

기본 알고리즘

제1장



2017. Fall

국민대학교 소프트웨어학부 최준수

Recursion

- Recursion (재귀, 점화, 귀납) in Mathematics
 - 어떤 수학적 개체 (혹은 함수)를 정의함에 있어서, 그 개체 (혹은 함수)의 정의를 이용하여 정의하는 것

$$n! = \begin{cases} 1 & n = 0 & \text{(base case)} \\ n \times (n-1)! & n > 0 & \text{(recursive step)} \end{cases}$$

- **base case** : 함수의 값을 직접 계산할 수 있는 단순한 경우
- **recursive step** : 직접적으로 그 함수의 값을 계산할 수 없고, base case가 만들어 질 때까지 계속 환산해 나가면서 계산하는 단계

Recursion

- Recursion (재귀, 되부름) in Computer Science
 - 프로그램에서 어떤 함수(혹은 프로시저, 서브프로그램)에서 직접적으로 혹은 간접적으로 자기 자신 함수를 호출하는 것
 - **base case** : 직접 함수값을 계산할 수 있는 경우.
 - Recursion에서는 제일 먼저 입력값이 base case 에 해당하는 지를 먼저 검사하고 처리함.
 - 적어도 한 개 이상의 base case 가 있어야 함
 - **recursive step** : 직접 함수값을 계산할 수 없는 경우에는 base case를 이용하여 처리할 수 있도록 base case가 만들어 질 때까지 계속 자신을 재귀호출해 나가면서 계산하는 과정
 - 적절한 base case가 없으면 무한 loop 발생 (실제로는 stack overflow 발생)

Recursion

- 전산학에서 가장 기초적인 개념
- 많은 문제를 해결하는 가장 효율적인 알고리즘을 개발하는데 많이 사용되는 가장 강력한 수단
 - 분할정복기법
 - 동적계획법
 - 되추적기법
 - 기타
- 문제의 예
 - 가벼운 구슬 찾기
 - Merge/Quick Sort
 - 기타

Recursion

- Types of recursion
 - Unary recursion
 - a single recursive call for each recursion
 - examples
 - factorial, linear sum, reversing array, computing powers
 - Binary recursion
 - two recursive calls for each recursion
 - examples
 - Fibonacci numbers, Hanoi tower, merge sorting, quick sorting
 - Multiple recursion
 - more than two recursive calls for each recursion
 - example
 - flood fill, knight's tour

Factorial

- Factorial (recursive)

$$n! = \begin{cases} 1 & n = 0 \quad (\text{base case}) \\ n \times (n-1)! & n > 0 \quad (\text{recursive step}) \end{cases}$$

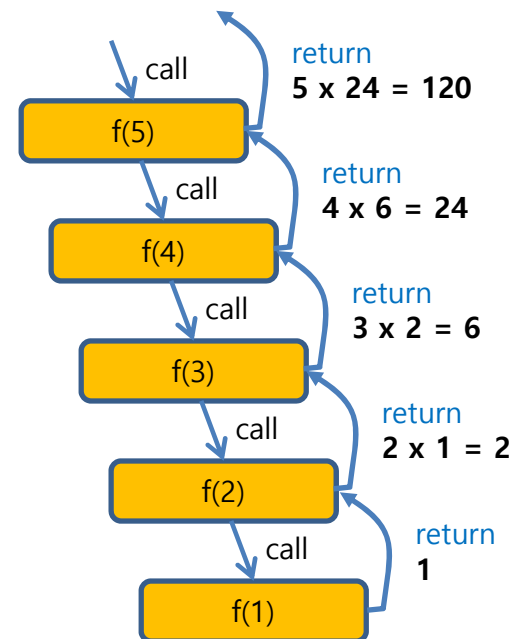
```
int f(int n)
{
    if (n <= 1) /* base case */
        return 1;
    else /* recursive step */
        return n*f(n-1);
}
```

```
int main()
{
    f(5);
}
```

```
int f(int n)
{
    int i, fact = 1;

    for(i=1; i<=n; i++)
        fact *= i;

    return fact;
}
```



Call Stack

- Function call stack

- 필요성:

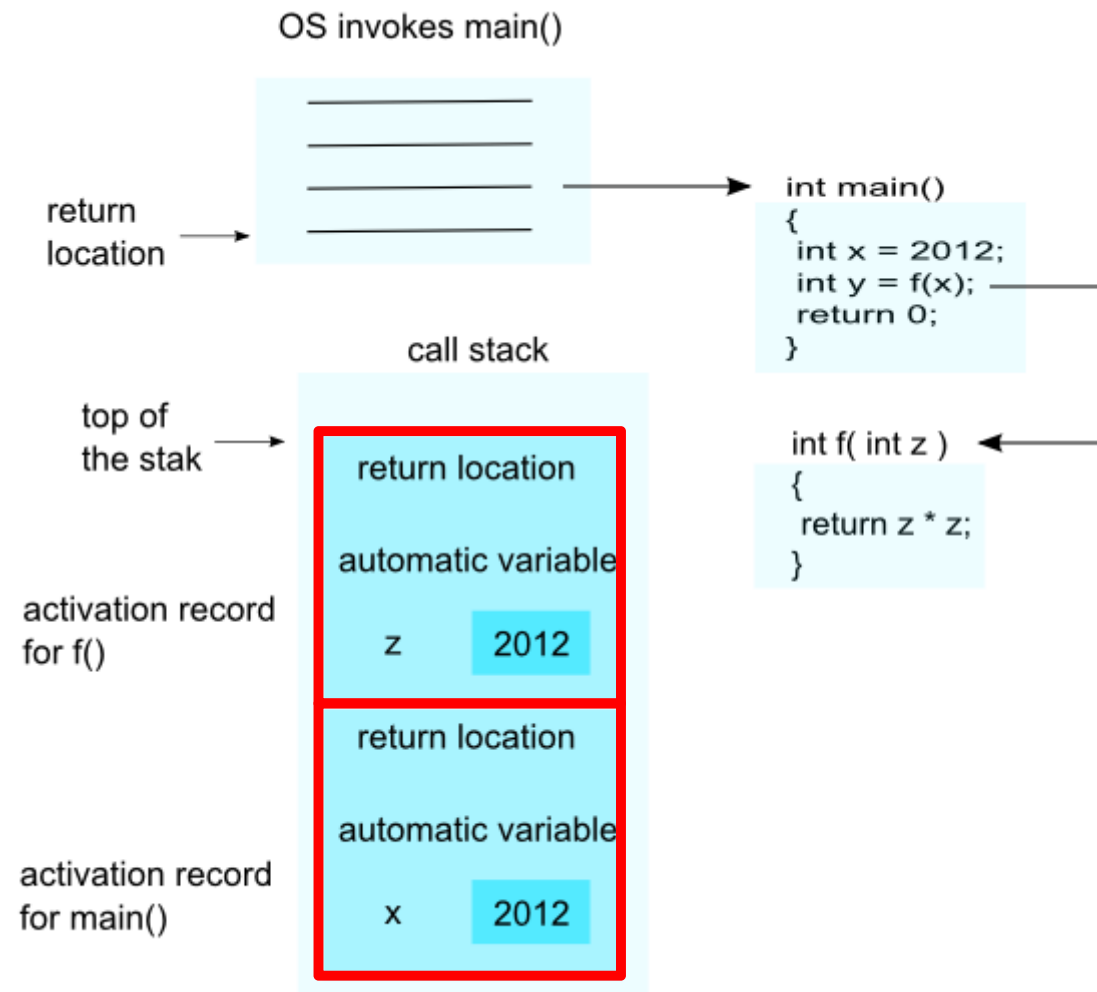
- 프로그램 실행 중 함수를 호출 할 때, 지금 실행중인 함수는 지역변수 등과 같은 자신이 가지고 있는 정보를 저장할 필요가 있다.
 - 새로운 함수 수행이 완료되면 다시 원래 호출했던 함수로 제어가 돌아와야 하고, 그 함수에서 사용하던 지역변수등과 같은 정보를 복원해서 사용해야 한다.
 - 또한 원래 함수로 되돌아 왔을 때 수행해야 할 프로그램의 주소(반환주소, return address) 도 저장해야 한다.
 - 현재함수가 가지고 있던 정보를 저장하는 연속적인 메모리 공간을 "activation record"라 하고, 이 activation record 를 "call stack"이라는 공간에서 관리함.

- Why stack?

- 함수를 호출하고 반환하는 과정이 "last call first return" 이므로 함수의 activation record를 LIFO(last in first out) 구조인 stack 에 저장하는 것이 가장 적합함.

Call Stack

- Function call stack

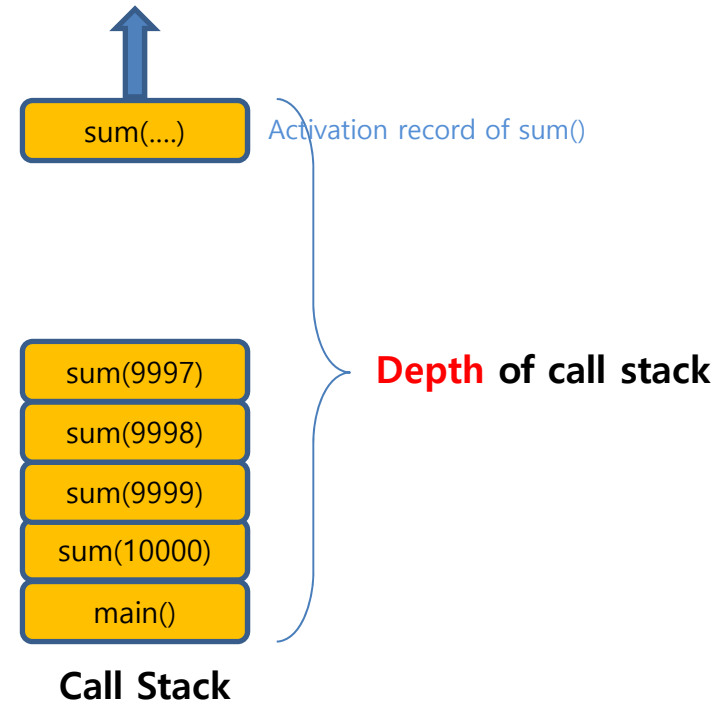


Stack Overflow

```
int sum(int n)
{
    if (n == 1) /* base case */
        return 1;
    else /* recursive step */
        return sum(n-1) + n;
}

int main()
{
    sum(10000);
}
```

Maximum Depth of call stack : 10000



Call Stack Overflow:
Shortage of memory for a call stack
caused by too many calls for recursion.

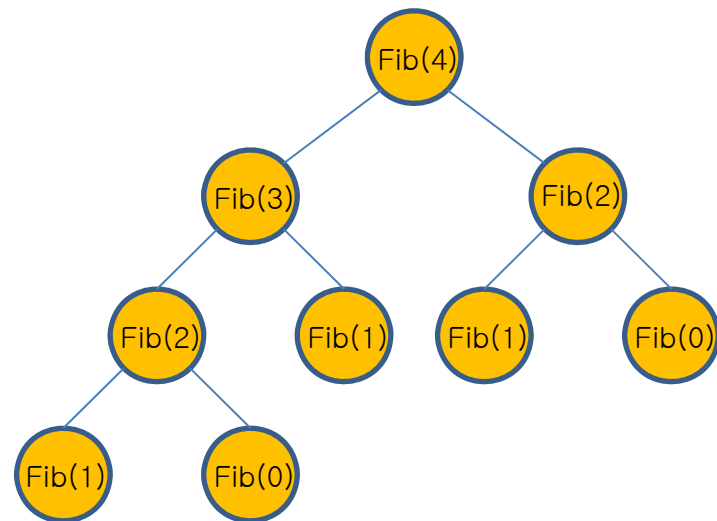
Fibonacci

- Fibonacci (recursive)

$$Fib(n) = \begin{cases} 0 & n = 0 \quad (\text{base case}) \\ 1 & n = 1 \quad (\text{base case}) \\ Fib(n-1) + Fib(n-2) & n > 1 \quad (\text{recursive step}) \end{cases}$$

```
int fib(int n)
{
    if (n <= 1) /* base case */
        return n;
    else /* recursive step */
        return fib(n-1)+fib(n-2);
}

int main()
{
    fib(4);
}
```



Maximum Depth of fib(n)? Call stack overflow?

Linear Sum

- Linear Sum (recursive)

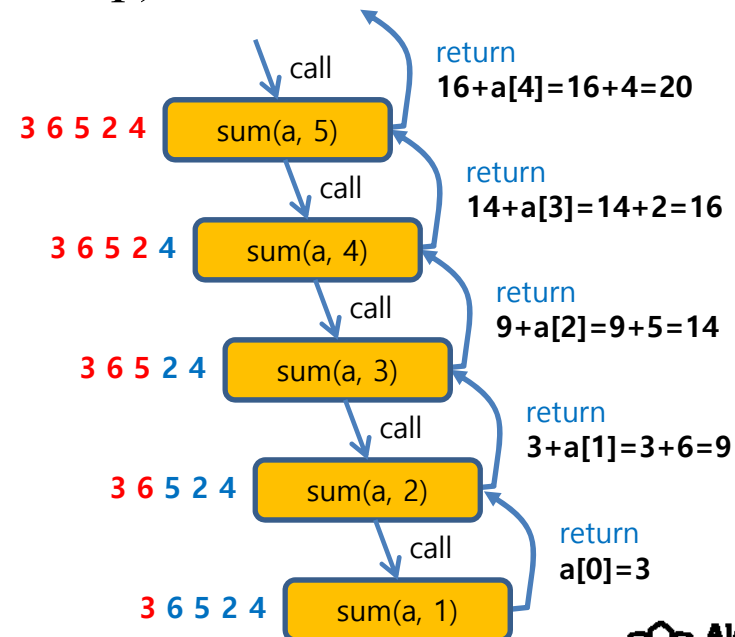
- n 개의 정수 a_1, a_2, \dots, a_n 이 주어졌을 때, 그 정수의 합

- $S(n) = \sum_{k=1}^n a_k$ 을 계산하시오.

$$S(n) = \begin{cases} a_1 & n = 1 \quad (\text{base case}) \\ S(n-1) + a_n & n > 1 \quad (\text{recursive step}) \end{cases}$$

```
int sum(int a[], int n)
{
    if (n == 1) /* base case */
        return a[0];
    else /* recursive step */
        return sum(a, n-1) + a[n-1];
}

int main()
{
    int a[5] = {3, 6, 5, 2, 4};
    sum(a, 5);
}
```



Reversing Array

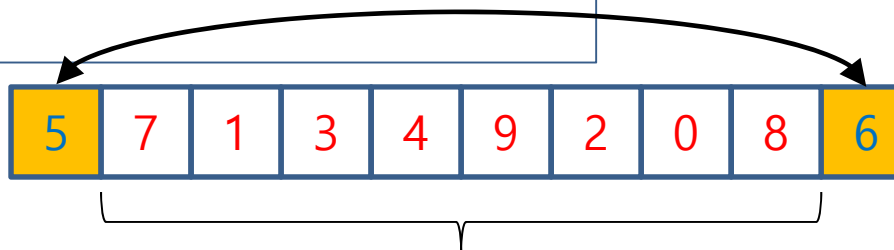
- Reversing Array (recursive)

$$[a_1, a_2, a_3, \dots, a_{n-1}, a_n] \Rightarrow [a_n, a_{n-1}, \dots, a_3, a_2, a_1]$$

```
void reverseArray(int a[], int i, int j)
{
    /* base case ?? */
    if (i < j)
    {
        swap(a, i, j); /*swap a[i] and a[j] */
        reverseArray(a, i+1, j-1);
    }
}
```

```
int main()
{
    int a[] = {5, 7, 1, 3, 4, 9, 2, 0, 8, 6};
    reverseArray(a, 0, 4);
}
```

Maximum Depth of reverseArray()?
Call stack overflow?



reverseArray(a, 1, 8)

Computing Powers

- Computing Powers (recursive)

$$p(x, n) = x^n$$

$$p(x, n) = \begin{cases} 1 & n = 0 & \text{(base case)} \\ x \cdot p(x, n-1) & n > 0 & \text{(recursive step)} \end{cases}$$

```
double p(double x, int n)
{
    if (n == 0) /* base case */
        return 1.0;
    else /* recursive step */
        return x * p(x, n-1);
}
```

Analysis:

- base operation : multiplication
- O(n)

Fast Computing Powers

- Fast Computing Powers (recursive)
 - double squaring

$$x^{2^n} = \left(\dots \left((x^2)^2 \right)^2 \dots \right)^2$$

$$2^2 = 2^{2^1} = 4$$

$$2^4 = 2^{2^2} = (2^2)^2 = 16$$

$$2^8 = 2^{2^3} = \left((2^2)^2 \right)^2 = 256$$

$$2^{16} = 2^{2^4} = \left(\left((2^2)^2 \right)^2 \right)^2 = 65536$$

Fast Computing Powers

- Fast Computing Powers (recursive)

$$p(x, n) = x^n$$

$$p(x, n) = \begin{cases} 1 & n = 0 & \text{(base case)} \\ x \cdot p(x, (n-1)/2)^2 & \text{if } n > 0 \text{ is odd} & \text{(recursive step)} \\ p(x, n/2)^2 & \text{if } n > 0 \text{ is even} & \text{(recursive step)} \end{cases}$$

$$2^4 = (2^{(4/2)})^2 = (2^2)^2 = 4^2 = 16$$

$$2^5 = 2 \cdot (2^{(4/2)})^2 = 2 \cdot (2^2)^2 = 2 \cdot 4^2 = 32$$

$$2^6 = (2^{(6/2)})^2 = (2^3)^2 = 8^2 = 64$$

$$2^7 = 2 \cdot (2^{(6/2)})^2 = 2 \cdot (2^3)^2 = 2 \cdot 8^2 = 128$$

Fast Computing Powers

- Fast Computing Powers (recursive)

```
double p(double x, int n)
{
    double y;

    if (n == 0)                /* base case */
        return 1.0;
    else if (n%2 == 1){        /* n is odd */
        y = p(x, (n-1)/2);
        return x*y*y;
    }
    else {                    /* n is even */
        y = p(x, n/2);
        return y*y;
    }
}
```

Analysis:

- base operation : multiplication
- $O(\log n)$

최대공약수

- Greatest Common Divisor (최대공약수)
 - 유클리드 호제법
 - 두 정수 a, b 의 최대공약수 $\gcd(a, b)$ 를 구하고자 한다.
 - a 를 b 로 나눈 나머지를 r 이라고 하자.
 - a 와 b 의 최대공약수는 b 와 r 의 최대공약수와 같다.
 - 따라서, b 와 r 에 대하여 위 방법을 반복적으로 적용한다.
 - 위 방법을 반복적으로 적용하여 나머지가 0 되었을 때, 나누는 수가 a, b 의 최대공약수이다.
 - 예
 - $\gcd(1071, 1029) \Rightarrow 1071$ 을 1029 로 나눈 나머지 : 42
 - $\gcd(1029, 42) \Rightarrow 1029$ 를 42 로 나눈 나머지 : 21
 - $\gcd(42, 21) \Rightarrow 42$ 는 21 로 나누어 떨어지므로
 - 최대공약수는 21이다.

최대공약수 (2)

- Greatest Common Divisor (recursive)

$$\text{gcd}(a,b) = \begin{cases} a & b = 0 & \text{(base case)} \\ \text{gcd}(b, a \% b) & b > 0 & \text{(recursive step)} \end{cases}$$

```
int gcd(int a, int b)
{
    if (b == 0) /* base case */
        return a;
    else /* recursive step */
        return gcd(b, a % b);
}
```

```
void main(void)
{
    gcd(1071, 1029);
}
```

```
int gcd(int a, int b)
{
    int r;

    do
    {
        r = a % b;
        a = b;
        b = r;
    } while ( r != 0 )

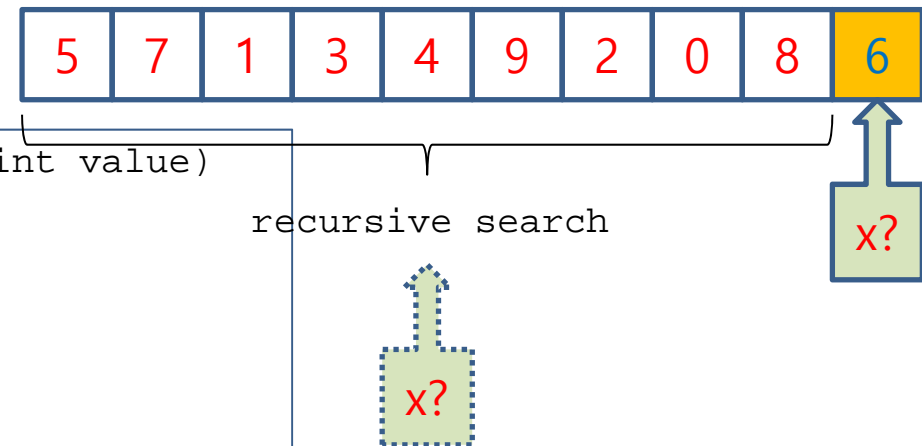
    return a;
}
```

Sequential Search

- Sequential Search (recursive)
 - 1차원 배열에 n 개의 정수가 주어졌을 때, 정수 x 가 배열에 있는지를 검사하시오 (순차검색)

```
int recLinearSearch(int a[], int n, int value)
{
    if (n <= 0)
        return -1;
    if (a[n-1] == value)
        return n-1;
    else
        return recLinearSearch(a, n-1, value);
}
```

```
int main()
{
    int a[] = {5, 7, 1, 3, 4, 9, 2, 0, 8, 6};
    recLinearSearch (a, 10, 4);
}
```



교내 프로그래밍 경시대회

- 일시 : 2017. 09. 23 (토) 오후 1:00 ~ 6:00
12:00 부터 점심 제공
알고리즘 수강학생은 필수로 참석해야 함.
- 참가신청방법 : 추후 발표
- 사용언어 : C/C++/Python/Java (단, 표준만 가능함)
- 3인/팀 참가, 국민대학교 재학/휴학생 모두 참가 가능
- 팀별로 컴퓨터 1대 사용 (팀별 준비)
- 인터넷 사용 불가, 책 사용 불가,
- 개인 노트(최대 30페이지) 소지 가능
- 상품 (2016년도 기준) :
 - 대상 (30만원상당CJ상품권) 1팀
 - 금상 1팀, 은상 1팀, 동상 2팀
 - 특별상 (비전공자)
 - 기타 다수의 특별상 있음

Count Up / Count Down

- Count Up / Count Down (recursive)

```
int countUp(int nCount)
{
    if (nCount > 0)
        countUp(nCount-1);

    printf("%d \n", nCount);
}
```

```
int countDown(int nCount)
{
    printf("%d \n", nCount);

    if (nCount > 0)
        countDown(nCount-1);
}
```

```
void main(void)
{
    countUp(4);
    countDown(4);
}
```

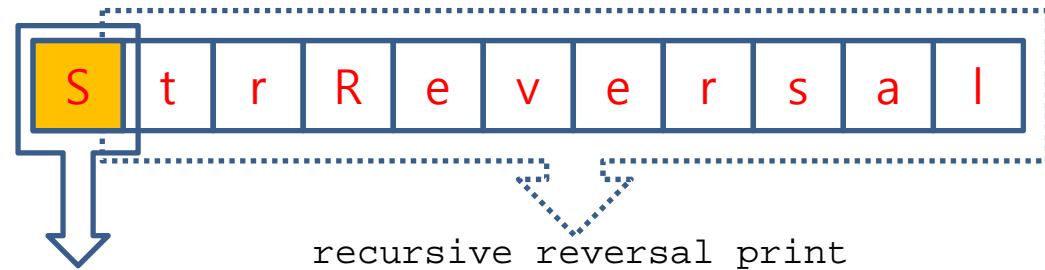
Count Up Count Down

0
1
2
3
4

4
3
2
1
0

String Reversal

- 문자열 뒤집어 출력하기 (recursive)
 - "StrReversal" -> "lasreveRrtS"

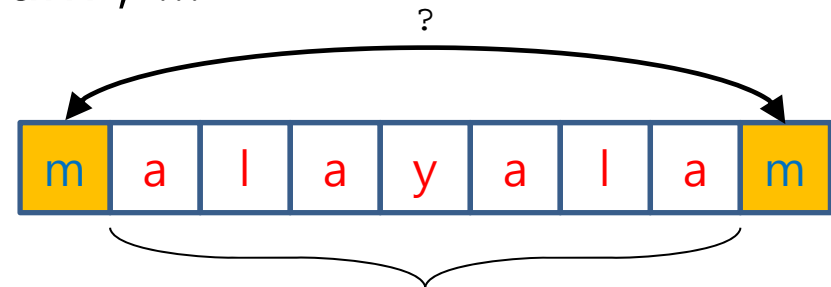


```
void printStrReversal(char str[], int index)
{
    if (str[index] != NULL)
    {
        printStrReversal(str, index+1);
        printf("%c", str[index]);
    }
}
```

```
void main(void)
{
    char str[] = "StrReversal";
    printStrReversal(str, 0);
}
```

팰린드롬 (Palindrome) 검사

- Palindrome (회문) 검사하기 (recursive)
 - “기러기”, “madam”, “malayalam”, ...



Recursive check

```
int checkPalindrome(char str[], int first, int last)
{
    if (last <= first)
        return 1;
    else if (str[first] != str[last])
        return 0;
    else
        return checkPalindrome(str, first+1, last-1);
}
```

```
void main(void)
{
    char line[256] = "malayalam";
    printf("%d \n", checkPalindrome(line, 0, strlen(line)-1));
}
```

십진수 진수변환 출력하기

- 십진수 진수변환 출력하기 (recursive)
 - 주어진 십진수 정수를 다른 진수의 숫자로 변환하여 출력한다. 변환되는 진수의 기수(base)는 2~16 의 범위를 가진다.

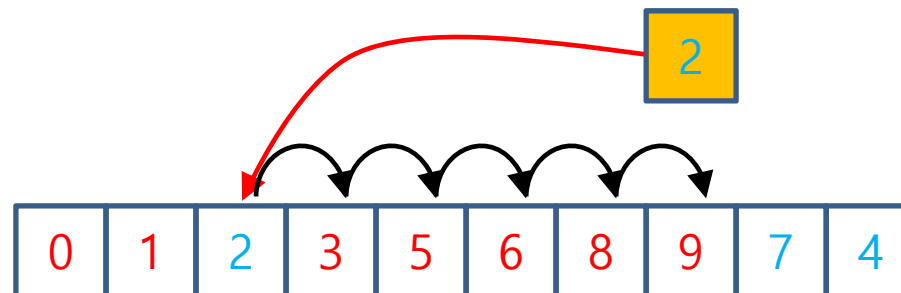
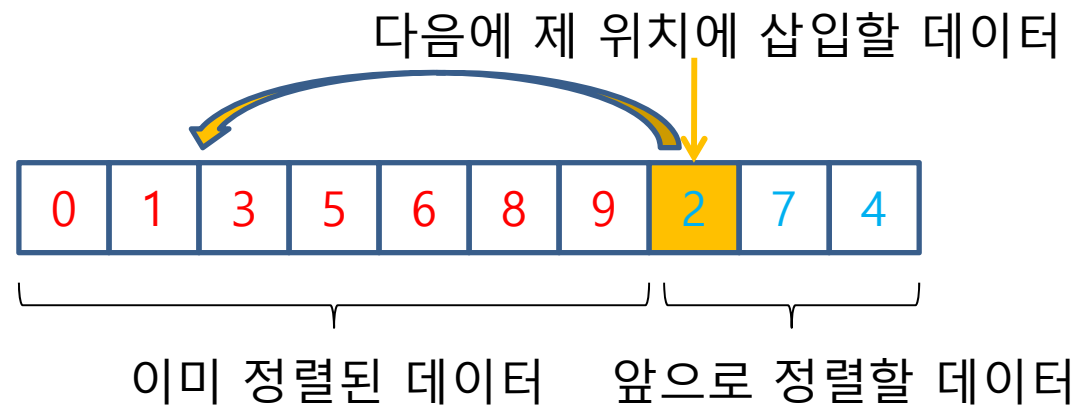
```
void baseConversion(int n, int base)
{
    static baseTable[] = "0123456789abcdef";

    if (n >= base)
        baseConversion(n/base, base);
    printf("%c\n", baseTable[n%base]);
}
```

```
void main(void)
{
    int num = 1234567;
    baseConversion(num, 16);
}
```

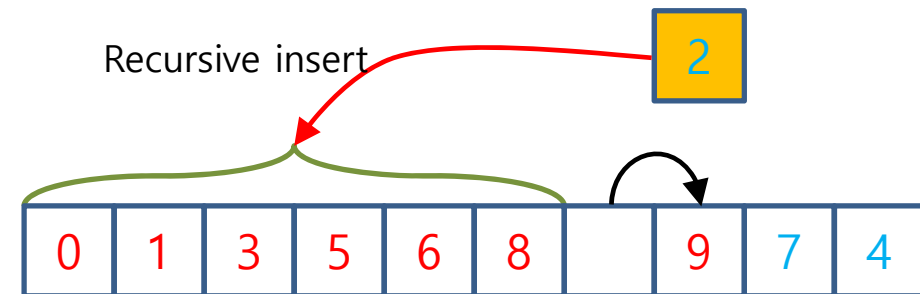
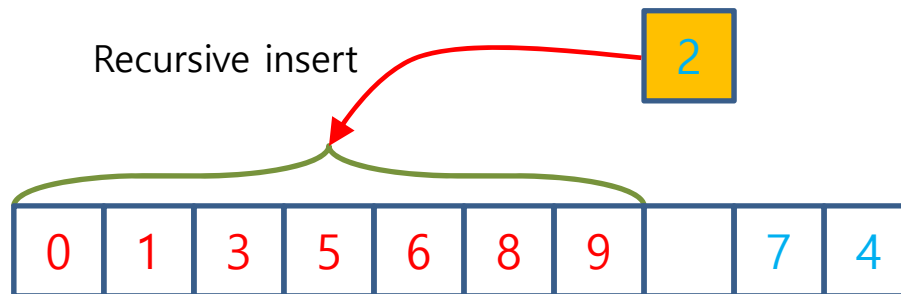
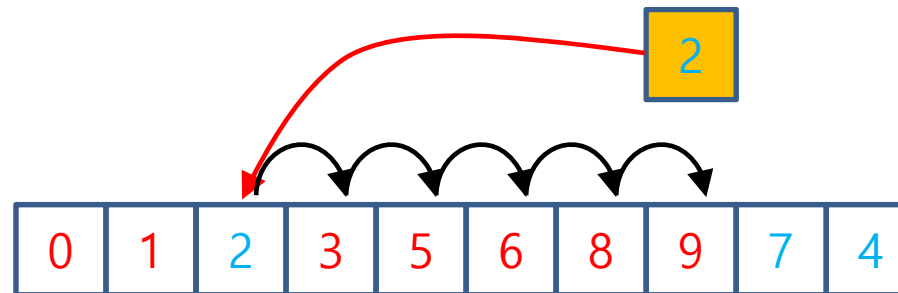

Recursive Insertion Sorting

- Insertion Sorting (recursive)
 - recursive insert



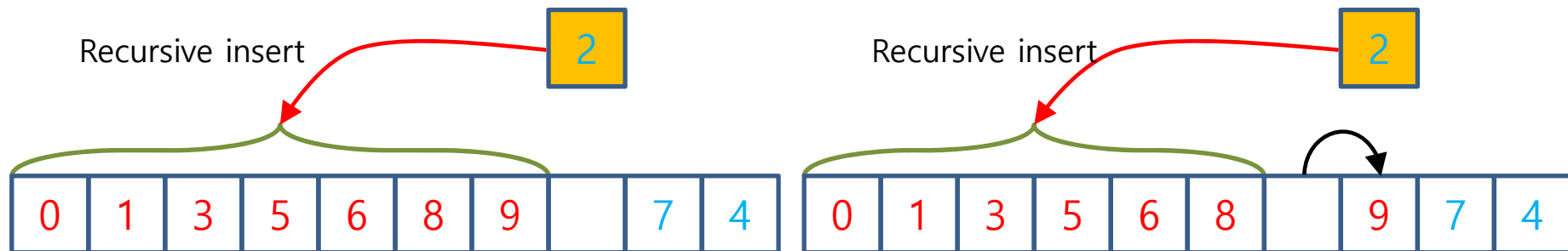
Recursive Insertion Sorting (2)

– recursive insert



Recursive Insertion Sorting (2)

– recursive insert

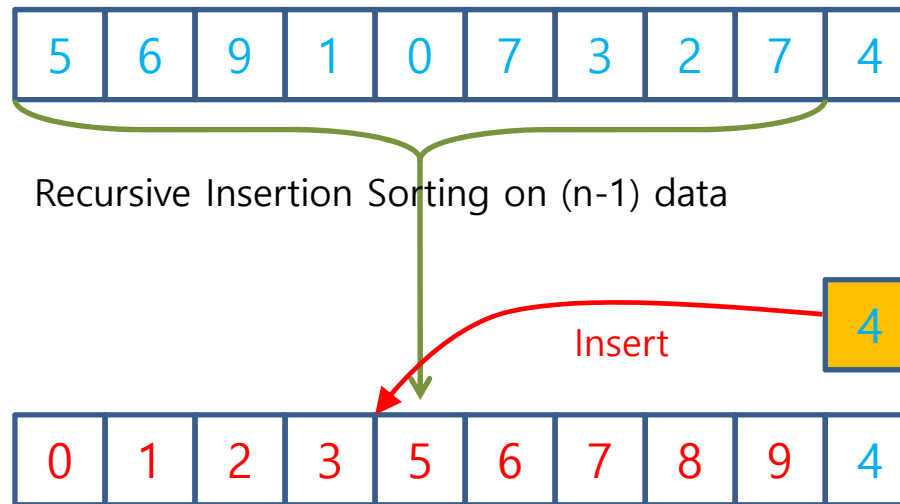


```
void insert(int data[], int size, int value)
{
    if (size == 0)
        data[size] = value;
    else
    {
        if (data[size-1] < value)
            data[size] = value;
        else
        {
            data[size] = data[size-1];
            insert(data, size-1, value);
        }
    }
}
```

Recursive Insertion Sorting (3)

– recursive insertion sorting

Recursive Insertion Sorting on n data



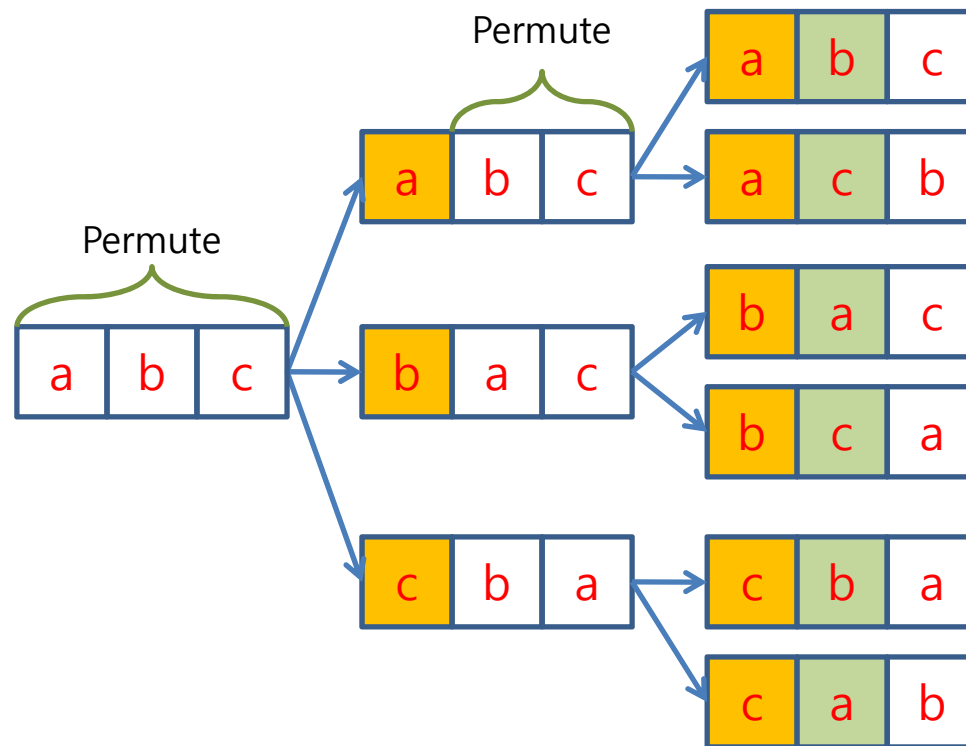
```
void insertionSort(int data[], int size)
{
    if (size == 0)
        return;
    else
    {
        insertionSort(data, size-1);
        insert(data, size-1, data[size-1]);
    }
}
```

Permutation 만들기

- Compute All Permutation (recursive)
 - n 개의 서로 다른 문자로 만들어진 스트링이 주어졌을 때, 이 문자열에 속하는 문자들의 모든 순열 (permutation)으로 만들어진 $n!$ 개의 문자열을 출력하시오.
 - 예: "abc"
 - "abc", "acb", "bac", "bca", "cab", "cba"

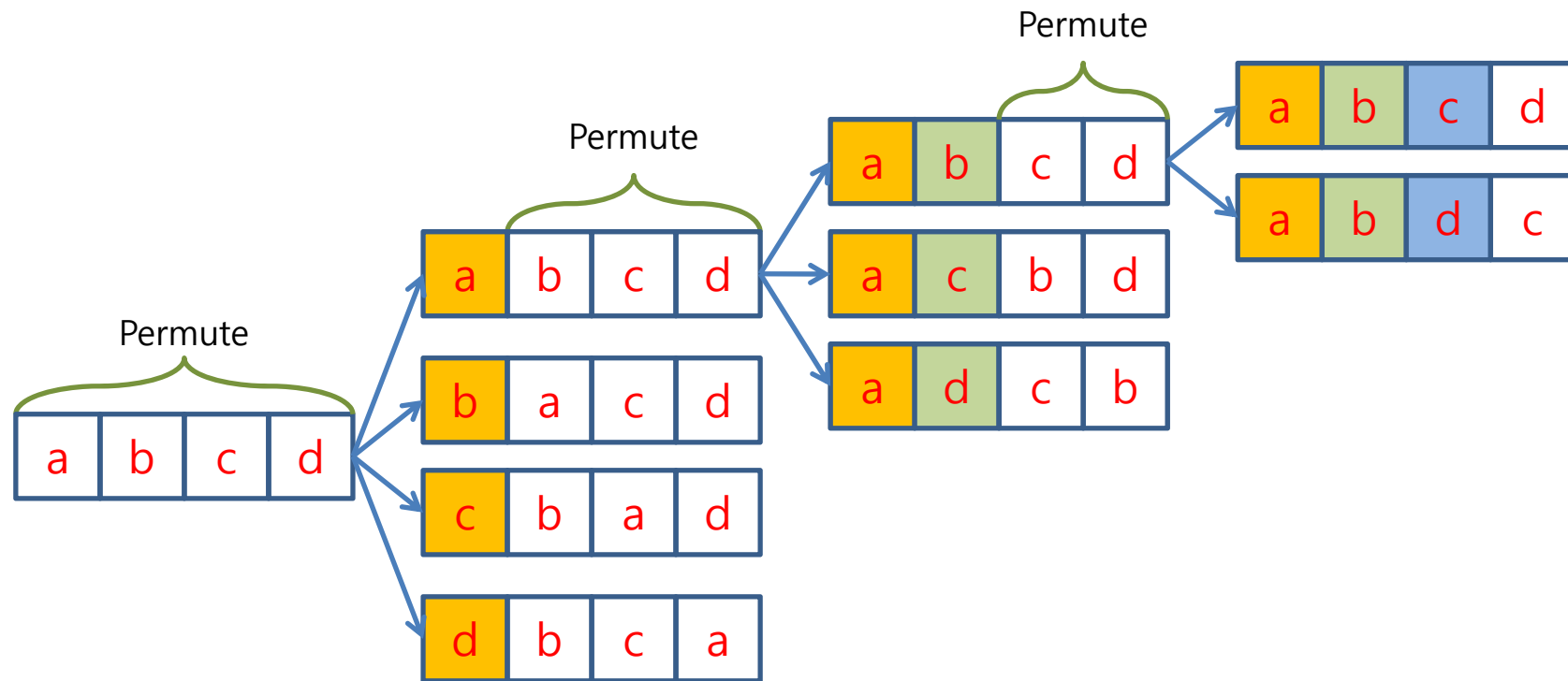
Permutation 만들기 (2)

- Compute All Permutation (recursive)



Permutation 만들기 (3)

- Compute All Permutation (recursive)



Permutation 만들기 (4)

- Compute All Permutation (recursive)

```
void permuteString(char *str, int begin, int end)
{
    int i;
    int range = end - begin;

    if(range == 1)
        printf("%s\n", str);
    else
    {
        for(i=0; i<range; i++)
        {
            swap(&str[begin], &str[begin+i]);
            permuteString(str, begin+1, end);
            swap(&str[begin], &str[begin+i]); /* recover */
        }
    }
}

void permute(char *str)
{
    permuteString(str, 0, strlen(str));
}
```

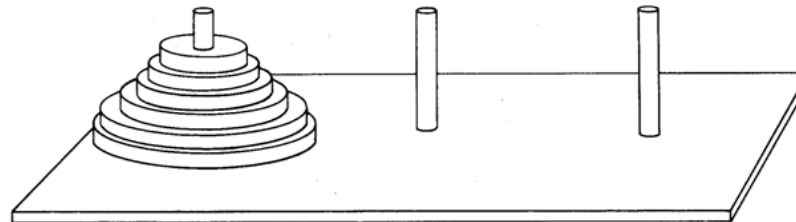
```
void main(void)
{
    int str[] = "abcd";
    permute(str)
}
```


Hanoi Tower

- Hanoi Tower

- 문제

- 아래 그림에서와 같이 세 개의 기둥과 이 기둥에 꽂을 수 있는 크기가 서로 다른 원판이 여러 개 있다.
 - 초기에는 모든 원판이 가장 큰 원판부터 가장 작은 원판까지 아래로부터 위로 가장 왼쪽 기둥에 꽂혀있다.

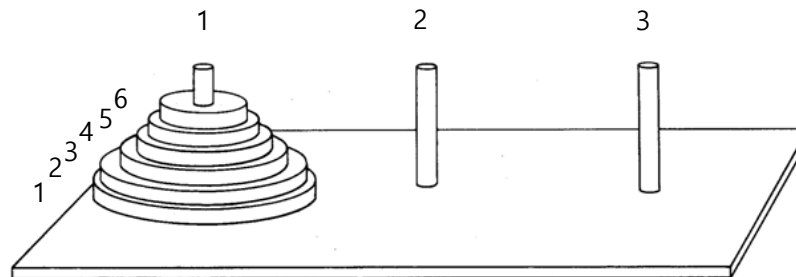


Hanoi Tower (2)

- Hanoi Tower

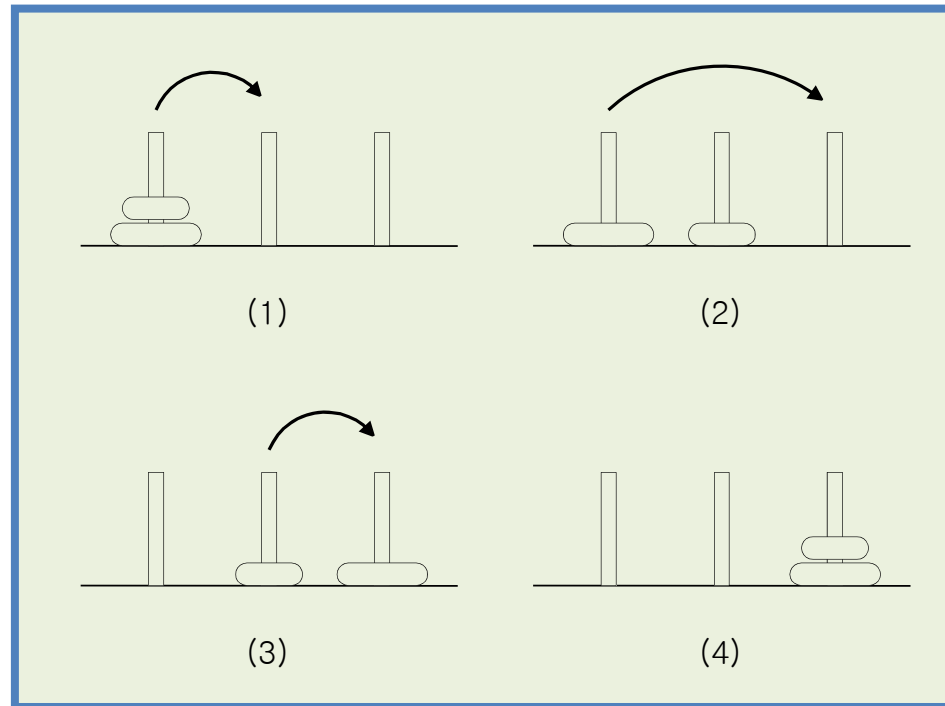
- 문제

- 다음 조건을 만족하면서, 가장 왼쪽에 꽂혀있는 모든 원판을 가장 오른쪽 기둥으로 옮기고자 한다.
 - 한 번에 한 개의 원판만 옮길 수 있다.
 - 한 개의 원판을 옮길 때는 어떤 기둥에 꽂혀있는 원판의 가장 위에 놓여져 있는 원판을 다른 기둥에 꽂혀있는 원판의 가장 위에 놓는다.
 - 크기가 큰 원판이 작은 원판 위에 놓여져서는 안된다.



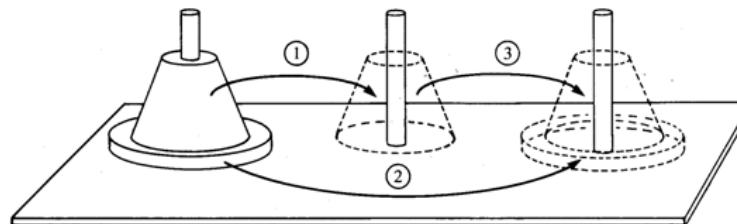
Hanoi Tower (3)

- Hanoi Tower
 - Example



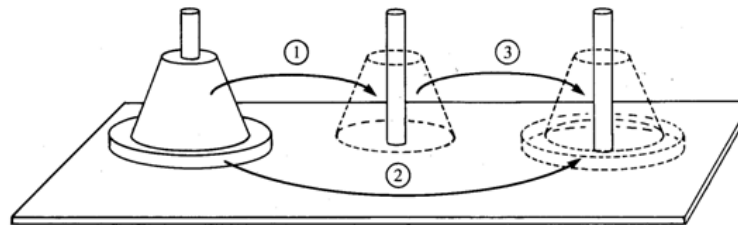
Hanoi Tower (4)

- Hanoi Tower Solution (recursive)
 - 아래 그림과 같이
 - (1) 1번 기둥에 쌓여져 있는 $(n-1)$ 개의 원판을 모두 2번 기둥으로 옮긴다.
 - (2) 1번 기둥에 남아 있는 가장 큰 원판을 3번 기둥으로 옮긴다.
 - (3) 2번 기둥으로 옮겨진 $(n-1)$ 개의 원판을 모두 3번 기둥으로 옮긴다.



Hanoi Tower (5)

- Hanoi Tower Solution (recursive)



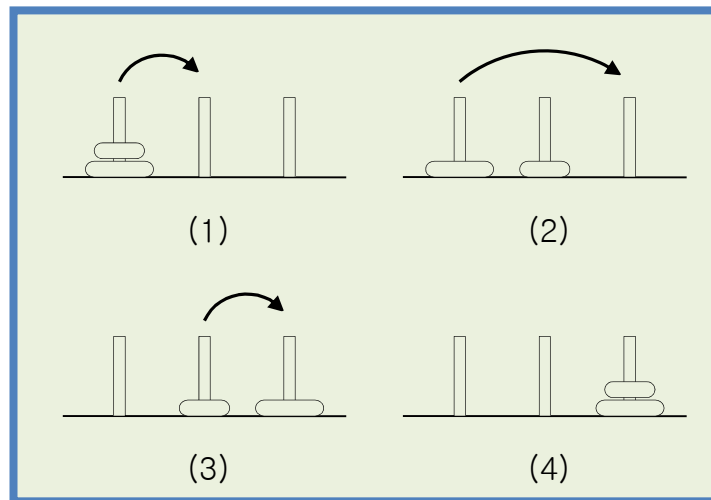
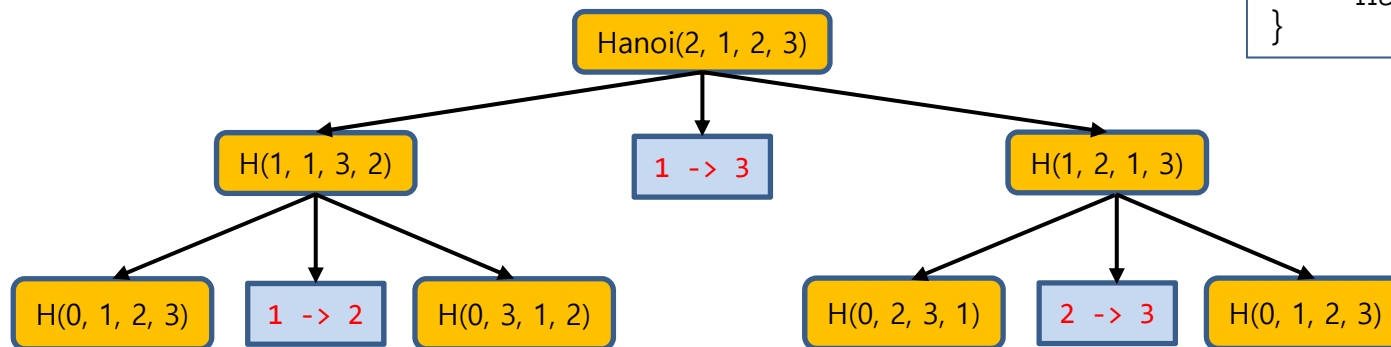
```
void Hanoi(int n, int a, int b, int c)
{
    if (n>0)
    {
        Hanoi(n-1, a, c, b);
        printf("Move disk from %d to %d.\n", a, c);
        Hanoi(n-1, b, a, c);
    }
}
```

```
void main(void)
{
    int numDisks = 4;
    printf("Number of disks to move: %d\n", numDisks);
    Hanoi(numDisks, 1, 2, 3);
}
```

Hanoi Tower (6)

- Example

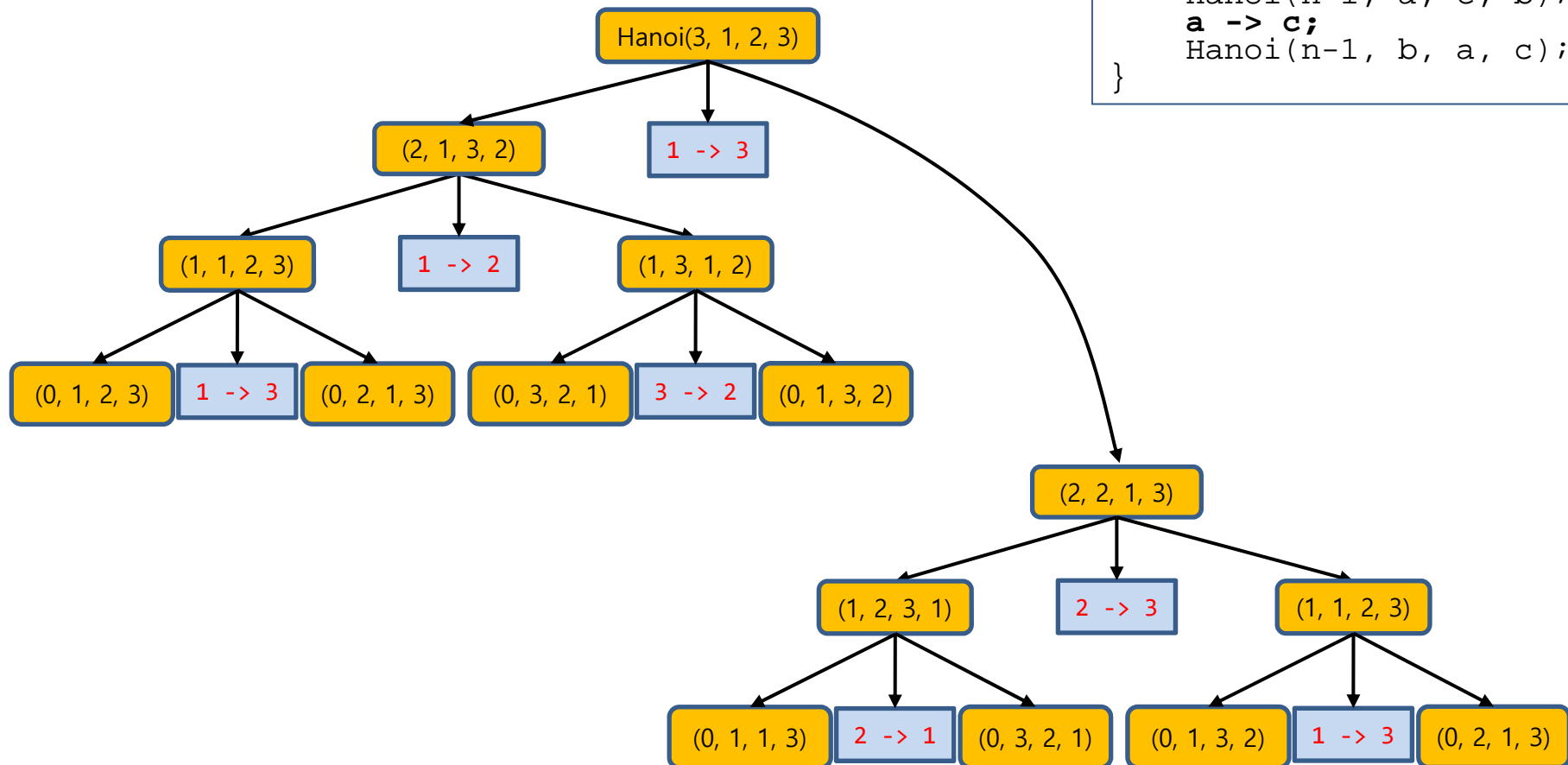
```
Hanoi(n, a, b, c)
{
    Hanoi(n-1, a, c, b);
    a -> c;
    Hanoi(n-1, b, a, c);
}
```



Hanoi Tower (7)

- Example

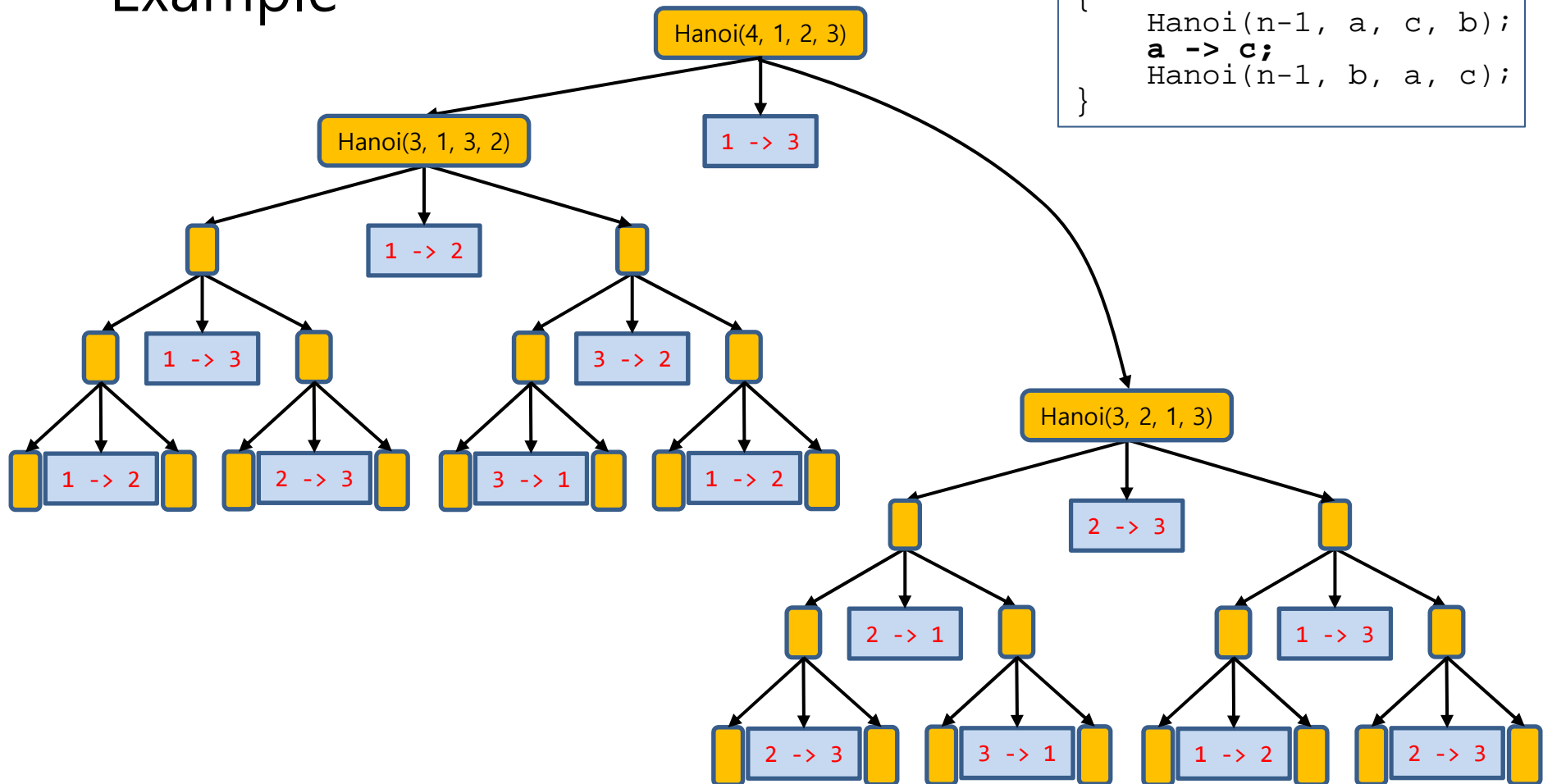
```
Hanoi(n, a, b, c)
{
    Hanoi(n-1, a, c, b);
    a -> c;
    Hanoi(n-1, b, a, c);
}
```



Hanoi Tower (8)

- Example

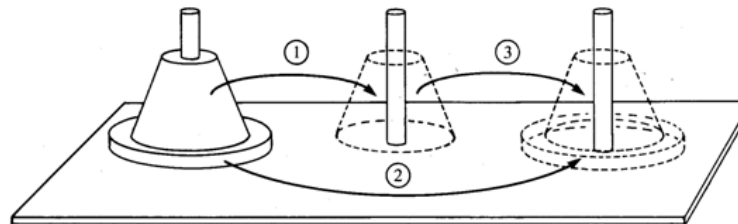
```
Hanoi(n, a, b, c)
{
    Hanoi(n-1, a, c, b);
    a -> c;
    Hanoi(n-1, b, a, c);
}
```



Hanoi Tower (9)

- Analysis of Hanoi Tower Solution (recursive)
 - $T(n)$
 - n 개의 원판이 주어졌을 때, 원판을 한 개씩 옮기는 총 회수

$$T(n) = \begin{cases} 1 & n = 1 & \text{(base case)} \\ 2T(n-1) + 1 & n > 1 & \text{(recursive step)} \end{cases}$$



Hanoi Tower (10)

- Analysis of Hanoi Tower Solution (recursive)

- $T(n)$

- n 개의 원판이 주어졌을 때, 원판을 한 개씩 옮기는 총 회수

$$T(n) = \begin{cases} 1 & n = 1 & \text{(base case)} \\ 2T(n-1) + 1 & n > 1 & \text{(recursive step)} \end{cases}$$

$$T(n) = 2T(n-1) + 1$$

$$= 2\{2T(n-2) + 1\} + 1 = 2^2T(n-2) + 2 + 1$$

$$= 2^2\{2T(n-3) + 1\} + 2 + 1 = 2^3T(n-3) + 2^2 + 2 + 1$$

...

$$= 2^{n-1}T(1) + 2^{n-2} + \dots + 2^2 + 2 + 1$$

$$= 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2 + 1$$

$$= 2^n - 1$$

원래 이 문제의 $n=64$ 이었다고 한다.
그러면, 모든 원판을 옮기는데 어느 정도의
시간이 걸릴까?

$$2^{64} - 1 = 18,446,744,073,709,551,615$$

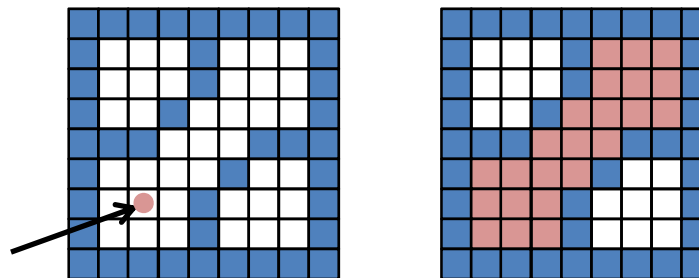
Flood Fill

- Flood Fill

- 문제

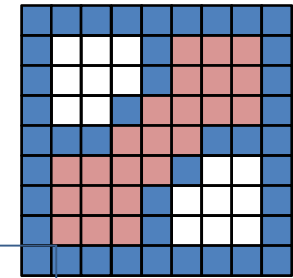
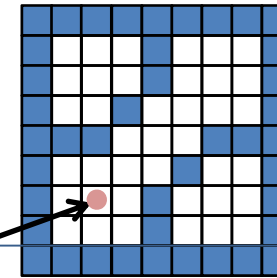
- 아래 그림에서와 같이 포토샵과 같은 그림 프로그램에서 마우스로 클릭된 픽셀의 색깔과 같은 색깔을 가지면서 클릭된 픽셀에 같은 색깔로 연속적으로 연결된 모든 픽셀을 주어진 다른 색깔로 바꾸는 문제이다.

- 여기에서 두 픽셀이 연결된 경우는 서로 아래/위 혹은 좌/우로 인접한 경우를 말한다.



Flood Fill (1)

- Flood Fill (recursive)



```
#define SIZE 9

void floodFill(char fig[][SIZE+1], int x, int y, char oldColor,
               char newColor)
{
    if (fig[x][y] != oldColor)
        return;
    else
    {
        fig[x][y] = newColor;

        floodFill(fig, x-1, y, oldColor, newColor);
        floodFill(fig, x+1, y, oldColor, newColor);
        floodFill(fig, x, y-1, oldColor, newColor);
        floodFill(fig, x, y+1, oldColor, newColor);
    }
}
```

Case of Multiple Recursion

Stack Overflow? Yes
Alternative Method:
Line Scan Algorithm (iterative)
(Computer Graphics)

```
void main(void)
{
    int i;
    char fig[SIZE][SIZE+1] = {
        "*****", "*.*.*.*.*", "*.*.*.*.*", "*.*.*.*.*", "*****",
        "*****", "*****", "*****", "*****", "*****";
    };

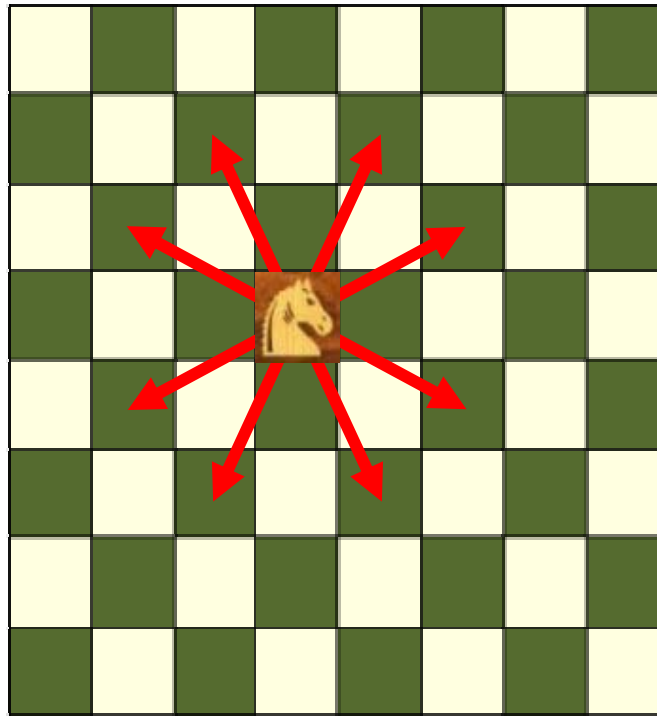
    floodFill(fig, 4, 4, '.', '+');
}
```

Knight's Tour Problem

- Knight's Tour

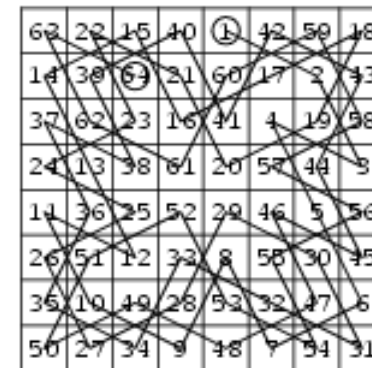
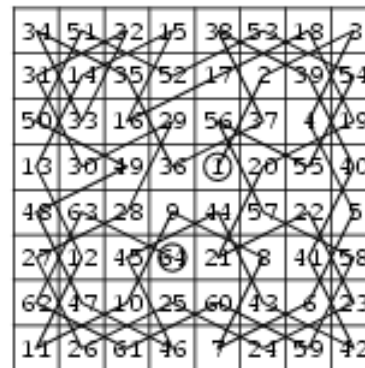
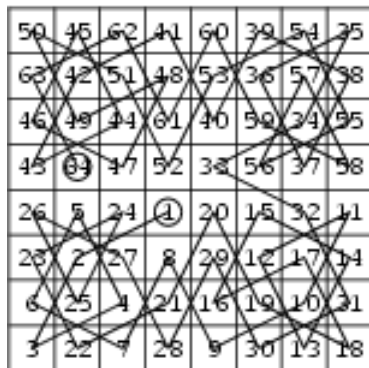
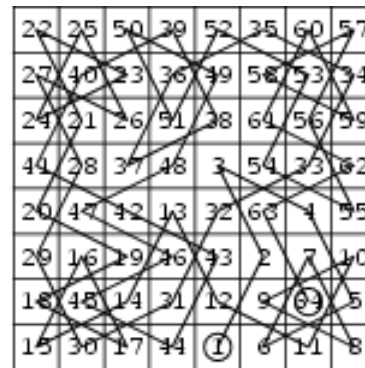
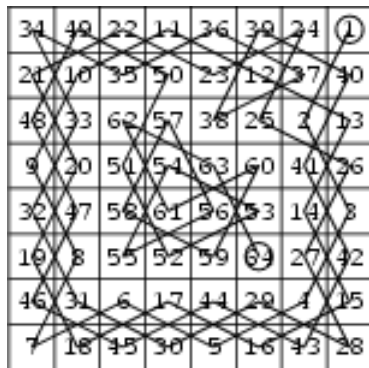
- 문제

- 체스판에서 기사(Knight) 말의 움직임은 아래 그림과 같다.
 - 임의의 위치에 놓여진 기사를 움직여서 모든 64개의 격자를 모두 방문하도록 기사말을 옮기는 방법을 계산하시오. 단, 기사가 이미 방문한 격자는 다시 방문하지 않는다.



Knight's Tour Problem (2)

- Knight's Tour
 - 예



Knight's Tour Problem (3)

- Knight's Tour Solution (recursive)

```
#define MAXSIZE 9

#define MARK 1
#define UNMARK 0

typedef struct Point {int x, y;} point;
point direction[8] = {{1, -2}, {2, -1}, {2, 1}, {1, 2},
                     {-1, 2}, {-2, 1}, {-2, -1}, {-1, -2}};
int board[MAXSIZE][MAXSIZE], path[MAXSIZE][MAXSIZE];

int knightTour (int m, int n, point pos, int counter)
{
    int i;
    point next;

    if (counter == m * n)
        return 1;

    for (i=0; i<8; i++)
    {
        ←
    }

    return 0;
}
```

Case of Multiple Recursion

Homework:

Convert the recursive algorithm to iterative algorithm

```
{
    next.x = pos.x + direction[i].x;
    next.y = pos.y + direction[i].y;

    if ( next.x > 0 && next.x <= n &&
        next.y > 0 && next.y <= m &&
        board[next.y][next.x] != MARK )
    {
        board[next.y][next.x] = MARK;
        path[next.y][next.x] = counter+1;

        if ( knightTour(m, n, next, counter+1) )
            return 1;

        board[next.y][next.x] = UNMARK;
    }
}
```

Maximum Depth of KnightTour()?
Call stack overflow?

Knight's Tour Problem (4)

- Knight's Tour Solution (recursive)

```
void main ( void )
{
    int i, j, m, n;
    point start;

    m = 6; n = 8;
    start.y = 3; start.x = 4;

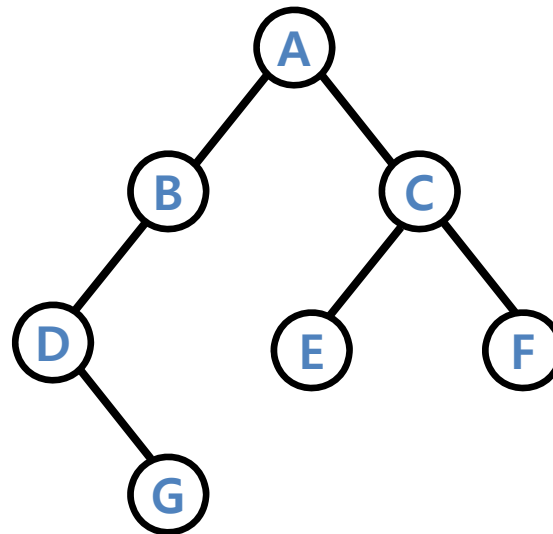
    for (i=1; i<=m; i++)
        for (j=1; j<n; j++)
            board[i][j] = UNMARK;

    board[start.y][start.x] = MARK;
    path[start.y][start.x] = 1;

    if ( knightTour(m, n, start, 1) )
        printTour(m, n);
}
```

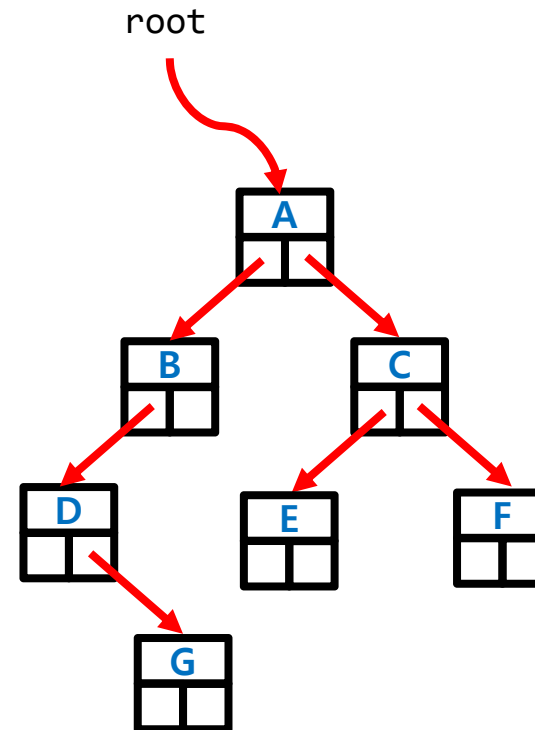
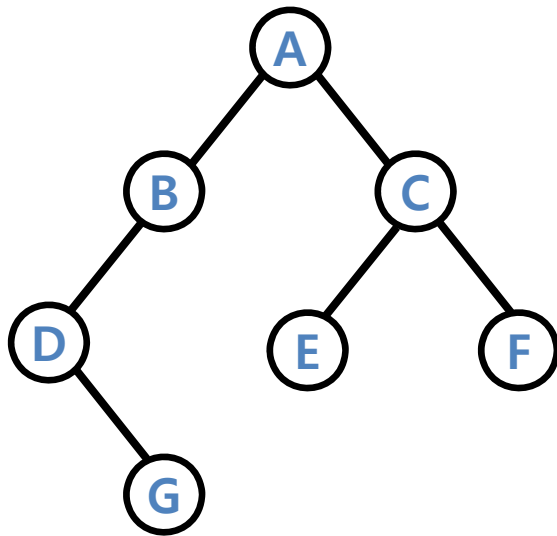

Binary Tree

- Definition of Binary Tree (recursively)
 - Binary Tree
 - Empty, or (base case)
 - Consists of a **root node** together with **left and right subtrees**, both of which are **binary trees** (recursive step)



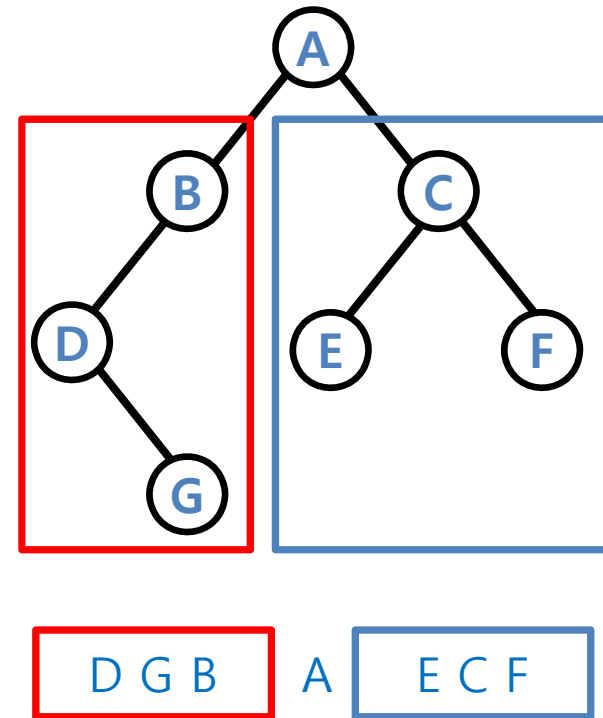
Binary Tree

```
struct node {  
    char data; /* int data */  
    struct node* leftSubTree;  
    struct node* rightSubTree;  
}
```



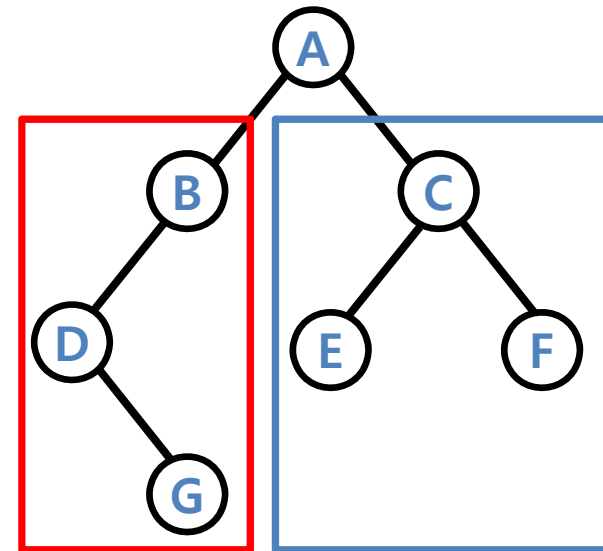
InOrder

```
void inOrder( node *root )
{
    if (root == NULL)    /* base case */
        return;
    else                  /* recursive step */
    {
        inOrder( root->leftSubtree );
        printf("%c ", root->data);
        inOrder( root->rightSubtree );
    }
}
```



PreOrder

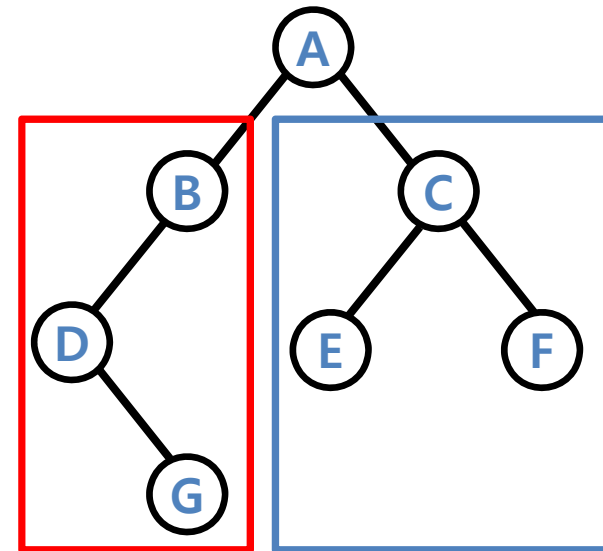
```
void preOrder( node *root )
{
    if (root == NULL)    /* base case */
        return;
    else                  /* recursive step */
    {
        printf("%c ", root->data);
        preOrder( root->leftSubtree );
        preOrder( root->rightSubtree );
    }
}
```



A **B D G** C E F

PostOrder

```
void postOrder( node *root )
{
    if (root == NULL)    /* base case */
        return;
    else                  /* recursive step */
    {
        postOrder( root->leftSubtree );
        postOrder( root->rightSubtree );
        printf("%c ", root->data);
    }
}
```



G D B

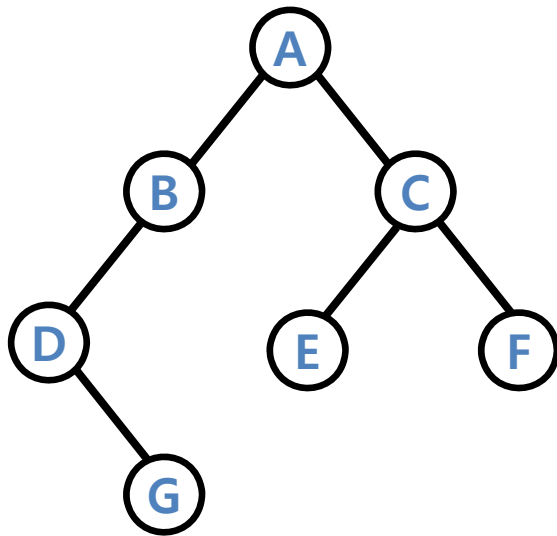
E F C

A

Question

- size() (recursive)

```
int size(struct node *root)
{
}
}
```

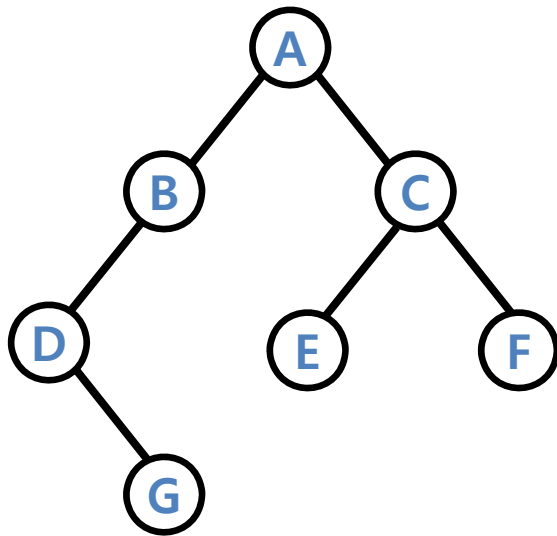


7

Question

- height() (recursive)

```
int height(struct node *root)
{
}
}
```

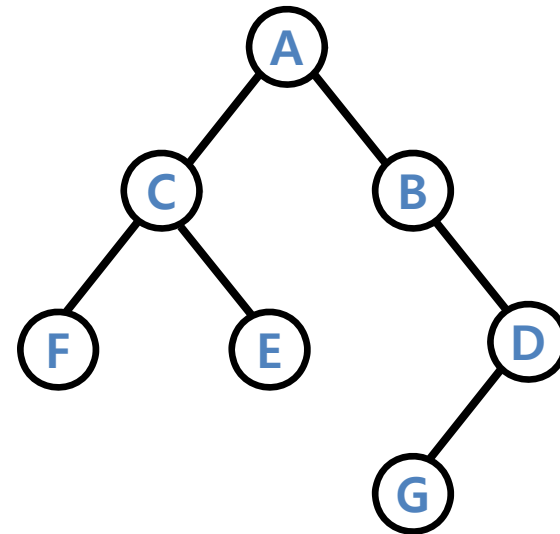
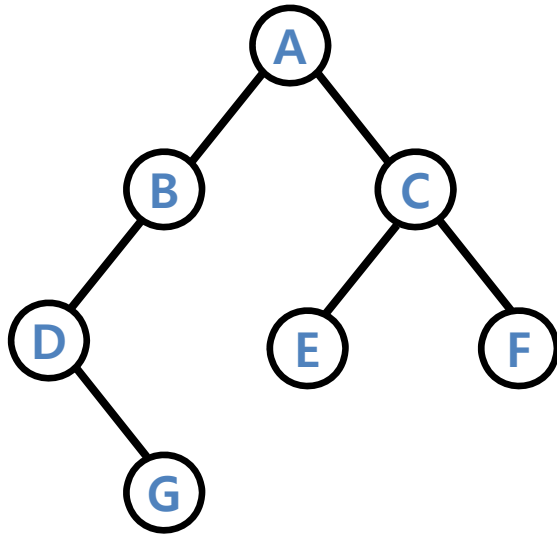


3

Question

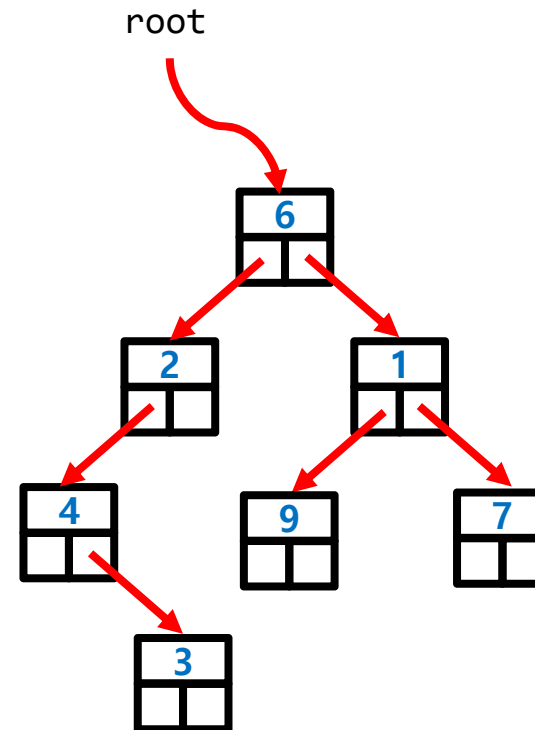
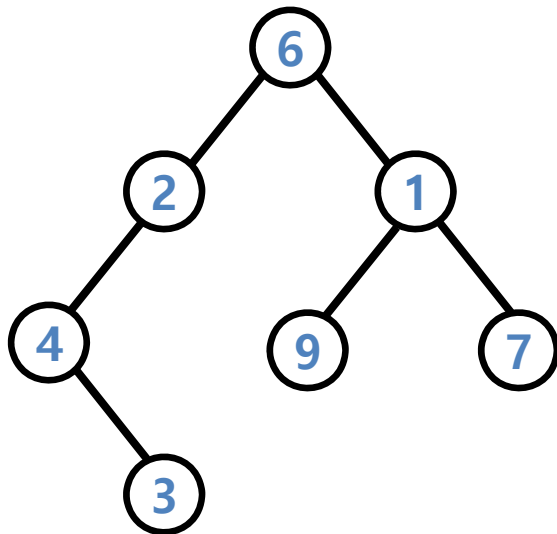
- mirror() (recursive)

```
void mirror(struct node *root)
{
}
}
```



Binary Tree with Weights

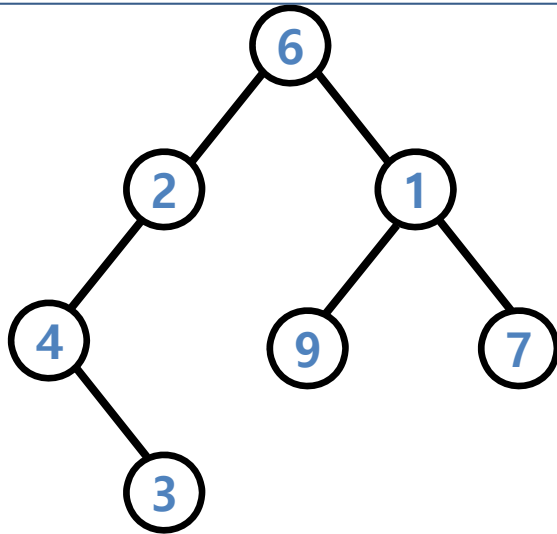
```
struct node {  
    int weight;  
    struct node* leftSubTree;  
    struct node* rightSubTree;  
}
```



Binary Tree with Weights

- sumOfWeight()

```
int sumOfWeight(struct node *root)
{
    if(root == NULL)
        return 0;
    else
        return sumOfWeight(root->leftSubTree)+sumOfWeight(root->rightSubTree)+
            root->weight;
}
```

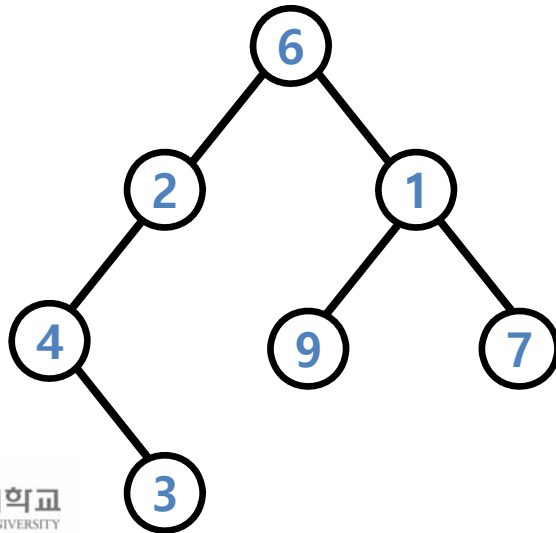


32

Binary Tree with Weights

- maxPathWeight()

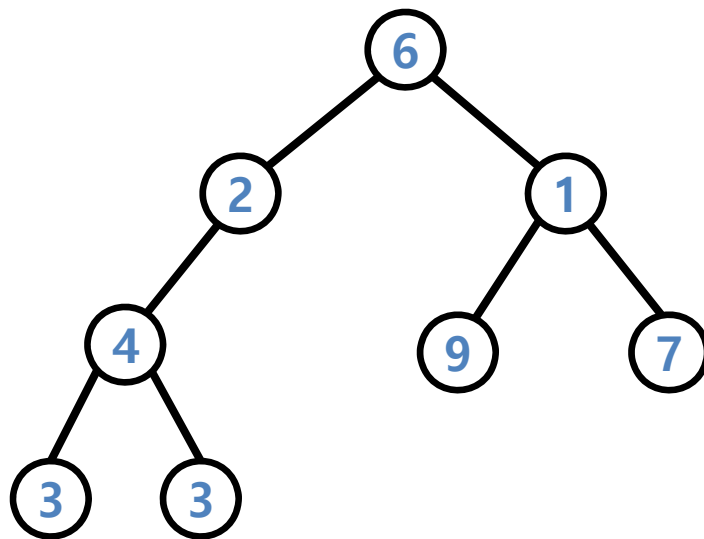
```
int maxPathWeight(struct node *root)
{
    if(root == NULL)
        return 0;
    else {
        int leftWeight, rightWeight;
        leftWeight = maxPathWeight(root->leftSubTree);
        rightWeight = maxPathWeight(root->rightSubTree);
        return root->weight + leftWeight >= rightWeight ? leftWeight : rightWeight;
    }
}
```



16

Perfect Binary Tree

- Perfect Binary Tree
 - a binary tree in which all interior nodes have two children *and* all leaves have the same *depth* or same *level*.
 - Sometimes, ambiguously called *complete* tree



Recursion (Review)

- Types of recursion
 - Linear recursion
 - a single recursive call for each recursion
 - examples
 - factorial, linear sum, reversing array, computing powers
 - Binary recursion
 - two recursive calls for each recursion
 - examples
 - Fibonacci numbers, Hanoi tower, merge sorting, quick sorting
 - Multiple recursion
 - more than two recursive calls for each recursion
 - example
 - flood fill, knight's tour