DATA
STRUCTURES
USING C

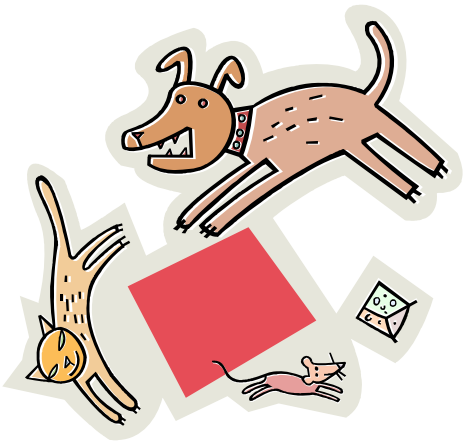
- 순환의 개념을 이해한다.
- 순환 알고리즘의 구조를 이해한다.
- 순환 알고리즘과 반복 알고리즘의 관계를 이해한다.
- 다양한 순환 호출 문제를 이해한다.
- 순환 호출의 응용 능력을 기른다.

CHAPTER 07

순환

순환(recursion), 재귀 호출

- 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법
  - 정의자체가 순환적으로 되어 있는 경우에 적합



순환의 예

- 팩토리얼 값 구하기 $$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$
- 피보나치 수열 $$fib(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$
- 이항계수 $${}_nC_k = \begin{cases} 1 & k = 0 \quad or \quad k = n \\ {}_{n-1}C_{k-1} + {}_{n-1}C_k & otherwise (0 < k < n) \end{cases}$$
- 하노이의 탑
- 이진탐색
- 영역채색 (Blob Coloring)

예: 팩토리얼 프로그래밍 #1

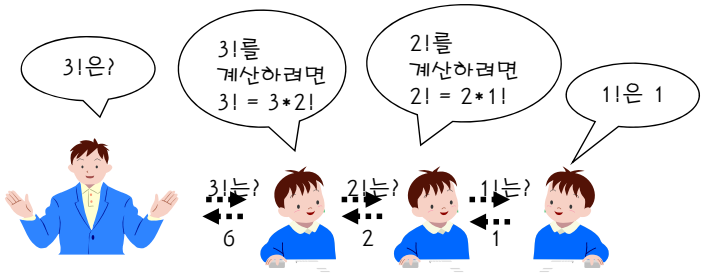
- 팩토리얼의 정의 $$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$
- 팩토리얼 프로그래밍 #1:
  - 정의대로 구현
  - (n-1)! 을 구하는 서브 함수 factorial_n_1를 따로 제작

```
int factorial(int n)
{
   if( n == 1 ) return 1;              // n==1인 경우(종료 조건)
   else return (n * factorial(n-1)); // n >1인 경우(순환 호출)
}
```

예: 팩토리얼 프로그래밍 #2

- 팩토리얼 프로그래밍 #2:
  - (n-1)! 팩토리얼을 현재 작성중인 함수를 다시 호출하여 계산 (순환 호출)

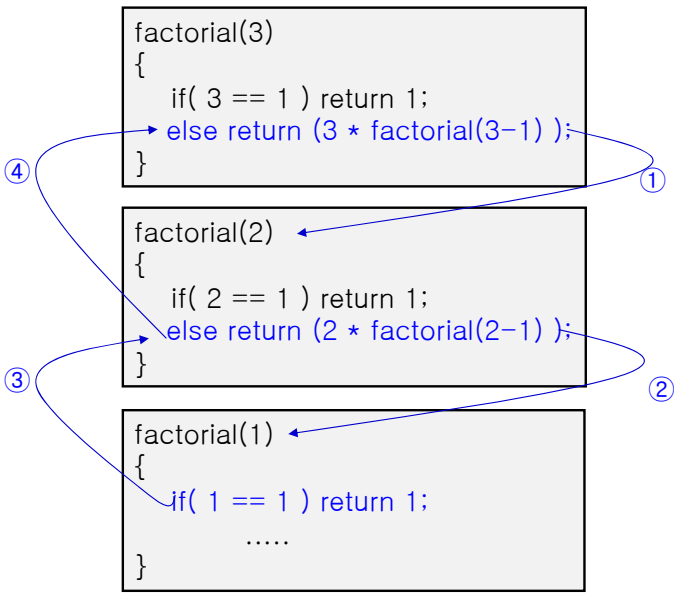
```
int factorial(int n)
{
    printf("factorial(%d)\n",n); // 순환 호출 순서 확인을 위한 출력문
    if( n == 1 ) return 1;
    else return (n * factorial(n-1) );
}
```



순환 호출 순서

- 팩토리얼 함수의 호출 순서

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 3 * 2
= 6



7.2 순환 알고리즘의 구조

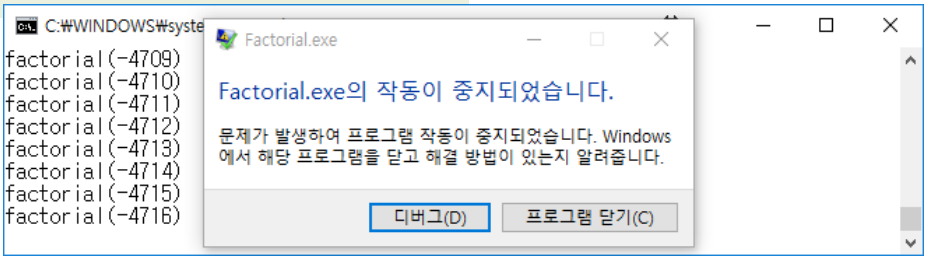
- 순환 알고리즘은 다음과 같은 부분들을 포함한다.
  - 순환 호출을 멈추는 부분
  - 순환 호출을 하는 부분

```
int factorial(int n)
{
    if( n == 1 )  return 1;          ← 순환을 멈추는 부분
    else return n * factorial(n-1);  ← 순환호출을 하는 부분
}
```


순환 알고리즘의 구조

- 만약 순환 호출을 멈추는 부분이 없다면?
  - 시스템 오류가 발생할 때까지 무한정 호출하게 된다

```
int factorial(int n)
{
  printf("factorial(%d)\n",n);
  // if( n == 1 ) return 1;
  // else
  return (n * factorial(n-1) );
}
```



- 1부터 n까지의 숫자를 전부 합하여 반환하는 순환 함수를 작성하라.

```
// (1+2+3+...+n)을 계산하여 반환한다.
int sum(int n)
{
    ...
}
```


- 다음 함수를 sum(5)로 호출하였을 때, 출력되는 내용과 함수의 반환 값을 구하라.

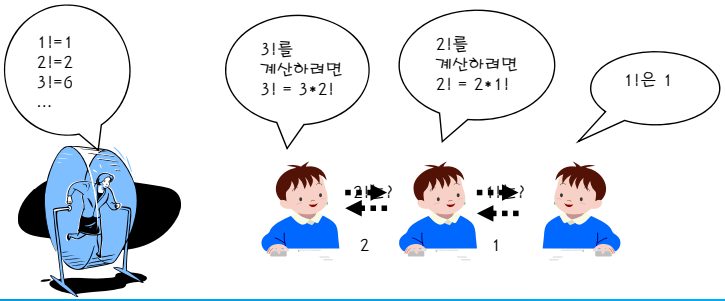
```
int sum(int n)
{
    printf("%d₩n", n);
    if ( n < 1 ) return 0;
    else return (n + sum(n-1));
}
```

- 다음 함수에서 asterisk(5)와 같이 호출할 때 출력되는 *의 개수는?

```
void asterisk(int i)
{
    if( i > 1 ) {
        asterisk(i/2);
        asterisk(i/2);
    }
    printf("*");
}
```


순환 <-> 반복

- 컴퓨터에서의 되풀이 : 순환과 반복
- 순환(recursion): 순환 호출 이용
  - 순환적인 문제에서는 자연스러운 방법
  - 함수 호출의 오버헤드
- 반복(iteration): for나 while을 이용한 반복
  - 수행속도가 빠르다.
  - 순환적인 문제에서는 프로그램 작성이 아주 어려울 수도 있다.
- 대부분의 순환은 반복으로 바꾸어 작성할 수 있음



팩토리얼의 반복적 구현

$$n! = \begin{cases} 1 & n = 1 \\ n*(n-1)*(n-2)*\Lambda\ *1 & n \geq 2 \end{cases}$$

```
int factorial_iter( int n )
{
    int result=1;
    for( int k=n ; k>0 ; k-- )
        result = result * k;
    return result;
}
```


분할 정복(divide and conquer)

- 순환은 문제를 나누어 해결하는 분할 정복 방법을 사용

```
factorial(int n)
{
    if( n == 1 ) return 1;
    else return ( n * factorial(n-1) );
}
```

해결된 부분

남아있는 부분

7.3 거듭제곱 값의 계산

- 순환적인 방법이 반복적인 방법보다 더 효율적인 예
- 숫자 x의 n제곱값을 구하는 문제: x^n

- **방법 1: 반복문 사용**

```
double slow_power(double x, int n)
{
  int i;
  double r = 1.0;
  for(i=0; i<n; i++)
     r = r * x;
  return(r);
}
```


순환적인 거듭제곱 함수

- **방법 2: 순환적인 호출**

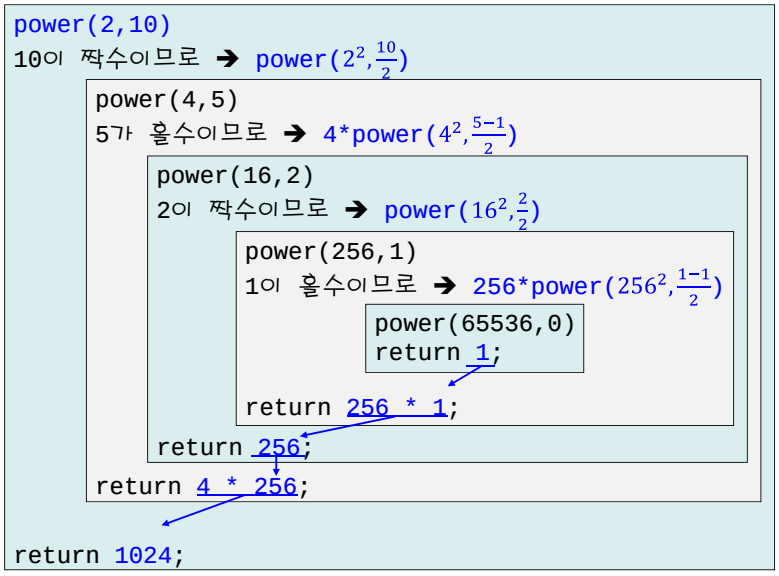
```
power(x, n)

if n = 0
    then return 1;
else if n이 짝수
    then return power(x2, n/2);
else if n이 홀수
    then return x*power(x2, (n-1)/2);
```

power(x, n) = power(x^2, n / 2)
$= (x^2)^{n/2}$
$= x^{2(n/2)}$
$= x^n$

power(x, n) = x • power(x^2, (n-1) / 2)
$= x \cdot (x^2)^{(n-1)/2}$
$= x \cdot x^{n-1}$
$= x^n$

```
double power(double x, int n)
{
      if( n==0 ) return 1;
      else if ( (n%2)==0 )
           return power(x*x, n/2);
      else
           return x*power(x*x, (n-1)/2);
}
```

거듭제곱을 구하는 순환 호출의 예

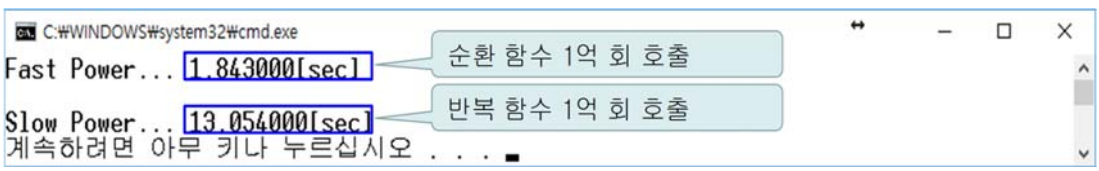


복잡도 분석

- 순환적인 방법의 시간 복잡도
  - n이 2의 제곱이라면 문제의 크기가 절반씩 줄어든다.

$$2^n \rightarrow 2^{n-1} \rightarrow \Lambda\ 2^2 \rightarrow 2^1 \rightarrow 2^0$$

- 시간 복잡도
  - 순환적인 함수: O(logn)
  - 반복적인 함수: O(n)



- 순환을 사용하여 배열 요소를 역순으로 화면에 출력하는 프로그램을 작성하라.

  요소 #3: 2

  요소 #2: 4

  요소 #1: 6

  요소 #0: 8

- 다음을 계산하는 순환적인 프로그램을 작성하라.

  1 + ½ + 1/3 + … + 1/n

7.4 피보나치 수열의 계산

- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열: 0,1,1,2,3,5,8,13,21,...

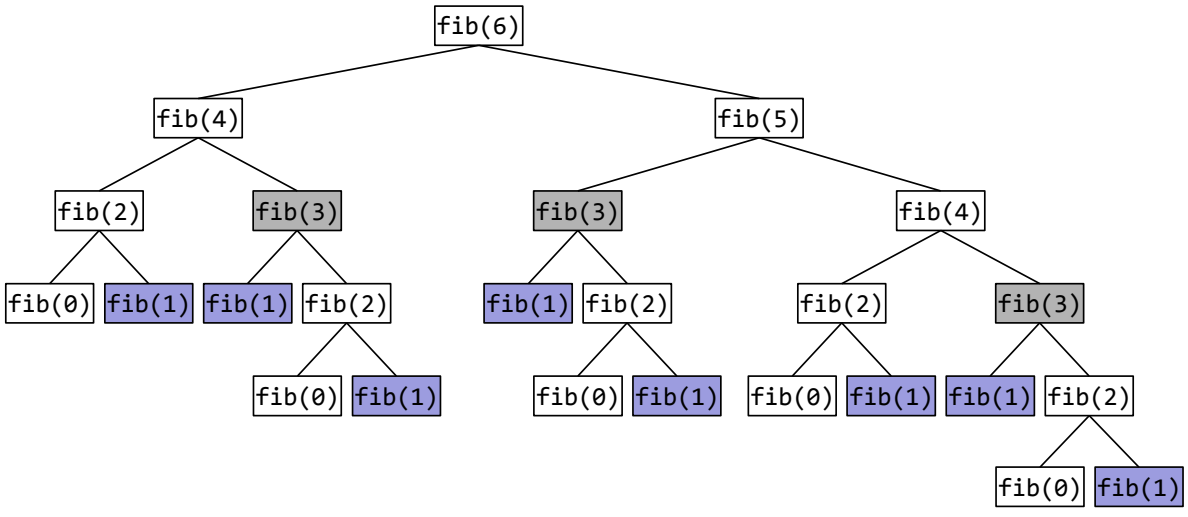
$$fib(n)\begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$$

- 순환적인 구현

```
int fib(int n)
{
    if( n==0 ) return 0;
    if( n==1 ) return 1;
    return (fib(n-1) + fib(n-2));
}
```


순환적인 피보나치의 비효율성

- 같은 항이 중복해서 계산됨!
  - n이 커지면 더욱 심각



- 순환적인 방법으로 피보나치수열을 호출하였을 때 함수가 중복되어 호출되는 것을 확인할 수 있도록 각 함수의 매개 변수별 호출 빈도를 측정해 출력하라.

  예) n = 10을 넣었을 때

  Fibo(10) = 1번

  Fibo(9) = ??번

  …

  Fibo(0) = ??번

- 자료형이 long인 경우 가장 큰 피보나치 수를 구하라.

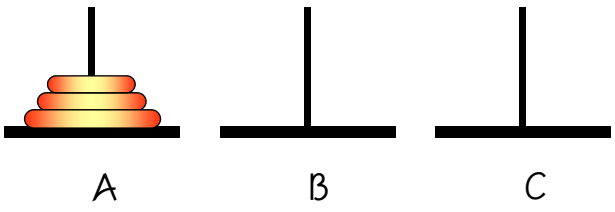
반복적인 피보나치 수열 함수

```
int fibIter(int n)
{
        if( n < 2 ) return n;
        else {
                int i, tmp, current=1, last=0;
                for(i=2;i<=n;i++){
                        tmp = current;
                        current += last;
                        last = tmp;
                }
                return current;
        }
}
```

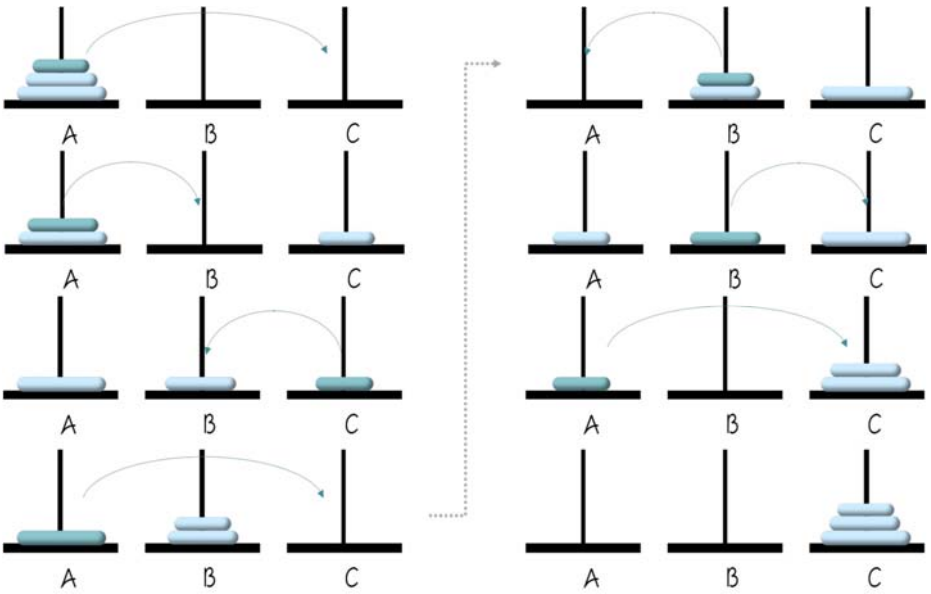
- 시간 복잡도: O(n)
- 순환적인 방법은?

7.5 하노이 탑 문제

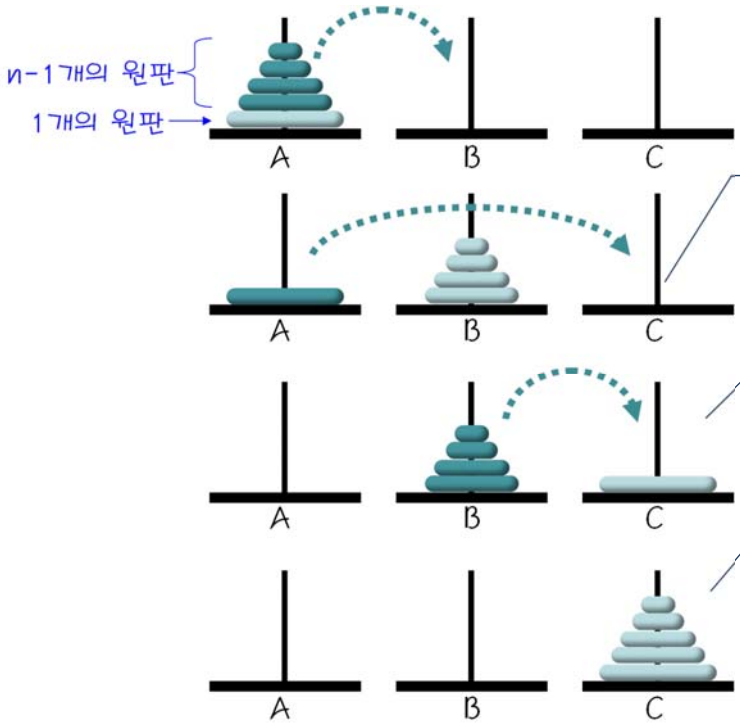
- 막대 A에 쌓여있는 원판 n개를 막대 C로 옮기는 문제
  - 한 번에 하나의 원판만 이동할 수 있음
  - 맨 위에 있는 원판만 이동할 수 있음
  - 크기가 작은 원판 위에 큰 원판이 쌓일 수 없음
  - 중간의 막대를 이용할 수 있으나 앞의 조건들을 지켜야 함



n=3인 경우의 해답



일반적인 경우에는?



C를 임시버퍼로 사용하여 A에 쌓여있는 n-1개의 원판을 B로 옮긴다.

A의 가장 큰 원판을 C로 옮긴다.

A를 임시버퍼로 사용하여 B에 쌓여있는 n-1개의 원판을 C로 옮긴다.

남아있는 문제?

- 어떻게 n-1개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
  - **순환을 이용**

```
// 막대 from에 쌓여있는 n개의 원판을 막대 tmp를 사용하여 막대 to로 옮긴다.
void hanoiTower(int n, char from, char tmp, char to)
{
   if (n==1){
       from에서 to로 원판을 옮긴다.
   }
   else{
   ① from의 맨 밑의 원판을 제외한 나머지 원판들을 tmp로 옮긴다.
   ② from에 있는 한 개의 원판을 to로 옮긴다.
   ③ tmp의 원판들을 to로 옮긴다.
   }
}
```


하노이탑 최종 프로그램

```
#include <stdio.h>
void hanoiTower(int n, char from, char tmp, char to)
{
     if( n==1 ) printf("원판 1을 %c에서 %c으로 옮긴다.\n",from,to);
     else {
          hanoiTower(n-1, from, to, tmp);
          printf("원판 %d을 %c에서 %c으로 옮긴다.\n",n, from, to);
          hanoiTower(n-1, tmp, from, to);
     }
}
void main() {
     hanoiTower(4, 'A', 'B', 'C');
}
```

하노이탑(n=3) 실행 결과

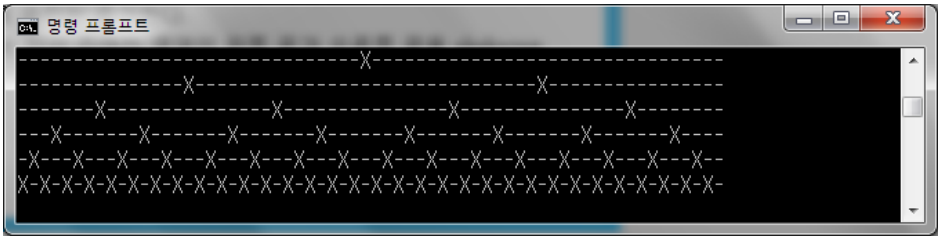
- 문자열의 내용을 반대로 바꾸는 순환적인 함수 reverse()를 구현하라. 예를 들어 reverse("ABCDE")는 "EDCBA"를 반환해야 한다.

7.5 순환을 이용한 미로 탐색(DFS)

```
void search_recur( int x, int y )
{
        if( done ) return;
        printf( "(%d,%d) ", x, y );
        if( x==xExit && y==yExit ) {
                done = 1;
                return;
        }
        map[y][x] = '5';
        if( is_valid(x-1, y) ) search_recur( x-1, y );
        if( is_valid(x+1, y) ) search_recur( x+1, y );
        if( is_valid(x, y-1) ) search_recur( x, y-1 );
        if( is_valid(x, y+1) ) search_recur( x, y+1 );
}
void main()
{
        search_recur( 0, 1 );
        if(done) printf("\n ==> 출구가 탐지되었습니다.\n");
        else printf("\n ==> 출구를 찾지 못했습니다.\n");
}
```

```
C:₩WINDOWS₩system32₩cmd.exe
(0,1) (1,1) (1,2) (2,2) (3,2) (3,1) (4,1) (3,3) (3,4) (4,4) (5,4)
 ==> 출구가 탐지되었습니다.
계속하려면 아무 키나 누르십시오 . . .
```


- 다음과 같은 모양을 출력하는 순환적인 함수를 작성하라.



– 이 함수의 원형은 다음과 같다.

void draw_tree(int row, int left, int right);

- row: X를 그리는 행을 표시한다. 가장 위에 있는 행이 0이고 아래로 내려갈수록 숫자는 증가한다.
- left와 right: 각각 주어진 영역의 왼쪽 끝과 오른쪽 끝을 나타낸다.