SNU 046.016 컴퓨터과학이 여는 세계 (Computational Civilization) Part III

Prof. Kwangkeun Yi

Department of Computer Science & Engineering

차례

이전

이전

다음

소프트웨어

컴퓨터(universal turing machine)라는
"마음의 도구"를 다루는 방법

- 소프트웨어(turing machine): 지혜와 언어로 짜고 짓는
- 소프트웨어는 무섭다: 구체적인 실천

소프트웨어

사람이	기계가
만들고	실행하고

소프트웨어를 잘 만들 방법

알고리즘(algorithm)	언어(language)
일하는 방도	표현하는 방식
푸는 솜씨	담는 그릇
박차	고삐

개괄

- 푸는 솜씨, 알고리즘과 복잡도
- 담는 그릇, 언어와 논리

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법

- 담는 그릇, 언어와 논리

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법
  - 비용의 계층
- 담는 그릇, 언어와 논리

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법
  - 비용의 계층
  - 컴퓨터로 풀 수 있는 쉬운 문제와 어려운 문제의 경계
- 담는 그릇, 언어와 논리

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법
  - 비용의 계층
  - 컴퓨터로 풀 수 있는 쉬운 문제와 어려운 문제의 경계
- 담는 그릇, 언어와 논리
  - 언어의 계층. 번역과 실행

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법
  - 비용의 계층
  - 컴퓨터로 풀 수 있는 쉬운 문제와 어려운 문제의 경계
- 담는 그릇, 언어와 논리
  - 언어의 계층. 번역과 실행
  - 프로그래밍 언어의 두개의 중력

개괄

- 푸는 솜씨, 알고리즘과 복잡도
  - 컴퓨터 문제 풀이법
  - 비용의 계층
  - 컴퓨터로 풀 수 있는 쉬운 문제와 어려운 문제의 경계
- 담는 그릇, 언어와 논리
  - 언어의 계층. 번역과 실행
  - 프로그래밍 언어의 두개의 중력
  - 프로그래밍 = 논리증명

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 = 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 = 문제풀이를 자동으로 진행

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 $=$ 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 $=$ 문제풀이를 자동으로 진행

묻게된다:

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 = 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 = 문제풀이를 자동으로 진행

묻게된다:

- ∃더 좋은(싸고 빠른) 방법?

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 = 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 = 문제풀이를 자동으로 진행

묻게된다:

- $\exists$더 좋은(싸고 빠른) 방법?
- 있다면 얼마나 더 좋은가?

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 = 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 = 문제풀이를 자동으로 진행

묻게된다:

- ∃더 좋은(싸고 빠른) 방법?
- 있다면 얼마나 더 좋은가?
- 더 좋기는 불가능한가?

알고리즘

알고리즘은 컴퓨터로 돌릴 수 있는 문제푸는 방도

- 하나의 소프트웨어 = 하나의 튜링기계
- 소프트웨어(튜링기계)는 어떤 문제를 푸는 것
- 컴퓨터는 소프트웨어를 실행 = 문제풀이를 자동으로 진행

묻게된다:

- ∃더 좋은(싸고 빠른) 방법?
- 있다면 얼마나 더 좋은가?
- 더 좋기는 불가능한가?
- 경계는 어디인가?
  - 현실적인 비용으로 해결가능한 문제들
  - 그렇지 않은 문제들

알고리즘 예

알고리즘 예

- 책읽기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다.

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다.
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다.
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기
- 뒤죽박죽인 숫자를 크기순서로 나열하기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다.
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기
- 뒤죽박죽인 숫자를 크기순서로 나열하기
- 패스워드 해킹하기

알고리즘 예

- 책읽기
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기
- 주머니의 숫자중에서 제일 큰 숫자를 찾기
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다.
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기
- 뒤죽박죽인 숫자를 크기순서로 나열하기
- 패스워드 해킹하기
- 자연수 인수분해하기

알고리즘 복잡도

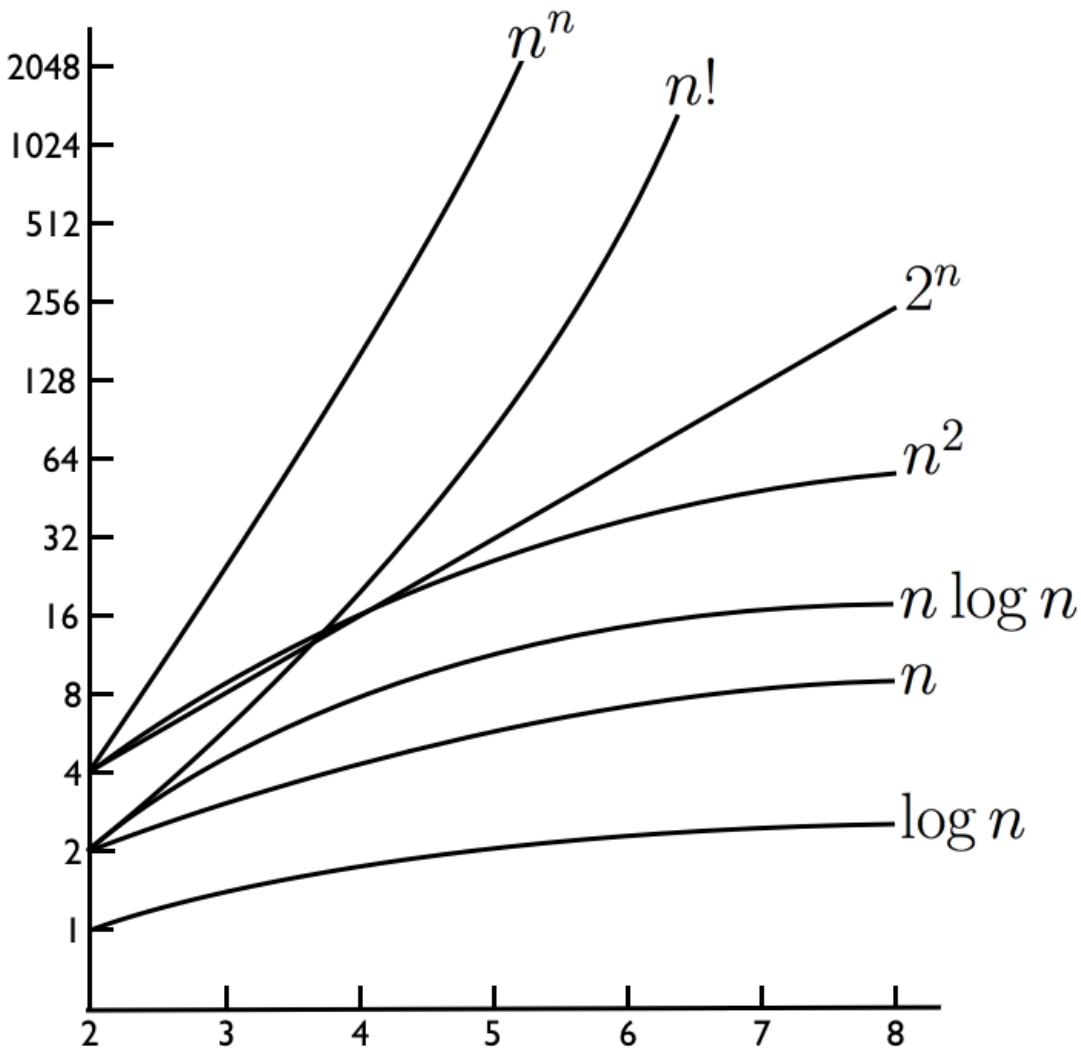
- 시간과 메모리소모량
- 입력크기의 함수로
- 결국 어떻게 되는지(asymptotic complexity)

알고리즘 복잡도

- 시간과 메모리소모량
- 입력크기의 함수로
- 결국 어떻게 되는지(asymptotic complexity)

n	n^2	2^n	$n!$
1	1	2	1
2	4	4	2
3	9	8	6
4	16	16	24
5	25	32	120
6	36	64	720
7	49	128	5,040
8	64	256	40,320
9	81	512	362,880
10	100	1,024	3,628,800
20	400	1,048,576	2,432,902,008,176,640,000
30	900	1,073,741,824	너무 크다! $> 10^{39}$

복잡도 증가 속도


n^n
$n!$
2^n
n^2
$n \log n$
n
$\log n$
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8

알고리즘 복잡도 표기법

"$O(f(n))$" (입력크기 n) = $f(n)$의 상수배보다 작다는 뜻

복잡도	알고리즘 스텝횟수가
$O(1)$	상수이면
$O(\log n)$	$\log n + 10$, $9871 \log n + 100$ 등이면
$O(n)$	$n + 10$, $999n + 9$, $2016n - 18$등이면
$O(n^2)$	$999n^2 + 9$, $2016n^2 + 199000n - 18$등이면

알고리즘 복잡도 표기법

"$O(f(n))$" (입력크기 n) = $f(n)$의 상수배보다 작다는 뜻

복잡도	알고리즘 스텝횟수가
$O(1)$	상수이면
$O(\log n)$	$\log n + 10$, $9871 \log n + 100$ 등이면
$O(n)$	$n + 10$, $999n + 9$, $2016n - 18$등이면
$O(n^2)$	$999n^2 + 9$, $2016n^2 + 199000n - 18$등이면

복잡도	n이 2배가되면	일컫기를
$O(1)$	변함없다	*상수constant*
$O(\log n)$	약간의 상수만큼 커진다	*로그logarithmic*
$O(n)$	2배 커진다	*선형linear*
$O(n \log n)$	2배보다 약간 더 커진다	*엔로그엔n log n*
$O(n^2)$	4배 커진다	*제곱배quadratic*
$O(n^k)$ (상수 k)	2^k배 커진다	*k승배, 다항polynomial*
$O(k^n)$ (상수 $k > 1$)	k^n배 커진다	*기하급수배exponential*
$O(n!)$	n^n배정도 커진다[2]	*계승배factorial*

알고리즘 복잡도 예

알고리즘 복잡도 예

- 책읽기: $O(n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다: $O(n)$, $O(\log_2 n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다: $O(n)$, $O(\log_2 n)$
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기: $O(n)$, $O(\log_2 n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다: $O(n)$, $O(\log_2 n)$
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기: $O(n)$, $O(\log_2 n)$
- 뒤죽박죽인 숫자를 크기순서로 나열하기: $O(n^2)$, $O(n \log_2 n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다: $O(n)$, $O(\log_2 n)$
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기: $O(n)$, $O(\log_2 n)$
- 뒤죽박죽인 숫자를 크기순서로 나열하기: $O(n^2)$, $O(n \log_2 n)$
- n자리 자연수 패스워드 해킹하기: $O(10^n)$

알고리즘 복잡도 예

- 책읽기: $O(n)$
- 자연수 $n(\geq 2)$이 주어졌을 때 $1 \times 2 \times \cdots \times n$을 계산하기: $O(n)$
- 주머니의 숫자중에서 제일 큰 숫자를 찾기: $O(n)$
- 장보기 목록에 있는 모든걸 장바구니에 담았는지 확인하기: $O(n^2)$
- 그녀가 마음속에 품은 자연수 알아맞추기. 1부터 n 사이에 있다. 그녀는 예/아니오만 답할 수 있다: $O(n)$, $O(\log_2 n)$
- $a^n = \underbrace{a \times \cdots \times a}_{n}$ 계산하기: $O(n)$, $O(\log_2 n)$
- 뒤죽박죽인 숫자를 크기순서로 나열하기: $O(n^2)$, $O(n \log_2 n)$
- n자리 자연수 패스워드 해킹하기: $O(10^n)$
- n자리 자연수 인수분해하기: $O(10^n)$

현실적(n^k) vs 비현실적(k^n)

n: 입력의 크기, k: 상수

- $O(n^k)$: 컴퓨터 성능이 곱하기로 빨라지면, 같은 시간에 처리할 수 있는 입력의 크기도 곱하기로 는다.
- $O(k^n)$: 컴퓨터 성능이 곱하기로 빨라져도, 같은 시간에 처리할 수 있는 입력의 크기는 더하기로 는다.

k^n의 무시무시함: $2^{100} \simeq 10^{30}$초 $>$ 우주의 나이

$\mathcal{P}$ 클래스 문제

그 문제를 푸는 알고리즘의 복잡도가 $O(n^k)$인 문제

- 알고리즘 복잡도:

$$O(\log n), O(n \log n), O(n), O(n^{1.5}), O(n^{28}), \cdots$$

- "현실적인 비용"으로 컴퓨터가 풀 수 있는 문제

실제로는

- "현실적"이려면: $O(n^3)$ 이하여야
- 일단 $O(n^k)$가 찾아졌다면, 우리는 늘 k를 3이하로 줄일 수 있었다.

컴퓨터로 풀려고 할 때의 질문

컴퓨터로(자동으로) 풀기 어려운 문제인지 아닌지를 미리 알고 싶다

컴퓨터에게 쉽고 어려운 문제의 경계?

- 쉬운 문제: $\mathcal{P}$ 클래스 문제.
- 어려운 문제: $\mathcal{P}$ 클래스 바깥의 문제.

컴퓨터로 풀려고 할 때의 질문

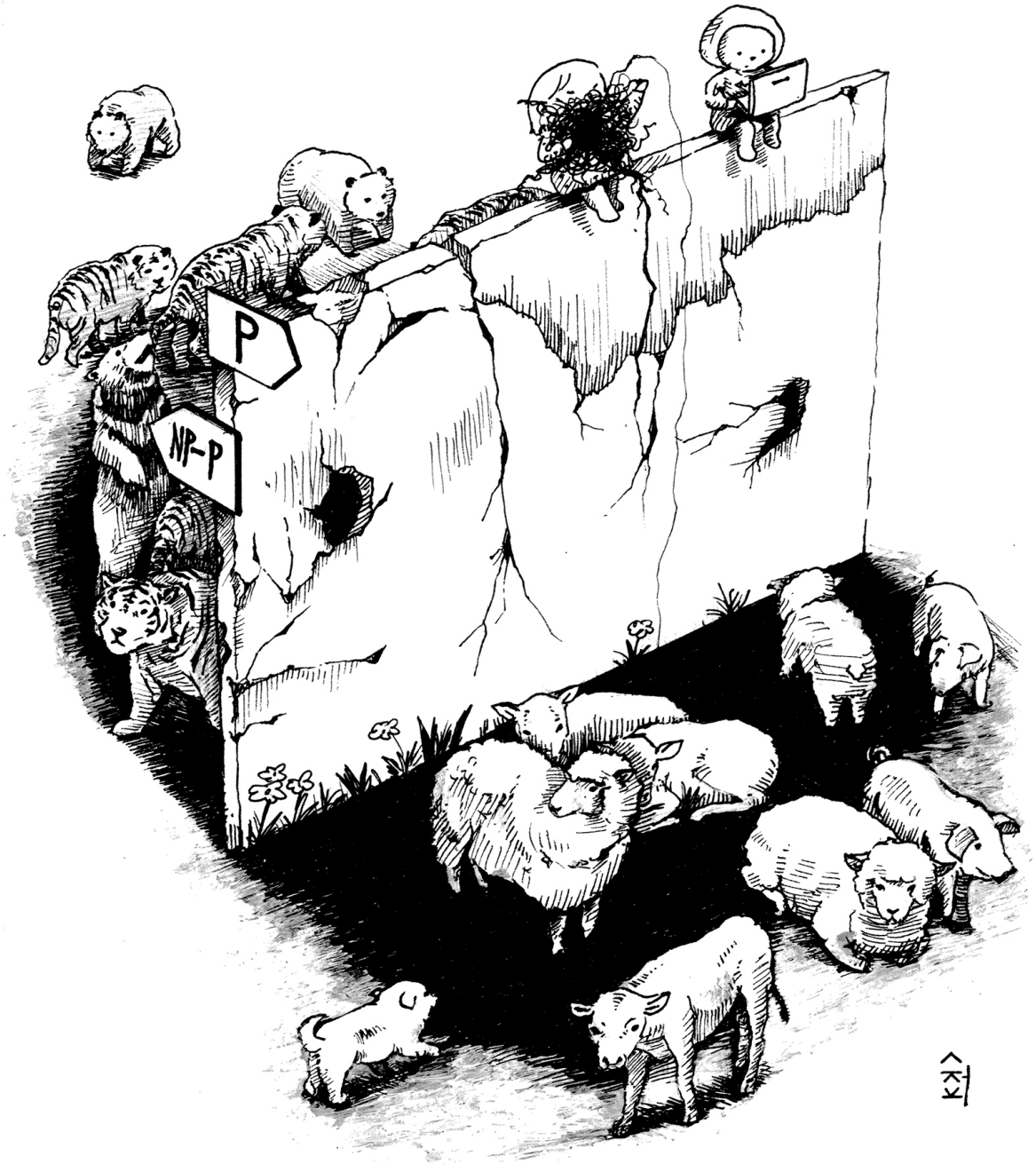
컴퓨터로(자동으로) 풀기 어려운 문제인지 아닌지를 미리 알고 싶다

컴퓨터에게 쉽고 어려운 문제의 경계?

- 쉬운 문제: $\mathcal{P}$ 클래스 문제.
- 어려운 문제: $\mathcal{P}$ 클래스 바깥의 문제.

어려운 문제인지를 판별하는 방법?

- 알고리즘 찾아나선다
- $\mathcal{P}$ 알고리즘이 찾아졌다면 쉬운 문제다
- $\mathcal{P}$ 알고리즘을 못찾겠다면?
  - 계속 열심히 찾으면 나올까?
  - 아예 그런 알고리즘은 없는 것일까?


P
NP-P

어려운 문제인지 판별하는 방법?

- 어려운 문제 $=$ $\mathcal{P}$ 클래스 바깥의 문제
- $\mathcal{P}$ 바깥의 문제가 존재하나?

그 쪽 경계로 가까이 가보자.

경계더듬기: $\mathcal{P}$ 클래스 vs $\mathcal{NP}$ 클래스

$\mathcal{NP}$ 클래스 문제의 정의:

- 예-아니오 문제(decision problem)이고
- "예"라는 답을
- *운에 기대어 현실적인 비용*$_{\textit{non-deterministic polynomial}}$으로 할 수 있으면.

경계더듬기: $\mathcal{P}$ 클래스 vs $\mathcal{NP}$ 클래스

$\mathcal{NP}$ 클래스 문제의 정의:

- 예-아니오 문제(decision problem)이고
- "예"라는 답을
- *운에 기대어 현실적인 비용*$_{\textit{non-deterministic polynomial}}$으로 할 수 있으면.
- 그래서, 이렇게도 정의한다:
  - "예"라고 답한 근거를 받아서 그 근거가 맞는지를 현실적인 비용에 확인할 수 있는 문제
  - "답을 받아서" = "운에 기대어" 이므로

경계더듬기: $\mathcal{P}$ 클래스 vs $\mathcal{NP}$ 클래스

$\mathcal{NP}$ 클래스 문제의 정의:

- 예-아니오 문제(decision problem)이고
- "예"라는 답을
- *운에 기대어 현실적인 비용*$_{\textit{non-deterministic polynomial}}$으로 할 수 있으면.
- 그래서, 이렇게도 정의한다:
  - "예"라고 답한 근거를 받아서 그 근거가 맞는지를 현실적인 비용에 확인할 수 있는 문제
  - "답을 받아서" = "운에 기대어" 이므로
- 경계의 문제 = $\mathcal{NP}$지만 $\mathcal{P}$인지는 아직 모르겠는

$\mathcal{NP}$ 문제의 예

아주 흔하다

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)
- 주어진 예산으로 주어진 지도의 도시들을 다 방문하고 돌아올 수 있을까?

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)
- 주어진 예산으로 주어진 지도의 도시들을 다 방문하고 돌아올 수 있을까?
- 주어진 *부울식*$_{\textit{boolean formula}}$이 참이되게 할 수 있을까?

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)
- 주어진 예산으로 주어진 지도의 도시들을 다 방문하고 돌아올 수 있을까?
- 주어진 *부울식*$_{\textit{boolean formula}}$이 참이되게 할 수 있을까?
- 주어진 자연수를 인수분해하라

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)
- 주어진 예산으로 주어진 지도의 도시들을 다 방문하고 돌아올 수 있을까?
- 주어진 *부울식*$_{\textit{boolean formula}}$이 참이되게 할 수 있을까?
- 주어진 자연수를 인수분해하라
- 1차원의 아미노산 실을 접어서 3차원의 단백질 구조물을 만들라. 단, 만들어진 구조를 유지하는데 필요한 에너지가 어느 이하여야 한다

$\mathcal{NP}$ 문제의 예

아주 흔하다

- 주어진 지도위의 모든 도시들을 한 번씩만 방문하는 경로가 있을까? (*해밀턴 경로*$_{\textit{Hamiltonian path}}$)
- 주어진 예산으로 주어진 지도의 도시들을 다 방문하고 돌아올 수 있을까?
- 주어진 *부울식*$_{\textit{boolean formula}}$이 참이되게 할 수 있을까?
- 주어진 자연수를 인수분해하라
- 1차원의 아미노산 실을 접어서 3차원의 단백질 구조물을 만들라. 단, 만들어진 구조를 유지하는데 필요한 에너지가 어느 이하여야 한다
- 1000만 관객을 넘기는 영화를 제작하라. 제작중에 n번 디자인 선택을 해야한다

오리무중: $\mathcal{P} \neq \mathcal{NP}$?

아직 아무도 모름

- 정말 어려운 문제인지가 아리송한 경계의 문제들($\mathcal{NP}$)
- 이것들이 혹시 쉬운 문제($\mathcal{P}$)는 아닐까?

오리무중: $\mathcal{P} \neq \mathcal{NP}$?

아직 아무도 모름

- 정말 어려운 문제인지가 아리송한 경계의 문제들($\mathcal{NP}$)
- 이것들이 혹시 쉬운 문제($\mathcal{P}$)는 아닐까?

현재의 디지털 컴퓨터(튜링기계)로는 아마도

- $\mathcal{P} \neq \mathcal{NP}$ 라고 추측

오리무중: $\mathcal{P} \neq \mathcal{NP}$?

아직 아무도 모름

- 정말 어려운 문제인지가 아리송한 경계의 문제들($\mathcal{NP}$)
- 이것들이 혹시 쉬운 문제($\mathcal{P}$)는 아닐까?

현재의 디지털 컴퓨터(튜링기계)로는 아마도

- $\mathcal{P} \neq \mathcal{NP}$ 라고 추측

$\mathcal{P} \neq \mathcal{NP}$라면

- 어려운 문제의 시작/경계가 $\mathcal{NP}$이지 않을까
- 아슬아슬하므로: 운에 기대기만 하면 $\mathcal{P}$로 넘어가지만 $\mathcal{P}$는 될 수 없다니

현실적인 알고리즘은 없다고 판정하는 방법 (1/3)

$\mathcal{NP}$ 문제들의 온순한 성질때문에 가능:

$\mathcal{NP}$ 문제중 제일 어려운 문제가 존재한다!

Cook 1971년: "모든 $\mathcal{NP}$ 문제중에서 *부울식 만족시키기 문제*$_{satisfiability\ problem}$(SAT)가 제일 어렵다"

- 모든 $\mathcal{NP}$ 문제를 다항비용으로 *부울식 만족시키기 문제*$_{satisfiability\ problem}$로 각색해서 건너풀 수 있다

현실적인 알고리즘은 없다고 판정하는 방법 (2/3)

지렛대

- *건너풀기*$_{problem\ reduction}$
- 문제 A를 문제 B로 *건너풀 수 있다*$_{reducible}$ = B를 풀면 A가 풀린다
  - 예) "곱하기"를 "더하기"로
  - 예) "k-번째로 큰 수 찾기"를 "정열"로
- "B는 A보다 어렵다" (정확히는, "B는 적어도 A만큼 어렵다")

현실적인 알고리즘은 없다고 판정하는 방법 (3/3)

$\mathcal{NP}$ 문제들 중에서 제일 어려운 문제가 존재하므로

주의: 이 방법은 가정 $\mathcal{P} \neq \mathcal{NP}$ 아래에서 *믿을만*$_{sound}$ 하지만 *완전하진않다*$_{not\ complete}$

현실적인 알고리즘은 없다고 판정하는 방법 (3/3)

$\mathcal{NP}$ 문제들 중에서 제일 어려운 문제가 존재하므로

- 맞닥뜨린 문제가 그런 "왕초" 문제만큼 어렵다면

주의: 이 방법은 가정 $\mathcal{P} \neq \mathcal{NP}$ 아래에서 *믿을만*$_{sound}$ 하지만 *완전하진않다*$_{not\ complete}$

현실적인 알고리즘은 없다고 판정하는 방법 (3/3)

$\mathcal{NP}$ 문제들 중에서 제일 어려운 문제가 존재하므로

- 맞닥뜨린 문제가 그런 "왕초" 문제만큼 어렵다면
- 그리고 $\mathcal{P} \neq \mathcal{NP}$가 사실이라면 (그렇다고 추측)
  - 즉, $\mathcal{P}$ 바깥에 문제들이 존재한다면

주의: 이 방법은 가정 $\mathcal{P} \neq \mathcal{NP}$ 아래에서 *믿을만*$_{sound}$ 하지만 *완전하진않다*$_{not\ complete}$

현실적인 알고리즘은 없다고 판정하는 방법 (3/3)

$\mathcal{NP}$ 문제들 중에서 제일 어려운 문제가 존재하므로

- 맞닥뜨린 문제가 그런 "왕초" 문제만큼 어렵다면
- 그리고 $\mathcal{P} \neq \mathcal{NP}$가 사실이라면 (그렇다고 추측)
  - 즉, $\mathcal{P}$ 바깥에 문제들이 존재한다면
- 맞닥뜨린 문제는 $\mathcal{P}$ 바깥에 있는 것이 확실하다

주의: 이 방법은 가정 $\mathcal{P} \neq \mathcal{NP}$ 아래에서 *믿을만*$_{sound}$ 하지만 *완전하진않다*$_{not\ complete}$

현실적인 알고리즘은 없다고 판정하는 방법 (3/3)

$\mathcal{NP}$ 문제들 중에서 제일 어려운 문제가 존재하므로

- 맞닥뜨린 문제가 그런 "왕초" 문제만큼 어렵다면
- 그리고 $\mathcal{P} \neq \mathcal{NP}$가 사실이라면 (그렇다고 추측)
  - 즉, $\mathcal{P}$ 바깥에 문제들이 존재한다면
- 맞닥뜨린 문제는 $\mathcal{P}$ 바깥에 있는 것이 확실하다
- 즉, 맞닥뜨린 문제는 컴퓨터로 풀 수 있지만 현실적인 비용으로는 불가능하다

주의: 이 방법은 가정 $\mathcal{P} \neq \mathcal{NP}$ 아래에서 *믿을만*$_{sound}$ 하지만 *완전하진않다*$_{not\ complete}$

어쩔것인가?

주변의 흔한 문제들이 $\mathcal{NP}$ 문제들이다, 컴퓨터로 풀고싶다.

- 알고리즘의 조건 = 모든 입력에 정확한 답을 내기
- 조건을 느슨하게 하면 어떨까?
  - "모든 입력"을 "흔한 입력"으로
  - "정답"을 "적당한 답"으로

어쩔것인가?

주변의 흔한 문제들이 $\mathcal{NP}$ 문제들이다, 컴퓨터로 풀고싶다.

- 알고리즘의 조건 = 모든 입력에 정확한 답을 내기
- 조건을 느슨하게 하면 어떨까?
  - "모든 입력"을 "흔한 입력"으로
  - "정답"을 "적당한 답"으로
- 동원하는 기법

어쩔것인가?

주변의 흔한 문제들이 $\mathcal{NP}$ 문제들이다, 컴퓨터로 풀고싶다.

- 알고리즘의 조건 = 모든 입력에 정확한 답을 내기
- 조건을 느슨하게 하면 어떨까?
  - "모든 입력"을 "흔한 입력"으로
  - "정답"을 "적당한 답"으로
- 동원하는 기법
  - *통밥*$_{heuristic}$: 맞을듯한 직관. 선택 기로에서 기대어 지체없이 진행

어쩔것인가?

주변의 흔한 문제들이 $\mathcal{NP}$ 문제들이다, 컴퓨터로 풀고싶다.

- 알고리즘의 조건 = 모든 입력에 정확한 답을 내기
- 조건을 느슨하게 하면 어떨까?
  - "모든 입력"을 "흔한 입력"으로
  - "정답"을 "적당한 답"으로
- 동원하는 기법
  - *통밥*$_{heuristic}$: 맞을듯한 직관. 선택 기로에서 기대어 지체없이 진행
  - *무작위*$_{randomization}$: 통밥의 낭패를 막는 데 유효

통밥과 무작위

아이러니/미스테리

- *통밥*$_{heuristic}$과 *무작위*$_{randomization}$가 잘 작동
  - 흔한 경우에 정답에 가까운 답을 내 놓는다(통밥)
  - 드문 경우에만 낭패(통밥)
  - 비현실적인 비용의 덫에 걸리지 않는 페인팅모션 (무작위)
  - 알고리즘의 복잡도가 널뛰는 것을 안정시킨다(무작위)
- 왜그럴까?

컴퓨터로 불가능한 문제들

- 당연히 존재하겠지요
- 명확히 존재: 예) *멈춤문제*$_{halting\ problem}$
- 무수히 많다
  - 문제 1개 $=$ 함수 1개 (자연수에서 예-아니오로 가는)
    (입력과 답의 짝을 자연수 하나로 표현가능)
  - 그런 함수의 개수는 $2^{|\mathbb{N}|}$
  - 위의 개수는 자연수의 개수 $|\mathbb{N}|$(sw의 개수) 보다 무한히 많다

알고리즘 만들때의 흔한 기본기

- *모조리 훑기*$_{\textit{exhaustive search}}$
  - 최대공약수: $2, 3, 4, \cdots$로 나눠보기
- *되돌아가기*$_{\textit{backtracking}}$
  - 미로찾기: 모든 경우를 차례로하면서
- *나눠풀어 합치기*$_{\textit{divide-and-conquer}}$
  - 인구조사
- *기억하며 풀기*$_{\textit{dynamic programming}}$
  - 피보나치 수열 값
- *질러놓고 다듬기*$_{\textit{iterative improvement}}$
  - 로봇-대피소 짝짓기 (일단 짝짓고 겹친경로 바꾸기)
  - 정열 (일단 줄세우고 어긋난 이웃 바꾸기)

양자 컴퓨팅(quantum computing)

양자현상을 이용

- *중첩*$_{superposition}$
- *얽힘*$_{entanglement}$
- *확률진폭*$_{probability\ amplitude}$

양자 컴퓨팅(quantum computing)

양자현상을 이용

- *중첩*$_{superposition}$
- *얽힘*$_{entanglement}$
- *확률진폭*$_{probability\ amplitude}$

양자컴퓨터의 성질

- 튜링기계와 동일: 기계적인 계산의 한계를 넓히지는 못함
- 단, 훨씬 빨리(시간) 훨씬 알뜰하게(메모리) 계산

양자 알고리즘(quantum algorithm)

근본적으로 *확률 알고리즘*$_{probabilistic\ algorithm}$: 높은 확률로 답이 나타난다

- 데이터를 중첩시켜 연산 효율을 높임
- 답이 다른 후보들과 양자안에 중첩되어 있음
- 알고리즘 연산 $=$ 답이 매우 높은 확률진폭을 가지도록 하기
- 연산을 끝내고 양자를 관찰하면 매우 높은 확률로 답이 나타남

양자 인수분해(quantum factorization) 알고리즘

1994 Shor. 수학에서 알려진 사실을 이용: 자연수 N의 인수분해

1. N보다 작은 임의의 자연수 r을 선택
2. $GCD(r, N)$이 1이 아니면 N의 인수. 아니면,
3. 아래 숫자열을 계산

$$r^1 \bmod N,\ r^2 \bmod N,\ r^3 \bmod\ N, \cdots$$

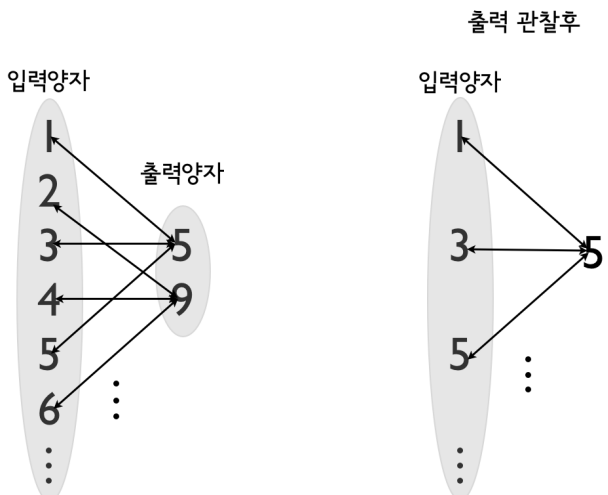
4. 위 숫자열에는 주기 p가 존재한다.
5. $GCD(N, r^{p/2}+1)$와 $GCD(N, r^{p/2}-1)$가 매우 큰 확률로 1이 아니다(N의 인수가 된다). (p가 홀수면 다시 반복)

이 과정을 양자 알고리즘으로 옮기자.

양자현상을 이용해서 구현

다항시간에 가능

1. 입력 큐빗: m개의 *큐빗*$_{qbit}$에 $\overbrace{0\cdots1}^{m}(1)$ 부터 $\overbrace{1\cdots1}^{m}(2^m-1)$까지 같은 확률로 중첩시킴
2. 출력 큐빗: 입력큐빗과 동일하게 중첩시킴
3. 입출력 큐빗에 서로 해당하는 숫자를 얽혀놓음
4. 출력 큐빗에 $r^i \bmod N$을 계산하는 레이저를 쏜다. i는 m개의 큐빗에 중첩되어 있는 $1,\cdots,2^m-1$. (이만큼의 숫자열 계산이 중첩덕분에 한꺼번에 가능)
5. 출력 큐빗을 관찰; 입력 큐빗에는 관찰한 결과를 만들었던 i들만 남는다
6. 숫자열 주기 $p=$ 입력 큐빗에 남은 것들의 차이. 이를 감지하는 레이저를 쏜다



양자 탐색(quantum search) 알고리즘

1996 Grove. 데이터가 아무렇게나 널려있는 경우 $O(\sqrt{n})$에 가능

- 데이터는 일렬번호가 있고, 찾고자 하는 것은 주어진 번호에 해당하는 데이터라고 하자
- 데이터들과 일렬번호들은 두 양자에 따로 중첩해 넣는다

  (n개의 데이터는 $\log_2 n$개의 큐빗에 중첩해서 저장가능)
- 데이터와 해당 일렬번호를 두 양자사이에 얽혀놓는다

준비 끝

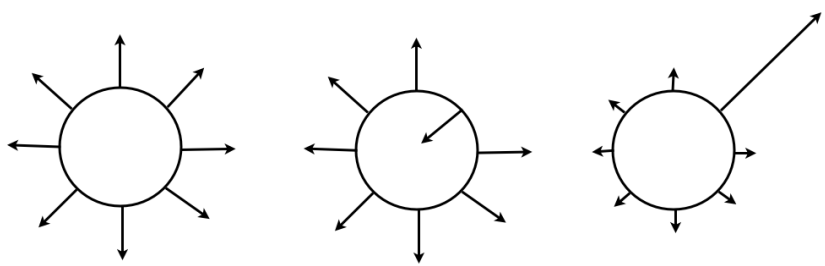
양자 탐색 본 과정

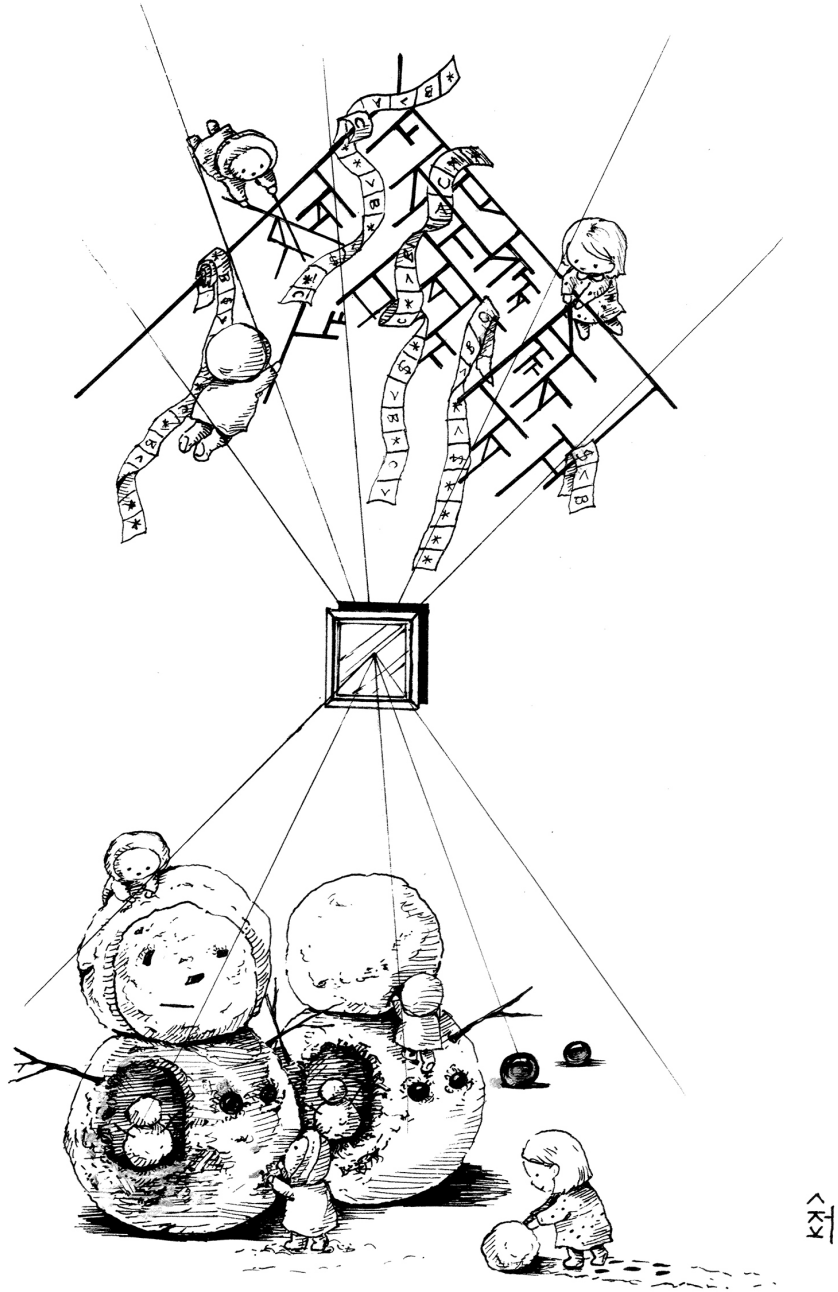
아래 과정

을 $O(\sqrt{n})$번 반복하고 관찰하면 매우 높은 확률로 답이 나타남

(n의 경우마다 예를들어 $\sqrt{n}-1$번, $10\sqrt{n}+4$번 등을 반복하면 충분하다는 뜻)

1. 일렬번호가 중첩되있는 큐빗에 "레이저"를 쏴서, 찾고자 하는 번호만 그 *확률진폭*probability amplitude을 반대 방향으로 뒤집는다
2. 그렇게 변한 확률진폭의 평균을 구한 후, (대략적으로) 평균의 두배에서 각 확률진폭을 다시 빼는 "레이저"를 쏜다
3. 그러면 찾고자 하는 일렬번호의 확률진폭은 증가하고 다른 것은 줄어든다





언어(language)

소프트웨어: 사람이 만들고 컴퓨터가 실행

- SW의 표현
  - 튜링기계 부품들(유한개의 상태, 유한개의 심볼, 유한개의 규칙표)로
  - 폰노이만 기계어로
  - 인텔(x86) 기계어로
  - 보다 상위의 언어들로
- SW를 표현하는 언어: "프로그래밍 언어"(programming language)
  - 튜링기계의 규칙표 혹은 기계어는 너무 하위의 언어
  - 나날이 복잡해지는 소프트웨어를 표현하는 그릇으로는 무리
  - 분자배열로 짜장면을 표현하는 셈

큰 간격

소프트웨어: 사람이 만들고 컴퓨터가 실행

큰 간격: 컴퓨터가 이해하는 언어 vs 사람이 편히쓰는 언어

- 컴퓨터가 이해하는 언어 (기계어)
  - Mac: “PowerPC” 기계어
  - PC: “Pentium” 기계어
  - Galaxy: “Arm” 기계어
  - iPhone: “Arm” 기계어
- 모두 *튜링-완전*$_{turing\text{-}complete}$
- 그러나 사람에겐 너무 원시적

해결책

언어계층과 자동번역

- 언어의 계층: 상위의 상위의 언어를 고안
- 그 언어들 사이의 "번역사슬"
- 번역사슬의 맨 위는 사람에게 편한 상위의 언어
- 번역사슬의 맨 아래는 기계에게 편한 "기계어"

Coq

```
Fixpoint sum (n:nat) :=
match n with
| O => O
| S m => sum m + S m
end.

Theorem sum_property:
  forall n:nat, 2 * sum n = n * n + n.
Proof.
induction n.
- (* base case: 2 * sum 0 = 0 * 0 + 0 *)
  reflexivity.
- (* inductive step *)
  simpl. rewrite <- plus_n_O. rewrite plus_assoc.
  rewrite plus_comm.
  replace (sum n + S n + sum n) with (sum n + sum n + S n)
    by omega.
  replace (sum n + sum n) with (2 * sum n) by omega.
  rewrite IHn.
  replace (n * S n) with (n * n + n)
    by (rewrite mult_n_Sm;omega). omega.
Qed.
```

ML

```
let rec qsort l =
  let rec split pivot l (l1, l2) =
    match l with
    | [] -> (l1, l2)
    | x::l' ->
      if x <= pivot then
        split pivot l' (x::l1, l2)
      else
        split pivot l' (l1, x::l2)
  in
  match l with
  | [] -> []
  | pivot::l' ->
    let (less, greater) = split pivot l' ([],[]) in
    qsort less @ (pivot :: qsort greater)
```

Scala

```
object Qsort {
  def sort(arr: Array[Int]) : Array[Int] = {
    def pivot = arr(0)
    val left = new ArrayBuffer[Int]()
    val right = new ArrayBuffer[Int]()
    if (arr.length <= 1) arr
    else {
      for (i <- 1 to arr.length - 1) {
          if(arr(i) <= pivot) left += arr(i)
          else right += arr(i)
      }
      (sort (left.toArray)) ++ (pivot+:(sort (right.toArray)))
    }
  }
  def main(args: Array[String]) {
    def arr = Array(3, 6, 4, 1, 9, 2, 8, 7, 5)
    def sorted = sort(arr)
    for (x <- sorted) {
      println(x)
    }
  }
}
```

Python

```
def part(l, pivot):
    left = []
    right = []
    while(l != []):
        head = l.pop(0)
        if head <= pivot:
            left = [head] + left
        else:
            right = [head] + right
    return (left, right)

def qsort(l):
    if l == []:
        return l
    else:
        pivot = l[0]
        left, right = part(l[1:], pivot)
        return qsort(left) + [pivot] + qsort(right)
```

Java

```
class Qsort {
    private int arr[];
    public Qsort(int a[]) { arr = a; }
    private void swap(int i, int j) {
        int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
    }
    public void sort(int li, int ri) {
        int pivot, i, j;
        if(li >= ri) return;
        pivot = arr[li]; i = li + 1; j = ri;
        while(i < j) { // split
            if(arr[i] > pivot) {
                swap(i, j);
                --j;
            }
            else ++i;
        }
        if(arr[i] > pivot) --i;
        swap(li, i);

        sort(li, i-1);
        sort(i+1, ri);
    }
```

C

```c
#include<stdio.h>

void swap(int *p, int *q) {
    int temp = *p; *p = *q; *q = temp;
}

void qsort(int *arr, int size) {
    int pivot, i, j;
    if(size <= 1) return;
    pivot = arr[0]; i = 1; j = size - 1;
    while(i < j) { // split
        if(arr[i] > pivot) {
            swap(&arr[i], &arr[j]);
            --j;
        }
        else ++i;
    }
    if(arr[i] > pivot) --i;
    swap(&arr[0], &arr[i]);

    qsort(arr, i);
    qsort(arr + i + 1, size - i - 1);
}
```

Dalvik

```
[000270] Qsort.sort:(II)V
0000: if-lt v5, v6, 0003 // +0003
0002: return-void
0003: iget-object v0, v4, LQsort;.arr:[I // field@0000
0005: aget v2, v0, v5
0007: add-int/lit8 v0, v5, #int 1 // #01
0009: move v1, v6
000a: if-ge v0, v1, 001b // +0011
000c: iget-object v3, v4, LQsort;.arr:[I // field@0000
000e: aget v3, v3, v0
0010: if-le v3, v2, 0018 // +0008
0012: invoke-direct {v4, v0, v1}, LQsort;.swap:(II)V // method@0004
0015: add-int/lit8 v1, v1, #int -1 // #ff
0017: goto 000a // -000d
0018: add-int/lit8 v0, v0, #int 1 // #01
001a: goto 000a // -0010
001b: iget-object v1, v4, LQsort;.arr:[I // field@0000
001d: aget v1, v1, v0
001f: if-le v1, v2, 0023 // +0004
0021: add-int/lit8 v0, v0, #int -1 // #ff
0023: invoke-direct {v4, v5, v0}, LQsort;.swap:(II)V // method@0004
0026: add-int/lit8 v1, v0, #int -1 // #ff
0028: invoke-virtual {v4, v5, v1}, LQsort;.sort:(II)V // method@0003
002b: add-int/lit8 v0, v0, #int 1 // #01
002d: invoke-virtual {v4, v0, v6}, LQsort;.sort:(II)V // method@0003
0030: goto 0002 // -002e
```

x86

```
0000000000400560 <qsort>:
  400560:  55                         push   %rbp
  400561:  48 89 e5                   mov    %rsp,%rbp
  400564:  48 83 ec 20                sub    $0x20,%rsp
  400568:  48 89 7d e8                mov    %rdi,-0x18(%rbp)
  40056c:  89 75 e4                   mov    %esi,-0x1c(%rbp)
  40056f:  83 7d e4 01                cmpl   $0x1,-0x1c(%rbp)
  400573:  0f 8e dd 00 00 00          jle    400656 <qsort+0xf6>
  400579:  48 8b 45 e8                mov    -0x18(%rbp),%rax
  40057d:  8b 00                      mov    (%rax),%eax
  40057f:  89 45 fc                   mov    %eax,-0x4(%rbp)
  400582:  c7 45 f4 01 00 00 00       movl   $0x1,-0xc(%rbp)
  400589:  8b 45 e4                   mov    -0x1c(%rbp),%eax
  40058c:  83 e8 01                   sub    $0x1,%eax
  40058f:  89 45 f8                   mov    %eax,-0x8(%rbp)
  400592:  eb 46                      jmp    4005da <qsort+0x7a>
  400594:  8b 45 f4                   mov    -0xc(%rbp),%eax
  400597:  48 98                      cltq
  400599:  48 c1 e0 02                shl    $0x2,%rax
  40059d:  48 03 45 e8                add    -0x18(%rbp),%rax
  4005a1:  8b 00                      mov    (%rax),%eax
  4005a3:  3b 45 fc                   cmp    -0x4(%rbp),%eax
  4005a6:  7e 2e                      jle    4005d6 <qsort+0x76>
  4005a8:  8b 45 f8                   mov    -0x8(%rbp),%eax
  4005ab:  48 98                      cltq
```

번역기(compiler)

- 출발어 L, 도착어 L'
- L로 짜여진 모든 프로그램을 같은 일을 하는 L' 프로그램으로 번역
- 컴퓨터 언어들 사이의 번역은 항상 자동으로 가능
  - 단어의 뜻은 하나로 고정되 있다
  - 사용되는 정황에 따라 달라지지 않는다
- 번역의 원리: *조립식*$_{compositional}$*으로 불변성질*$_{invariant}$ 유지하기
  - 전체의 번역 결과는 부분의 번역결과를 가지고 조립
  - 조합이 쉽도록: 각 결과는 같은 조건(invariant)을 만족

자동번역 원리의 예

출발어

$$
\begin{array}{rcl}
E & ::= & n \\
 & | & E + E \\
 & | & E \times E \\
 & | & -E
\end{array}
$$

도착어

$$
\begin{array}{rcl}
C & ::= & push\ n \\
 & | & add \\
 & | & sub \\
 & | & mult \\
 & | & C.C
\end{array}
$$

해석실행(interpretation)

컴퓨터가 소프트웨어를 실행

- 컴퓨터를 보편만능이게 하는 핵심아이디어
- 보편만능의 튜링기계: 테입에 표현된 17개 심볼의 문자열(프로그램, 소프트웨어, 튜링기계)을 해석실행
- 해석실행: 일상에서 우리가 늘 하는(“1+2”)

해석실행(interpretation)

컴퓨터가 소프트웨어를 실행

- 컴퓨터를 보편만능이게 하는 핵심아이디어
- 보편만능의 튜링기계: 테입에 표현된 17개 심볼의 문자열(프로그램, 소프트웨어, 튜링기계)을 해석실행
- 해석실행: 일상에서 우리가 늘 하는("1+2")

컴퓨터는 *해석실행기*$_{interpreter}$

- 전기회로로 구현된(폰노이만 기계, Pentium, Arm) 실행기
- 번역사슬의 맨 아래 언어("기계어")의 실행기
- 메모리에 표현된 기계어 프로그램을 해석실행

해석실행(interpretation)

컴퓨터가 소프트웨어를 실행

- 컴퓨터를 보편만능이게 하는 핵심아이디어
- 보편만능의 튜링기계: 테입에 표현된 17개 심볼의 문자열(프로그램, 소프트웨어, 튜링기계)을 해석실행
- 해석실행: 일상에서 우리가 늘 하는("1+2")

컴퓨터는 *해석실행기*$_{interpreter}$

- 전기회로로 구현된(폰노이만 기계, Pentium, Arm) 실행기
- 번역사슬의 맨 아래 언어("기계어")의 실행기
- 메모리에 표현된 기계어 프로그램을 해석실행

소프트웨어 구현된 *해석실행기*$_{interpreter}$도 가능

- 해석실행기를 컴퓨터가 해석실행 (해석실행이 두 겹으로 겹침)

언어 정글

프로그래밍 언어의 두 개의 뿌리

- *튜링기계*$_{turing\ machine}$ (Alan Turing)

- *람다계산법*$_{lambda\ calculus}$ (Alonzo Church)

언어 정글

프로그래밍 언어의 두 개의 뿌리

- *튜링기계*$_{turing\ machine}$ (Alan Turing)
  - 상위로 상위로: 기계에 명령하는 언어
  - C, Java, C++, C#, JavaScript, Objective C, Python, PHP, Scala, Swift, Rust 등
- *람다계산법*$_{lambda\ calculus}$ (Alonzo Church)

언어 정글

프로그래밍 언어의 두 개의 뿌리

- *튜링기계*$_{\textit{turing machine}}$ (Alan Turing)
  - 상위로 상위로: 기계에 명령하는 언어
  - C, Java, C++, C#, JavaScript, Objective C, Python, PHP, Scala, Swift, Rust 등
- *람다계산법*$_{\textit{lambda calculus}}$ (Alonzo Church)
  - 상위로 상위로: 값을 계산하는 언어
  - ML, OCaml, Haskell, F#, Lisp, Clojure, Rua, Python, Scala, C++14 등

언어 정글

프로그래밍 언어의 두 개의 뿌리

- *튜링기계*$_{\textit{turing machine}}$ (Alan Turing)
  - 상위로 상위로: 기계에 명령하는 언어
  - C, Java, C++, C#, JavaScript, Objective C, Python, PHP, Scala, Swift, Rust 등
- *람다계산법*$_{\textit{lambda calculus}}$ (Alonzo Church)
  - 상위로 상위로: 값을 계산하는 언어
  - ML, OCaml, Haskell, F#, Lisp, Clojure, Rua, Python, Scala, C++14 등
- 최신 언어들은 두 방식 모두를 지원

기계의 중력 vs 람다의 중력

- 기계의 중력
  - 명령하며 기계상태를 변화시키는 주문
  - 요리법, 장보기목록

```
a ← 0
b ← read
a ← a+b
b ← b+1
```

기계의 중력 vs 람다의 중력

- 기계의 중력
  - 명령하며 기계상태를 변화시키는 주문
  - 요리법, 장보기목록

```
a ← 0
b ← read
a ← a+b
b ← b+1
```

- 람다의 중력
  - 기계는 없다. 값을 계산하는 식. "산수 스타일"
  - 초등학교때부터 보아온 산수 언어. $\sum_{n=1}^{10} n^2$
  - "$P \subset Q$ 이고 $P \neq Q$라고하자. 그러면 $P^c \cup Q = U$ 이고 $P \cap Q^c = \emptyset$ 이고 $P \cup Q^c \neq U$이다."
  - 프로그래밍 언어의 요건 $\supseteq$ 수학 언어의 요건

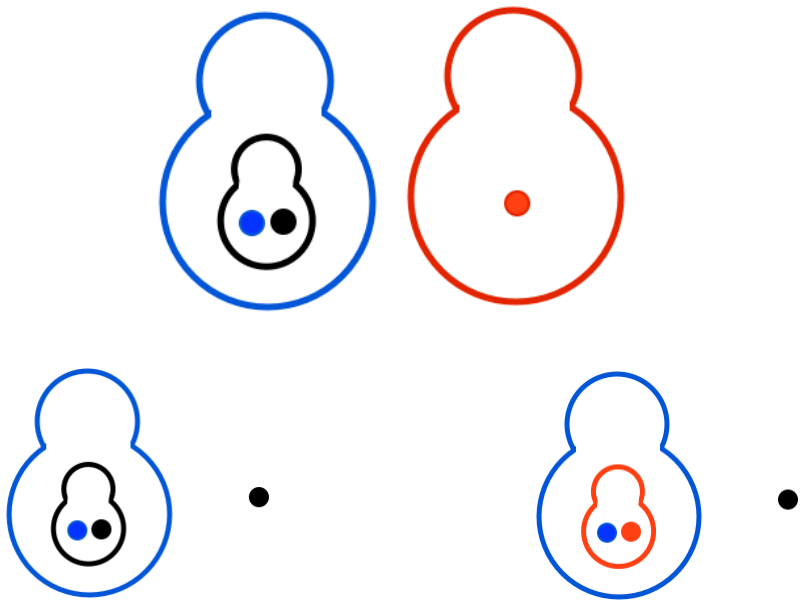
기계의 중력 vs 람다의 중력

- 기계의 중력
  - 명령하며 기계상태를 변화시키는 주문
  - 요리법, 장보기목록

```
a ← 0
b ← read
a ← a+b
b ← b+1
```

- 람다의 중력
  - 기계는 없다. 값을 계산하는 식. "산수 스타일"
  - 초등학교때부터 보아온 산수 언어. $\sum_{n=1}^{10} n^2$
  - "$P \subset Q$ 이고 $P \neq Q$라고하자. 그러면 $P^c \cup Q = U$ 이고 $P \cap Q^c = \emptyset$ 이고 $P \cup Q^c \neq U$이다."
  - 프로그래밍 언어의 요건 $\supseteq$ 수학 언어의 요건
- 다른 기원은 다른 방식의 생각을 유도하는 프로그래밍 언어를 탄생시킴

람다 계산법(lambda calculus)(1/2)



x	구슬(색깔대신 이름 x)
$\lambda x.E$	눈사람(색깔은 x)
$E\ E$	나란히 있기

람다 계산법(lambda calculus) (2/2)

$$
\begin{aligned}
\underline{3} &= \lambda s.(\lambda z.\underbrace{s(s(s\,z))}_{3}) \\
\underline{+}\,n\,m &= \lambda s.(\lambda z.ns(msz)) \\
\underline{\times}\,n\,m &= \lambda s.(\lambda z.n(ms)z) \\
\underline{T} &= \lambda x.(\lambda y.x) \\
\underline{F} &= \lambda x.(\lambda y.y) \\
\underline{and}\,a\,b &= a\,b\,\underline{F} \\
\underline{\mathsf{zero?}}\,n &= n\,(\lambda x.\underline{F})\,\underline{T} \\
\underline{\mathsf{repeat}}\,E &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\,E
\end{aligned}
$$

논리는 언어의 거울

(람다 중력권)

컴퓨터 세계에서 언어와 논리는 동전의 양면일 뿐, 같은 것이다

증명하기	$\longleftrightarrow$	프로그램짜기
"논리적인 비약없이 새로운 사실을 확인해가는 과정."	$\longleftrightarrow$	공짜없이 새로운 데이터를 만들어가는 과정.
"참인 사실 혹은 사실이라고 가정한 것들로부터 시작."	$\longleftrightarrow$	기초적인 데이터에서 부터 시작. 기초적인 데이터는 정수, 문자, 참거짓 등.
"사실을 기반으로 해서 새로운 사실들을 만듬."	$\longleftrightarrow$	이미 만든 데이터를 가지고 새로운 데이터들을 만듬.
"만들어가는 과정은 근거없는 건너뛰기 없이, 논리적으로 누구나 수긍하는 추론의 징검다리를 밟고 가는 과정만 있다."	$\longleftrightarrow$	새 데이터를 만드는 과정은 공짜가 없다. 사용하는 프로그래밍 언어에서 제공하는 프로그램 조립방식을 써서 만든다.

논리 추론의 징검다리

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$$\frac{\begin{array}{c}\overline{A} \\ \vdots \\ B\end{array}}{A \Rightarrow B} \qquad \frac{A \Rightarrow B \quad A}{B}$$

$$\frac{A}{A \vee B} \qquad \frac{A}{B \vee A} \qquad \frac{A \vee B \quad \begin{array}{c}\overline{A} \\ \vdots \\ C\end{array} \quad \begin{array}{c}\overline{B} \\ \vdots \\ C\end{array}}{C}$$

논리 증명나무

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (A \Rightarrow C)}}{A \Rightarrow B} \quad \dfrac{}{A}}{B} \quad \dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (A \Rightarrow C)}}{A \Rightarrow C} \quad \dfrac{}{A}}{C}}{B \wedge C}}{A \Rightarrow (B \wedge C)}}{(A \Rightarrow B) \wedge (A \Rightarrow C) \Rightarrow (A \Rightarrow (B \wedge C))}$$

거울: 증명하기 = 프로그램짜기 (1/2)

$$\frac{A \quad B}{A \wedge B} \longleftrightarrow \text{데이터 뭉치 만들기}$$

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B} \longleftrightarrow \text{데이터 뭉치 사용하기}$$

$$\frac{\begin{array}{c}\overline{A} \\ \vdots \\ B\end{array}}{A \Rightarrow B} \longleftrightarrow \text{함수 만들기}$$

$$\frac{A \Rightarrow B \quad A}{B} \longleftrightarrow \text{함수 사용하기}$$

거울: 증명하기 = 프로그램짜기 (2/2)

$$\frac{A}{A \vee B} \quad \longleftrightarrow \quad \text{데이터 뭉뚱그리기}$$

$$\frac{A \vee B \qquad \begin{array}{c}\overline{A} \\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B} \\ \vdots \\ C\end{array}}{C} \quad \longleftrightarrow \quad \text{뭉뚱그린 데이터 사용하기}$$

$$\overline{A} \quad \longleftrightarrow \quad \text{함수의 인자를 사용하기}$$

거울의 효능

- 프로그램 짜기전: 짤 프로그램의 구도잡기
  - *타입*$_{type}$이라는 언어로 겉 얼개를 표현
  - 참 논리식 $\longleftrightarrow$ 타입
- 프로그램 짜고나서: 짠 프로그램이 무난히 작동할지 살피기
  - 무난히 작동 $=$ 타입에 맞게 돈다
  - 논리분야의 타입이 활용되던 모습(1950-1970년대)
  - 그렇다면 프로그램에서도 가능하겠지(1970-현재)

짤 프로그램의 구도잡기

타입으로("보통명사"로) 프로그램 얼개를 표현

$$
\begin{aligned}
\text{결혼하기} &: \text{남자} \times \text{여자} \to \text{부부} \\
\text{사랑하기} &: \text{부부} \to \text{사랑} \\
\text{아기만들기} &: \text{부부} \times \text{사랑} \to \text{아기} \\
\text{가족만들기} &: \text{남자} \times \text{여자} \to \text{가족}
\end{aligned}
$$

```
가족만들기(x,y) =
    let  couple = 결혼하기(x,y)              ('남자'와 '여자'가 '부부'를 만들고)
    let  love = 사랑하기(couple)             ('부부'가 '사랑'을 만들고)
    let  baby = 아기만들기(couple, love)     ('부부'가 '사랑'으로 '아기'를 만들고)
     in  (couple, baby)                      (그런 '부부'와 '아기'가 '가족'이다)
```

짠 프로그램이 무난히 작동할지 검진하기 (1/2)

프로그래머의 불안

- 기획한 얼개에 따라 잘 메꾼걸까?
- 그래서 제대로 작동하는 프로그램일까?
- 만 페이지가 넘는, 대하소설 보다 복잡한 흐름
- 프로그램 실수는 고스란히 드러날 것이다

프로그램 검진 vs 다른 분야

만든것	프로그램	$\longleftrightarrow$	기계/건물/약물 디자인
실행기	컴퓨터	$\longleftrightarrow$	자연
바램	실행에대해 미리 확인	$\longleftrightarrow$	작동에대해 미리 확인
안심	“생각대로 돌것이다”	$\longleftrightarrow$	“생각대로 작동할것이다”
확인도구	컴퓨터과학의 성과	$\longleftrightarrow$	자연과학의 성과

짠 프로그램이 무난히 작동할지 검진하기 (2/2)

자동검진이 가능

- 과거: 개인의 기예에 의존하던 프로그램의 완성도
- 현재, 미래: 누구나의 기술로 한 발 전진
  - 자동 검진
- 컴퓨터는 더 이상 맹목적이지 않다
- 프로그램을 검진해서 통과한 것만 실행

$f(x) = x+1$ $\quad$ $sum(f) = f(0)+f(1)$ $\quad$ *무게(춘향이)+무게(그네)*

요약(abstraction)의 그물

컴퓨터과학에서 늘 동원하는 지혜: "모두 포섭하면서 간단히"

- 프로그램 검진에서도: 요약
  - 모든 디테일을 보는 것은 비현실적
  - 외부입력 가짓수가 너무 많고, 만들어지는 현상이 너무 복잡: *기하급수로 커지며 속수무책*combinatorial explosion

요약(abstraction)의 그물

컴퓨터과학에서 늘 동원하는 지혜: "모두 포섭하면서 간단히"

- 프로그램 검진에서도: 요약
  - 모든 디테일을 보는 것은 비현실적
  - 외부입력 가짓수가 너무 많고, 만들어지는 현상이 너무 복잡: *기하급수로 커지며 속수무책*$_{combinatorial\ explosion}$
- 다양한 수준의 요약을 동원:

$$10 \times \textit{무게}(\textit{춘향이}) + 8$$

요약(abstraction)의 그물

컴퓨터과학에서 늘 동원하는 지혜: "모두 포섭하면서 간단히"

- 프로그램 검진에서도: 요약
  - 모든 디테일을 보는 것은 비현실적
  - 외부입력 가짓수가 너무 많고, 만들어지는 현상이 너무 복잡: *기하급수로 커지며 속수무책*combinatorial explosion
- 다양한 수준의 요약을 동원:

$$10 \times \textit{무게}(\textit{춘향이}) + 8$$

  - 타입으로 요약: 정수

요약(abstraction)의 그물

컴퓨터과학에서 늘 동원하는 지혜: "모두 포섭하면서 간단히"

- 프로그램 검진에서도: 요약
  - 모든 디테일을 보는 것은 비현실적
  - 외부입력 가짓수가 너무 많고, 만들어지는 현상이 너무 복잡: *기하급수로 커지며 속수무책*$_{combinatorial\ explosion}$
- 다양한 수준의 요약을 동원:

$$10 \times \textit{무게}(\textit{춘향이}) + 8$$

  - 타입으로 요약: 정수
  - 양수음수로 요약: 양수

요약(abstraction)의 그물

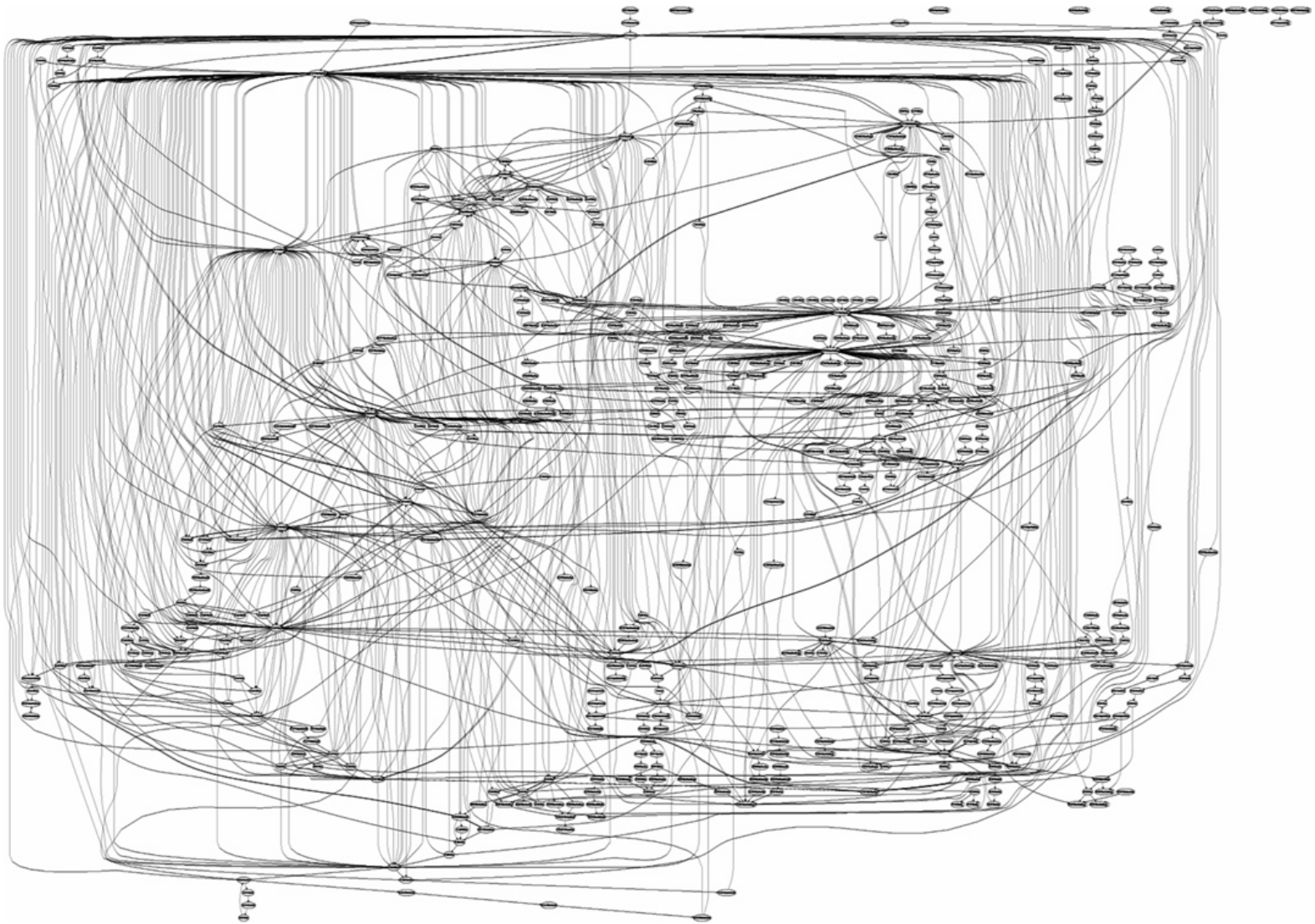
컴퓨터과학에서 늘 동원하는 지혜: "모두 포섭하면서 간단히"

- 프로그램 검진에서도: 요약
  - 모든 디테일을 보는 것은 비현실적
  - 외부입력 가짓수가 너무 많고, 만들어지는 현상이 너무 복잡: *기하급수로 커지며 속수무책*$_{\textit{combinatorial explosion}}$
- 다양한 수준의 요약을 동원:

$$10 \times \textit{무게}(\textit{춘향이}) + 8$$

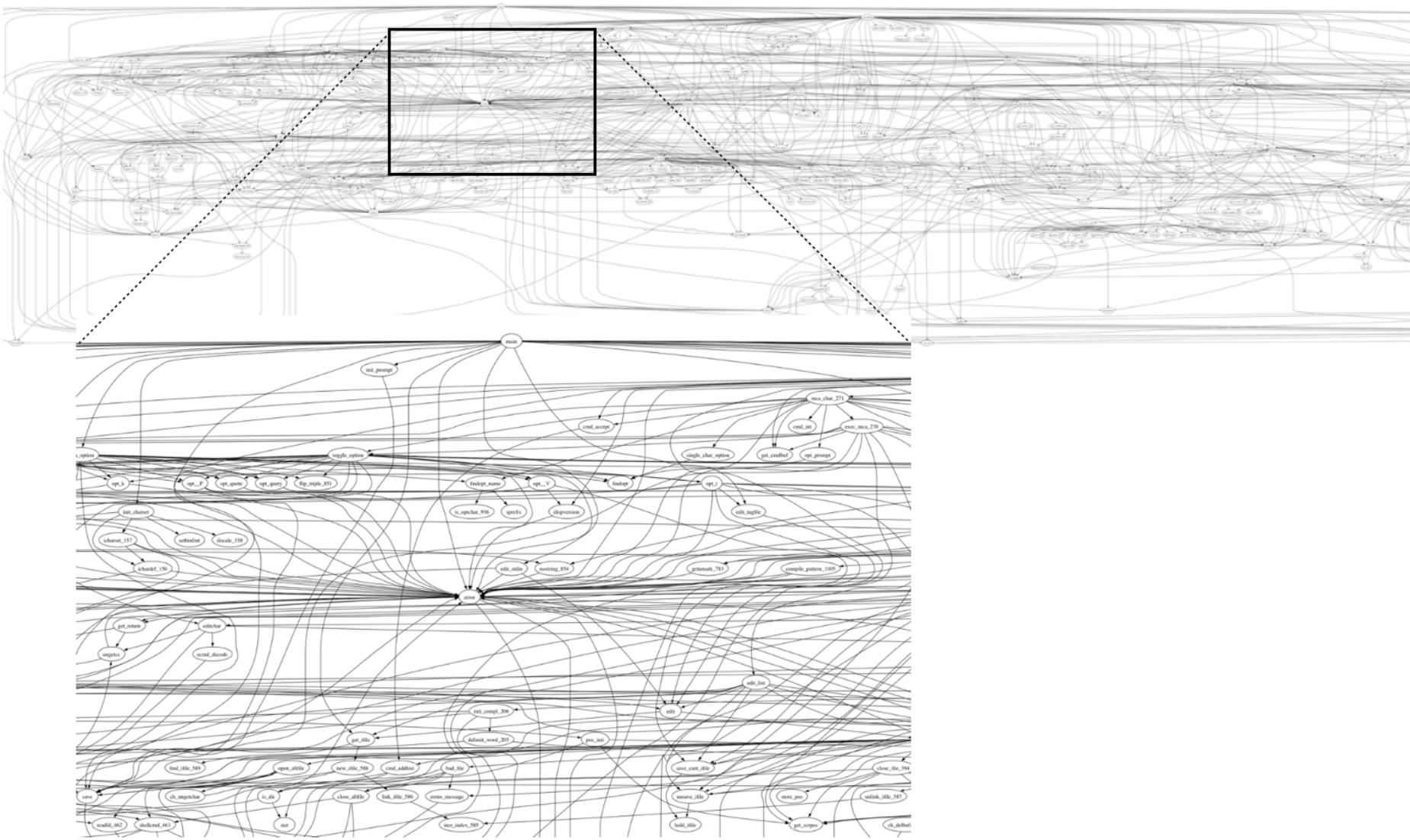
  - 타입으로 요약: 정수
  - 양수음수로 요약: 양수
  - 구간으로 요약: 100과 3000사이

SW는 복잡하다



SW는 복잡하다

less-382(23,822 LoC)



자연물 만큼 복잡

포유류 뇌의 1만개 뉴런의 넷트웍, Ecole Polytechnique Fédérale de Lausanne, *Blue Brain Project*, 2008

SW 제작의 어려움

- 프로그램의 규모와 복잡도가 점점커짐
  - 프로그램 복잡성의 증가속도 $>>$ hw 성능의 성장속도
  - “sw는 가스다.” “Software is gas.”

SW 제작의 어려움

- 프로그램의 규모와 복잡도가 점점커짐
  - 프로그램 복잡성의 증가속도 $>>$ hw 성능의 성장속도
  - "sw는 가스다." "Software is gas."
- 프로그램은 기계가 자동으로 실행함
  - 기계는 우리가 바라는 바를 실행하지 않음
  - 기계는 sw에 적힌 바를 실행할 뿐

    "장보기 = 우유 1리터, 신라면 4봉지, 그리고 쌀과자두 사오기."

  - 모든 상황을 고려해야 $\times$ 사소한 실수가 없어야

SW 제작의 어려움

- 프로그램의 규모와 복잡도가 점점커짐
  - 프로그램 복잡성의 증가속도 $>>$ hw 성능의 성장속도
  - "sw는 가스다." "Software is gas."
- 프로그램은 기계가 자동으로 실행함
  - 기계는 우리가 바라는 바를 실행하지 않음
  - 기계는 sw에 적힌 바를 실행할 뿐

    "장보기 = 우유 1리터, 신라면 4봉지, 그리고 쌀과자두 사오기."

  - 모든 상황을 고려해야 $\times$ 사소한 실수가 없어야
- 프로그램의 실행을 "미리 완벽히 알기"가 어려움
  - 자동으로는 불가능: 증명됨(1936년, Alan Turing)
  - 사람이 확인해가야: 다양한 도구로 비용절감
  - 현재의 기술수준: 걸음마 "3발짝", 초보수준

고백: 컴퓨터분야는 아직 미숙합니다

분야의 역사 $\sim$ 겨우 5-60년

그래서

- 미숙하기 때문에 $\Longrightarrow$ 기회가 많다.
- 미숙하기 때문에 $\Longrightarrow$ 지금 “Newton”, “Galileo”, “Curie”, 와 같이 살고있다.

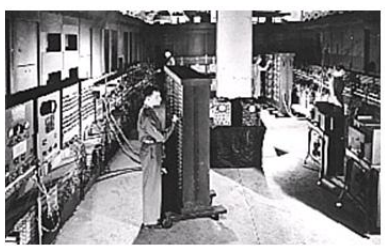
컴퓨터분야 발전 속도가 빠르다? (1/3)

- 컴퓨터속도: 10^7배/80년 발전

10^3flops/cpu ENIAC(1945)

$\longrightarrow$ (2×10^{10}flops/core IBM Sequoia(2010))

$\longrightarrow$ 6×10^{10}flops/core Fujitsu Fugaku(2020)



컴퓨터분야 발전 속도가 빠르다? (2/3)

다른 분야와 얼추 비슷: $10^7 \sim 10^9$배 발전/100년

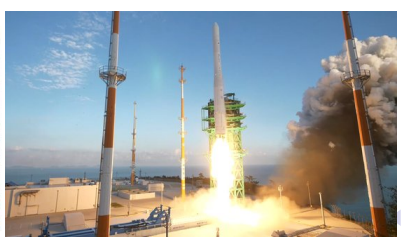
- 에너지분야: 10^7배/100년 발전

  10^{-1}hp/hr (하인) $\longrightarrow$ 1.35×10^6 hp/hr (원전 1GW/hr)

- 교통분야: 10^8배/100년 발전

  10^3j (가마) $\longrightarrow$ 10^{11}j (Delta II), 10^{10}j (누리호)





컴퓨터분야 발전 속도가 빠르다? (3/3)

- 하드웨어: 다른 분야와 비슷한 발전
- 소프트웨어: 많이 미개함
  - 크기만 증가:

    휴대폰 100만줄

    아래아한글 200만줄

    스마트폰 1000만줄

    윈도우 3000만줄
- 소프트웨어: 다른 분야가 이루어 놓은 것을 아직 이루지 못했슴

무엇일까?

다른 분야의 현재 수준

- 제대로 작동할 지를 미리 검증할 수 없는 기계/회로/공정/건축 설계는 없다.
- 제대로 작동할 지를 미리 검증할 수 없는 백신 설계는 없다(?)
- 제대로 작동할 지를 미리 검증할 수 없는 금융상품 설계는 없다(?)

뉴튼방정식, 미적분 방정식, 통계역학, 네비에-스톡스 방정식...

모든 기술의 질문

우리가 만든 것이
우리가 의도한대로 움직인다는 것을
어떻게 미리
확인할 수 있는가?

(사실, 일상에서도 잦은 질문과 답: 입학시험, 입사시험, 궁합, 클럽관리)

소프트웨어 분야에서의 이 질문

작성한 SW의 오류를
자동으로 미리 모두 찾아주거나,
없으면 없다고 확인해주는
기술들은 있는가?

그래서, SW의 오류때문에 발생하는
개인/기업/국가/사회적 비용을
절감시켜주는 기술들은 있는가?

프로그램 검진: 타입오류를 넘어서

SW 오류(bug)

- 소프트웨어에 있는 오류
- 소프트웨어가 생각대로 실행되지 않는 것
- 사람이 소프트웨어를 잘못 만들었기 때문
  - 천재지변이 아님

SW 오류의 예

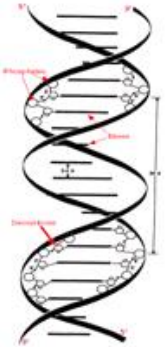
어머니가 전해준 슈퍼마켓 심부름 목록(프로그램)

1. 우유 1리터 2병
2. 우유가 없으면 오렌지쥬스 1리터 3병
3. 우유가 있으면 식빵 500그람 1봉
4. 쌀과 자두

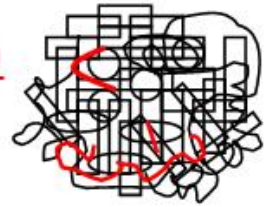
만약에: 우유 1리터 1병만 있으면? 쌀과 자두? 쌀과자두?

SW 오류의 문제

DNA 오류



SW 오류



⇓ ⇓

살다가 병이든다 — SW로 동작하는 제품이 고장난다

사회적 비용 — 사회적 비용

해결책 = 경제적기회 — 해결책 = 경제적기회

SW 오류: 사람이 손으로 손쉽게 잡을 수 없다

- 엄청나게 커지고 복잡해 진 소프트웨어
- 새로운 컴퓨팅 환경: 지구 $=$ 컴퓨터
  - 전지구적 네트웍을 타고
  - 불특정 다수의 코드가 내게로 온다(스마트폰 게임/apps)

자동 기술이 필요

SW 오류 검증 기술: 1세대

문법 검증기: 70년대에 완성된 기술. lexical analyzer & parser

오류 = 소프트웨어의 생김새가 틀린 것

1. 우유 1리터 2병
2. 우유가 없으면 오렌지ㅈㅅ 1리터 3병
3. 있으면 빵식 우유가 500그람 1봉
4. 쌀과 자두

SW 오류 검증 기술: 2세대

타입 검증(type checking): 90년대에 완성.
(30년간 프로그래밍언어 분야의 대표 성과)

오류 = 생긴것은 멀쩡한데, 잘못된 값이 계산에 흘러드는 경우

1. 우유를 담은 얼음을 가스 불위에 올려놓고 데운다
2. 식빵을 유리접시에 담고 방아로 빻는다
3. 2의 접시에 1의 우유를 튀긴다
4. 남자와 소나무를 결혼시킨다

SW 오류 검증 기술: 3세대

프로그램분석검증: 아직 미완성

(static analysis, verification, model checking, ...)

오류 = 생긴것도 멀쩡하고 타입도 맞지만, 바라는 바가 아닌 것

오류 = 필요한 조건을 만족시킬 수 없는 경우

1. 우유를 담은 냄비를 가스 불위에 데운다
2. 식빵을 스텐접시에 담고 방아로 빻는다
3. 남자1명과 여자1명을 결혼 시킨다

- 가스불이 포항제철 용광로가 된다면?
- 방아가 현대중공업의 프레스가 된다면?
- 남녀 나이의 차가 100살이된다면?

데이터의 중력: 새 프로그래밍 중력 (1/7)

많아진 데이터, 싸진 컴퓨터

기계 학습$_{machine\ learning}$
인덕$_{induction}$하는
프로그램짜기

확률 추론 프로그래밍$_{probabilistic\ pgm'}$
앱덕$_{abduction}$하는
프로그램짜기

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{machine\ learning}$ (*인덕*$_{induction}$)

- *확률 추론*$_{probabilistic\ pgm'ng}$ (*앱덕*$_{abduction}$)

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{\textit{machine learning}}$ (*인덕*$_{\textit{induction}}$)
  - 입력: 관찰한 데이터 $\{(a_0, b_0), \cdots, (a_k, b_k)\}$
  - 출력: 짐작하는 함수

- *확률 추론*$_{\textit{probabilistic pgm'ng}}$ (*앱덕*$_{\textit{abduction}}$)

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{\textit{machine learning}}$ (*인덕*$_{\textit{induction}}$)
  - 입력: 관찰한 데이터 $\{(a_0, b_0), \cdots, (a_k, b_k)\}$
  - 출력: 짐작하는 함수
  - 예: $\{(-1, 0), (2, 3)\}$에서?
- *확률 추론*$_{\textit{probabilistic pgm'ng}}$ (*앱덕*$_{\textit{abduction}}$)

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{\textit{machine learning}}$ (*인덕*$_{\textit{induction}}$)
  - 입력: 관찰한 데이터 $\{(a_0, b_0), \cdots, (a_k, b_k)\}$
  - 출력: 짐작하는 함수
  - 예: $\{(-1, 0), (2, 3)\}$에서?
- *확률 추론*$_{\textit{probabilistic pgm'ng}}$ (*앱덕*$_{\textit{abduction}}$)
  - 입력: 인과관계 $A \implies B$, 관찰한 데이터 B
  - 출력: 짐작하는 원인 A

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{\textit{machine learning}}$ (*인덕*$_{\textit{induction}}$)
  - 입력: 관찰한 데이터 $\{(a_0, b_0), \cdots, (a_k, b_k)\}$
  - 출력: 짐작하는 함수
  - 예: $\{(-1, 0), (2, 3)\}$에서?
- *확률 추론*$_{\textit{probabilistic pgm'ng}}$ (*앱덕*$_{\textit{abduction}}$)
  - 입력: 인과관계 $A \implies B$, 관찰한 데이터 B
  - 출력: 짐작하는 원인 A
  - 예: 시험점수에서? 선호영화에서?

데이터의 중력: 새 프로그래밍 중력 (2/7)

- *기계 학습*$_{\textit{machine learning}}$ (*인덕*$_{\textit{induction}}$)
  - 입력: 관찰한 데이터 $\{(a_0, b_0), \cdots, (a_k, b_k)\}$
  - 출력: 짐작하는 함수
  - 예: $\{(-1, 0), (2, 3)\}$에서?
- *확률 추론*$_{\textit{probabilistic pgm'ng}}$ (*앱덕*$_{\textit{abduction}}$)
  - 입력: 인과관계 $A \implies B$, 관찰한 데이터 B
  - 출력: 짐작하는 원인 A
  - 예: 시험점수에서? 선호영화에서?
- 얼추짐작을 자동으로/과학적으로
- 주의: 엉망인 데이터, 빠뜨린 인과관계, 관찰못한 데이터

데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

- 언어로 논리적인 문제풀이법을 표현

데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

- 언어로 논리적인 문제풀이법을 표현
- 계산방법을 꼼꼼히 상위의 언어로 손수작성

데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

- 언어로 논리적인 문제풀이법을 표현
- 계산방법을 꼼꼼히 상위의 언어로 손수작성
  - 현실적인 비용의 계산과정이어야
  - 비용 분석, 비용한계 검진 (과학)

데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

- 언어로 논리적인 문제풀이법을 표현
- 계산방법을 꼼꼼히 상위의 언어로 손수작성
  - 현실적인 비용의 계산과정이어야
  - 비용 분석, 비용한계 검진 (과학)
- 쓴 글을 "튜링기계"(기계어)로 번역해서 컴퓨터에 싣는다

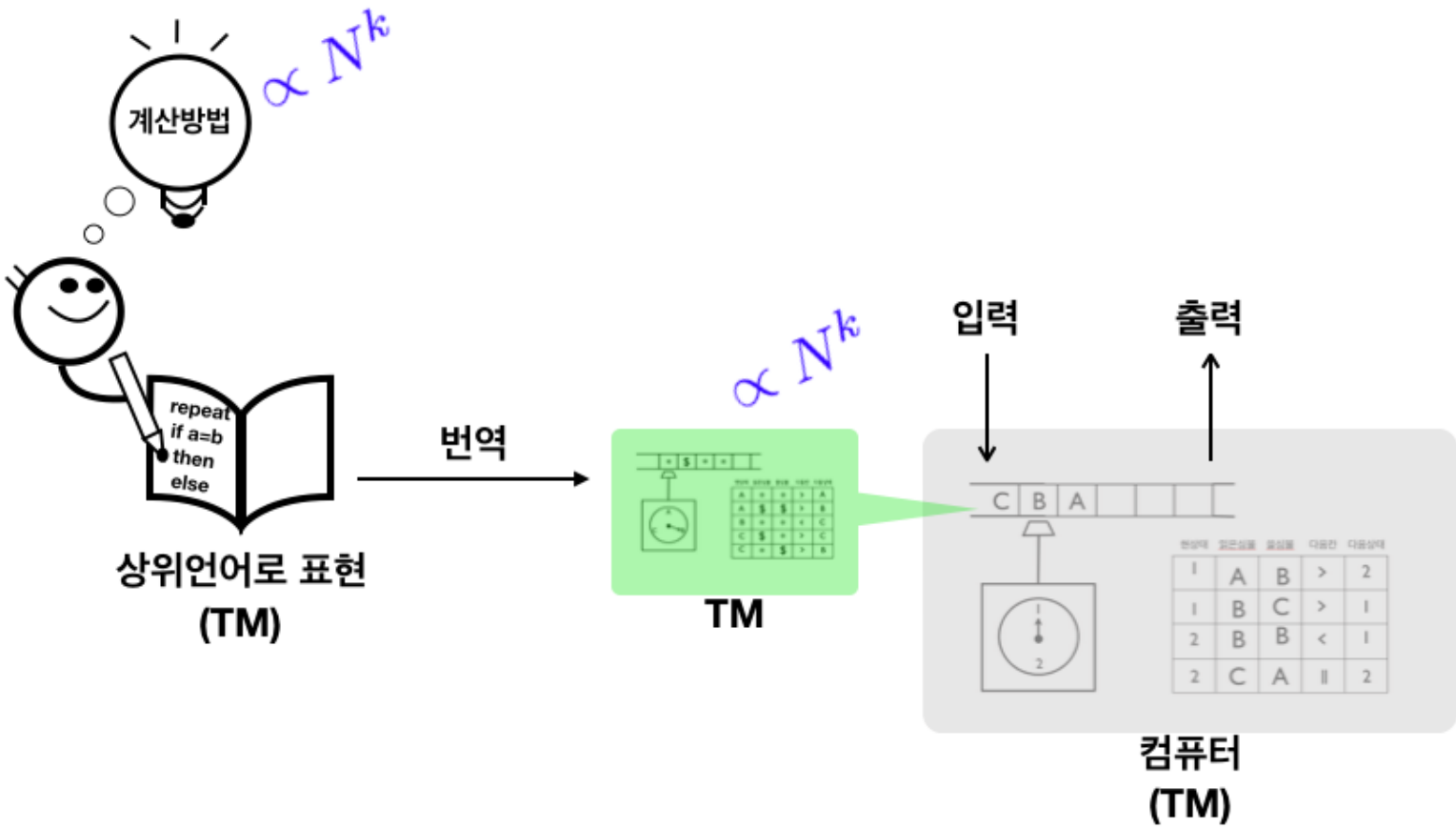
데이터의 중력: 지금까지 프로그래밍 기둥 (3/7)

기둥 A: 기계학습 이전

- 언어로 논리적인 문제풀이법을 표현
- 계산방법을 꼼꼼히 상위의 언어로 손수작성
  - 현실적인 비용의 계산과정이어야
  - 비용 분석, 비용한계 검진 (과학)
- 쓴 글을 "튜링기계"(기계어)로 번역해서 컴퓨터에 싣는다

프로그래밍$_{programming}$, *알고리즘*$_{algorithm}$

데이터의 중력: 지금까지 프로그래밍 기둥 A (4/7)



데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - “보고 따라하길”

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - "보고 따라하길"
- 기계학습: "서당개 5만년" 훈련

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - "보고 따라하길"
- 기계학습: "서당개 5만년" 훈련
  - 수많은 정답예시 주입해서 SW 자동생성
  - 기계학습 과정과 결과 SW 모두
  - 현실적인 비용의 계산과정이어야

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - "보고 따라하길"
- 기계학습: "서당개 5만년" 훈련
  - 수많은 정답예시 주입해서 SW 자동생성
  - 기계학습 과정과 결과 SW 모두
  - 현실적인 비용의 계산과정이어야
  - 학습가능 범주분석, 학습비용 한계 검진 (과학)

데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - "보고 따라하길"
- 기계학습: "서당개 5만년" 훈련
  - 수많은 정답예시 주입해서 SW 자동생성
  - 기계학습 과정과 결과 SW 모두
  - 현실적인 비용의 계산과정이어야
  - 학습가능 범주분석, 학습비용 한계 검진 (과학)
- 결과 SW -*뉴럴넷*$_{neural\ net}$- 을 컴퓨터에 싣는다

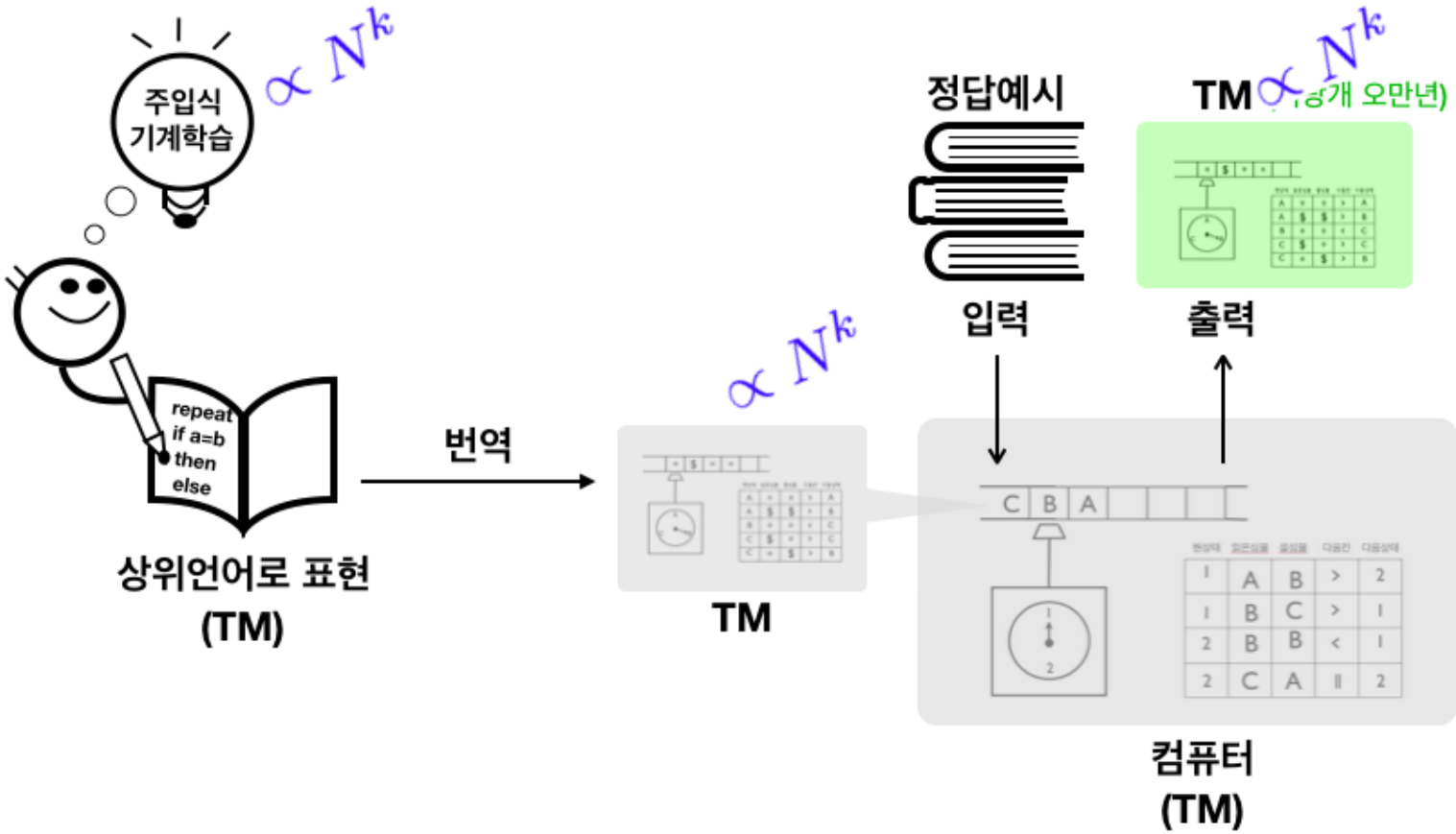
데이터의 중력: 새 프로그래밍 기둥 (5/7)

기둥 Z: 기계학습의 출현

- 어떻하지? 언어로 논리적으로 표현못하겠다
- 정답예시를 주입식으로
  - "보고 따라하길"
- 기계학습: "서당개 5만년" 훈련
  - 수많은 정답예시 주입해서 SW 자동생성
  - 기계학습 과정과 결과 SW 모두
  - 현실적인 비용의 계산과정이어야
  - 학습가능 범주분석, 학습비용 한계 검진 (과학)
- 결과 SW -*뉴럴넷*$_{neural\ net}$- 을 컴퓨터에 싣는다

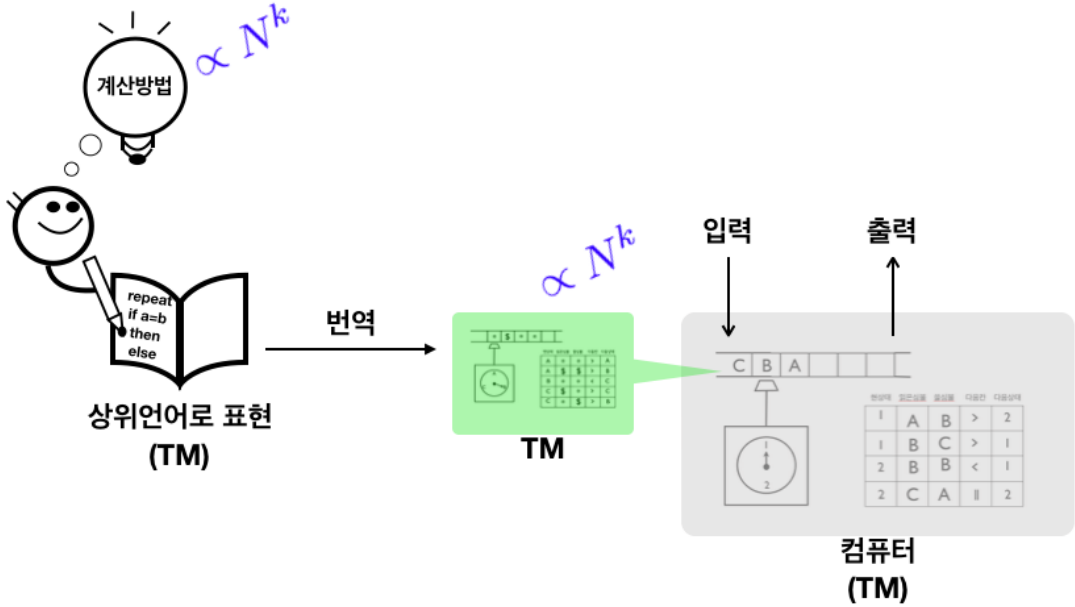
기계학습$_{machine\ learning}$, *에코리즘*$_{ecorithm}$

데이터의 중력: 새 프로그래밍 기둥 Z (6/7)

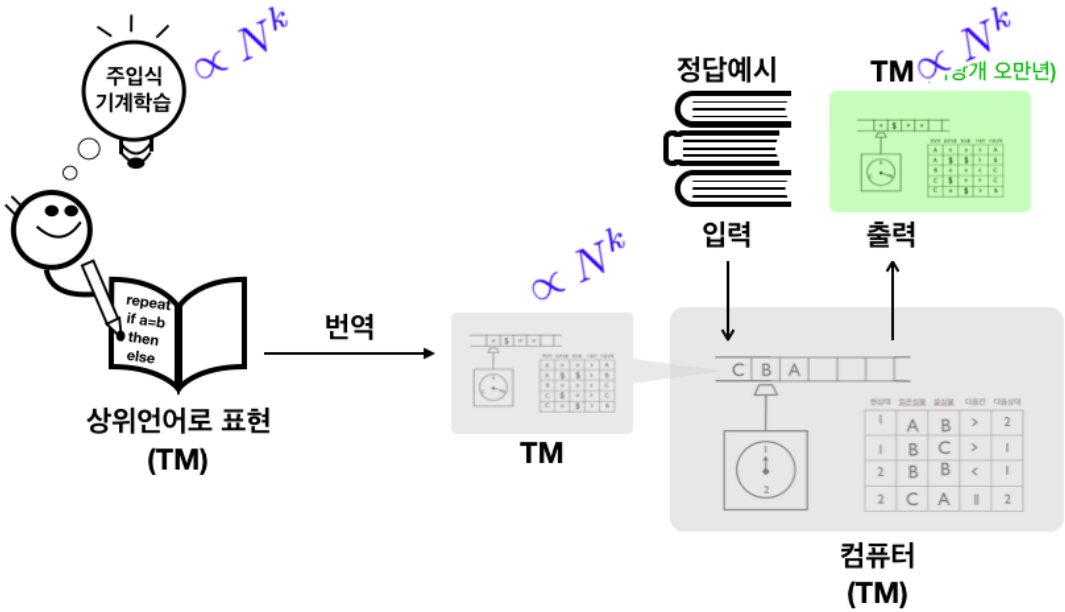


데이터의 중력: 소프트웨어 만들기 두 기둥 (7/7)

기둥 A



기둥 Z



다음