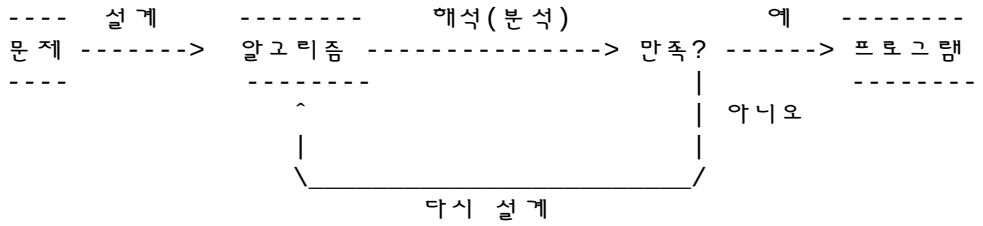
제 1 장

서막

- 학습 목표
  - ▷ 알고리즘의 설계(design) 방법을 배운다.
  - ▷ 알고리즘의 분석(analysis)하여 계산복잡도를 구하는 방법을 배운다.
  - ▷ 문제에 대한 계산복잡도(computational complexity)를 공부한다.
- 프로그램 설계 과정



제 1 절 알고리즘

- **정의 1.1: 알고리즘(Algorithm)** - 문제에 대한 답을 찾기 위해서 계산하는 절차를 알고리즘이라고 한다. 다시 말하면,
  - ▷ 단계별로 주의 깊게 설계된 계산 과정
  - ▷ 입력을 받아서 출력으로 전환시켜주는 일련의 계산 절차 (computational steps)
- 보기
  - ▷ 문제(problem): 전화번호부에서 “홍길동”이라는 사람의 이름 찾기
  - ▷ 알고리즘(algorithm):
    1. 순차검색(sequential search): 첫 쪽부터 홍길동이라는 이름이 나올 때까지 순서대로 찾는다.
    2. 수정된 이진검색(modified binary search): 전화번호부는 가나다 순으로 되어 있으므로 먼저 “ㅎ”이 있을 만한 곳으로 넘겨본 후 앞뒤로 뒤적여 가며 찾는다.
  - ▷ 분석(analysis): 어떤 알고리즘이 더 좋은가?

1.1 문제의 표기

- 문제를 표기하는데 필요한 사항
  - ▷ 문제(problem): 답을 찾고자 주는 질문
  - ▷ 매개변수(parameter): 문제를 설명하는 과정에서 어떤 특정한 값이 지정되어 있지 않은 변수(variable)
  - ▷ 문제의 사례(instance) = 입력(input): 문제에 주어진 매개변수에 어떤 특정 값을 지정한 것
  - ▷ 사례에 대한 해답(solution) = 출력(output): 어떤 사례에 대하여 문제에 의해서 제기된 질문에 대한 답
- 보기: 검색(Searching)
  - ▷ 문제: n개의 수(number)를 가진 리스트 S에 x라는 수가 있는지 결정하시오. 답은 x가 S에 있으면 "예", 그렇지 않으면 "아니오".
  - ▷ 매개변수: S(리스트), n(S 안에 있는 수의 개수), x(찾고자 하는 항목)
  - ▷ 입력의 예: $S = [10, 7, 11, 5, 13, 8]$, $n = 6$, $x = 5$
  - ▷ 출력의 예: "예"

1.2 알고리즘의 표기

- 자연어(영어 또는 한글)
- 프로그래밍언어: C, C++, Java, ML 등
- 의사코드(pseudo-code): 직접 실행할 수 있는 프로그래밍언어는 아니지만 거의 실제 프로그램에 가깝게 계산과정을 표현할 수 있는 언어 (알고리즘은 보통 의사코드로 표현한다) - 여기서는 C/C++에 가까운 의사코드를 사용함

1.3 보기: 검색하기

1.3.1 순차검색(Sequential Search) 알고리즘

- 문제: 크기가 `n`인 배열(array) `S`에 `x`가 있는가?
- 입력(매개변수): (1) 양수 `n`, (2) 배열 `S[1..n]`, (3) 찾고자 하는 항목 `x`
- 출력: `x`가 `S`의 어디에 있는지의 위치. 만약 `x`가 `S`에 없다면 0.
- 알고리즘(자연어): `x`와 같은 항목을 찾을 때까지 `S` 배열에 있는 모든 항목을 차례로 비교한다. 만일 `x`와 같은 항목을 찾으면 `S` 배열 상의 위치를 내주고, `S`에 있는 모든 항목을 검사하고도 찾지 못하면 0을 내준다.
- 알고리즘(의사코드):

```
void seqsearch(int n,                    // 입력 (1)
               const keytype S[],        //      (2)
               keytype x,                //      (3)
               index& location)          // 출력
{
   location = 1;
   while (location <= n && S[location] != x)      // (A)
      location++;
   if (location > n)                              // (B)
      location = 0
}
```

- 조건문의 의미

  - (A): 아직 검사할 항목이 있고, x를 찾지 못했나?
  - (B): 모두 검사하였으나 x를 찾지 못했나?

- 관찰사항: 이 알고리즘으로 어떤 항목(key)을 찾기 위해서 배열 S에 있는 항목을 몇 개나 검색해야 하는가? 이는 찾고자 하는 항목이 어디에 위치하고 있는가에 달려 있겠지만, 최악의 경우, 즉, 찾고자 하는 항목이 S[n]에 위치하고 있거나, 아예 없는 경우에는 최소한 n개는 검색해야 한다.

- 좀 더 빠르게 찾을 수는 없을까? 사실 더 이상 빨리 찾을 수 있는 알고리즘은 존재하지 않는다. 왜냐하면 배열 S에 들어있는 항목들에 대한 정보가 전혀 없는 상황에서는 모든 항목을 검색하지 않고 임의의 항목 x를 항상 찾을 수 있다는 보장은 없기 때문이다. 그러나 다음 절에서와 같이 만약 배열 S가 이미 오름차순으로 정렬(sorting) 되어 있는 경우에는 좀 더 빠른 시간 안에 검색할 수가 있다.

1.3.2 이분검색(Binary Search) 알고리즘

- 문제: 크기가 n인 정렬된 배열(array) S에 x가 있는가?
- 입력(매개변수): (1) 양수 n, (2) 배열 S[1..n], (3) 찾고자 하는 항목 x
- 출력: x가 S의 어디에 있는지의 위치. 만약 x가 S에 없다면 0.
- 알고리즘:

```
void binsearch (int n,                  // 입력 (1)
                const keytype S[],      //      (2)
                keytype x,              //      (3)
                index& location)        // 출력
{
   index low, high, mid;

   low = 1; high = n;
   location = 0;
   while (low <= high && location == 0) {       // (A)
      mid = (low + high) / 2;
            // 정수 나누셈 (나머지 버림)
      if (x == S[mid])
         location = mid;
      else if (x < S[mid])
         high = mid - 1;
      else
         low = mid + 1;
   }
}
```

- 관찰사항: 조건문의 의미
  - (A): 아직 검사할 항목이 있고, x를 찾지 못했나?
- 관찰사항: 이 알고리즘으로 어떤 항목(key)을 찾기 위해서 배열 S에 있는 항목을 몇 개나 검색해야 하는가? 이 경우 while문을 수행할 때마다 검색 대상의 총 크기가 반 씩 감소하기 때문에 최악의 경우라도 $\lg n + 1$개만 검색하면 된다.

1.3.3 비교: 순차검색 대 이분검색

- 아래 표에서 볼 수 있듯이, 배열의 크기가 커질수록 두 알고리즘의 검색 횟수의 차이는 크게 벌어진다.

배열의 크기	순차검색	이분검색	
n	n	lg n + 1	최악의 경우
128	128	8	
1,024	1,024	11	
1,048,576	1,048,576	21	
4,294,967,296	4,294,967,296	33	

1.4 보기: n번째 피보나찌 수 구하기

1.4.1 피보나찌(Fibonacci) 수열의 정의

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

예: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1264, \cdots$

1.4.2 되부름(recursive) 방법

- 문제: n번째 피보나찌 수를 구하시오.
- 입력: 양수 n
- 출력: n번째 피보나찌 수.

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

- 관찰사항: 위의 알고리즘은 수행속도가 매우 느리다(inefficient). 왜냐하면, 같은 피보나찌 수를 중복하여 계산하기 때문이다. 예를 들면, `fib(5)`를 계산할 때 `fib(2)`를 3번 중복하여 계산한다. `fib(5)`를 계산하는데 `fib` 함수를 부르는 횟수를 계산하기 위해서는 교재 14쪽의 그림 1.2와 같이 되부름 나무(recursion tree)를 그려서 그 마디(node)의 개수를 세어보면 된다. $T(n)$을 `fib(n)`을 계산하기 위해서 `fib` 함수를 부르는 횟수, 즉, 되부름 나무 상의 마디의 수라고 하면, 다음과 같이 계산할 수 있다.

$$\begin{aligned} T(0) &= 1; \\ T(1) &= 1; \\ T(n) &= T(n-1) + T(n-2) + 1 && \text{for } n \geq 2 \\ &> 2 \times T(n-2) && \text{since } T(n-1) > T(n-2) \\ &> 2^2 \times T(n-4) \\ &> 2^3 \times T(n-6) \\ &\cdots \\ &> 2^{n/2} \times T(0) \\ &= 2^{n/2} \end{aligned}$$

- **정리 1.1:** 위의 알고리즘으로 구성한 되부름나무의 마디의 수를 $T(n)$이라고 하면, $n \geq 2$인 모든 n에 대해서 $T(n) > 2^{n/2}$이다.

증명: n에 대해서 수학적 귀납법(mathematical induction)으로 증명

귀납출발점(induction base):

$$T(2) = T(1) + T(0) + 1 = 3 > 2 = 2^{2/2}$$

$$T(3) = T(2) + T(1) + 1 = 5 > 2.83 \approx 2^{3/2}$$

귀납가정(induction hypothesis): $2 \leq m < n$인 모든 m에 대해서 $T(m) > 2^{m/2}$라 가정.

귀납절차(induction step): $T(n) > 2^{n/2}$임을 보여야 한다.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{귀납가정에 의하여} \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} \\ &= 2 \times 2^{(n/2)-1} \\ &= 2^{n/2} \end{aligned}$$

1.4.3 반복적(iteration) 방법

- 문제: n번째 피보나찌 수를 구하시오.
- 입력: 양수 `n`
- 출력: n번째 피보나찌 수.

```
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];
}
```

- 관찰사항: 반복적 알고리즘은 수행속도가 빠르다(efficient). 왜냐하면, 되부름 알고리즘과는 달리 중복하여 계산하는 경우가 없기 때문이다. 계산하는 항의 총 갯수는 $T(n) = n+1$이 된다. 즉, `fib2(n)`을 계산하기 위해서는 `f[0]`부터 `f[n]`까지 한번씩만 계산하면 된다.
- 관찰사항: 교재 16쪽의 표 1.2에 위의 두 알고리즘의 수행시간을 비교해 놓았다.

제 2 절 알고리즘 분석: 시간복잡도 분석

- 알고리즘 분석(analysis of algorithms)이란? - 입력 크기에 따라서 기본동작이 몇번 수행되는지를 결정하는 절차 [= 시간복잡도(time complexity) 분석]

2.1 시간복잡도를 표현하는데 필요한 척도

- 기본 동작(basic operation): 비교문, 지정문(assignment statement) 등
- 입력 크기(input size, parameter): 배열의 크기, 리스트의 길이, 행렬에서 행(row)와 열(column)의 크기, 나무구조(tree)에서 마디(vertex)의 수와 가지(edge)의 수 등

2.2 분석 방법의 종류

- 모든 경우를 고려한 분석(every-case analysis)
  - ▷ 모든 경우를 고려하여 분석한 복잡도(complexity)는 다음의 성질을 가진다. "입력의 크기와는 관련이 있고(dependent), 입력의 값과는 관련이 없다(independent)."
  - ▷ 기본적인 동작이 수행되는 횟수는 입력의 값에 상관없이 항상 일정하다.
- 최악의 경우를 고려한 분석(worst-case analysis)
  - ▷ 최악의 경우를 고려하여 분석한 복잡도는 다음의 성질을 가진다. "입력의 크기와도 관련이 있고(dependent), 입력의 값과도 관련이 있다(dependent)."
  - ▷ 기본적인 동작이 수행되는 횟수가 최대인 경우를 택한다.
- 평균적인 경우를 고려한 분석(average-case analysis)
  - ▷ 평균의 경우를 고려하여 분석한 복잡도는 "모든 입력에 대해서 알고리즘이 기본 동작을 수행하는 횟수의 평균(기대치)이다."
  - ▷ 각 입력에 대해서 확률을 할당할 수도 있다.
  - ▷ 최악의 경우를 고려한 시간복잡도 분석보다 대체로 계산하기가 복잡하다.
- 최선의 경우를 고려한 분석(best-case analysis)
  - ▷ 최선의 경우를 고려하여 분석한 복잡도는 모든 입력 중에서 알고리즘이 최소로 기본동작을 수행하는 횟수이다.

2.3 보기: 배열(Array) 덧셈

2.3.1 알고리즘

- 문제: 크기가 `n`인 배열 `S`의 모든 수를 더하시오.
- 입력: 양수 `n`, 배열 `S[1..n]`
- 출력: 배열 `S`의 모든 수의 합

```
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;
}
```

2.3.2 시간복잡도 분석 I

- 기본동작: 덧셈
- 입력의 크기: 배열의 크기 n
- 모든 경우를 고려한 시간복잡도 분석: 배열에 어떤 수가 있는지에 상관없이 for-맴돌이(loop)가 n번 반복된다. 그리고 각 맴돌이 마다 덧셈이 1회 수행된다. 따라서 n에 대해서 덧셈이 수행되는 총 횟수는 $T(n) = n$이다.

2.3.3 시간복잡도 분석 II

- 기본동작: 지정문 - for-맴돌이의 첨자 대입문 포함
- 입력의 크기: 배열의 크기 n
- 모든 경우를 고려한 시간복잡도 분석: 배열에 어떤 수가 있는지에 상관없이 for-맴돌이가 n번 반복된다. 따라서 지정문이 $T(n) = n + n + 1$번 수행된다.

2.3.4 관찰사항

- 위에서 기본동작을 다르게 봄으로서 시간복잡도가 다르게 나왔다. 그러나 사실은 둘 다 같은 복잡도 카테고리에 속한다고 봐도 된다. 자세한 사항은 뒤에 차수(order)를 배울때 다루게 된다.

2.4 보기: 교환정렬(Exchange Sort)

2.4.1 알고리즘

- 문제: 비내림차순(nondecreasing order)으로 `n`개의 키(key)를 정렬
- 입력: 양수 `n`, 배열 `S[1..n]`
- 출력: 비내림차순으로 정렬된 배열 `S`

```
void exchangesort (int n, keytype S[])
{
   index i,j;

   for (i = 1; i <= n-1; i++)
      for (j = i+1; j <= n; j++)
         if (S[j] < S[i])
            exchange S[i] and S[j];
}
```

2.4.2 시간복잡도 분석 I

- 기본동작: 조건문 - `S[j]`와 `S[i]`를 비교하는 동작
- 입력의 크기: 정렬할 항목의 수 n
- 모든 경우를 고려한 시간복잡도 분석: j-맴돌이가 수행될 때마다 어떠한 경우에도 조건문이 1번씩 수행된다. 따라서 조건문이 수행되는 총 횟수는 다음과 같이 구할 수 있다.

$i = 1$ 일때, j-맴돌이 $n-1$번 수행
$i = 2$ 일때, j-맴돌이 $n-2$번 수행
$i = 3$ 일때, j-맴돌이 $n-3$번 수행
$\cdots$
$i = n-1$ 일때, j-맴돌이 1번 수행
따라서 $T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = (n-1)n/2$

2.4.3 시간복잡도 분석 II

- 기본동작: 교환하는 동작 - `exchange S[i] and S[j]`
- 입력의 크기: 정렬할 항목의 수 n
- 최악의 경우를 고려한 시간복잡도 분석: 조건문의 결과에 따라서 교환하는 동작이 수행될 수도 있고, 그렇지 않을 수도 있다. 따라서 최악의 경우, 즉, 조건문이 항상 참값을 가지는 경우(입력 배열이 역순으로 정렬되어 있는 경우)를 고려해 보면 위의 분석과 마찬가지로 $T(n) = (n-1)n/2$이 된다.

2.5 보기: 순차검색

- 순차검색(sequential search) 알고리즘의 경우 입력 배열의 값에 따라서 검색하는 횟수가 다르므로, 모든 경우를 고려한 해석은 불가능하다.

2.5.1 시간복잡도 분석 I

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 최악의 경우를 고려한 시간복잡도 분석: x가 배열의 마지막 항목이거나 배열 안에 없을 경우에는 기본동작이 n번 수행된다. 따라서 $W(n) = n$

2.5.2 시간복잡도 분석 II

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 평균의 경우를 고려한 시간복잡도 분석: 배열의 항목이 모두 다르다고 가정하고, 다음과 같이 시간복잡도를 계산한다.
  - ▷ 경우 1: (x가 배열 S 안에 있는 경우)
    - ⋆ $1 \le k \le n$에 대해서, x가 배열의 k번째 있을 확률 $= 1/n$
    - ⋆ 만약 x가 배열의 k번째 있다면, k를 찾기 위해서 수행하는 기본 동작의 횟수 $= k$
    - ⋆ 따라서,

$$\begin{aligned} A(n) &= \sum_{k=1}^{n} (k \times \frac{1}{n}) \\ &= \frac{1}{n} \times \sum_{k=1}^{n} k \\ &= \frac{1}{n} \times \frac{n(n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

  - ▷ 경우 2: (x가 배열 S 안에 없는 경우)
    - ⋆ x가 배열 S 안에 있을 확률을 p라고 하면,
      - · x가 배열의 k번째 있을 확률 $= p/n$
      - · x가 배열의 k번째 없을 확률 $= 1-p$
    - ⋆ 따라서,

$$\begin{aligned} A(n) &= \sum_{k=1}^{n} (k \times \frac{p}{n}) + n(1-p) \\ &= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) \\ &= n(1-\frac{p}{2}) + \frac{p}{2} \end{aligned}$$

2.5.3 시간복잡도 분석 III

- 기본동작: 배열의 항목과 찾는 키 x와를 비교하는 동작
- 입력의 크기: 배열 안에 있는 항목의 수 n
- 최선의 경우를 고려한 시간복잡도 분석: x가 $S[1]$일때, 입력의 크기에 상관없이 기본동작이 한번만 수행된다. 따라서 $B(n) = 1$

2.5.4 관찰사항

- 최악의 경우보다 평균의 경우의 분석이 직관적으로 더 이치에 맞아보인다. 그러나 위의 경우 두 분석의 결과가 모두 같은 복잡도 카테고리 $\Theta(n)$에 속한다. 사실 일반적으로 거의 대부분의 경우 두 분석 방법에 따른 결과의 차이는 없다. (물론 예외는 있다.) 따라서 계산하기가 훨씬 간단한 최악의 경우를 이용하는 것이 일반적이다. 그리고 최선의 경우를 선택하는 것은 비현실적이다.

2.6 정확도 분석(Analysis of Correctness)

1. 알고리즘이 원래 의도한 대로 실제로 수행이 되는지를 증명하는 절차
2. 정확한 알고리즘이란 무었인가? - "어떠한 입력에 대해서도 맞는 답을 출력하면서 멈추는 알고리즘"
3. 정확하지 않은 알고리즘이란? - "어떤 입력에 대해서 멈추지 않거나 또는 틀린 답을 출력하면서 멈추는 알고리즘"

제 3 절 차수

- 차수(order)는 알고리즘의 복잡도(complexity)를 표시하기 위하여 쓰는 일종의 표기법이다.
- 예를 들면, $\Theta(n^2)$은 2차함수(quadratic function)로 분류되는 모든 복잡도 함수의 집합을 나타낸다. 다시말해서, $\Theta(n^2)$에 속해있는 모든 복잡도 함수는 차수가 n^2이라고 대표해서 부를 수 있다. 따라서 $5n^2$, $5n^2+100$, $0.1n^2+n+100$ 모두 차수가 n^2이다. (교재 27쪽의 표 1.3 참조)
- 이때 낮은 차수의 항(low-order term)은 무시해도 상관없다. 왜냐하면 가장 높은 차수의 항이 전체 항의 성질을 지배하기 때문이다.

3.1 복잡도 카테고리

- $\Theta(\lg n)$
- $\Theta(n)$: 선형(linear)
- $\Theta(n \lg n)$
- $\Theta(n^2)$: 2차(quadratic)
- $\Theta(n^3)$: 3차(cubic)
- $\Theta(2^n)$: 지수형(exponential)
- $\Theta(n!)$

1. 교재 28쪽의 그림 1.3과 29쪽의 표 1.4를 보시오.
2. 같은 카테고리에 속한 어떤 함수도 사실은 그 카테고리를 대표할 수 있다. 그러나 편의상 가장 간단히 표시할 수 있는 함수로 그 카테고리를 표현하는 것이 통례이다.

3.2 큰(Big)O 표기법

- **정의 2: 점근적 상한(Asymptotic Upper Bound)**
  주어진 복잡도 함수 $f(n)$에 대해서 $g(n) \in O(f(n))$ 이면 다음을 만족한다: $n \geq N$인 모든 정수 n에 대해서 $g(n) \leq c \times f(n)$이 성립하는 실수 $c > 0$와 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(a)를 보시오.)
- $g(n) \in O(f(n))$은 "$g(n)$은 $f(n)$의 큰 오(big O)"라고 부른다.

- 어떤 함수 $g(n)$이 $O(n^2)$에 속한다는 말은, 그 함수는 궁극에 가서는 (즉, 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차함수 cn^2의 값보다는 **작은** 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 **아래**에 위치) 다시말해서, 그 함수 $g(n)$은 어떤 2차함수 cn^2 보다는 궁극적으로 **좋다**고 (기울기가 **낮다**고) 말할 수 있다.

- 어떤 알고리즘의 시간복잡도가 $O(f(n))$이라면, 입력의 크기 n에 대해서 이 알고리즘의 수행시간은 **아무리 늦어도** $f(n)$은 된다. ($f(n)$이 **상한**이다.) 다시말하면, 이 알고리즘의 수행시간은 $f(n)$보다 절대로 더 느릴 수는 없다는 말이다.

- 보기 1.3: $n^2+10n \in O(n^2)$임을 보이시오.
(1) $n \geq 10$인 모든 정수 n에 대해서 $n^2+10n \leq 2n^2$이 성립한다. 그러므로, $c=2$와 $N=10$을 선택하면, "큰 O"의 정의에 의해서 $n^2+10n \in O(n^2)$이라고 결론지을 수 있다. (교재 30쪽의 그림 1.5를 보시오.)
(2) $n \geq 1$인 모든 정수 n에 대해서 $n^2+10n \leq n^2+10n^2 = 11n^2$이 성립한다. 그러므로, $c=11$와 $N=1$을 선택하면, "큰 O"의 정의에 의해서 $n^2+10n \in O(n^2)$이라고 결론지을 수 있다.

- 보기 1.4: $5n^2 \in O(n^2)$임을 보이시오.
$c=5$와 $N=0$을 선택하면, $n \geq 0$인 모든 정수 n에 대해서 $5n^2 \leq 5n^2$이 성립한다.

- 보기 1.5: $T(n) = \frac{n(n-1)}{2}$은 어떻게 될까?
$n \geq 0$인 모든 정수 n에 대해서 $\frac{n(n-1)}{2} \leq \frac{n^2}{2}$이 성립한다. 그러므로, $c=\frac{1}{2}$와 $N=0$을 선택하면, $T(n) \in O(n^2)$이라고 결론지을 수 있다.

- 보기 1.6: $n^2 \in O(n^2+10n)$임을 보이시오.
$n \geq 0$인 모든 정수 n에 대해서, $n^2 \leq 1 \times (n^2+10n)$이 성립한다. 그러므로, $c=1$와 $N=0$을 선택하면, $n^2 \in O(n^2+10n)$이라고 결론지을 수 있다.

- 보기 1.7: $n \in O(n^2)$임을 보이시오.
$n \geq 1$인 모든 정수 n에 대해서, $n \leq 1 \times n^2$이 성립한다. 그러므로, $c=1$와 $N=1$을 선택하면, $n \in O(n^2)$이라고 결론지을 수 있다.

- 보기 1.8: $n^3 \in O(n^2)$이 아님을 보이시오.
$n \geq N$인 모든 n에 대해서 $n^3 \leq c \times n^2$이 성립하는 c와 N 값은 존재하지 않는다. 즉, 양변을 n^2으로 나누면, $n \leq c$가 되는데 c를 아무리 크게 잡더라도 그 보다 더 큰 n이 존재한다.

- 참고: 교재 32쪽의 그림 1.6(a)를 보시오.

3.3 Ω 표기법

- **정의 1.3: 점근적 하한(Asymptotic Lower Bound)**
주어진 복잡도 함수 $f(n)$에 대해서 $g(n) \in \Omega(f(n))$이면 다음을 만족한다: $n \geq N$인 모든 정수 n에 대해서 $g(n) \geq c \times f(n)$이 성립하는 실수 $c>0$와 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(b)를 보시오.)

- $g(n) \in \Omega(f(n))$은 "$g(n)$은 $f(n)$의 오메가(omega)"라고 부른다.

- 어떤 함수 $g(n)$이 $\Omega(n^2)$에 속한다는 말은, 그 함수는 궁극에 가서는 (즉, 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차함수 cn^2의 값보다는 **큰** 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 **위**에 위치) 다시말해서, 그 함수 $g(n)$은 어떤 2차함수 cn^2 보다는 궁극적으로 **나쁘다**고 (기울기가 **높다**고) 말할 수 있다.

- 어떤 알고리즘의 시간복잡도가 $\Omega(f(n))$이라면, 입력의 크기 n에 대해서 이 알고리즘의 수행시간은 **아무리 빨라도** $f(n)$밖에 되지 않는다. ($f(n)$이 **하한**이다.) 다시말하면, 이 알고리즘의 수행시간은 $f(n)$보다 절대로 더 빠를 수는 없다는 말이다.

- 보기 1.9: $n^2+10n \in \Omega(n^2)$임을 보이시오.
$n \geq 0$인 모든 정수 n에 대해서 $n^2+10n \geq n^2$이 성립한다. 그러므로, $c=1$와 $N=0$을 선택하면, $n^2+10n \in \Omega(n^2)$이라고 결론지을 수 있다.

- 보기 1.10: $5n^2 \in \Omega(n^2)$임을 보이시오.
  $n \geq 0$인 모든 정수 n에 대해서, $5n^2 \geq 1 \times n^2$이 성립한다. 그러므로, $c = 1$와 $N = 0$을 선택하면, $5n^2 \in \Omega(n^2)$이라고 결론지을 수 있다.

- 보기 1.11: $T(n) = \frac{n(n-1)}{2}$은 어떻게 될까?
  $n \geq 2$인 모든 n에 대해서 $n - 1 \geq \frac{n}{2}$이 성립한다. 그러므로, $n \geq 2$인 모든 n에 대해서 $\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4}n^2$이 성립한다. 따라서 $c = \frac{1}{4}$과 $N = 2$를 선택하면, $T(n) \in \Omega(n^2)$이라고 결론지을 수 있다.

- 보기 1.12: $n^3 \in \Omega(n^2)$임을 보이시오.
  $n \geq 1$인 모든 정수 n에 대해서, $n^3 \geq 1 \times n^2$이 성립한다. 그러므로, $c = 1$와 $N = 1$을 선택하면, $n^3 \in \Omega(n^2)$이라고 결론지을 수 있다.

- 보기 1.13: n이 $\Omega(n^2)$이 아님을 보이시오.
  모순유도에 의한 증명(Proof by contradiction): $n \in \Omega(n^2)$이라고 가정. 그러면 $n \geq N$인 모든 정수 n에 대해서, $n \geq c \times n^2$이 성립하는 실수 $c > 0$, 그리고 음이 아닌 정수 N이 존재한다. 위의 부등식의 양변을 cn으로 나누면 $\frac{1}{c} \geq n$가 된다. 그러나 이 부등식은 절대로 성립할 수 없다. 따라서 위의 가정은 모순이다.

- 참고: 교재 32쪽의 그림 1.6(b)를 보시오.

3.4 Θ 표기법

- **정의 1.4: (Asymptotic Tight Bound)**
  주어진 복잡도 함수 $f(n)$에 대해서 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$. 다시말하면, $\Theta(f(n))$은 다음을 만족하는 복잡도함수 $g(n)$의 집합이다: $n \geq N$인 모든 정수 n에 대해서 $c \times f(n) \leq g(n) \leq d \times f(n)$이 성립하는 실수 $c > 0$와 $d > 0$, 그리고 음이 아닌 정수 N이 존재한다. (교재 29쪽의 그림 1.4(c)를 보시오.)

- 교재 32쪽의 그림 1.6(c)를 보시오.

- 참고: $g(n) \in \Theta(f(n))$은 "$g(n)$은 $f(n)$의 차수(order)"라고 부른다.

- 보기 1.14: $T(n) = \frac{n(n-1)}{2}$은 $O(n^2)$이면서 $\Omega(n^2)$이다. 따라서 $T(n) = \Theta(n^2)$

3.5 작은(Small) o 표기법

- 작은 o는 복잡도 함수 끼리의 관계를 나타내기 위한 표기법이다.

- **정의 1.5: 작은 o**
  주어진 복잡도 함수 $f(n)$에 대해서 $o(f(n))$은 다음을 만족하는 모든 복잡도 함수 $g(n)$의 집합이다: 모든 실수 $c > 0$에 대해서 $g(n) \leq c \times f(n)$ (여기서 $n \geq N$인 모든 n에 대해서)이 성립하는 음이 아닌 정수 N이 존재한다.

- 참고: $g(n) \in o(f(n))$은 "$g(n)$은 $f(n)$의 작은 오(o)"라고 부른다.

- 참고: 큰 O와의 차이점

  ▷ 큰 O - 실수 $c > 0$ 중에서 하나만 성립하어도 됨

  ▷ 작은 o - 모든 실수 $c > 0$에 대해서 성립하여야 함

- 참고: $g(n) \in o(f(n))$은 쉽게 설명하자면 $g(n)$이 궁극적으로 $f(n)$보다 훨씬 낫다(좋다)는 의미이다. 실례를 보자.

- 보기 1.15: $n \in o(n^2)$임을 보이시오.
  증명: $c > 0$이라고 하자. $n \geq N$인 모든 n에 대해서 $n \leq cn^2$이 성립하는 N을 찾아야 한다. 이 부등식의 양변을 cn으로 나누면 $\frac{1}{c} \leq n$을 얻는다. 따라서 $N \geq \frac{1}{c}$가 되는 어떤 N을 찾으면 된다. 여기서 N의 값은 c에 의해 좌우된다. 예를 들어 만약 $c = 0.0001$이라고 하면, N의 값은 최소한 10,000이 되어야 한다. 즉, $n \geq 10,000$인 모든 n에 대해서 $n \leq 0.0001n^2$이 성립한다.

- 보기 1.16: n이 $o(5n)$이 아님을 보이시오.
  모순 유도에 의한 증명: $c = \frac{1}{6}$이라고 하자. $n \in o(5n)$이라고 가정하면, $n \geq N$인 모든 정수 n에 대해서, $n \leq \frac{1}{6} \times 5n = \frac{5}{6}n$이 성립하는 음이 아닌 정수 N이 존재해야 한다. 그러나 그런 N은 절대로 있을 수 없다. 따라서 위의 가정은 모순이다.

- 보기 1.17: n^2이 $o(n)$이 아님을 보이시오.
  모순 유도에 의한 증명: (숙제)

- **정리 1.2: 작은 o와 다른 표기법과의 관계**
  $g(n) \in o(f(n))$이라면, $g(n) \in O(f(n)) - \Omega(f(n))$이 성립한다. 즉, $g(n)$은 $O(f(n))$이기는 하지만, $\Omega(f(n))$은 아니다.

  ▷ $g(n)$은 $O(f(n))$임을 증명(직접증명)
  $g(n) \in o(f(n))$이므로 모든 실수 $c > 0$에 대해서 $g(n) \leq c \times f(n)$이 $n \geq N$인 모든 n에 대해서 성립하는 N이 존재한다. 여기서 임의의 c를 선택하여도 이 부등식은 성립하므로 $g(n)$은 당연히 $O(f(n))$이다.

  ▷ $g(n)$은 $\Omega(f(n))$이 아님을 증명(모순유도에 의한 증명)
  $g(n) \in \Omega(f(n))$이라고 가정하자. 그러면 $n \geq N_1$인 모든 n에 대해서 $g(n) \geq c \times f(n)$을 만족시키는 실수 $c > 0$과 음이 아닌 정수 N_1이 반드시 존재한다. 그러나 $g(n) \in o(f(n))$이기 때문에, $n \geq N_2$인 모든 n에 대해서 $g(n) \leq \frac{c}{2} \times f(n)$을 만족시키는 N_2가 존재한다. 따라서,

  $$cf(n) \leq g(n) \leq \frac{c}{2} f(n)$$

  각 항을 $f(n)$으로 나누면,

  $$c \leq \frac{g(n)}{f(n)} \leq \frac{c}{2}$$

  이 두 부등식에 의하면 N_1과 N_2보다 큰 모든 n에 대해서 성립해야 하는데, 이는 실제로 불가능하다. 모순유도 성공!

- 참고: 일반적으로 $O(f(n)) - \Omega(f(n))$인 복잡도 함수는 $o(f(n))$이기도 하다. 그러나 항상 그렇지는 않다. (자세한 사항은 교재 36쪽 참조)

3.6 차수의 주요 성질

1. $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$
3. $b > 1$이고 $a > 1$이면, $\log_a n \in \Theta(\log_b n)$은 항상 성립. 다시 말하면 로그(logarithm) 복잡도 함수는 모두 같은 카테고리에 속한다. 따라서 통상 $\Theta(\lg n)$으로 표시한다.
4. $b > a > 0$이면, $a^n \in o(b^n)$. 다시말하면 지수형(exponential) 복잡도 함수가 모두 같은 카테고리 안에 있는 것은 아니다.
5. $a > 0$인 모든 a에 대해서, $a^n \in o(n!)$. 다시말하면, $n!$은 어떤 지수형 복잡도 함수보다도 나쁘다.
6. 복잡도 함수를 다음 순으로 나열해 보자.

   $$\Theta(\lg n), \Theta(n), \Theta(n \lg n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$

   여기서 $k > j > 2$이고 $b > a > 1$이다. 복잡도함수 $g(n)$이 $f(n)$을 포함한 카테고리의 왼쪽에 위치한다고 하면, $g(n) \in o(f(n))$.
7. $c \geq 0, d \geq 0, g(n) \in O(f(n))$, 그리고 $h(n) \in \Theta(f(n))$이면, $c \times g(n) + d \times h(n) \in \Theta(f(n))$.

3.7 극한(limit)를 이용하여 차수를 구하는 방법

- **정리 1.3:**

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases} c>0 & \text{이면} \quad g(n)\in\Theta(f(n)) \\ 0 & \text{이면} \quad g(n)\in o(f(n)) \\ \infty & \text{이면} \quad f(n)\in o(g(n)) \end{cases}$$

- 보기 1.16: 다음이 성립함을 정리 3을 이용하여 보이시오.
  - ▷ $\frac{n^2}{2}\in o(n^3)$

$$\lim_{n\to\infty}\frac{n^2/2}{n^3}=\lim_{n\to\infty}\frac{1}{2n}=0$$

  - ▷ $b>a>0$일때, $a^n\in o(b^n)$

$$\lim_{n\to\infty}\frac{a^n}{b^n}=\lim_{n\to\infty}\left(\frac{a}{b}\right)^n=0 \text{ 왜냐하면, } 0<\frac{a}{b}<1$$

- **정리 1.4: 로피탈(L'Hopital)의 법칙**

$$\lim_{n\to\infty}f(n)=\lim_{n\to\infty}g(n)=\infty \text{ 이면}$$

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\lim_{n\to\infty}\frac{g'(n)}{f'(n)} \text{ 이다.}$$

- 보기 1.17: 다음이 성립함을 정리 3과 4를 이용하여 보이시오.
  - ▷ $\lg n\in o(n)$

$$\lim_{n\to\infty}\frac{\lg n}{n}=\lim_{n\to\infty}\frac{\frac{1}{n\ln 2}}{1}=0$$

  - ▷ $\log_a n\in\Theta(\log_b n)$

$$\lim_{n\to\infty}\frac{\log_a n}{\log_b n}=\lim_{n\to\infty}\frac{\frac{1}{n\ln a}}{\frac{1}{n\ln b}}=\frac{\log b}{\log a}>0$$