Design and Analysis of Algorithm (DAA)

Analysis of Recursive Algorithms

[Module 1]

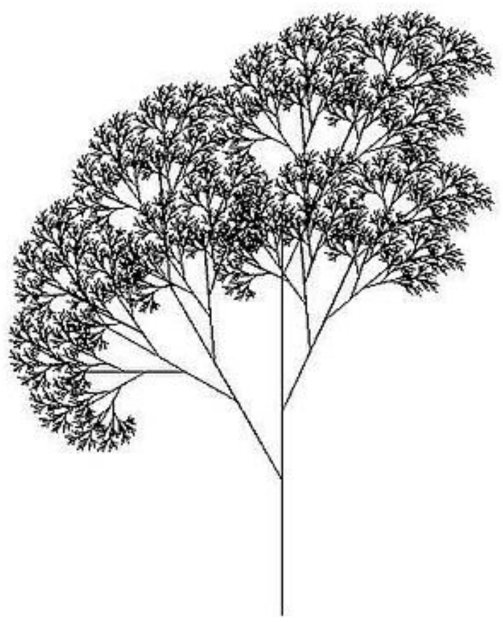
Dr. Dayal Kumar Behera

School of Computer Engineering
KIIT Deemed to be University, Bhubaneswar, India

Recursion

- **Recursion** is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem.

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.



Each branch can be seen as a smaller version of a tree.

Image Source: Wikipedia

Types of Recursion

Single recursion/Linear recursion:

- Recursion that contains only a single self-reference
- Single recursion can be further classified into:
  - **Tail recursion:** recursive call is the last statement executed within the function
  - **Head recursion:** recursive call occurs before any other operation within the function.

Multiple recursion/Tree recursion:

- Recursion that contains multiple self-references.

Single Recursion

```
fun(n)
{
    // some code
    if(n>0)
    {
        fun(n-1); // Calling itself only once
    }
    // some code
}
```

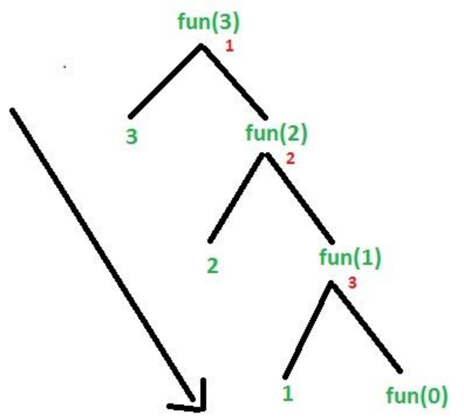
Tail Recursion

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
    }
}
```

Last statement is the recursive call.

Output for fun(3)
3 2 1

Tracing Tree Of Recursive Function



[Tail Recursion]

Output: 3 2 1
*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

Image Source: geeksforgeeks

Tail Recursion: Equivalent Loop

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
    }
}
```

```
// Equivalent Loop
void fun(int n)
{
    while (n > 0) {
        printf("%d ", n);
        n--;
    }
}
```

Tail Recursion: Time Complexity

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
    }
}
```

Recurrence equation

T(n) = T(n-1) + 1, T(1) = 1

Solving the above recurrence, the time complexity is linear: **O(n)**.

```
// Equivalent Loop
void fun(int n)
{
    while (n > 0) {
        printf("%d ", n);
        n--;
    }
}
```

Solving by step count method, the time complexity is linear: **O(n)**.

Tail Recursion: Space Complexity

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
    }
}
```

Each recursive call adds a new frame to the call stack, consuming memory.

Since there will be exactly n recursive calls (from n down to 1), the space complexity is **O(n)** (linear space).

```
// Equivalent Loop
void fun(int n)
{
    while (n > 0) {
        printf("%d ", n);
        n--;
    }
}
```

Regardless of the input value of n, the memory usage remains constant.

space complexity is **O(1)** (constant space).

Head Recursion

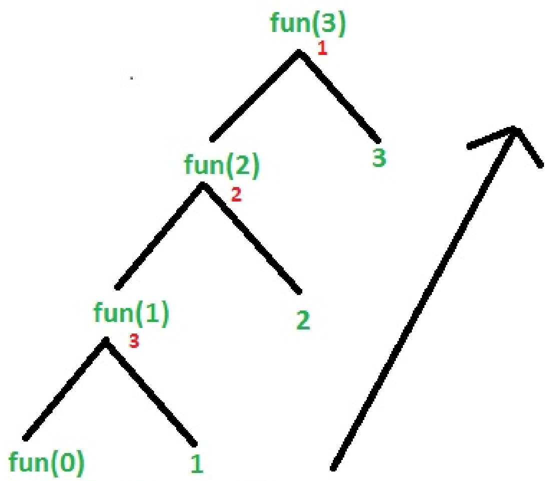
```
// Recursion function
void fun(int n)
{
    if (n > 0) {
       fun(n - 1);
       printf("%d ", n);

    }
}
```

first statement is the recursive call.

Output for fun(3)
1 2 3

Tracing Tree Of Recursive Function



[Head Recursion]

Output: 1 2 3
*Digits in red showing that the order in which the calls are made and note that printing done at returning time. And it does nothing at calling time.

Image Source: geeksforgeeks

Head Recursion: Equivalent Loop

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
       fun(n - 1);
       printf("%d ", n);
    }
}
```

Unlike Tail recursion, in head recursion the conversion is not easy.

```
// Wrong Output
void fun(int n)
{
    while (n > 0) {
        n--;
        printf("%d ", n);
    }
}
```

```
// Equivalent Loop
void fun(int n)
{
    int i=1;
    while (i <= n) {
       printf("%d ", i);
       i++;
    }
}
```

Head Recursion: Time Complexity

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
       fun(n - 1);
       printf("%d ", n);
    }
}
```

Recurrence equation

T(n) = T(n-1) + 1, T(1) = 1

Solving the above recurrence, the time complexity is linear: **O(n)**.

```
// Equivalent Loop
void fun(int n)
{
    int i=1;
    while (i <= n) {
       printf("%d ", i);
       i++;
    }
}
```

Solving by step count method, the time complexity is linear: **O(n)**.

Head Recursion: Space Complexity

```
// Recursion function
void fun(int n)
{
    if (n > 0) {
        fun(n - 1);
        printf("%d ", n);
    }
}
```

Each recursive call adds a new frame to the call stack, consuming memory.

Since there will be exactly n recursive calls (from n down to 1), the space complexity is **O(n)** (linear space).

```
// Equivalent Loop
void fun(int n)
{
    int i=1;
    while (i <= n) {
        printf("%d ", i);
        i++;
    }
}
```

The function fun contains only one extra integer variable: i.

Regardless of the input value n, the memory used for i remains the same.

There are no data structures (arrays, lists, etc.)

space complexity is **O(1)** (constant space).

Tree Recursion

```
// Recursion function
void fun(int n){
    if (n > 0) {
       printf("%d ", n);
       fun(n - 1);
       fun(n - 1);
}}
```

recursive call more than once in the code.

Output for fun(3)
3 2 1 1 2 1 1

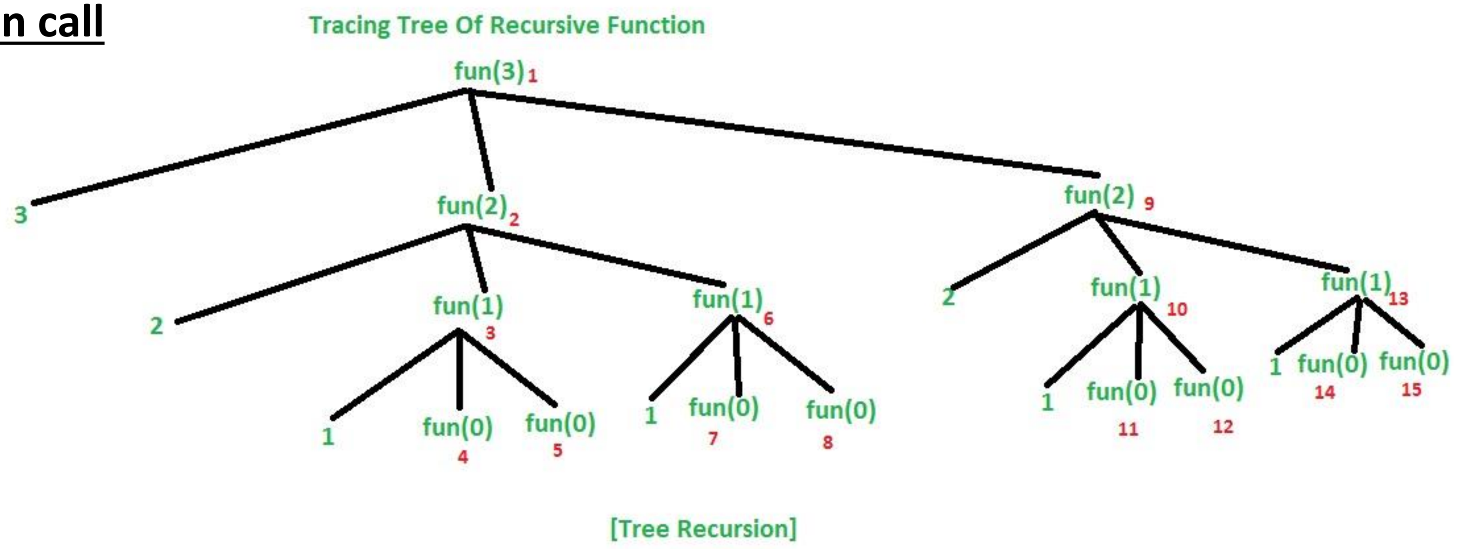
No. of function call

1

2

4

8



Output: 3 2 1 1 2 1 1
*Digits in red showing that the order in which the calls are made and according to the order of calling the output are printed on the screen.Note that for fun(0) it gives nothing as output.

Image Source: geeksforgeeks

Tree Recursion: Equivalent Loop

```
// Recursion function
void fun(int n){
    if (n > 0) {
       printf("%d ", n);
       fun(n - 1);
       fun(n - 1);
    }
}
```

Unlike Tail recursion and head recursion, here the direct conversion may not be possible.

Time Complexity

In recursion, time complexity depends on number of function call.

In this example, total function call for n=3

1 + 2 + 4 + 8 = 15

2^0 + 2^1 + 2^2 + 2^3 = (2^(n+1))-1 = **O(2^n)**

```
// Recursion function
void fun(int n){
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
        fun(n - 1);
    }
}
```

Recurrence equation

T(n) = T(n-1) + T(n-1) + 1, T(1) = 1

Solving the above recurrence, the time complexity is exponential: **O(2^n)**.

Solving the recurrence

Let us solve the following recurrence using iterative method

T(n) = T(n-1) + T(n-1) + 1, T(1) = 1

Recurrence Relation: T(n) = 2T(n-1) + 1

Iterative Solution:

- We'll use an iterative approach to find a pattern.
- Start with the base case: (T(1) = 1).
- Then compute (T(2)), (T(3)), and so on:

T(2) = 2T(1) + 1 = 2 + 1 = 3
T(3) = 2T(2) + 1 = 2 * 3 + 1 = 7
T(4) = 2T(3) + 1 = 2 * 7 + 1 = 15
T(5) = 2T(4) + 1 = 2 * 15 + 1 = 31

General Pattern:

T(n) = 2^n - 1

Solving the above recurrence, the time complexity is exponential T(n) = 2^n – 1 = **O(2^n)**.

Space Complexity

- When analyzing space complexity, we consider the memory used by the **call stack** during the execution of the recursive function.
- Each recursive call creates a new stack frame (activation record) with local variables and return addresses.
- The depth of the recursion tree is (n), as each level corresponds to decrementing (n) by 1.
- The maximum number of active function calls (stack frames) at any point during execution is the depth of the recursion tree.
- Therefore, the space complexity is O(n).

```c
// Recursion function
void fun(int n){
    if (n > 0) {
        printf("%d ", n);
        fun(n - 1);
        fun(n - 1);
    }
}
```

Each of your actions will have an impact on your future.

Once you know who is walking with you on your path. you will never be afraid.

Thank you