□ 튜토리얼 □

계산이론 기초개념의 소개

포항공과대학 김 철 언

목 차

계산이론에서 기초가 되는 몇가지 개념을 소개하고자 한다.

먼저 계산이론에서 다루는 문제란 함수를 계산하는 것이고 그의 해답이란 그 함수를 계산하는 알고리즘이라는 것을 설명한다. 그러나 어려운 문제는 알고리즘이 무엇인지 수학적으로 정의할 수 없는데 있다. 따라서 알고리즘을 규정하려는 계산모형들이 많이 제시되었는데, 여기서는 Turing 계산기, 일반 문법 그리고 μ-순화함수의 세 가지를 간략히 살펴본다. 이 세 모형이 모두 동등하다는 사실이 Turing 계산기로 구현할 수 있어야만 알고리즘이라고 할 수 있다는 명제의 바탕이 된다. 다음에는 풀수 없는 문제들을 다루어 본다. 풀수 없는 문제로서 가장 기본이 되는 멈춤문제가 왜 풀수 없는지 증명한다. 그리고 멈춤문제를 써서 얻어지는 다른 여러 풀수 없는 문제들을 살펴본다. 마지막으로 풀수 있는 문제들의 난이성을 알아본다. P, NP 그리고 완전 NP 분류란 어떠한 문제들로 이루어져 있고 또 어떠한 특성을 지니고 있는지 논의해 본다.

I. 문제와 해답

우리는 문제하면 흔히 아래와 같은 물음을 생각한다.

- (1a) 76과 32의 최대공약수는 무엇인가?
- (1b) $6x-2x^2y+1=0$를 만족시키는 x와 y의 정수값이 있는지?

그러나 계산이론에서는 위의 물음이 한 경우가 되는 다음과 같은 것을 문제라고 일컫는다.

- (1b) 두 정수 m과 n의 최대공약수는 무엇인가?
- (2b) 다항식 $p(x_1,x_2\cdots,x_k)=0$를 만족시키는 $x_1,x_2\cdots,x_k$의 정수값이 있는가?

이러한 문제는 아래와 같이 함수로 나타낼 수 있다.

- (1c) gcd : $N\times N\rightarrow N$은 $gcd(m,n)=1$이면 1은 m과 n의 최대공약수로 정의된 함수이다.
- (2c) $P=\{p(k,n)|p(k,n)$는 k변수의 n차 다항식$\}$가 다항식 (polynomial equation)의 모임(set)이라고 하자, 그러면

dio : P→{0,1}은

$$dio(p(k,n))=\begin{cases}1 & p(k,n)을\ 만족시키는\ x_1,x_2\cdots,\\ & x_k의\ 정수값이\ 있는\ 경우\\ 0 & 그렇지\ 않은\ 경우\end{cases}$$

로 정의된 함수이다.

정수나 다항식과 같은 계산이론에서 다루는 함수의 변수와 함수의 값은 모두 어떤 글자모임(alphabet)의 낱말(word또는 string)로 나타낼 수 있으므로, 문제란 한 글자모임의 낱말에 대하여 다른 글자모임의 낱말을 값으로 주는 함수라고 말할 수 있다. 그래서 문제란 함수이고 한 문제의 해답이란 그 함수를 계산하는 알고리즘이다.

보기로서, 위의 (1c)에 주어진 함수 gcd를 계산하는 알고리즘은 2300여년 앞서 그리이스(Greece)의 수학자 유클리드(Euclid)가 내놓은 아래와 같은 순환적 알고리즘(recursive algorithm)인 *euclid-gcd*이다.

```
algorithm euclid-gcd(m,n)
    if n=0
      then return m
      else euclid-gcd(n,m mod n);
```

이 알고리즘이 문제의 한 경우인 (1a)의 m=36과 n=24일 때에 적용시키면

```
euclid-gcd(36, 24)
  =euclid-gcd(24,12)=euclid-gcd(12,0)=12
```

그러나 위의 (2c)에 주어진 함수 dio를 계산하는 알고리즘은 없다. 아직 찾지 못한 것이 아니라 그러한 알고리즘은 있을 수 없다는 것이 증명되어 있다. 그래서 어떤 문제거나 그것을 푸는 알고리즘이 찾아졌으면 풀린 문제이고, 알고리즘이 없다는 것이 증명이 되었으면 풀수 없는 문제이고, 그 둘 가운데 어떤 하나도 아니면 아직 풀리지 않은 문제이다. 풀리지 않은 문제는 앞으로 풀릴 수도 있고, 풀수 없는 문제라고 밝혀질 수도 있지만 또한 풀리지 않은 문제로 남을 수도 있다.

II. 계산모형(*models of computation*)

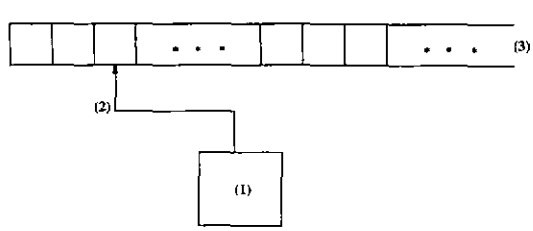
앞서 문제와 그의 해답에 관한 얘기를 했으나, 그것이 그리 알찬 것이 되지 못한다고 할 수 있겠는데, 그 까닭은 알고리즘의 정의도 없이 한 문제를 푸는 알고리즘이 있느니 없느니 또는 있을 수도 없느니 했기 때문이다. 그래서 이제 알고리즘의 정의를 살피도록 하겠다.

알고리즘(algorithm)을 정의하기에 앞서 계산절차(procedure)의 정의부터 보자. 계산절차는 유한한 지시(instruction)의 모임으로 각 지시는 명확하고 효과적이어야 하며, 이 지시들을 어느 순서로 수행하는지 분명해야 한다. 그러나 여기서 지시가 명확하고 효과적이어야 한다는 조건은 수학적으로 규정할 수 없는 것이다. 따라서 이러한 조건을 갖추었다고 여겨지는 지시의 형태를 계산의 모형으로 제시하여 쓰는 수 밖에 없게 된다. 이렇게 제시된 계산모형이 여러가지 있는데 우리가 다루려 하는 세가지는 Turing 계산기, 일반문법(general grammar)과 μ-순환함수(μ-recursive function)이다.

알고리즘은 어떠한 변수의 값이 주어지거나 지시를 정해진 순서를 따라 유한히 수행하고 나면 함수의 값을 얻어재고 끝이나는 계산절차이다. 즉 무한한 굴레(infinite loop)나 수행할 지시가 없는 막다름에 들어서지 않고 꼭 끝나는 계산절차가 알고리즘이다. 미리 말해두고 싶은 것은 한 계산절차가 알고리즘인지 아닌지 결정하는 문제 또한 풀수 없다는 사실이다.

2.1 Turing 계산기

영국의 수학자 Alan M. Turing이 1936년에 알고리즘을 구현하는 구체적인 계산모형으로 제안한 추상적인 계산장치가 이제는 그의 이름을



Turing 계산기 : (1)유한조정 (2)읽고쓰기바늘 (3)테이프

따서 불리어지는 Turing 계산기(Turing machine 또는 Tm)이다.

Turing 계산기는 그림에서 보다시피, 유한조정과 테이프 그리고 읽고쓰기바늘로 이루어져 있다. 그 테이프는 칸으로 나뉘어져 왼쪽의 첫 칸으로부터 오른쪽으로 끝없이 뻐쳐 있는데 언제나 왼쪽(또는 앞쪽)의 유한한 수의 칸만을 빼고는 모두 빈칸(빈글자로 메꾸어진 것)이다. 공식적으로 Turing 계산기를 다섯쌍인 M=(Q, Σ, Γ, δ, s)으로 나타내는데, 여기서

- Q는 유한조정이 들어설 수 있는 상태로 이루어진 유한한 모임으로, 정지상태(halt state) h는 들어 있지 않고,
- Σ는 입력낱말을 만드는 입력글자로 이루어진 유한한 글자모임,
- Γ는 테이프에 쓰이는 유한한 글자모임으로 빈글자(blank) #와 Σ의 모든 글자도 포함하고 있으며,
- s는 Q에 들어 있는데 출발상태라 불리우며,
- δ : Q×Γ→(Q∪{h})×Γ×{L,S,R}은 전이함수인데 Tm의 작동을 규정한다.

유한조정은 출발상태에서 시작하여 계산하는 동안 각 단계마다 전이함수 δ가 정하는 상태에 들어서며 정지상태 h에 들어서게 되면 계산과정이 끝나게 된다. 테이프바늘(읽고쓰기바늘)은 놓여져 있는 칸안의 글자를 읽기도 하고 고쳐쓰기도 하며, δ가 정하는데 따라 고쳐쓰고 나서 왼쪽 또는 오른쪽으로 한칸 옮기거나 그 칸에 머무른다. 즉 이제 Tm M의 유한조정이 상태 q에 들어 있고 읽고쓰기바늘이 놓여 있는 칸의 글자가 a이고 δ(q,a)=(p,b,D)이면, M의 유한조정은 상태를 p(q와 같을 수도 있음)로 바꾸어 들어서고, 읽고쓰기바늘은 a를 b(a와 같을 수도 있음)로 바꾸어쓰고 나서 D가 L, R 또는 S인지에 따라서 왼쪽이나 오른쪽으로 한 칸 움직이거나 그 칸에 그냥 머무른다. 이와 같은 작동이 Tm M의 계산과정의 한 단계를 이룬다.

Turing 계산기의 계산과정을 쉽고 명확히 나타내는데 uqav를 쓰는데, 이것이 뜻하는 것은 M의 유한 조정이 상태 q에 들어 있고 읽고쓰기바늘이 놓여진 칸에 쓰여진 글자가 a에 그 왼쪽의 테이프 칸에 쓰여진 글자들을 차례로 써놓은 낱말(word 또는 string)은 u이고 오른쪽 테이프 칸에 쓰여진 빈글자로 끝나지 않는 가장 긴 부분의 낱말이 v이다. 만일 u′pbv′이 uqav로부터 한 단계에 얻어지면 uqav ⊢ u′pbv′으로 나타낸다. 또 Turing 계산기에서 정수 n은 n개의 I,Iⁿ, 으로 나타낸다. 그래서 add(m,n)=m+n으로 정의 되는 함수 add : N^2→N을 계산하는 Tm M은 처음에 s#I^m#I^n에서 지작하여 h#I^{m+n}으로 멈추어야 한다. 이 함수를 계산하는 Tm의 한 보기인 M=({$q_0,q_1\cdots,q_6$}, {I},{I,#},δ,q_0)의 전이함수 δ는 아래와 같다.

Q	q_0	q_1	q_2	q_3	q_4	q_5	q_6
I	(q_0,I,L)	(q_1,I,R)	(q_2,I,R)	(q_3,I,R)	(q_4,#,L)	(q_5,I,L)	(q_6,I,L)
#	(q_1,#,R)	(q_2,#,R)	(q_6,#,L)	(q_4,#,L)	(q_4,#,S)	(q_6,I,L)	(h,#,S)

이 Tm M이 add(3,2)=5를 계산하는 과정을 위의 방법으로 써보면 아래와 같다.

q_0#III#II ⊢ #q_1III#II⊢* #IIIq_1#II #III#q_2II⊢* #III#Iq_3I⊢* #III#IIq_3#⊢ # #III#Iq_4I #III#q_5I⊢ #IIIq_5#I #IIq_6III⊢* #q_6IIIII⊢q_6#IIIII⊢h#IIIII.

2.2 일반문법

알고리즘을 구현하는 또 하나의 계산모형으로는 일반문법(general grammar)이 있는데, 간단히 문법(grammar)이라고 부른다. 문법 G는 네쌍인 G=(N,Σ,S,P)로 규정하는데, 여기서

- N은 유한한 변수글자의 모임,
- Σ는 유한한 (상수) 글자의 모임,
- S는 N에 든 변수글자인 출발글자이고,
- P는 유한한 바꾸어쓰기(rewriting rule) 또는 산출(production)의 모임이다.

산출은 α→β로 나타내는데 α ε (N∪Σ)* N(N∪Σ)*이고 β ε (N∪Σ)*이어서 글자수열(string of symbols)안에 들어 있는 부분수열 α는 β로 바꾸어 쓸 수 있다. 낱말 uαv에서 산출 α→β를 이용하여 α를 β로 바꾸어 써서 uβv를 얻으면 uαv⇒uβv로 나타내고, 낱말 x에서 여러번의 (0번 또는 그 이상의) 바꾸어쓰기를 거쳐서 y를 얻으면 x⇒*y로 나타내는데 이것을 x로부터 y로의 유

도수열(derivation sequence)이라고 일컫는다.

문법에서도 Turing 계산기에서와 같이 정수 n을 I^n으로 나타낸다. 그래서 함수 add : $N^2 \rightarrow N$을 계산하는 문법 G는 $SI^m\#I^n\#$에서 시작하여 I^{m+n}을 얻는 유도수열을 가져야 한다. 이 함수를 계산하는 문법의 한 보기로는 G=({S}, {I,#}, S, {SI→IS, S#→λ}가 있다. 이 문법 G가 add(3,2)=5를 계산하는 과정을 유도수열을 써서 보면 아래와 같다.

SIII#II⇒ISII#II⇒*IIIS#II⇒IIIII.

2.3 μ-순환함수

마지막으로 알아보려는 계산모형은 μ-순환함수이다. 한 함수가 μ-순환함수이면 다음과 같은 과정으로써 계산할 수 있어야 한다.

2.3.1. 기본함수

- (a) 값이 0인 함수 z : N→N는 z(n)=0으로 정의되며 변수의 값이 무엇이거나 함수의 값을 0으로 주면 된다.
- (b) 다음정수 함수 s : N→N은 $s(n)=n^+$로 정의 되는데 n^+는 n의 다음 정수, 즉 값이 n+1로, 변수의 값이 n이면 그 다음 정수를 함수의 값으로 주면 된다.
- (c) 투영함수 $P_j^k : N^k \rightarrow N$은 $P_j^k(n_1,\cdots,n_k)=n_j$로 정의 되는데 k개의 변수 값에서 j번째 변수의 값을 함수의 값으로 주면 된다.

위의 세 가지 기본함수는 모두 μ-순환함수이고, 이 기본함수로부터 시작해서 다음과 같은 방법으로 계산할 수 있는 함수는 모두 μ-순환함수이다.

2.3.2 복합(composition)

함수 $h : N^l \rightarrow N$과 $g_1,\cdots,g_l : N^k \rightarrow N$가 모두 μ-순환함수라 하고 $(n_1,\cdots,n_k)$를 $\overline{n}$으로 나타내자. 이들 함수로부터 함수 $f : N^k \rightarrow N$을 $f(\overline{n})=h(g_1(\overline{n}),\cdots,g_l(\overline{n}))$로 정의해서 계산하는 μ-순환함수이고, f는 h와 $g_1\cdots,g_l$로부터 복합에 의해서 얻어졌다고 한다.

2.3.3 초보순환(primitive recursion)

함수 $g : N^k \rightarrow N$과 $h : N^{k+2} \rightarrow N$이 μ-순환함수이고 $\overline{n}=(n_1\cdots,n_k)$라고 하자. 이 두 함수로부터 함수 $f : N^{k+1} \rightarrow N$을 아래처럼 정의하면 f는 g와 h로부터 초보순환으로 얻어진 함수라 일컫는다.

- $f(\overline{n},o)=g(\overline{n})$
- $f(\overline{n},m+1)=h(\overline{n},m,f(\overline{n},m))$

이렇게 계산할 수 있는 함수 f도 μ-순환함수이다.

2.3.4 최소화(minimization)

함수 $g : N_{k+1} \rightarrow N$이 μ-순환함수라 할 때, 함수 $f : N^k \rightarrow N$를 μ-연산(μ-operator)을 써서 아래처럼 정의하자.

$f(n)=\mu m[g(n,m)=0]$
$=\{g(n,m)=0\}$을 만족시키는 가장 작은 정수인 m의 값

그러면 f는 g로부터 최소화로 얻어진 μ-순환함수이다.

보기로 위에서 들었던 add : $N^2 \rightarrow N$이 μ-순환함수임은 기본함수 P_1^1, s, P_3^3로 시작해서, P_1^1인 g, 그리고 s와 P_3^3를 복합해서 얻은 함수인 h로부터 아래와 같이

$add(n,o)=P_1^1$
$add(n,m+1)=s(P_3^3,add(n,m)))$

초보순환으로 얻어진다는 것으로 증명된다. 그래서 위에서 보기로 들었던 add(3,2)=5의 계산은

$$\begin{aligned} add(3,2)=add(3,1+1)&=s(P_3^3(3,1,add(3,1))\\ &=s(add(3,1))=2(add(3,0+1))\\ &=s(s(P_3^3(3,0,add(3,0))))\\ &=s(s(add(3,0)))=s(s(P_1^1(3)))\\ &=s(s(s(3))=s(4)=5 \end{aligned}$$

여기서 μ-연산까지 써야 얻어지는 함수를 보기로 들지 않은 데는 까닭이 있다. 우리에게 잘 알려지고 또 쓰여지는 거의 모든 함수가 μ-연산을 필요로 하지 않고 μ-연산을 필요로 하는 함수는 그 유도가 무척 복잡하기 때문이다.

2.4 Church-Turing 명제

계산이론에서 가장 중요한 결과 가운데 하나를 증명을 하지 않고 정리만을 아래에 적는다.

- **정리** 현대 계산기로 계산할 수 있는 어떤 함수나 Turing 계산기로 계산할 수 있고, Turing 계산기로 계산할 수 있는 함수는 문법으로 계산할 수 있으며, 문법으로 계산할 수 있는 함수는 μ-순환함수이고, μ-순환함수는 현대 계산기로 계산할 수 있다.

이 정리가 말해주는 것은 이 여러 계산모형들이 모두 계산과정을 보는 관점이 다르나 똑같은 능력을 가졌다는 것과, 증명은 할 수 없는 것이지만, 이 모두가 알고리즘을 구현하는 계산모형의 역할을 할 수 있다는 것이다. 이 사실을 명백히 하는 것이 다음에 적는 명제(thesis)이다.

- **Church-Turing 명제**
  어느 알고리즘이나 Turing 계산기로 구현할 수 있어야 한다.

III. 풀 수 없는 문제

앞서 문제는 함수로 나타낼 수 있고, 그 함수를 계산하는 알고리즘이 있을 수 없다고 증명이 되면 그 문제는 풀 수 없는 문제라고 했었다. 그런데 Church-Turing의 명제에 의하면 알고리즘은 Turing 계산기로 구현될 수 있어야만 하기 때문에 어떤 문제가 Turing 계산기로 풀 수 없다는 것이 밝혀지면, 그 문제는 풀 수 없는 문제라고 결론지을 수 있다. 이것을 바탕으로 해서 풀 수 없는 문제들 가운데서 가장 기본이 되는 멈춤문제(the halting problem)를 살펴보기로 한다.

누구나 알다시피 숫자나 글자를 0과 1의 두 글자로 나타낼 수 있어서, 프로그램이나 그의 입력도 모두 0과 1의 두 글자를 써서 규정할 수 있다. 이제 $f_d : \{0,1\}^* \rightarrow \{0,1\}$이 아래와 같이 정의된 함수라고 하자.

$$f_d(w)=\begin{cases} 1 & w\text{가 Turing 계산기 M을 0과 1로 나타낸 것이고 이 Tm 계산기 M에 w가 입력으로 주어지면 그 계산 결과의 값이 0인 경우} \\ 0 & \text{위의 경우가 아닌 때} \end{cases}$$

이 함수 f_d는 풀 수 없는 함수(계산할 수 없는 함수)이다. 그 까닭은 다음과 같다. 만일 f_d가 풀 수 있는 함수라고 가정하면 f_d를 계산하는 Turing 계산기 M_d가 있어야 한다. 이 M_d를 0과 1로서 나타낸 것을 w_d라고 하자. 그러면 $f_d(w_d)=1$이거나 $f_d(w_d)=0$이어야 한다. 먼저 $f_d(w_d)=0$인 경우를 보면 f_d의 정의에 따라 M_d가 w_d를 입력으로 받아 계산하는 결과가 0이다. 그러나 M_d가 f_d를 계산하는 Turing 계산이므로 $f_d(w_d)=1$은 w_d를 입력으로 받은 M_d의 계산결과가 1이어서 모순이므로 $f_d(w_d)=1$일 수 없다. 그러면 $f_d(w_d)=0$일 수 밖에 없는데, $f_d(w_d)=0$는 f_d의 정의에 의하여 M_d가 w_d를 입력으로 받으면 그 결과가 1어야한다. 그러나 M_d가 w_d를 입력으로 받아들여 계산한 결과인 $f_d(w_d)$는 0이라는 모순에 이른다. 따라서 $f_d(w_d)=0$일 수도 없다. 그러므로 결론은 f_d를 계산하는 Turing 계산기가 없다는 것이고, 따라서 f_d는 계산할 수 없는 함수이다. 함수 $\overline{f}_d : \{0,1\}^* \rightarrow \{0,1\}$을 $\overline{f}_d(w)=1-f_d(w)$로 정의하면 $\overline{f}_d$ 또한 풀 수 없는 함수라는 것은 엄연하다.

3.1 풀 수 없는 멈춤문제(Unsolvability of the halting problem)

위의 결과로 풀 수 없는 문제 가운데서 가장 기본이 되는 멈춤문제를 다룰 준비가 되었다.

- **멈춤문제** Turing 계산기 M은 입력 낱말 w가 주어지면 그 결과가 1인지 0인지를 결정한다. (워낙은 M이 주어진 w에서 계산과정이 멎는지(1인지) 아니면 멎지 않고 끊임없이 돌아가는지(0인지)를 결정하는 문제로 다루어지기 때문에 멈춤문제라고 불리워진다.)
- **정리** 멈춤문제는 풀 수 없는 문제이다.
- **증명** 멈춤문제를 풀 수 있다고 가정하자. 그러면 Turing 계산기 M을 나타내는 0과 1로 된 낱말 w_M과 w를 나타내는 w_i를 함께 쓴 w_Mw_i가 주어졌을 때, w에서의 M의 계산결과가 1이면 1을 계산결과로, 0이면 0을 계산결과로 내는 Turing 계산기 M_u가 있어야 한다. 이제 다음과 같은 계산을 하는 Turing 계산기 M_r을 만들어 보자. M_r은 입력 w_M이 주어지면 w_M을 하나 더 만들어

$w_M w_M$을 M_u에게 주어 계산하도록 한다. 그리고 나서 M_t은 M_u의 계산결과를 자신의 계산결과로 쓴다. 그러면 M_t은 $\bar{T}_d$를 계산하는 Turing 계산기인 셈이다. 그러나 앞서 $\bar{T}_d$를 계산하는 Turing 계산기는 있을 수 없다는 것이 증명되었으므로 이 모순은 멈춤문제를 풀 수 있다는 가정에서 다다르게 되어 멈춤문제는 풀 수 없다는 증명이 된다.

멈춤문제는 풀 수 없다는 결과를 써서, 풀 수 없다는 것을 직접 쉽게 증명할 수 있는 문제들이 여러가지 있다. 그러한 문제들 가운데에서 세 가지를 들어 본다. 한 프로그램이 입력을 받아들여 작동을 시작하면 멎을 것인지 무한굴레(infinite loop)에 들어갈 것인지를 결정할 알고리즘이 없다. 다음으로 주어진 두 프로그램이 같은 함수를 계산하는 프로그램인지 아닌지를 결정하는 문제도 풀 수 없는 문제이다. 마지막으로 한 프로그램이 어떤 입력을 받고 작동을 시작하건 늘 무한굴레에 들어서는지 결정하는 문제도 풀 수 없다.

다시 되풀이 하고자 하는 것은 알고리즘이 없다고 해서 그 문제의 어느 경우도 답을 찾아낼 수 없다는 게 아니라는 것이다. 프로그램에 따라서는 입력이 어떠하든 언제나 작동을 멈춘다는 것을 알 수 있는 것도 있다.

3.2 그 밖의 풀 수 없는 문제들

위에서는 프로그램에 관한 풀 수 없는 문제들을 다루었는데 여기서는 그와는 다른 종류의 문제를 살펴본다.

- **Post의 상응문제(Post correspondence problem)**
  두 상응되는 글자수열$(x_1,x_2,\cdots x_n)$과 $(y_1,y_2,\cdots,y_n)$이 주어졌을 때 $x_{i1}x_{i2}\cdots x_{in}=y_{i1},y_{i2}\cdots y_{in}$가 되는 해답인 수열 $(i_1i_2,\cdots,i_n)$이 있는지 결정한다.

보기로 두 상응되는 글자수열 $(x_1,x_2,x_3)=(1,101\ 11,10)$과 $(y_1,y_2,y_3)=(111,10,0)$가 주어지면 수열 (2,1,1,3)이 해답인데 그 까닭은 $x_2x_1x_1x_3=1011\ 11110=y_2y_1y_1y_3$이기 때문이다. 그러나 두 상응되는 글자수열이 $(x_1,x_2,x_3)=(10,011,101)$과 $(y_1,y_2,y_3)=(101,11,011)$이면 해답이 없다.

멈춤문제는 풀 수 없다는 정리로부터 얻어지는 결과를 증명은 없이 아래에 정리로 적는다.

- **정리** Post의 상응문제는 풀 수 없다.

이 결과를 써서 다음의 문맥무관언어(context-free language)에 관한 결정문제도 풀 수 없다는 결과를 쉽게 얻을 수 있다.

- **정리** 두 문맥무관언어의 겹모임(intersection)이 비었는지 아닌지를 결정하는 문제는 풀 수 없다.
- **증명** 두 상응되는 글자수열$(x_1,x_2\cdots,x_n)$과 $(y_1,y_2,\cdots,y_n)$이 주어지면 두 문맥무관 문법

$G_x=(S,\Sigma,\{S\rightarrow ix_i|iSx_i : 1\leq i\leq n\},S)$와
$G_y=(S,\Sigma,\{S\rightarrow ix_i|iSy_i : 1\leq i\leq n\}S)$를

만든다. 그러면 두 문법이 생성하는 언어는 각각

$L(G_x)=\{i_m\cdots i_1x_{i1}\cdots x_{im}|1\leq i_k\leq n\}$과
$L(G_y)=\{i_m\cdots i_1x_{i1}\cdots x_{im}|1\leq i_k\leq n\}$

이다. 그래서 $L(G_x)\cap L(G_y)\neq\phi$인 필요하고 충분한 조건은 두 글자수열의 Post 상응문제가 해답을 갖는 것이다. 그러므로 두 문맥무관언어의 겹모임이 비었는지 않은지를 결정하는 문제를 풀 수 있다면 Post 상응문제의 해답이 있는지 없는지를 결정할 수 있다는 모순에 다달으므로 이문제는 풀 수 없어야 한다.

IV. 문제의 난이도

거듭 말하지만 문제의 해답은 그 문제를 푸는 알고리즘이다. 계산이론에서 한 문제의 난이도는 그 문제를 푸는 알고리즘을 찾기가 얼마나 어려웠다던가, 그 알고리즘이 얼마나 이해하기 어렵다거나 또는 알고리즘이 얼마나 복잡한가를 얘기하는 것이 아니다. 한 문제가 어렵거나 쉽다는 것은 그것을 푸는 알고리즘을 수행하는데 드는 자원이 얼마나 많으냐 적으냐에 달린다. 대개의 경우 가장 중요한 자원이 시간이어서 여기서는 알고리즘을 실행하는 데 드는 시간이 얼마나 되느냐에 따라 문제의 난이도를 정한다. 물론 한 문제를 푸는 알고리즘이 여러가지 있

으므로 문제의 난이도는 가장 실행시간이 덜 드는 알고리즘으로 견주는 것이 당연하다. 또한 알고리즘을 실행하는 시간은 문제의 크기에 따라 다를 것이고 같은 크기의 경우에도 데이타의 생김새에 따라 다르게 마련이다.

간단한 보기를 들어보자. 크기 순서로 놓여 있지 않은 n개의 정수를 순서대로 바꾸어 늘어놓는 문제를 SORT라고 표시하자. 다음은 역시 크기 순서로 놓이지 않은 n개의 정수들 가운데서 k번째로 큰 수를 찾아내는 문제를 SELECT라고 하자. 마지막으로 n개의 크기에 따라 늘어 놓은 정수 가운데서 한 특정한 정수를 찾아내는 문제를 SEARCH라고 부르자. 그러면 SORT를 푸는 알고리즘 가운데서 가장 빠른 것의 실행시간은 $n\log_2 n$에 비례하고, SELECT를 푸는 가장 빠른 알고리즘의 실행시간은 n에 비례하며, SEARCH를 푸는 알고리즘 가운데서 가장 빠른 것은 $\log_2 n$에 비례하는 시간에 계산을 끝낸다. 따라서 이 세 문제 가운데서 가장 어려운 것은 SORT이고, SEARCH가 가장 쉬우며, SELECT는 그 사이에 있다고 할 수 있다.

그래도 정해진 정수 k가 있어 실행시간이 n^k에 비례하거나 보다 적은 알고리즘을 가진 문제는 비교적 쉬운 문제로 다루며 이러한 문제들이 속하는 분류를 P-분류라 일컫는다.

4.1 NP-분류

한 홀수가 복수(composite)인지 결정하는 문제를 생각해 보자. 오랜동안 솟수로 알려졌던 보기로 $2^{67}-1$인 147,573,952,589,676,412,927가 복수인지 결정하려면 쉬운 일이 아니다. 아마 보통 사람이 손으로 계산해서 이 수가 복수라는 것을 알아내려면 밤낮을 쉬지 않고도 한해는 걸릴 것이다. 그러나 하나의 수 193,707,721를 주고 이 수로 앞의 수를 나눌 수 있다는 것을 보여서 그 수가 복수임을 증명하려면 몇 초만에 증명할 수 있다. 이렇게 결정하기는 어려워도, 한 결정이 옳다고 쉽게 증명할 수 있는 방법이 있는 문제는 NP에 들어 있다고 하고, 이러한 모든 결정 문제의 모임을 NP-분류(NP-class)라고 일컫는다.

이제 위의 문제와 보완적(complementary)인 문제를 생각해 보자. 즉 한 홀수가 솟수(prime)인지 결정하는 문제인데, 보기로 25,726,506,678,823,481,911가 솟수인지 결정하는 것은 쉽지 않다. 실은 이 수가 솟수인데 얼핏 생각하기로는 이 수가 솟수라고 쉽게 증명할 방법이 있을 것 같지 않다. 그러나 여기서 보여줄 수는 없지만, 이 문제 또한 쉽게 증명할 방법이 있어 NP에 들어 있다.

한 홀수가 복수인지 결정하는 문제처럼, 한 문제가 NP에 들어 있고 그의 보완되는 문제도 NP에 들어 있으면, 그 문제는 co-NP에 들어 있다고 한다.

다음은 배낭문제(knapsack problem)라고 불리우는 아래와 같은 문제를 살펴보자. 한 유한한 정수의(한 정수가 되풀이 들어 있어도 되는) 모임 $S=\{a_1,a_2,\cdots,a_n\}$과 또 하나의 정수 c가 주어졌을 때, 합이 c가 되는 S의 부분집합이 있는지 결정하는 문제이다. 보기로 $S=\{2,4,7,9,12,16\}$이고 $c=25$이면 조건을 만족시키는 부분집합이 있다. 합이 25가 되는 부분집합이 있다는 결정이 맞다는 증명은 부분집합 {4,9,12}를 주면 쉽게 증명이 된다. S가 많은 원소를 갖고 c가 아무리 큰 수라도 합이 c가 되는 S의 부분집합만 찾으면 이 결정 문제에 해답이 있다는 것을 증명하기는 쉽다. 따라서 이 배낭문제는 NP에 들어 있다.

이제 이와 보완되는 문제를 보자. 즉 S와 c가 주어졌을 때 합이 c가 되는 부분집합이 없는지 결정하는 문제인데, 위의 보기에서 $c=33$인 경우이다. 이 보완문제의 답을 쉽게 증명할 방법은 여태 찾지 못했고 또 있지도 않은 것 같지만, 이 문제가 co-NP에 들어 있지 않다는 증명 (즉 배낭문제의 보완문제는 NP에 들어 있지 않다는 증명)은 아직 되어 있지 않은데, 이것을 증명하는 것이 계산이론에서 가장 어려운 문제로 남아 있다.

다시 배낭문제로 돌아가 보자. 이 문제를 푸는 알고리즘으로 누구나 생각해 내는 것으로는 S의 모든 부분집합을 그 합이 c인 것을 찾을 때까지 하나씩 만들어 나가는 것이다. S의 원소의 수가 n이면 부분집합의 수는 2^n이고, 보기로 $n=64$인 경우에는 2^{64}이다. 한 부분집합을 만들어 그 합을

얻어내는데 1 μsec이 걸린다 하고 모든 부분집합을 검색해야 하는 경우에는 2^{64} μsec이 걸리는데 이것은 대략 500,000년이라는 긴 시간이다. 여태까지 찾아진 이보다 빠른 알고리즘들도 이 문제를 푸는데 몇만년이 걸리는 까닭에 이 문제는 어렵다고 보아야겠다.

4.2 완전 NP-분류(NP-complete class)

문제 사이에 다항시간안의 변형(polynomial-time reduction)이 여기서 먼저 소개하려는 개념이다. 그래서 분할문제(partition problem)라고 불리우는 아래의 문제를 보자. 한 유한한 정수의 (되풀이 되어도 좋은) 모임 $R=\{b_1,b_2\cdots,b_m\}$이 주어지면 R을 합이 같은 두 개의 부분집합으로 분할 할 수 있는지 결정하는 것이 분할문제이다. 앞서 보았던 배낭문제는 분할문제로 변형시켜서 풀 수 있다는 것을 다음과 같이 알 수 있다. 배낭문제의 $S=\{a_1,a_2,\cdots,a_n\}$와 정수 c가 주어졌을 때, $R=\{a_1,a_2,\cdots a_n,c+1,(\Sigma_{i=1}^{n}\ a_i)+1-c\}$를 가진 분할문제를 만들면, 이 두 문제의 해답은 같다. 즉, 합이 c가 되는 S의 부분집합이 있을 필요하고도 충분한 조건은 R을 합이 같은 두 부분집합으로 분할 할 수 있는 것이다. 여기서 배낭문제를 분할문제로 바꾸는데 든 시간은 문제의 크기 n과 비례한다. 이렇듯 한 문제 Q_1을 그와 답을 같이 갖는 문제 Q_2로 문제의 크기의 다항식으로 표현할 수 있는 시간안에 변형할 수 있으면, Q_1은 Q_2로 다항시간안에 변형될 수 있다고 말한다. 만일 Q_1은 Q_2로 또 Q_2는 Q_1으로 서로 다항시간안에 변형될 수 있으면, 이 두 문제는 다항시간안에서 동등하다(Polynomially equivalent)고 일컫는다.

따라서 만일 문제 Q_1을 Q_2로 다항시간안에 변형시킬 수 있고 Q_2를 쉽게 푸는 알고리즘이 있으면 Q_1을 쉽게 푸는 알고리즘이 있다는 뜻이다. 문제 Q가 NP에 들어 있고, NP-분류에 든 모든 문제를 Q로 다항시간안에 변형시킬 수 있으면, Q는 완전한 NP 문제(NP-complete problem)라고 부른다. 그리고 이 분류에 들어 있는 문제 Q의 특징은, Q를 쉽게 풀 수 있는 알고리즘이 있으면 NP-분류에 든 어떤 문제라도 쉽게 풀 수 있는 알고리즘이 있고, Q를 쉽게 풀 수 없다는 증명이 되면 완전한 NP 문제는 어느 것이나 쉽게 풀 수 없다는 것도 증명이 된다는 것이다. 이러한 완전 NP 문제가 있는지는 또 다른 문제이다. 1971년에 S. Cook가 이러한 문제가 있다는 것을, 충족가능성문제(satisfiability problem)가 완전 NP 문제라는 것을 보여줌으로서 증명하였다. 문제 Q가 완전 NP 문제이고 Q를 문제 T로 다항시간안에 변형시킬 수 있으면 T도 완전 NP 문제이다. 그래서 충족가능성문제로부터 시작해서 얻어진 완전 NP 문제로는 배낭문제와 분할문제 말고도 수백 가지가 되는데, 그 가운데에서도 잘 알려진 것으로는 출장판매원(travelling salesman), 그래프색칠(graph coloring), 정수선형 프로그래밍(integer linear programming) 그리고 처리기 분배(processor scheduling)의 문제들이다. 여기서 논의한 것을 간추려서 정리를 만들면 아래와 같다.

- **정리** 문제 Q를 완전한 NP 문제라고 하자. 그러면 Q가 P에 들어 있을 필요하고도 충분한 조건은 P와 NP가 같은 분류인 것이다.

P가 NP와 같은 분류인지 아닌지는 아직 밝혀지지 않았고, 계산이론 분야 뿐 아니라 계산과 관련된 모든 분야의 학자들이 알고싶어 하는 어려운 문제로 남아 있다. 그러나 많은 경험과 증거 때문에 모든 학자들이 P와 NP는 같지 않다고 믿고 있다. 즉 완전 NP 문제를 쉽게 풀 수 있는 알고리즘은 있을 수 없고, 따라서 이 모든 문제들이 풀기 어려운대로 남아 있으리라는 것이다.

김 철 언

1963 서울대학교 화학공학과 (학사)
1971 University of Minnesota 수학과(석사)
1975 University of Minnesota 전자계산학과(박사)
1975 ~1978 University of Maryland 조교수
1978 ~1979 University of Alabama 부교수
1979 ~1981 University of Maryland 조교수
1981 Washington State Univérsity 부교수, 교수
1989 서울대학교 공과대학 객원교수
관심 분야 : Design and analysis of algorithms Computational digital geometry