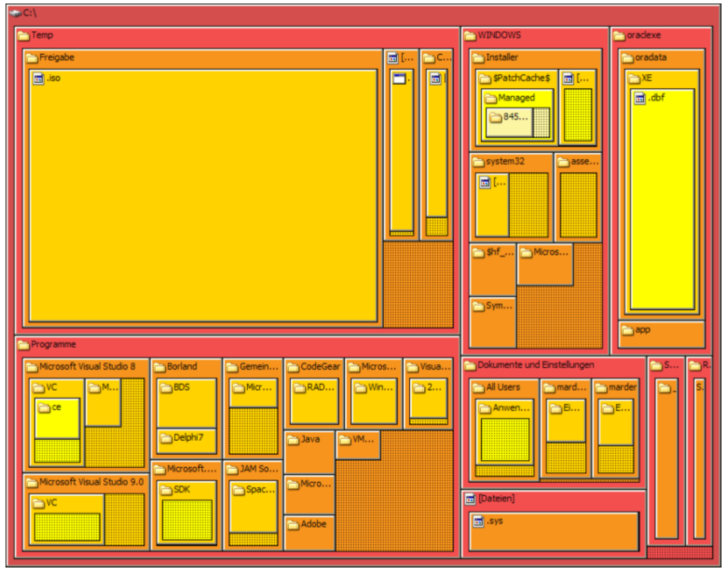
What will be the Sequence of Digits output

Note: This code will not be part of any quiz or exam. It is only for implementation / understanding

Source: Link

Recursion (Example)

- The green Dots that you are able to see in the above image are generated with the help of recursions.
- Here recursion is used to generate all possible safe and unsafe moves of a particular piece of the chess game.
- Generally, the safe moves are represented with the help of a Green dot and Red for the unsafe ones.



- Imagine that you want to find a file on your machine. You don't want to look for it manually, you can write a function to find it for you.

- How do you approach this?

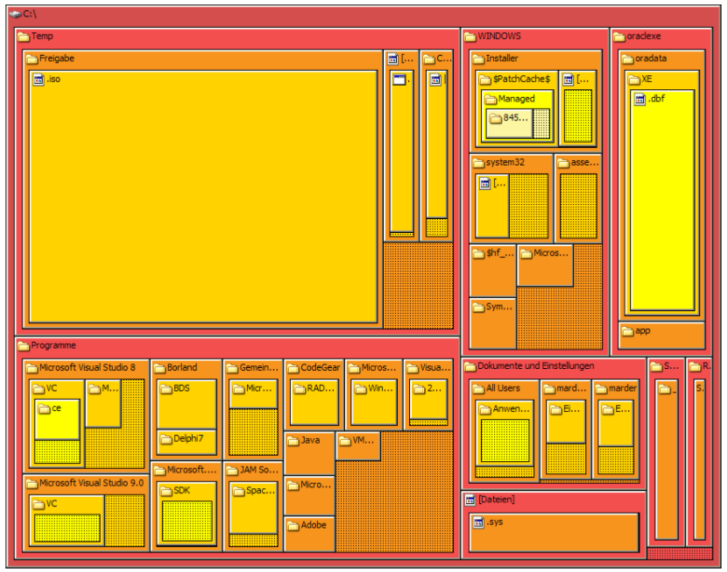
Source: Link & Link

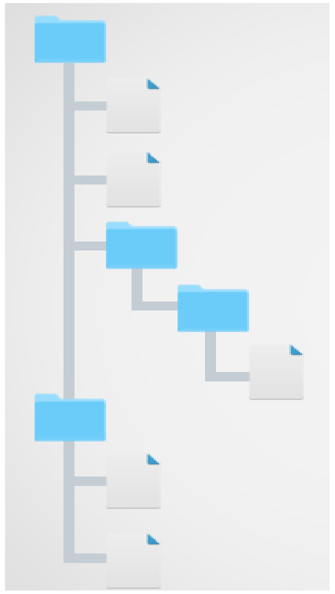
Recursion (Example)

- Imagine that you want to find a file on your machine. You don't want to look for it manually, and you figure this is a good exercise anyway, so you're going to write a function to find it for you.

- How do you approach this?

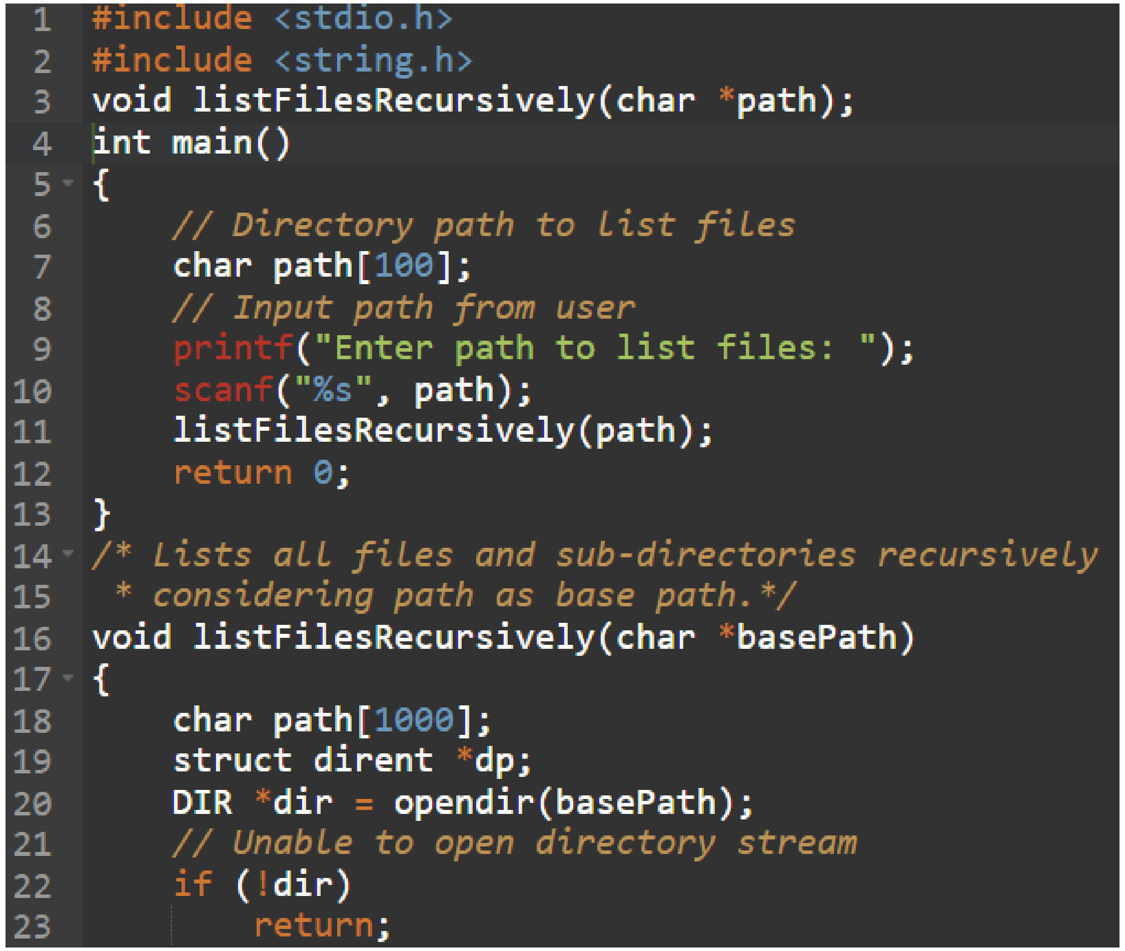
1. We'll start with the root directory.
2. Then we need to pick one of the sub-folder and look inside (i.e. tree data structure).
3. That sub-folder might have its own sub-folder, so we have to go deeper and deeper until there are no more sub-folders.
4. Then we go back and try one of the other sub-folders.



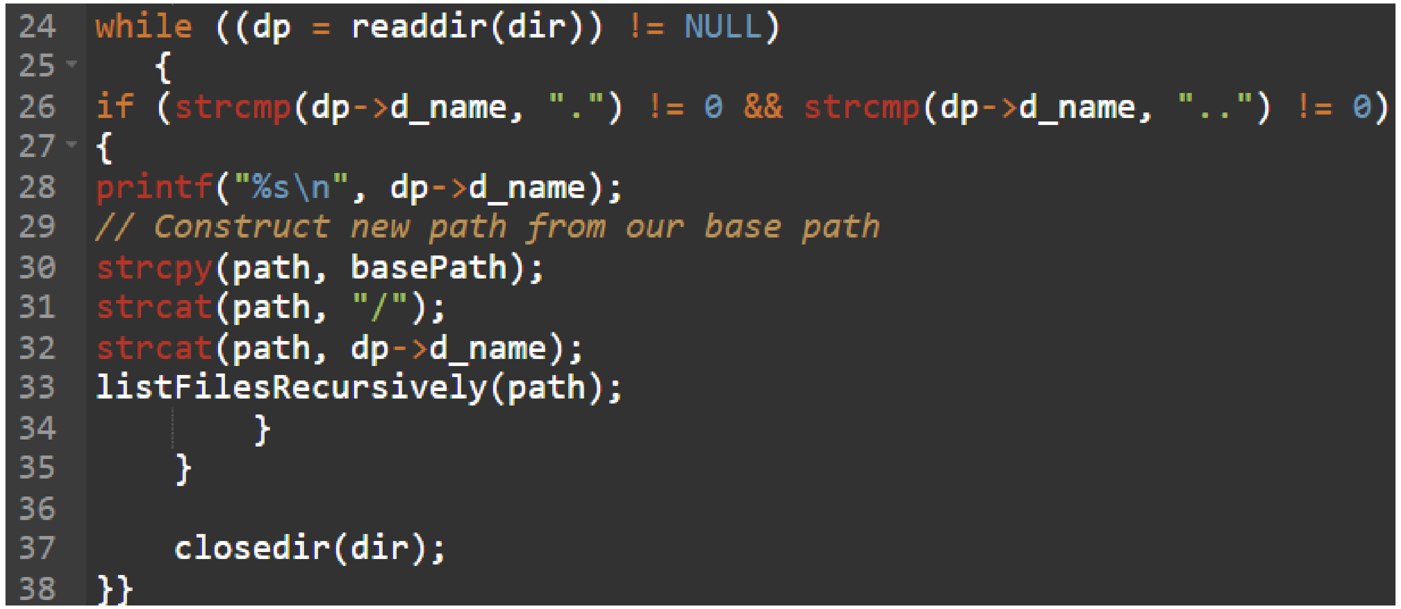


Source: Link & Link

Recursion (Example – File Search)



```c
#include <stdio.h>
#include <string.h>
void listFilesRecursively(char *path);
int main()
{
    // Directory path to list files
    char path[100];
    // Input path from user
    printf("Enter path to list files: ");
    scanf("%s", path);
    listFilesRecursively(path);
    return 0;
}
/* Lists all files and sub-directories recursively
 * considering path as base path.*/
void listFilesRecursively(char *basePath)
{
    char path[1000];
    struct dirent *dp;
    DIR *dir = opendir(basePath);
    // Unable to open directory stream
    if (!dir)
        return;
```



```c
while ((dp = readdir(dir)) != NULL)
   {
if (strcmp(dp->d_name, ".") != 0 && strcmp(dp->d_name, "..") != 0)
{
printf("%s\n", dp->d_name);
// Construct new path from our base path
strcpy(path, basePath);
strcat(path, "/");
strcat(path, dp->d_name);
listFilesRecursively(path);
        }
    }

    closedir(dir);
}}
```

Note: This code will not be part of any quiz or exam. It is only for implementation / understanding

Source

Binary Search (Revision)

- The binary search algorithm works by comparing the element to be searched by the middle element of the array and based on this comparison follows the required procedure.

1. Case 1 : element = middle, the element is found return the index.

2. Case 2 : element > middle, search for the element in the sub-array starting from middle+1 index to n.

3. Case 3 : element < middle, search for element in the sub-array starting from 0 index to middle -1.

Binary Search (Recursion vs Iteration)

- **Iteration Method**
- Do until the pointers low and high meet each other.

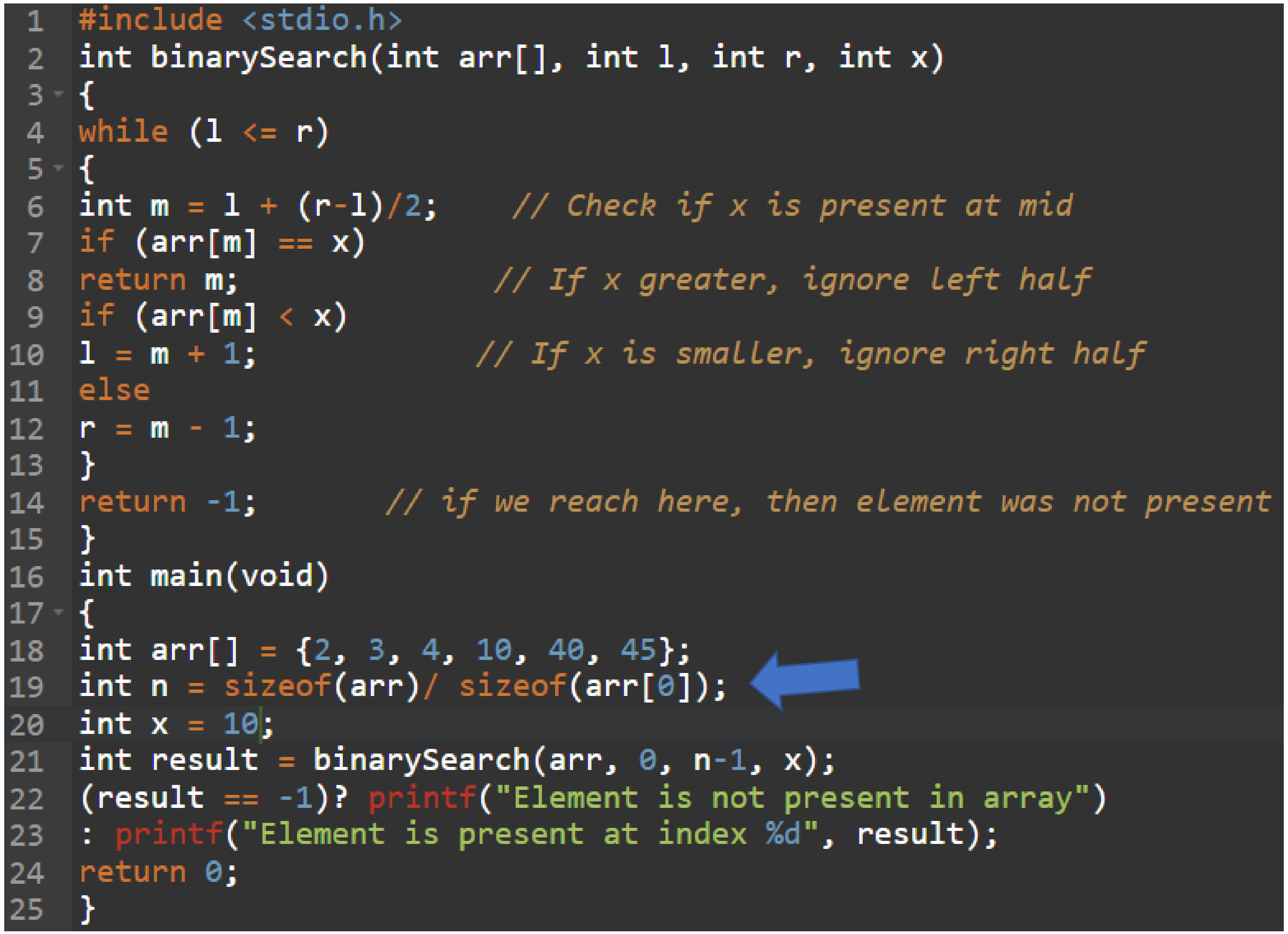
```
1.    mid = (low + high)/2
2.    if (x == arr[mid])
3.        return mid
4.    else if (x > arr[mid]) // x is on the right side
5.        low = mid + 1
6.    else                   // x is on the left side
7.        high = mid - 1
```

- **Recursive Method**

```
1.  binarySearch(arr, x, low, high)
2.      if low > high
3.          return False
4.      else
5.          mid = (low + high) / 2
6.          if x == arr[mid]
7.              return mid
8.          else if x > arr[mid]       // x is on the right side
9.              return binarySearch(arr, x, mid + 1, high)
10.         else                        // x is on the left side
11.             return binarySearch(arr, x, low, mid - 1)
```

- **Iterative call** is looping over the same block of code multiple times
- **Recursive call** is calling the same function again and again.

Binary Search (Iterative)

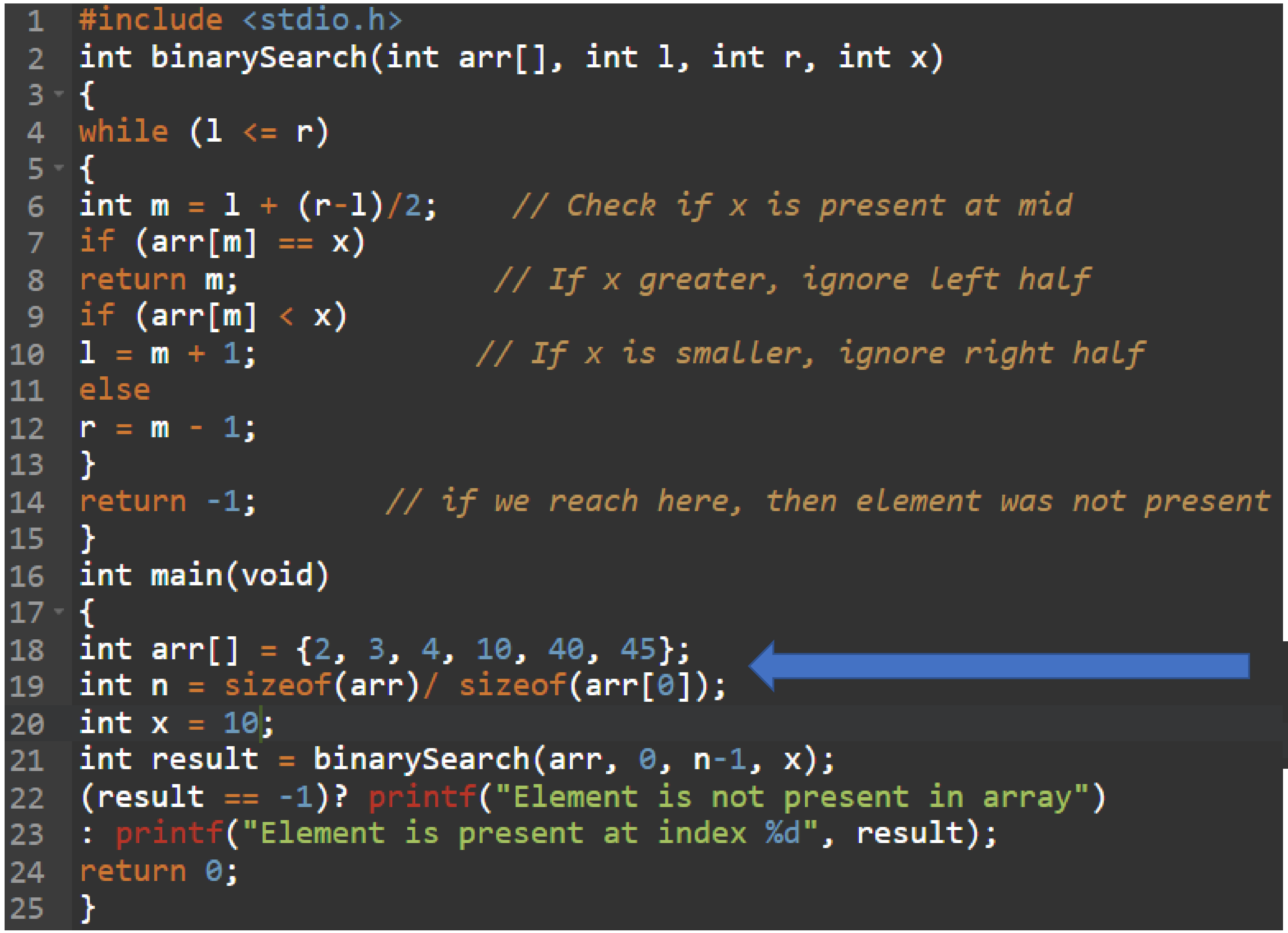


```
1  #include <stdio.h>
2  int binarySearch(int arr[], int l, int r, int x)
3  {
4  while (l <= r)
5  {
6  int m = l + (r-l)/2;     // Check if x is present at mid
7  if (arr[m] == x)
8  return m;               // If x greater, ignore left half
9  if (arr[m] < x)
10 l = m + 1;             // If x is smaller, ignore right half
11 else
12 r = m - 1;
13 }
14 return -1;         // if we reach here, then element was not present
15 }
16 int main(void)
17 {
18 int arr[] = {2, 3, 4, 10, 40, 45};
19 int n = sizeof(arr)/ sizeof(arr[0]);
20 int x = 10;
21 int result = binarySearch(arr, 0, n-1, x);
22 (result == -1)? printf("Element is not present in array")
23 : printf("Element is present at index %d", result);
24 return 0;
25 }
```

- Line 19 ??

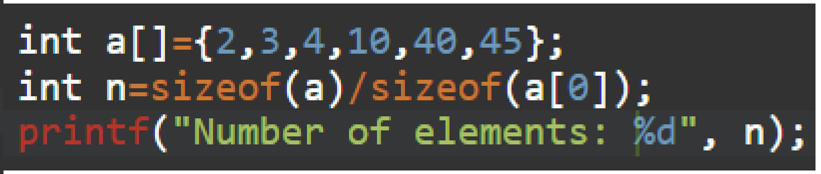
Try to implement and compute CPU cycle

Binary Search (Iterative)



```c
1  #include <stdio.h>
2  int binarySearch(int arr[], int l, int r, int x)
3  {
4  while (l <= r)
5  {
6  int m = l + (r-l)/2;     // Check if x is present at mid
7  if (arr[m] == x)
8  return m;               // If x greater, ignore left half
9  if (arr[m] < x)
10 l = m + 1;              // If x is smaller, ignore right half
11 else
12 r = m - 1;
13 }
14 return -1;         // if we reach here, then element was not present
15 }
16 int main(void)
17 {
18 int arr[] = {2, 3, 4, 10, 40, 45};
19 int n = sizeof(arr)/ sizeof(arr[0]);
20 int x = 10;
21 int result = binarySearch(arr, 0, n-1, x);
22 (result == -1)? printf("Element is not present in array")
23 : printf("Element is present at index %d", result);
24 return 0;
25 }
```

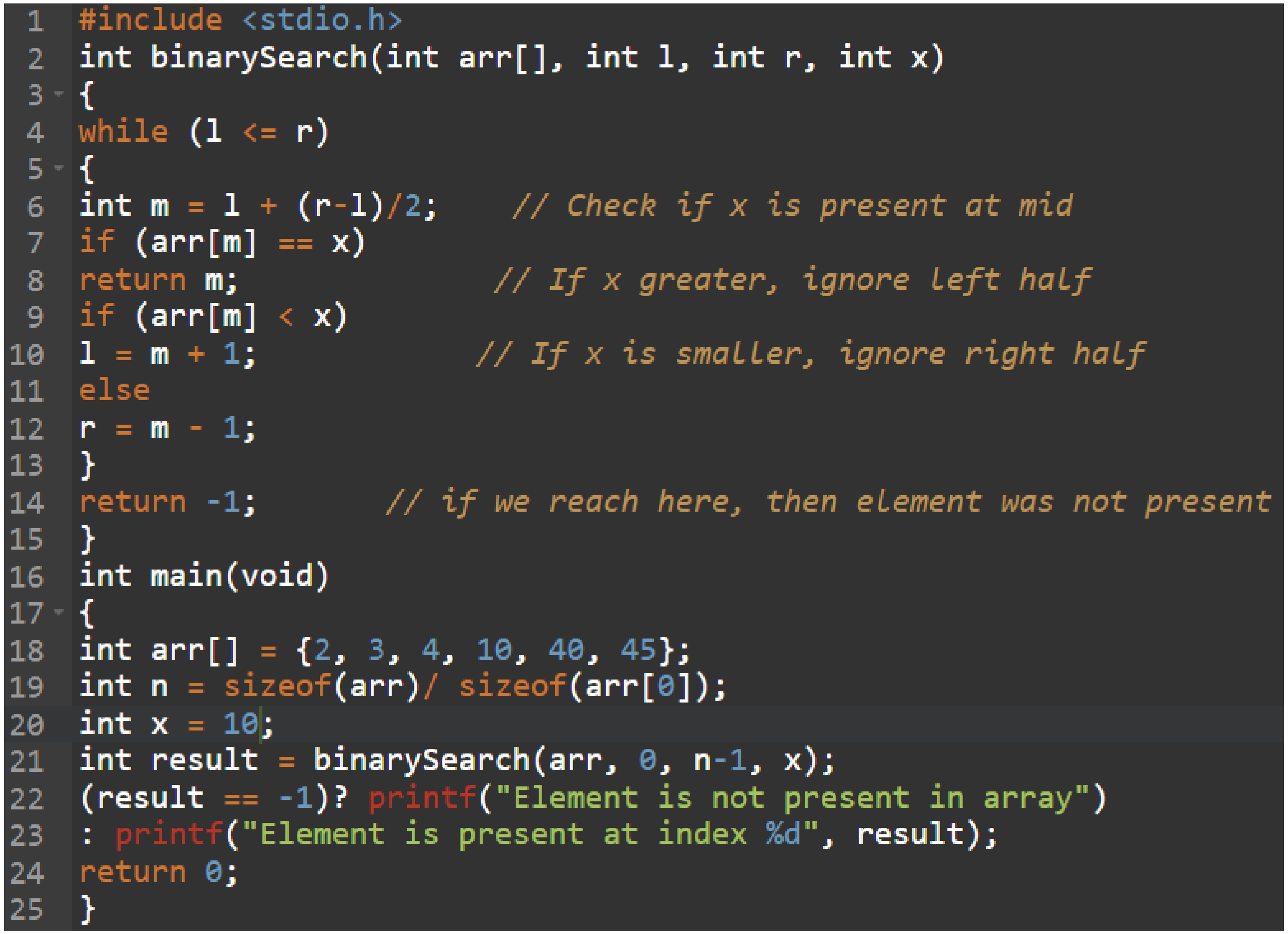
- <u>Line 19:</u>
- If you have an array then sizeof(array) returns the number of bytes the array occupies.
- Since each element can take more than 1 byte of space, you have to divide the result with the size of one element (sizeof(array[0])).
- This gives you number of elements in the array.



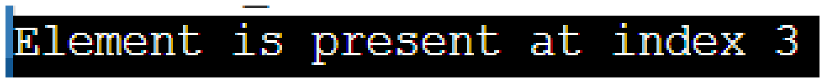
```c
int a[]={2,3,4,10,40,45};
int n=sizeof(a)/sizeof(a[0]);
printf("Number of elements: %d", n);
```

Try to implement and compute CPU cycle

Binary Search (Iterative)



```
#include <stdio.h>
int binarySearch(int arr[], int l, int r, int x)
{
while (l <= r)
{
int m = l + (r-l)/2;     // Check if x is present at mid
if (arr[m] == x)
return m;               // If x greater, ignore left half
if (arr[m] < x)
l = m + 1;             // If x is smaller, ignore right half
else
r = m - 1;
}
return -1;          // if we reach here, then element was not present
}
int main(void)
{
int arr[] = {2, 3, 4, 10, 40, 45};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n-1, x);
(result == -1)? printf("Element is not present in array")
: printf("Element is present at index %d", result);
return 0;
}
```



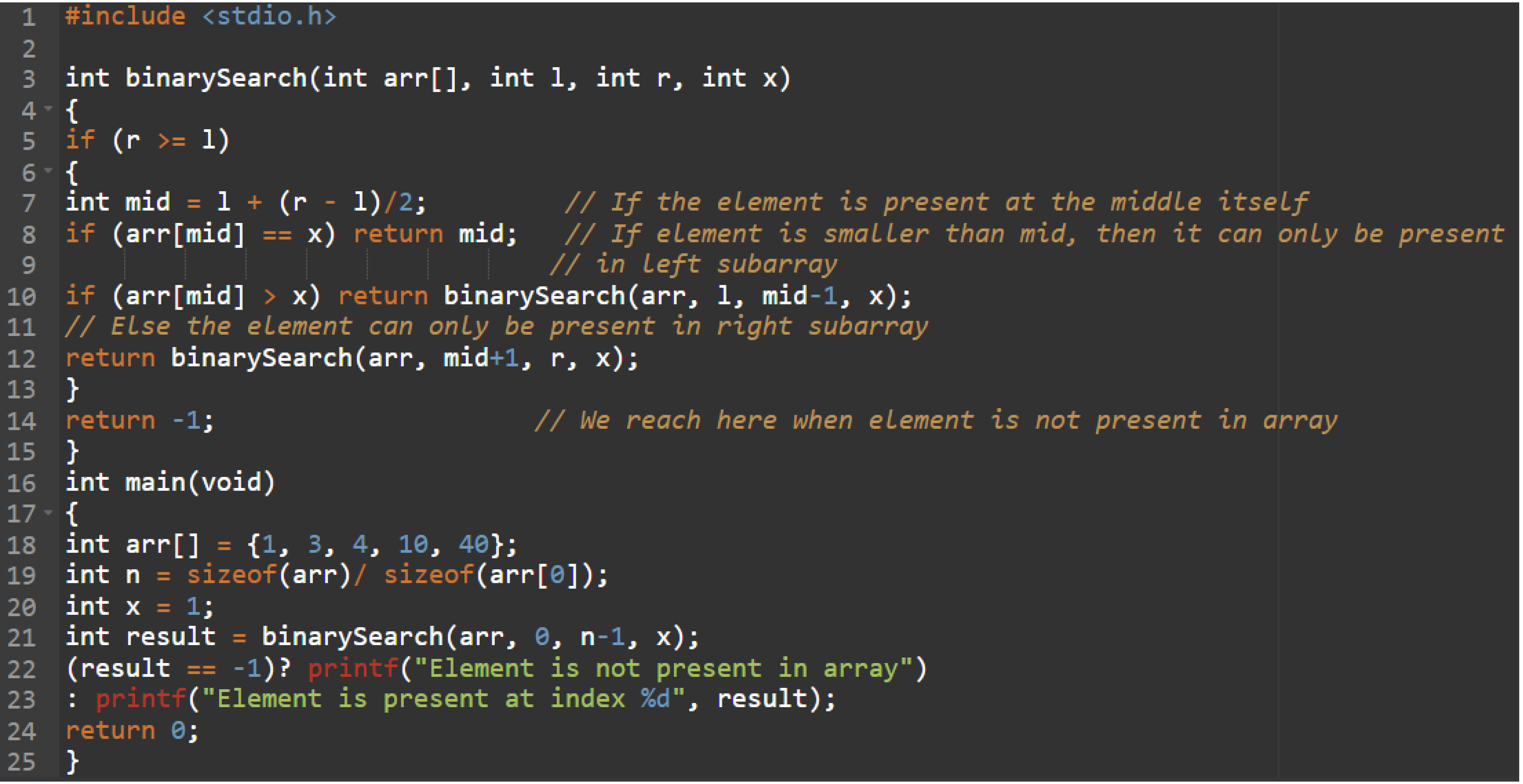
```
Element is present at index 3
```

2	3	4	10	40	45
[0]	[1]	[2]	[3]	[4]	[5]

Try to implement and compute CPU cycle

Binary Search (Recursion)

Try to implement and compute CPU cycle



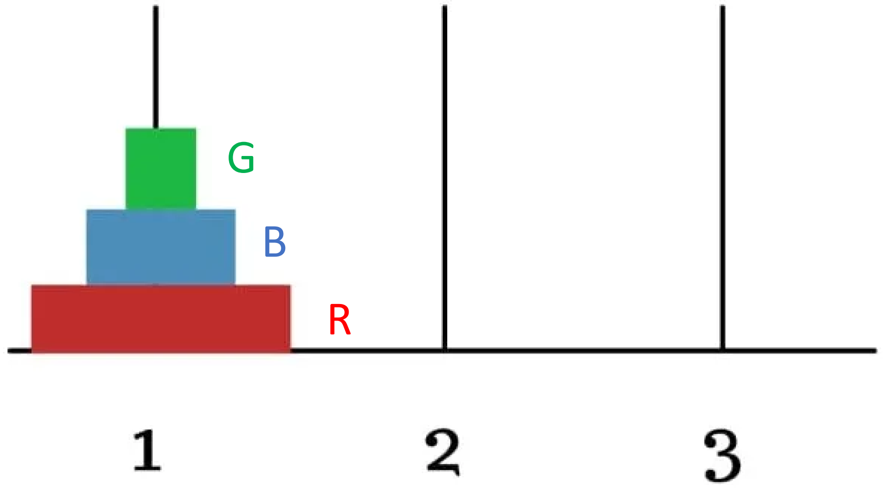
```c
#include <stdio.h>

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);
}
return -1;                     // We reach here when element is not present in array
}
int main(void)
{
int arr[] = {1, 3, 4, 10, 40};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 1;
int result = binarySearch(arr, 0, n-1, x);
(result == -1)? printf("Element is not present in array")
: printf("Element is present at index %d", result);
return 0;
}
```

Towers of Hanoi

- **Rules:**

1. Start with n rings on rod 1 (n = 3 i.e. G, B, R)
2. Our goal is to move them all rings to rod 3
3. Can only move 1 ring at a time
4. Larger rings cannot be placed on smaller rings

- How many steps to solve the game?

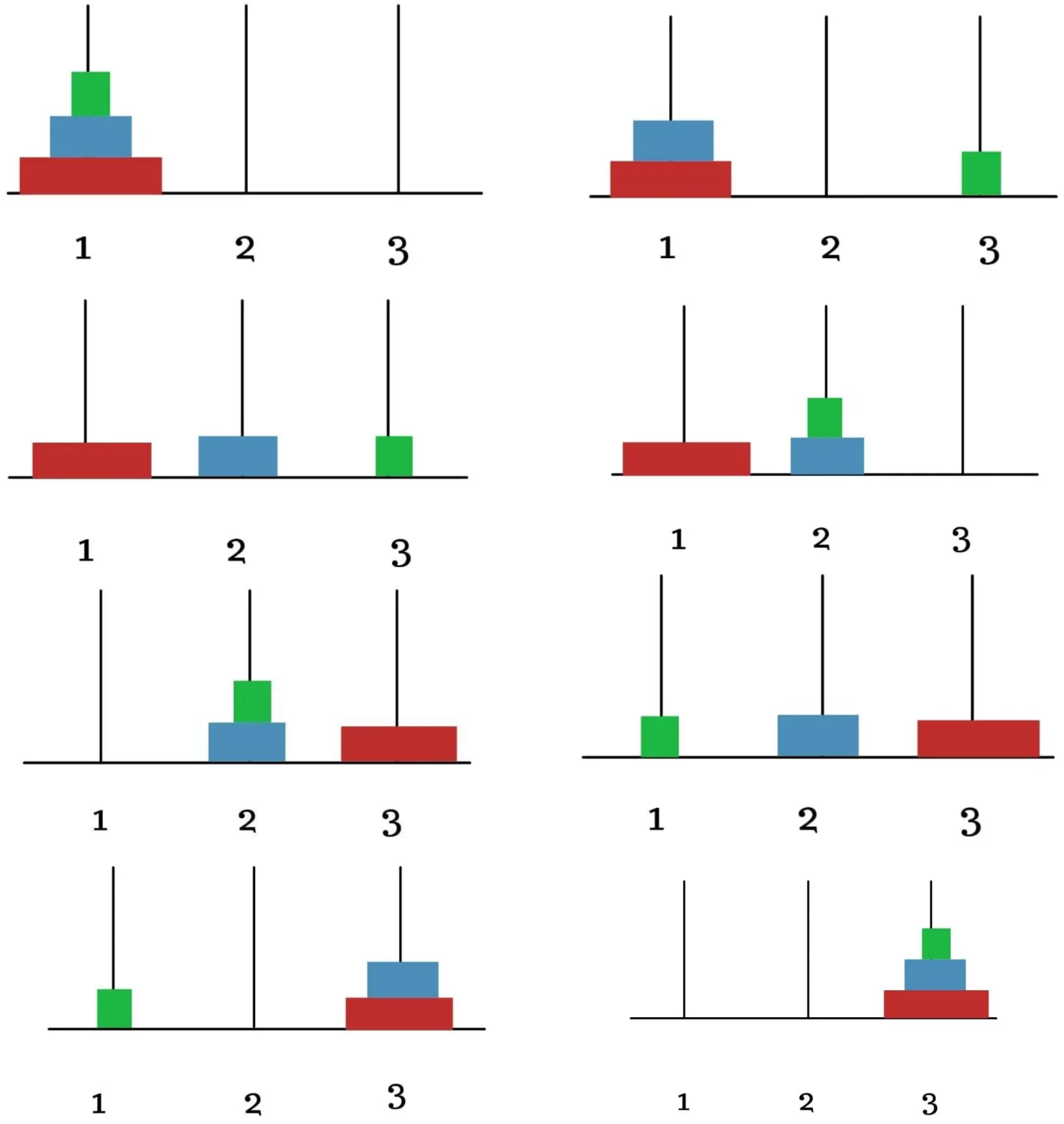


Towers of Hanoi

- Rules:

1. Start with n rings on rod 1 (n = 3 i.e. G, B, R)
2. Our goal is to move them all rings to rod 3
3. Can only move 1 ring at a time
4. Larger rings cannot be placed on smaller rings

- How many steps to solve the game?
- Total Steps: 7

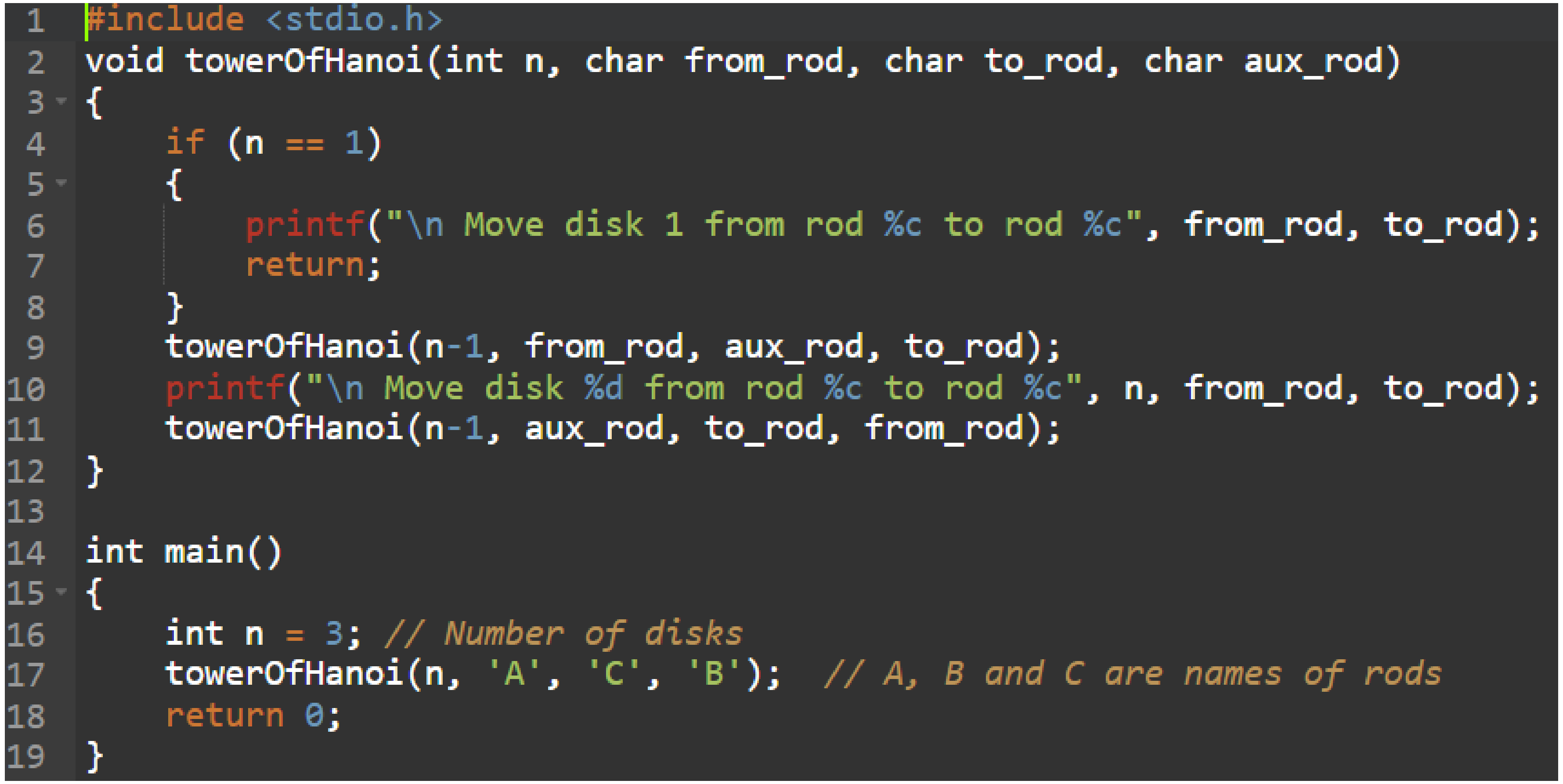


Towers of Hanoi

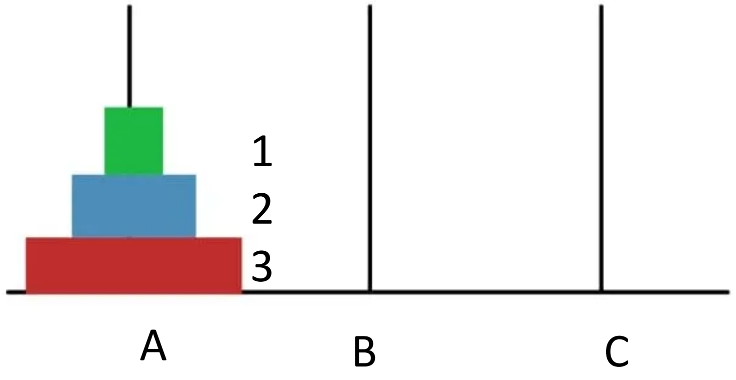
- Rods: A = Source ; B = Auxiliary, C = Destination
- If **1 disk** = Simply move the 1 disk from source to destination (A -> C)
- If **2 disks** = Move the disks to destination with the help of auxiliary rode (A -> B, A -> C, B -> C).
- If **3 disks** = First move n-1 disk to Auxiliary with the help of Destination (Divide and conquer approach)
    a) i.e. Move disk 1 and disk 2 to Auxiliary - B from Source – A
        i. Move disk 1 from A -> C
        ii. Move disk 2 from A -> B
        iii. Move disk 1 from C -> B
    b) Now Source – A has 1 disk and Destination – C has no disk. We can simply move disk 3 from A -> C.
    c) Next we have to move the disks from Auxiliary – B to Destination – C with the help of Source – A.
        i. Move disk 1 from B -> A
        ii. Move disk 2 from B -> C
        iii. Move disk 1 from A -> C
- For n = 1 simply move disk from Source to Destination
- For n > 1 move disks from Source to Destination with the help of Auxiliary

Towers of Hanoi

Will the following Program solve the game or not?

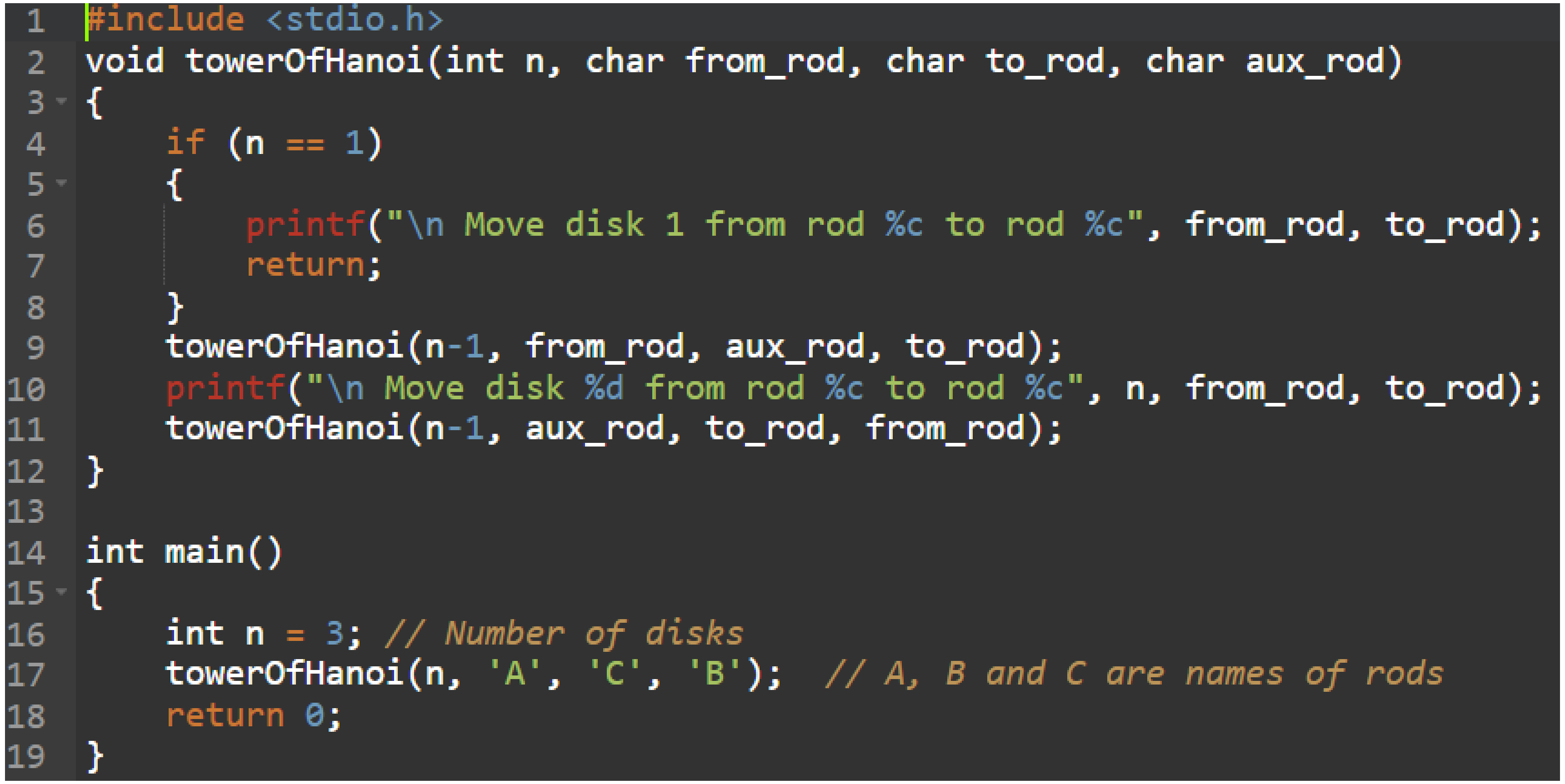


```c
#include <stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 3; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```



Source: Link

Towers of Hanoi



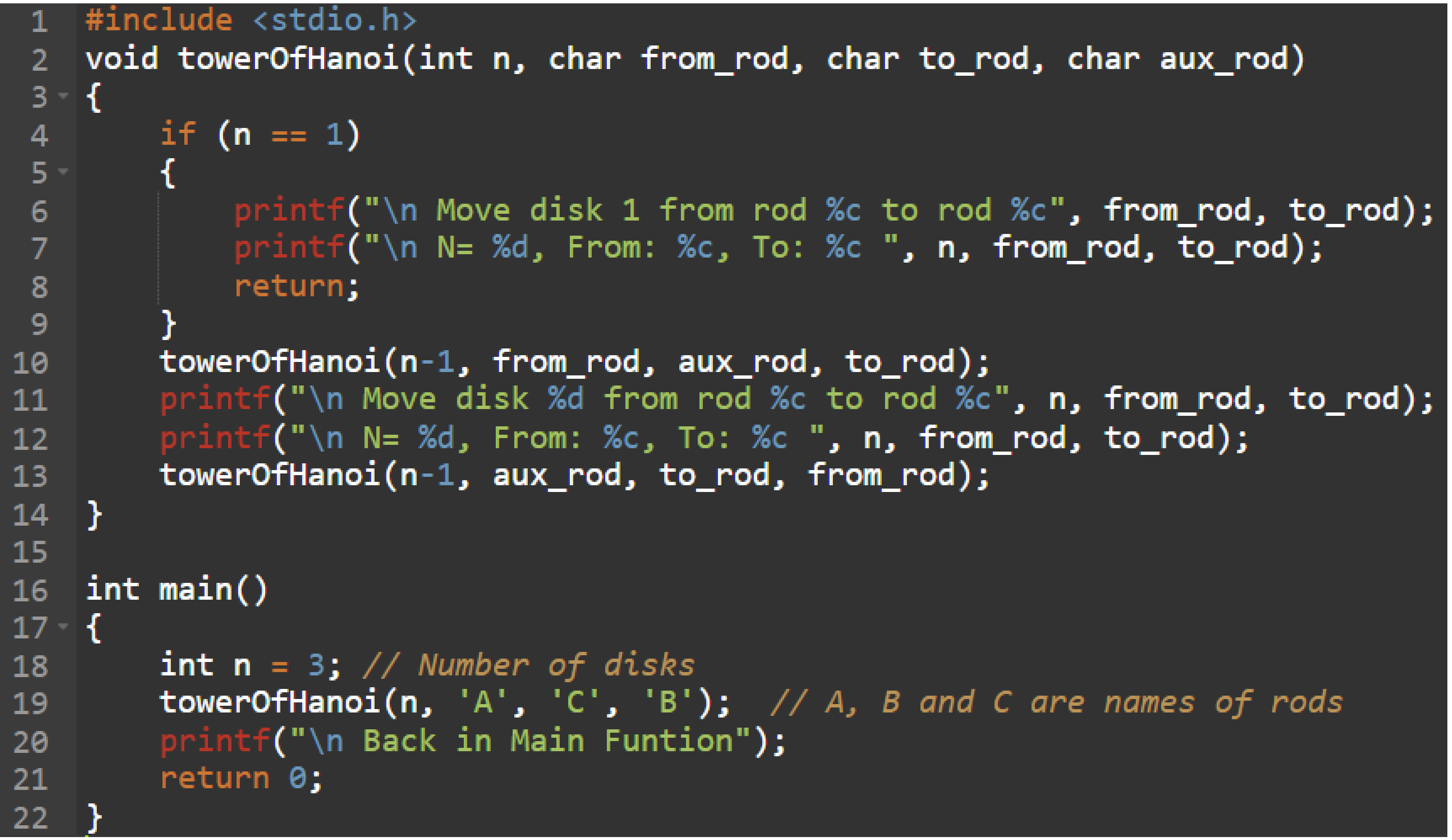
```c
#include <stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 3; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```

Steps	A	B	C
0	1,2,3		
1	2,3		1
2	3	2	1
3	3	1,2	
4		1,2	3
5	1	2	3
6	1		2,3
7			1,2,3

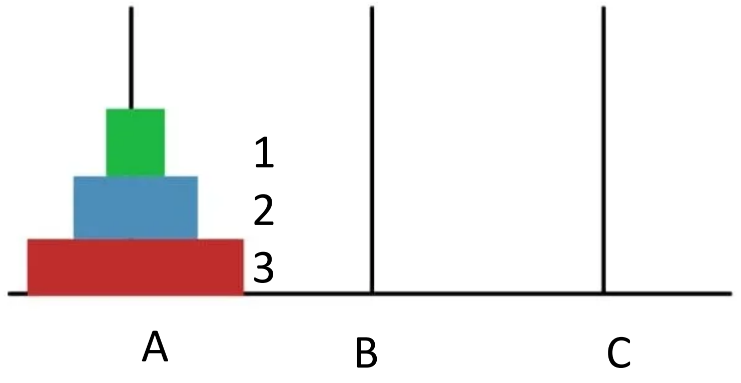
- **Step 0** = 1,2,3 mean disk 1 is on top and disk 3 is at bottom.
- The table is a visual representation of the output. Starting from Step:1
- We can further optimize it (in term of understanding) to see how these disk are moved as per discussion

Towers of Hanoi

(Optimizing to have better understanding)



```
1  #include <stdio.h>
2  void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
3  {
4      if (n == 1)
5      {
6          printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
7          printf("\n N= %d, From: %c, To: %c ", n, from_rod, to_rod);
8          return;
9      }
10     towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
11     printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
12     printf("\n N= %d, From: %c, To: %c ", n, from_rod, to_rod);
13     towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
14 }
15
16 int main()
17 {
18     int n = 3; // Number of disks
19     towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
20     printf("\n Back in Main Funtion");
21     return 0;
22 }
```



- Line 7 & line 12 will have same values or different ?
- i.e. will there be same values printed at any time during runtime

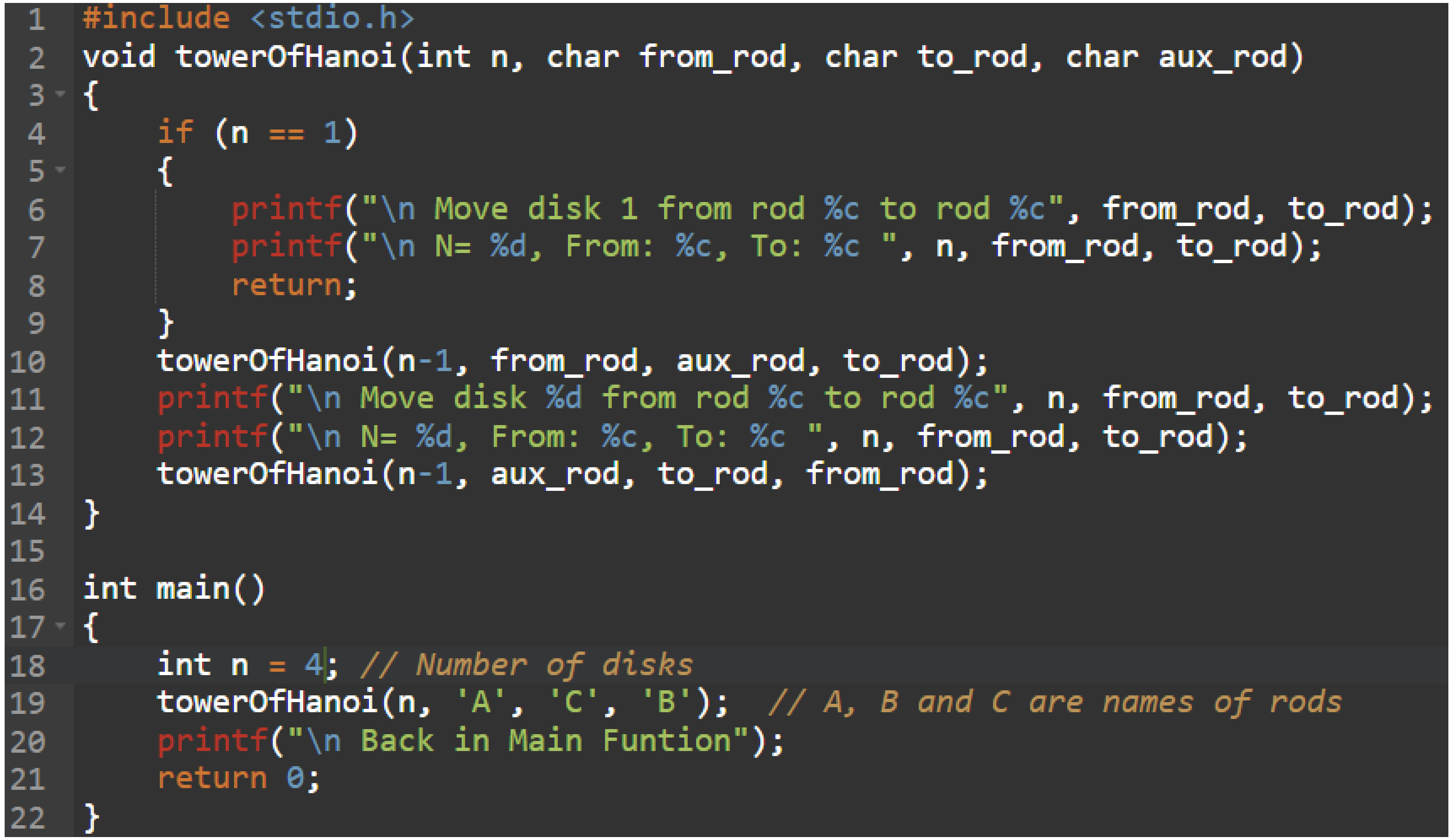
Towers of Hanoi

Will the same algorithm/code (i.e. slide) work for 4 disks?
To validate if the program works, fill the table as shown on slide 18.

```c
#include <stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
        if (n == 1)
        {
                printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
                return;
        }
        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
        printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}
int main()
{
        int n = 4;      // Number of disks
        printf("<Summer 2023>");
        towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
        return 0;
}
```

Steps	A	B	C
0	1,2,3,4		

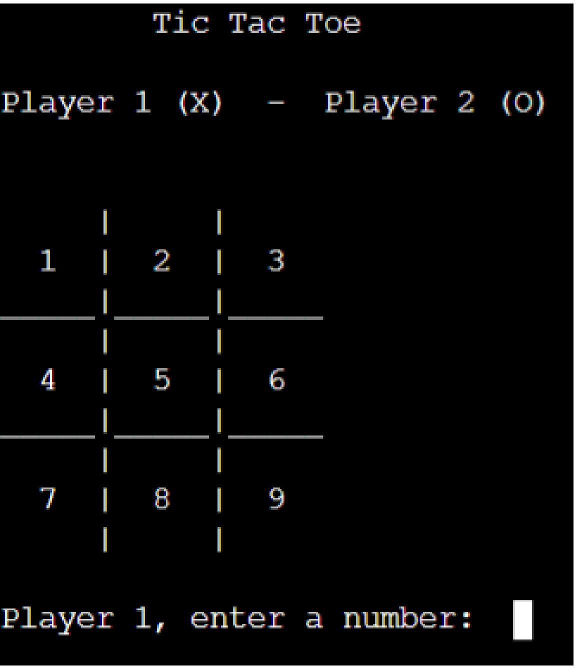
Towers of Hanoi



```c
#include <stdio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        printf("\n N= %d, From: %c, To: %c ", n, from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    printf("\n N= %d, From: %c, To: %c ", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    printf("\n Back in Main Funtion");
    return 0;
}
```

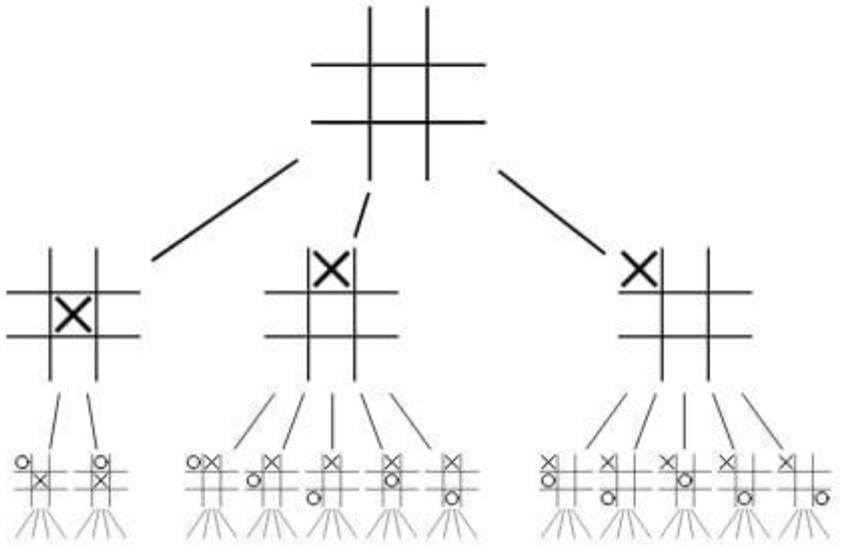
Steps	A	B	C
0	1,2,3,4		

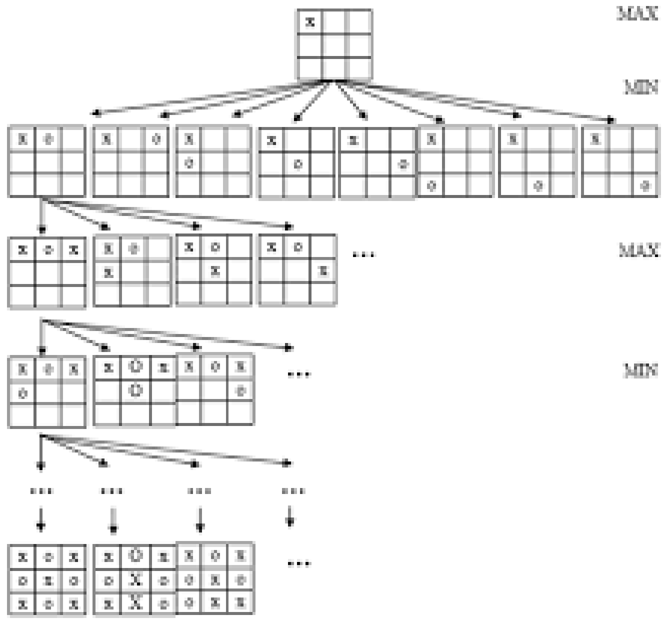
Recursion (Example: MiniMax Algorithm for Tic Tac Toe)



```
        Tic Tac Toe

Player 1 (X)  -  Player 2 (O)


      |     |
  1   |  2  |  3
______|_____|_____
      |     |
  4   |  5  |  6
______|_____|_____
      |     |
  7   |  8  |  9
      |     |

Player 1, enter a number:
```

- The MiniMax algorithm is a **recursive algorithm** used to determine the best move for a player in a game with perfect information, such as Tic Tac Toe.
- It aims to maximize the player's outcome while assuming that the opponent will also make optimal moves.
- The algorithm considers all possible moves and their consequences, thus enabling the player to make informed decisions.




MAX
MIN
MAX
MIN