Data Structure

자료구조

알고리즘의 효율성 분석

한국기술교육대학교
온라인평생교육원

학습내용

- 알고리즘의 효율성 분석 개념
- 알고리즘의 복잡도 분석 개념
- 복잡도 분석 예: 반복 알고리즘

학습목표

- 알고리즘의 효율성 분석 개념을 설명할 수 있다.
- 알고리즘의 시간 복잡도 개념을 설명할 수 있다.
- 반복 알고리즘의 시간 복잡도를 분석할 수 있다.

알고리즘의 효율성 분석 개념

1 효율적인 알고리즘의 필요성

컴퓨터 기술의 발전

- 이전의 컴퓨터에 비해 엄청난 계산 속도와 방대한 메모리 제공
- 효율적인 알고리즘을 더 이상 고민할 필요가 없나?

효율적인 알고리즘이 필요한 이유

- 처리할 데이터 양도 급속히 증가
- 여전히 더 빠른 프로그램을 선호
  예 스마트폰 터치 후 반응 속도

알고리즘의 효율성 분석 알고리즘의 효율성 분석 개념

1 효율적인 알고리즘의 필요성

◉ 알고리즘 효율성 평가 기준

시간 효율성

- 알고리즘의 실행 시간
- 전체 실행 시간이 짧은 알고리즘이 더 좋음

공간 효율성

- 알고리즘이 사용하는 메모리의 양
- 컴퓨터의 자원들을 더 적게 사용하는 알고리즘이 더 좋음

알고리즘의 효율성 분석 개념

알고리즘의 효율성 분석

알고리즘의 효율성 분석 개념

1 효율적인 알고리즘의 필요성

◉ 알고리즘 효율성 평가 전략

1 실행 시간(사용된 메모리)을 실제로 측정하는 방법

- 공간 효율성의 경우 사용되는 메모리를 실제로 측정

2 알고리즘의 효율성을 이론적으로 분석하는 방법

- 공간 효율성의 경우 필요한 메모리를 이론적으로 분석

알고리즘의 효율성 분석

알고리즘의 효율성 분석 개념

2 실행시간 측정 방법

알고리즘을 실행하고 실제 실행 시간을 측정

알고리즘 A

실행시간 10초

알고리즘 B

실행시간 50초

알고리즘의 효율성 분석 개념

2 실행시간 측정 방법

◉ 실행시간 측정 방법의 문제점

1 반드시 구현해야 함

- 이론적인 분석 방법은 구현하지 않아도 분석이 가능

2 같은 조건에서 실행해야 함

- 동일한 하드웨어, 프로그래밍 언어, 운영체제 등

3 동일한 테스트 데이터를 사용해야 함

- 알고리즘에 유리한 데이터, 불리한 데이터가 있음

알고리즘의 효율성 분석 알고리즘의 효율성 분석 개념

2 실행시간 측정 방법

◉ 파이썬에서 실행시간을 측정하는 코드

```
import time                          # time 모듈 불러오기

start = time.time()                  # 현재 시각을 저장
max = find_max(array)                # 실행 시간을 측정하려는 코드
end = time.time()                    # 현재 시각을 저장

print("최댓값 = ", max)
print("실행시간 = ", end-start) # 실행 시간(종료-시작)을 출력
```

알고리즘의 효율성 분석 개념

```
# 자료구조-2회차 : elapsed_time.py
#
# [실습 2-1] 알고리즘의 실행시간 측정
#=====================================================

# 속도를 측정하고자 하는 알고리즘
def find_max( A ):
    max = A[0]
    for item in A :
        if item > max :
            max = item
    return max

# 실행시간 측정 프로그램
import time                     # time 모듈 불러오기

start = time.time()             # 현재 시각을 저장
max = find_max(array)           # 실행 시간을 측정하려는 코드
end = time.time()               # 현재 시각을 저장

print("최댓값 = ", max)
```

실습 단계
파이썬에서 실행시간 측정하기 위해 time 모듈 포함
import를 이용하여 time모듈을 불러 현재 시작 측정
전체적인 프로그램의 구조
import를 시켜 time모듈 포함
코드의 앞뒤에 현재 시작 저장
현재 시각을 저장-알고리즘 실행- 함수의 끝- 다시 현재 시작 측정(end)
알고리즘 속도를 측정: 최댓값 테스트 후 실행시간(종료 - 시작)을 출력
array준비
알고리즘 실행시간 예측
디버그 하지 않고 실행(ctrl+F5실행)
Elapsed time을 시작 항목으로 설정
실행결과, 최댓값 9, 실행시간 0.0
random 모듈 사용하기 위한 random 모듈 포함시키기
10000개의 숫자를 랜덤으로 발생시키는 코드

알고리즘의 효율성 분석 개념

실습 단계
의미 있는 시간으로 결과 도출: 최댓값 10000, 실행시간 3.340….
알고리즘의 효율성을 평가하기 위한 방법
첫번째 실행시간을 측정
두번째 복잡도 분석

알고리즘의 효율성 분석 개념

3 이론적인 효율성 분석

복잡도(Complexity)
이론적으로 분석한 알고리즘의 효율성을 의미 ▪ 시간 복잡도(Time Complexity) ▪ 공간 복잡도(Space Complexity)

시간 복잡도 분석
▪ 알고리즘을 직접 구현하지 않고도 가능 ▪ 알고리즘이 수행하는 연산의 횟수를 계산 ▪ 입력의 개수가 증가함에 따라 연산의 횟수가 어떤 형태로 증가 하는지에만 관심

알고리즘의 효율성 분석 알고리즘의 효율성 분석 개념

3 이론적인 효율성 분석

◉ 복잡도 분석의 간단한 예

▸▸ 자료구조 Bag의 삽입 연산 – 두가지 알고리즘이 가능

방법 1	배열의 맨 뒤에 새로운 항목을 추가하는 방법
방법 2	배열의 맨 앞에 새로운 항목을 추가하는 방법

▸▸ 여행을 떠나기 위해 가방을 정리하는 과정과 유사

▪ 지금까지 정리된 가방에 새로 옷을 하나 추가하는 방법

방법 1	가방의 맨 위에 옷 넣기
방법 2	가방의 맨 아래에 옷 넣기

알고리즘의 효율성 분석 개념

알고리즘의 효율성 분석 | 알고리즘의 효율성 분석 개념

3 이론적인 효율성 분석

◉ 방법 1: 배열의 맨 뒤에 삽입하는 방법

▸▸ append() 함수 사용

```
def insert(bag, e) :    # bag 항목에 e를 넣는 함수
    bag.append(e)       # 파이썬 리스트의 맨 뒤에 추가
```

▸▸ 가방의 맨 위에 옷 넣기

열고 → 넣고 → 닫고

3번의 연산

알고리즘의 효율성 분석 | 알고리즘의 효율성 분석 개념

3 이론적인 효율성 분석

◉ 방법 2: 배열의 맨 앞에 삽입하는 방법

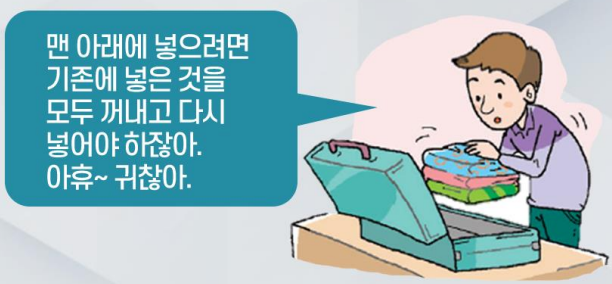
▸▸ append() 함수 사용

```
def insert(bag, e) :    # bag 항목에 e를 넣는 함수
    bag.insert(0, e)    # 파이썬 리스트의 맨 앞에 추가
```

▸▸ 가방의 맨 아래에 옷 넣기

열고 → 모든 옷을 꺼내고 → 넣고 → 다시 모든 옷을 넣고 → 닫고

필요한 연산의 수?
가방 안의 옷의 수 *2 + 3



알고리즘의 복잡도 분석 개념

1 시간 복잡도 함수

시간 복잡도 함수

- 알고리즘이 수행하는 연산의 횟수를 나타내는 함수
- T(n) – 입력의 개수 n에 대한 함수로 표현

연산

- 산술, 대입, 비교, 이동의 기본적인 연산들
- 보통 루프 제어 연산 등은 고려하지 않음

알고리즘의 효율성 분석 알고리즘의 복잡도 분석 개념

1 시간 복잡도 함수

◉ 복잡도 함수 구하기

예 n^2을 구하는 문제에 대한 3 가지 알고리즘: 복잡도 함수

알고리즘	A	B	C

유사 코드	sum $n*n$	for i←1 to n do
		sum ← sum + n
			for i ← 1 to n do
			for j ←1 to n do
			sum ← sum + 1

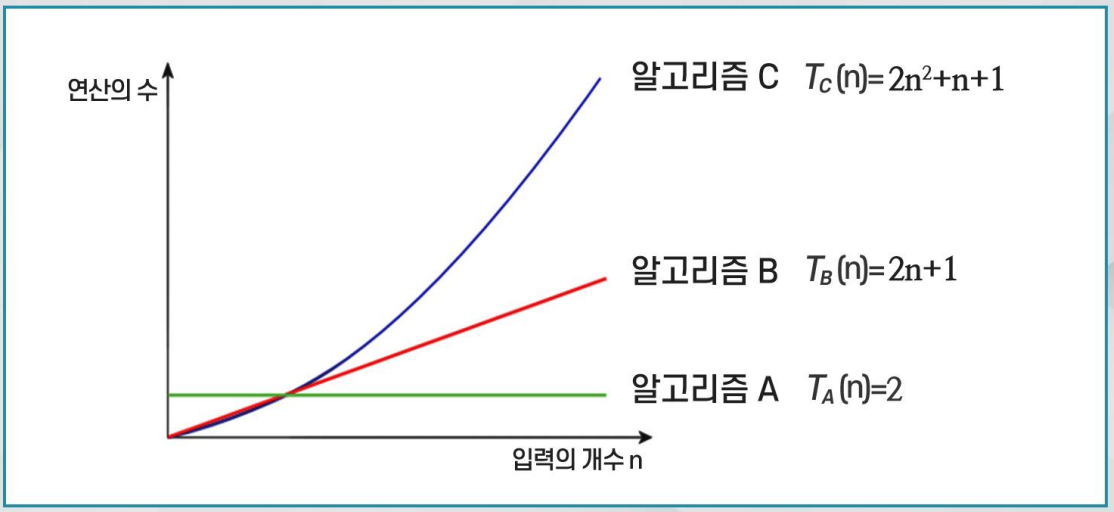
연산 횟수	대입연산: 1 곱셈연산: 1	대입연산: n+1 덧셈연산: n	대입연산: n^2+n+1 덧셈연산: n^2
복잡도 함수	$T_A(n)=2$	$T_B(n)= 2n+1$	$T_C(n)= 2n^2+n+1$

알고리즘의 복잡도 분석 개념

알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

1 시간 복잡도 함수

◉ 복잡도 함수와 증가 속도



알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

점근적 표기(Asymptotic)

- 복잡도 함수를 좀 더 단순화하여 나타내는 방법
- 아이디어
  - 복잡도 함수에서 n이 충분히 커지면?
  - 차수가 가장 큰 항이 절대적인 영향을 미치고, 다른 항들은 상대적으로 무시

예 $T(n) = n^2+n+1$

n=1일 때	T(n) = 1 + 1 + 1 = 3 (n^2항이 33. 3%)
n=10일 때	T(n) = 100 + 10 + 1 = 111 (n^2항이 90%)
n=100일 때	T(n) = 10000 + 100 + 1 = 10101 (n^2항이 99%)
n=1,000일 때	T(n) = 1000000 + 1000+ 1 = 1001001 (n^2항이 99.9%)

알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

점근적 표기(Asymptotic)

- 입력의 크기 n이 무한대로 커질 때의 복잡도를 간단히 표현하기 위해 사용하는 방법
- 복잡도 함수의 최고 차항만을 계수 없이 취해 단순화
- "정확히 몇 번"의 연산이 아니라 "무엇에 비례하는" 연산이 필요한가?

알고리즘의 효율성 분석 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 점근적 표기의 종류

▸▸ 복잡도 함수에 상한(upper bound), 하한(lower bound) 및 동일 등급(tight bound)의 개념을 적용해 분류

빅오	복잡도 함수의 상한
빅오메가	복잡도 함수의 하한
빅세타	점근적으로 상한이면서 하한인 경우

알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 빅오 표기법

빅오 표기법의 정의

- O(g(n))은 증가 속도가 g(n)과 같거나 낮은 모든 복잡도 함수를 포함하는 집합
- 복잡도 함수 증가 속도의 상한

예 $3n^2 + 4n \in O(n^2)$, $2n - 3 \in O(n^2)$, $2n(n+1) \in O(n^2)$

$3n^2 + 4n \notin O(n)$, $0.000001n^3 \notin O(n^2)$, $1000^n \in O(n!)$

알고리즘의 효율성 분석 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 빅오메가 표기법

빅오메가 표기법의 정의

- Ω(g(n))은 증가 속도가 g(n)과 같거나 높은 모든 복잡도 함수를 포함하는 집합
- 복잡도 함수 증가 속도의 하한

예 $2n^3 + 3n \in \Omega(n^2)$, $2n(n+1) \in \Omega(n^2)$,

$100000n + 8 \notin \Omega(n^2)$

알고리즘의 복잡도 분석 개념

알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 빅세타 표기법

빅세타 표기법의 정의

- $\Theta(g(n))$은 증가 속도가 $g(n)$과 같은 모든 복잡도 함수들만을 포함하는 집합
- 복잡도 함수 증가 속도의 상한인 동시에 하한

예 $2n^3 + 3n \in \Theta(n^3)$, $2n^3 + 3n \notin \Theta(n^2)$,

$2n^3 + 3n \notin \Theta(n^4)$

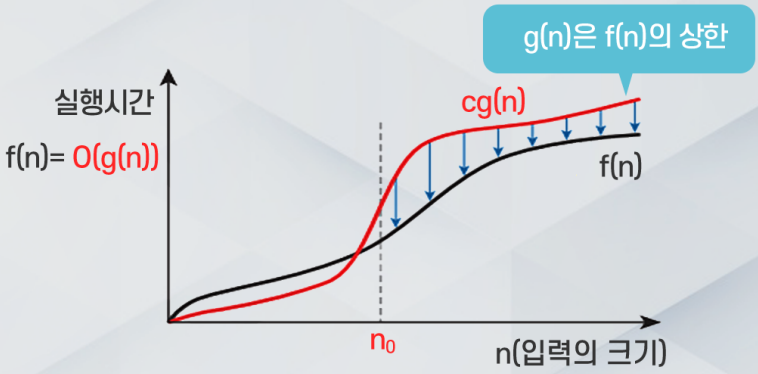
알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 점근적 표기의 정의

빅오 표기법의 정의

- 두개의 함수 f(n)과 g(n)이 주어졌을 때 모든 $n > n_0$에 대해 $|f(n)| \le c|g(n)|$을 만족하는 상수 c와 n_0가 존재하면 f(n)= O(g(n))이다.



- f(n)=2n+1이면 O(n)이다. 왜냐하면 n0=2, c=3일 때, n ≥ 2에 대하여 2n+1≤ 3n이 되기 때문이다.

알고리즘의 복잡도 분석 개념

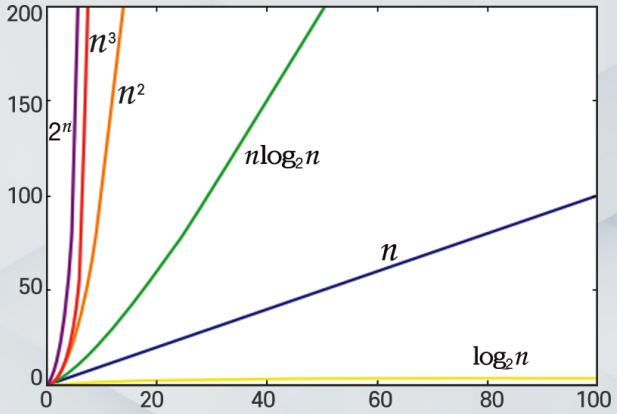
알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

2 점근적(Asymptotic)표기

◉ 자주 사용되는 점근적 클래스들

▸▸ 자주 사용되는 빅오 표기

O(1): 상수형
O(logn): 로그형
O(n): 선형
O(nlogn):선형로그형
O(n^2): 2차형
O(n^3): 3차형
O(2^n): 지수형
O(n!): 팩토리얼형



알고리즘의 효율성 분석 | 알고리즘의 복잡도 분석 개념

3 최선, 최악, 평균적인 효율성

같은 알고리즘이라도 입력의 종류에 따라 처리 시간이 크게 달라질 수 있음

알고리즘의 복잡도 분석 개념

알고리즘의 효율성 분석

알고리즘의 복잡도 분석 개념

3 최선, 최악, 평균적인 효율성

최선의 경우(Best case)

- 실행시간이 가장 적은 경우, 큰 의미가 없음

평균적인 경우(average case)

- 알고리즘의 모든 입력을 고려하고 각 입력이 발생할 확률을 고려한 평균적인 실행 시간
- 평균에 대한 가정이 필요하고, 정확히 계산하기가 어려움

최악의 경우(worst case)

- 입력의 구성이 알고리즘의 실행 시간을 가장 많이 요구하는 경우를 말하는데, 가장 중요하게 사용

예 실시간 공항 비행 관제 시스템

복잡도 분석 예: 반복 알고리즘

알고리즘의 효율성 분석 | 복잡도 분석 예: 반복 알고리즘

1 자연수의 제곱(n^2) 계산

자연수의 제곱을 계산하는 세 가지 알고리즘

- 알고리즘 A: 곱셈 1번
- 알고리즘 B: 덧셈 n번
- 알고리즘 C: 덧셈 n*n번

알고리즘의 복잡도 분석

- 입력의 크기는 무엇일까?
- 기본 연산으로 어떤 것을 선택하면 될까?
- 최선/최악/평균의 경우에 다른 복잡도를 보일까?
- 점근적 복잡도는?

알고리즘의 효율성 분석 | 복잡도 분석 예: 반복 알고리즘

1 자연수의 제곱(n^2) 계산

◉ 세 가지 알고리즘

A 알고리즘

```
def squareAlgorithmA( x ):
    sum = x * x
    return sum
```

B 알고리즘

```
def squareAlgorithmB( x ):
    sum = 0
    for i in range (x) :
        sum += x
    return sum
```

C 알고리즘

```
def squareAlgorithmC( x ):
    sum = 0
    for i in range (x) :
        sum += 1
    return sum
```

복잡도 분석 예: 반복 알고리즘

```python
# 자료구조-2회차 : compare_squares.py
#
# [실습 2-3-1] 제곱을 구하는 세 가지 알고리즘
#=====================================================
def squareAlgorithmA( x ):
    sum = x * x
    return sum

def squareAlgorithmB( x ):
    sum = 0
    for i in range (x) :
        sum += x
    return sum

def squareAlgorithmC( x ):
    sum = 0
    for i in range (x) :
        for j in range (x) :
            sum += 1
    return sum
```

실습 단계
자연수의 제곱을 구하는 알고리즘 A
자연수의 제곱을 구하는 알고리즘 B
자연수의 제곱을 구하는 알고리즘 C
입력의 크기 x, 자연수 그 자체
알고리즘 A, B, C 기본연산
최선, 최악, 평균의 복잡도는 나누어서 계산할 필요가 없음
알고리즘 A, B, C : 9*9의 실행결과
알고리즘 A, B, C : 10000*10000의 실행시간 결과
알고리즘 A, B, C 효율 비교: C 가장 비효율적

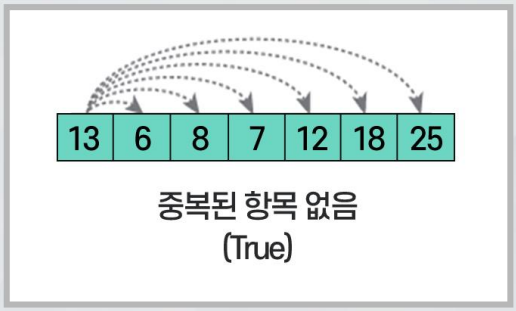
복잡도 분석 예: 반복 알고리즘

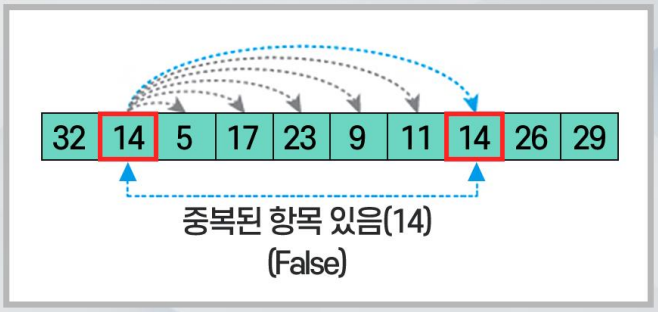
알고리즘의 효율성 분석 | 복잡도 분석 예: 반복 알고리즘

2 리스트의 중복 항목 검사

리스트에 같은 항목이 있는지 검사

- 중복된 항목이 하나라도 있으면 → False 반환(유일하지 않음)
- 중복된 항목이 없으면 → True 반환(유일함)





알고리즘의 효율성 분석 | 복잡도 분석 예: 반복 알고리즘

2 리스트의 중복 항목 검사

◉ 알고리즘

```
def unique_elements(A) :            # 리스트 A입력
    n = len(A)                      # 입력의 크기 = 리스트의 크기
    for i in range(n-1) :           # i : 0, 1, ... n-2
        for j in range(i+1,n) :     # j : i+1, i+2, ... n-1
            if A[i] == A[j] :       # 기본 연산
                return False        # 같은 항목이 있으면 True 반환
    return True                     # 같은 항목이 없으면 False 반환
```

◉ 복잡도 분석

복잡도 분석 예: 반복 알고리즘

```python
# 자료구조-2회차 : unique_elem.py
#
# [실습 2-3-2] 리스트의 중복 항목 검사 알고리즘
#=====================================================

def unique_elements(A) :          # 리스트 A입력
    n = len(A)                    # 입력의 크기 = 리스트의 크기
    for i in range(n-1) :         # i : 0, 1, ... n-2
        for j in range(i+1,n) :   # j : i+1, i+2, ... n-1
            if A[i] == A[j] :     # 기본 연산
                return False      # 같은 항목이 있으면 False 반환
    return True                   # 같은 항목이 없으면 True 반환


A = [32, 14, 5, 17, 23, 9, 11, 14, 26, 29]
B = [13, 6, 8, 7, 12, 25]
C = [22, 16, 14, 12, 7, 13, 27, 17]

print(A, unique_elements(A))
print(B, unique_elements(B))
print(C, unique_elements(C))
```

실습 단계
A, B, C세개의 알고리즘 샘플 적용: A=False반환/ B=True 반환/ C=True 반환
알고리즘의 입력의 크기=리스트의 크기
i번째 항목과 j번째 항목이 같으면 유니크한 항목이 아님
기본연산
최선의 복잡도
최악의 복잡도
알고리즘 실행 및 결과: A=False/ B=True / C=True

복잡도 분석 예: 반복 알고리즘

3 2진수 표기의 비트 수

자연수의 2진수 표기에서 총 비트 수

예 13 → 1 1 0 1 (4비트)

알고리즘의 효율성 분석 복잡도 분석 예: 반복 알고리즘

3 2진수 표기의 비트 수

◉ 알고리즘

```
def binary_digits(n) :          # 입력 양의 정수 n
    count = 1                   # 이진수의 최소 길이는 0
    while n > 1 :               # n이 1보다 크면 더 나눌 수 있음
        count = count + 1       # count 증가
        n = n // 2              # n의 몫을 다시 n에 저장 (정수 나눗셈)
    return count                # count를 반환
```

복잡도 분석 예: 반복 알고리즘

```python
# 자료구조-2회차 : bin_digits.py
#
# [실습 2-3-3] 2진수 표기의 비트 수
#=====================================================

def binary_digits(n) :          # 입력 양의 정수 n
    count = 1                   # 이진수의 최소 길이는 0
    while n > 1 :               # n이 1보다 크면 더 나눌 수 있음
        count = count + 1       # count 증가
        n = n // 2              # n의 몫을 구해 다시 n에 저장 (정수 나눗셈)
    return count                # count를 반환

#=====================================================
print("총 비트수 (  0) = ", binary_digits(0))
print("총 비트수 (  1) = ", binary_digits(1))
print("총 비트수 ( 13) = ", binary_digits(13))
print("총 비트수 (123) = ", binary_digits(123))
print("총 비트수 (123456789012345) = ", binary_digits(123456789012345))
```

실습 단계
알고리즘의 이름: binary_digits(n)
count = 1, 최소의 비트 수는 1
1보다 클 때, 반복문 연산을 하게 됨
예) 3이 주어짐 → 3>1으로 count 1 증가 → 3//2 → count 2 반환
알고리즘 실행 및 결과
입력의 크기는 정수 n의 크기 즉, n자체가 입력의 크기
기본연산
최선, 최악, 평균의 복잡도 동일 즉, 분리해서 사용할 필요 없음
시간 복잡도 분석
반복문의 횟수가 결국 나눗셈의 횟수가 됨
결국 알고리즘의 복잡도는 $\log_2 n$

복잡도 분석 예: 반복 알고리즘

3 2진수 표기의 비트 수

◉ 복잡도 분석

- 기본 연산의 실행 회수 → while 문의 반복 회수
- n이 2의 k거듭제곱이라면: $n = 2^k$
- 반복할 때마다 문제의 크기가 절반씩 줄어듦
  $2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \cdots \rightarrow 2^1 \rightarrow 2^0$

시간 복잡도
▪ While 문의 반복 회수 = = k ▪ 복잡도: $O(k) = O(\log_2 n)$