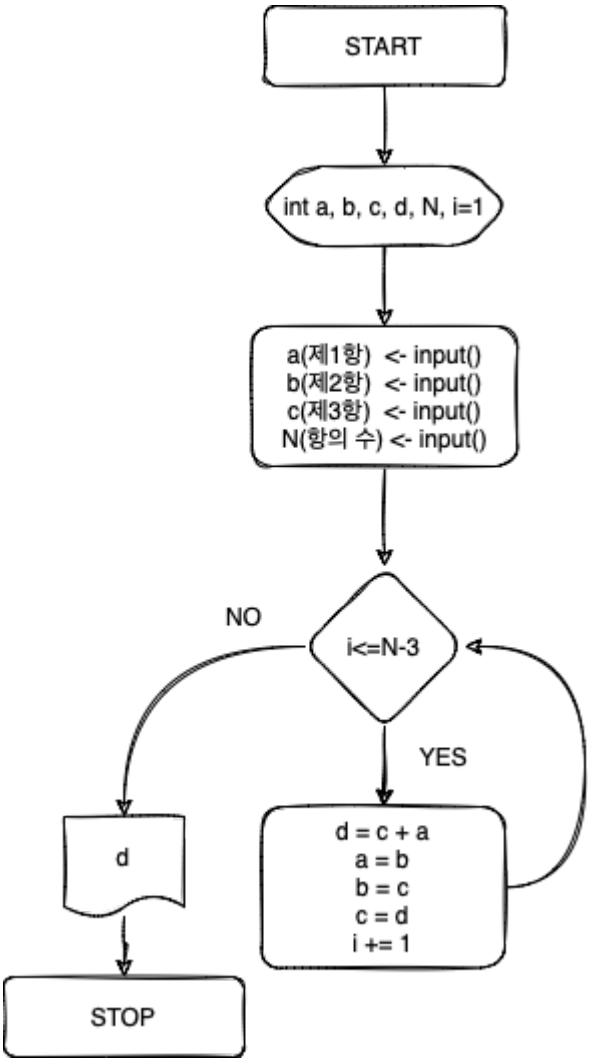
SW3106 : 프로그래밍 입문 Project #1

1-1. Problem1_FlowChart



1-2. Problem1_iteration

점화식

F(n+3) = F(n+2) + Fn (n=1, 2, 3 ...)

F1 = 1, F2 = 2, F3 = 3이라 하고 **10**번째 항의 수를 구한다고 해보자.

시행수(i)	F(i)	F(i+1)	F(i+2)	F(i+3)
1	1	2	3	4
2	2	3	4	6
3	3	4	6	9
4	4	6	9	13
5	6	9	13	19

시행수(i)	F(i)	F(i+1)	F(i+2)	F(i+3)
6	9	13	19	28
7	**13**	**19**	**28**	**41**

여기서 알 수 있는 것은 시행수가 i일 때, i+3번째 값을 구할 수 있다. 즉, i-3번 반복하면 i번째 값을 구할 수 있다.

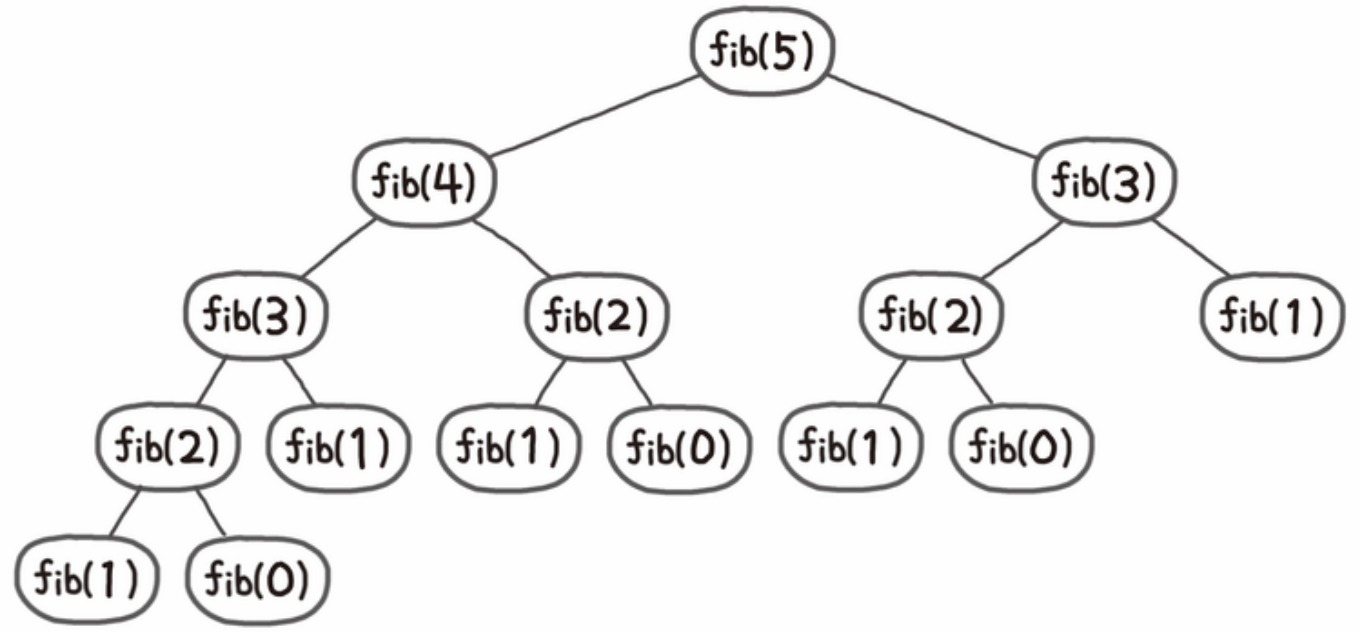
[Prob1 iteration: Main Source Code]

```
#include <stdio.h>

int main()
{
    int a, b, c, d, N, i;

    printf("제 1항(a), 제 2항(b), 제 3항(c), 항의 수(N)을 입력하시오:\n");
    scanf("%d %d %d %d", &a, &b, &c, &N);
    for(i = 1; i <= N-3; i++) {
        d = c + a;
        a = b;
        b = c;
        c = d;
    }
    printf("%d항: %d\n", N, d);

    return 0;
}
```

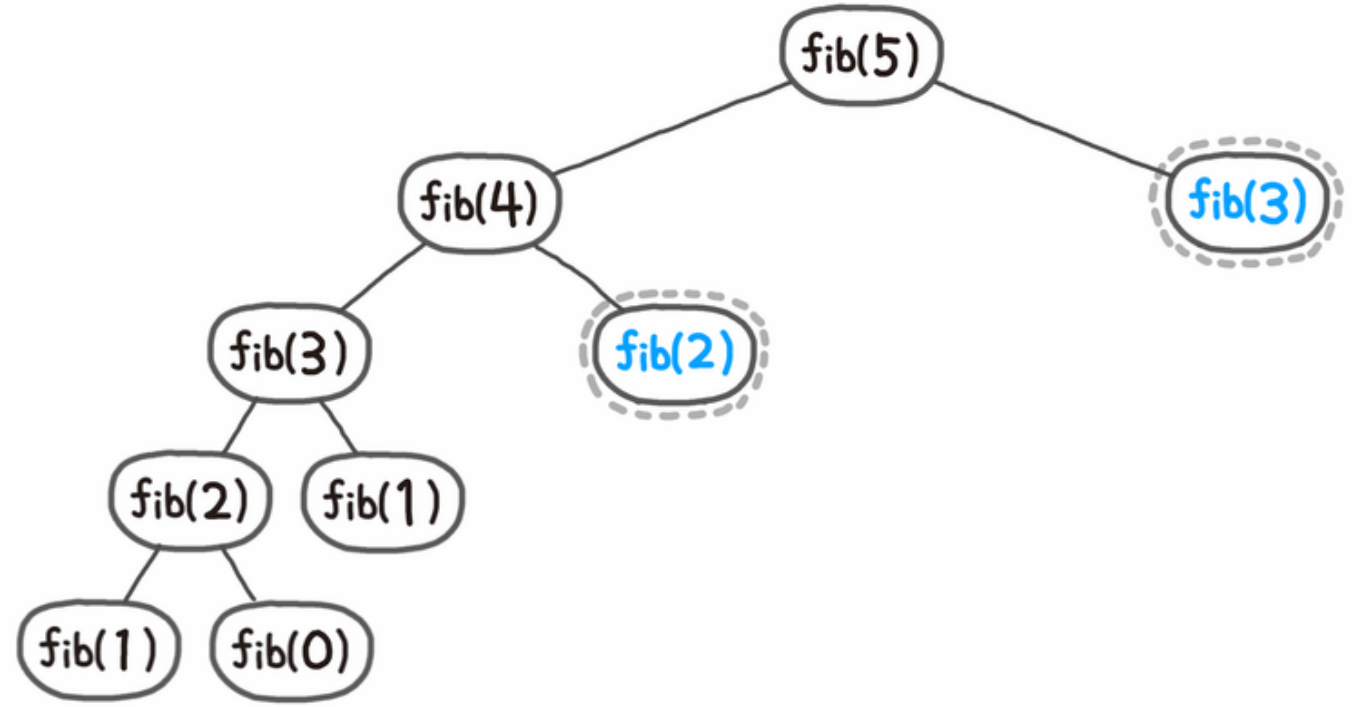
1-3. Problem1_recursive

피보나치 점화식

Fib(n+2) = Fib(n+1) + Fib(n) (n=1, 2, 3 ...)



위와 같이 5번째 피보나치 수열을 구하는 데 함수 f를 호출하는 횟수는 총 15번이다. 위의 예시에서 중복해서 계산하는 값만 따져 봐도 Fib(3)이 2번, Fib(2)가 3번, Fib(1)을 5번, Fib(0)을 3번 계산한다. 15번의 계산 중에 무려 11번을 중복해서 계산하는 셈이다. 비록 위 예시는 비교적 작은 값을 제시했지만, 피보나치 수열을 순진(naive)한 방법으로 구할 경우 시간복잡도는 피보나치 수열의 값에 따라 폭발적으로 증가한다. 즉 O(2^N)다.



하지만 메모이제이션(Memoization) 기법을 사용하면 한 번 계산한 결과를 메모리에 저장해두었다가 꺼내 씀으로써 중복 계산을 방지할 수 있고 이를 이용하면 O(N)이 되며 Fib(5)를 구하는 과정은 위와 같아진다. [Prob1 recursive : Main Source Code]

```
#include <stdio.h>

int d[100] = {0};

int fibo(int a, int b, int c, int n) {
    if (n == 1)
```

```
        return a;
    if (n == 2)
        return b;
    if (n == 3)
        return c;
    if (d[n] != 0)
        return d[n];

    return d[n] = fibo(a, b, c, n - 1) + fibo(a, b, c, n - 3);
}

int main() {
    int a, b, c, N;

    printf("제 1항(a), 제 2항(b), 제 3항(c), 항의 수(N)을 입력하시오:\n");
    scanf("%d %d %d %d", &a, &b, &c, &N);

    printf("%d항: %d\n", N, fibo(a, b, c, N));

    return 0;
}
```

2. Problem2

커피, 케익, 샌드위치의 주문 수량을 입력받을 때, 아래의 함수를 이용해보자.

[Prob2 : Source Code #1]

```
int* get_setArray(int coffee, int cake, int sandwich) {
    int temp[3] = {coffee, cake, sandwich};
    int *set = (int*)malloc(5);
    int min = temp[0];

    // 수량 중 0이 하나라도 있을 때
    if(min == 0) {
        set[1] = 0;
        set[4] = SET_NOT_EXIST;
        return set;
    }

    int i;
    int min_idx = 0;
    for(i=1; i<3; i++) {
        // 수량 중 0이 하나라도 있을 때
        if(temp[i] == 0) {
            set[1] = 0;
            set[4] = SET_NOT_EXIST;
            return set;
        }

        if(min > temp[i]) {
```

```
            min = temp[i];
            min_idx = i;
        }
    }

    set[0] = 12000;          // 세트가격
    set[1] = min;            // 주문 수량 제일 적은 수 = 세트 수
    set[2] = 12000*min;      // 세트 총 가격
    set[3] = min_idx;        // 주문 수량 제일 적은 것의 인덱스

    // 세트로 묶이는지 여부
    // SET_EXIST : 세트 존재, SET_NOT_EXIST : 세트 존재 x, ALL_SAME : 수량이 같음
    if(temp[0] == temp[1] && temp[0] == temp[2])
        set[4] = ALL_SAME;    // 모두 수량이 같을 때
    else
        set[4] = SET_EXIST;

    return set;
}
```

위 함수를 호출할 때 인자값으로 커피, 케익, 샌드위치의 주문 수량에 따라 경우를 분류하여 최종적으로 동적할당된 set를 반환한다.

이때 우리가 만날 수 있는 경우는 다음과 같다.

1. 주문 수량이 모두 같을 때
2. 주문 수량이 모두 다를 때
    1. 주문 수량 중 0이 포함될 때
    2. 주문 수량 중 0이 포함되지 않을 때

먼저 1번째 경우 소스코드를 보면 get_setArray() 함수가 set를 반환하기 전 set[4]에 -1을 대입한다. 이는 모든 수량이 같은 경우이며 함수를 통해 받아온 set를 통해 set[4]가 -1일 때 세트만 출력하도록 설정한다.

2-1의 경우 0이 있으면 set[1] = 0, set[4] = 0을 한 뒤 set를 반환하도록 했는데, 아래의 코드를 먼저 봐보도록 하자.

[Prob2 : Source Code #2]

```
void print_price(int coffee, int cake, int sandwich) {
    int total = 0;
    int* set = get_setArray(coffee, cake, sandwich);
    int price[3] = {4000, 5000, 6000};
    int product_num[3] = {coffee, cake, sandwich};
    char* product[3] = {"커피", "케익", "샌드위치"};

    printf("%-10s\t %s\t %s\t %s\t\n", "품목", "가격", "갯수", "금액");

    // set가 존재할 경우
    if(set[4] == SET_EXIST) {
        printf("%-8s\t %-5d\t %-2d\t %-5d\t\n", "세트", set[0], set[1],
set[2]);
        total += set[2];
```

```
    }

    // 수량이 모두 같지 않을 때
    int i;
    if(set[4] != ALL_SAME)
        for(i=0; i<3; i++) {
            if(i == set[3] && set[4] != SET_NOT_EXIST)
                continue;
            int temp_price = price[i] * (product_num[i] - set[1]);
            printf("%-8s\t %-5d\t %-2d\t %-5d\t\n", product[i], price[i],
product_num[i] - set[1], temp_price);
            total += temp_price;
        }
    printf("-----------------------------------------\n");
    printf("총 지불 금액\t\t\t %-7d\n", total);
}
```

print_price() 함수에서 반복문을 보면 알 수 있듯이 세트메뉴로 묶일 것을 염두하여 작성했다. 그렇기 때문에 추가코드를 작성하지 않기 위해 0이 있으면 set[1] = 0, set[4] = 0을 한 뒤 set를 반환하도록 했다.

2-2의 경우는 주문수량 중 제일 적은 것을 찾아낸 내용을 담은 set를 바탕으로 주문수량이 제일 적은 것이 출력될 자리에 세트메뉴가 들어가고 나머지 메뉴는 세트메뉴 다음으로 출력하도록 했다.

[Prob2 : Main Source Code]

```
#include <stdio.h>
#include <stdlib.h>
#define SET_EXIST 1
#define SET_NOT_EXIST 0
#define ALL_SAME -1

void print_price(int coffee, int cake, int sandwich);
int* get_setArray(int coffee, int cake, int sandwich);

int main() {
    int coffee, cake, sandwich;
    printf("주문하고자 하는 커피, 케익, 샌드위치의 갯수를 각각 입력하세요:\n");
    scanf("%d %d %d", &coffee, &cake, &sandwich);

    print_price(coffee, cake, sandwich);

    return 0;
}

void print_price(int coffee, int cake, int sandwich) {
    int total = 0;
    int* set = get_setArray(coffee, cake, sandwich);
    int price[3] = {4000, 5000, 6000};
    int product_num[3] = {coffee, cake, sandwich};
    char* product[3] = {"커피", "케익", "샌드위치"};
```

```
    printf("%-10s\t %s\t %s\t %s\t\n", "품목", "가격", "갯수", "금액");

    // set가 존재할 경우
    if(set[4] == SET_EXIST) {
        printf("%-8s\t %-5d\t %-2d\t %-5d\t\n", "세트", set[0], set[1],
set[2]);
        total += set[2];
    }

    // 수량이 모두 같지 않을 때
    int i;
    if(set[4] != ALL_SAME)
        for(i=0; i<3; i++) {
            if(i == set[3] && set[4] != SET_NOT_EXIST)
                continue;
            int temp_price = price[i] * (product_num[i] - set[1]);
            printf("%-8s\t %-5d\t %-2d\t %-5d\t\n", product[i], price[i],
product_num[i] - set[1], temp_price);
            total += temp_price;
        }
    printf("-----------------------------------------\n");
    printf("총 지불 금액\t\t\t %-7d\n", total);
}

int* get_setArray(int coffee, int cake, int sandwich) {
    int temp[3] = {coffee, cake, sandwich};
    int *set = (int*)malloc(5);
    int min = temp[0];

    // 수량 중 0이 하나라도 있을 때
    if(min == 0) {
        set[1] = 0;
        set[4] = SET_NOT_EXIST;
        return set;
    }

    int i;
    int min_idx = 0;
    for(i=1; i<3; i++) {
        // 수량 중 0이 하나라도 있을 때
        if(temp[i] == 0) {
            set[1] = 0;
            set[4] = SET_NOT_EXIST;
            return set;
        }

        if(min > temp[i]) {
            min = temp[i];
            min_idx = i;
        }
    }

    set[0] = 12000;          // 세트가격
    set[1] = min;            // 주문 수량 제일 적은 수 = 세트 수
```

```
    set[2] = 12000*min;     // 세트 총 가격
    set[3] = min_idx;       // 주문 수량 제일 적은 것의 인덱스

    // 세트로 묶이는지 여부
    // SET_EXIST : 세트 존재, SET_NOT_EXIST : 세트 존재 x, ALL_SAME : 수량이 같음
    if(temp[0] == temp[1] && temp[0] == temp[2])
        set[4] = ALL_SAME;    // 모두 수량이 같을 때
    else
        set[4] = SET_EXIST;

    return set;
}
```

3. Problem3

먼저 입력받은 정수를 16자리의 이진수로 표현하기 위해 0xFFFF와 AND 연산을 한다.

&연산을 마친 데이터를 target이라는 변수에 대입하고 맨 처음부터 하나씩 >> 시프트 연산을 한다. 만약 입력받은 데이터가 17이라는 정수라고 하자. 그럼 target이 의미하는 데이터는 0000000000010001일 것이다.

제일 앞에 있는 숫자는 target>>15로 시프트 연산한 뒤 0b01 즉, 10진수 기준 1과 &연산하여 해당 숫자가 0인지 1인지 판단한다. 이 과정에서 기본적으로 0은 출력하지 않지만 처음으로 1이 나온 시점부터 이후의 0들은 의미가 있기에 find라는 변수를 만들어 처음으로 1이 나온 시점부턴 0도 같이 출력하도록 제어한다.

[Prob3 : Main Source Code]

```
#include <stdio.h>

void binaryFunc(int n);

int main() {
    int input_num;
    printf("값을 입력해주세요: ");
    scanf("%d", &input_num);

    binaryFunc(input_num);

    return 0;
}

void binaryFunc(int n) {
    unsigned short int target = (n&0xFFFF);
    char find = 0;

    for(int i=15; i>-1; i--)
    {
        if(find == 0 && ((target>>i)&0b01) == 0)
        {
        }else{
            find = 1;
            printf("%d", ((target>>i)&0b01));
```

```
            }
        }
        printf("\n");
    }
```

4. Problem4

4번 문제에서 나온 특징을 가지는 이진수를 간단히 **이친수**라고 하자. 일단 문제에서 요구한 것은 다음과 같다

1. 이친수의 개수를 출력하기
2. 배열의 첫 원소는 n자리수, 그다음으로는 n자리수의 모든 이친수
3. 모든 이친수 출력하기

이것을 해결하기 전에 먼저 알아보아야 할 것이 있다.

n=1	n=2	n=3	n=4	n=5	n=6
1	10	100	1000	10000	100000
		101	1001	10001	100001
			1010	10010	100010
				10100	100100
				10101	100101
					101000
					101001
					101010
1개	**1개**	**2개**	**3개**	**5개**	**8개**

여기서 우리는 이친수의 갯수가 피보나치 수열과 같다는 것을 알 수 있다. 그렇기 때문에 이친수의 갯수 점화식은 아래와 같다.

점화식

F(n+2) = F(n+1) + Fn (n=1, 2, 3 ...)

이제 우리가 알아야 할 것은 이친수의 규칙이다. 위의 이친수 표를 잘 보면 알 수 있지만 n자리수의 이친수는 다음과 같은 과정을 거치면 만들 수 있다.

1. 1 << (n-1)을 한 기본틀
2. n-1자리 이친수들과 1 << (n-2)를 XOR 연산한 결과를 모은다.
3. 1 << (n-1)을 한 기본틀에 2번 결과들과 (n-2)자리 이친수들을 더한다.

n=5일 때를 예로 들어보자.

n=5	1 << (5-1)
10000	10000

n=5	1 << (5-1)
10001	
10010	
10100	
10101	

n=4	^1000
1000	0000
1001	0001
1010	0010

n=3
100
101

이를 소스코드로 구현하면 다음과 같다.

[Prob4 : Source Code #1]

```
#define MAX 1000
int binaryArray[MAX][MAX] = {0};

binaryArray[0][0] = 0b1;
binaryArray[1][0] = 0b10;

for(int i=2; i<n; i++) {
    int root = 1 << i;
    int idx_1 = getMaxIdx(binaryArray[i-1]);
    int idx_2 = getMaxIdx(binaryArray[i-2]);
    int new_idx = getMaxIdx(binaryArray[i]);

    for(int j=0; j<idx_1; j++)
        binaryArray[i][j] = root + (binaryArray[i-1][j] ^ (1 << (i-1)));

    new_idx = getMaxIdx(binaryArray[i]);

    for(int j=0; j<idx_2; j++)
        binaryArray[i][j+new_idx] = root + binaryArray[i-2][j];
}

int getMaxIdx(int* arr) {
    int idx = 0;
    for(int i=0; i<MAX; i++)
        if(arr[i] == 0 && arr[i+1] == 0) {
            idx = i;
            break;
```

```
        }
    return idx;
}
```

이차원 배열을 만들어 1 ~ n(입력받은 수)까지의 이친수를 저장한다. 이때 getMaxIdx() 함수의 역할은 n자리 이친수 일차원 배열 기준 이친수가 덜 채워진 부분을 찾아내기 위함이고 이를 이용하여 반복을 돌 때 덜 채워진 부분부터 이친수를 채울 수 있도록 한다.

[Prob4 : Main Source Code]

```
#include <stdio.h>
#define MAX 1000

int MyFinalArray[MAX] = {0};
int binaryArray[MAX][MAX] = {0};
int DP[MAX] = {0};
void getBinary(int n);
int getMaxIdx(int* arr);
void binaryFunc(int n);
long long int binaryCount(int n);

int main() {
    int n;
    printf("N을 입력하세요:\n");
    scanf("%d", &n);

    binaryArray[0][0] = 0b1;
    binaryArray[1][0] = 0b10;

    for(int i=2; i<n; i++) {
        int root = 1 << i;
        int idx_1 = getMaxIdx(binaryArray[i-1]);
        int idx_2 = getMaxIdx(binaryArray[i-2]);
        int new_idx = getMaxIdx(binaryArray[i]);

        for(int j=0; j<idx_1; j++)
            binaryArray[i][j] = root + (binaryArray[i-1][j] ^ (1 << (i-
1)));

        new_idx = getMaxIdx(binaryArray[i]);

        for(int j=0; j<idx_2; j++)
            binaryArray[i][j+new_idx] = root + binaryArray[i-2][j];
    }

    // 문제 조건에 맞는 배열 만들기
    long long int pinary_cnt = binaryCount(n);
    MyFinalArray[0] = n;
    for(int i=1; i<=pinary_cnt; i++)
        MyFinalArray[i] = binaryArray[n-1][i-1];
```

```
    printf("%d자리 갯수: %lld\n", n, pinary_cnt);
    for(int i=1; i<=getMaxIdx(binaryArray[n-1]); i++)
    {   if(i % 10 == 0)
            printf("\n");
        binaryFunc(MyFinalArray[i]);
    }
    printf("\n");

    return 0;
}

int getMaxIdx(int* arr) {
    int idx = 0;
    for(int i=0; i<MAX; i++)
        if(arr[i] == 0 && arr[i+1] == 0) {
            idx = i;
            break;
        }
    return idx;
}

void binaryFunc(int n) {
    unsigned short int target = (n&0xFFFF);
    char find = 0;
    int str_idx = 0;

    for(int i=15; i>-1; i--) {
        if(find == 0 && ((target>>i)&0x01) == 0)
        {
            //printf(" x");
        }else{
            find = 1;
            printf("%d", ((target>>i)&0x01));
        }
    }
    printf(" ");
}

long long int binaryCount(int n) {
    if(n == 1)
        return 1;
    if(n == 2)
        return 1;
    if(DP[n] != 0)
        return DP[n];

    return DP[n] = binaryCount(n - 1) + binaryCount(n - 2);
}
```