Data Structures

Time Complexity – Introduction

CODE
FORTHINGS

Complexity – Introduction

Data Structure –Complexity

Introduction

Solution 1

Solution 5

Solution 2

Solution 4

Solution 3

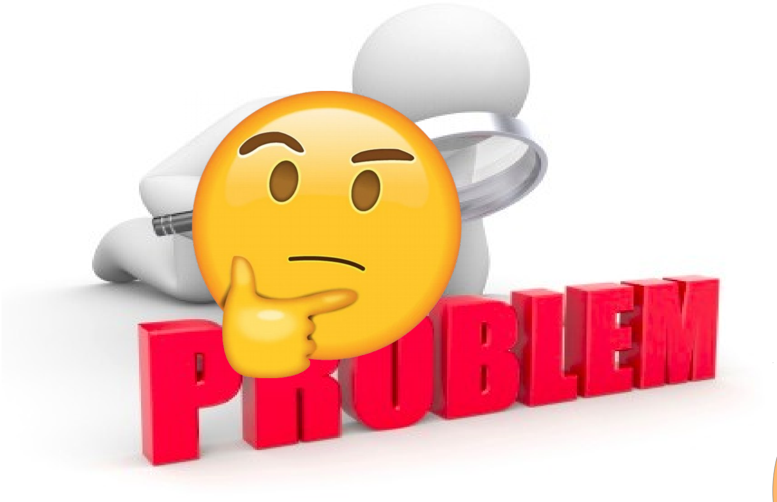
Data Structure –Complexity

Introduction

Solution 1

Solution 5

Solution 2



Solution 4

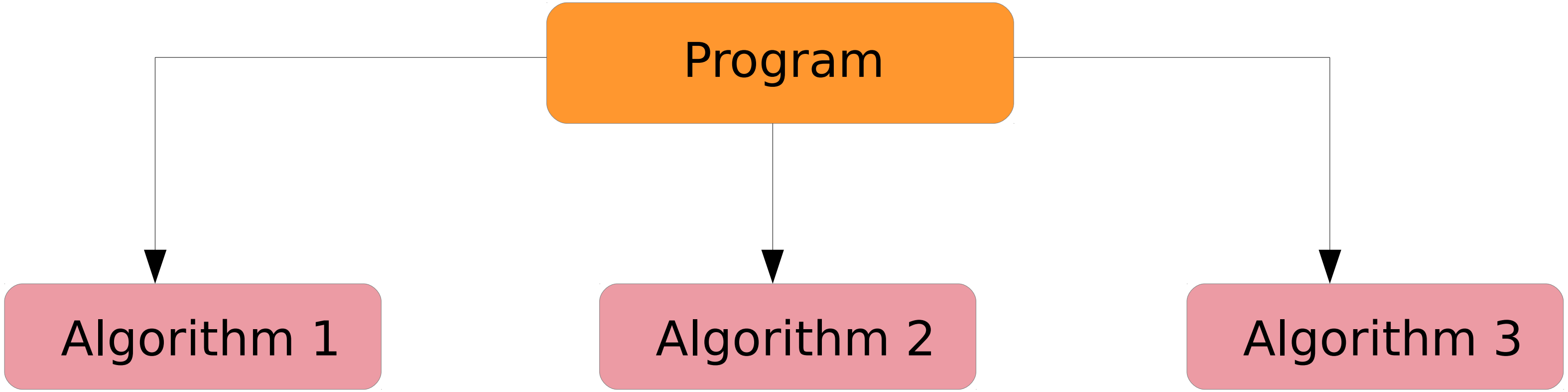
Solution 3

Data Structure –Complexity

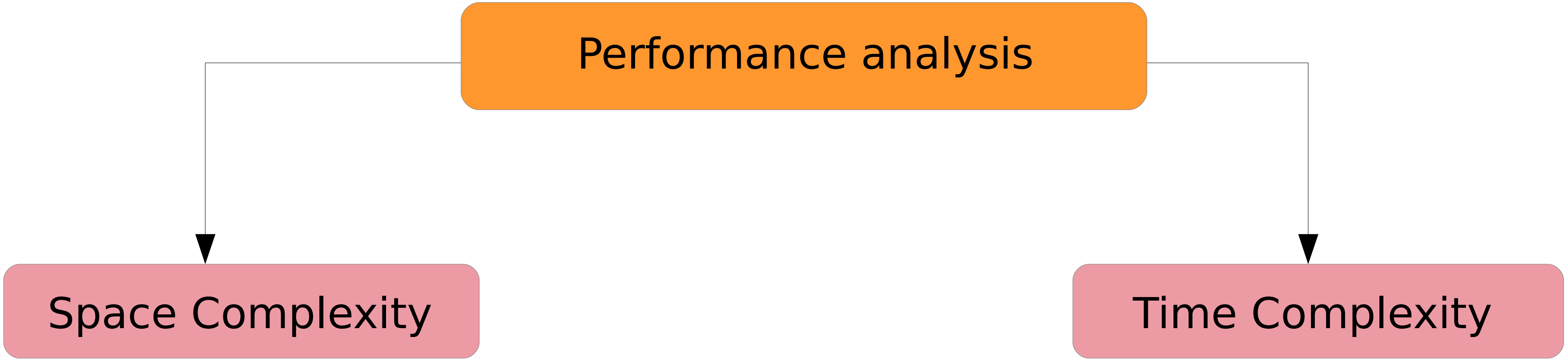
Introduction

Data Structure –Complexity

Introduction



Introduction



Introduction

Space Complexity

Amount of memory it takes to run the program completely and efficiently

Time Complexity

Amount of time it takes to run the program completely and efficiently

Time Complexity – Part 1

Introduction

Time Complexity

Amount of time required to run the program completely and efficiently

Introduction

Time Complexity

Amount of time required to run the program completely and efficiently

Time – Number of memory access

Number of comparision

Number of time some loop is executed

Introduction

Asymptotic notations

There are mathematical tools to represent time and space complexity of algorithm

Types

- Big Ohmega
- Big Theta
- Big Oh

Introduction

Code

```
int main()
{
      printf("Hello World\n");
}
```

- It will execute for 1 time.

$T(n) = O(1)$

Introduction

Code

```
int main()
{
    printf("Hello World\n");
    printf("Hello World\n");
    printf("Hello World\n");
}
```

- 3 printf statements.
- Each printf will be executed for 1 time.

$T(n) = O(1)$

Introduction

Code

```
int main()
{
	for (int i = 0; i < 10; i++)
	{
		printf("Hello World\n");
	}
}
```

- Printf will be executed for 10 times.
- 10 is constant.
- Here is the 1st rule.
  - **Whenever the constant comes ignore it. It can be in the place of addition, multiplication and divisio**
- TC is O(10 * 1) -> 10 is constant.

$T(n) = O(1)$

Introduction

Code

```
int main()
{
        for (i = 1; i<= n ; i++ )
        {
            statement;
        }
}
```

- This for loop will execute for **n** times.
- **n** is variable.
- Here is the 2nd rule,
  - **Whenever variable comes, consider as Infinity.**
- So, TC is O(1 * n)

$T(n) = O(n)$

Introduction

Code

```
int main()
{
        for (i = n; i>= 1 ; i-- )
        {
            statement;
        }
}
```

T(n) = O(n)

Introduction

Code

```
int main()
{
        for (i = 1; i <= n ; i = i+2)
        {
            statement;
        }
}
```

- Value of i getting incremented by 2.
- If **n** is 20, then loop will run for 10 times.
- Means **n/2** times
- As per rule 1, ignore the constant.
- So, TC is O(1 * n)

T(n) = O(n)

Example

Sequential for Loop

```
for ( i = 1; i <= n ; i++ )
{
    count++;
}

for ( i = 1; i <= n ; i++ )
{
     k++;
}
```

- 1st for loop will run for **n** times.
- 2nd for loop will run for **n** times.
- **n** is variable, so as per 2nd rule we need to consider as infinity.
- It is Independed for loops.
- So, TC -> O(n) + O(n) -> O(2n)
- As per 1st rule ignore constant.

$T(n) = O(n)$

Example

Nested for Loop

```
for ( i = 1; i <= n ; i++ )
{
    for ( j = 1; j <= n ; j++ )
    {
        k++;
    }
}
```

- Outer for loop will run for **n** times.
- Inner for loop will run for **n** times.
- It is nested for loops.
- So, TC -> O(n) * O(n)

$T(n) = O(n^2)$

Example

Code

```
P = 0
for (i = 1; P <= n ; i++)
{
      P = P + i;
}
```

Assume, $P > n$

$$P = k(k + 1) / 2$$
$$k(k + 1) / 2 > n$$
$$k^2 > n$$
$$k^2 = n$$
$$k = \sqrt{n}$$

- Here this loop will not execute for **n** times.
- Because condition is different.
- P is getting added by i for every iteration.

i	P	
1	0+1	= 1
2	1+2	= 3
3	1+2+3	= 6
4	1+2+3+4	= 10
.		
.		
k	1+2+3+4+5+...+k	

$T(n) = O(\sqrt{n})$

Example

Nested for Loop

```
for ( i = n/2; i <= n ; i++ )
{
     for (j = 1; j <= n ; j++ )
     {
          k++;
     }
}
```

- Outer for loop will run for **n/2** times.
- Inner for loop will run for **n** times.
- It is nested for loops.
- So, TC -> O(n/2) * O(n) -> O(n^2/2) (rule 1)

$T(n) = O(n^2)$

Example

Nested for Loop

```
for ( i = 0; i < n ; i++ )
{
      for ( j = 0; j < i ; j++ )
      {
            k++;
      }
}
```

$T(n) = O(n^2)$

i	j	no.of times
0	0	0
1	0	
	1x	1
2	0	
	1	
	2x	2
.		.
.		.
n		n

So, TC -> O(n) * O(n)

Example

Non Linear for Loop

```
for ( i = 1; i < n ; i = i * 2 )
{
      c++;
}
```

Example

Non Linear for Loop

```
for ( i = 1; i < n ; i = i * 2 )
{
      c++;
}
```

n = 16

i < n

1 < 16	2 ^ 0
2 < 16	2 ^ 1
4 < 16	2 ^ 2
8 < 16	2 ^ 3
16 < 16	2 ^ 4

Example

Non Linear for Loop

```
for ( i = 1; i < n ; i = i * 2 )
{
     c++;
}
```

$i = n$

$2^k = n$

$k = \log_2 n$

$T(n) = O(\log_2 n)$

n = 16

i < n	
1 < 16	2 ^ 0
2 < 16	2 ^ 1
4 < 16	2 ^ 2
8 < 16	2 ^ 3
16 < 16	2 ^ 4

Example

Non Linear for Loop

```
for ( i = n; i >= 1 ; i = i  / 2 )
{
      c++;
}
```

Example

Non Linear for Loop

```
for ( i = n; i >= 1 ; i = i  / 2 )
{
      c++;
}
```

n = 16

i	>=	1	
16	>	1	2 ^ 4
8	>	1	2 ^ 3
4	>	1	2 ^ 2
2	>	1	2 ^ 1
1	=	1	2 ^ 0

Example

Non Linear for Loop

```
for ( i = n; i >= 1 ; i = i  / 2 )
{
      c++;
}
```

$2^k = n$

$k = \log_2 n$

$T(n) = O(\log_2 n)$

n = 16

i	>=	1	
16	>	1	2 ^ 4
8	>	1	2 ^ 3
4	>	1	2 ^ 2
2	>	1	2 ^ 1
1	=	1	2 ^ 0

Example

Non Linear for Loop

```
for ( i = 1;  i * i  <= n ;  i++ )
{
      k++;
}
```

Example

Non Linear for Loop

```
for ( i = 1;  i * i  <= n ;  i++ )
{
      k++;
}
```

$$T(n) = O(\sqrt{n})$$

$n = 16$

$i * i \quad <= \quad n$

$1 * 1 \;<= 16$

$2 * 2 \;<= 16$

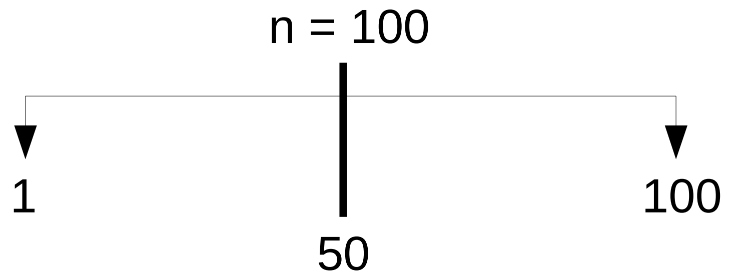
$3 * 3 \;<= 16$

$4 * 4 \quad = 16$

$25 \;\not\leq\; 16$

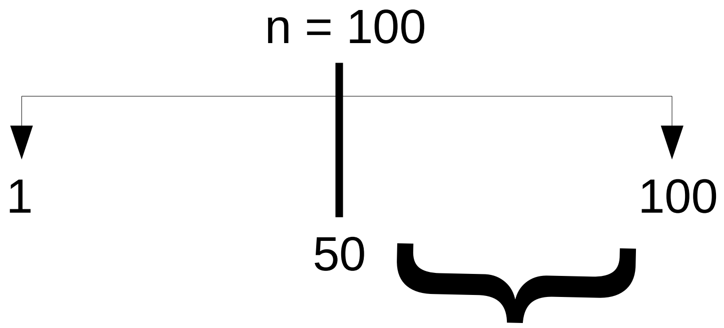
Example

```
for ( i = n / 2; i <= n ; i++ )
{
   for ( j = 1; j + n/2 <= n ; j++ )
   {
      for ( k = 1;  k <= n ; k = k * 2 )
      {
            k++;
      }
```



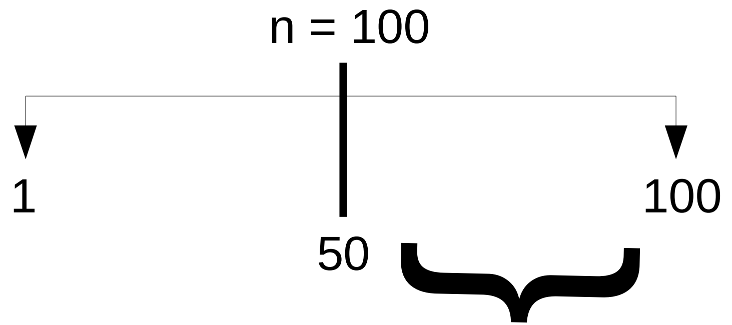
Example

```
for ( i = n / 2; i <= n ; i++ )
{
   for ( j = 1; j + n/2 <= n ; j++ )
   {
      for ( k = 1;  k <= n ; k = k * 2 )
      {
             k++;
      }
```



Example

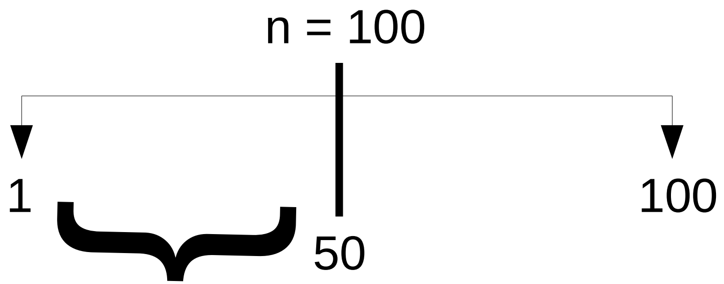
```
for ( i = n / 2; i <= n ; i++ )
{
   for ( j = 1; j + n/2 <= n ; j++ )
   {
      for ( k = 1;  k <= n ; k = k * 2 )
      {
             k++;
      }
```



T(C) = O(n / 2)

Example

```
for ( i = n / 2; i <= n ; i++ )
{
  for ( j = 1; j + n/2 <= n ; j++ )
   {
     for ( k = 1;  k <= n ; k = k * 2 )
     {
         count++;
     }
```

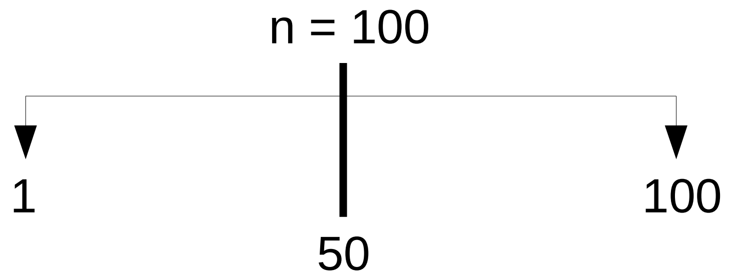


T(C) = O((n / 2) * (n / 2)

T(C) = O(n ^2 / 4)

Example

```
for ( i = n / 2; i <= n ; i++ )
{
   for ( j = 1; j + n/2 <= n ; j++ )
   {
      for ( k = 1;  k <= n ; k = k * 2 )
      {
            count++;
      }
```



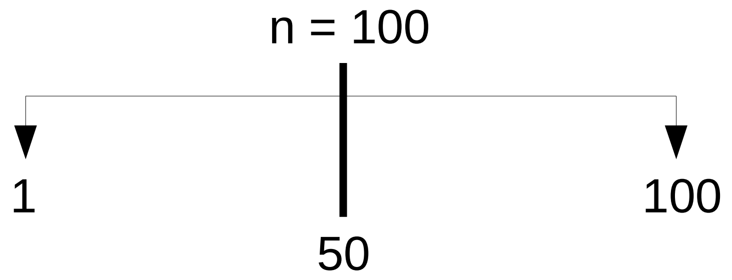
T(C) = O((n / 2) * (n / 2)

T(C) = O(n ^2 / 4)

T(C) = O((n ^2 / 4) * logn)

Example

```
for ( i = n / 2; i <= n ; i++ )
{
   for ( j = 1; j + n/2 <= n ; j++ )
   {
      for ( k = 1;  k <= n ; k = k * 2 )
      {
            count++;
      }
```



T(n) = O((n / 2) * (n / 2)

T(n) = O(n ^2 / 4)

T(n) = O((n ^2) * logn)

Time Complexity -Recursion

Recursive Tree

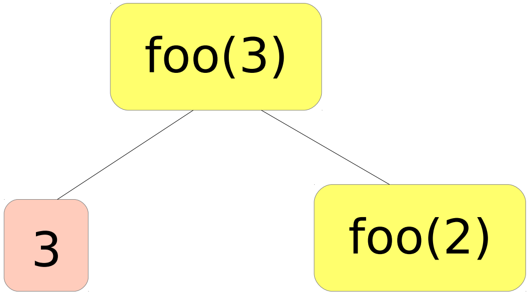
```
void foo(int n)
{
    if( n > 0)
    {
        printf("%d",n);
        foo(n-1);
    }
}
```

foo(3)

Data Structure –Time Complexity

Recursive Tree

```
void foo(int n)
{
      if( n > 0)
      {
            printf("%d",n);
            foo(n-1);
      }
 }
```

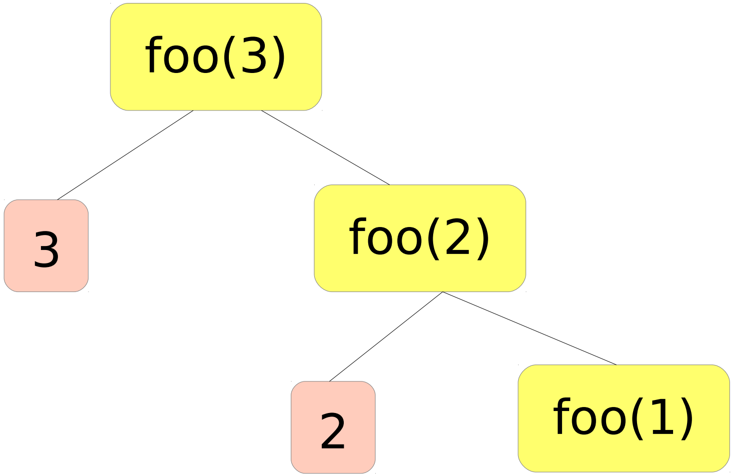


foo(3)

Recursive Tree

```
void foo(int n)
{
    if( n > 0)
    {
        printf("%d",n);
        foo(n-1);
    }
}
```

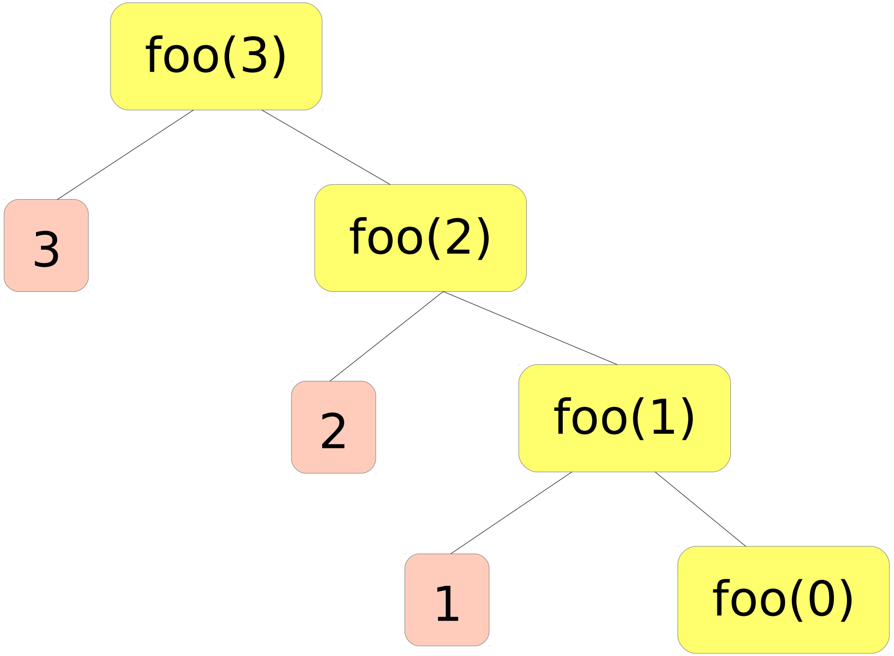
foo(3)



Recursive Tree

```
void foo(int n)
{
     if( n > 0)
     {
          printf("%d",n);
          foo(n-1);
     }
 }
```

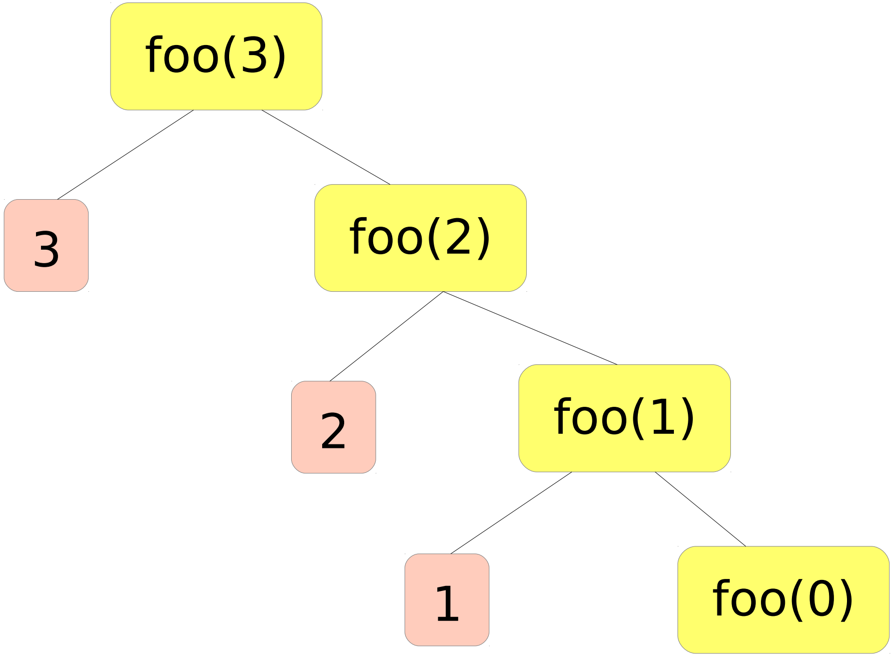
foo(3)



Recursive Tree

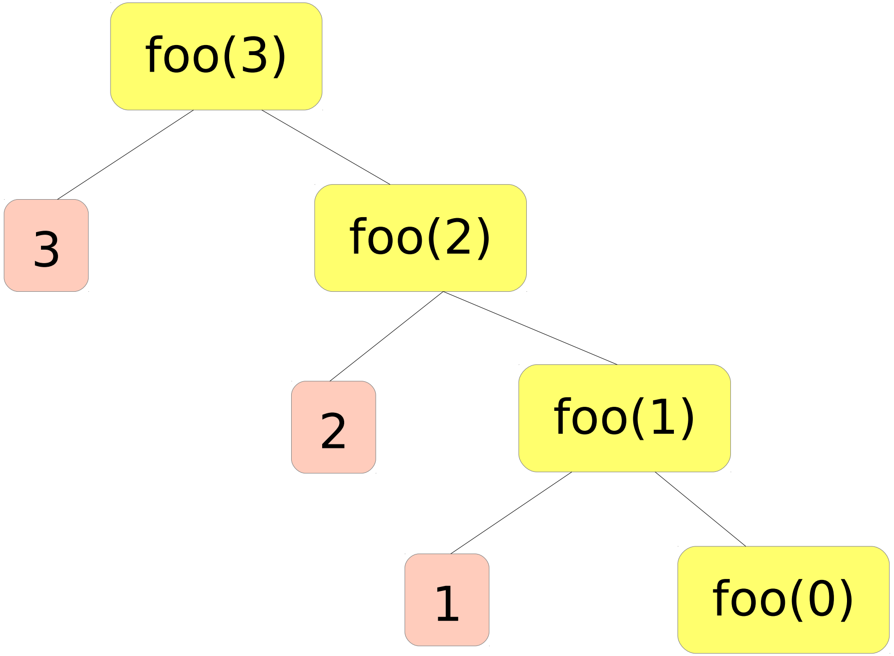
```
void foo(int n)
{
    if( n > 0)
    {
        printf("%d",n);
        foo(n-1);
    }
}
```

foo(3)



Recursive Tree

```
void foo(int n)
{
     if( n > 0)
     {
          printf("%d",n);
          foo(n-1);
     }
}
```



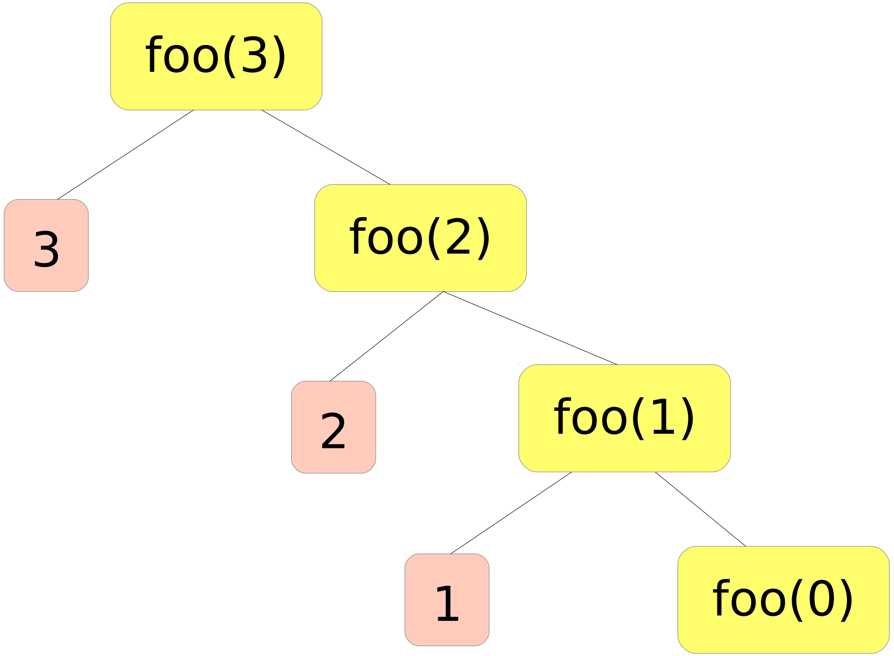
foo(3)

TC = O(n+1)

Data Structure –Time Complexity

Recursive Tree

```
void foo(int n)
{
     if( n > 0)
     {
          printf("%d",n);
          foo(n-1);
     }
}
```



TC = O(n)

Example

Range

$1 < \log n < \text{root}(n) < n < n\log n < n^2 < n^3 < \ldots\ldots\ldots\ldots\ldots\ldots < n^n$

Rules:

- <u>Rule 1</u>: Whenever the constant comes discard it (in place of addition, multiplication and divition).
- <u>Rule 2</u>: Whenever the variable comes consider as infinite value.
- <u>Rule 3</u>: Always consider highest degree.
- <u>Rule 4</u>: When differemt complexities are there, always consider worst case.