자료구조

L03 Algorithm Analysis

2022년 1학기

국민대학교 소프트웨어학부

Summary

- **알고리즘 개요**
- 알고리즘 성능 분석
- 복잡도 예제
- 문제의 복잡도

알고리즘 (Algorithm)?

- **알고리즘**
  - 어떤 문제를 해결하기 위한 절차나 방법을 공식화한 형태로 표현한 것
- **프로그램**
  - 알고리즘을 컴퓨터가 이해하고 실행할 수 있는 특정 프로그래밍 언어로 표현한 것
- **좋은 알고리즘의 조건**
  - 입력(input): 정의된 입력을 받아들일 수 있어야 한다.
  - 출력(output): 답으로 출력을 내보낼 수 있어야 한다.
  - 알고리즘은 명확하게 작성되어야 하고, 실현 가능해야 한다.

알고리즘 (Algorithm)?

결과 (출력)

[요리 재료] **자료 (입력)**

스펀지케이크(20×20cm) 1개, 크림치즈 200g, 달걀 푼 물 2개 분량, 설탕 3큰술, 레몬즙·바닐라에센스 1큰술씩, 딸기시럽(딸기 500g, 설탕 1½ 컵, 레몬즙 1작은술), 딸기 1개, 플레인 요구르트 2큰술

[요리법] **절차 (알고리즘)**

① 케이크 틀의 가장자리에 필름을 돌린 다음 스펀지케이크를 놓는다.

② 볼에 크림치즈를 넣고 거품기로 젓다가 달걀 푼 물과 설탕 3큰술을 세번에 나누어 넣으면서 크림 상태로 만든다.

③ ②에 레몬즙과 바닐라에센스를 넣고 살짝 저은 다음 ①에 붓는다. 이것을 180℃의 오븐에 넣고 20분 정도 굽는다.

④ 냄비에 슬라이스한 딸기와 설탕 1½ 컵을 넣고 끓이다가 약한 불에서 눌어붙지 않도록 저으면서 거품을 걷어낸다. 되직해지면 레몬즙을 넣고 차게 식힌다.

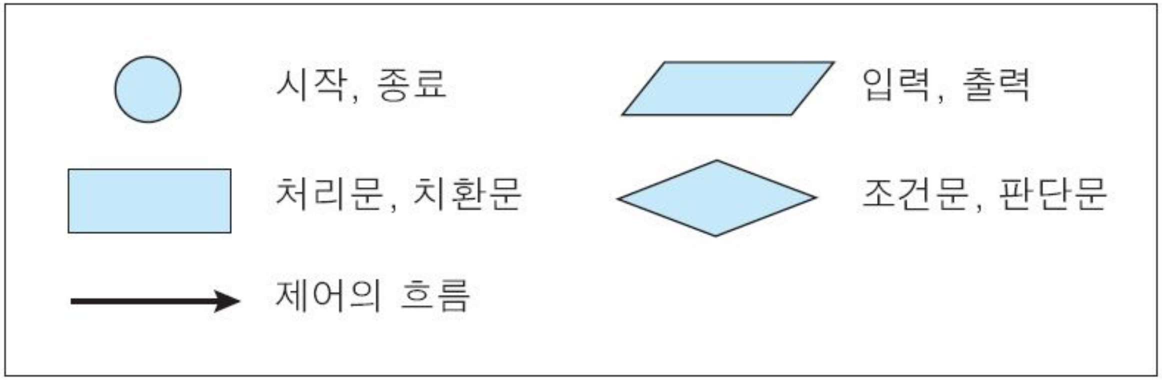
⑤ 접시에 치즈케이크를 한 조각 담고 ④의 시럽을 뿌린 다음 플레인 요구르트와 딸기를 얹어낸다 .

알고리즘 표현 방법

- 자연어를 이용한 서술적 표현 방법
- **순서도**(Flow chart)를 이용한 도식화 표현 방법
- 프로그래밍 언어를 이용한 구체화 방법
- **의사코드**(Pseudo-code)를 이용한 추상화 방법

알고리즘 표현 방법 - 순서도

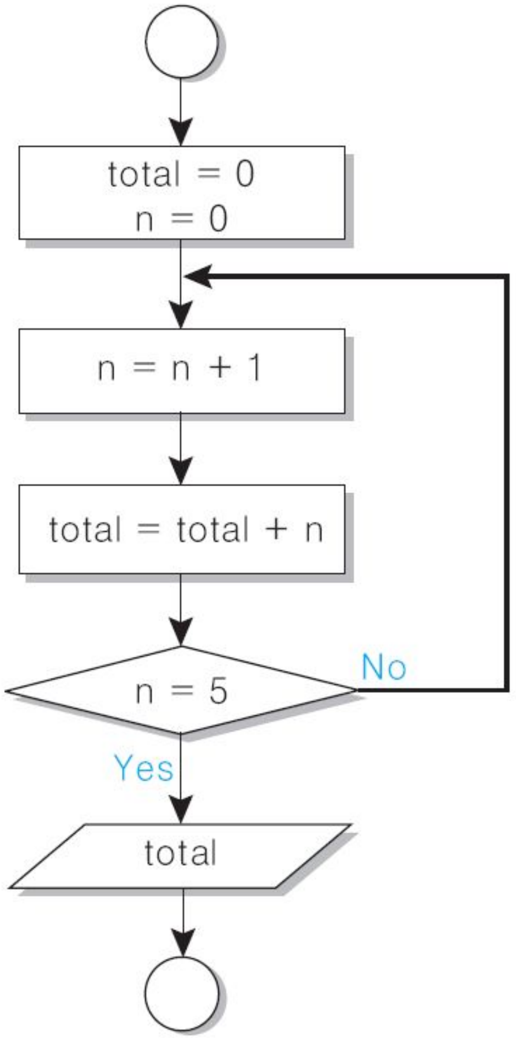
- 순서도를 이용한 알고리즘의 표현
  - 순서도에서 사용하는 기호



- 장점 : 알고리즘의 흐름 파악이 용이함
- 단점 : 복잡한 알고리즘의 표현이 어려움

알고리즘 표현 방법 - 순서도

- 순서도 예)



알고리즘 표현 방법 - 의사코드 (Pseudo-code)

- 일반적인 언어로 프로그램 언어를 흉내 내어 써 놓은 알고리즘 코드
- 특정 프로그래밍 언어가 아니므로 **직접 실행은 불가**능
- 일반적인 프로그래밍 언어의 형태이므로 원하는 특정 **프로그래밍 언어로의 변환 용이**
- 표현이 자유롭기 때문에 **사람(저자)마다 문법이 다를 수 있음**

알고리즘 표현 방법 - 의사코드 (Pseudo-code)

- 의사코드 예)

```
 fibonacci(n)
if (n<0) then
  stop
if (n≤1) then
  return n
f1 ← 0
f2 ← 1
for (i in {2, 3, …, n}) do
  fn ← f1 + f2
  f1 ← f2
  f2 ← fn
return fn
```

Summary

- 알고리즘 개요
- **알고리즘 성능 분석**
- 복잡도 예제
- 문제의 복잡도

알고리즘의 효율성

Q. 같은 문제를 해결하는 여러 알고리즘이 있을 경우, 어떤 알고리즘을 선택해야 할까?

- 컴퓨터 프로그램 디자인의 두 가지 핵심 목표:
  - **목표 1 (용이성).** 알고리즘을 이해하기 쉽고, 코딩하기 쉽고, 디버깅하기 쉽도록 디자인한다
  - **목표 2 (효율성).** 자원을 효율적으로 사용하도록 알고리즘을 디자인한다

알고리즘의 효율성

- **목표 1 (용이성)**은 소프트웨어 공학의 관점에서 중요
- **목표 2 (효율성)**은 자료구조 및 알고리즘 분석의 관점에서 중요

알고리즘의 효율성은 어떻게 측정해야할까?

알고리즘 성능 분석

Q. 알고리즘의 효율성(성능)은 어떻게 측정할 수 있을까?

- 효율성 측정 방법
  - 실험적 분석 - 실제 돌려보고, 측정한다
  - 점근적 분석 (Asymptotic Algorithm Analysis)
- 중요한 자원: 시간, 공간, 개발자의 노력, 사용하기 쉬움
- 대부분의 알고리즘은 입력(input)의 크기에 따라 수행시간이 결정됨
- 수행시간은 입력 크기 n에 대한 함수 T(n)로 표현됨
  - T(n) = <u>기본연산</u>의 실행 빈도수
  - <u>기본연산</u>: 피연산자나 n에 의해 수행시간이 변하지 않는 연산
    - 명령행 (for/while 제외), 사칙연산, 할당, 비교 등…

알고리즘 성능 분석

- 피보나치 수열 알고리즘의 빈도수 구하기

fibonacci(n)	n < 0	n = 0	n = 1	n > 1
01 **if** (n<0) **then**				
02 **stop**				
03 **if** (n≤1) **then**				
04 **return** n				
05 f1 ← 0				
06 f2 ← 1				
07 **for** (i in {2, 3, …, n}) **do**				
08 fn ← f1 + f2				
09 f1 ← f2				
10 f2 ← fn				
11 **return** fn				
T(n) =				

알고리즘 성능 분석

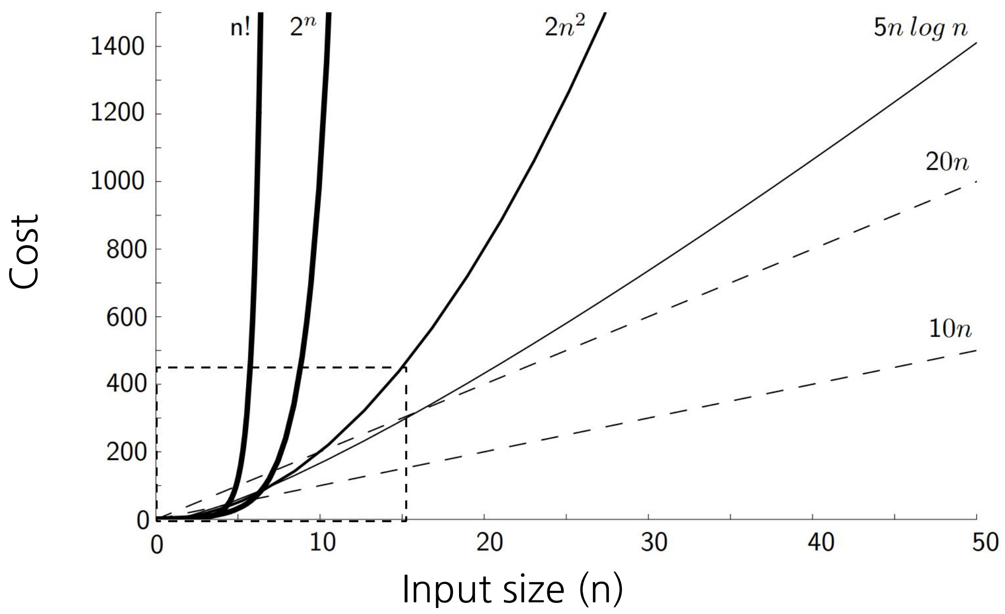
- 피보나치 수열 알고리즘의 빈도수 구하기

fibonacci(n)	n < 0	n = 0	n = 1	n > 1
01 **if** (n<0) **then**	1	1	1	1
02 **stop**	1	0	0	0
03 **if** (n≤1) **then**	0	1	1	1
04 **return** n	0	1	1	0
05 f1 ← 0	0	0	0	1
06 f2 ← 1	0	0	0	1
07 **for** (i in {2, 3, …, n}) **do**	0	0	0	n-1
08 fn ← f1 + f2	0	0	0	n-1
09 f1 ← f2	0	0	0	n-1
10 f2 ← fn	0	0	0	n-1
11 **return** fn	0	0	0	1
T(n) =	**2**	**3**	**3**	**4n+1**

알고리즘 성능 분석

- 각 실행 시간 함수에서 n값의 변화에 따른 실행 빈도수 비교

logn	<	n	<	nlogn	<	n^2	<	n^3	<	2^n
0		1		0		1		1		2
1		2		2		4		8		4
2		4		8		16		64		16
3		8		24		64		512		256
4		16		64		256		4096		65536
5		32		160		1024		32768		4294967296



점근적 분석법 (Asymptotic Analysis)

시간복잡도 표기법 빅-오(Big-Oh)

(informal) 아주 큰 n 값에 대해서 $T(n) \le c \cdot f(n)$ 을 만족하면, T(n)은 O(f(n))에 속한다.

- 빅-오 표기법 순서
  - 실행 빈도수를 구하여 실행시간 함수 찾기
  - 실행시간 함수의 값에 가장 큰 영향을 주는 n에 대한 항을 선택하여
  - 계수는 생략하고 O의 오른쪽 괄호 안에 표시

- 예) 피보나치 수열의 시간 복잡도
  - T(n): 4n+1
  - n에 대한 항을 선택: 4n
  - 계수 4는 생략하고 O의 오른쪽 괄호 안에 표시: O(n)

점근적 분석법 (Asymptotic Analysis)

- 빅-오 표기법 예제
  - $T(n) = 5n$ 는 $O(n)$에 속한다.
  - $T(n) = 5n + 6$ 는 $O(n)$에 속한다.
  - $T(n) = 4n^2 + 3n + 7$ 는 $O(n^2)$에 속한다.
  - $T(n) = 4n^2 + 3n + 7$ 는 $O(n)$에 속하지 **않는다**.
    - Why?

점근적 분석법 (Asymptotic Analysis)

빅-오 표기법의 정의

For $T(n)$ a non-negative valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n > n_0$

- 의미: 충분히 큰 모든 입력 데이터에 대해 (즉, $n > n_0$), 해당 알고리즘은 항상 $c \cdot f(n)$보다 적은 연산을 수행한다.
- 사용법:

The algorithm is in $O(n^2)$ in [best, average, worst] case.

The algorithm [requires/takes] $O(n^2)$ computations in [best, average, worst] case.

The [time/space] complexity of the algorithm is $O(n^2)$ in [best, average, worst] case.

점근적 분석법 (Asymptotic Analysis)

- 빅-오 표기법은 **상한(upper-bound)**을 의미

- 예) $T(n) = 3n^2$ 일 경우, $T(n)$은 $O(n^2)$에도 속하고, $O(n^3)$에도 속함.
  - 가장 **tight**한 bound를 찾는 것이 중요!

빅-오 예제

예 1) 양의 실수 c1과 c2에 대해서 $T(n) = c_1n^2 + c_2n$ 일 경우

- $c_1n^2 + c_2n \leq c_1n^2 + c_2n^2 = (c_1 + c_2)n^2$ for all $n > 1$
- Then, $T(n) \leq cn^2$ whenever $n > n_0$ for $c = c_1 + c_2$ and $n_0 = 1$
- Therefore, $T(n)$ is in $O(n^2)$

예 2) $T(n) = d$ 일 경우 $T(n)$은 $O(1)$에 속한다.

- c 값은? n_0 값은?

빅-오 예제

예 3) 크기가 n인 배열에서 x값 찾기 (average case)

Best, Worst, Average Cases

- 크기가 n으로 같은 모든 입력에 대해 알고리즘의 수행시간이 항상 같진 않다.
- 예) 크기가 n인 배열에서 x값 찾기: 배열의 첫번째 요소부터 시작하여 K값이 발견될 때 까지 요소들을 순차적으로 확인한다.
  - Best case cost?
  - Worst case cost?
  - Average case cost?

Best, Worst, Average Cases

- Best, Word, Average case 중 어떤 분석을 해야 할까?
  - Best case?
  - Worst case?
  - Average case?

Best, Worst, Average Cases

- Average case 분석이 가장 공평해 보이지만…, Best case 분석과 Worst case 분석도 필요하다.
- 언제 Best case 분석이 중요할까?
- 언제 Worst case 분석이 중요할까?

점근적 분석법: Big-Omega, Big-Theta

빅-오메가 (Ω) 표기법의 정의

For $T(n)$ a non-negative valued function, $T(n)$ is in the set $\Omega(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \geq c \cdot f(n)$ for all $n > n_0$

- 의미: 충분히 큰 모든 입력 데이터에 대해 (즉, $n > n_0$), 해당 알고리즘은 항상 $c \cdot f(n)$보다 많은 연산을 수행한다.
- **하한 (Lower bound)**

점근적 분석법: Big-Omega, Big-Theta

- 빅-오메가 (Ω) 표기법 예
  - $T(n) = c_1n^2 + c_2n$ 인 경우
    - $c_1n^2 + c^2n \geq c_1n^2$ for all $n > 1$
    - $T(n) \geq cn^2$ for $c = c_1$ and all $n > n_0 = 1$
    - Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

점근적 분석법: Big-Omega, Big-Theta

- 빅-세타 (Θ) 표기법
  - T(n) 이 O(g(n))과 Ω(g(n))에 모두 속할 경우, T(n)은 Θ(g(n))에 속한다.

Simplifying Rules

1. $f(n) \in O(g(n))$ 이고 $g(n) \in O(h(n))$ 이면 $f(n) \in O(h(n))$ 이다.
2. 상수 $k > 0$에 대해서 $f(n) \in O(kg(n))$이면 $f(n) \in O(g(n))$ 이다.
3. $f_1(n) \in O(g_1(n))$ 이고 $f_2(n) \in O(g_2(n))$ 이면
   $(f_1 + f_2)(n) \in O(\max(g_1(n), g_2(n)))$ 이다.
   - Ω의 경우에는? Θ의 경우에는?
4. $f_1(n) \in O(g_1(n))$ 이고 $f_2(n) \in O(g_2(n))$ 이면
   $f_1(n)f_2(n) \in O(g_1(n)g_2(n))$이다.

공간 복잡도

- **공간복잡도**Space complexity: 알고리즘/프로그램이 필요로하는 메모리의 양에 대한 복잡도
  - **Overhead**: 입력 데이터를 제외한 메모리 사용량

- **Space/Time trade-off**: 대게, 공간을 더 써서 시간을 줄일 수 있다. 반대도 가능하다.
  - 예 1) Factorial 계산
  - 예 2) 데이터 압축 (encoding/decoding)
- **Disk-based Space/Time Tradeoff**: 대게, 디스크 I/O를 줄이면, 시간도 줄어든다.
  - Disk I/O의 속도가 CPU의 computation 속도보다 너무 느려서.

Summary

- ❖ 알고리즘 개요
- ❖ 알고리즘 성능 분석
- ❖ **복잡도 예제**
- ❖ 문제의 복잡도

복잡도Complexity 예제

예제 1)

```
a = b;
```

예제 2)

```
sum = 0;
for (i = 0; i < n; i++)
    sum += n;
```

예제 3)

```
for (j=1; j<=n; j++)
    for (i=1; i<=j; i++)
        sum++;
for (k=0; k<n; k++)
    A[k] = k;
```

복잡도Complexity 예제

예제 4)

```
sum1 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum1++;
sum2 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
        sum2++;
```

복잡도Complexity 예제

예제 5)

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;
sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```

복잡도Complexity 예제

예제 6) 0~**C**의 값을 갖는 크기가 **P**인 배열 `value`가 주어졌을 때 가장 많이 등장하는 값 찾기

```
for (i=0; i<C; i++) // Initialize count
    count[i] = 0;
for (i=0; i<P; i++) // Look at all pixels
    count[value[i]]++; // Increment count
sort(count); // Sort pixel counts
```

복잡도Complexity 예제

예제 7) 이진 탐색Binary Search

45 찾기

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

Worst Case에 얼마나 많은 item들을 확인해야 할까?

이진 탐색 Binary Search

```java
/** @return The position of an element in sorted array A
with value k. If k is not in A, return A.length. */
static int binary(int[] A, int k) {
    int l = -1;
    int r = A.length; // l and r are beyond array bounds
    while (l+1 != r) { // Stop when l and r meet
        int i = (l+r)/2; // Check middle of remaining subarray
        if (k < A[i]) r = i; // In left half
        if (k == A[i]) return i; // Found it
        if (k > A[i]) l = i; // In right half
    }
    return A.length; // Search value not in A
}
```

Summary

- 알고리즘 개요
- 알고리즘 성능 분석
- 복잡도 예제
- **문제의 복잡도**

문제의 복잡도 Complexity of Problem

- 입력 혹은 입력의 수가 n인 어떤 문제 P(n)에 대해서,
- $P(n) \in O(f(n))$: upper bound가 $O(f(n))$인 알고리즘이 존재한다.
  - 최고의 알고리즘의 upper bound보다 P(n)의 upper bound가 더 나쁠 수는 없다.
- $P(n) \in \Omega(f(n))$: **모든 알고리즘**의 lower bound가 $\Omega(f(n))$에 속한다.
- 예) 정렬하기 문제의 upper bound? lower bound?

Questions?