C basic language

임베디드스쿨 2기

Lv1과정

2021. 04. 09

김효창

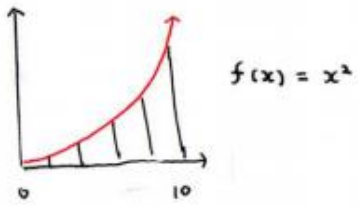
- 목 차 -

정적분

$T_n \simeq \int_a^b f(x)\,dx$ $\quad y = x^2$ 직사각형 5등분

곡선 아래 면적을 0 ~ 10 까지 추정 $\quad n = 5$

음영 영역의 면적을 추정

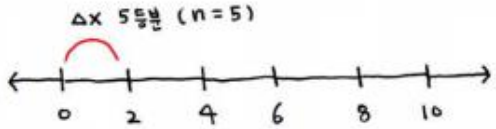


각 하위 구간의 폭인 $\Delta x = \frac{b-a}{n} = \frac{10-0}{5} = 2$

사다리꼴 공식

$T_n = \frac{\Delta x}{2}$

$T_n = \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) \cdots 2f(x_{n-1}) + f(x_n)\right]$

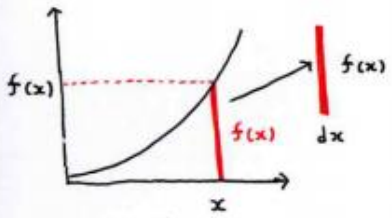


$T_5 = \frac{2}{2}\left[f(0) + 2f(2) + 2f(4) + 2f(6) + 2f(8) + f(10)\right]$ (x^2)

$= 8 + 32 + 72 + 128 + 100 = 340$

$\int_0^{10} x^2 dx = \frac{x^3}{3}\Big|_0^{10} = \frac{10^3}{3} - \frac{0^3}{3} = \frac{1000}{3} = 333.3$

$\int_a^b f(x)\,dx$ "x를 a부터 b까지 변화시키면서 f(x)에 dx를 곱한 것들 전부 합쳐라"



f(x)는 빨간색 선의 높이

Δx 는 어떤 구간에서의 변화량

dx 는 밑변의 길이

x의 순간 변화량

delta (Δ) : 변화

변수 x가 물체의 움직임을 나타내는 경우

Δx 는 움직임의 변화

10진수 "100" → "25" (↓ 1/4 배)

N 비트를 왼쪽 시프트 하면 2^N 배 곱셈이 된다 (오버플로 주의)

오른쪽 " $1/2^N$ 배 나눗셈이 된다

Fibonacci number

Fibonacci Sequence

$F_0 = 0 \quad F_1 = 1 \quad F_n = F_{n-1} + F_{n-2} \quad n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377

1.618 1.618

0.618 0.618

F_0 F_1 F_5 F_{10}

610, 987, 1597, 2584, 4181, 6765

F_{20}

$\frac{1}{1} = 1, \quad \frac{2}{1} = 2, \quad \frac{5}{3} = 1.667, \quad \frac{21}{13} = 1.61538$

$\frac{55}{34} = 1.61765, \quad \frac{89}{55} = 1.61818, \quad \frac{233}{144} = 1.61805$

1.618

$F^n = F^{n-1} + F^{n-2}$

$\frac{F^n}{F^n} \quad \frac{F^{n-1}}{F^n} \quad \frac{F^{n-2}}{F^n}$

$(n-1) - (n)$

$n-1-n = -1$

$(1 = F^{-1} + F^{-2})F^2$

$F^2 = F^1 + 1$

$F^2 - F - 1 = 0$

$aF^2 + bF + c = 0$

$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

$r = \frac{-(-1) \pm \sqrt{(-1)^2 - 4(1)(-1)}}{2(1)}$

$r = \frac{1 \pm \sqrt{1+4}}{2}$

$\frac{\sqrt{5}+1}{2} = 1.618 \quad \frac{\sqrt{5}-1}{2} = 0.618 \quad \frac{1}{1.618} = 0.618$

$F_{20} \simeq F_{12} \left(\frac{\sqrt{5}+1}{2} \right)^8$

$144 (1.618)^8 = 6764.935$

$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}} \qquad F_{20} = \frac{(1\pm\sqrt{5})^{20} - (1-\sqrt{5})^{20}}{2^{20}\sqrt{5}} = 6765$

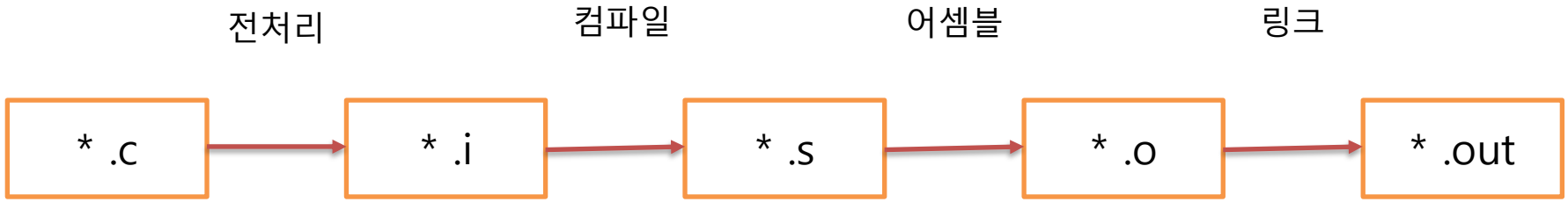
$F_{13} \simeq \sqrt{F_{12} \cdot F_{14}} = \sqrt{144 \cdot 377} = 232.997$

preprocessor

지시자	의미
#define	매크로 정의
#include	파일 포함
#undef	매크로 정의 해제
#if, #else, #endif	조건에 따른 컴파일
#ifedf, #endif	매크로가 정의되어 있는 경우의 컴파일
#ifndef, #endif	매크로가 정의되어 있지 않은 경우의 컴파일
#line	행 번호 출력

매크로 : 자주 사용하는 여러 개의 명령어를 묶어 하나의 새로운 명령어로 만드는 방법
함수를 사용하지 않고도 간단한 함수의 기능을 이용

함수 매크로 : 매크로가 함수처럼 매개 변수를 가지는 것 , 스택에 저장하지 않기 때문에 실행 속도가 빠르다
함수 구현이 어렵다, 소스 파일 크기가 커져서 가독성이 어렵다



기타

gdb –q *.out 를 입력하면 실행 시 시스템 내용을 생략할 수 있다

opcode 는 명령어

operand 명령어로 만든 인자 값이나 피연산자 , cpu 가 사용하는 변수

GDB 에서 메모리를 조사하는 방법

examine 의 줄임 명령어인 x를 사용하여 메모리를 조사한다.

[p or x] / [개수] [출력형식] - 주소 값 출력

표현 방식

o : 8진수 , x : 16진수 , u : 10진수 , t : 2진수 , a : 주소 , s : 문자열 , c : ascii 문자 표시 , f: floating point

표현 방식 + 데이터 크기

b : 단일바이트 , h : halfword 2바이트 , w : double word 4바이트 , g : long word 8바이트

i 를 통해서는 어셈블리 언어의 명령 메모리를 볼 수 있다.
x/i $rip : rip를 기준으로 cpu에 명령 내용을 보여줌

기타

파란색 값은 해당 함수가 어느 메모리 주소에 위치했는지를 표시한 것 (16진수 표기)

<+0> , <+4> , <+5> , <+8>

첫 번째 명령줄을 0으로 기준을 두었을 때 0으로부터 주소가 얼마만큼 떨어져 있는지를 표시한 것

ESI (Extended Source Index) 레지스터가 지시하는 주소에 있는 데이터가 EDI (Extended Destination Index)
에 복사가 되는 형태 (printf 출력)

ESI = 0x11 , ESI = 45

LEA : 0x11 주소값을 저장, MOV : 45 값을 저장

스택의 용도

- 지역 변수의 저장
- 매개변수 전달
- 임시 데이터 백업
- 함수 호출, 복귀 정보 저장

기타

Stack Frame

스택에 저장되는 함수의 호출 정보를 스택 프레임(Stack Frame)이라 하며, 이러한 스택 프레임에는 함수로 전달되는 인수 와 함수 실행을 모두 마치면 돌아올 복귀 주소, 지역 변수 등의 정보가 들어간다.

함수 호출 시 생성되고 복귀 시 삭제한다.

함수는 Stack Frame을 독립적으로 사용한다.

NOP 는 PipeLine 기법 내에서 1 Cycle을 아무 것도 처리하지 않고 쉬어가도록 의도적으로 지정한 것

CPU 가 쉬어간다 ← 명령어 처리 부분을 낭비한다는 의미

NOP를 의도적으로 사용한 경우, Timing 적인 Delay를 유발할 때 사용

Branch , Jump , Call 은 모두 다른 위치로 넘어가서 실행을 하게 되는 명령어

상기 과정에서 다음 명령어는 실행되는 도중에 다른 라인으로 이동하기 때문에 파이프라인 구조에서 규격대로 연산이 수행되지 않게 될 수 있다. 영향을 받을 수도 있기에 NOP 가 Default 로 들어가는 경우도 있다

실제 Cycle 을 계산하면 어셈블리 코드 라인 수보다 Cycle 이 많이 Count 될 수도 있다

가상메모리를 제어할 때만 bit 단위

어셈블리어에서 보고 있는 것들은 (기계어의 동작과정) byte 단위이다

기타

레지스터

CPU가 요청을 처리하는 데 필요한 데이터(명령어의 종류, 연산결과, 복귀주소 등)를 일시적으로 저장하는 기억장치

RAX (Accumulator) : 더하기, 빼기 등 산술/논리 연산을 수행, 함수의 return값 저장

시스템콜 함수를 사용하려면 RAX에 함수의 syscall 번호를 넣어준다.

RBX (Base) : 메모리 주소를 저장하기 위한 용도로 사용

RCX (Count) : 반복문에서 카운터로 사용되는 레지스터. for문의 i 와 같은 역할

ECX는 미리 반복 값을 정해두고 명령어를 사용할 때마다 값이 하나씩 줄어든다

syscall을 호출했던 사용자프로그램의 return 주소를 가진다.

RDX (Data) : 다른 레지스터를 서포트하는 여분의 레지스터. 큰 수의 곱셈이나 나눗셈 연산에서 EAX와 함께 사용

je : 값이 같으면 점프
jne : 값이 다르면 점프
jg : 왼쪽 인자 값이 오른쪽보다 크면 점프
jl : 왼쪽 인자 값이 오른쪽보다 작으면 점프
jge : 왼쪽 값이 오른쪽보다 크거나 같으면 점프
jle : 왼쪽 값이 오른쪽보다 작거나 같으면 점프

기타

```
hyochangkim@hyochangkim-ThinkPad-X390-Yoga:~/proj/es02/Lv01-02/HyochangKim/C$ gdb -q fibw
Reading symbols from fibwork.out...
(gdb) b *0x000055555555524b
Breakpoint 1 at 0x55555555524b
(gdb) r
Starting program: /home/hyochangkim/proj/es02/Lv01-02/HyochangKim/C/fibwork.out
몇 번째 피보나치 항을 구할까요 ? 5

Breakpoint 1, 0x000055555555524b in main () at fibwork.c:85
85              res = recursive_fib(num);
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555204 <+0>:     endbr64
   0x0000555555555208 <+4>:     push   %rbp
   0x0000555555555209 <+5>:     mov    %rsp,%rbp
   0x000055555555520c <+8>:     sub    $0x10,%rsp
   0x0000555555555210 <+12>:    mov    %fs:0x28,%rax
   0x0000555555555219 <+21>:    mov    %rax,-0x8(%rbp)
   0x000055555555521d <+25>:    xor    %eax,%eax
   0x000055555555521f <+27>:    lea    0xe0a(%rip),%rdi        # 0x555555556030
   0x0000555555555226 <+34>:    mov    $0x0,%eax
   0x000055555555522b <+39>:    callq  0x5555555550a0 <printf@plt>
   0x0000555555555230 <+44>:    lea    -0x10(%rbp),%rax
   0x0000555555555234 <+48>:    mov    %rax,%rsi
   0x0000555555555237 <+51>:    lea    0xe21(%rip),%rdi        # 0x55555555605f
   0x000055555555523e <+58>:    mov    $0x0,%eax
   0x0000555555555243 <+63>:    callq  0x5555555550b0 <__isoc99_scanf@plt>
   0x0000555555555248 <+68>:    mov    -0x10(%rbp),%eax
=> 0x000055555555524b <+71>:    mov    %eax,%edi
```

중단점 사용

b func : 함수에 중단점 설정
b *0x0000524b : 주소 값에 중단점 설정

```
hyochangkim@hyochangkim-ThinkPad-X390-Yoga:~/proj/es02/Lv01-02/HyochangKim/C$ gdb -q fibwork.out
Reading symbols from fibwork.out...
(gdb) b recursive_fib
Breakpoint 1 at 0x11a9: file fibwork.c, line 62.
(gdb) r
Starting program: /home/hyochangkim/proj/es02/Lv01-02/HyochangKim/C/fibwork.out
몇 번째 피보나치 항을 구할까요 ? 5

Breakpoint 1, recursive_fib (num=21845) at fibwork.c:62
62      {
(gdb) return
Make recursive_fib return now? (y or n) y
#0  0x0000555555555252 in main () at fibwork.c:85
85              res = recursive_fib(num);
(gdb) disas
Dump of assembler code for function main:
   0x0000555555555204 <+0>:     endbr64
   0x0000555555555208 <+4>:     push   %rbp
   0x0000555555555209 <+5>:     mov    %rsp,%rbp
   0x000055555555520c <+8>:     sub    $0x10,%rsp
   0x0000555555555210 <+12>:    mov    %fs:0x28,%rax
   0x0000555555555219 <+21>:    mov    %rax,-0x8(%rbp)
   0x000055555555521d <+25>:    xor    %eax,%eax
   0x000055555555521f <+27>:    lea    0xe0a(%rip),%rdi        # 0x555555556030
   0x0000555555555226 <+34>:    mov    $0x0,%eax
   0x000055555555522b <+39>:    callq  0x5555555550a0 <printf@plt>
   0x0000555555555230 <+44>:    lea    -0x10(%rbp),%rax
   0x0000555555555234 <+48>:    mov    %rax,%rsi
   0x0000555555555237 <+51>:    lea    0xe21(%rip),%rdi        # 0x55555555605f
```

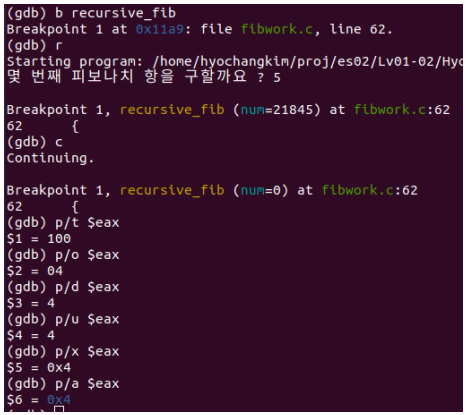
return

현재 함수를 실행하지 않고 탈출

기타

```
hyochangkim@hyochangkim-ThinkPad-X390-Yoga:~/proj/es02/Lv01-02/HyochangKim/C$ gdb -q fibwork.out
Reading symbols from fibwork.out...
(gdb) b main
Breakpoint 1 at 0x1204: file fibwork.c, line 79.
(gdb) r
Starting program: /home/hyochangkim/proj/es02/Lv01-02/HyochangKim/C/fibwork.out

Breakpoint 1, main () at fibwork.c:79
79      {
(gdb) p recursive_fib
$1 = {int (int)} 0x5555555551a9 <recursive_fib>
```



p recursive_fib

함수의 주소를 확인한다

for 문

(gdb) u 입력 시 printf 출력 발생

기타

```
58      num
59      */
60
61      int recursive_fib(int num)
62      {
63              if(num <= 0)
64              {
65                      printf("올바른 값을 입력하세요!\n");
66                      return -1;
(gdb) list 5
1       #include <stdio.h>
2
3       // recursive_fib(6)   -> 1.recursive_fib(5) + 2.recursive_fib(4)
4       // 1.recursive_fib(5) -> 1.recursive_fib(4) + 2.recursive_fib(3)
5       // 1.recursive_fib(4) -> 1.recursive_fib(3) + 2.recursive_fib(2)
6       // 1.recursive_fib(3) -> 1.recursive_fib(2) + 2.recursive_fib(1)
7       // 1.recursive_fib(2) -> 1
8       // 2.recursive_fib(1) -> 1
9       // 1.recursive_fib(3) -> 2
10      // 2.recursive_fib(2) -> 1
(gdb) list 57
52      num      fib(2) + fib(1)
53      fib(2) = 1 ---------------------------------------------
54       2
55      num
56      fib(1) = 1 ---------------------------------------------
57       1
58      num
59      */
60
61      int recursive_fib(int num)
(gdb) list 85
80              int num, res;
81
82              printf("몇 번째 피보나치 항을 구할까요 ? ");
83              scanf("%d", &num);
84
85              res = recursive_fib(num);
86              printf("res = %d\n", res);
87
88              return 0;
89      }
(gdb) l
Line number 90 out of range; fibwork.c has 89 lines.
```

```
hyochangkim@hyochangkim-ThinkPad-X390-Yoga:~/proj/es02/Lv01-02/HyochangKim/C$ gdb -q fibwork.out
Reading symbols from fibwork.out...
(gdb) b main
Breakpoint 1 at 0x1204: file fibwork.c, line 79.
(gdb) r
Starting program: /home/hyochangkim/proj/es02/Lv01-02/HyochangKim/C/fibwork.out

Breakpoint 1, main () at fibwork.c:79
79      {
(gdb) x/c 0x555555556062
0x555555556062: 114 'r'
(gdb)
0x555555556063: 101 'e'
(gdb)
0x555555556064: 115 's'
(gdb)
0x555555556065: 32 ' '
(gdb)
0x555555556066: 61 '='
(gdb)
0x555555556067: 32 ' '
(gdb)
0x555555556068: 37 '%'
(gdb)
0x555555556069: 100 'd'
```

소스 확인

l : main 함수 주변 내용 출력
l 5 : 10 행 주변 내용 출력 1 ~10

printf 문자 확인

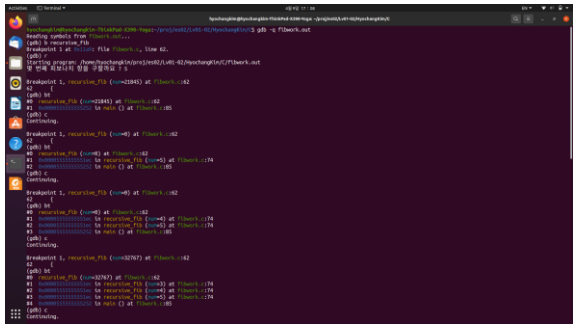
lea , 0x00(%rip), %rdi , #0x6062

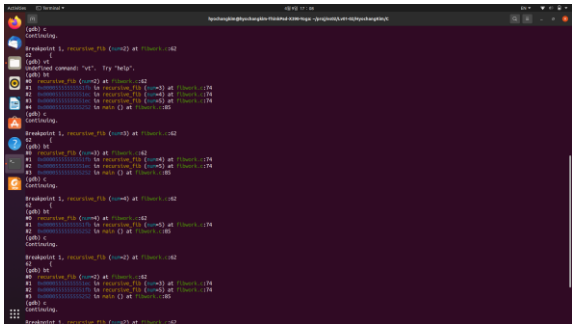
기타 (rbx ....)

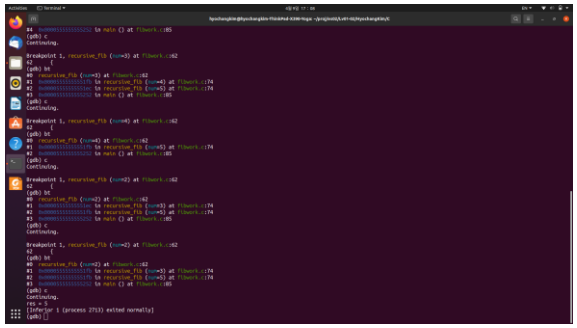
```
(gdb) x $rbx
0x555555555290 <__libc_csu_init>:       0xfa1e0ff3
```

```
Dump of assembler code for function __libc_csu_init:
=> 0x0000555555555290 <+0>:     endbr64
   0x0000555555555294 <+4>:     push   %r15
   0x0000555555555296 <+6>:     lea    0x2b03(%rip),%r15        # 0x555555557da0
   0x000055555555529d <+13>:    push   %r14
   0x000055555555529f <+15>:    mov    %rdx,%r14
   0x00005555555552a2 <+18>:    push   %r13
   0x00005555555552a4 <+20>:    mov    %rsi,%r13
   0x00005555555552a7 <+23>:    push   %r12
   0x00005555555552a9 <+25>:    mov    %edi,%r12d
   0x00005555555552ac <+28>:    push   %rbp
   0x00005555555552ad <+29>:    lea    0x2af4(%rip),%rbp        # 0x555555557da8
   0x00005555555552b4 <+36>:    push   %rbx
   0x00005555555552b5 <+37>:    sub    %r15,%rbp
   0x00005555555552b8 <+40>:    sub    $0x8,%rsp
   0x00005555555552bc <+44>:    callq  0x555555555000 <_init>
   0x00005555555552c1 <+49>:    sar    $0x3,%rbp
   0x00005555555552c5 <+53>:    je     0x5555555552e6 <__libc_csu_init+86>
   0x00005555555552c7 <+55>:    xor    %ebx,%ebx
   0x00005555555552c9 <+57>:    nopl   0x0(%rax)
   0x00005555555552d0 <+64>:    mov    %r14,%rdx
   0x00005555555552d3 <+67>:    mov    %r13,%rsi
   0x00005555555552d6 <+70>:    mov    %r12d,%edi
   0x00005555555552d9 <+73>:    callq  *(%r15,%rbx,8)
   0x00005555555552dd <+77>:    add    $0x1,%rbx
   0x00005555555552e1 <+81>:    cmp    %rbx,%rbp
   0x00005555555552e4 <+84>:    jne    0x5555555552d0 <__libc_csu_init+64>
   0x00005555555552e6 <+86>:    add    $0x8,%rsp
   0x00005555555552ea <+90>:    pop    %rbx
   0x00005555555552eb <+91>:    pop    %rbp
   0x00005555555552ec <+92>:    pop    %r12
   0x00005555555552ee <+94>:    pop    %r13
   0x00005555555552f0 <+96>:    pop    %r14
   0x00005555555552f2 <+98>:    pop    %r15
   0x00005555555552f4 <+100>:   retq
End of assembler dump.
```

기타 (재귀 함수 스택 쌓이는 화면)







어셈블리 분석 (test_fun)

Double word

```
0x15b   <+0>    endbr64
0x15f   <+4>    push   %rbp                          메모리를 0x10 16 Byte 지역변수 공간 할당
0x160   <+5>    mov    %rsp, %rbp
0x163   <+8>    sub    $0x10, %rsp
0x167   <+12>   movl   $0x3, -0x8(%rbp)             rbp 기준 -8 Byte 지점에 num 선언 후 3 저장
0x16e   <+19>   mov    -0x8(%rbp), %eax             eax = num , 값을 저장
0x171   <+22>   mov    %eax, %edi                   edi = eax        res =my_func(num);  의 num 을 int my_func(int num)으로 전달
0x173   <+24>   callq  0x149 <my_fuc>               0x149 함수 호출
0x178   <+29>   mov    %eax, -0x4(%rbp)             0x4 = eax , 1 값을 저장
0x17b   <+32>   mov    -0x4(%rbp), %eax             eax = 0x4
0x17e   <+35>   mov    %eax, %esi                   esi = eax
0x180   <+37>   lea    0xe7d(%rip), %rdi    #0x6004 rdi = 0xe7d , 주소 값을 저장     스트림 값을 rip + 0xe7d의 주소에 있는 것을 가진다
0x187   <+44>   mov    $0x0, %eax
0x18c   <+49>   callq  0x050<printf@plt>            printf 수행
0x191   <+54>   mov    $0x0, %eax
0x196   <+59>   leaveq                              mov rsp, rbp     pop rbp   할당했던 공간을 회수 기존의 rbp 위치로 돌아온다
0x197   <+60>   retq                                pop eip                    다음 실행할 명령어의 주소를 불러온다
```

8바이트를 메모리에 저장하거나 음수값을 설정할 수 있다

```
0x149   <+0>    endbr64
0x14d   <+4>    push   %rbp
0x14e   <+5>    mov    %rsp, %rbp
0x151   <+8>    mov    %edi, -0x4(%rbp)     0x4 = edi
0x154   <+11>   mov    -0x4(%rbp), %eax     eax = 0x4
0x157   <+14>   sar    %eax                 3을 >> 수행하여 eax 에 저장          함수의 리턴값을 가진다
0x159   <+16>   pop    %rbp                 현재 rsp에서 값을 추출하여 main 함수 메모리에 값을 전달한다 ( main -0x4(%rbp) )
0x15a   <+17>   retq                        call로 호출된 함수 종료, call 다음 명령줄로 이동
```

test _ fun

어셈블리 분석 (if)

```
0x1a9   <+0>    endbr64
0x1ad   <+4>    push   %rbp                          메모리를 0x10 16 Byte 할당
0x1ae   <+5>    mov    %rsp, %rbp
0x1b1   <+8>    sub    $0x10, %rsp
0x1b5   <+12>   mov    %fs:0x28, %rax                fs:0x28에서 canary 값을 가져와 rax에 저장해킹 방지
0x1be   <+21>   mov    %rax, -0x8(%rbp)
0x1c2   <+25>   xor    %eax, %eax                    xor 2 Byte, mov 3 Byte          mov 보다 크기가 작다
0x1c4   <+27>   lea    0xe39(%rip), %rdi    #0x6004
0x1cb   <+34>   mov    $0x0, %eax
0x1d0   <+39>   callq  0x080<printf@plt>
0x1d5   <+44>   lea    -0xc(%rbp), %rdx
0x1d9   <+48>   lea    -0x10(%rbp), %rax
0x1dd   <+52>   mov    %rax, %rsi
0x1e0   <+55>   lea    0xe2a(%rip), %rdi    #0x6011
0x1e7   <+62>   mov    $0x0, %eax
0x1ec   <+67>   callq  0x090<scanf@plt>
0x1f1   <+72>   mov    -0xc(%rbp), %edx              edx = 0xc scanf a 값 저장
0x1f4   <+75>   mov    -0x10(%rbp), %eax             eax = 0x10 scanf b 값 저장
0x1f7   <+78>   mov    %edx, %esi                    esi = edx
0x1f9   <+80>   mov    %eax, %edi                    edi = eax
0x1fb   <+82>   callq  0x189<if_test>                0x189 함수 호출
0x200   <+87>   mov    %eax, %esi                    esi = eax( a 또는 b)
0x202   <+89>   lea    0xe0e(%rip), %rdi    #0x6017
0x209   <+96>   mov    $0x0, %eax                    printf 수행
0x20e   <+101>  callq  0x080<printf@plt>
0x213   <+106>  mov    $0x0, %eax
0x218   <+111>  mov    -0x8(%rbp), %rcx
0x21c   <+115>  xor    %fs:0x28, %rcx
0x225   <+124>  je     0x22c<main+131>               두 값이 같으면 <main+131>로 점프
0x227   <+126>  callq  0x070<fail@plt>
0x22c   <+131>  leaveq
0x22d   <+132>  retq


0x189   <+0>    endbr64
0x18d   <+4>    push   %rbp
0x18e   <+5>    mov    %rsp, %rbp
0x191   <+8>    mov    %edi, -0x4(%rbp)              0x4 = edi , scanf a 값 저장
0x194   <+11>   mov    %esi, -0x8(%rbp)              0x8 = esi , scanf b 값 저장
0x197   <+14>   mov    -0x4(%rbp), %eax              eax = 0x4
0x19a   <+17>   cmp    -0x8(%rbp), %eax              0x8에 저장된 값과 eax 비교
0x19d   <+20>   jle    0x1a4<if_test+27>             eax 값이 작거나 같으면 27번으로 이동
0x19f   <+22>   mov    -0x4(%rbp),%eax               eax = a값 저장 후 <+30> 점프
0x1a2   <+25>   jmp    0x1a7<if_test+30>
0x1a4   <+27>   mov    -0x8(%rbp), %eax              eax = b 값 저장
0x1a7   <+30>   pop    %rbp                          할당했던 공간을 회수 기존의 rbp 위치로 돌아온다
0x1a8   <+31>   retq                                 call로 호출된 함수 종료, call 다음 명령줄로 이동
```

어셈블리 분석 (for)

```
0x178   <+0>    endbr64
0x17c   <+4>    push   %rbp
0x17d   <+5>    mov    %rsp, %rbp
0x180   <+8>    mov    $0x0, %eax                    eax 레지스터 초기화
0x185   <+13>   callq  0x149<for_test>               0x149 함수 호출
0x18a   <+18>   mov    $0x0, %eax                    eax 레지스터 초기화
0x18f   <+23>   pop    %rbp                          기존 rbp 위치로 돌아온다
0x190   <+24>   retq                                 다음 실행할 명령어의 주소를 불러온다


0x149   <+0>    endbr64
0x14d   <+4>    push   %rbp                          메모리를 0x10 16 Byte 지역변수 공간 할당
0x14e   <+5>    mov    %rsp, %rbp
0x151   <+8>    sub    $0x10, %rsp
0x155   <+12>   movl   $0x0, -0x4(%rbp)              값 0 을 rbp 에서 0x4만큼 떨어진 주소에 저장 DWORD 4 Byte 만큼 저장
0x15c   <+19>   jmp    0x16e<for_test+37>            <+37> 으로 이동
0x15e   <+21>   lea    0xe9f(%rip), %rdi    #0x6004  rdi = 0xe9f , 주소 값 저장
0x165   <+28>   callq  0x050<puts@plt>               printf 출력
0x16a   <+33>   addl   $0x1, -0x4(%rbp)              1(0x4) 저장 = 0(0x4) + 1($0x1) , i++
0x16e   <+37>   cmpl   $0x2, -0x4(%rbp)              0x4 주소의 저장된 값과 0x2(2)를 비교
0x172   <+41>   jle    0x15e<for_test+21>            <+37> 참이면 for_test <+21> 로 점프
0x174   <+43>   nop                                  아무 것도 하지 않음
0x175   <+44>   nop                                  아무 것도 하지 않음
0x176   <+45>   leaveq                               할당했던 공간을 회수 기존의 rbp 위치로 돌아온다
0x177   <+46>   retq                                 call로 호출된 함수 종료, call 다음 명령줄로 이동
```

어셈블리 분석 (while)

```
0x182   <+0>    endbr64
0x186   <+4>    push   %rbp
0x187   <+5>    mov    %rsp, %rbp
0x18a   <+8>    mov    $0x0, %eax
0x18f   <+13>   callq  0x149<while_test>          0x149 함수 호출
0x194   <+18>   mov    $0x0, %eax
0x199   <+23>   pop    %rbp                       할당했던 공간을 회수 기존의 rbp 위치로 돌아온다
0x19a   <+24>   retq


0x149   <+0>    endbr64
0x14d   <+4>    push   %rbp                       메모리를 0x10 16 Byte 지역변수 공간 할당
0x14e   <+5>    mov    %rsp, %rbp
0x151   <+8>    sub    $0x10, %rsp
0x155   <+12>   movl   $0x3, -0x4(%rbp)           rbp 기준 -4 Byte 지점에 num 선언 후 3 저장
0x15c   <+19>   jmp    0x178<while_test+47>       <while_test+47> 로 점프
0x15e   <+21>   mov    -0x4(%rbp), %eax           eax = 3
0x161   <+24>   mov    %eax, %esi                 esi = eax
0x163   <+26>   lea    0xe9a(%rip), %rdi    #0x6004
0x16a   <+33>   mov    $0x0, %eax
0x16f   <+38>   callq  0x050<printf@plt>
0x174   <+43>   subl   $0x1, -0x4(%rbp)           2(0x4) = 3(0x4) - 1(0x0)
0x178   <+47>   cmpl   $0x0, -0x4(%rbp)           0x4(3) != 0x0(0)
0x17c   <+51>   jne    0x15e<while_test+21>       비교 결과가 다를 때 <+21>로 점프
0x17e   <+53>   nop
0x17f   <+54>   nop
0x180   <+55>   leaveq                            할당했던 공간을 회수 기존의 rbp 위치로 돌아온다
0x181   <+56>   retq                              call로 호출된 함수 종료, call 다음 명령줄로 이동
```

어셈블리 분석 (메모리)

main <+0>
(gdb) p/x $rbp
$1 = 0x0
(gdb) p/x $rsp
$2 = 0x7fffffffde68

<+4> push %rbp

0x68	rbp	rbp를 rsp에 밀어 넣는다
0x60	rsp	0x68 rbp

<+5> mov $rsp, %rbp

0x68		rbp 는 이전 주소보다 8 Byte 확장된 공간을 지시
0x60	rsp =rbp	0x68

<+8> sub $0x10, %rsp

0x68		rsp 주소 값 - 0x10 = ?
0x60	rbp	? 를 rsp에 저장해서 16 Byte 공간 확보
0x58		
0x50	rsp	

<+12> sub $0x3, -0x8(%rbp)

0x68		rbp 기준 - 8 Byte 지점에 3 저장
0x60	rbp	
0x58	3 (num)	
0x50	rsp	

<+19> mov -0x8(%rbp), %eax

eax	3 (num)	eax 에 3 값 저장

<+22> mov %eax, %edi

eax	3 (num)	edi (임시 저장 레지스터) = eax
edi	3 (num)	res = my func(num); 의 num을 int my_func(int num)으로 전달

어셈블리 분석 (메모리)

<+24> callq 0x149 <my_func> 진입

0x68	
0x60	rbp (main)
0x58	3 (num)
0x50	
0x48	<main + 29> 복귀 주소

복귀 주소를 스택에 저장
my_func 작업 종료되면 call 다음 주소 값으로 이동

my_func <+0>

(gdb) x $rbp
0x7fffffffde60: 0x00000000
(gdb) x $rsp
0x7fffffffde48: 0x55555178

call 이휴 8 Byte 할당 받음

<+4> push %rbp

0x68	
0x60	rbp (main)
0x58	3
0x50	
0x48	<main + 29> 복귀 주소, rbp

rbp를 rsp에 밀어 넣는다

어셈블리 분석 (메모리)

0x40	rsp

<+5> mov %rsp, %rbp

0x68	
0x60	rbp (main)
0x58	3
0x50	
0x48	<main + 29> 복귀 주소
0x40	rsp = rbp

rbp 는 이전 주소보다 8 Byte 확장된 공간을 지시

<+8> mov %edi, -0x4(%rbp)

0x68	
0x60	rbp (main)
0x58	3 (num)
0x50	
0x48	<main + 29> 복귀 주소
0x40	rsp = rbp
0x3c	3 (num)

0x3c 주소 값에 3 값 저장

<+14> sar %eax

3 (eax) 값을 >> 수행하여 다시 eax에 저장 (함수 리턴값 사용)

pop %rbp

0x68	
0x60	rbp
0x58	3 (num)
0x50	
0x48	<main + 29> 복귀 주소

값을 추출하고 할당했던 공간을 회수, 기존의 rbp 위치로 복귀

<+17> retq

0x68	
0x60	rbp
0x58	3 (num)
0x50	rsp

<main +29> 로 돌아온다

어셈블리 분석 (메모리)

<+29> mov %eax, -0x4(%rbp)

0x68	
0x60	rbp
0x5c	1
0x58	3 (num)
0x50	rsp

rbp 기준 - 4 Byte 지점에 1 저장

<+35> mov %eax, %esi — printf 출력하기 위해 %esi에 1 값 저장

<+37> lea 0xe7d(%rip), %rdi — 스트림 (프로그램과 입력장치 또는 출력장치 사이의 다리 역할을 하는 매개체)

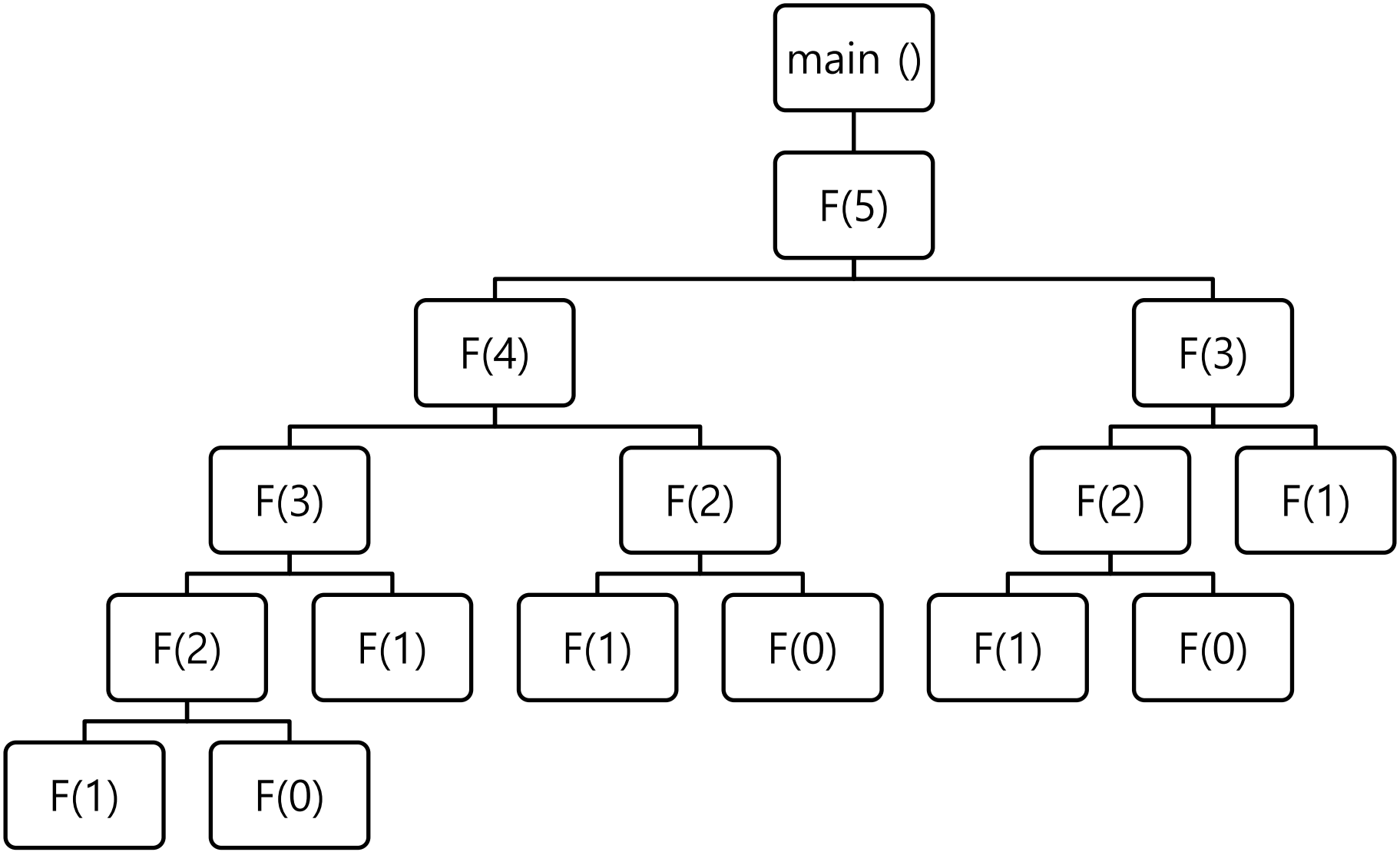
<+54> mov $0x0 %eax — eax 레지스터 초기화

<+59> leaveq — 할당했던 공간 회수 , 기존 rbp 위치로 돌아온다

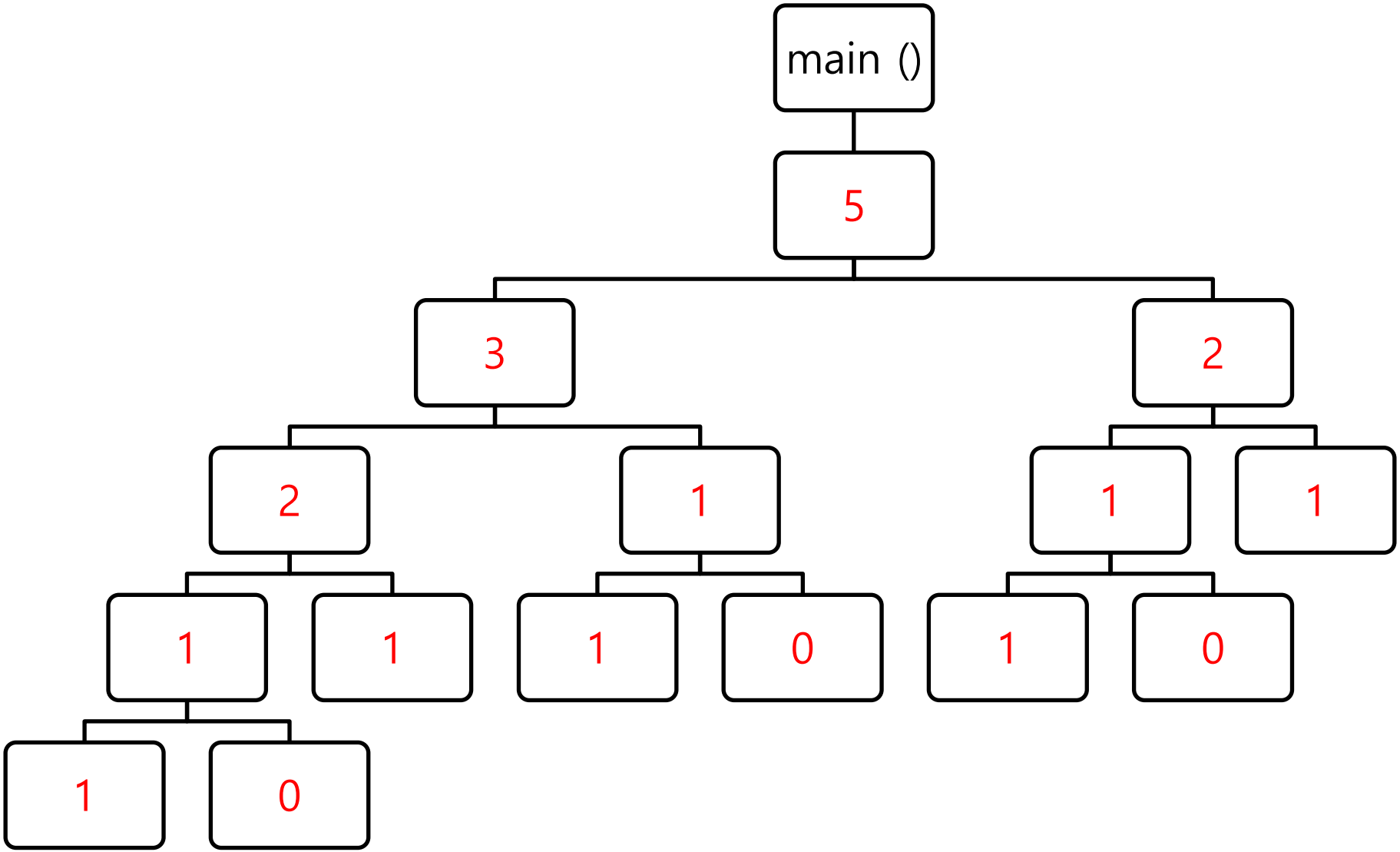
main <+0>
(gdb) p/x $rbp
$1 = 0x0
(gdb) p/x $rsp

$2 = 0x7fffffffde68

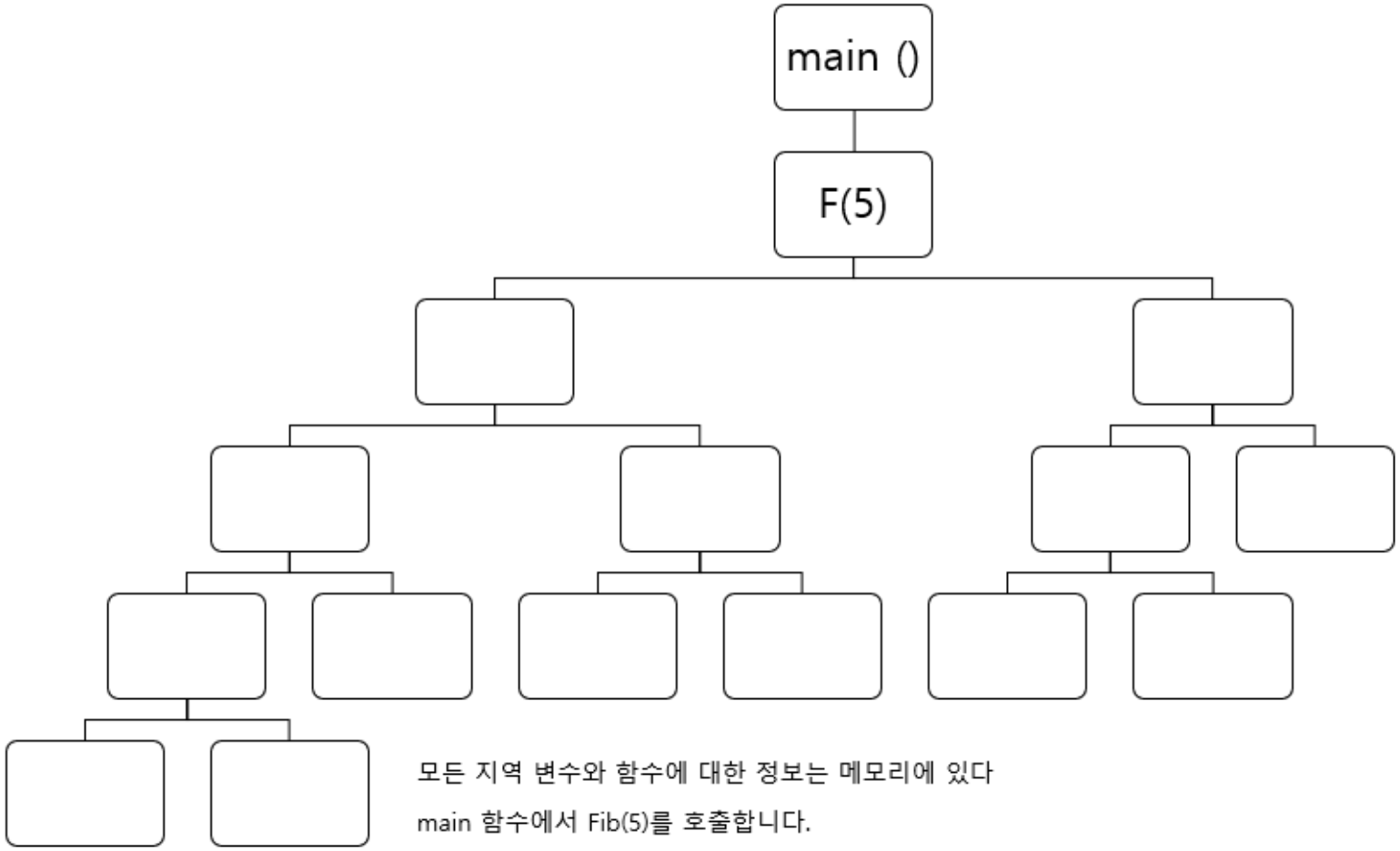
재귀 함수 분석



재귀 함수 분석



재귀 함수 분석



모든 지역 변수와 함수에 대한 정보는 메모리에 있다

main 함수에서 Fib(5)를 호출합니다.

F(5) 가 메모리에 저장되고 실행 중... main () 은 일시 중지된 상태로 저장되어 있다

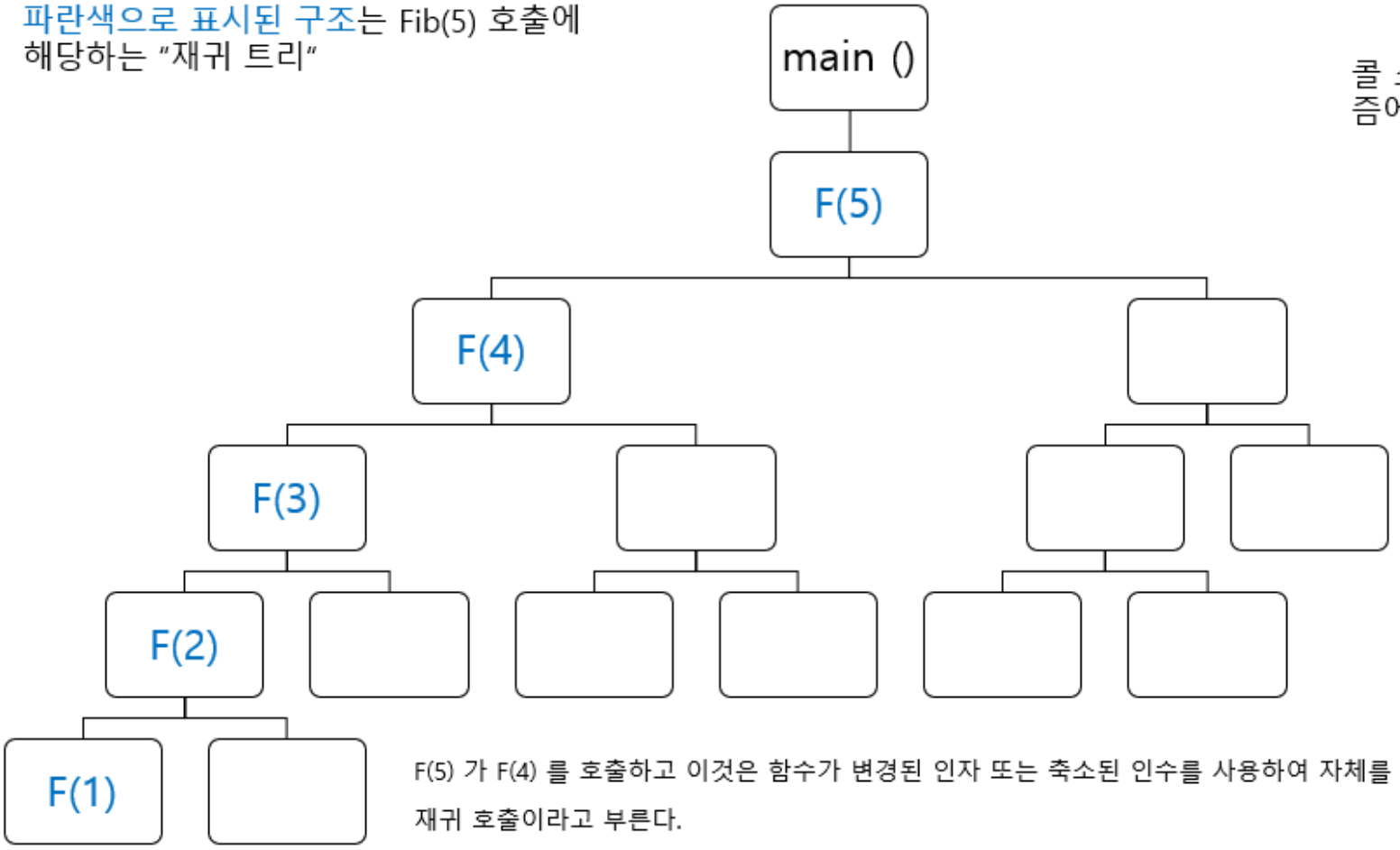
→ 화살표는 나머지 기능이 일시 중지된 상태에서 현재 실행 중인 기능을 지시.

Memory

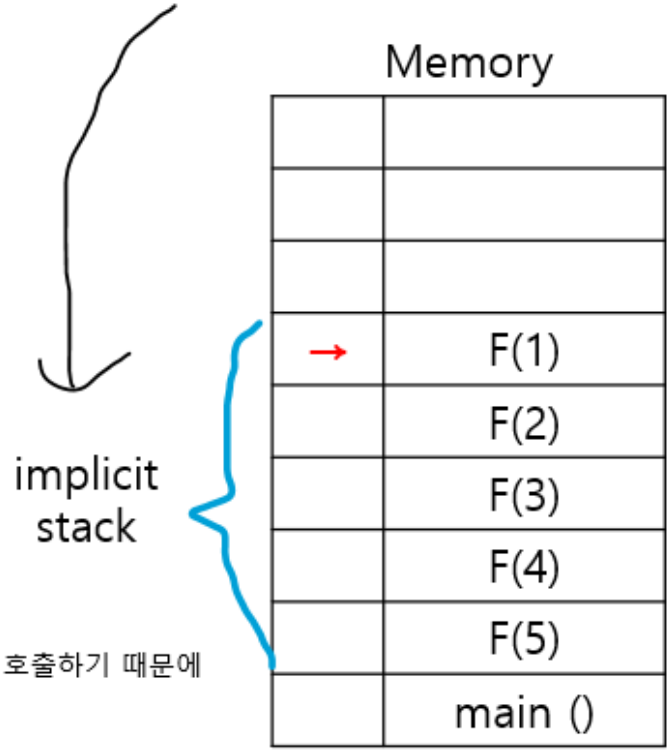
→	F(5)
	main ()

재귀 함수 분석

파란색으로 표시된 구조는 Fib(5) 호출에 해당하는 "재귀 트리"



콜 스택의 이 부분은 우리의 재귀 알고리즘에 의한 메모리 소비의 종류이다

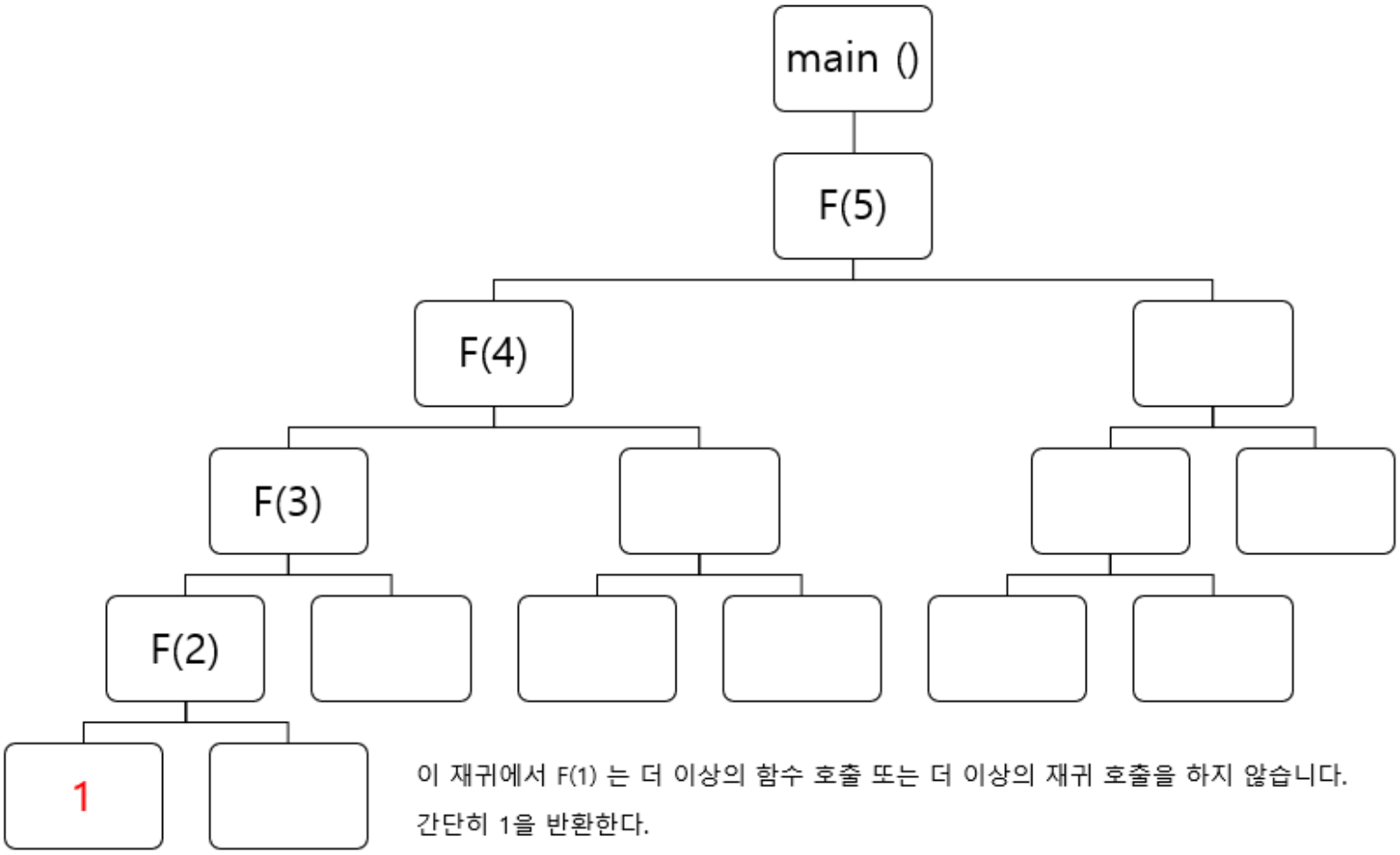


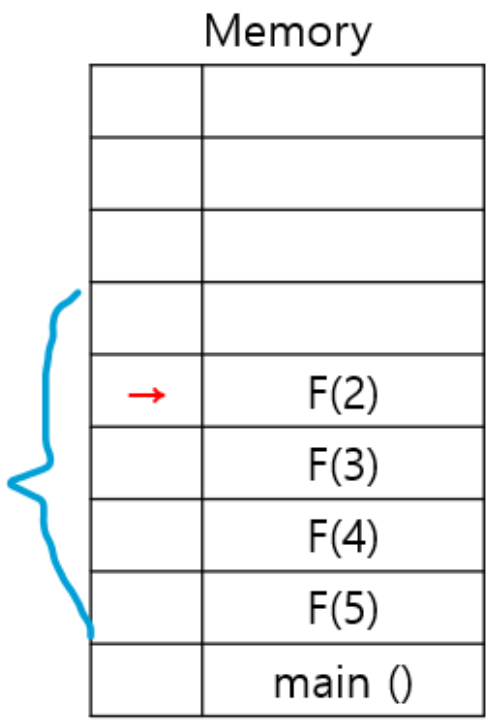
F(5) 가 F(4) 를 호출하고 이것은 함수가 변경된 인자 또는 축소된 인수를 사용하여 자체를 호출하기 때문에 재귀 호출이라고 부른다.

실행 측면에서 별도의 메모리와 리소스가 새 함수 호출에 할당된다

F(4) 실행 중이고 F(3), F(2), F(1) 순서대로 호출 , 현재 F(1) 실행 중...

재귀 함수 분석



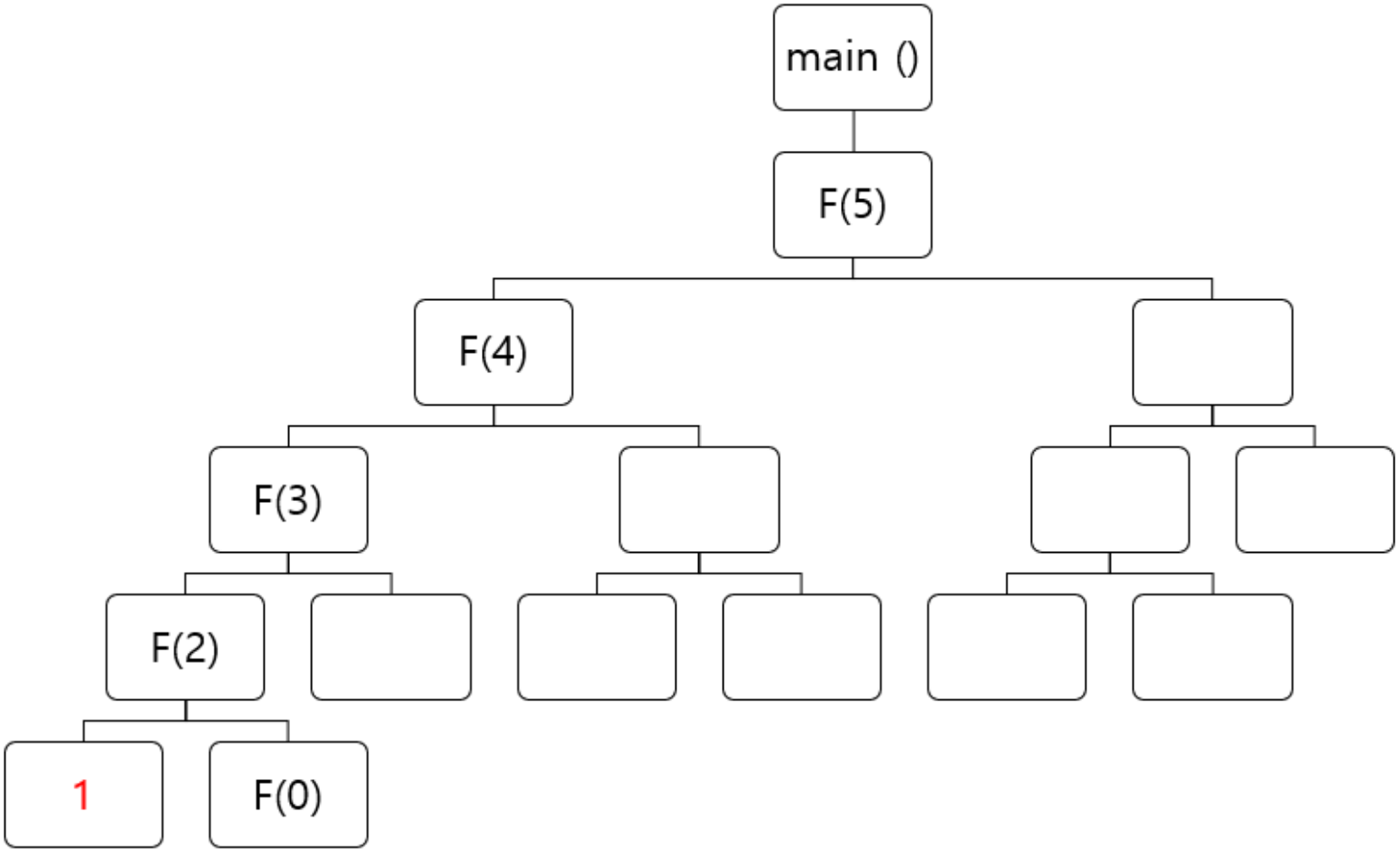


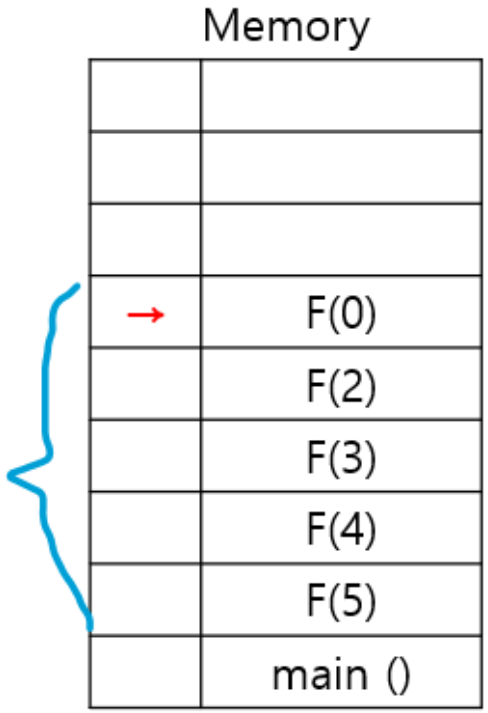
이 재귀에서 F(1) 는 더 이상의 함수 호출 또는 더 이상의 재귀 호출을 하지 않습니다.

간단히 1을 반환한다.

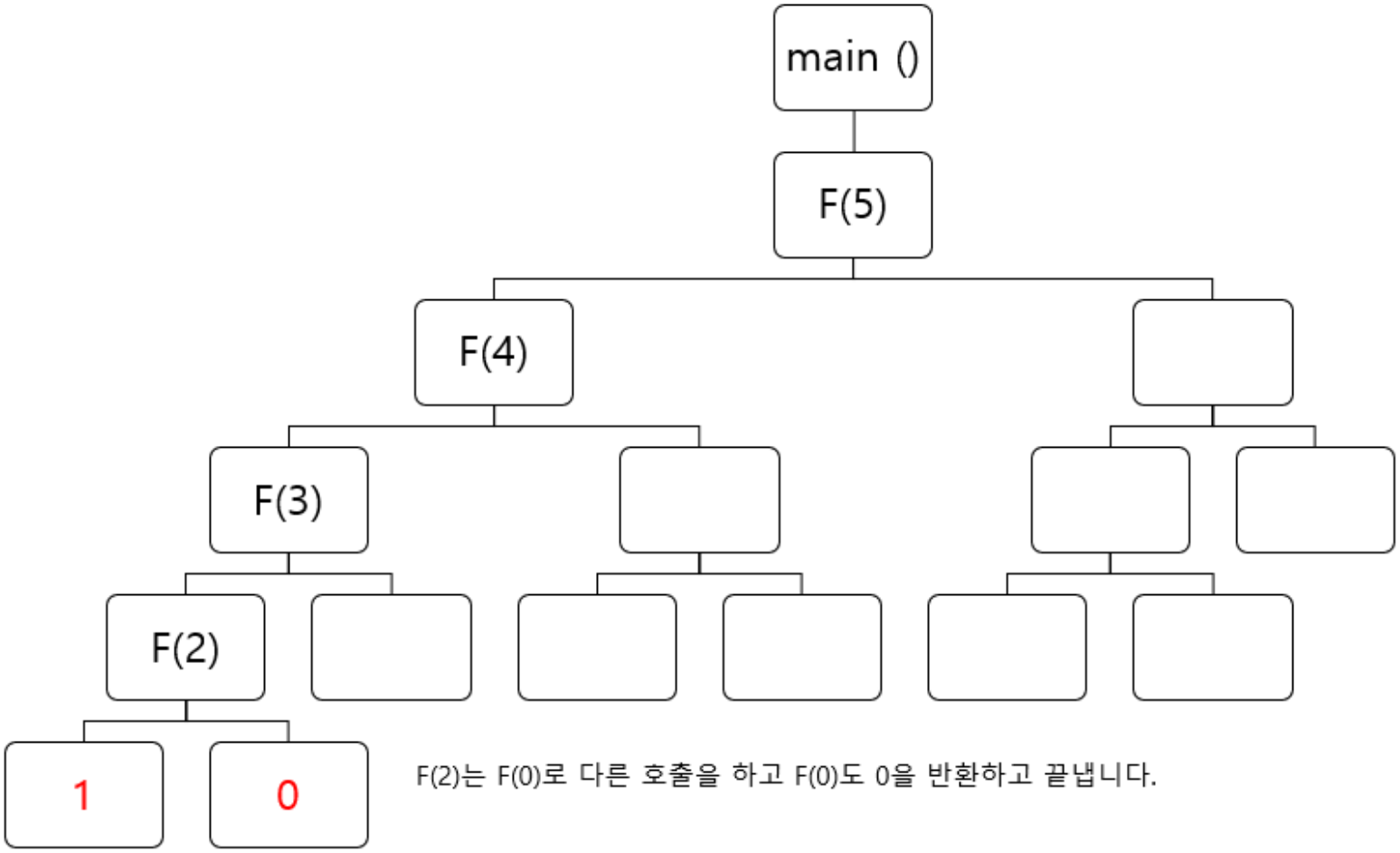
F(1) 이 완료되고 메모리에서도 pop , F(2) 로 돌아간다

재귀 함수 분석

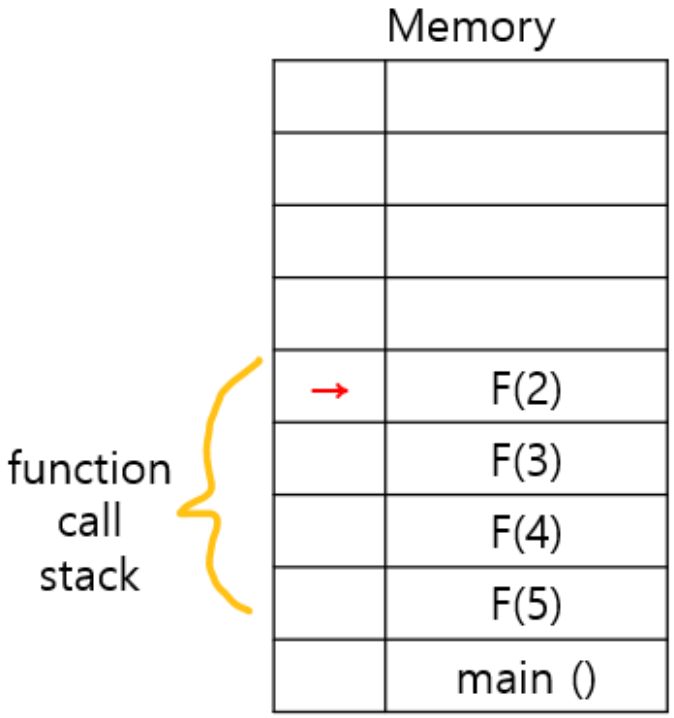




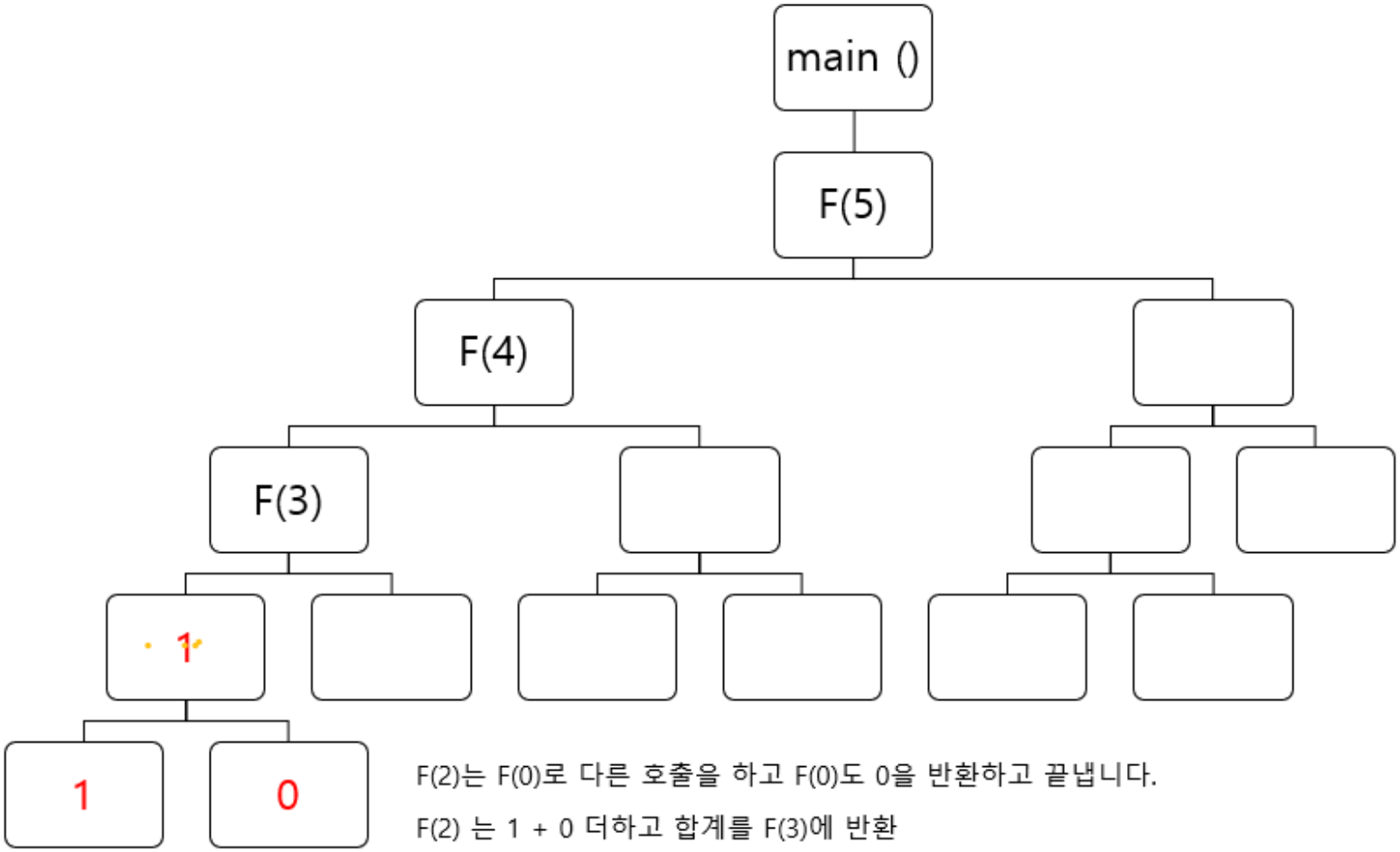
재귀 함수 분석

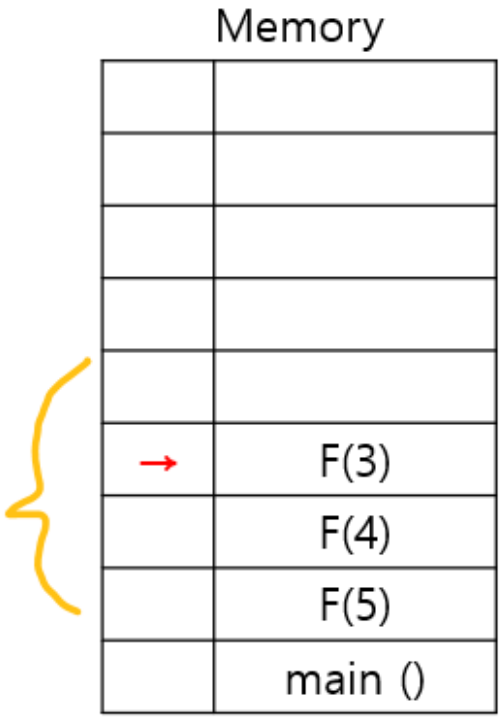


F(2)는 F(0)로 다른 호출을 하고 F(0)도 0을 반환하고 끝냅니다.



재귀 함수 분석





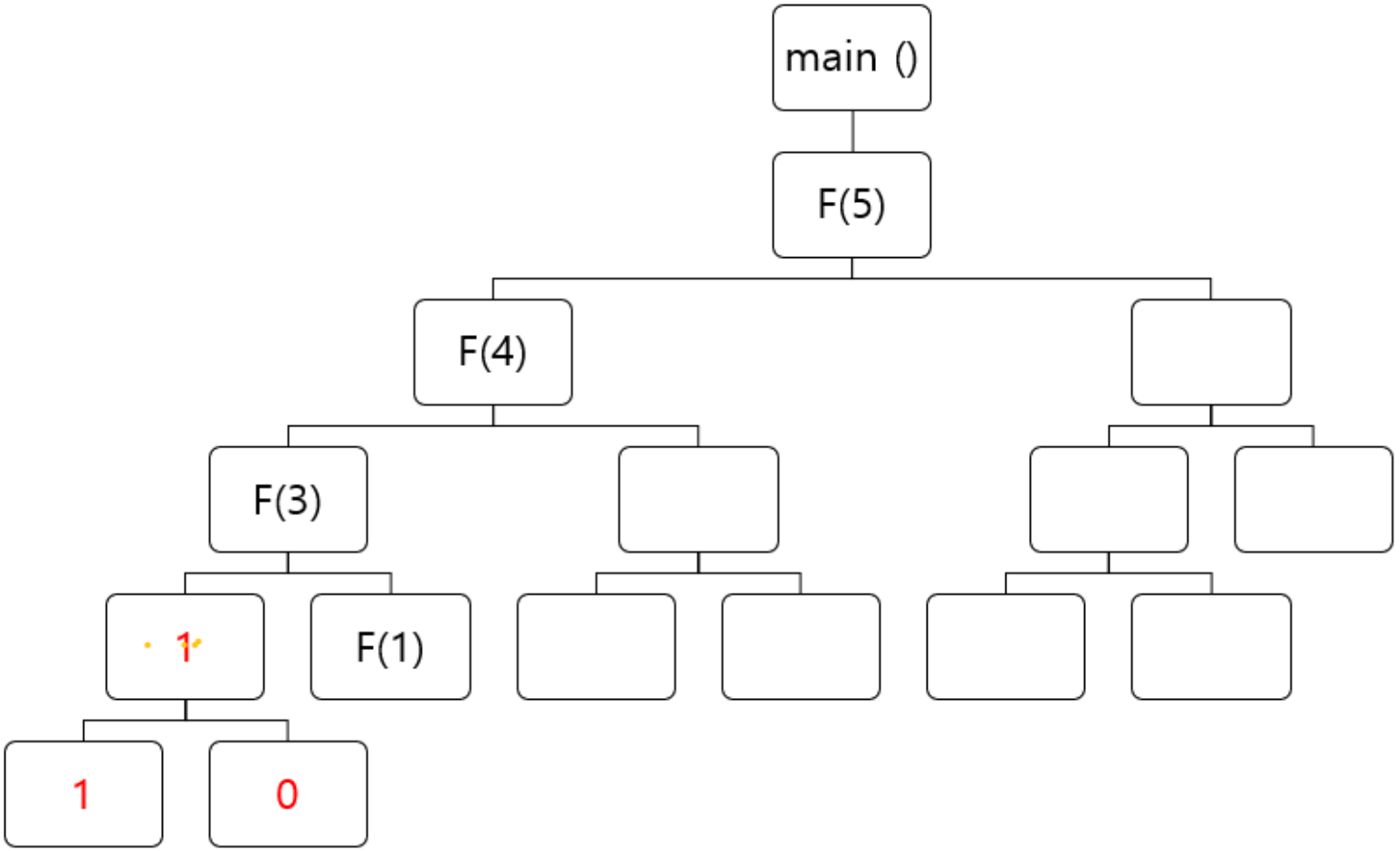
F(2)는 F(0)로 다른 호출을 하고 F(0)도 0을 반환하고 끝냅니다.

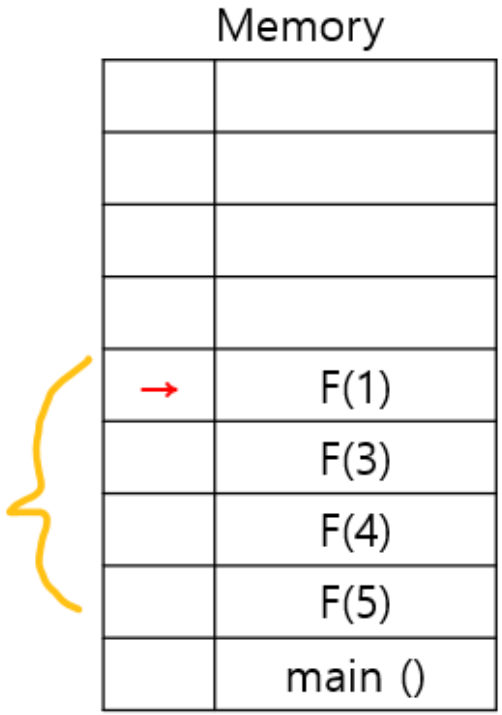
F(2) 는 1 + 0 더하고 합계를 F(3)에 반환

F(3)이 다시 재개되고 이제 F (3)이 F (2)를 얻었으므로 F (1)을 다시 호출하고

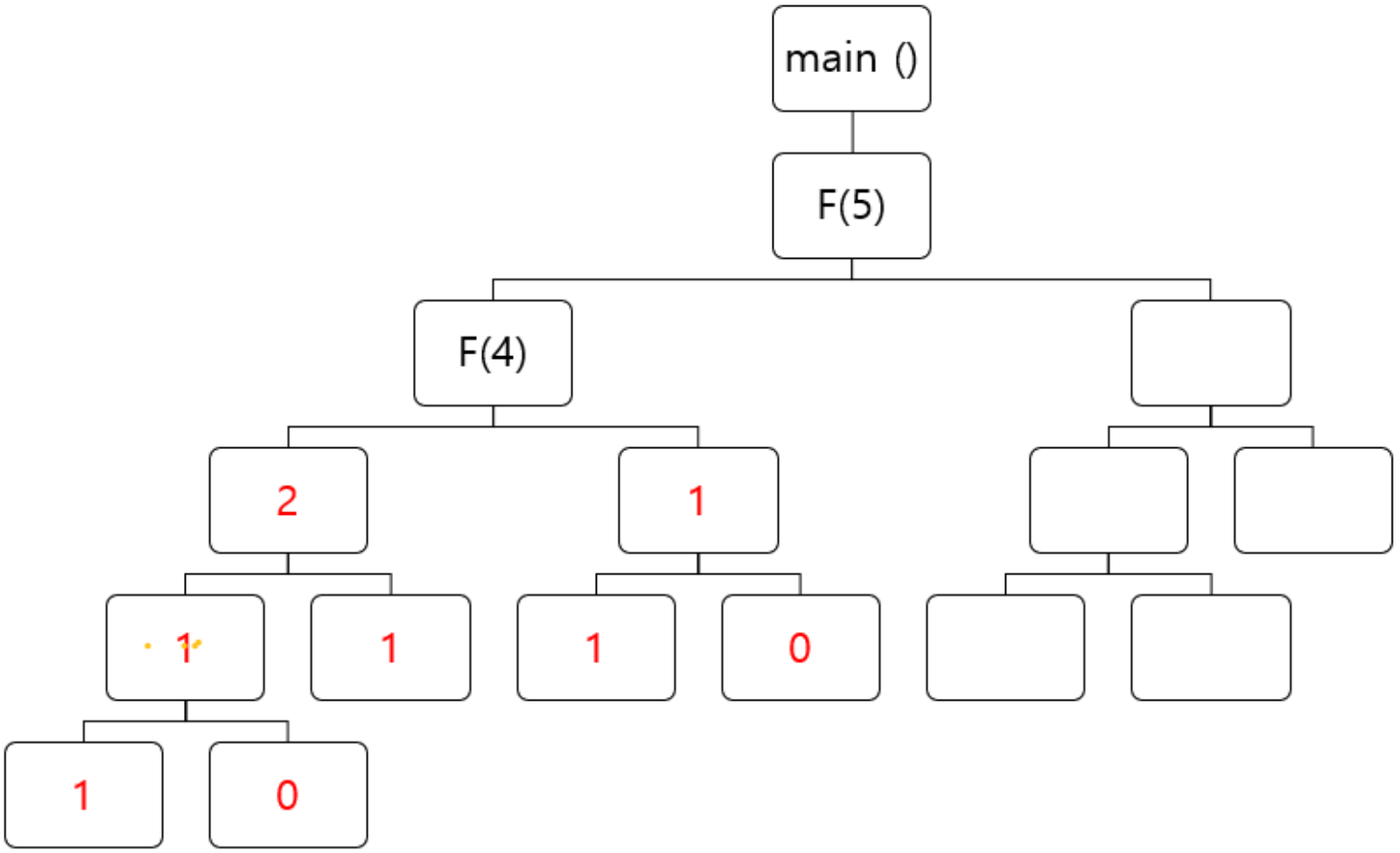
메모리에 밀어 넣고 pop 이 계속된다

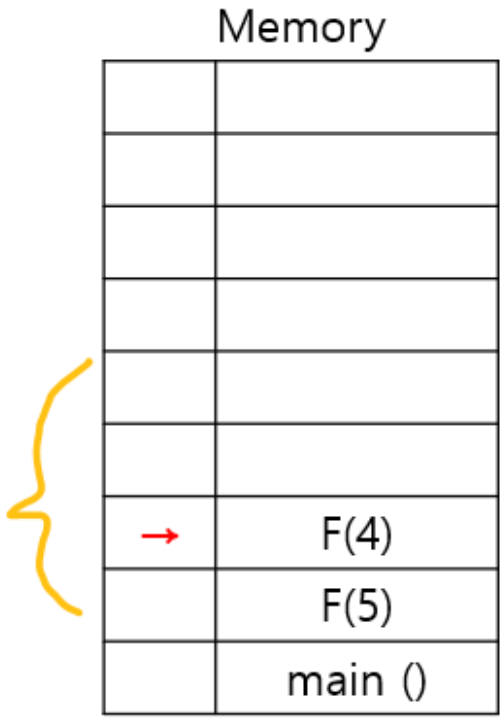
재귀 함수 분석



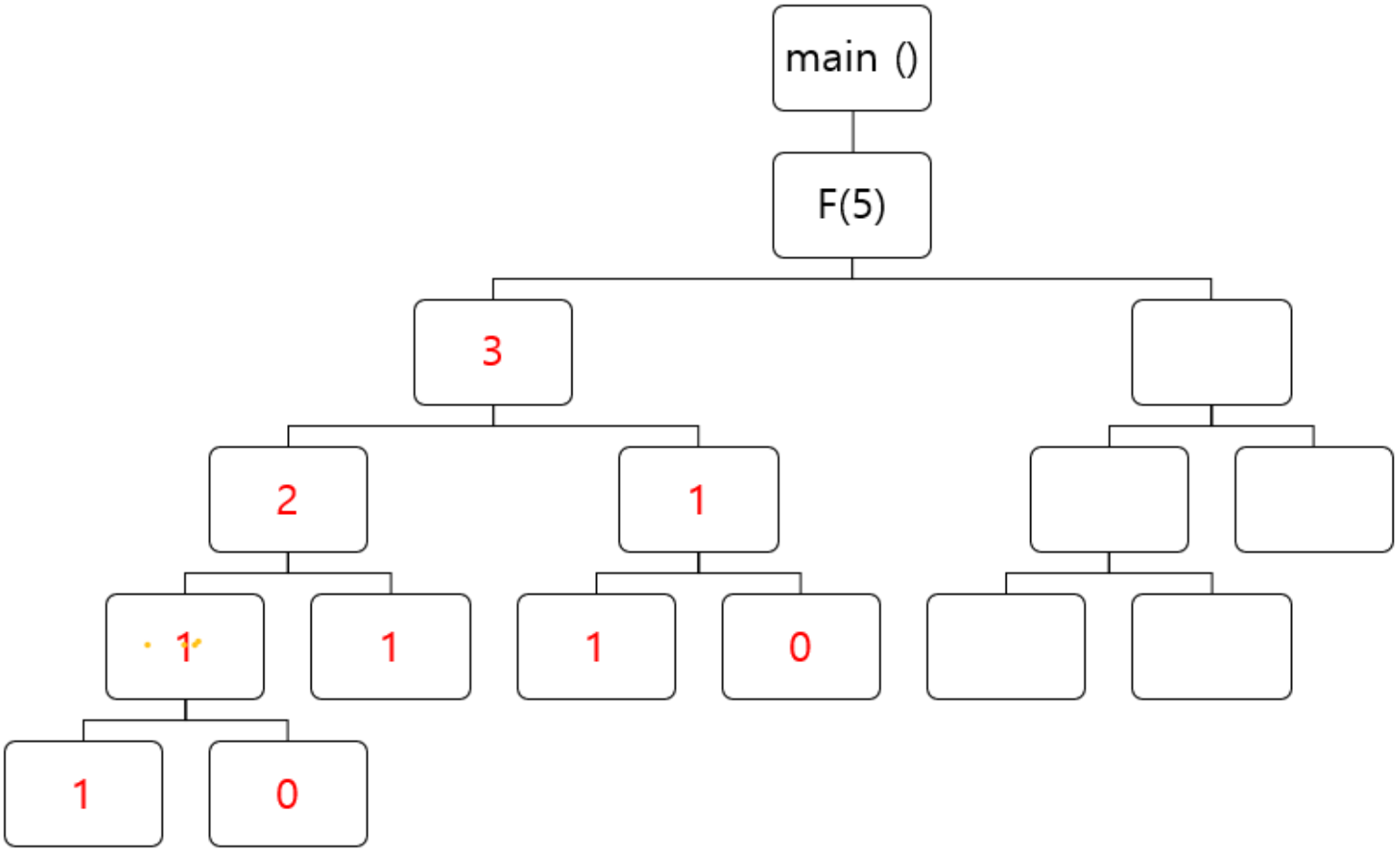


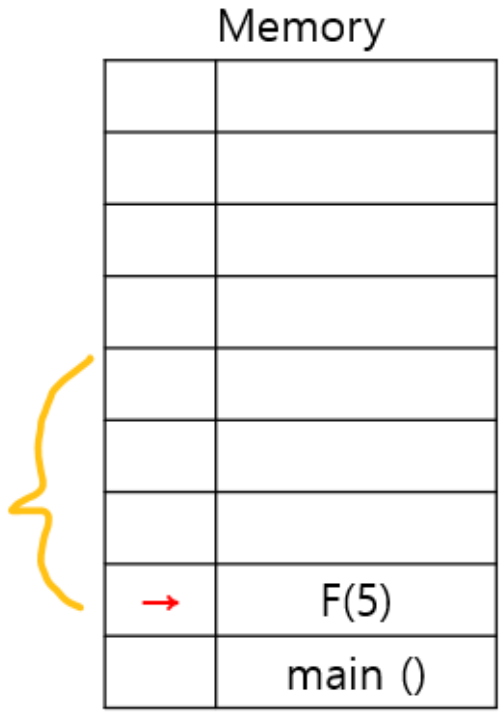
재귀 함수 분석



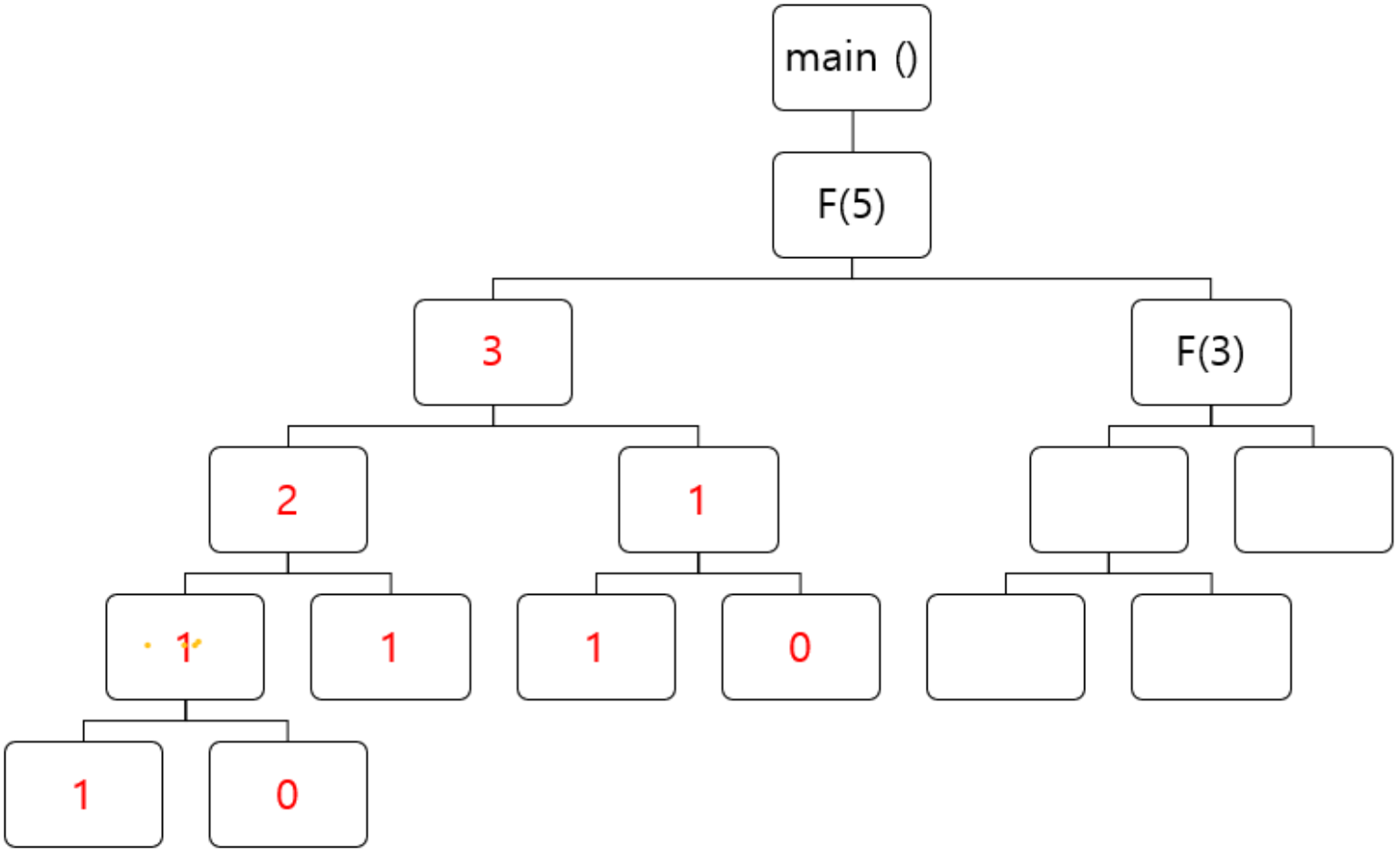


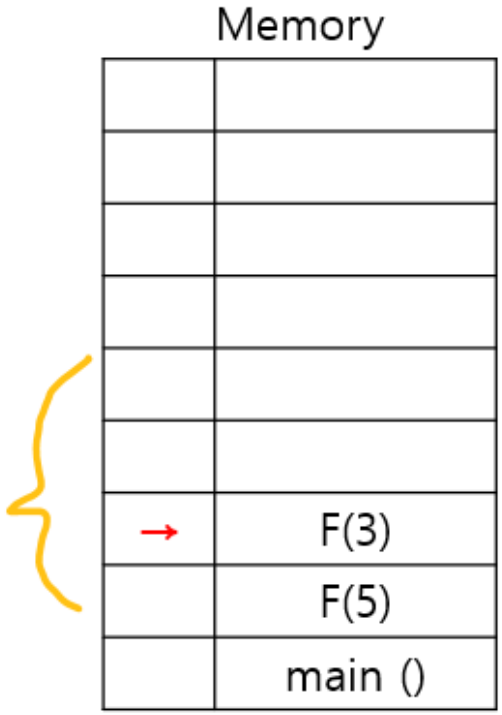
재귀 함수 분석



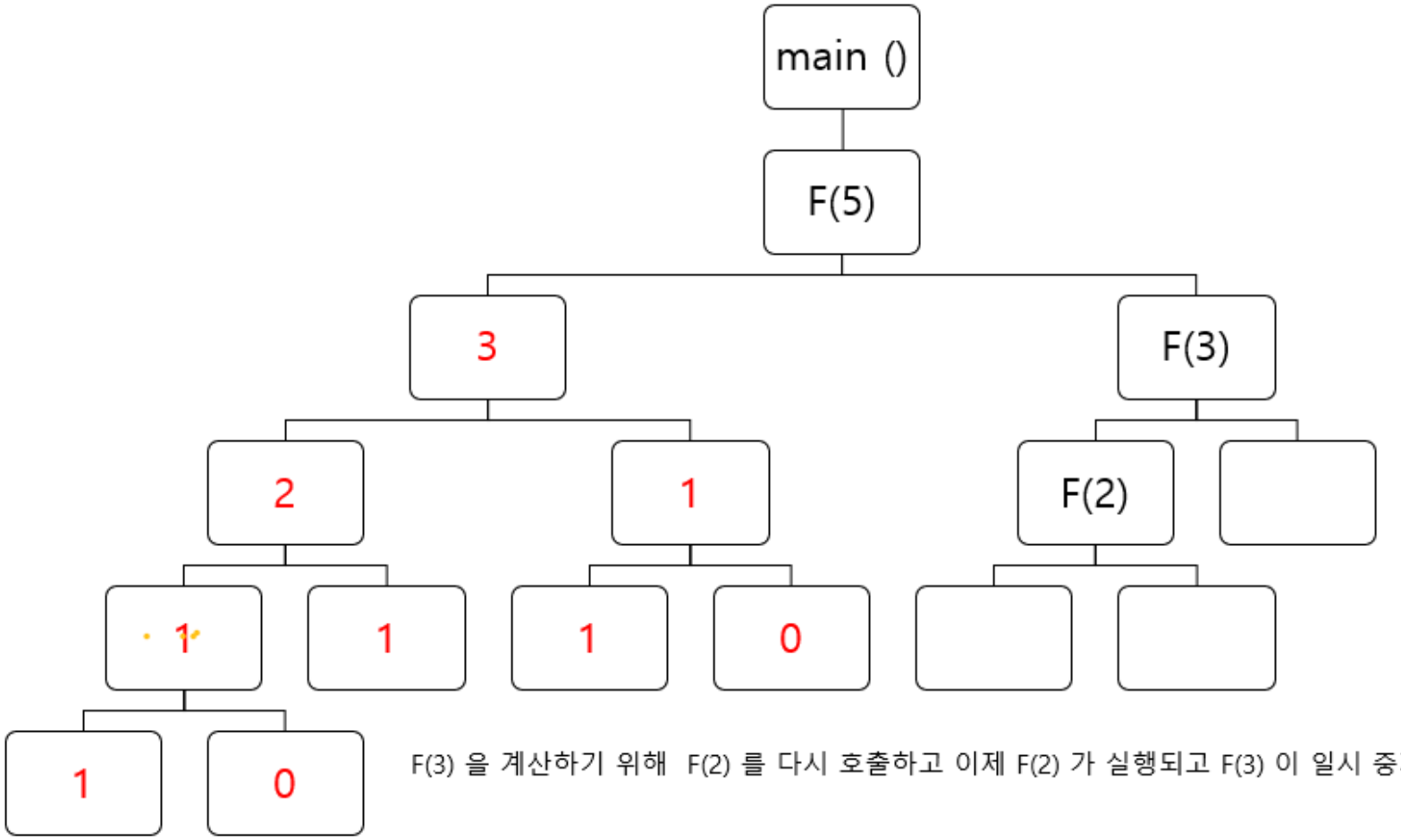


재귀 함수 분석

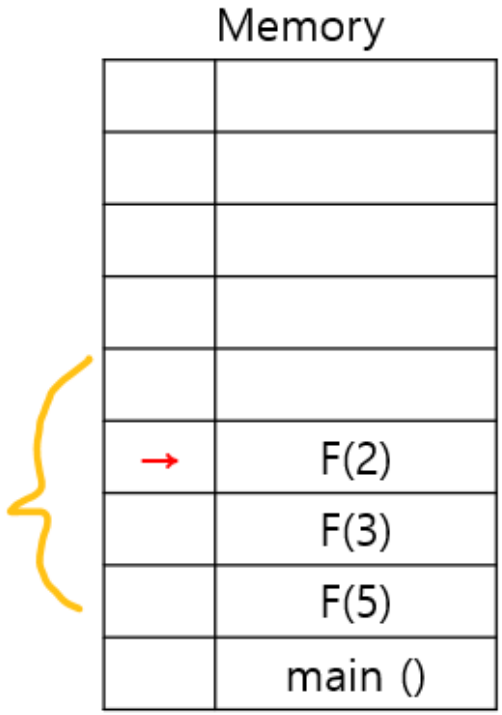




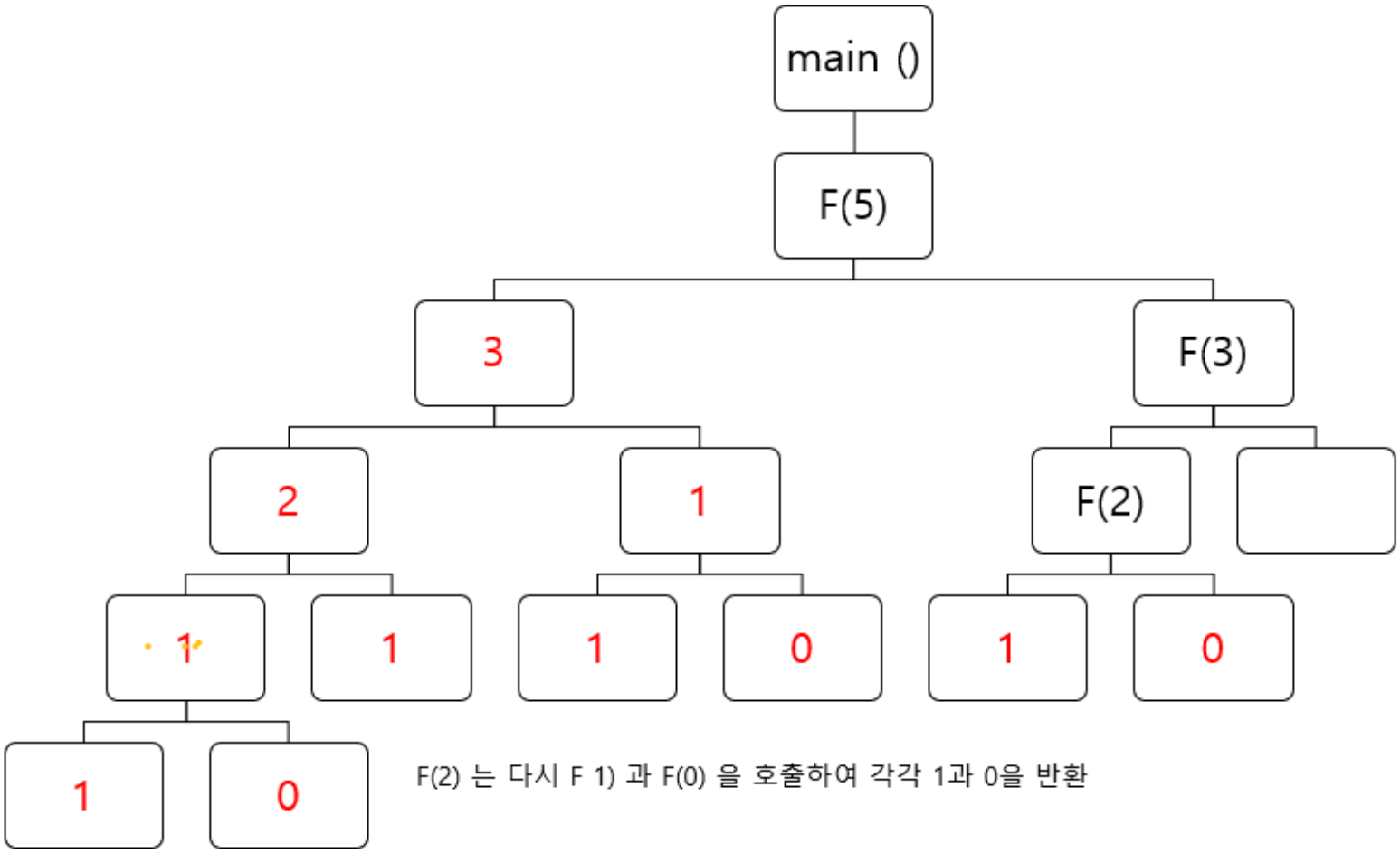
재귀 함수 분석



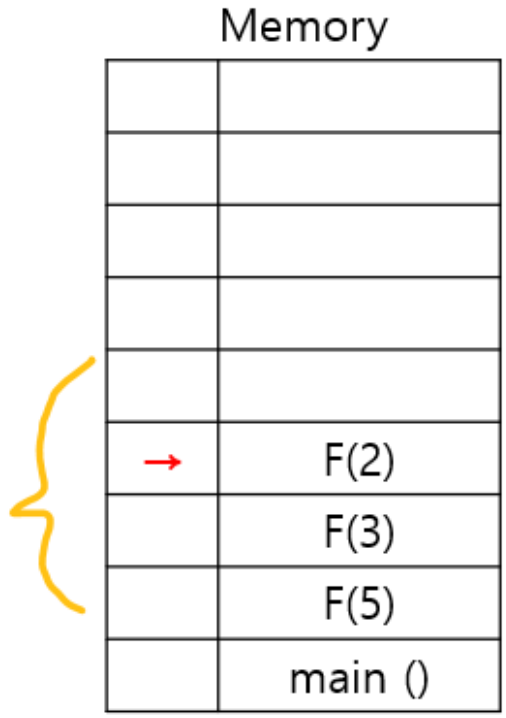
F(3) 을 계산하기 위해 F(2) 를 다시 호출하고 이제 F(2) 가 실행되고 F(3) 이 일시 중지된 상태



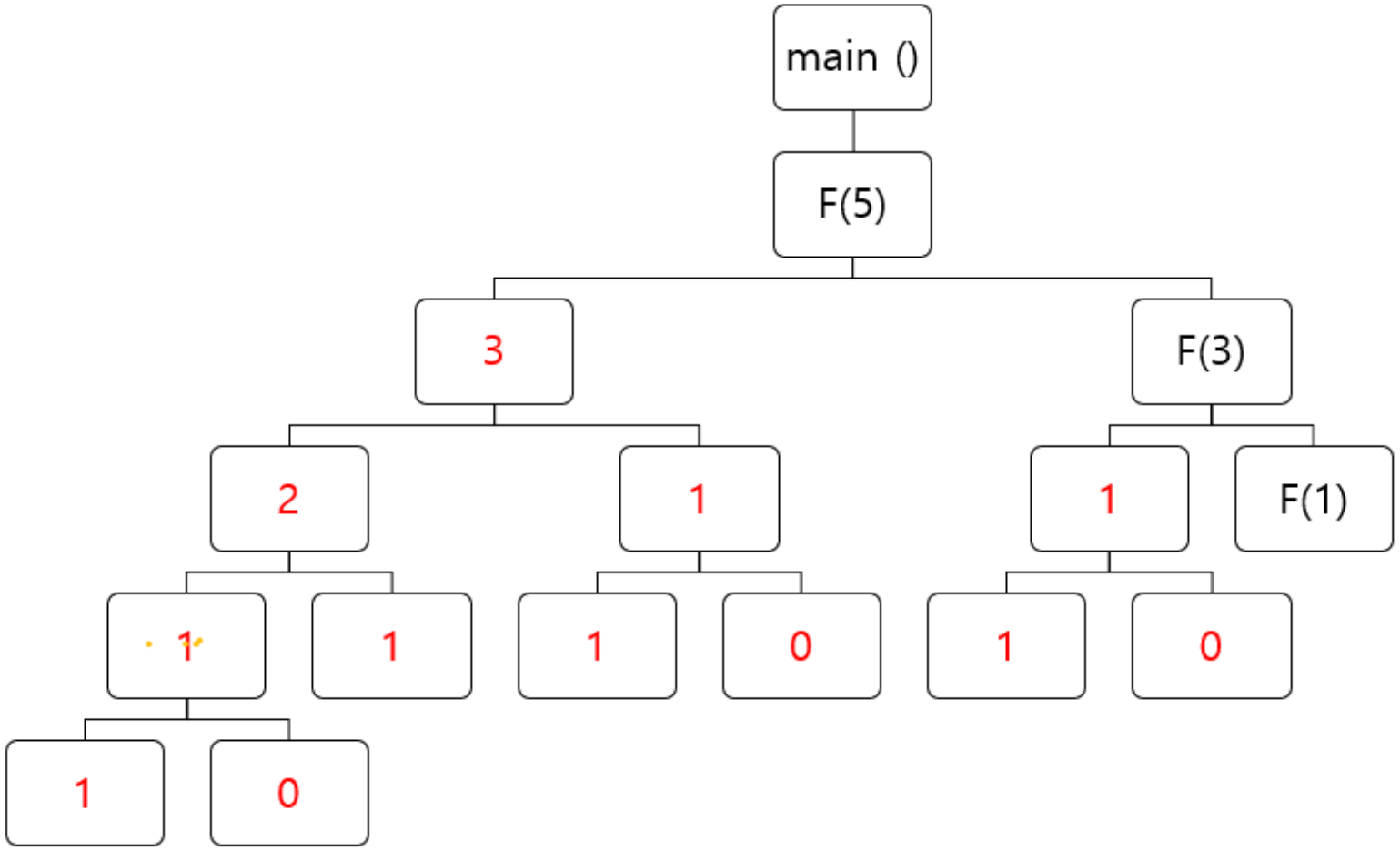
재귀 함수 분석

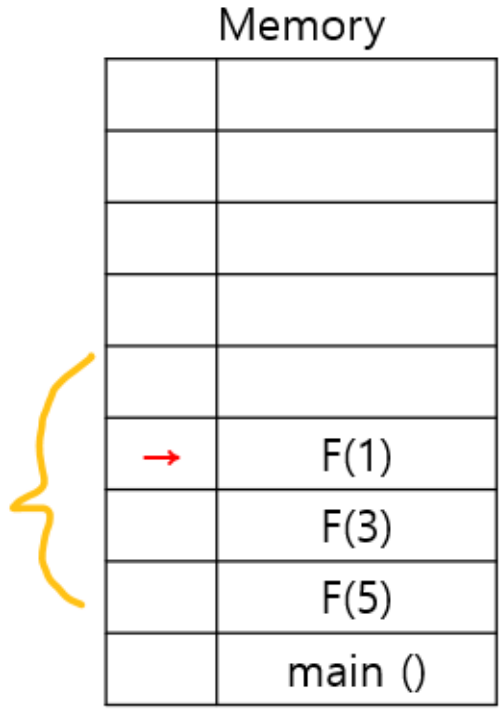


F(2) 는 다시 F 1) 과 F(0) 을 호출하여 각각 1과 0을 반환

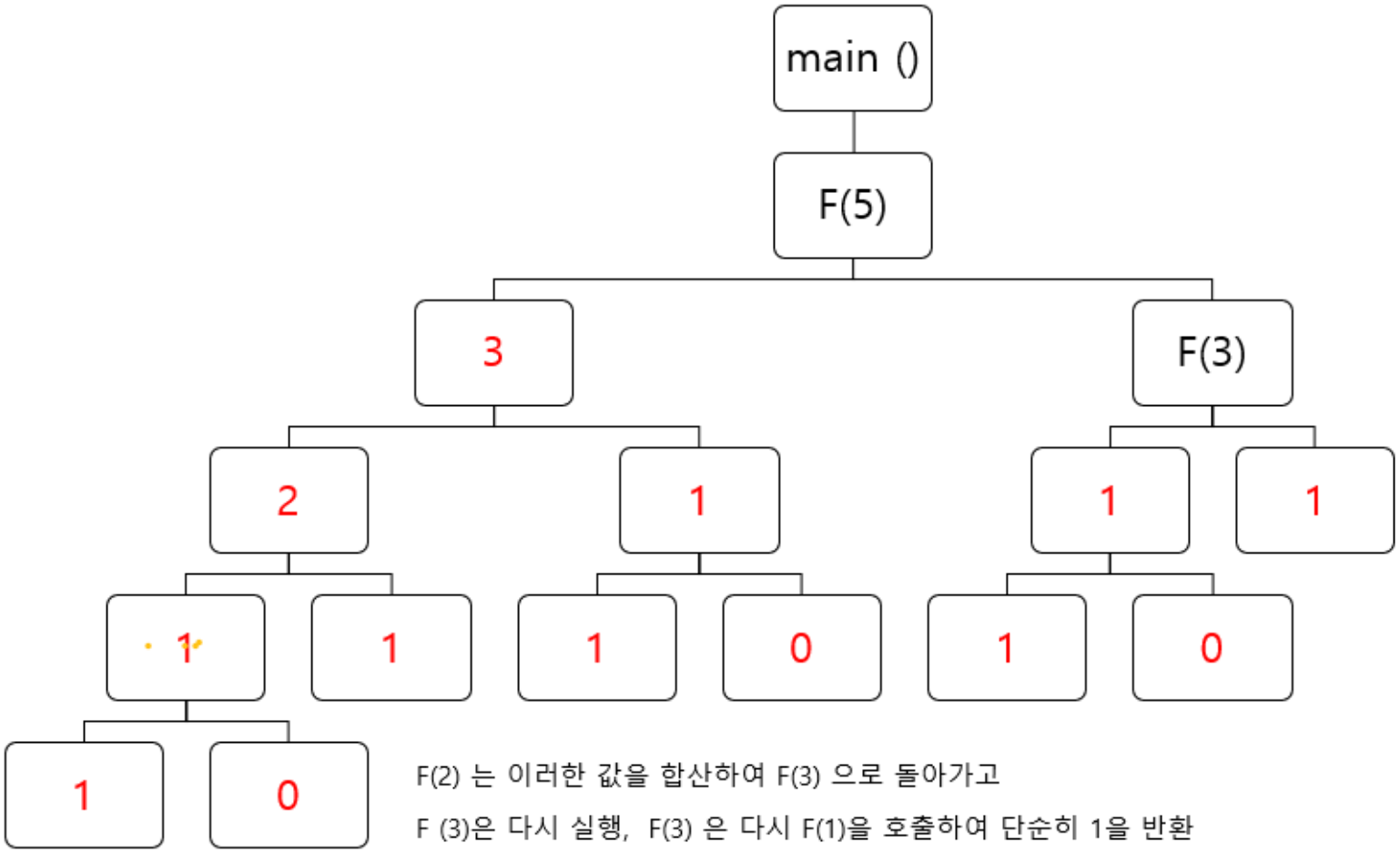


재귀 함수 분석



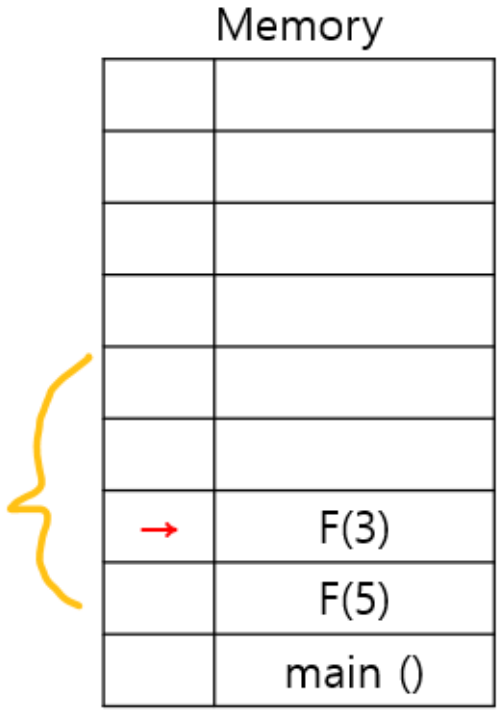


재귀 함수 분석

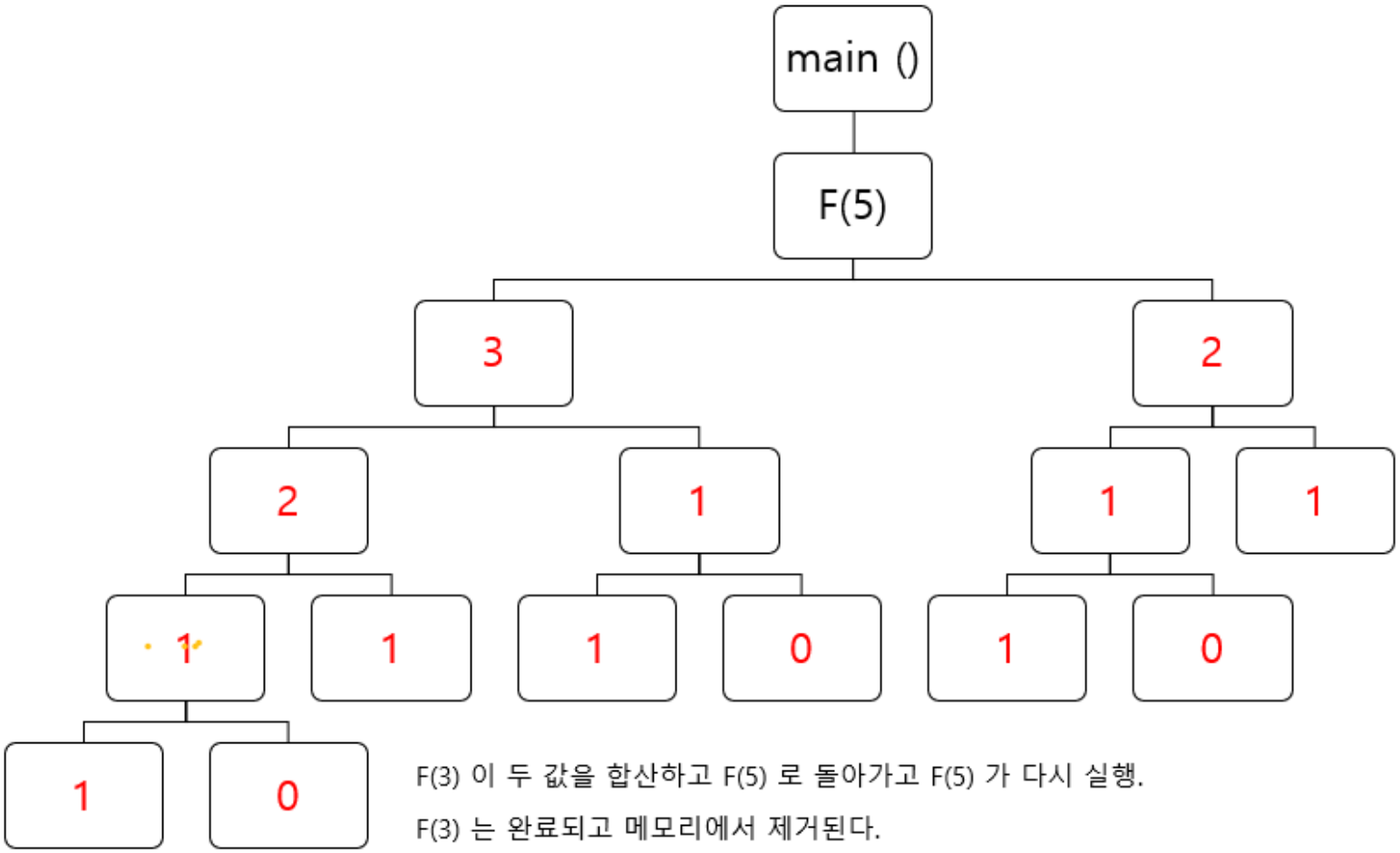


F(2) 는 이러한 값을 합산하여 F(3) 으로 돌아가고

F (3)은 다시 실행, F(3) 은 다시 F(1)을 호출하여 단순히 1을 반환



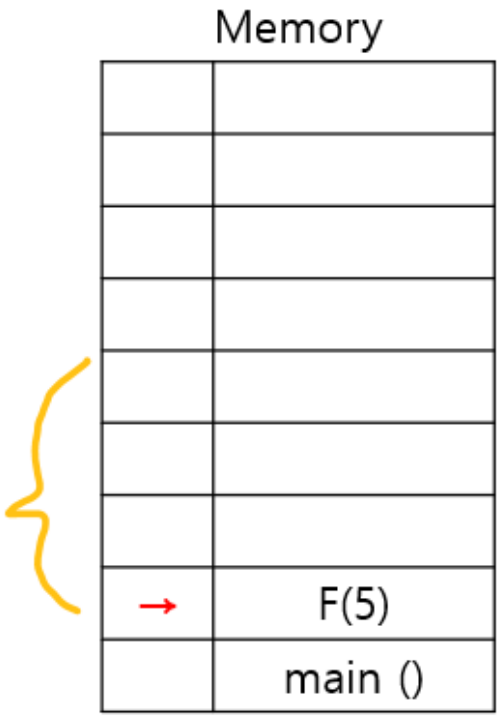
재귀 함수 분석



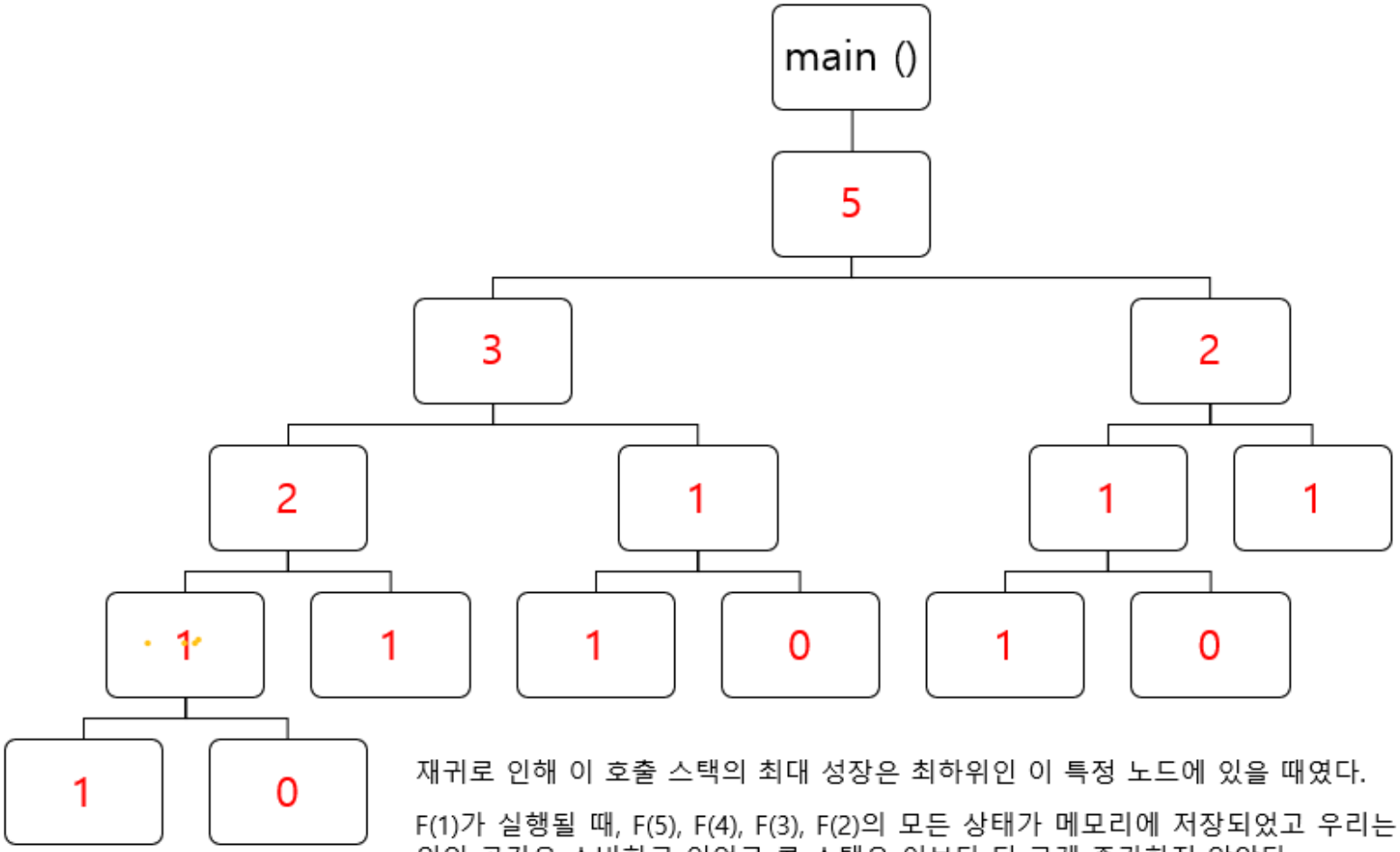
F(3) 이 두 값을 합산하고 F(5) 로 돌아가고 F(5) 가 다시 실행.

F(3) 는 완료되고 메모리에서 제거된다.

F(5) 는 두 값을 합산하고 호출자인 main 함수로 돌아간다.



재귀 함수 분석



재귀로 인해 이 호출 스택의 최대 성장은 최하위인 이 특정 노드에 있을 때였다.

F(1)가 실행될 때, F(5), F(4), F(3), F(2)의 모든 상태가 메모리에 저장되었고 우리는 메모리에서 5 단위의 공간을 소비하고 있었고 콜 스택은 이보다 더 크게 증가하지 않았다.

함수 호출의 수는 5개, 소비되는 최대 메모리는 5개