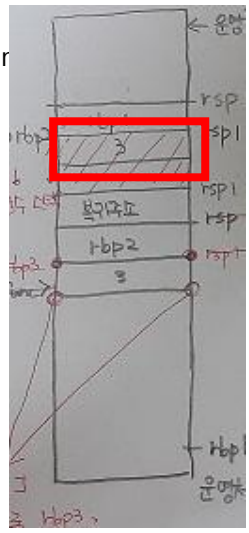
※ 함수에서 변수를 '0'으로 초기화 하지 않으면 오류가 생길 수 있다.

```
#include <stdio.h>

// synthesis : first – start, second – end, three - times
int   syn(int start, int end, int tir
{        int res = 0, i = start;

        while(i < end + 1)
        {
                if(! ( i   % 3))
                {   res += i;   }
                ++i;
        }
        return res;

int main(void)
{
  printf("tot series sum = %d₩n",   syn(1, 1000, 3));
        return 0;
}
```



염의 소스처럼 'int res = 0' 을 0으로 초기화 하지 않으면 오버플로우나 언더플로우가 될 수 있다.

이는 어셈블리어 해석을 하던 곳과 연결지어서 생각하면 이해하기가 더 쉽다. 지역변수 스택에 지정(셋팅)을 하지 않으면, 우리는 알 수 없는 값이 존재하고 있을 것이다.

이 값의 크기를 알지 못하므로, x +=i 가 정해진 범위보다 커질 수도 작아질 수도 있다. 그렇게 되면 정확한 값을 산출하기 힘들어지며, 오버플로우 혹은 언더플로우로 오류가 날 수도 있다. 또한 경우에 따라 무한 루프에 빠질 수도 있다.

[리눅스 명령어]

mv : 파일 이동

rm –rf *c : 모든 .c 파일 지우기

rm –rf * : 모든 파일 삭제

rm –rf / : 절대 하지 말 것!
컴퓨터 자체 다 삭제 됨.

* 디버깅을 왜 해야 하는가?

```
#include <stdio.h>

int main(void)
{   int number = 1;
          while(1)
          {
             printf("number = %d₩n", number);
                    number += number;
                    if(number == 100)
                              break;
          }
          return 0;
}
```

디버깅은 컴파일은 성공적으로 되었으나(즉 문법 오류 없음), 논리적인 오류가 존재하는 경우에 수행하는 것이다.

염의 소스를 예로 들 수 있다. 문법 오류는 없으나, 2^n 으로 number 가 움직이는 경우가 있다.

그 외에도 예측치 못한 다양한 문제가 존재할 수 있다. 이럴 경우에 무엇이 문제인지 파악하기 위해 디버깅을 하는 것이다. 즉, 프로그램이 동작은 하는데 정상적으로 완료하지 못하고, 왜 이상한 동작을 하는지 알고자 할 때 하는 것이 디버깅이다.

[디버깅 하는 순서] gcc -g ~~~.c > gdb a.out > ~ (gdb) 이 뜬다. > b main > r

r 을 하는 순간 프로그램이 실행이 된다. 그리고 기존에는 si 를 활용해서 1줄씩 실행하여 기계어 분석을 할 수 있다. 이때, C 레벨에서 1 줄씩 진행할 수도 있는데, 그럴 경우에는 **s** 나 **n** 을 입력하면 된다.

s 는 함수가 있다면 함수 내부로 진입하고 없다면 1 줄 진행한다.

n 은 함수가 있던 없던 그냥 1 줄 진행한다.

l 은 'list' 의 약자로 C 코드가 보인다.

c 는 'continue'의 약자 다음 브레이크 포인트를 만날 때까지 계속 작업한다.

*scope

```
int main(void)
{       int global_area = 1;
        {   int local_area1 = 2;
            printf("global_area = %d₩n", global_area);
            printf("local_area1 = %d₩n", local_area1);       }
        {   int local_area2 = 3;
            printf("global_area = %d₩n", global_area);
            printf("local_area2 = %d₩n", local_area2);       }

    printf("global_area = %d₩n", global_area);
    printf("local_area1 = %d₩n", local_area1);
    printf("local_area2 = %d₩n", local_area2);

    return 0; }
```

'{'으로 시작해서 '}'으로 끝나는 영역을 말한다. 이 구간 안에서 지역 공간이 생길 수 있다. 스택을 관리한다. '{'로 생성하고 '}'로 해제한다고 생각하면 된다.

옆의 코드를 그냥 입력하게 되면 오류가 나는데, '{ }'이 스코프 영역이 메인이 아니기 때문이다. 앞의 local_area1, local_area2는 '{ }'로 묶여서 **본인 영역에서 끝난 것이기 때문에 다른 곳에 영향을 줄 수가 없다.** 따라서, 아예 표시 된 두 줄을 삭제 하던지, main 함수 안에서 선언되는 local을 main 밖으로 써야 한다.

Ex) int local_area1 = 2;
int local_area2 = 3;
int main (void)

> 이 경우, 이 두 줄은 함수 밖에서 선언한 변수로 '전역변수'라 한다.
전역변수는 변수의 위치에 관계 없이 data값을 참조할 수 있다.
즉, 전역변수는 어디든 갈 수 있다.

```
int number = 0;
  while(1)
  {   number++;
            if(number == 5)
              continue;
            printf("%d₩n",number);
                    if(number == 10)
                  break;              }
```

*static Keyword

정적 변수를 말한다. 전역변수를 억지로 만드는 것이다. static 변수는 전역 변수와 마찬가지로 data 영역에 load된다. 지역 변수를 static으로 선언했다면, 이 변수를 선언한 함수 내에서만 접근 가능하다.

*continue 문

무한대 혹은 값이 무조건 0이 나올 때 등의 특정 조건을 수행하지 않고 넘기고 싶을 때, 사용된다. 옆의 예시를 보면 number가 5일 때, 아래 조건을 수행하지 않고 while의 조건을 검색하러 돌아간다.

***do while문**

일단 조건이 만족하지 않더라도 1번은 수행한다. 즉, 무조건 1번은 실행하게 되어 있는 함수이다. 궁극적으로 do while을 사용하는 이유는 매크로 확장 때문이다. Kernel의 매크로에 자주 사용된다.

*** #define**

#define A B는 A를 B로 대체해주는 것이다. 쉬운 대입과 변경 때문에 사용된다. 예를 들어서 회사 코드에 100 번 루프(반복)를 돌아야 하는 코드가 777 개 있다. 이때, 회사 규모가 커져서 100 번 루프가 아니라 500 번 루프를 돌아야 하는 상황이 되었다고 가정하자. 그러면 while(i < 100) 했던 부분을 전부다 찾아서 while(i < 500) 으로 변경해야 한다. 또한, while 만 있는것이 아니라 다른 코드들도 100 이라는 숫자에 관계된 코드들이 존재할 수 있다는 문제가 있다. 그러면 숫자 하나 바꾸는 것이 모든 프로그램을 뜯어고치는 대 공사가 될 수 있는데, #define TEST 100으로 선언하고 애초에 while(i < TEST) 로 만들어놨다면, #define TEST 500 으로 1 번 변경해서 모든 변경을 수월하게 진행할 수 있다.

***for 문**

```
int main(void)
 {      int i = 0, result = 'A';
      while(i < 10)
       { printf("%c₩n", result);
         result++;
         i++;       }
return 0;
 }
```

```
#include <stdio.h>
 int main(void)
 {   int i, result;
   for(i = 0, result = 'A'; i < 10; i++, result++)
   {
      printf("%c₩n", result);
   }
   return 0;
 }
```

while문은 초기화, 조건식, 증감식은 가독성이 떨어지는 면이 있다. 이럴 때, for 문을 쓴다. **for(초기화; 조건식; 증감식)**으로 구성되기 때문이다. 초기화와, 조건식 끝날 때 **';'** 를 써주어야 한다. 이 때, **for(;;)**를 사용하면 다른 조건을 막론하고 for의 조건식이 비어있으면 **무한루프**가 된다.

*** goto 의 이점과 CPU 파이프라인**

흔히, goto문은 초보자가 다루기 힘들어 사용하지 말라고 말하는 코드이다. 그러나 System Programming에서 배우겠지만 goto를 활용하면 특정한 상황을 아주 간결하게 해결할 수 있는 경우가 많다.

밑에 goto 예시는 if 와 break 를 조합한 버전과 goto 로 처리하는 버전을 가지고 있다. for문 수가 많을수록, break 거는 수도 많아진다(for에 break를 걸기 때문이다). break를 걸기 위해선 flag선언이 필요하다. 그리고 for문 끝마다 if(flag){ break; }를 해주어야 break를 할 수 있다. 또한, error가 났지만 코드는 계속 진행된다. 나중에 확인하고 error가 났기 때문에 사용할 수 없어 그냥 시간을 낭비하는 경우도 있다. 그러나 goto문을 사용하면 error가 난 순간에 멈출 수 있다. 즉, 가독성과 효율성 면에서 goto문이 더 좋다.

```
#include <stdio.h>
int main(void)
{       int i, j, k; int flag = 0;

        for(i=0; i<5; i++)
        {
                for(j=0; j<5; j++)
                {
                        for(k=0; k<5; k++)
                        {
                                if((i==2) && (j==2) && (k==2))
                                {       printf("Error!!!\n");
                                        flag = 1;               }
                                else
                                {       printf("Data\n");       }
                                if(flag)
                                {       break;                  }
                        }
                        if(flag)
                        {       break;                  }
                }
                if(flag)
                {       break;                  }
        }
        return 0;
}
```

```
#include <stdio.h>
int main(void)
{       int i, j, k;

        for(i=0; i<5; i++)
        {
                for(j=0; j<5; j++)
                {
                        for(k=0; k<5; k++)
                        {
                                if((i==2) && (j==2) && (k==2))
                                {       printf("Error!!!\n");
                                        goto err_handler;       }
                                else
                                {       printf("Data\n");       }
                        }
                }
        }

        return 0;
err_handler:
printf("Goto Zzang!\n");

 return -1;

}
```

if 문은 기본적으로 (어셈블리어로 볼 때) mov, cmp, jmp 로 구성된다. 그러나 goto 는 jmp 하나로 끝이다. 즉, for 문이 여러 개 생기면 if, break 조합의 경우, for 문의 갯수만큼 mov, cmp, jmp 를 해야 한다. 여기서 문제는 바로 jmp 명령어다.

call 이나 jmp 를 CPU Instruction(명령어) 레벨에서 분기 명령어라고 하고 이들은 CPU 파이프라인에 매우 치명적인 손실을 가져다 준다. 기본적으로 아주 단순한 CPU 의 파이프라인을 설명하자면 아래와 같은 3 단계로 구성된다.

1. Fetch - 실행해야할 명령어를 물어옴
2. Decode - 어떤 명령어인지 해석함
3. Execute - 실제 명령어를 실행시킴

파이프라인이 짧은 것부터 긴 것이 5 단계 ~ 수십 단계로 구성된다. ARM, Intel 등등 다양한 프로세서들 모두 마찬가지이다. 그런데 왜 jmp 나 call 등의 분기 명령어가 문제가 될까?

기본적으로 분기 명령어는 파이프라인을 때려부수기 때문이다. 이 뜻은 위의 가장 단순한 CPU 가 실행까지 3 clock 을 소요하는데 파이프라인이 깨지니, 쓸데없이 또 다시 3 clock 을 버려야 하는 것을 의미한다.

만약 파이프라인의 단계가 수십 단계라면, 분기가 여러 번 발생하면 파이프라인 단계 x 분기 횟수만큼 CPU clock 을 낭비하게 된다. 즉 성능면에서도 goto 가 월등히 압도적이다. jmp 1 번에 끝나기 때문이다.

*** SW 와 HW 동작을 생각할 때 주의할 점**

SW 는 멀티 코어 상황이 아니면 어떤 상황에서도 한 번에 한 가지 동작만 실행할 수 있다. 좀 더 정확하게 이야기 하자면, CPU 한 개는 오로지 한 순간에 한 가지 동작만 할 수 있다는 것이다. 반면 HW 회로는 병렬 회로가 존재하듯이 모든 회로가 동시에 동작할 수 있다. 파이프라인은 CPU 에 구성된 회로이기 때문에 모든 모듈들이 동시에 동작할 수 있는 것이다. FPGA 프로그래밍은 병렬 동작이다. 실제로 CPU 설계를 FPGA 가지고 한다. FPGA 프로그래밍은 회로를 만든다고 생각하면 된다.

***재귀 함수 호출**

사용한 함수를 다시 호출하는 방식이다. Program 구현상 반드시 필요한 경우가 있다. 무조건 좋지는 않지만, 편의가 따른다. '좋지 않다.'란 함수 호출로 파이프 라인이 깨지기 때문이다. 스택프레임을 만들고 없애는 과정이 생기 때문이다. 즉, 성능이 떨어진다. 그래서 성능 등을 잘 고려해서 구현해야 한다.

밑은 재귀 함수 호출에 대한 예시이다. 코드와 사진

```
#include <stdio.h>
int fib(int num)
{       if(num == 1 || num == 2)
            return 1;
        else
            return fib(num – 1) + fib(num – 2);
 }
int main(void)
{
        int result, final_val;
        printf("피보나치 수열의 항의 개수를 입력하시오 : ");
        scanf("%d", &final_val);        result = fib(final_val);
        printf("%d번째 항의 수는 = %d₩n", final_val, result);
        return 0;}
```

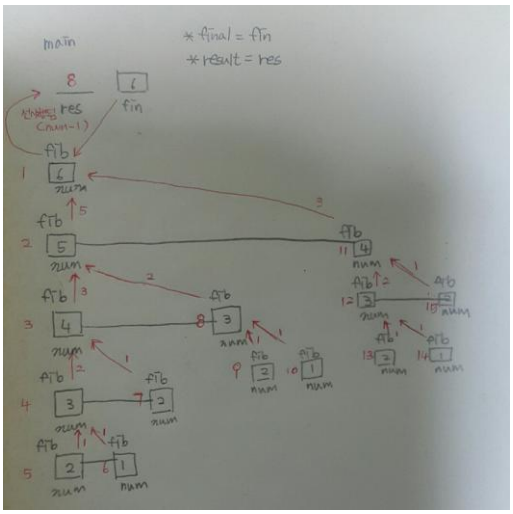
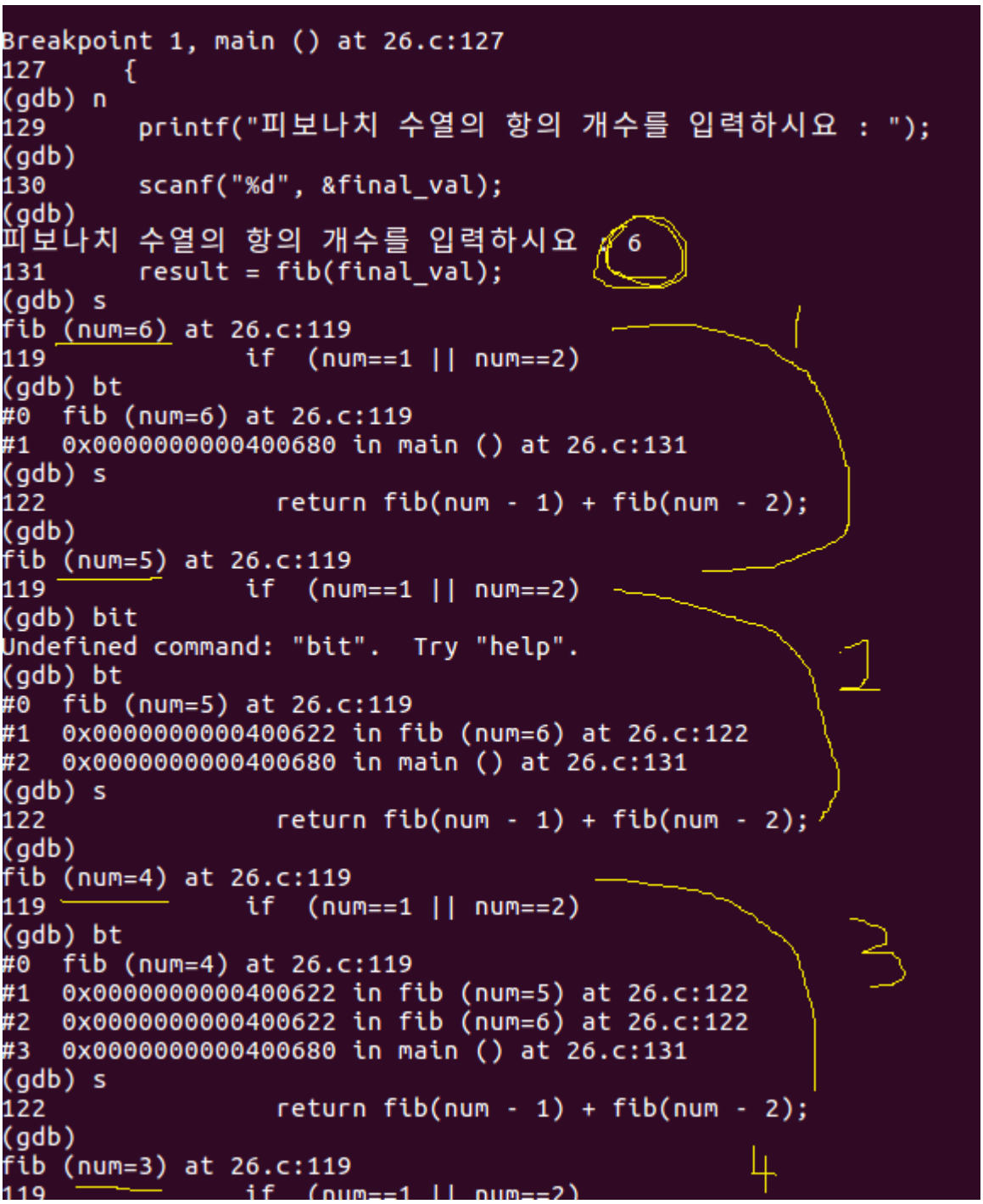
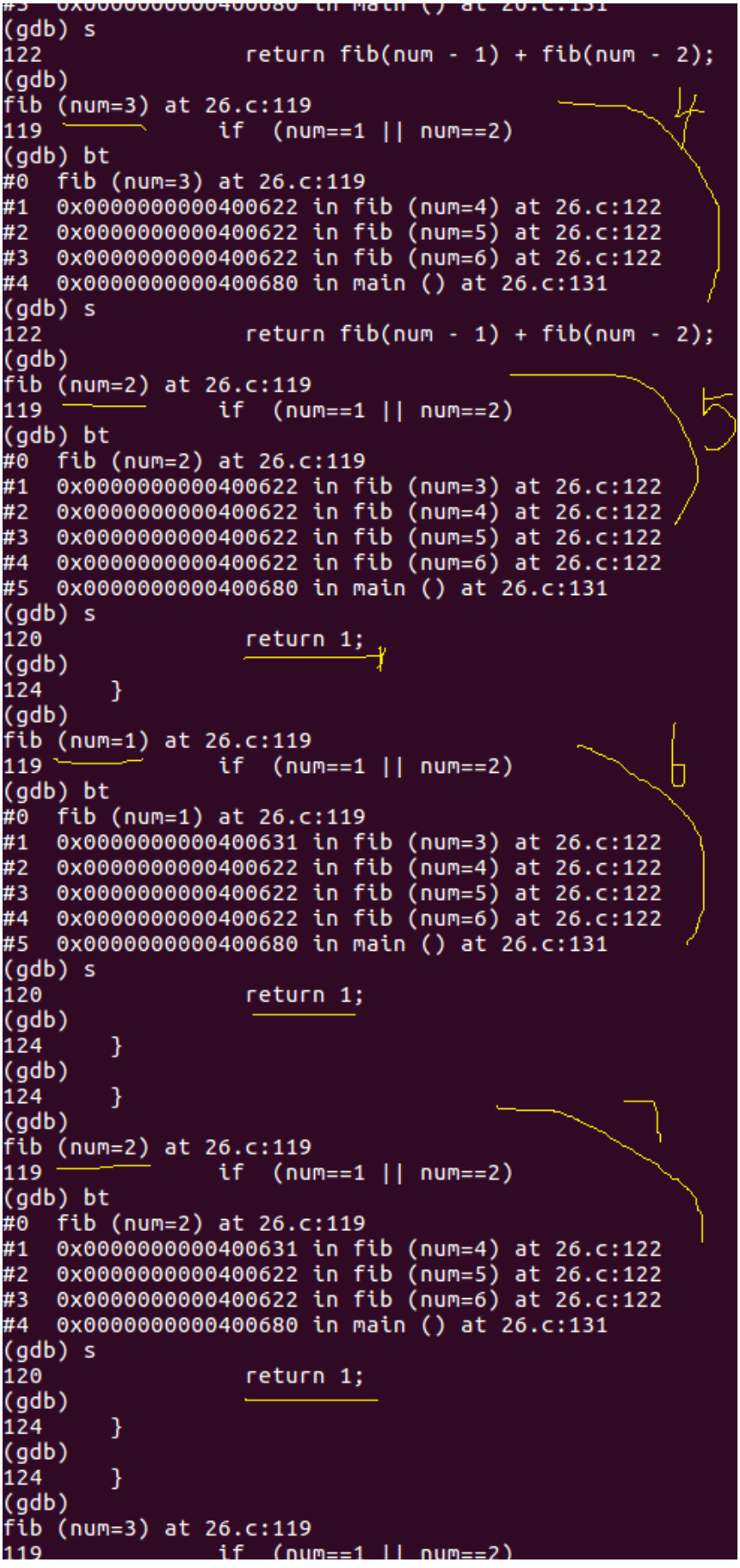
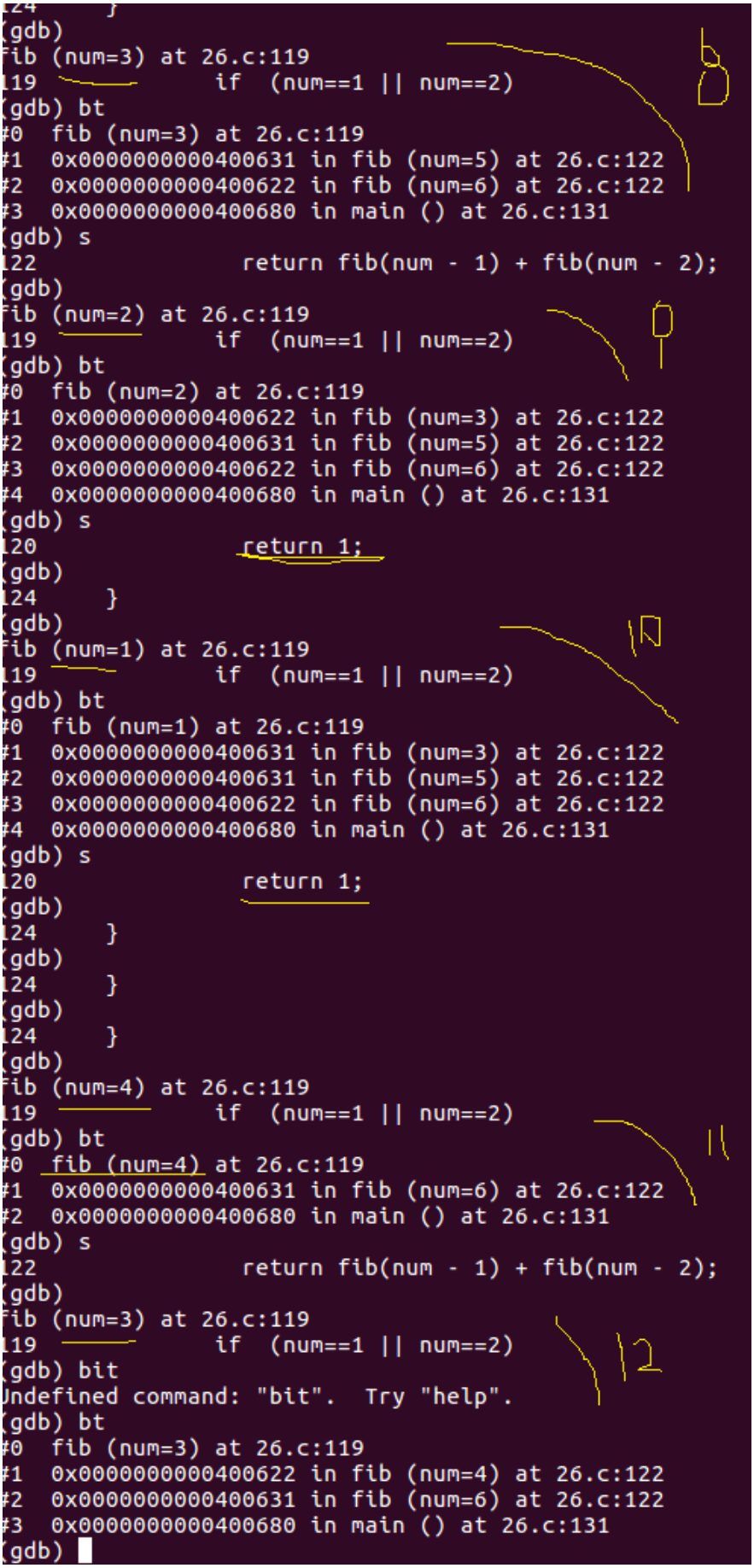


그림 예시로 순번과 반환되는 값이다. 답은 8이다.


Breakpoint 1, main () at 26.c:127
127 {
(gdb) n
129 printf("피보나치 수열의 항의 개수를 입력하시요 : ");
(gdb)
130 scanf("%d", &final_val);
(gdb)
피보나치 수열의 항의 개수를 입력하시요 : 6
131 result = fib(final_val);
(gdb) s
fib (num=6) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=6) at 26.c:119
#1 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=5) at 26.c:119
119 if (num==1 || num==2)
(gdb) bit
Undefined command: "bit". Try "help".
(gdb) bt
#0 fib (num=5) at 26.c:119
#1 0x0000000000400622 in fib (num=6) at 26.c:122
#2 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=4) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=4) at 26.c:119
#1 0x0000000000400622 in fib (num=5) at 26.c:122
#2 0x0000000000400622 in fib (num=6) at 26.c:122
#3 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=3) at 26.c:119
119 if (num==1 || num==2)
1
3
4


(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=3) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=3) at 26.c:119
#1 0x0000000000400622 in fib (num=4) at 26.c:122
#2 0x0000000000400622 in fib (num=5) at 26.c:122
#3 0x0000000000400622 in fib (num=6) at 26.c:122
#4 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=2) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=2) at 26.c:119
#1 0x0000000000400622 in fib (num=3) at 26.c:122
#2 0x0000000000400622 in fib (num=4) at 26.c:122
#3 0x0000000000400622 in fib (num=5) at 26.c:122
#4 0x0000000000400622 in fib (num=6) at 26.c:122
#5 0x0000000000400680 in main () at 26.c:131
(gdb) s
120 return 1;
(gdb)
124 }
(gdb)
fib (num=1) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=1) at 26.c:119
#1 0x0000000000400631 in fib (num=3) at 26.c:122
#2 0x0000000000400622 in fib (num=4) at 26.c:122
#3 0x0000000000400622 in fib (num=5) at 26.c:122
#4 0x0000000000400622 in fib (num=6) at 26.c:122
#5 0x0000000000400680 in main () at 26.c:131
(gdb) s
120 return 1;
(gdb)
124 }
(gdb)
124 }
(gdb)
fib (num=2) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=2) at 26.c:119
#1 0x0000000000400631 in fib (num=4) at 26.c:122
#2 0x0000000000400622 in fib (num=5) at 26.c:122
#3 0x0000000000400622 in fib (num=6) at 26.c:122
#4 0x0000000000400680 in main () at 26.c:131
(gdb) s
120 return 1;
(gdb)
124 }
(gdb)
124 }
(gdb)
fib (num=3) at 26.c:119
119 if (num==1 || num==2)


124 }
(gdb)
fib (num=3) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=3) at 26.c:119
#1 0x0000000000400631 in fib (num=5) at 26.c:122
#2 0x0000000000400622 in fib (num=6) at 26.c:122
#3 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=2) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=2) at 26.c:119
#1 0x0000000000400622 in fib (num=3) at 26.c:122
#2 0x0000000000400631 in fib (num=5) at 26.c:122
#3 0x0000000000400622 in fib (num=6) at 26.c:122
#4 0x0000000000400680 in main () at 26.c:131
(gdb) s
120 return 1;
(gdb)
124 }
(gdb)
fib (num=1) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=1) at 26.c:119
#1 0x0000000000400631 in fib (num=3) at 26.c:122
#2 0x0000000000400631 in fib (num=5) at 26.c:122
#3 0x0000000000400622 in fib (num=6) at 26.c:122
#4 0x0000000000400680 in main () at 26.c:131
(gdb) s
120 return 1;
(gdb)
124 }
(gdb)
124 }
(gdb)
124 }
(gdb)
fib (num=4) at 26.c:119
119 if (num==1 || num==2)
(gdb) bt
#0 fib (num=4) at 26.c:119
#1 0x0000000000400631 in fib (num=6) at 26.c:122
#2 0x0000000000400680 in main () at 26.c:131
(gdb) s
122 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=3) at 26.c:119
119 if (num==1 || num==2)
(gdb) bit
Undefined command: "bit". Try "help".
(gdb) bt
#0 fib (num=3) at 26.c:119
#1 0x0000000000400622 in fib (num=4) at 26.c:122
#2 0x0000000000400631 in fib (num=6) at 26.c:122
#3 0x0000000000400680 in main () at 26.c:131
(gdb)

```
(gdb) bt
#0  fib (num=3) at 26.c:119
#1  0x0000000000400622 in fib (num=4) at 26.c:122
#2  0x0000000000400631 in fib (num=6) at 26.c:122
#3  0x0000000000400680 in main () at 26.c:131
(gdb) s
122                 return fib(num - 1) + fib(num - 2);
(gdb)
fib (num=2) at 26.c:119
119         if  (num==1 || num==2)
(gdb) bt
#0  fib (num=2) at 26.c:119
#1  0x0000000000400622 in fib (num=3) at 26.c:122
#2  0x0000000000400622 in fib (num=4) at 26.c:122
#3  0x0000000000400631 in fib (num=6) at 26.c:122
#4  0x0000000000400680 in main () at 26.c:131
(gdb) s
120                 return 1;
(gdb)
124     }
(gdb)
fib (num=1) at 26.c:119
119         if  (num==1 || num==2)
(gdb) bt
#0  fib (num=1) at 26.c:119
#1  0x0000000000400631 in fib (num=3) at 26.c:122
#2  0x0000000000400622 in fib (num=4) at 26.c:122
#3  0x0000000000400631 in fib (num=6) at 26.c:122
#4  0x0000000000400680 in main () at 26.c:131
(gdb) s
120                 return 1;
(gdb)
124     }
(gdb)
124     }
(gdb)
fib (num=2) at 26.c:119
119         if  (num==1 || num==2)
(gdb) bt
#0  fib (num=2) at 26.c:119
#1  0x0000000000400631 in fib (num=4) at 26.c:122
#2  0x0000000000400631 in fib (num=6) at 26.c:122
#3  0x0000000000400680 in main () at 26.c:131
(gdb) s
120                 return 1;
(gdb)
124     }
(gdb)
124     }
(gdb)
124     }
(gdb)
main () at 26.c:132
132     printf("%d번째 항의 수는 = %d\n", final_val, result);
(gdb)
__printf (format=0x400786 "%d번째 항의 수는 = %d\n") at printf.c:28
28      printf.c: No such file or directory.
(gdb)
2       in printf.c
(gdb)
3       in printf.c
(gdb)
```

While문 > for 문

3번

#include <stdio.h>

```
// synthesis : first – start, second – end, three - times
int   syn(int start, int end, int times)
{
        int res = 0, i = start;

        while(i < end + 1)
        {
                if(! ( i   % 3))
                {       res += i;
                }
                ++i;
        }
        return res;
}


int main(void)
{
        printf("tot series sum = %d\n",   syn(1, 1000, 3));
        return 0;
}
```

for문으로..

```
#include <stdio.h>

// synthesis : first – start, second – end, three - times
int   syn(int start, int end, int times)
{
        int res, i;

        for(res=0, i=start; i < end + 1; ++i)
        {
                if(! ( i   % 3))
                {       res += i;
                }
         }
        return res;
}


int main(void)
{
        printf("tot series sum = %d\n",   syn(1, 1000, 3));
        return 0;
}
```

4번

```
#include <stdio.h>

int syn(int start, int end, int t1, int t2)
{
	int res = 0, i = start;

	while(i < end + 1)
	{
		if(((i % 4) == 1) || ((i % 6) == 1))
		{
			res += i;
		}
		i++;
	}

	return res;
}

int main(void)
{
	printf("tot series sum = %d\n", syn(1, 1000, 4, 6));
	return 0;
}
```

For 문으로

```
#include <stdio.h>

int syn(int start, int end, int t1, int t2)
{
	int res, i;

	for(res=0, i=start; i < end + 1; i++)
	{
		if(((i % 4) == 1) || ((i % 6) == 1))
		{
			res += i;
		}
	}

	return res;
}

int main(void)
{
	printf("tot series sum = %d\n", syn(1, 1000, 4, 6));
	return 0;
}
```

10번

```
#include <stdio.h>

void print_rom(void)
{
	int i =2, j=1;

	while(i<10)
	{
		while(j<10)
		{
			printf("%d x %d = %d\n",i,j,i*j);
			j++;
		}
		j = 1;
		i++;
	}

}

int main(void)
{
	print_rom();
	return 0;
}
```

for문으로

```
#include <stdio.h>

void print_rom(void)
{
	int i =2, j=1;

	for(i=2; i<10; i++)
	{
		for(j=1; j<10; j++)
		{
			printf("%d x %d = %d\n",i,j,i*j);
		 }
	}

}

int main(void)
{
	print_rom();
	return 0;
}
```