KAIST 전산학과 최광무

(정의) (**알고리즘** A의 복잡도)
크기가 n인 **항상 끝나는**(type 1, **recursive**, **algorithm**, μ-recursive **total** function) 문제 P를 푸는 알고리즘(프로그램) A가 걸리는 시간이 $O(f(n))$일 때, **알고리즘** A의 **시간복잡도**(time-complexity)를 $f(n)$이라 정의한다. 이 시간복잡도 $f(n)$을 문제 P에 대한 **알려진 알고리즘**의 **최소**(upper-bound) **복잡도** 라고 부른다.

(사실) 알고리즘의 복잡도 순서는 아래와 같다.
$O(1) \leq O(\log(n)) \leq O(n) \leq O(n\log(n)) \leq O(n^2) \leq \cdots \leq O(2^n) \leq O(n^n) \leq \cdots$

(정의) (**문제** P의 복잡도)
어떤 문제 P를 푸는 **가장 빠른 알고리즘**(optimal)의 복잡도를 그 **문제** P의 **복잡도** 라고 부른다.

(예) Bubble sorting **알고리즘**의 복잡도는 $O(n^2)$이다. 이 경우 sorting **문제**의 **알려진 최소** 복잡도는 $O(n^2)$라고 볼 수 있다. 그러나 quick sorting **알고리즘**의 복잡도가 $O(n\log(n))$이고, quick sorting **알고리즘**이 bubble sorting **알고리즘**보다 더 **빠르고**, 이것이 **최소**임(**optimal**)임이 이미 증명 되었으므로, sorting **문제**의 **최소 복잡도**는 $O(n\log(n))$으로 줄어든다.

Optimal 알고리즘이 아닌 경우, **알고리즘**의 **복잡도**와 **문제**의 **복잡도**는 구분해야한다. 문제의 복잡도가 정의되지 않은 경우도 있다.

(정의) **알고리즘**이 **결정적**(deterministic)일 때, 이 알고리즘의 복잡도를 **결정적** 복잡도라고 부르고, 알고리즘이 **비결정적**(non-deterministic)일 때, 이 알고리즘의 복잡도를 **비결정적** 복잡도라고 부른다.

(정의) 문제의 복잡도가 $O(n^k)$, $k \in N$ 또는 $O(p(n))$[1], 다항식 이하인(polynomial) **결정적** 알고리즘이 **알려진** 경우 이 문제를 **풀기 쉬운**(**tractable**) 문제라고 부른다. **알려진 가장 빠른**(**optimal**) **결정적** 알고리즘의 복잡도가 $O(2^n)$ 또는 $O(k^{p(n)})$ 이상, 즉 지수 이상인(exponential) 문제를 **풀기 어려운**(**intractable**) 문제라고 부른다.

(사실) 자연수의 집합 N에 대하여, $|N| = \aleph$이라 할 때, 무한의 크기순서는 아래와 같다.
$k < \log\aleph = \aleph = \aleph^2 = \ldots = \aleph^k = \ldots < 2^{\aleph} = \ldots = \ldots\ k^{\aleph} = \ldots$

(뒷사실) **풀기 쉽고**, **어려운**(tractable, intractable)에 관한 정의의 배경에는 셀 수 있게(countably) 무한한(infinite) $\aleph$에 관하여 $\aleph^k = \aleph$(countable)이나 $2^{\aleph} > \aleph$(uncountable)이라는 위의 사실의 발견이 있었음에 기인한다. 그러나 유한한 n에서도 $O(p(n))$은 풀기 쉽고, $O(k^{p(n)})$은 풀기 어렵다는 단순 대응이 성립하는지는 의문이

1) $p(n) = a_0 n^k + a_1 n^{k-1} + \cdots + a_k$, $k \in N$, $0 \leq \forall i \leq k$: $a_i \in R$.

다[2].

(정의) **결정적** 알고리즘의 복잡도가 $O(p(n))$으로 알려진 문제들의 집합을 **P**, **비결정적** 알고리즘의 복잡도가 $O(p(n))$으로 알려진 문제들의 집합을 **NP**[3]라 한다.

(사실) **P** $\subseteq$ **NP**이지만 **P** $=$ **NP**나 **P** $\neq$ **NP**(**P** $\subset$ **NP**)라는 증명은 없다.

NP 문제 중에는 **P**인 문제도 많다. 7장에서 배운 Context-free 언어의 membership 문제의 non-deterministic 좌파서, 우파서 방법(**알고리즘**)은 non-deterministic $O(n)$이므로, **NP**이지만, CYK **알고리즘**은 deterministic $O(n^3)$으로 **P**이다.

Non-deterministic 좌파서, 우파서 방법의 deterministic 방법인 LL이나 LR파서는 optimal인 deterministic $O(n)$이나, 이는 모든 context-free 문법에 해당되는 것이 아니고 부분집합인 LL문법이나 LR문법에만 가능하므로 문제의 복잡도를 $O(n^3)$에서 $O(n)$로 줄였다고 볼 수는 없다. 다만 우리가 사용하는 대부분의 context-free 문법이 LR 문법이므로 실용적으로는 널리 사용된다[4].

어떤 문제가 **NP**라는 최소한의 증명은 값 하나가 주어진 문제조건에 맞는가, 아닌가를 확인(verify)하는데 $O(p(n))$시간 걸리고, 이러한 값들이 여러 개 있는 경우, 이를 non-deterministic하게 확인(verify)하면 되므로, **NP**라고 주장할 수 있다.

(정의) **Polynomial time reduction**(PTR)
문제 P_1의 **모든** yes-instance Y_{P_1}과 no-instance N_{P_1}을 문제 P_2의 모든 yes-instance Y_{P_2}와 no-instance N_{P_2}로 바꾸어 주는 함수 h: $D_1 \rightarrow D_2$[5]가 존재하고, 함수 h를 수행하기 위한 **알고리즘**이 **다항식**($p(n)$) 이하일 때, 문제 P_1의 복잡도를 문제 P_2로 복잡도로 **다항식 시간 안에서 줄였다**고 하고(**Polynomial time reduction**), $P_1 \leq_{PTR} P_2$로 쓴다.

(사실) $P_1 \leq_{PTR} P_2$라고 하자.

$$P_2 \in \mathbf{P} \Rightarrow P_1 \in \mathbf{P}.$$
$$P_1 \notin \mathbf{P} \Rightarrow P_2 \notin \mathbf{P}.$$

(정의) 문제 P가 아래 조건을 만족하면 NP-complete문제라고 부른다.

---

2) 이러한 질문은 최광무교수가 NP-completness를 공부할 때 가졌던 개인적 의문이고, 전산학계에서 공인된 의견은 아니다.
3) **NP** 문제의 알려진 최소 복잡도 알고리즘은 decision tree를 이용하여 여러(k) 개의 clone들을 반복적으로 배당하는 exponential 복잡도이다. Nondeterminism의 최대값이 k일 때, $O(k^{p(n)})$.
4) 이러한 도구로 yacc(Yet Another Compiler-Compiler)이라는 도구가 unix환경에 있는데, 사용하는 프로그램언어에 따라, mlyacc등이 있고 이용되는 파서는 LR 파서의 부분집합인 $LALR$ 파서이다.
5) $Y_{P_1} \subseteq Y_{P_2} \wedge N_{P_1} \subseteq N_{P_2}$이므로 P_2가 P_1보다 더 어렵거나 느리거나 같다.

(1) $P \in$ **NP**.
(2) $\forall P' \in$ **NP**, $P' \leq_{PTR} P$.

즉 NP-complete는 **NP** 중 **가장 어려운** 문제를 말한다.

NP-complete 정의에 **씨앗**이 되는 **이해하기 쉬운** NP-complete 문제는, n개의 변수를 가진 논리식에서 그 결과를 **true**로 만드는 n개의 변수 값에 할당(assignment; 각 변수 값이 **true**나 **false**로 배당된다)가 있는가, 없는가를 묻는 **satisfiability**(SAT)문제이다. NP-complete 정의는 **모든 NP** 문제를 SAT로 polynomial-time reduction하며 시작[6]한다 ($\forall P \in$ **NP**, $P \leq_{PTR} SAT$). 즉 SAT 문제는 **최초**의 NP-complete 문제이다.

(정리 10.4) P가 NP-complete이고, $P \leq_{PTR} P_1$이면 P_1도 NP-complete이다.
(증명) $\forall P' \in$ **NP**, $P' \leq_{PTR} P$, $P \leq_{PTR} P_1$. $\therefore$ $P' \leq_{PTR} P_1$.

정리 10.4는 **새로운 문제** P_1이 **NP-complete**임을 증명하는데 쓰이는 정리이다. 이 때 P는 **씨앗** NP-complete 문제인 SAT이어도 좋으나($SAT \leq_{PTR} P_1$), 이미 **씨앗** SAT를 이용하여 NP-complete임이 이미 증명된($SAT \leq_{PTR} P$) 다른 문제 P이어도($P \leq_{PTR} P_1$) 좋다[7].

(정리 10.5) ∃ P가 NP-complete이고, $P \in$ **P**이면 **P** = **NP**이다.
(증명) $\forall P' \in$ **NP**, $P' \leq_{PTR} P \in$ **P**, **NP** $\subseteq$ **P**. $\therefore$ **P** = **NP**.

NP 문제의 **알려진 최소**(worst-case)[8] 복잡도가 exponential[9]이므로 **intractable**로 볼 수 있으나, **NP**중 가장 어려운 NP-complete 문제 중 **하나**만 **P**로 밝혀지면 **모든 NP** 문제는 **P**이므로 **NP**는 **tractable**이라고 주장할 수 있다[10].

(정의) 문제 P가 아래 조건을 만족하면 NP-hard문제라고 부른다.
(2) $\forall P' \in$ **NP**, $P' \leq_{PTR} P$.

NP-hard는 **NP**는 아닐 수 있지만, NP-complete보다 더 **어려운** 문제를 말한다.

따라서 정리 10.5의 증명과 같이 어떤 NP-hard인 문제 P가 **P**임이 밝혀져도 **P** = **NP**이고 **tractable**이라고 주장할 수 있다.

---

6) 다음 장에서 나올 Cook's Theorem의 증명
7) 좌교수님, 이 문장 맞나요? 혹은 꼭 필요한가요?
8) 좌교수님, 이 표현 적절한가요?
9) Decision tree를 생각하자.
10) 그러나 그런 증명은 **아직** 없다.