2. 알고리즘 개요

한국외국어대학교
고 석 훈

목차

2.1 알고리즘(Algorithm)

- 알고리즘?
  - 문제를 해결하기 위한 명령어의 단계적 절차
- 알고리즘은 요리법과 유사하다.
  - 요리법의 단계적인 절차를 따라 하면 요리가 만들어지듯이, 알고리즘의 단계적인 절차를 따라 하면 문제의 답이 구해진다.
- 문제를 해결하는 방법은 다양하다.
  - 주어진 문제에 대해 여러 종류의 알고리즘이 있을 수 있으나, 보다 효율적인 알고리즘을 고안하는 것이 중요하다.
  - 알고리즘 효율성의 기준: 실행 시간, 메모리 사용량

- 알고리즘 예) 라면 끓이는 방법

1. 물을 끓인다.
2. 물이 끓으면 면과 스프를 넣는다.
3. 면이 익도록 3분 더 끓인다.

- 알고리즘 예) 된장찌개 끓이는 방법

1. 멸치육수를 우린다.
2. 감자, 양파, 애호박을 먼저 끓인다.
3. 양파가 익으면 된장을 더해 끓인다.
4. 감자가 익으면 두부와 대파를 넣어 끓인다.
5. 갖은 양념을 넣고 소금으로 간을 조절한다.

알고리즘의 조건

- 알고리즘의 5가지 조건
  - **입력(input)** :
  - **출력(output)** :
  - **명백성(definiteness)** :
  - **유효성(effectiveness)** :
  - **유한성(finiteness)** :

- 알고리즘과 프로그램의 차이
  - 유한하지 않은 프로그램, 데몬(deamon)

알고리즘의 유래

- 알고리즘 단어의 유래:
  9세기경 페르시아 수학자
  알콰리즈미(al-Khwārizmī)
  - 아랍식 기수법(記數法)을 뜻하는 알고리즘(algorism)은 이 이름에서 전용된 것이다.
    대수학 저서인 "복원(復元)과 대비의 계산"에는
    1차방정식과 2차방정식의 해석적 해법과
    2차방정식의 기하학적 해법도 보여 주고 있다.
    대수학을 뜻하는 영어의 algebra는 아랍어로
    복원(復元)을 뜻하는 al-jabr에서 유래한다. [두산백과]

인류 최초의 알고리즘

- Euclid's Algorithm
  - BC 300년 경에 그리스 수학자 Euclid(기하학의 아버지)가 제안한 최대공약수(GCD: Greatest Common Divisor) 계산 알고리즘

```
gcd(a, b) {
    입력: 정수 a, b
    출력: 최대공약수(a, b)

    if (a < b) swap(a, b)      // a, b 교환
    if (b = 0) return a
    return gcd(b, a − b)
}
```

2.2 알고리즘의 표현 방법

- 자연어로 표기된 알고리즘
- 흐름도로 표기된 알고리즘
- 프로그래밍 언어로 표현된 알고리즘
- Pseudo 코드로 표현된 알고리즘

자연어로 알고리즘 표현

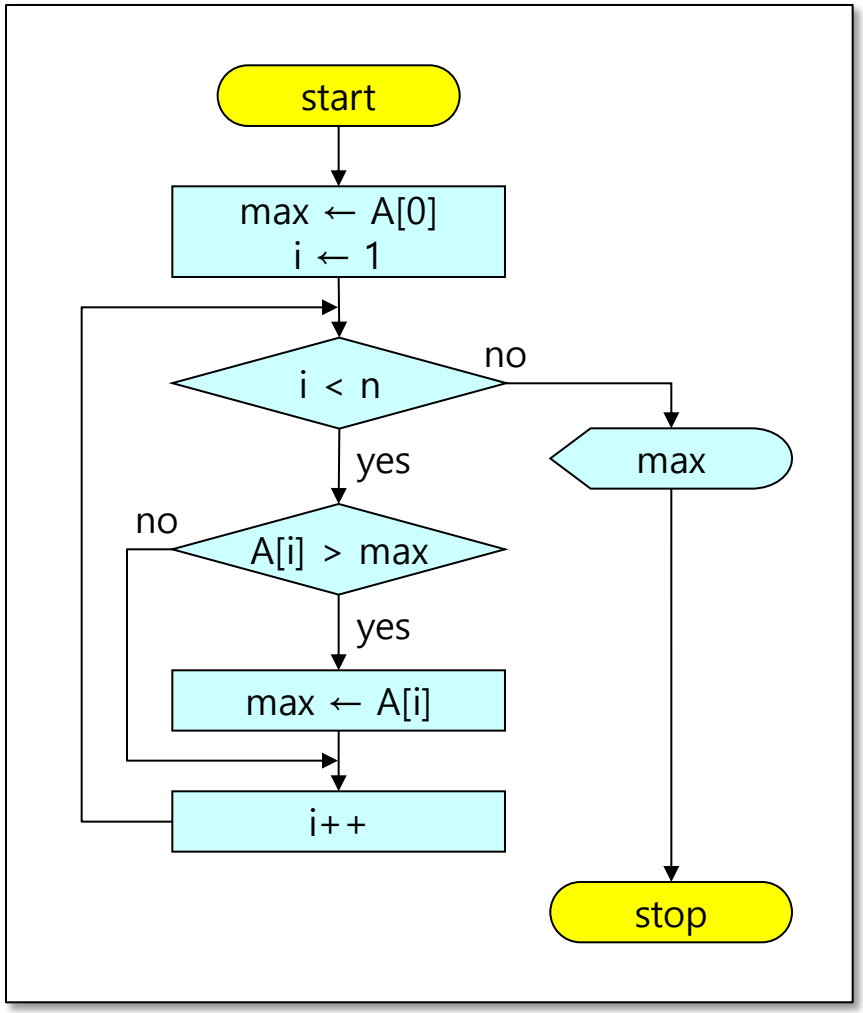
- 영어나 한국어로 표기된 알고리즘
  - 인간이 읽기가 쉽다. 누구나 읽을 수 있다.
  - 내용이 많아질 경우 작성이 번거롭고, 수정하기 어렵다.
  - 의미 전달이 모호해질 우려가 있다.

- 예제) 배열에서 최대값 찾기

ArrayMax(배열 A, 배열의 크기 n)
1. 배열 A의 첫 번째 요소를 변수 max에 복사
2. 배열 A의 다음 요소를 max와 비교하여 더 크면 max로 복사
3. 배열 A의 모든 요소를 비교했으면 max를 반환, 아니면 2번 반복

흐름도로 알고리즘 표현

- 흐름도 그리기
  - 직관적이고 이해하기 쉽다.
  - 규모가 커지면 복잡해 진다.
  - 작성, 수정이 매우 어렵다.



프로그래밍 언어로 알고리즘 표현

- C언어로 알고리즘 표현
  - 알고리즘을 가장 정확하고, 상세하게 기술할 수 있다.
  - 컴퓨터로 실행할 수 있다.
  - 반면에, 구현에 관련된 많은 구체적인 사항들이 알고리즘의 핵심 내용의 이해를 방해할 수 있다.

```
#define MAX_ELEMENTS 100
int A[MAX_ELEMENTS];

int find_array_max(int n)
{
   int i, max;

   max = A[0];
   for (i = 1; i < n; i++) {
      if (A[i] > max) {
         max = A[i];
      }
   }
   return max;
}
```

Pseudo 코드로 알고리즘 표현

- **Pseudo 코드**
  - 알고리즘 기술에 가장 많이 사용하는 고수준 기술 방법으로 자연어보다는 구조적이며, 프로그래밍 언어보다는 덜 구체적이다.
  - 프로그램을 구현할 때에 필요한 복잡한 문제들을 감출 수 있다. 즉, 알고리즘의 핵심적인 내용에만 집중할 수 있다.

$$
\begin{aligned}
&ArrayMax(A, n)\ \{\\
&\quad input\colon\ array\ A, array\ size\ n\\
&\quad output\colon\ \max\ element\ of\ A\\
&\quad \max = A[0]\\
&\quad for\ i \leftarrow 1\ to\ n-1\ do\\
&\quad\quad if\ \max < A[i]\ then\\
&\quad\quad\quad \max \leftarrow A[i]\\
&\quad return\ \max\\
&\}
\end{aligned}
$$

Pseudo 코드 작성 방법

- Pseudo 코드 작성 방법
  - C, Pascal 등 프로그래밍 언어의 문법을 바탕으로
    알고리즘 기술에 불필요한 부분은 제거하고 간결하게 작성

- Pseudo 코드의 기본 연산
  - 배정문: $A \leftarrow B$, $A = B$, $A := B$
  - 비교: $A < B$, $A <= B$, $A == B$, $A != B$, $A > B$, $A >= B$
  - 사칙연산: $A + B$, $A - B$, $A * B$, A / B, $A \% B$
  - 비교, 논리 연산: and, or, not
  - 제어 명령: for, while, repeat–until(do-while), if–then–else
  - 함수(function) 호출: function_name(parameter_list)

2.3 알고리즘의 성능 분석

- **공간 복잡도 (Space Complexity)**
  - 알고리즘을 프로그램으로 실행하여 완료하기까지 필요한 총 저장 공간의 양
  - 공간 복잡도 = 고정 공간 + 가변 공간

- **시간 복잡도 (Time Complexity)**
  - 알고리즘을 프로그램으로 실행하여 완료하기까지 총 소요시간
  - 시간 복잡도 = 컴파일 시간 + 실행 시간
    - 컴파일 시간: 프로그램마다 거의 고정적인 시간 소요
    - 실행 시간: 컴퓨터의 성능에 따라 달라질 수 있으므로 실제 실행시간 보다는 명령문의 실행 빈도수에 따라 계산

시간 복잡도 측정: C언어

- C로 구현하여 측정
  - 알고리즘을 C로 구현하여 컴퓨터에서 실행하여 수행시간을 측정
  - 즉시 컴퓨터로 실행 가능
  - C 코드의 상세한 부분을 모두 구현해야 한다.
- 수행시간 측정 방법
  - clock_t clock(void)로 프로세스 타임을 읽는다.
  - CLOCKS_PER_SEC 단위

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main()
{
    clock_t start, finish;
    double  duration;

    start = clock();

    // 시간 측정 대상 알고리즘

    finish = clock();
    duration = (double)(finish
            - start)/CLOCKS_PER_SEC;
   printf("%f초입니다.\n", duration);
}
```

시간 복잡도 측정: Pseudo 코드

- Pseudo 코드로 측정
  - 알고리즘을 pseudo 코드로 작성하여 입력 개수 n에 대한 연산의 수행 회수를 측정(추상적으로 실행)하여 시간 복잡도 함수 $T(n)$를 찾아낸다.
  - 실제 프로그램 코딩환경을 반영하지 못하여 실제 성능과 괴리가 생길 수 있다.
  - 입력 데이터의 조건에 따라 연산 경로가 달라질 수 있는데 시간 복잡도는 최악의 경우를 기준으로 한다.

- 피보나치 수열의 시간 복잡도
  - 앞의 두수의 합이 다음 수가 되는 수열
    1 1 2 3 5 8 13 . . .

```
     fibonacci(n)
01:  if (n < 0) then
02:       stop;
03:  if (n <= 1) then
04:       return n;
05:  f1 = 0;
06:  f2 = 1;
07:  for (i = 2; i <= n; i = i + 1) {
08:       fn = f1 + f2;
09:       f1 = f2;
10:       f2 = fn;
11:  }
12:  return fn;
```

행	$n < 0$	$0 \le n \le 1$	$n > 1$
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

알고리즘 분석 기준

- 최선의 경우 분석 (best case analysis)
  - 분석 결과가 큰 의미가 없다.
- 평균의 경우 분석 (average case analysis)
  - 수학적으로 평균을 추정하기가 매우 어렵다.
- 최악의 경우 분석 (worst case analysis)
  - 계산하기 쉽고, 분석 결과가 중요한 의미를 갖는다.

2.4 복잡도의 점근적 표기

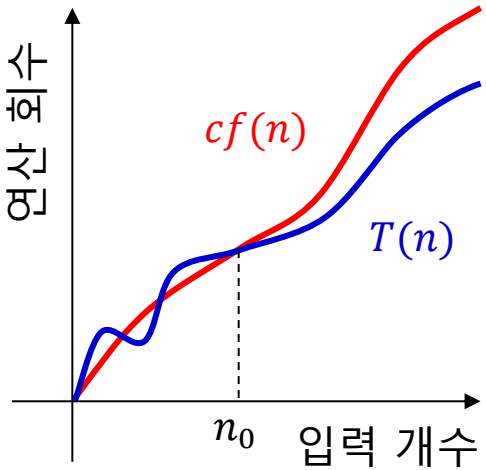
- **점근적 표기(Asymptotic Notation)**
  - 시간복잡도를 입력 크기에 대한 함수 $T(n)$으로 표기하는데, $T(n)$은 주로 여러 개의 항을 가지는 다항식이 된다.
  - 시간복잡도 함수 $T(n)$에 대해 입력 크기 n이 무한대로 커질 때의 복잡도를 간단히 표현하는 것을 점근적 표기라 한다.

- 점근적 표기 방법의 종류
  - 점근적 상한을 의미하는 O(Big-Oh) 표기
  - 점근적 하한을 의미하는 Ω(Big-Omega) 표기
  - 상한과 하한이 동일한 경우 Θ(Theta) 표기

O(Big-Oh) 표기법

- O(Big-Oh)의 정의

> 모든 $n \geq n_0$에서 $T(n) \leq cf(n)$을 만족하는
> 상수 c와 n_0가 존재하면 $T(n)$은 $O(f(n))$이다.



  - O(Big-Oh)는 함수의 상한을 표시한다.
  - O(Big-Oh)를 구하는 방법
    - 실행시간 함수의 값에 가장 큰 영향을 주는 n에 대한 항에 대해
    - 계수는 생략하고 O(Big-Oh)의 오른쪽 괄호 안에 표시
  - 예) 피보나치 수열의 시간 복잡도
    - $T(n) = 4n + 2$
    - $n \geq 2$ 에서 $4n + 2 \leq 5n$ $(c = 5, n_0 = 2)$
    - 따라서, 피보나치 수열의 시간 복잡도는 $O(n)$

O(Big-Oh)의 종류 비교

$O(1)$: 상수형

$O(\log n)$: 로그형(logarithmic)

$O(n)$: 선형(linear)

$O(n \log n)$: 로그선형(Log-linear)

$O(n^2)$: 2차형(quadratic)

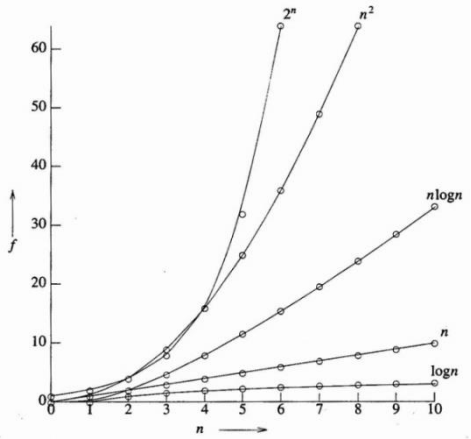
$O(n^3)$: 3차형(Cubic)

$O(n^k)$: k차형(polynomial) ***efficient***

$O(2^n)$: 지수형(exponential) ***inefficient***

$O(n!)$: 팩토리얼형

시간복잡도	n				
	1	2	4	8	16
1	1	1	1	1	1
$\log n$	0	1	2	3	4
n	1	2	4	8	16
$n \log n$	0	2	8	24	64
n^2	1	4	16	64	256
n^3	1	8	64	512	4096
2^n	2	4	16	256	65536
$n!$	1	2	24	40326	20922789888000



O(Big-Oh) 예제 1

```
      algorithmA(n)
01:   a = 0;
02:   for (i = 1; i <= n; i = i + 1) {
03:      if (i%2 == 0)
04:         a = a + i;
05:   }
06:   return a;
```

행	연산 회수
1	
2	
3	
4	
5	
6	

$T(n)$

O(Big-Oh) 예제 2

```
        algorithmB(n)
01:     x = 0;
02:     y = 0;
03:     for (i = 1; i <= n; i++) {
04:         for (j = 1; j <= i; j++) {
05:             x = x + j;
06:             y = y + x;
07:         }
08:     }
09:     return x;
```

행	연산 회수
1	
2	
3	
4	
5	
6	
7	
8	
9	

$T(n)$

O(Big-Oh) 예제 3

$$T(n) = 2n - 16 = \square$$

$$T(n) = 55n^2 + 10n + 5 = \square$$

$$T(n) = 6 = \square$$

$$T(n) = 3 \log n + 4 \log \log n = \square$$

$$T(n) = 2n^3 + 4\,n^2 \log n = \square$$

$$T(n) = 2^n + n^{17} - 7n^2 = \square$$

왜 효율적인 알고리즘이 필요한가?

- 10억 개의 숫자를 정렬하는데 PC에서
  $O(n^2)$ 알고리즘은 300여 년이 걸리는 반면
  $O(n \log n)$ 알고리즘은 5분 만에 정렬한다.

$O(n^2)$	1,000	1백만	10억
PC	< 1초	2시간	300년
슈퍼컴	< 1초	1초	1주일

$O(n \log n)$	1,000	1백만	10억
PC	< 1초	<1초	5분
슈퍼컴	< 1초	<1초	<1초

- 효율적인 알고리즘은 슈퍼컴퓨터보다 더 큰 가치가 있다.
  - 값 비싼 H/W의 기술 개발보다
    효율적인 알고리즘 개발이 훨씬 더 경제적이다.

2.5 알고리즘의 분류

- 문제의 해결 방식에 따른 분류
  - 분할 정복(Divide-and-Conquer) 알고리즘
  - 그리디(Greedy) 알고리즘
  - 동적 계획(Dynamic Programming) 알고리즘
  - 근사(Approximation) 알고리즘
  - 백트래킹(Backtracking) 기법
  - 분기 한정(Branch-and-Bound) 기법

- 문제에 기반한 분류
  - 정렬 알고리즘
  - 그래프 알고리즘
  - 기하 알고리즘
- 특정 환경에 따른 분류
  - 병렬(Parallel) 알고리즘
  - 분산(Distributed) 알고리즘
- 기타 알고리즘들
  - 확률 개념이 사용되는 랜덤(Random) 알고리즘
  - 유전자(Genetic) 알고리즘

Q&A