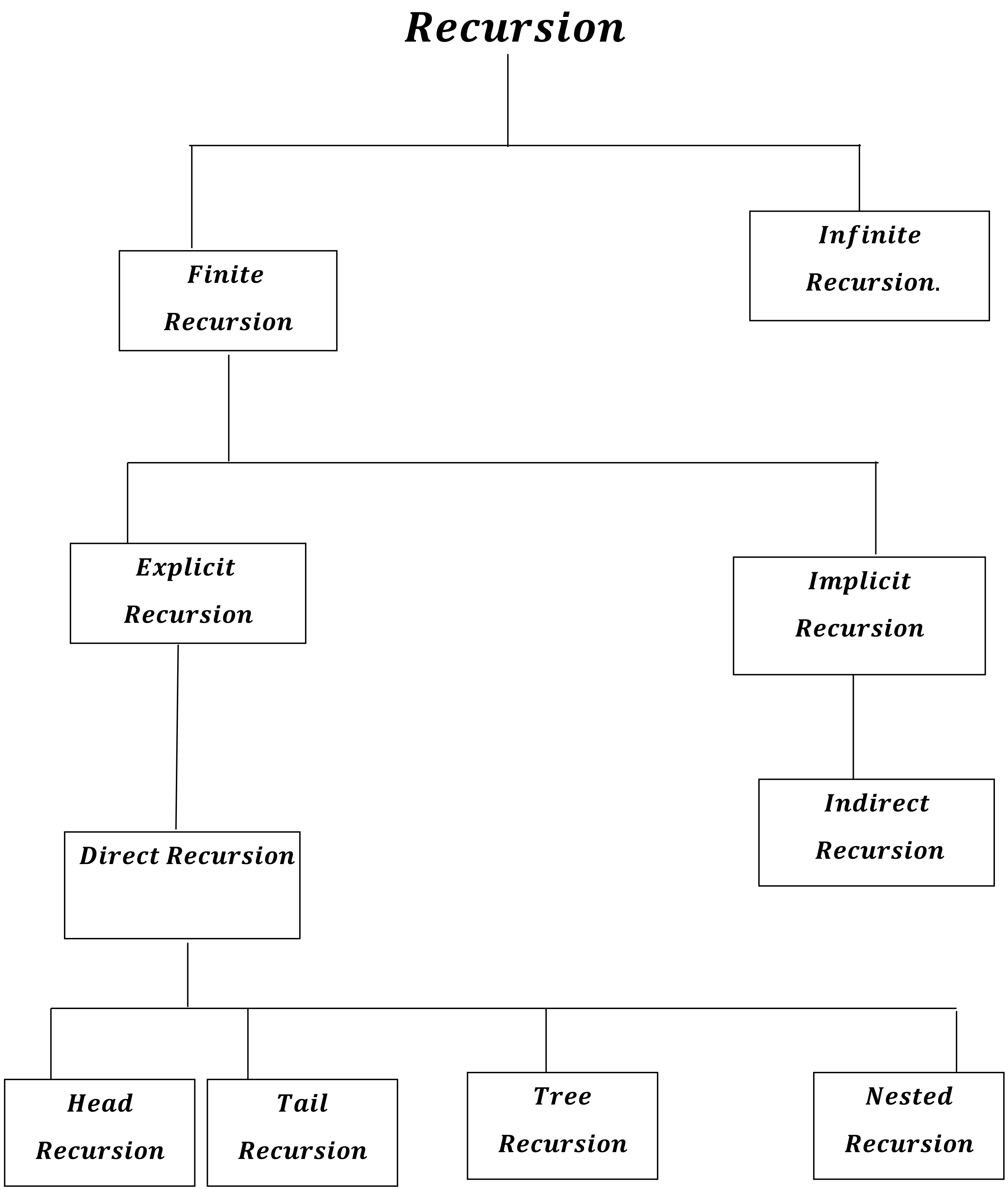
Types of Recursion



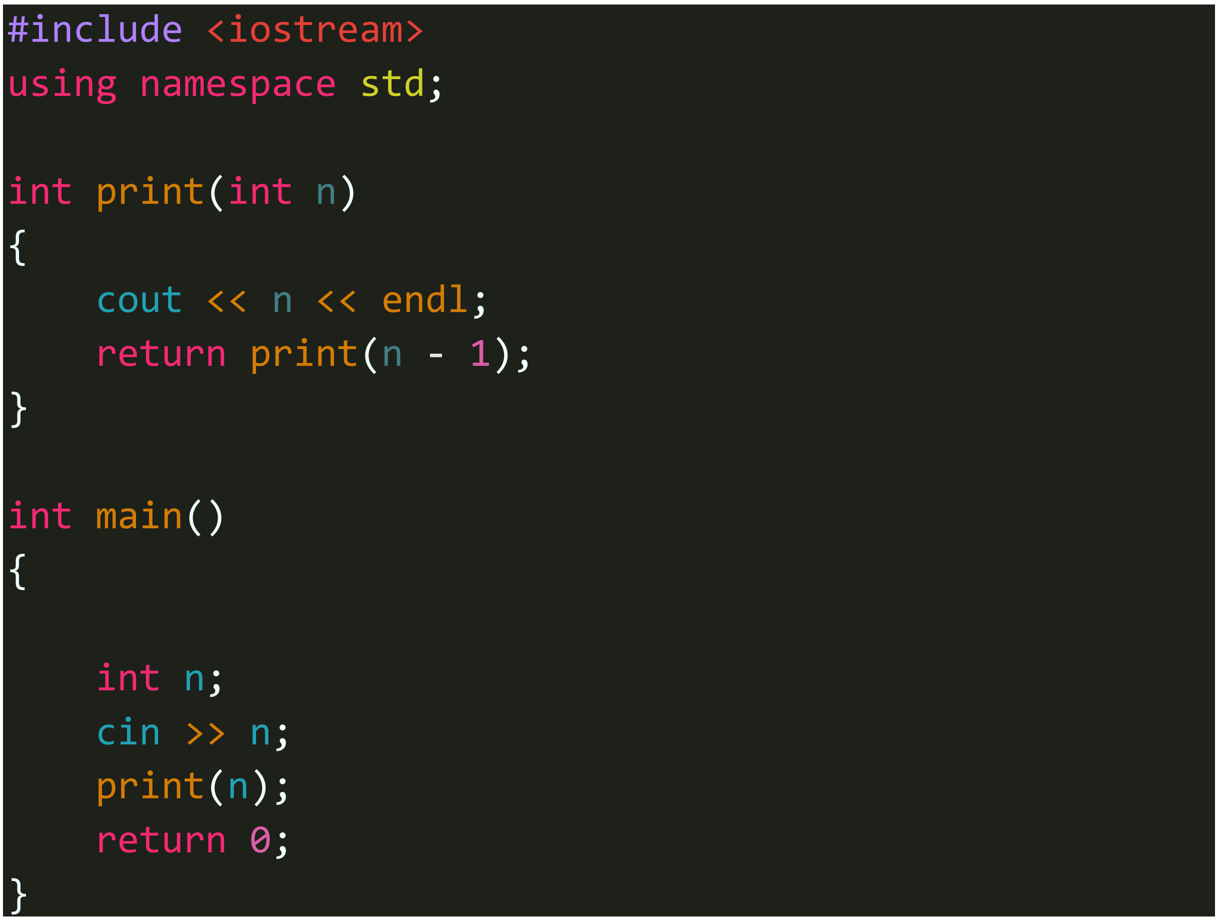
1. *Finite Recursion*:

Finite recursion are those recursion that stop after a finite number of recursive calls.

Notably, finite recursions have reasonable base cases and base case meets after a finite number of recursive calls.

2. *Infinite Recursion*:

Infinite recursion are those recursion that will continue infinite times. Such as:



```
#include <iostream>
using namespace std;

int print(int n)
{
    cout << n << endl;
    return print(n - 1);
}

int main()
{

    int n;
    cin >> n;
    print(n);
    return 0;
}
```

That is those recursion that doesnot have base cases.

Note: Those recursion that doesnot have base cases, runs infinite times , hence infinite recursion doesnot have base cases to exit.

As it create infinite stack frame , hence there will occur stack overflow and segementation fault.

Segmentation Fault: A segmentation fault occurs when a program tries to access memory that it is not allowed to access. This can happen when a program tries to read or write to memory that is not allocated to it, or when a program tries to access memory that is marked as read−only. When a segmentation fault occurs, the program will typically crash and generate an error message.

In most cases, a stack overflow will cause a segmentation fault. However, there are some cases where a stack overflow will not cause a segmentation fault.

For example, if the program is using a guard page, then the operating system will catch the stack overflow and prevent it from causing a segmentation fault.

Guard Page: A guard page is a special type of page that is used to protect the stack from overflow.
A guard page is marked as read − only, so if the stack tries to grow past the guard page, the operating system will raise an exception. This exception can then be used to handle the stack overflow gracefully.
The exception is: STATUS GUARD PAGE VIOLATION .

Explicit Recursion

Explicit recursion is a programming technique in which a function calls itself directly.

Explicit recursion is also known as Direct Recursion.

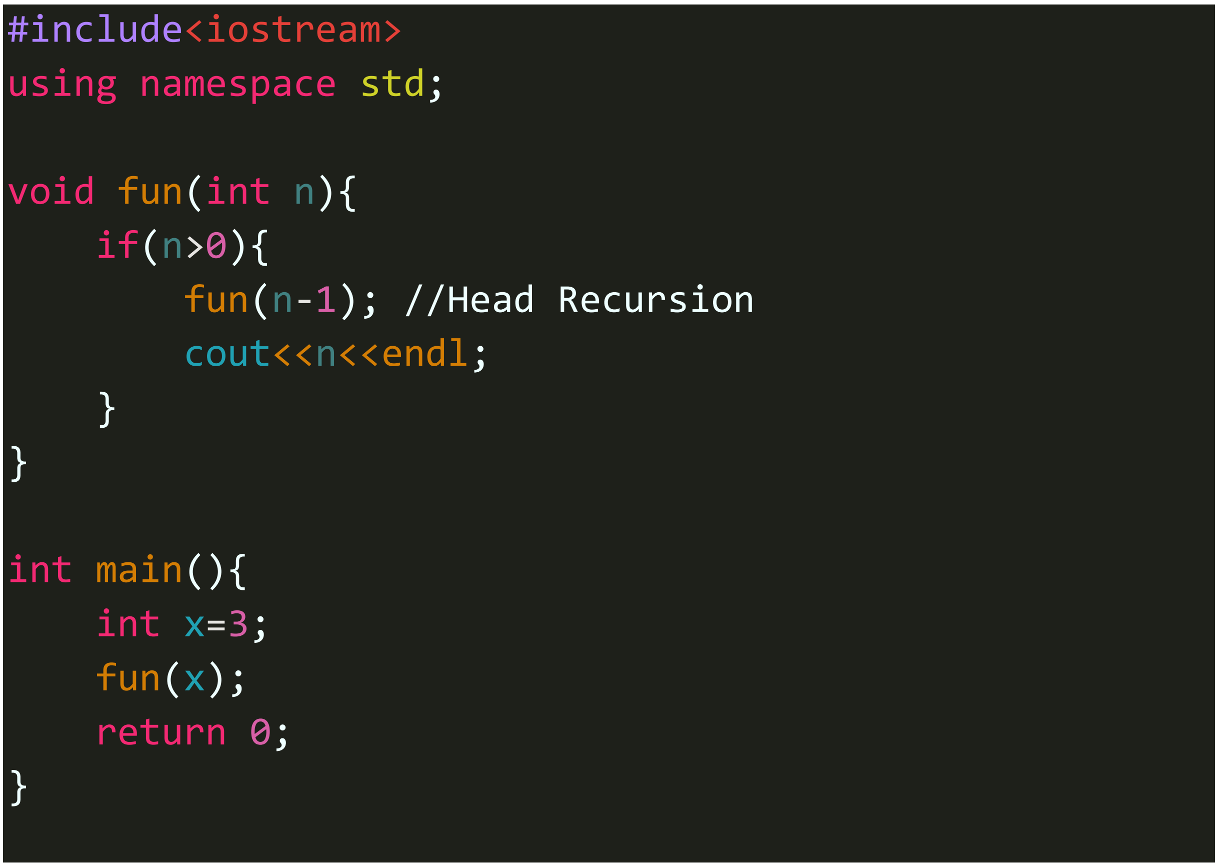
Direct Recursion

Direct Recursion , is a type of recursion in which a function calls itself directly.

Direct Recursion divided into :

A. Head Recursion

Head Recursion is a type of recursion in which the recursive call is the first statement of the function. This means that the function doesnot do any processing before it calls itself.



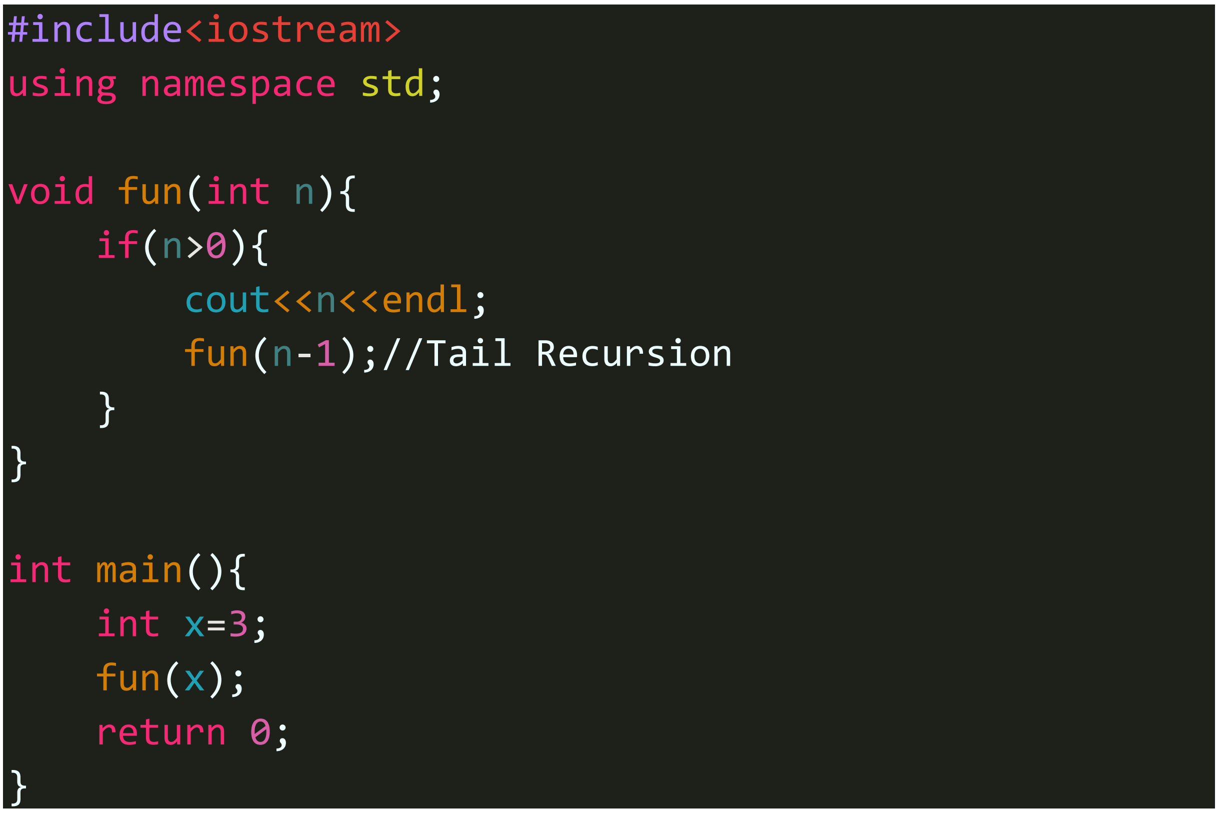
```
#include<iostream>
using namespace std;

void fun(int n){
    if(n>0){
        fun(n-1); //Head Recursion
        cout<<n<<endl;
    }
}

int main(){
    int x=3;
    fun(x);
    return 0;
}
```

***fun*(*n* − 1) *is executed at first ,then cout is executed*.**

***if n becomes less than* 0 *the function exits act as base case simultaneously*.**

B*.*Tail Recursion

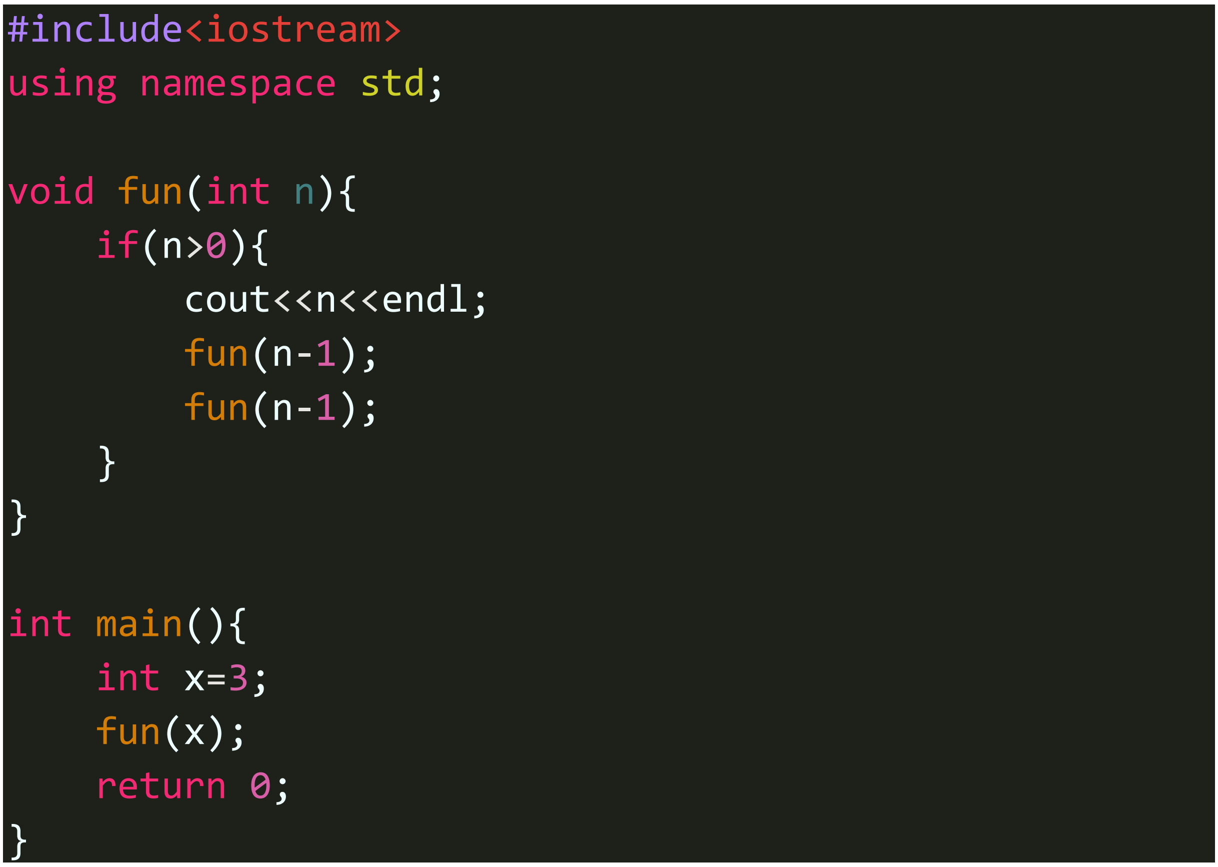
***Tail recursion*, *where the last operation done by function is recursive call*.**



```cpp
#include<iostream>
using namespace std;

void fun(int n){
    if(n>0){
        cout<<n<<endl;
        fun(n-1);//Tail Recursion
    }
}

int main(){
    int x=3;
    fun(x);
    return 0;
}
```

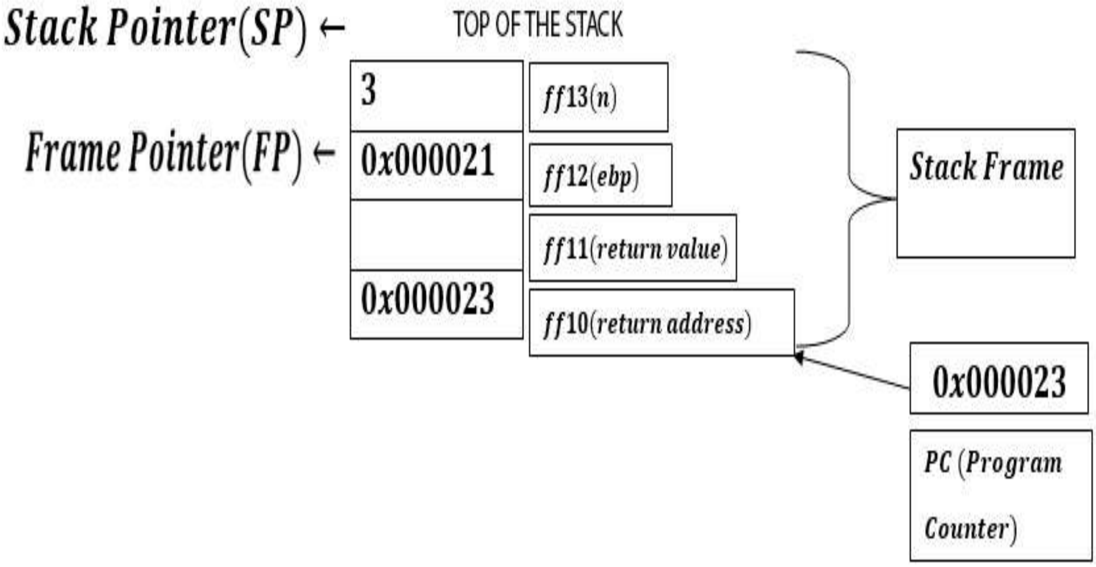
Here, $fun(n-1)$*is executed as last operation in the function.*

C. Tree Recursion

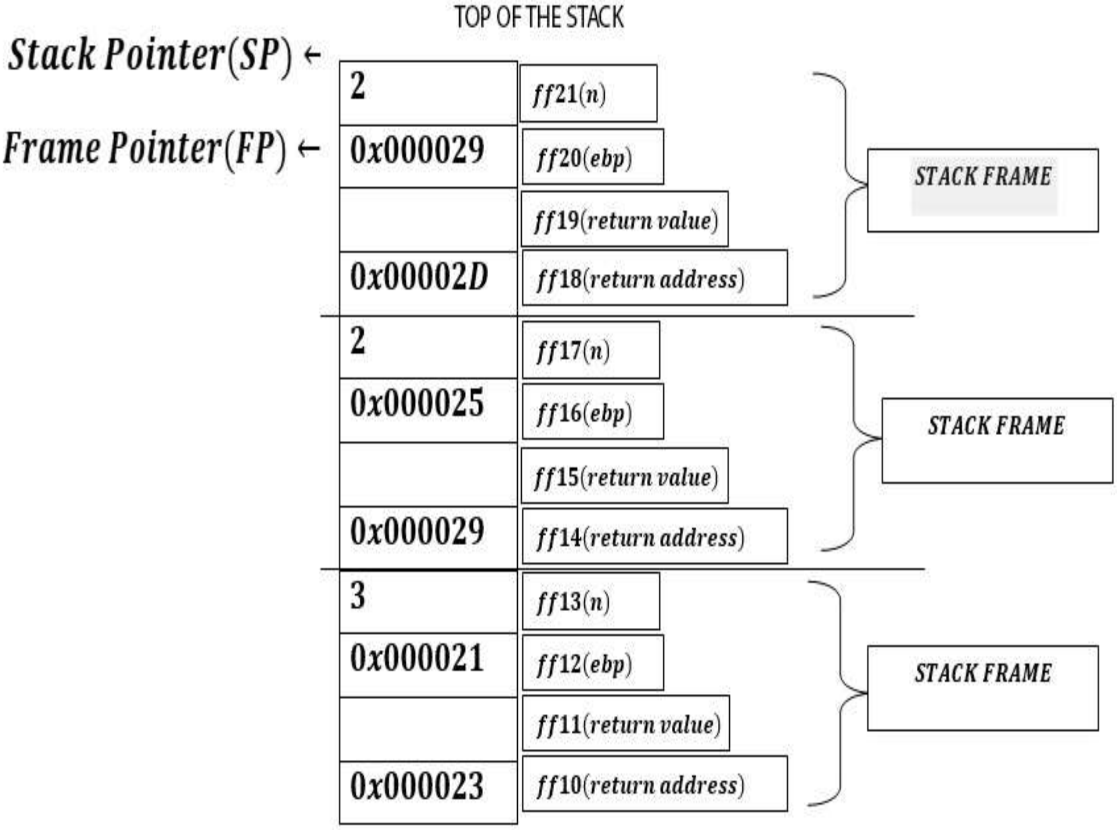


```
#include<iostream>
using namespace std;

void fun(int n){
    if(n>0){
        cout<<n<<endl;
        fun(n-1);
        fun(n-1);
    }
}

int main(){
    int x=3;
    fun(x);
    return 0;
}
```

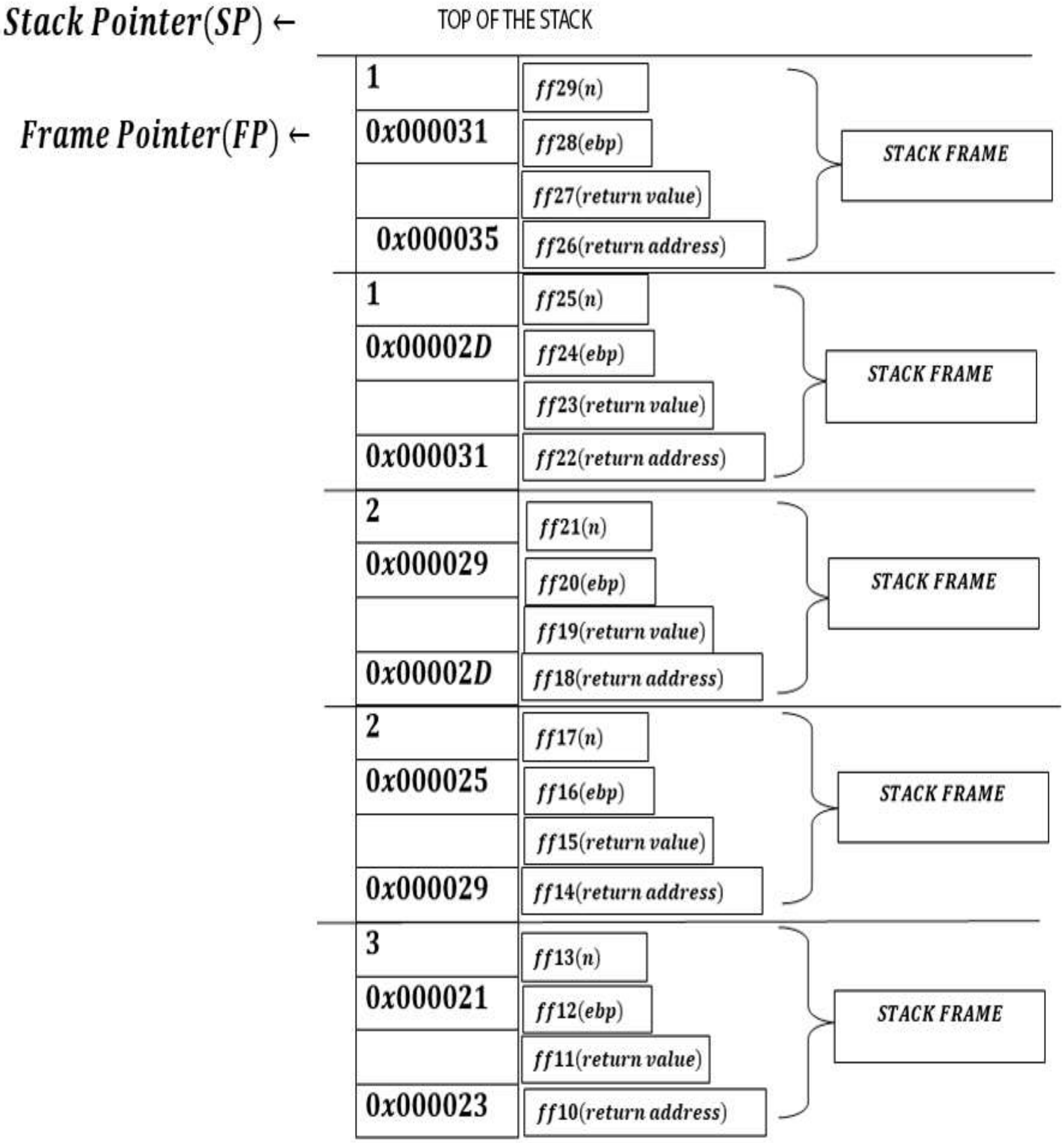
***Here first the initial stack frame will get created*:**



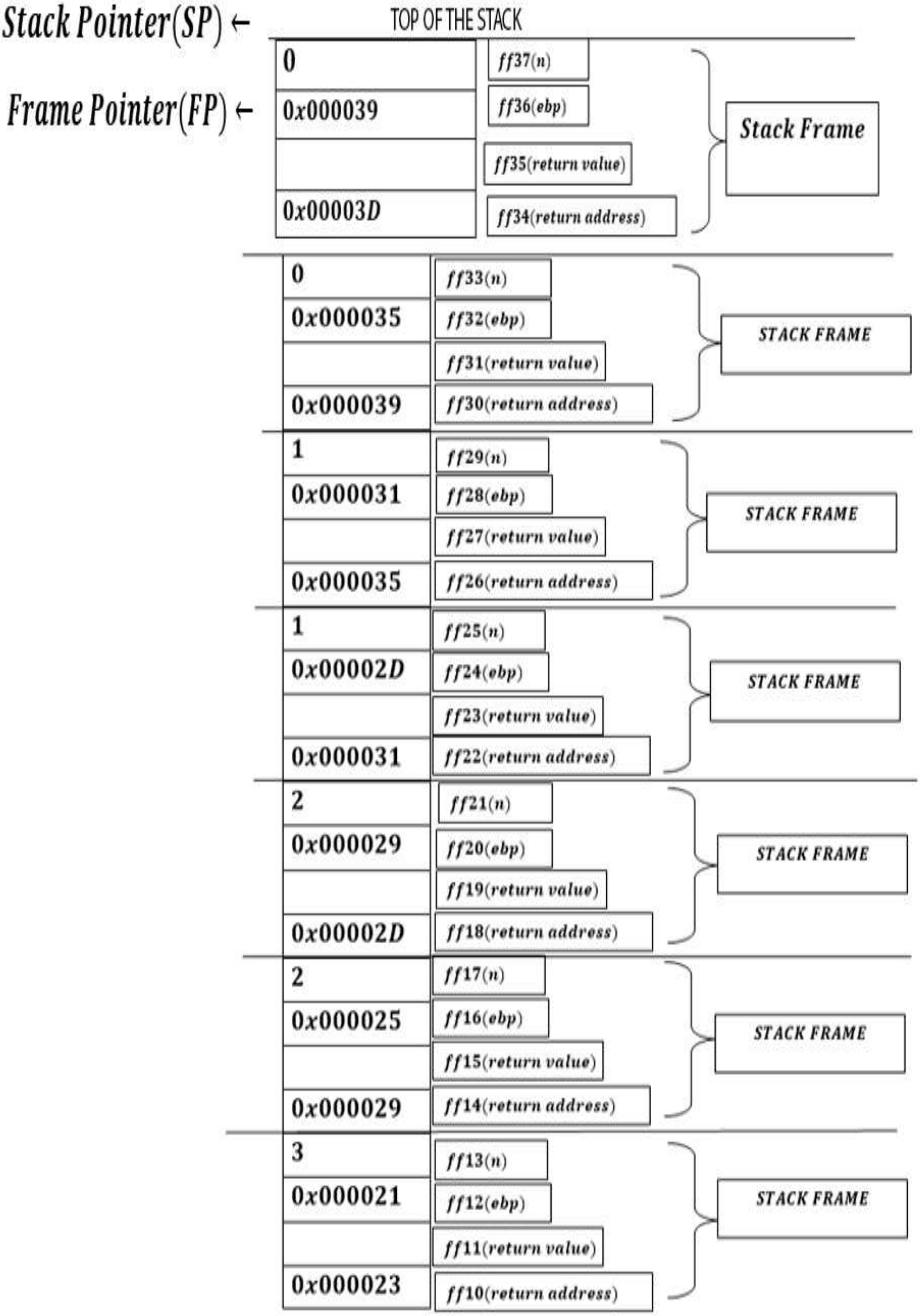
Now as we have two recursive calls. It will create two stack frames simultaneously i.e. $f(2)$ and $f(2)$.



Now as Frame Pointer pointing at 0x000029 activates the current stack frame. Hence for 0x000029 , two more stack frame will be created for f(1) and f(1).



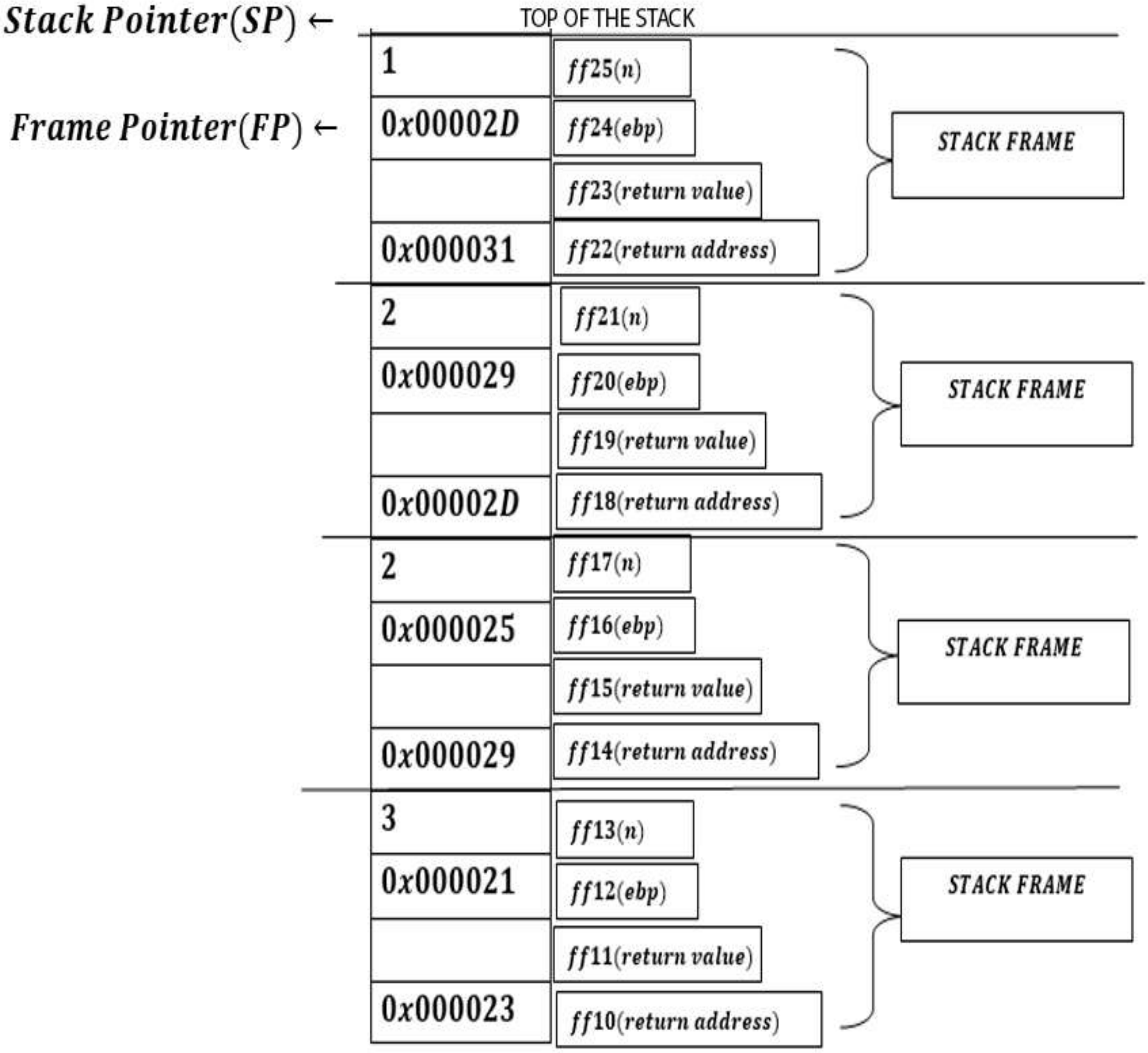
Now as Frame Pointer pointing at 0x000031 activates the current stack frame. Hence for 0x000031 , two more stack frame will be created for f(0) and f(0).


Stack Pointer(SP) ←
Frame Pointer(FP) ←
TOP OF THE STACK
0
0x000039
0x00003D
ff37(n)
ff36(ebp)
ff35(return value)
ff34(return address)
Stack Frame
0
0x000035
0x000039
ff33(n)
ff32(ebp)
ff31(return value)
ff30(return address)
STACK FRAME
1
0x000031
0x000035
ff29(n)
ff28(ebp)
ff27(return value)
ff26(return address)
STACK FRAME
1
0x00002D
0x000031
ff25(n)
ff24(ebp)
ff23(return value)
ff22(return address)
STACK FRAME
2
0x000029
0x00002D
ff21(n)
ff20(ebp)
ff19(return value)
ff18(return address)
STACK FRAME
2
0x000025
0x000029
ff17(n)
ff16(ebp)
ff15(return value)
ff14(return address)
STACK FRAME
3
0x000021
0x000023
ff13(n)
ff12(ebp)
ff11(return value)
ff10(return address)
STACK FRAME

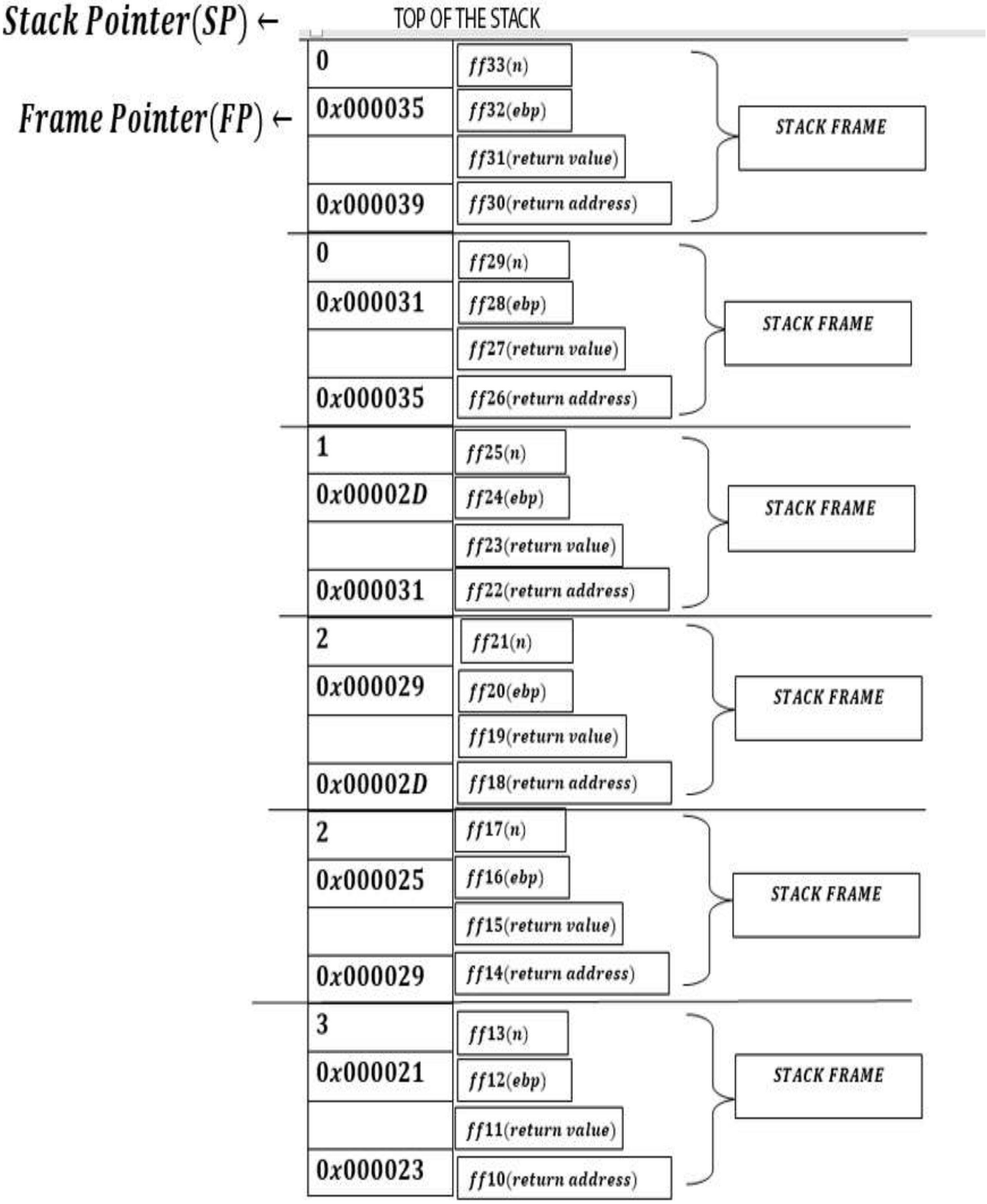
And now the curent activated stack is 0x000039 , of $f(0)$hence now Program Counter recieves the return address and next instruction is to pop out the current stack frame as recursion ends and base case i.e. if$(n > 0)$.

Next it will activate : 0x000035 of $f(0)$ frame pointer′s stack frame, PC (Program Counter) recieves address and next instruction will get the current stack frame popped out.

Next it will activate : 0x000031 of $f(1)$ frame pointer′s stack frame, PC (Program Counter)recieves address and next instruction will get the current stack frame popped out.



Next Frame Pointer will be* 0x00002D *activated and
PC(Program Counter)will have address* 0x000031 *and
thus through the next instruction by CPU ,
it will create again two stack frame for f*(1) *i.e.
***f*(0) *and f*(0).**



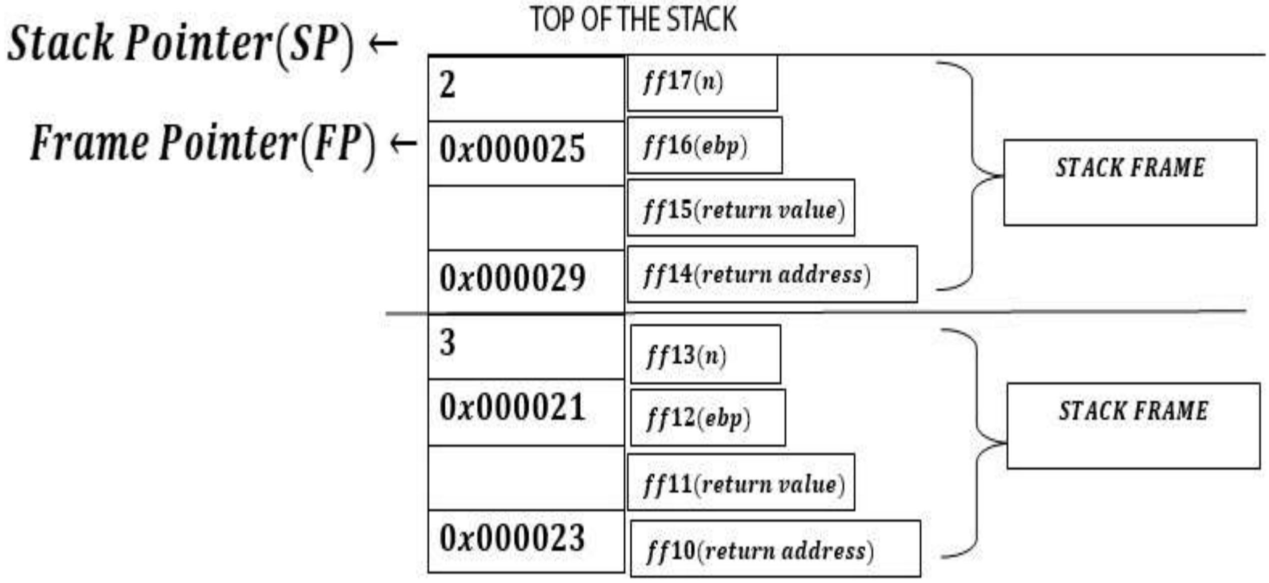
0	$ff33(n)$
$0x000035$	$ff32(ebp)$
	$ff31(return\ value)$
$0x000039$	$ff30(return\ address)$
0	$ff29(n)$
$0x000031$	$ff28(ebp)$
	$ff27(return\ value)$
$0x000035$	$ff26(return\ address)$
1	$ff25(n)$
$0x00002D$	$ff24(ebp)$
	$ff23(return\ value)$
$0x000031$	$ff22(return\ address)$
2	$ff21(n)$
$0x000029$	$ff20(ebp)$
	$ff19(return\ value)$
$0x00002D$	$ff18(return\ address)$
2	$ff17(n)$
$0x000025$	$ff16(ebp)$
	$ff15(return\ value)$
$0x000029$	$ff14(return\ address)$
3	$ff13(n)$
$0x000021$	$ff12(ebp)$
	$ff11(return\ value)$
$0x000023$	$ff10(return\ address)$

And now the curent activated stack is 0x000035 , of $f(0)$hence now Program Counter recieves the return address and next instruction is to pop out the current stack frame as recursion ends and base case i.e. if$(n > 0)$.

Next it will activate : 0x000031 of $f(0)$ frame pointer's stack frame, PC (Program Counter) recieves address and next instruction will get the current stack frame popped out.

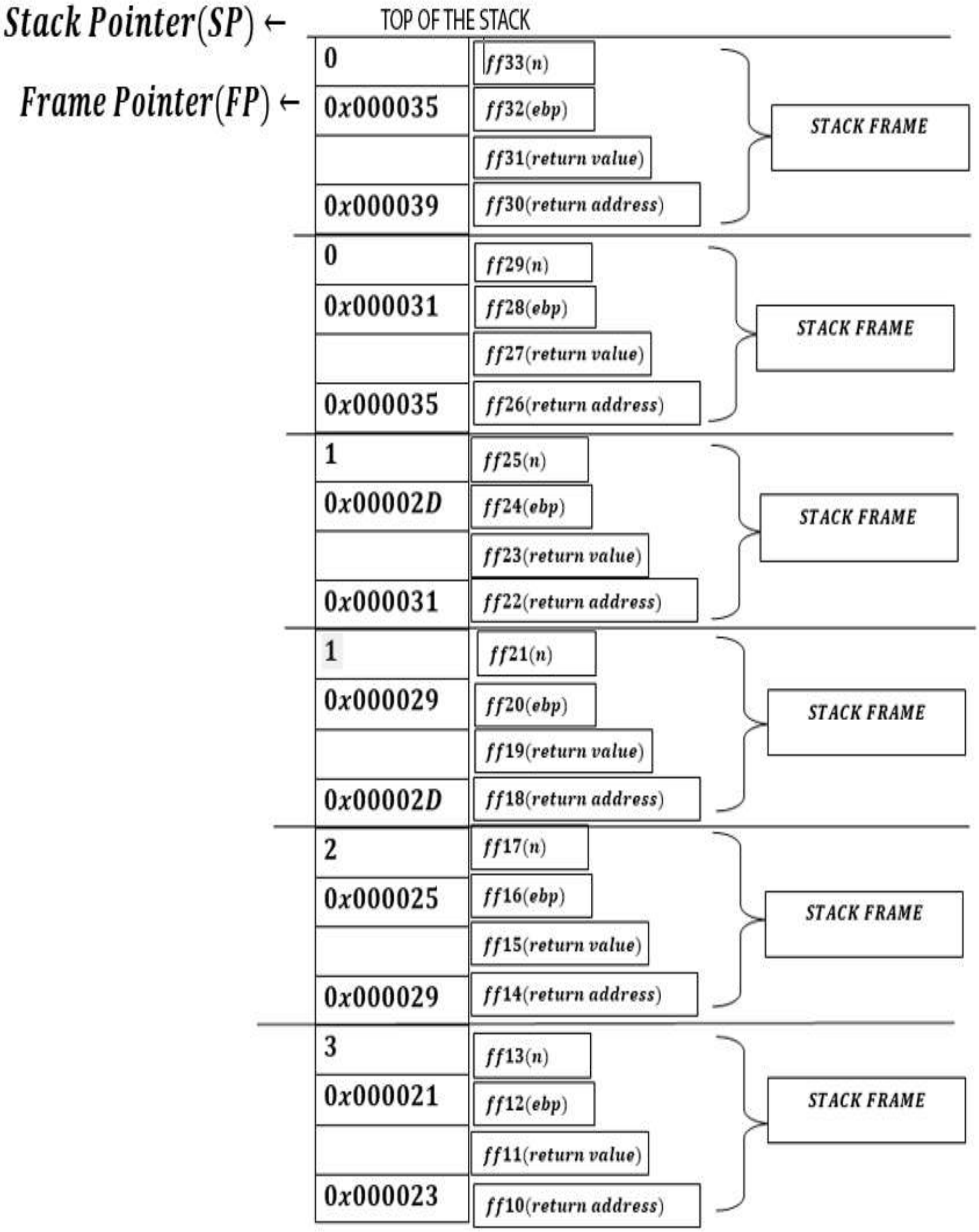
Next it will activate : 0x00002D of $f(1)$ frame pointer's stack frame, PC (Program Counter) recieves address and next instruction will get the current stack frame popped out.

Now $f(2)$ stack frame will also get popped out, hence it will activate : 0x000029 of $f(2)$ frame pointer's stack frame, PC (Program Counter) recieves address and next instruction will get the current stack frame popped out.



Next Frame Pointer will be* 0x000025 *activated and PC(Program Counter)will have address* 0x000029 *and thus through the next instruction by CPU ,

***it will create again two stack frame for* $f(2)$ *i.e.* $f(1)$ *and* $f(1)$ *and for stack frame for* $f(1)$ *,* $f(0)$*and* $f(0)$ *will be created also for another* $f(1)$*,* $f(0)$*and* $f(0)$ *will be created, as shown below*:**



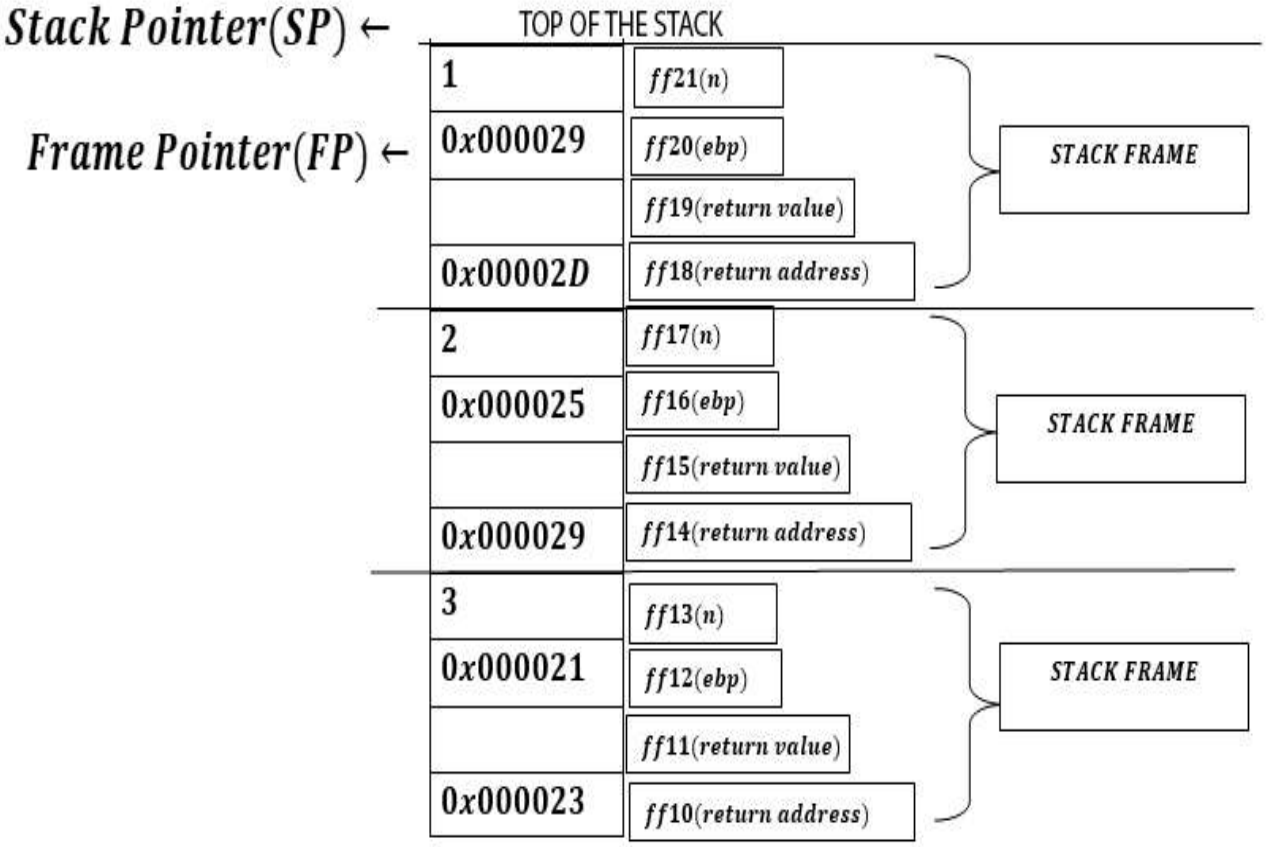
Stack Pointer(SP) ←

Frame Pointer(FP) ←

TOP OF THE STACK

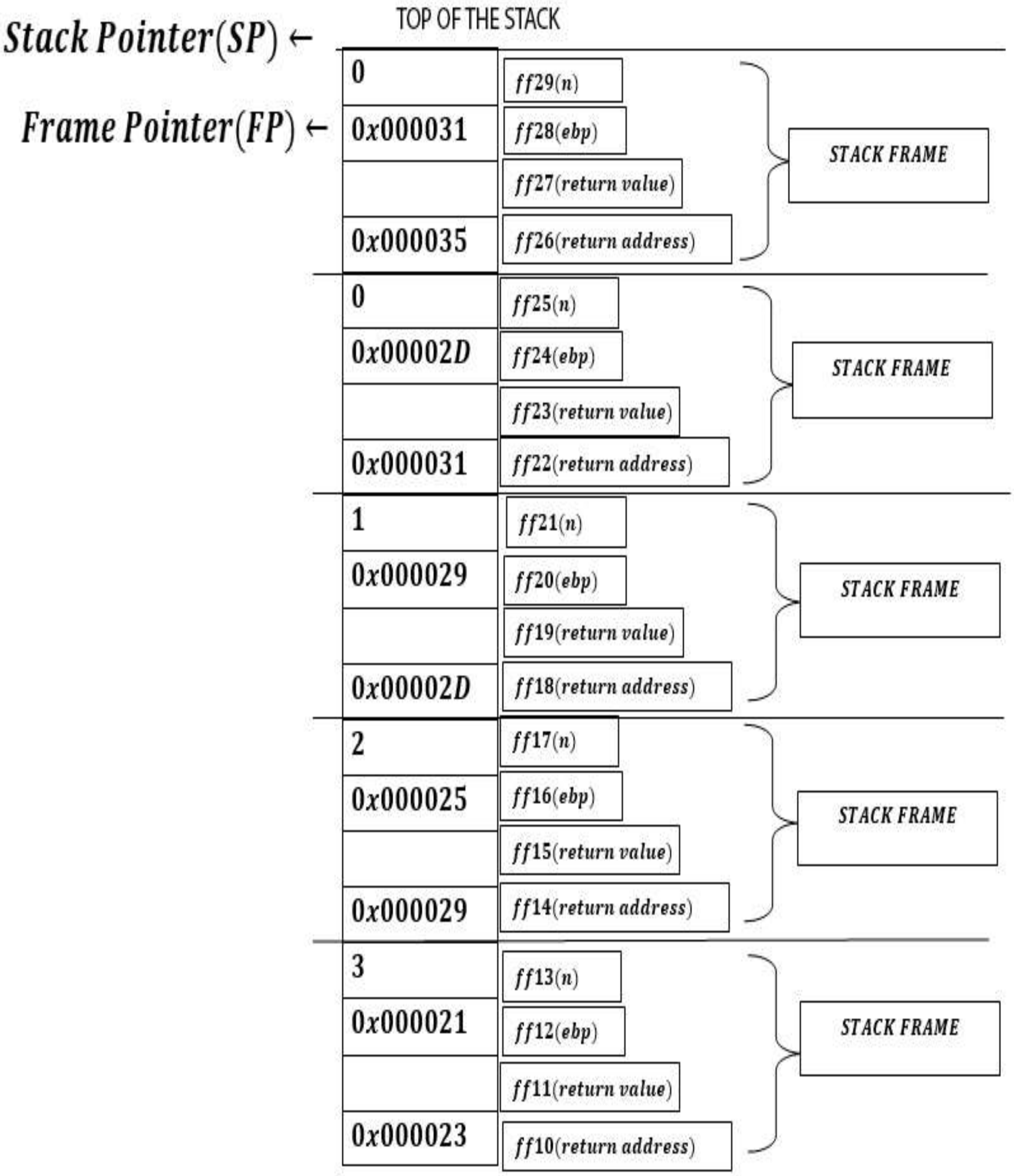
0	ff33(n)	
0x000035	ff32(ebp)	STACK FRAME
	ff31(return value)	
0x000039	ff30(return address)	
0	ff29(n)	
0x000031	ff28(ebp)	STACK FRAME
	ff27(return value)	
0x000035	ff26(return address)	
1	ff25(n)	
0x00002D	ff24(ebp)	STACK FRAME
	ff23(return value)	
0x000031	ff22(return address)	
1	ff21(n)	
0x000029	ff20(ebp)	STACK FRAME
	ff19(return value)	
0x00002D	ff18(return address)	
2	ff17(n)	
0x000025	ff16(ebp)	STACK FRAME
	ff15(return value)	
0x000029	ff14(return address)	
3	ff13(n)	
0x000021	ff12(ebp)	STACK FRAME
	ff11(return value)	
0x000023	ff10(return address)	

And then again stack frames $\boldsymbol{f(0), f(0), f(1)}$ ***will get popped out*** .



Again we will have $\boldsymbol{f(0)}$, $\boldsymbol{f(0)}$ ***for*** $\boldsymbol{f(1)}$ ***stack frame,***

as shown below:


Stack Pointer(SP) ←
Frame Pointer(FP) ←
TOP OF THE STACK
0
ff29(n)
0x000031
ff28(ebp)
ff27(return value)
0x000035
ff26(return address)
STACK FRAME
0
ff25(n)
0x00002D
ff24(ebp)
ff23(return value)
0x000031
ff22(return address)
STACK FRAME
1
ff21(n)
0x000029
ff20(ebp)
ff19(return value)
0x00002D
ff18(return address)
STACK FRAME
2
ff17(n)
0x000025
ff16(ebp)
ff15(return value)
0x000029
ff14(return address)
STACK FRAME
3
ff13(n)
0x000021
ff12(ebp)
ff11(return value)
0x000023
ff10(return address)
STACK FRAME

Now every stack frame will be popped out i.e.:

1st f(0) → will be popped out.

2nd f(0) → will be popped out.

3rd f(1) → will be popped out.

4rth f(2) → will be popped out.

5th f(3) → will be popped out.

And Cout will be executed at each time before the recursive calls takes place.

Hence what happens is :

1. Initial call: \`fun(3)\`

    - The cout statement \`cout ≪ n ≪ endl\`; will print 3 to the console.

    - The first recursive call \`fun(n − 1)\` will be made: \`fun(2)\`.

    - The second recursive call \`fun(n − 1)\` will be made: \`fun(2)\`.

2. ***Recursive call* 1:`*fun*(2)`**

- ***The cout statement* `*cout* ≪ *n* ≪ *endl*`;** ***will print* 2 *to the console*.**

- ***The first recursive call* `*fun*(*n* − 1)`** ***will be made*:`*fun*(1)`.**

- ***The second recursive call* `*fun*(*n* − 1)`** ***will be made*:`*fun*(1)`.**

3. ***Recursive call* 2:`*fun*(2)`**

- ***The cout statement* `*cout* ≪ *n* ≪ *endl*`;** ***will print* 2 *to the console*.**

- ***The first recursive call* `*fun*(*n* − 1)`** ***will be made*:`*fun*(1)`.**

- ***The second recursive call* `*fun*(*n* − 1)`** ***will be made*:`*fun*(1)`.**

4. ***Recursive call* 1:` *fun*(1)`**

- ***The cout statement cout* ≪ *n* ≪ *endl*;** ***will print* 1 *to the console*.**

- ***The first recursive call* `*fun*(*n* − 1)`** ***will be made*:`*fun*(0)`.**

- ***The second recursive call* `*fun*(*n* − 1)`** ***will be made*:` *fun*(0)`.**

5. ***Recursive call 1:` fun(1)`***

- ***The cout statement cout ≪ n ≪ endl;***
  will print 1 to the console.

- ***The first recursive call `fun(n − 1)`***
  will be made: `fun(0)`.

- ***The second recursive call `fun(n − 1)`***
  will be made: ` fun(0)`.

6. ***Recursive call 1: `fun(0)`***

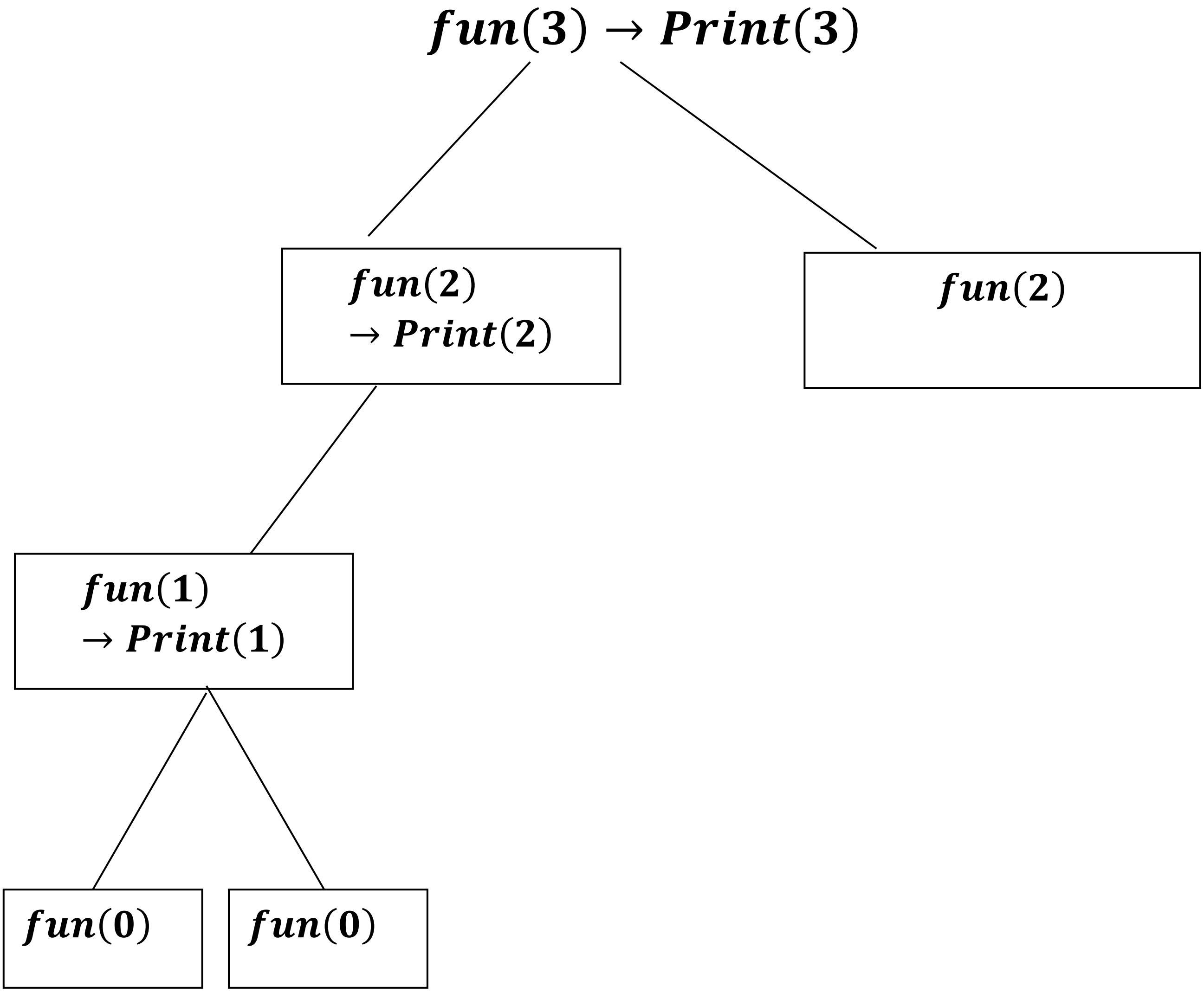
- **The base case is reached, and the function will not make any further recursive calls.**
  The cout statement will not be executed for this call.

7. **Recursive call 2: `fun(0)`**

- **The base case is reached, and the function will not make any further recursive calls.**
  The cout statement will not be executed for this call.

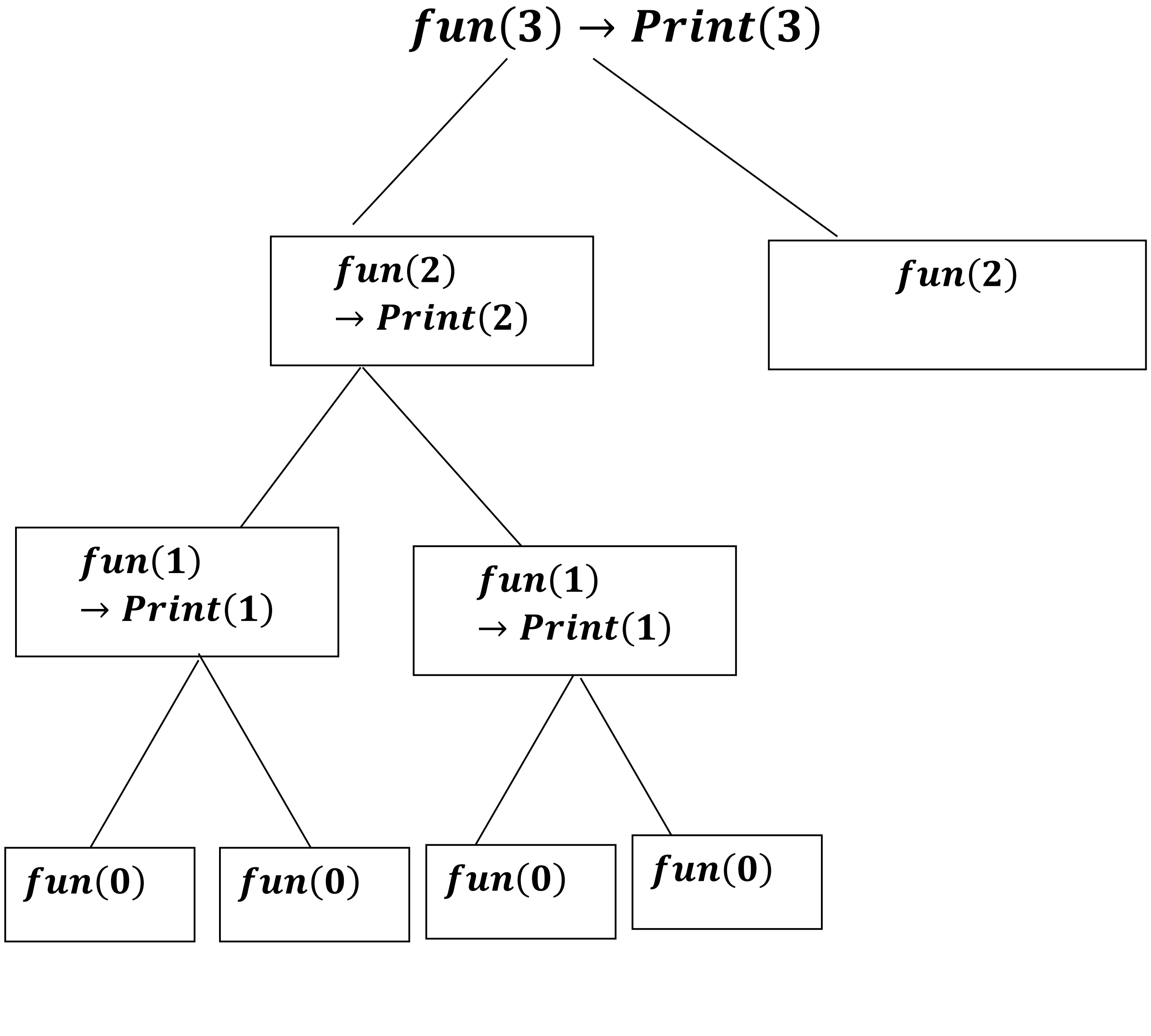
According to the stackframe , it will occur like this:

Part − 1



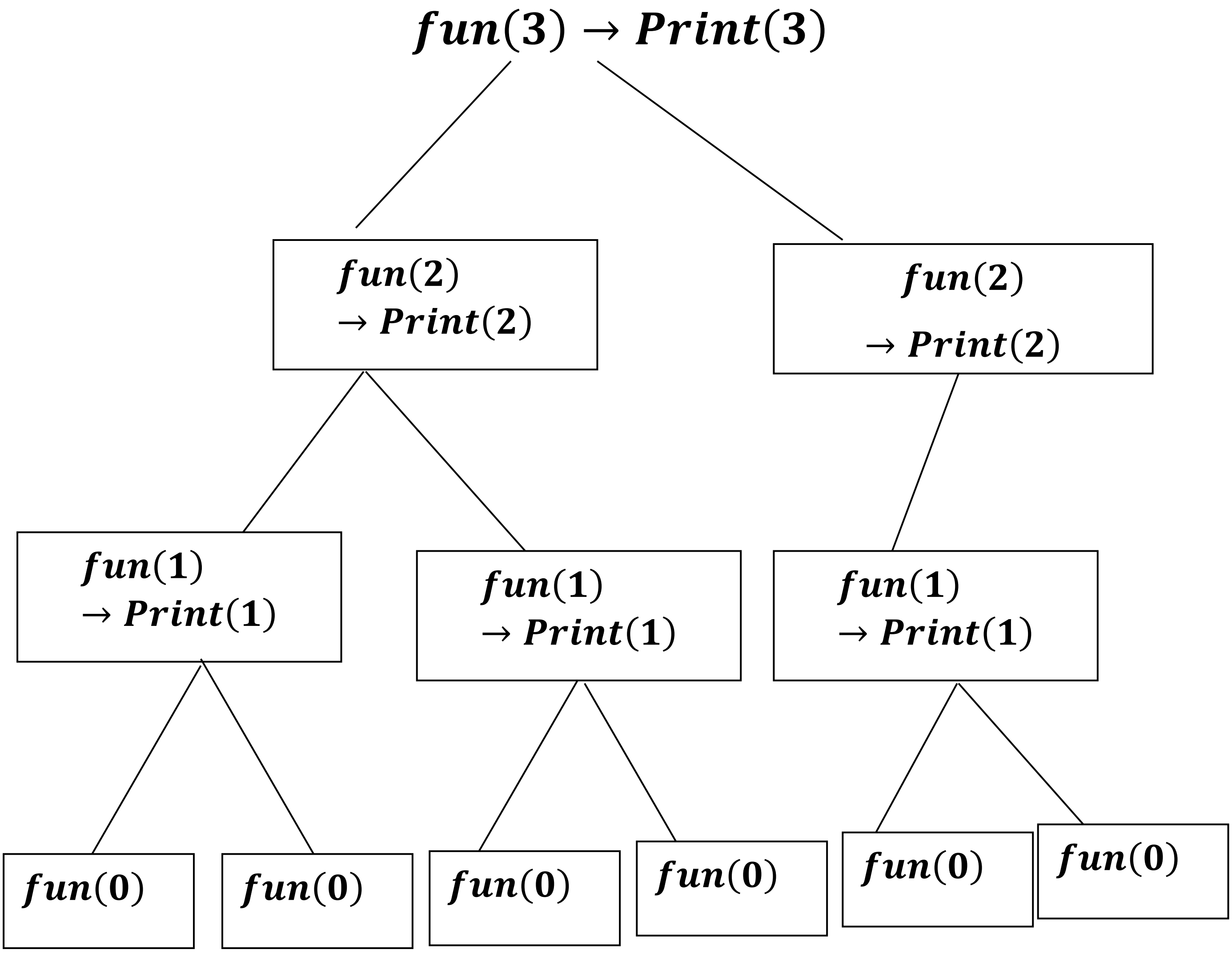
$fun(3) \rightarrow Print(3)$
$fun(2) \rightarrow Print(2)$
$fun(2)$
$fun(1) \rightarrow Print(1)$
$fun(0)$
$fun(0)$

$Part\ 2$

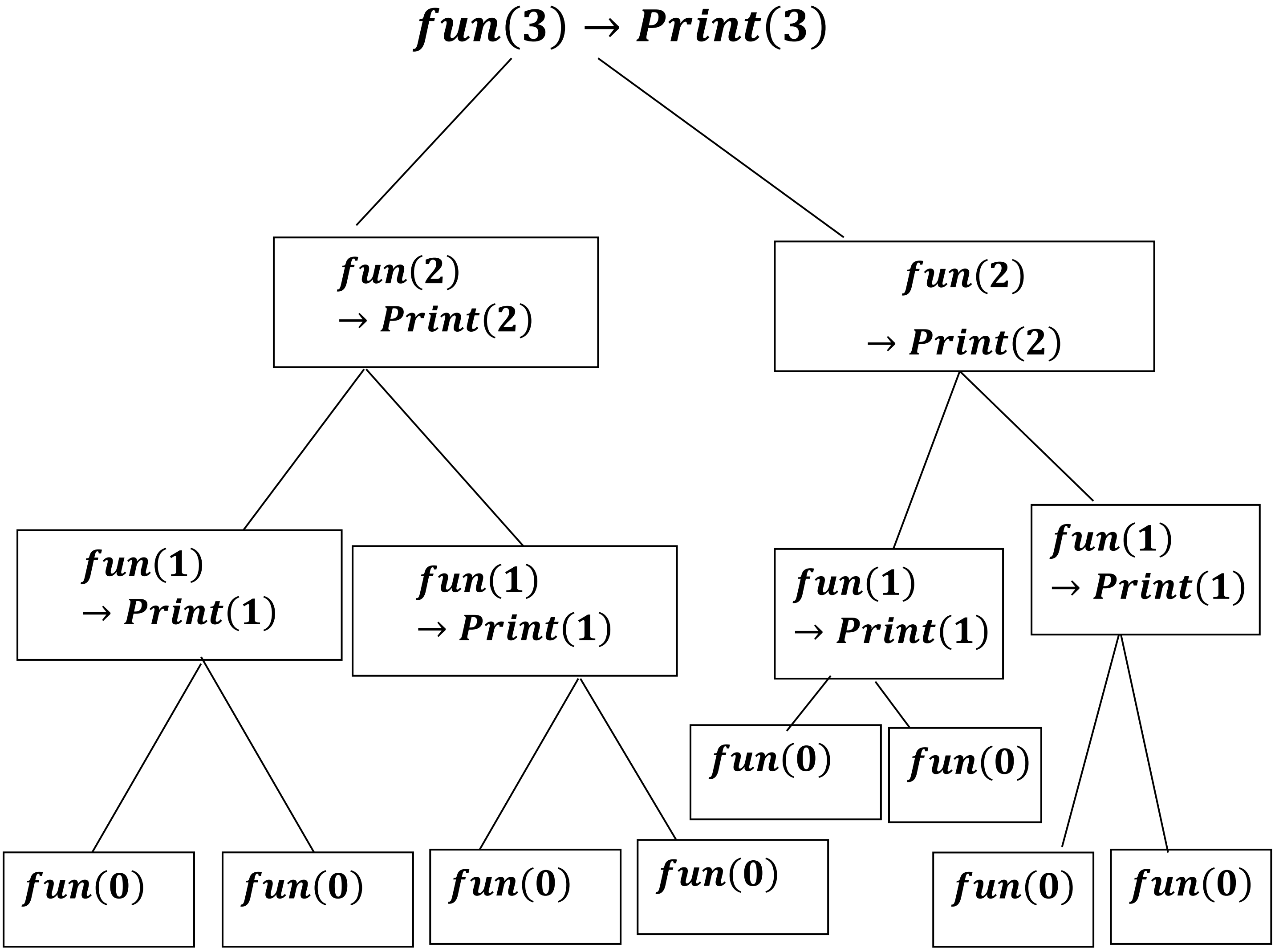


$fun(3) \rightarrow Print(3)$
$fun(2)$
$\rightarrow Print(2)$
$fun(2)$
$fun(1)$
$\rightarrow Print(1)$
$fun(1)$
$\rightarrow Print(1)$
$fun(0)$
$fun(0)$
$fun(0)$
$fun(0)$

$Part\ 3$


fun(3) → Print(3)
fun(2) → Print(2)
fun(2) → Print(2)
fun(1) → Print(1)
fun(1) → Print(1)
fun(1) → Print(1)
fun(0)
fun(0)
fun(0)
fun(0)
fun(0)
fun(0)

Part 4



Tree Recursion

****