자료구조 및 알고리즘개론 개요

HaRim Jung, Ph.D.

Visiting Professor / Senior Researcher

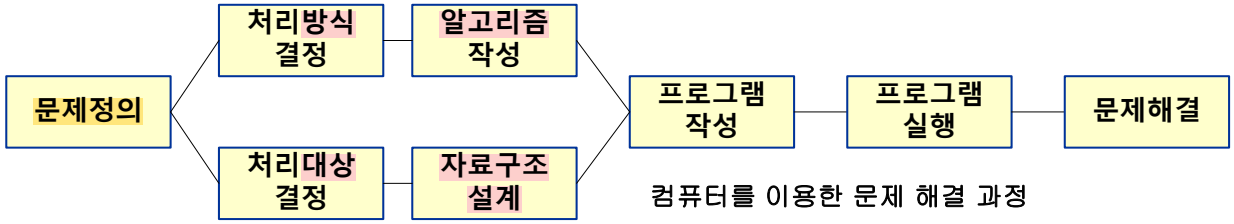
SKKU Institute for Convergence / Convergence Research Institute

Sungkyunkwan University, Korea

자료구조 및 알고리즘 소개 (1/7)

❑ 자료구조 및 알고리즘

- 우리는 컴퓨터를 이용하여 **프로그램을 작성하고 실행**함으로써 주어진 **문제를 해결**한다고 가정



컴퓨터를 이용한 문제 해결 과정

- **자료구조(data structure)**
  - **(1)** 현실 세계의 데이터(자료)들을 **논리적으로(logically) 나열하여 표현(논리적 구조로 표현)**하고,
  - **(2)** 프로그램 작성 시 (1)의 과정을 거친 데이터들을 컴퓨터가 (시간·공간) **효율적으로** 처리할 수 있도록 **컴퓨터(메모리)에 저장**하고 **조작(연산)**하는 방법 → 주어진 문제에 알맞은 자료구조 선택(혹은 설계)은 효율적인 알고리즘을 선택(혹은 설계)할 수 있게 함
- **알고리즘(algorithm)**
  - 주어진 문제에 어떠한 입력이 주어지더라도 해당 입력을 유한한 시간 내에 (정확한) 출력으로 전환시켜줄 수 있는 일련의 연산 절차
  - **문제(problem)**: 해답(solution)을 찾기 위해 물어보는 질문
    - 예: N 개의 항목으로 구성된 리스트 L에서 x라는 수가 있는가?
    - **매개변수(parameter):** 문제에서 어떤 특정 값이 주어지지 않은 변수(variable), e.g., L, n, x
    - **입력(input):** 매개변수에 특정 값을 지정한 것, e.g., L = [10, 7, 11, 5, 3, 8], n = 6, x = 5
    - **출력(output):** 입력에 대한 해답, e.g., "예"

자료구조 및 알고리즘 소개 (2/7)

❑ 자료구조 및 알고리즘 contd.

- **알고리즘의 표기(기술)**
  - **한글 또는 영어 등의 자연어(natural language)**

> 1. Start from the leftmost element of a list L of size n and one by one compare x with each element of L.
> 2. If *x* matches with an element, return “yes”.
> 3. If *x* does not match with any of elements, return “no”.

    - 프로그램(program)으로 전환하기 용이하지 않으며, 모호한 경우가 많음
  - **C, C++, C#, JAVA 등의 프로그래밍 언어(programming language)**
    - 프로그램이므로 직접 실행 가능(executable)하지만 가독성(readability)이 떨어짐

```c
#include <stdio.h>

int search(int lst[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (lst[i] == x)
            return i;
    return -1;
}

int main(void)
{
    int lst[] = {10, 7, 11, 5, 3, 8};
    int x = 5;
    int n = sizeof(lst) / sizeof(lst[0]);
    int result = search(lst, n, x);
    (result == -1) ? printf("Not found.") : printf("Found at index %d", result);
    return 0;
}
```

자료구조 및 알고리즘 소개 (3/7)

❑ 자료구조 및 알고리즘 contd.

- **알고리즘의 표기(기술) contd.**
  - **의사코드(pseudocode):** 프로그래밍 언어와 유사하게 연산 과정을 표현할 수 있는 언어

Algorithm 1 Linear Search

Parameter(Input) a list L, the number n of items in L, a number x
Output "yes" if x is in L and "no" if it is not.
Procedure
1: for each item i in L do
2: if $i == x$ then
3: return "yes"
4: end if
5: end for
6: return "no"
End of Procedure

    - 컴퓨터에서 직접 실행이 불가능하지만 가독성이 좋음
  - **Python(본 강의에서는 Python으로 알고리즘을 표기)**

```
1  def linear_search(lst, n, x):
2      for i in range(n):
3          if lst[i] == x:
4              return i
5      return -1
6
7  lst = [10, 7, 11, 5, 3, 8, 16, 13]
8  n = len(lst)
9  result = linear_search(lst, n, 30)
10 if result != -1:
11     print("Found at index {}.".format(result))
12 else:
13     print("Not found.")
```

자료구조 및 알고리즘 소개 (4/7)

❑ 자료구조 및 알고리즘 contd.

- **주어진 문제에 알맞은 자료구조 및 알고리즘 선택(혹은 설계)의 중요성**
  - **정렬되지 않은 리스트**에서 **선형검색(linear search)** vs. **정렬된 리스트**에서 **이진검색(binary search)**

    NOTE: 리스트는 데이터를 순차적으로 나열해 놓은 논리적 선형 구조
  - **정렬되지 않은 리스트에서 선형검색**
    - **문제**: 크기가 n인 정렬되지 않은 리스트 L에 x라는 수가 존재하는가?
    - **매개변수(입력)**: (1) 정렬되지 않은 리스트 L, (2) 양수 n(= 8), (3) 검색키 x(= 13)
    - **출력**: 존재하면 L에서 x의 위치, 존재하지 않으면 Not found

L	10	7	11	5	3	8	16	13

10	7	11	5	3	8	16	13	10 ≠ 13이므로 검색 계속
10	**7**	11	5	3	8	16	13	7 ≠ 13 이므로 검색 계속
10	7	**11**	5	3	8	16	13	11 ≠ 13이므로 검색 계속
10	7	11	**5**	3	8	16	13	5 ≠ 13이므로 검색 계속
10	7	11	5	**3**	8	16	13	3 ≠ 13이므로 검색 계속
10	7	11	5	3	**8**	16	13	8 ≠ 13이므로 검색 계속
10	7	11	5	3	8	**16**	13	16 ≠ 13이므로 검색 계속
15	40	30	80	10	70	50	**13**	13 =13이므로 검색 성공

- 더 빠르게 검색할 수는 없을까? 리스트 L이 **정렬**되어 있지 않다면 불가능

자료구조 및 알고리즘 소개 (5/7)

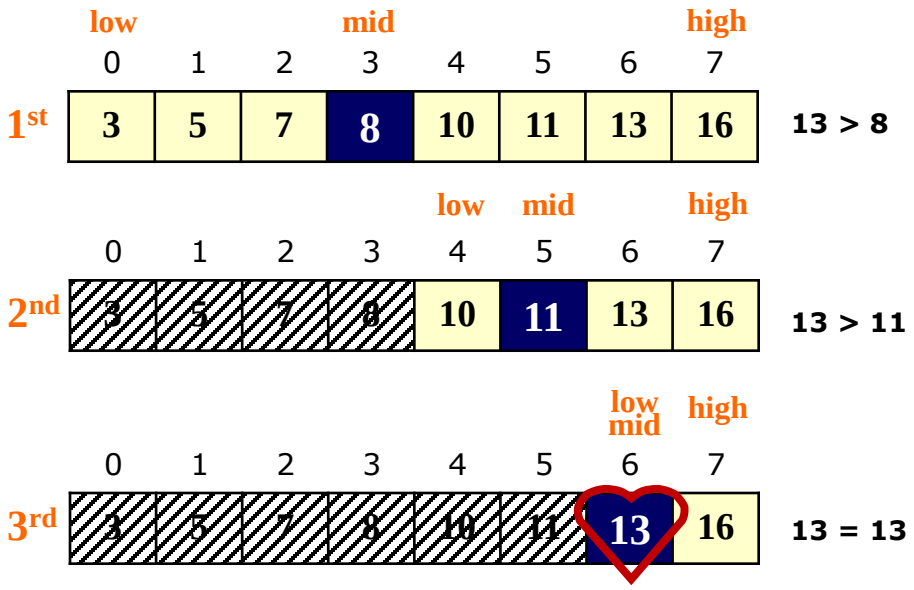
❑ 자료구조 및 알고리즘 contd.

- **주어진 문제에 알맞은 자료구조 및 알고리즘 선택(혹은 설계)의 중요성 contd.**
  - **정렬된 리스트에서 이진검색**
    - **문제**: 크기가 n인 정렬된 리스트 L에 x라는 수가 존재하는가?
    - **매개변수(입력)**: (1) 정렬되지 않은 리스트 L, (2) 양수 n(= 8), (3) 검색키 x(= 13)
    - **출력**: 존재하면 L에서 x의 위치, 존재하지 않으면 Not found

L	3	5	7	8	10	11	13	16

```
1  def binary_search(lst, n, x):
2      mid = 0
3      low = 0
4      high = n - 1
5      while low <= high:
6          mid = (low + high) // 2
7          if x == lst[mid]:
8              return mid
9          if x < lst[mid]:
10             high = mid - 1
11         else:
12             low = mid + 1
13     return -1
14
15 lst = [3, 5, 7, 8, 10, 11, 13, 16]
16 n = len(lst)
17 result = binary_search(lst, n, 30)
18 if result != -1:
19     print("Found at index {}.".format(result))
20 else:
21     print("Not found.")
```

lst → L



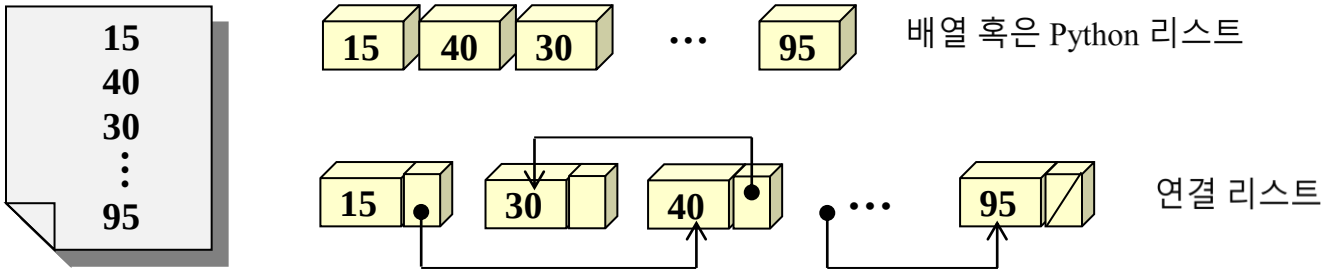
while문을 수행할 때마다 검색 대상의 크기가 절반으로 감소하기 때문에, 입력 사이즈를 **n**이라고 하면 최악의 경우 **lgn + 1**번을 비교

자료구조 및 알고리즘 소개 (6/7)

❑ 자료구조 및 알고리즘개론 과목의 목표

- 현실 세계의 모든 경우에 대해 효율적인 논리적 데이터 나열 방법(논리적 구조)은 존재하지 않음
- 보편적으로 가장 많이 사용되는 논리적 구조 학습을 통해 (1) **특정 문제 해결에 적합한 논리적 구조를 선택(혹은 설계)**할 수 있는 능력을 키울 수 있고, (2) 프로그램을 통해 **어떻게 선형 구조로 저장되는 컴퓨터 메모리에 다양한 논리적 구조를 가진 데이터들이 저장될 수 있는지를 이해**하도록 함
- 보편적으로 가장 많이 사용되는 데이터의 논리적 구조는 형태에 따라 크게 두 가지 종류로 나뉨
  - **선형 구조(리스트)**: 데이터 간의 앞뒤 관계가 일대일로 고정되어 있는 구조, e.g., 배열(array), Python의 리스트(list), 연결 리스트(linked list), 스택(stack), 큐(queue) 등

  NOTE: 배열 혹은 Python의 리스트는 데이터의 논리적 구조와 메모리에 저장되는 물리적 구조가 일치하지만, 연결 리스트는 일치하지 않음



  - **비선형 구조**: 데이터 간의 앞뒤 관계가 일대일로 고정되지 않은 구조, e.g., 트리(tree), 그래프(graph)

자료구조 및 알고리즘 소개 (7/7)

❑ 자료구조 및 알고리즘개론 과목의 목표 contd.

- (1) 알고리즘을 설계하는 **대표적인 전략을 학습**하고 (2) 각 전략을 여러 **기존 문제에 적용**시키는 학습을 통해서, **새로운 문제**가 주어졌을 때 그 문제를 해결하기 위해 **가장 적합한 알고리즘 설계 전략을 선택**하여 알고리즘을 설계할 수 있도록 함
  - '적합한'이란 설계한 알고리즘이 **정확성**과 **효율성**을 보장할 수 있어야 한다는 의미
- 알고리즘의 효율성은 (**시간**·공간) **계산 복잡도(computational complexity)**로 분석(analysis)하는 것이 일반적임. 따라서 알고리즘의 계산 복잡도를 구하는 방법을 학습함으로써 **새로운 문제**를 해결하기 위해 설계한 **자료구조 및 알고리즘의 효율성을 분석**할 수 있도록 함
  - NOTE: 자료구조의 효율성은 해당 자료구조를 조작하기 위한 연산의 수행 시간으로 측정하며, **측정 방식은 알고리즘의 효율성을 분석하는 방식과 동일**

리스트 크기	선형검색	이진검색	비고
n	n	lgn + 1	최악의 경우에 대한 계산 복잡도
128	128	8	
1,024	1,024	11	
1,048,576	1,048,576	21	
4,294,967,296	4,294,967,296	33	

알고리즘의 효율성 분석 (1/6)

❑ 시간 효율성과 공간 효율성

- 시간 효율성(time efficiency)은 문제를 해결하는데 얼마나 많은 시간을 요하는가를 지칭
- 공간 효율성(space efficiency)은 문제를 해결하는데 얼마나 많은 공간을 필요로 하는가를 지칭
- 일반적으로 **시간 효율성**이 공간 효율성보다 더욱 강조가 됨
  - 그 이유는 시간이 공간보다 비싸기 때문임(시간 효율성은 CPU 성능과 직결되며 공간 효율성은 메모리 용량과 직결됨)
- 시간·공간 효율성을 뒤집어 표현한 것이 시간·공간 **복잡도(complexity)** → 복잡도가 높을수록 효율성이 저하
- 알고리즘의 효율성은 복잡도에 기반하여 분석

❑ 알고리즘의 시간 복잡도(time complexity) 분석

- **입력 크기**에 따라서 **단위 연산**이 몇 번 수행되는지 결정하는 절차
  - 입력 크기: 리스트 크기(예: 선형 검색, 이진 검색), 트리의 노드 수, 그래프의 정점(vertex)과 간선(edge)의 수 등
  - 단위 연산: 알고리즘을 수행하는 데 있어서 가장 핵심적인 역할을 담당하는 연산으로 비교문에 있는 비교 연산(예: 선형 검색, 이진 검색), 할당문에 있는 수치 연산 후 할당 연산(예: 리스트 내의 모든 항목의 값 더하기) 등

핵심적인 역할을 담당하는 연산을 주관적으로 선택

하지만 대부분의 전문가들은 동일한 단위 연산을 선택하게 되며, 서로 다른 단위 연산을 선택하더라고 최종적인 시간 복잡도는 동일하게 될 정도로 유사한 중요도를 지닌 단위 연산을 택함

알고리즘의 효율성 분석 (2/6)

❑ (시간) 복잡도 분석 방법의 종류

- 모든 경우 분석(every-case analysis)
  - 복잡도는 입력 크기 n에만 종속(dependent)적임
  - 입력 값(입력 내용)과는 무관(independent)하게 복잡도는 항상 일정
  - 복잡도 표기 방법: T(n)
  - 예: 리스트 내의 모든 항목의 값 더하기
    - **문제**: 크기가 n인 리스트 L의 모든 항목의 값을 더하라
    - **입력**: (1) 리스트 L, (2) 양수 n **출력**: L 내의 모든 항목의 합
    - **입력 크기**: n

```
1  def sum(lst, n):
2      result = 0
3      for i in range(n):
4          result = result + lst[i]
5      return result
6
7  lst = [10, 7, 11, 5, 13, 8]
8  n = len(lst)
9  print(sum(lst, n))
```

    - **단위 연산**: result = result + lst[i] (수치 연산 후 할당 연산)
    - 리스트 L에 있는 항목 값에 상관 없이 for 루프를 n번 실행하므로 시간 복잡도 T(n) = n

알고리즘의 효율성 분석 (3/6)

❑ (시간) 복잡도 분석 방법의 종류 contd.

- 최악의 경우 분석(worst-case analysis)
  - 복잡도는 입력 크기 n과 입력 값 모두에 종속
  - 단위 연산이 수행되는 횟수가 최대(최악)인 경우 선택
  - 복잡도 표기 방법: W(n)
  - 예: 선형 검색 알고리즘에 대한 최악의 경우 분석
    - **문제**: 크기가 n인 리스트 L에 x라는 수가 존재하는가?
    - **입력**: (1) 리스트 L, (2) 양수 x **출력**: x라는 수가 존재하면 x의 위치
    - **입력 크기**: n

```
1  def linear_search(lst, n, x):
2      for i in range(n):
3          if lst[i] == x:
4              return i
5      return -1
```

    - **단위 연산**: if lst[i] == x (비교문에 있는 비교 연산)
    - x가 리스트의 마지막에 존재하거나 리스트에 존재하지 않을 경우 단위 연산이 n번 수행되므로 W(n) = n
    - 선형 검색 알고리즘은 리스트 항목 값에 따라서 검색하는 횟수가 달라지므로 **모든 경우 분석이 불가능**

알고리즘의 효율성 분석 (4/6)

(시간) 복잡도 분석 방법의 종류 contd.

- 최선의 경우 분석(best-case analysis)
  - 복잡도는 입력 크기와 입력 값 모두에 종속
  - 단위 연산이 수행되는 횟수가 최소(최선)인 경우 선택
  - 복잡도 표기 방법: B(n)
  - 예: 선형 검색 알고리즘에 대한 최선의 경우 분석
    - **문제**: 크기가 n인 리스트 L에 x라는 수가 존재하는가?
    - **입력**: (1) 리스트 L, (2) 양수 x **출력**: x라는 수가 존재하면 x의 위치
    - **입력 크기**: n

```
1    def linear_search(lst, n, x):
2        for i in range(n):
3            if lst[i] == x:
4                return i
5        return -1
```

    - **단위 연산**: if lst[i] == x (비교문에 있는 비교 연산)
    - x가 리스트의 처음에 존재할 경우 단위 연산이 1번 수행되므로 B(n) = 1

알고리즘의 효율성 분석 (5/6)

❑ (시간) 복잡도 분석 방법의 종류 contd.

- 평균의 경우 분석(average-case analysis)
  - 복잡도는 입력 크기와 입력 값 모두에 종속
  - 모든 입력에 대해서 단위 연산이 수행되는 횟수의 기대치(평균)
  - 복잡도 표기 방법: A(n)
  - 확률적 계산이 필요함
    - 각 입력 값에 대해서 확률 할당이 다를 수 있음: 예를 들어 1부터 100까지 수의 정렬된 값을 가지는 리스트를 순차적으로 검색하는 경우, 검색하고자 하는 값이 90보다 큰 경우가 더 많다면 평균 단위 연산 수행 횟수는 검색하고자 하는 값이 골고루 분포되는 경우(균등분포)와 다름
  - 예: 선형 검색 알고리즘에 대한 평균의 경우 분석
    - **문제**: 크기가 n인 리스트 L에 x라는 수가 존재하는가?
    - **입력**: (1) 리스트 L, (2) 양수 x **출력**: x라는 수가 존재하면 x의 위치
    - **입력 크기**: n

```
1    def linear_search(lst, n, x):
2        for i in range(n):
3            if lst[i] == x:
4                return i
5        return -1
```

    - **단위 연산**: if lst[i] == x (비교문에 있는 비교 연산)

알고리즘의 효율성 분석 (6/6)

❑ (시간) 복잡도 분석 방법의 종류 contd.

- 평균의 경우 분석(average-case analysis) contd.
  - **가정:** L 내의 모든 항목이 서로 다른 값을 가짐
  - **경우 1**: x가 L 내에 **반드시 존재하는 경우**만 고려
    - $1 \leq i \leq n$에 대해서 x가 L의 i 번째에 존재할 확률 = $1/n$
    - x가 L의 k 번째에 존재한다면 x를 찾기 위해서 수행하는 단위 연산의 횟수는 i번이므로
    - $A(n) = \sum_{i=1}^{n} i \times \frac{1}{n} = 1/n \times n(n+1)/2 = (n+1)/2$
  - **경우 2**: x가 L 내에 **존재하지 않을 경우**도 고려
    - x가 L 내에 존재할 확률을 p라고 하면,
      - » x가 L 내에 i 번째 있을 확률 = p/n
      - » x가 L 내에 존재하지 않을 확률 = $1-p$
      - » 따라서 $A(n) = \sum_{i=1}^{n}(i \times \frac{p}{n}) + n(1-p) = p/n \times n(n+1)/2 + n(1-p) = \frac{n+1}{2}p + n(1-p) = \boldsymbol{n}(1-\frac{\boldsymbol{p}}{\boldsymbol{2}}) + \frac{\boldsymbol{p}}{\boldsymbol{2}}$
- 최악·최선·평균의 경우 분석 방법 중에서 어떤 분석을 사용할 것인가?
  - 일반적으로 최악 혹은 평균의 경우 분석 방법을 사용