Ch1. 알고리즘 : 효율성, 분석 및 주문

목차

알고리즘 : 효율성, 분석 및 주문

알고리즘효율적인 알고리즘 개발의 중요성알고리즘 분석순서

알고리즘

예선

□ '문제'라는 단어

○ 문제의 정의

➢ 컴퓨터 프로그램은 컴퓨터가 이해할 수 있는 개별 모듈로 구성되어 있습니다.
특정 작업(예: 정렬)."이 텍스트에서 우리의 관심은 전체 프로그램의 설계가 아니라 프로그램의 설계입니다.
특정 작업을 수행하는 개별 모듈입니다. 이러한 특정 작업을 문제라고 합니다.

○ '문제'의 예시

예시1.

n개의 숫자 중 목록 S를 내림차순으로 정렬합니다. 대답은 sortedsequence의 숫자입니다.

목록이란 특정 순서로 정렬된 항목의 모음을 의미합니다. 예를 들어

에스 = [10, 7, 11, 5, 13, 8]

는 첫 번째 숫자가 10, 두 번째 숫자가 7 등인 6개 숫자의 목록입니다. 이 예에서는 동일한 숫자가 목록에 두 번 이상 나타날 수 있는 가능성을 허용하기 위해 순서를 늘리는 대신 목록을 "내림차순"으로 정렬해야 한다고 말합니다.

예선

□ '문제'라는 단어 (계속)

○ '문제'의 예 (계속)

예시2.

숫자 x가 n개 숫자 중 목록 S에 있는지 확인합니다. 대답은 x가 모래에 있으면 예, 그렇지 않으면 아니오입니다.

문제에는 문제 설명에서 특정 값이 지정되지 않은 변수가 포함될 수 있습니다. 이러한 변수를 문제에 대한 매개 변수라고 합니다. Example1에는 S(목록)와 n(S의 항목 수)이라는 두 개의 매개 변수가 있습니다. 예제 2에는 S, n 및 숫자 x의 세 가지 매개 변수가 있습니다. 이 두 예에서는 n의 값이 S에 의해 고유하게 결정되기 때문에 n을 매개변수 중 하나로 만들 필요가 없습니다. 그러나 n을 매개변수로 만들면 문제를 쉽게 설명할 수 있습니다.

예선

□ '해결책'이라는 단어

○ '솔루션'의 정의

➢ 문제는 매개변수를 포함하기 때문에, 문제의 각 할당에 대해 하나씩 문제의 클래스를 나타냅니다. 매개변수에 값을 할당하는 각 특정 값을 문제의 인스턴스라고 합니다. A

문제 인스턴스에 대한 해결책은 해당 인스턴스에서 문제가 묻는 질문에 대한 답변입니다.

예시3.

Example1의 문제 인스턴스는 S = [10, 7, 11, 5, 13, 8] 및 n = 6입니다.

이 인스턴스의 솔루션은 [5, 7, 8, 10, 11, 13]입니다.

예시4.

Example2의 문제 인스턴스는 S = [10, 7, 11, 5, 13, 8], n = 6 및 x = 5입니다.

이 인스턴스의 해결책은 "예, x는 S에 있습니다"입니다.

예선

□ '알고리즘'이라는 단어

○ '알고리즘'의 정의

- 우리는 S를 조사하고 마음이 생산할 수 있도록 함으로써 예 3의 사례에 대한 해결책을 찾을 수 있습니다 구체적으로 설명할 수 없는 인지 단계를 기준으로 정렬된 순서입니다. 이것은 S가 너무 작아서 의식 수준에서 마음이 S를 빠르게 스캔하고 거의 즉각적으로 해결책을 만들어내는 것처럼 보이기 때문에 가능할 수 있다(따라서 마음이 해결책을 얻기 위해 따르는 단계를 설명할 수 없다)."문제의 모든 사례를 해결할 수 있는 컴퓨터 프로그램을 만들기 위해, 우리는 다음을 지정해야 한다.

  각 인스턴스에 대한 솔루션을 생성하기 위한 일반적인 단계별 절차입니다. 이 단계별 절차를 알고리즘이라고 합니다.

○ '알고리즘'의 예시

예시5.

S의 첫 번째 항목부터 시작하여 x가 발견될 때까지 또는 S가 소진될 때까지 x를 S의 각 항목과 순서대로 비교합니다. x가 발견되면 예라고 대답하십시오. X를 찾을 수 없으면 아니오라고 대답하십시오.

⬇ 설명 알고리즘의 문제점

첫째, 복잡한 알고리즘을 이런 식으로 작성하는 것은 어렵고, 설령 작성하더라도 알고리즘을 이해하는 데 어려움을 겪을 것입니다. 둘째, 알고리즘에 대한 영어 설명으로부터 알고리즘에 대한 컴퓨터 언어 설명을 생성하는 방법이 명확하지 않습니다.

예선

□ '알고리즘'의 단어 (계속)

○ 의사 코드에 의한 예

➢ 다음 알고리즘은 목록 S를 배열로 나타내며, 단순히 yes 또는 no를 반환하는 대신 x가 S에 있으면 배열에서 x의 위치를 반환하고 그렇지 않으면 0을 반환합니다.

Sequential Search

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
{
    location = 1;
    while (location<= n && S[location] != x)
        location++;
    if (location > n)
        location=0;
}
```

예선

□ '알고리즘'의 단어 (계속)

○ 배열의 의사 코드와 C/C++ 코드의 차이점

➢ 우리는 가변 길이 2차원 배열을 루틴에 대한 매개변수로 허용합니다.

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
```

➢ 우리는 지역 가변 길이 배열을 선언합니다.

```
void example (int n)
{
   keytype S[2..n];
      ⋮
}
```

➢ 실제 C ++을 사용하는 것보다 수학 표현이나 영어와 같은 설명을 허용합니다.
지시

```
if (low ≤ x ≤ high) {                        if (low <= x && x <= high){
        ⋮                  rather than              ⋮
}                                            }
```

예선

□ '알고리즘'의 단어 (계속)

○ 유사 코드와 C/C++ 코드 배열의 차이점 (계속)

➢ 실제 C ++을 사용하는 것보다 수학 표현이나 영어와 같은 설명을 허용합니다.
지시. (계속)

exchange x and y; rather than $temp = x$; $x = y$; $y = temp$;

➢ 데이터 유형 키 유형 외에도 미리 정의된 C++ 데이터 유형이 아닌 다음을 자주 사용합니다

데이터 형식	의미
색인	인덱스로 사용되는 정수 변수
부울	"true" 또는 "false" 값을 사용할 수 있는 변수
수	정수(int) 또는 실수(float)로 정의할 수 있는 변수

➢ 때때로 우리는 다음과 같은 비표준 제어 구조를 사용합니다.

```
repeat (n times){
   :
}
```

예선

□ '알고리즘'의 단어 (계속)

○ 유사 코드와 C/C++ 코드 배열의 차이점 (계속)

➢ 우리는 비표준 논리 연산자와 특정 관계 연산자를 익숙하지 않게 허용합니다.

Operator	C++ symbol
and	&&
or	||
not	!

Comparison	C++ code
$x = y$	$(x == y)$
$x \neq y$	$(x != y)$
$(x \leq y)$	$(x <= y)$
$x \geq y$	$(x >= y)$

➢ 함수 이름 앞에 입력 할 수 없습니다.

Add Array Members

Problem: Add all the numbers in the array *S* of *n* numbers.

Inputs: positive integer *n*, array of numbers *S* indexed from 1 to *n*.

Outputs: *sum*, the sum of the numbers in *S*.

```
number sum (int n, const number S[ ])
{
    index i;
    number result;

    result = 0;
    for (i = 1; i <= n; i++)
       result = result + S[i];
    return result;
}
```

예선

□ '알고리즘'의 단어 (계속)

○ 두 개의 2 x 2 행렬을 사용한 행렬 곱셈의 예

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

their product $C = A \times B$ is given by

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}.$$

For example,

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 \times 5 + 3 \times 6 & 2 \times 7 + 3 \times 8 \\ 4 \times 5 + 1 \times 6 & 4 \times 7 + 1 \times 8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}.$$

In general, if we have two $n \times n$ matrices A and B, their product C is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \qquad \text{for } 1 \le i, j \le n.$$

예선

□ '알고리즘'의 단어 (계속)

○ 두 개의 2 x 2 행렬을 사용한 행렬 곱셈의 예 (계속)

Matrix Multiplication

Problem: Determine the product of two $n \times n$ matrices.

Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from 1 to n.

Outputs: a two-dimensional array of numbers C, which has both its rows and columns indexed from 1 to n, containing the product of A and B.

```
void matrixmult (int n,
                 const number A[][],
                 const number B[][],
                       number C[][])
{
  index i, j, k;

  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++){
      C[i][j] = 0;
      for (k=1; k<=n; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
}
```

개발의 중요성

효율적인 알고리즘

순차 검색(Sequential Search) vs. 이진 검색 (Binary Search)

□ 이진 검색 알고리즘

Binary Search

Problem: Determine whether x is in the sorted array S of n keys.

Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

```
void binsearch (int n,
                const keytype S[],
                keytype x,
                index& location)
{
  index low, high, mid;

  low = 1; high = n;
  location = 0;
  while (low <= high && location == 0){
     mid = ⌊(low + high)/2⌋;
     if (x == S[mid])
        location = mid;
     else if (x < S[mid])
        high = mid - 1;
     else
        low = mid + 1;
  }
}
```

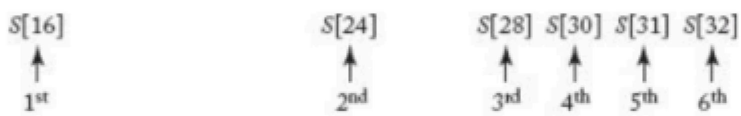
순차 검색(Sequential Search) vs. 이진 검색 (Binary Search)

□ 이진 검색 알고리즘 (계속)

○ while 루프에서 두 개의 비교 또는 하나의 비교?

➢ while 루프를 통한 각 패스에서 x와 S[mid]의 두 가지 비교가 있습니다(x가 found)를 사용합니다. 알고리즘의 효율적인 어셈블러 언어 구현에서 x는 각 패스에서 한 번만 S[mid]와 비교되며, 해당 비교의 결과는 조건 코드를 설정하고, 조건 코드의 값에 따라 적절한 분기가 발생합니다. 이것은 while 루프를 통한 각 패스에서 x와 S[mid]의 비교가 한 번만 있음을 의미합니다.

○ 이진 검색에서 배열 매개변수의 32개의 연속된 숫자에서 비교 횟수 알고리즘



➢ x가 모든 것보다 클 때 Sequential Search 및 Binary Search에 의해 수행되는 비교 횟수 배열 항목

배열 크기	# 비교 기준 순차 검색	# 비교 기준 이진 검색
128128		8
1,0241,024		11
1,048,5761,048,576		21
4,294,967,296	4,294,967,2968	33

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘

○ 재귀적 방식으로

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2 .$$

○ 처음 몇 개의 항 계산

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$
$$f_3 = f_2 + f_1 = 1 + 1 = 2$$
$$f_4 = f_3 + f_2 = 2 + 1 = 3$$
$$f_5 = f_4 + f_3 = 3 + 2 = 5, \text{ etc.}$$

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

○ 의사 코드

nth Fibonacci Term (Recursive)

Problem: Determine the nth term in the Fibonacci sequence.

Inputs: a nonnegative integer n.

Outputs: *fib*, the nth term of the Fibonacci sequence.

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

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

○ 재귀 트리(recursion tree)에 의한 알고리즘의 비효율성을 재귀적으로 증명하기 위해

➢ 처음 7개 값의 경우 트리의 항 수가 모든 값의 두 배 이상임을 주목하십시오.
시간 n이 2 증가합니다.

n	Number of Terms Computed
0	$\boxed{1}$
1	1
2	$\boxed{3}$
[illegible]	5
4	$\boxed{9}$
5	15
6	$\boxed{25}$

2회 이상

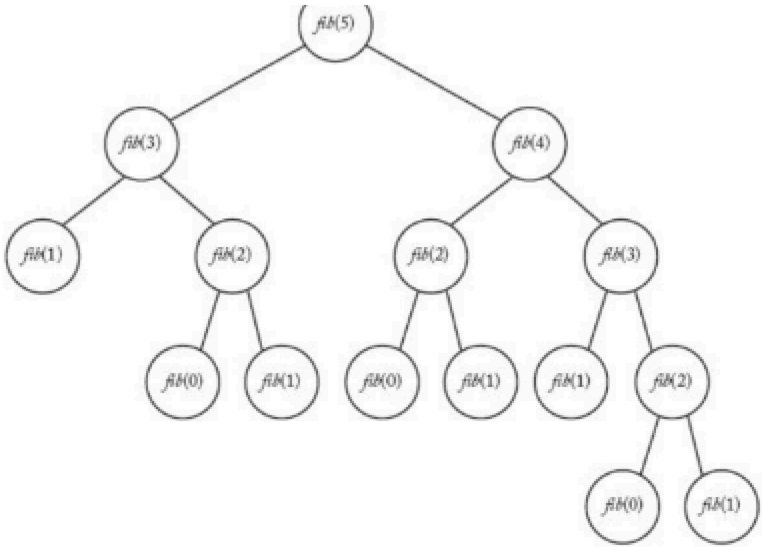
2회 이상

2회 이상

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

○ 재귀 트리(recursion tree)에 의한 알고리즘의 비효율성을 재귀적으로 증명하기 위해 (cont.)



$$
\begin{aligned}
T(n) &> 2 \times T(n-2) \\
&> 2 \times 2 \times T(n-4) \\
&> 2 \times 2 \times 2 \times T(n-6) \\
&\ \ \vdots \\
&> \underbrace{2 \times 2 \times 2 \times 2 \times \cdots \times 2}_{n/2 \text{ terms}} \times T(0)
\end{aligned}
$$

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

- ○ 정리
  - ➢ T(n)이 피보나치 알고리즘에 해당하는 재귀 트리의 항 수인 경우,
    ≥ 2 번, $$T(n) > 2^{n/2}$$
- ○ 증명(Proof)
  - ➢ 유도 기반: 귀납 단계는 이전 두 개의 결과를 가정하기 때문에 두 개의 기본 사례가 필요합니다.
    경우. n = 2 및 n = 3인 경우, $$T(2) = 3 > 2 = 2^{2/2}$$
    $$T(3) = 5 > 2.8323 \approx 2^{3/2}$$

귀납 가설(Induction hypothesis): 귀납 가설을 세우는 한 가지 방법은 그 진술이 참이라고 가정하는 것이다 모든 M < N. 그런 다음 귀납 단계에서 이것이 n에 대해 진술이 참이어야 함을 암시한다는 것을 보여줍니다. 이 기술은 이 증명에 사용됩니다. 모든 m에 대해 2 ≤ m < n

$$T(m) > 2^{m/2}.$$

- ➢ 유도 단계: T(n)이 2n/2> 것을 보여주어야 합니다. T(n)의 값은 T(n − 1)과 T(n − 2)의 합입니다 루트에 있는 하나의 노드도 있습니다. 그러므로

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{by induction hypothesis} \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{(\frac{n}{2})-1} = 2^{n/2}. \end{aligned}$$

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

○ 반복적인 방식의 의사 코드

nth Fibonacci Term (Iterative)

Problem: Determine the nth term in the Fibonacci sequence.

Inputs: a nonnegative integer n.

Outputs: *fib*2, the nth term in the Fibonacci sequence.

```
int fib2 (int n)
{
    index i;
    int f[0..n];

    f[0]=0;
    if (n > 0)
        f[1]=1;
        for (i=2; i<=n; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

피보나치 수열

피보나치 수열의 n번째 항을 계산하는 □ 알고리즘(계속)

○ 반복 방식의 의사 코드 (계속)

➢ 가장 최근의 두 용어 만 있기 때문에 배열 f를 사용하지 않고 알고리즘을 작성할 수 있습니다. 루프의 각 반복에 필요합니다."fib2(n)을 결정하기 위해 이전 알고리즘은 처음 n + 1 항을 모두 한 번만 계산합니다. 그래서 그것

n + 1 항을 계산하여 n 번째 피보나치 항을 결정합니다.

• 하나의 항을 1ns 안에 계산할 수 있다고 가정합니다.

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 μs†
60	61	1.1×10^{9}	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^{7} years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns = 10^{-9} second.

†1 μs = 10^{-6} second.

알고리즘 분석

알고리즘 분석

복잡성 분석을 위한 첫 번째 단계□ (입력 크기 결정)

○ 알고리즘의 효율성을 종속 요인에 의해 시간 측면에서 분석하지 마십시오.

➢ 실제 CPU 사이클 수는 특정 컴퓨터에 따라 다르기 때문에 결정하지 않습니다.
알고리즘이 실행되는 곳입니다."명령어의 수가 다르기 때문에 실행된 모든 명령어를 계산하고 싶지도 않습니다

알고리즘을 구현하는 데 사용되는 프로그래밍 언어와 프로그래머가 프로그램을 작성하는 방법.

○ DO 알고리즘의 효율성을 독립적인 요인에 의해 시간적으로 분석한다

➢ 일반적으로 알고리즘의 실행 시간은 입력의 크기와 총 실행 횟수에 따라 증가합니다
시간은 어떤 기본 작업(예: comparisoninstruction)이 몇 번 수행되는지에 대략 비례합니다."따라서 우리는 일부 기본 연산의 횟수를 결정하여 알고리즘의 효율성을 분석합니다.

연산은 입력의 크기에 대한 함수로 수행됩니다."많은 알고리즘의 경우 입력 크기에 대한 합리적인 측정을 쉽게 찾을 수 있습니다.

입력 크기.

- 입력 크기가 아닙니다.

➢ 이진 표현을 사용하는 경우 입력 크기는 n을 인코딩하는 데 필요한 비트 수가 됩니다.
LG N + 1 .

$$n = 13 = \underbrace{1101}_{4\ \text{bits}}{}_2$$

- 따라서 입력 n의 크기 = 13입니다.

알고리즘 분석

복잡성 분석을 위한 □ 두 번째 단계(기본 작업 선택)

- ○ 몇 가지 지침 또는 지침 그룹을 선택하십시오.
  - ➢ 알고리즘에 의해 수행된 총 작업이 이 횟수에 대략 비례하도록 합니다.
    명령어 또는 명령어 그룹, 기본 작동이 완료됩니다."예를 들어, x는 순차 검색의 루프를 통해 각 패스의 항목 S와 비교됩니다.
    이진 검색 알고리즘. 그러므로, compare 명령어는 이러한 알고리즘 각각에서 기본 연산에 대한 좋은 후보입니다."알고리즘에서 n의 각 값에 대해 이 기본 연산을 몇 번이나 수행하는지 결정함으로써, 우리는
    두 알고리즘의 상대적 효율성에 대한 통찰력을 얻었습니다.

복잡성 분석에 대한 일반적인 방법 □

- ○ 알고리즘의 시간 복잡도 분석
  - ➢ 입력 크기의 각 값에 대해 기본 연산이 몇 번 수행되는지에 대한 결정.➢ 알고리즘이 어떻게 구현되는지에 대한 세부 사항을 고려하고 싶지는 않지만 일반적으로
    기본 작업이 가능한 한 효율적으로 구현되었다고 가정합니다.
- ○ 제어하기 위해 인덱스를 증가시키고 비교하는 지침을 포함하지 마십시오.
  는 while 루프를 통과합니다.
  - ➢ 때로는 루프를 한 번 통과하는 것을 기본 실행의 한 번의 실행으로 간주하는 것만으로도 충분합니다.
    수술.

알고리즘 분석

복잡성 분석에 대한 일반적인 방법 □ (계속)

○ 알고리즘의 시간 복잡도 분석

➢ 입력 크기의 각 값에 대해 기본 연산이 몇 번 수행되는지에 대한 결정.➢ 알고리즘이 어떻게 구현되는지에 대한 세부 사항을 고려하고 싶지는 않지만 일반적으로
기본 작업이 가능한 한 효율적으로 구현되었다고 가정합니다.

○ 제어하기 위해 인덱스를 증가시키고 비교하는 지침을 포함하지 마십시오.

는 while 루프를 통과합니다.

➢ 때로는 루프를 한 번 통과하는 것을 기본 실행의 한 번의 실행으로 간주하는 것만으로도 충분합니다.
수술.

○ 두 가지 기본 작업을 고려할 수 있습니다.

예를 들어, 키를 비교하여 정렬하는 알고리즘에서 우리는 종종 비교를 고려하기를 원합니다 instruction과 assignment instruction은 각각 기본 연산으로서 개별적으로 존재한다."우리는 두 개의 뚜렷한 기본 연산을 가지고 있는데, 하나는 비교 명령어이고 다른 하나는 "따라서 알고리즘의 효율성에 대한 더 많은 통찰력을 얻을 수 있습니다."따라서 알고리즘의 수를 결정함으로써 알고리즘의 효율성에 대한 더 많은 통찰력을 얻을 수 있습니다
각각의 시간이 수행됩니다.

알고리즘 분석

모든 경우에 □ 시간 복잡도 (배열 멤버 추가)

○ 'Add Array Members' 예시

➢ 기본 작업: 배열에서 항목을 더하여 합산하는 것입니다.➢ 입력 크기: n, 배열의 항목 수입니다.➢ 배열의 숫자 값에 관계없이 for 루프를 통해 n개의 패스가 있습니다. 그러므로

기본 작업은 항상 n번 수행되고

$$\boxed{T}(n) = n$$

알고리즘 분석

모든 경우의 시간 복잡성 □

○ '거래소 정렬' 예시

➢ 앞서 언급했듯이 키를 비교하여 정렬하는 알고리즘의 경우 다음을 고려할 수 있습니다. 비교 명령 또는 할당 명령을 기본 작업으로 사용합니다. 여기서 비교 횟수를 분석해 보겠습니다.➢ 기본 연산: S[j]와 S[i]의 비교.➢ 입력 크기: n, 정렬할 항목의 수입니다.➢ for-j 루프를 통해 얼마나 많은 패스가 있는지 확인해야 합니다. 주어진 n에 대해 항상 n이 있습니다.

− 1은 for-i 루프를 통과합니다. for-i 루프를 통한 첫 번째 패스에서는 for-j 루프를 통한 n − 1 패스가 있고, 두 번째 패스에는 for-j 루프를 통한 n − 2 패스가 있고, 세 번째 패스에는 for-j 루프를 통한 -3 패스가 있으며, …, 마지막 패스에는 for-j 루프를 통한 1 패스가 있습니다. 따라서 for-j 루프를 통과하는 총 패스 수는 다음과 같이 지정됩니다.

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

알고리즘 분석

□ 모든 경우의 시간 복잡도(계속)

○ '행렬 곱셈' 예시

➢ 가장 안쪽의 for 루프에있는 유일한 명령어는 곱셈과 덧셈을 수행하는 명령어입니다. 이 알고리즘은 곱셈보다 더 적은 덧셈이 수행되는 방식으로 구현할 수 있습니다. 따라서 곱셈 명령어만 기본 연산으로 간주합니다.➢ 기본 연산: 가장 안쪽의 for 루프에 있는 곱셈 명령어.➢ 입력 크기: n, 행과 열의 수입니다.➢ for-i 루프를 통해 항상 n개의 패스가 있으며, 각 패스에는 항상 for-i를 통과하는 n개의 패스가 있습니다.

j 루프를 통과하고, for-j 루프를 통과하는 각 패스에는 항상 for-k 루프를 통한 n 패스가 있습니다. 기본 작업이 for-k 루프 내부에 있기 때문에

$$T(n) = n \times n \times n = n^3$$

알고리즘 분석

□ 최악의 경우 시간 복잡도(계속)

○ '순차검색' 예시

➢ 기본 작업: 배열의 항목을 x와 비교합니다.➢ 입력 크기: n, 배열의 항목 수입니다.➢ 기본 작업은 최대 n번 수행되며, x가 배열의 마지막 항목이거나 x가 다음과 같은 경우입니다.

배열에 없습니다. 그러므로

$$\boxed{W}(n) = n$$

알고리즘 분석

□ 평균 사례 시간 복잡도

○ 개요

➢ A(n)은 알고리즘이 기본 작업을 수행하는 횟수의 평균(기대값)으로 정의됩니다.
n.➢ A(n)의 입력 크기에 대한 연산은 알고리즘의 평균 사례 시간 복잡도라고 하며 A(n)의 결정은 다음과 같습니다.
평균 사례 시간 복잡도 분석이라고 합니다.➢ W(n)의 경우와 마찬가지로 T(n)이 존재하면 A(n) = T(n)입니다.➢ A(n)을 계산하려면 n 크기의 가능한 모든 입력에 확률을 할당해야 합니다. 다음과 같은 것이 중요합니다.

사용 가능한 모든 정보를 기반으로 확률을 할당합니다.

○ '순차검색' 예시

➢ 기본 조작 : 배열의 항목을 x와 비교합니다.➢ 입력 크기 : n, 배열의 항목 수입니다.➢ 먼저 x가 S에 있고 S의 항목이 모두 구별되는 경우를 분석합니다.

x 가 다른 어레이 슬롯에있을 가능성보다 한 어레이 슬롯에있을 가능성이 더 높다고 믿을 이유가 없습니다. 이 정보를 바탕으로 $1 \le k \le n$에 대해 x가 k번째 배열 슬롯에 있을 확률은 1/n입니다. x가 k번째 배열 슬롯에 있는 경우 x를 찾기 위해(따라서 루프를 종료하기 위해) 기본 작업이 수행되는 횟수는 k입니다. 이는 평균 시간 복잡도가 다음과 같이 주어진다는 것을 의미합니다.

$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{1}{n}\right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

알고리즘 분석

□ 평균 사례 시간 복잡도(계속)

○ '순차 탐색' 예시 (계속)

➢ 다음으로 x가 배열에 없을 수 있는 경우를 분석합니다. 이 경우를 분석하려면 다음과 같은 몇 가지를 할당해야 합니다.

확률 p를 x가 배열에 있는 이벤트로 변경합니다. x가 배열에 있으면 1에서 n까지의 슬롯 중 하나에 있을 가능성이 동일하다고 다시 가정합니다. x가 k번째 슬롯에 있을 확률은 p/n이고 배열에 없을 확률은 1 − p입니다. x가 k번째 슬롯에 있는 경우 k가 루프를 통과하고 x가 배열에 없는 경우 n이 루프를 통과한다는 것을 기억하십시오. 따라서 평균 시간 복잡도는 다음과 같이 주어집니다.

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{p}{n}\right) + n(1-p)$$

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

➢ p = 1인 경우 이전과 같이 A(n) = (n + 1)/2인 반면 p = 1/2인 경우 A(n) = 3n/4 + 1/4입니다. 이것은 약 3/4을 의미합니다.

배열은 평균적으로 검색됩니다.

알고리즘 분석

□ 최상의 시간 복잡성

○ 개요

➢ B(n)은 알고리즘이 기본 작업을 수행하는 최소 횟수로 정의됩니다.
n의 입력 크기입니다.➢ 따라서 B(n)은 알고리즘의 최적 사례 시간 복잡도라고 하고 B(n)의 결정은 다음과 같습니다.
최상의 시간 복잡도 분석입니다.➢ W(n) 및 A(n)의 경우와
마찬가지로 T(n)이 존재하면 B(n) = T(n)입니다.

○ '순차검색' 예시

➢ 기본 작업: 배열의 항목을 x와 비교합니다.➢ 입력 크기: n, 배열의 항목 수입니다.➢ n이 1≥이기 때문에 루프를 하나 이상 통과해야 하며, x = S[1]인 경우 한 번의 통과가 있습니다

n의 크기에 관계없이 루프를 통해. 그러므로

$$B(n) = 1$$

알고리즘 분석

□ 복잡도 함수

- ○ 일반적으로,
  - ➢ 복잡성 함수는 양의 정수를 음이 아닌 실수에 매핑하는 모든 함수가 될 수 있습니다.
    특정 알고리즘에 대한 시간 복잡도 또는 메모리 복잡성을 언급하지 않을 때 일반적으로 f(n) 및 g(n)과 같은 표준 함수 표기법을 사용하여 복잡성 함수를 나타냅니다.

$$f(n) = n$$
$$f(n) = n^2$$
$$f(n) = \lg n$$
$$f(n) = 3n^2 + 4n$$

알고리즘 분석

□ 다른 지침

○ 오버헤드 지침

➢ 루프 이전의 초기화 명령어와 같은.➢ 이러한 명령어가 실행되는 횟수는 입력 크기에 따라 증가하지 않습니다.

○ 제어 지시

➢ 루프를 제어하기 위해 인덱스를 증가시키는 것과 같은.➢ 이러한 명령어가 실행되는 횟수는 입력 크기에 따라 증가합니다.

다음과 같은 모든 경우의 시간 복잡도를 가진 동일한 문제에 대해 두 개의 알고리즘이 있다고 가정합니다 : 첫 번째 알고리즘에 대해 n, 두 번째 알고리즘에 대해 n2. 첫 번째 알고리즘이 더 효율적입니다. 그러나 주어진 컴퓨터가 첫 번째 알고리즘에서 기본 작업을 한 번 처리하는 데 걸리는 시간이 두 번째 알고리즘에서 기본 작업을 한 번 처리하는 데 걸리는 시간보다 1,000배 더 오래 걸린다고 가정합니다. "프로세스"는 제어 명령을 실행하는 데 걸리는 시간을 포함한다는 것을 의미합니다. 따라서, t가 두 번째 알고리즘에서 기본 연산을 한 번 처리하는 데 필요한 시간이라면, 1,000t는 첫 번째 알고리즘에서 기본 연산을 한 번 처리하는 데 필요한 시간이다. 단순화를 위해 오버헤드 명령을 실행하는 데 걸리는 시간이 두 알고리즘 모두에서 무시할 수 있다고 가정해 보겠습니다. 즉, 컴퓨터가 n 크기의 인스턴스를 처리하는 데 걸리는 시간은 첫 번째 알고리즘의 경우 n× 1,000t이고 두 번째 알고리즘의 경우 n2 × t입니다. 첫 번째 알고리즘이 더 효율적인 경우를 결정하기 위해 다음 부등식을 해결해야 합니다.

$$n^2 \times t > n \times 1{,}000t.$$

알고리즘 분석

□ 정확성 분석

○ 지원자격

- 알고리즘이 실제로 수행한다는 증거를 개발하여 알고리즘의 정확성을 분석할 수 있습니다. 그것이해야 할 일.

주문

주문

□ 개요

○ 선형 시간 알고리즘

➢ 예: n 및 100n➢ 시간 복잡도는 입력 크기
n에서 선형입니다.

○ 2차 시간 알고리즘

➢ 예: n2 및 0.01n2➢ 시간 복잡도는 입력 크기 n에서 2차입니다.

○ 선형 시간 알고리즘이 2차 시간 알고리즘보다 우수함

"모든 선형 시간 알고리즘은 결국 어떤 2차 시간 알고리즘보다 더 효율적입니다.", "알고리즘의 이론적 분석에서, 우리는 최종 동작에 관심이 있습니다."다음으로, 알고리즘이 최종 동작에 따라 어떻게 그룹화될 수 있는지 보여줄 것입니다."이런 식으로 한 알고리즘의 최종 동작이 다른 알고리즘의 동작보다 더 나은지 여부를 쉽게 결정할 수 있습니다.

주문

□ 주문에 대한 직관적인 소개

○ 기능 유형

➢ 순수 2차 함수

- 예: 5n2 및 5n2 + 100
- 선형 항이 포함되어 있지 않기 때문입니다.

➢ 2차 시간 알고리즘

- 예: 0.1n2 + n + 100(완전 2차)
- 선형 항을 포함하기 때문입니다.

➢ 2차 항 이외의 다른 값은 결국 와 비교하여 중요하지 않게 됩니다.
2차 항의 값입니다.

n	0.1n2	0.1n2 + n + 100
1010		120
2040		160
50	250400	
100	1,0001,200	
1,000	100,000101,100	

➢ 따라서 함수가 순수 2차 함수는 아니지만 순수한 함수로 분류할 수 있습니다.
2차 함수.

- 직관적으로, 복잡성 함수를 분류할 때 항상 낮은 차수의 항을 버릴 수 있어야 하는 것 같습니다.

주문

□ 질서에 대한 직관적 소개 (계속)

○ 복잡성 범주

➢ 예

- 순수 3차 함수로 0.1n3 + 10n2 + 5n + 25 분류
- 순수 2차 함수로 분류할 수 있는 모든 복잡성 함수의 집합을 Θ(n2)라고 하며, 여기서 Θ는 그리스 대문자 "theta"입니다.
- 함수가 집합 Θ(n2)의 멤버인 경우 함수가 n2의 차수라고 합니다.

$$g(n) = 5n^2 + 100n + 20 \in \Theta(n^2)$$

- 교환 정렬 알고리즘의 경우 T(n) = n(n - 1) / 2

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

- 알고리즘의 시간 복잡도가 Θ(n2) 단위인 경우 해당 알고리즘을 2차 시간 알고리즘 또는 Θ(n2) 알고리즘이라고 합니다.

주문

□ 질서에 대한 직관적 소개 (계속)

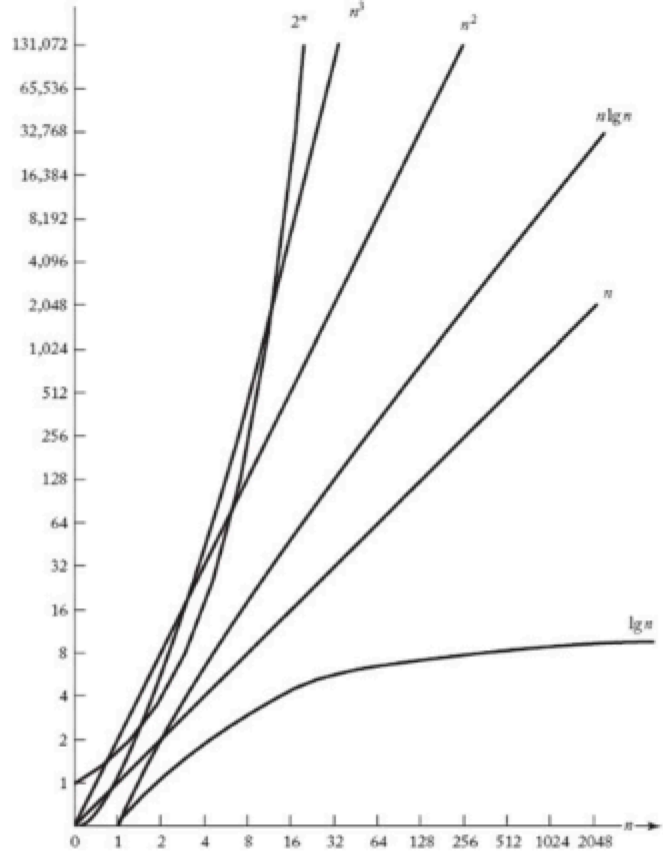
○ 복잡성 범주 (계속)

➢ 가장 일반적인 복잡성 범주

- Θ(lg n), Θ(n), Θ(nlg n), Θ(n2), Θ(n3), Θ(2n)
- 이 순서에서 f(n)이 g(n)을 포함하는 범주의 왼쪽에 있는 범주에 있으면 f(n)은 결국 그래프에서 g(n) 아래에 있습니다.

➢ 가상 알고리즘

- 100n 및 0.01n2 또는 Θ(n) 및 Θ(n2)



주문

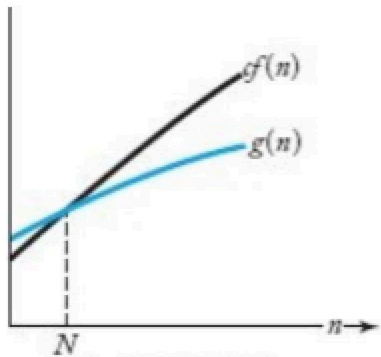
□ Big O 표기법

○ 정의

주어진 복잡성 함수 f(n)에 대해 O(f(n))는 복잡성 함수 g(n)의 집합이며, 이에 대해 양의 실수 상수 c와 음이 아닌 정수 N이 존재하므로 모든 n≥ N에 대해,

$$g(n) \le c \times f(n)$$

➢ g(n)이 O(f(n))∈ 경우, g(n)은 f(n)의 큰 O라고 말합니다. g(n)은 해당 그림에서 cf(n) 위에서 시작하지만 결국에는 cf(n) 아래에 속하여 그 자리에 머뭅니다.



(a) $g(n) \in O(f(n))$

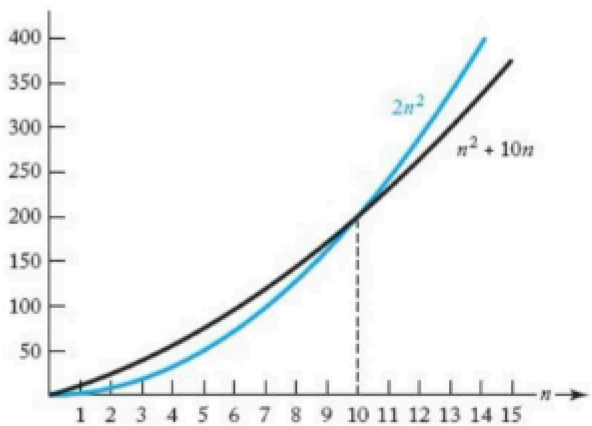
주문

□ Big O 표기법 (계속)

○ 예시

➢ g (n) = n2 + 10n 및 f (n) = 2n2 , n ≥ 10 °C 예를 들어 g (n)이 O (n2)에 있으면 결국 g (n)은 순수 2 차 함수 cn2 아래에 있습니다.
그래프입니다. 이것은 g(n)이 일부 알고리즘의 시간 복잡도인 경우 결국 알고리즘의 실행 시간은 적어도 2차만큼 빠르다는 것을 의미합니다. 분석의 목적을 위해, 우리는 결국 g(n)이 적어도 순수 2차 함수만큼 좋다고 말할 수 있다.""Big O"(그리고 곧 소개될 다른 개념들)는 점근적 거동을 설명한다고 한다

그들은 궁극적인 행동에만 관심이 있기 때문에 함수의 것입니다. 우리는 "big O"가 함수에 아나점스틱적 상한을 둔다고 말합니다.



주문

□ Big O 표기법 (계속)

○ 예시 (계속)

우리는 5n2 ∈ O (n2)를 보여줍니다. 왜냐하면, n ≥ 0

$$5n^2 \leq 5n^2$$

원하는 결과를 얻기 위해 c = 5 및 N = 0을 사용할 수 있습니다

지정된 교환 정렬 알고리즘입니다.

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

왜냐하면, n ≥ 0

$$\frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2}n^2$$

C = 1/2 및 N = 0을 사용하여 T n ∈ O(n2)

주문

□ Big O 표기법 (계속)

○ 예시 (계속)

우리는 n2 + 10n ∈ O (n2)를 보여줍니다. 왜냐하면, n ≥ 1

$$n^2 + 10n \le n^2 + 10n^2 = 11n^2$$

원하는 결과를 얻기 위해 c = 11 및 N = 1을 사용할 수 있습니다

➢ 이 마지막 예제의 목적은 "big O" 내부의 함수가 다음 중 하나일 필요가 없음을 보여주는 것입니다. 간단한 함수.

우리는 n2 ∈ O (n2 + 10n)임을 보여줍니다. 왜냐하면, n ≥ 0

$$n^2 \le 1 \times (n^2 + 10n)$$

원하는 결과를 얻기 위해 c = 1 및 N = 0을 사용할 수 있습니다

➢ 복잡도 함수는 O (n2)에있을 2 차 항을 가질 필요가 없습니다.

- 결국에는 그래프의 순수한 2차 함수 아래에 있어야 합니다.
- 따라서 모든 로그 또는 선형 복잡도 함수는 O(n2)에 있습니다. 마찬가지로 모든 로그, 선형 또는 2차 복잡도 함수는 O(n3) 등에 있습니다.

우리는 n ∈ O (n 2)를 보여줍니다. 왜냐하면, n ≥ 1

$$n \le 1 \times n^2$$

원하는 결과를 얻기 위해 c = 1 및 N = 1을 사용할 수 있습니다

주문

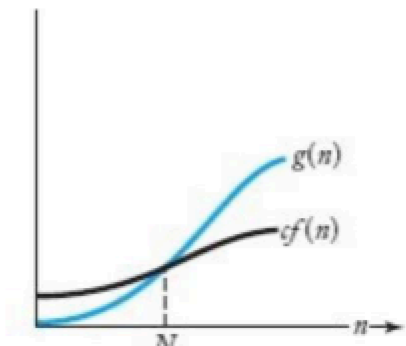
□ 오메가 Ω 표기법

○ 정의

주어진 복잡성 함수 f(n)에 대해 Ω(f(n))은 복잡성 함수 g(n)의 집합이며, 여기에는 양의 실수 상수 c와 음이 아닌 정수 N이 존재하므로 모든 n ≥ N에 대해

$$g(n) \ge c \times f(n)$$

➢ 그 그림에서 g(n)은 cf(n) 아래에서 시작하지만 결국에는 cf(n) 아래로 떨어지고 거기에 머무릅니다.



(b) $g(n) \in \Omega(f(n))$

주문

□ Omega Ω 표기법 (계속)

○ 예시

우리는 5n2 ∈ Ω(n2)를 보여줍니다. 왜냐하면, n ≥ 0

$$5n^2 \geq 1 \times n^2$$

원하는 결과를 얻기 위해 c = 1 및 N = 0을 사용할 수 있습니다

우리는 n2 ∈ Ω(n2 + 10n)을 보여줍니다. 왜냐하면, n ≥ 0

$$n^2 + 10n \geq n^2$$

원하는 결과를 얻기 위해 c = 1 및 N = 0을 사용할 수 있습니다

주문

□ Omega Ω 표기법 (계속)

○ 예시

지정된 교환 정렬 알고리즘입니다.

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$$

왜냐하면, n ≥ 2

$$n - 1 \geq \frac{n}{2}$$

따라서 n ≥ 2

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$$

c = 1/4 및 N = 2를 사용하여 T n ∈ Ω(n2)

➢ 함수가 Ω(n2)에 있으면 결국 함수는 의 순수 2차 함수 위에 있습니다. 그래프.

- 분석을 위해 이것은 결국 적어도 순수한 2차 함수만큼 나쁘다는 것을 의미합니다

주문

□ Omega Ω 표기법 (계속)

○ 예시

지정된 교환 정렬 알고리즘입니다.

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$$

왜냐하면, n ≥ 2

$$n - 1 \geq \frac{n}{2}$$

따라서 n ≥ 2

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$$

c = 1/4 및 N = 2를 사용하여 T n ∈ Ω(n2)

➢ 함수가 Ω(n2)에 있으면 결국 함수는 의 순수 2차 함수 위에 있습니다. 그래프.

- 분석을 위해 이것은 결국 적어도 순수한 2차 함수만큼 나쁘다는 것을 의미합니다

우리는 n3 ∈ Ω(n2)를 보여줍니다. 왜냐하면, n ≥ 1

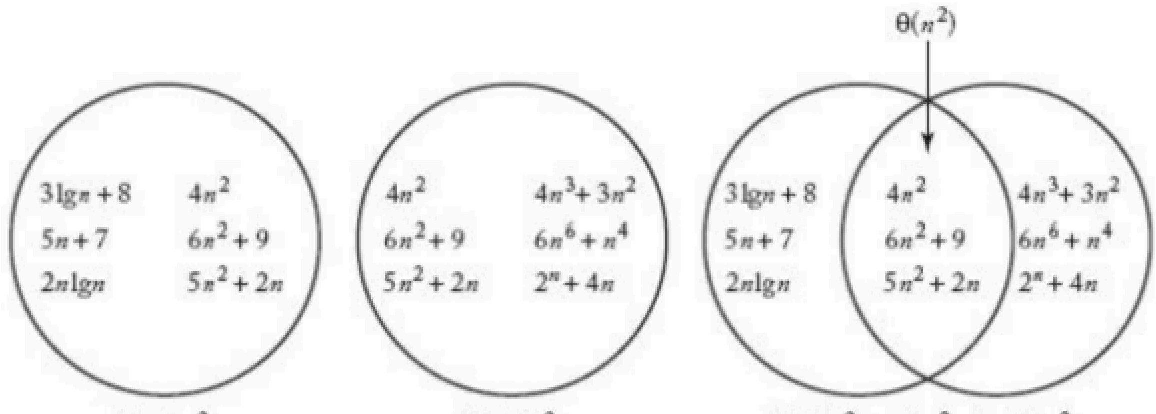
$$n^3 \geq 1 \times n^2$$

원하는 결과를 얻기 위해 c = 1 및 N = 1을 사용할 수 있습니다

주문

□ 세타 Θ 표기법

○ O n2 , Ω n2 , Θ n2



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

주문

□ Theta Θ 표기법 (계속)

○ 정의

주어진 복잡도 함수 f(n)에 대해

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

즉, Θ (f (n))는 모든 n ≥ N에 대해 어떤 양의 실수 상수 c와 음이 아닌 정수 N이 존재하는 복잡성 함수 g (n)의 집합입니다.

$$c \times f(n) \le g(n) \le d \times f(n)$$

➢ g(n)이 Θ(f(n))∈ 경우 g(n)은 f(n)의 차수라고 합니다.

○ 예시

지정된 교환 정렬 알고리즘입니다.

$$T(n) = \frac{n(n-1)}{2} \quad \text{is in both} \quad O(n^2) \quad \text{and} \quad \Omega(n^2)$$

이것은 T (n) ∈ O (n2) ∩ Ω (n2) = Θ (n2)를 의미합니다.

□ Theta Θ 표기법 (계속)

○ 예시 (계속)

n이 Ω(n2)에 없다는 것을 증명합니다. n ∈ Ω(n2) 가 n ≥ N에 대해 다음과 같이 양의 상수 c와 음이 아닌 정수 N이 존재한다고 가정한다는 것을 의미한다고 가정합니다.

$$n \geq cn^2$$

이 부등식의 양쪽을 cn으로 나누면 n ≥ N에 대해 다음과 같습니다.

$$\frac{1}{c} \geq n$$

그러나 n > 1/c에 대해 이 부등식은 성립할 수 없으며, 이는 모든 ≥ N에 대해 성립할 수 없음을 의미합니다. 이 모순은 n이 Ω(n2)에 없다는 것을 증명합니다.

주문

□ 스몰 o 표기법

○ 정의

주어진 복잡성 함수 f(n)에 대해 o(f(n))는 다음을 충족하는 모든 복잡성 함수 g(n)의 집합입니다.모든 양의 실수 상수 c에 대해 음이 아닌 integerN이 존재하며, 모든 n에 대해 N≥

$$g(n) \le c \times f(n)$$

○ Big O와 비교

➢ "big O"는 경계가 유지되는 실제 양의 상수 c가 있어야 함을 의미합니다. 이 정의에 따르면 경계는 모든 실제 양수 상수 C에 대해 유지되어야 합니다.

- 한계는 모든 양수 c에 대해 성립하기 때문에 임의로 작은 c에 대해 성립합니다. 예를 들어, g(n)이 o(f(n))∈ 경우 n > N에 대해 다음과 같은 anN이 있습니다.

$$g(n) \le 0.00001 \times f(n)$$

- n이 커짐에 따라 g(n)이 f(n)에 비해 중요하지 않게 되는 것을 볼 수 있습니다. 분석을 위해 g (n)이 ino (f (n))이면 g (n)은 결국 f (n)과 같은 함수보다 훨씬 낫습니다.

주문

□ 스몰 o 표기법 (계속)

○ 예시

우리는 그것을 보여줍니다

n∈ o(n2)입니다.

c > 0이 주어진다고 합시다. n ≥ N에 대해 다음과 같은 N을 찾아야 합니다.

N ≤ CN2

이 부등식의 양쪽을 cn으로 나누면 다음과 같습니다.

1c ≤ n.따라서 N ≥ 1/c를 선택하는 것으로 충
분합니다.

N의 값은 상수 c에 따라 달라집니다. 예를 들어, c = 0.00001인 경우 N이 최소 100,000과 같아야 합니다.
즉,

N ≤ 0.00001N2

주문

□ 스몰 o 표기법 (계속)

○ 예시 (계속)

우리는 n이 o(5n)에 없다는 것을 보여줍니다. 우리는 이것을 보여주기 위해 모순에 의한 증거를 사용할 것입니다. c=1/6 n이 o(5n)∈이면 n ≥ N에 대해 다음과 같은 N이 존재해야 합니다.

$$n \le \frac{}{6}5n = \frac{}{6}n$$

이 모순은 n이 o(5n)에 없다는 것을 증명합니다.

주문

□ 스몰 o 표기법 (계속)

○ 정리 1.2

g(n)이 o(f(n))∈

$$g(n) \in O(f(n)) - \Omega(f(n))$$

즉, g (n)은 O (f (n))에 있지만 Ω (f (n))에는 없습니다. 증명: g(n)∈ o(f(n)이기 때문에, 모든 양의 실수 상수 c에 대해 N이 존재하며, 모든 n≥ N에 대해,

$$g\ n \leq c \times f(n)$$

이는 경계가 일부 c에 대해 확실히 유지됨을 의미합니다. 그러므로

$$g\ n \in O\ f\ n\ .$$

우리는 g(n)이 Ω(f(n))에 있지 않다는 것을 모순에 의한 증명을 사용하여 보여줄 것입니다. g(n) ∈ Ω(f(n))이면 모든 n ≥ N1에 대해 0 > 일부 실수 상수가 존재하고 N1이 존재합니다.

$$g\ n \geq c \times f\ n\ .$$

그러나 g (n)이 o (f (n))∈하기 때문에 모든 n ≥ N2,g n ≤ c2 × f n . 두 부등식은 N1과 N2보다 큰 모든 n에 대해 유지되어야 합니다. 이것은 g(n)이 Ω(f(n))에 있을 수 없다는 것을 증명합니다.

주문

□ 스몰 o 표기법 (계속)

○ 예시

함수를 고려하십시오.

$$g(n) = \begin{cases} n & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$$

$$g(n) \in O(n) - \Omega(n) \quad \text{but that} \quad g(n) \text{ is not in } o(n)$$

- 복잡성 함수가 실제 알고리즘의 시간 복잡도를 나타낼 때, 일반적으로 O (f (n)) – Ω (f (n))의 함수는 O (f (n))에있는 것과 동일합니다.