Computational Thinking and Algorithms

컴퓨터개론

(Introduction to Computer Systems)

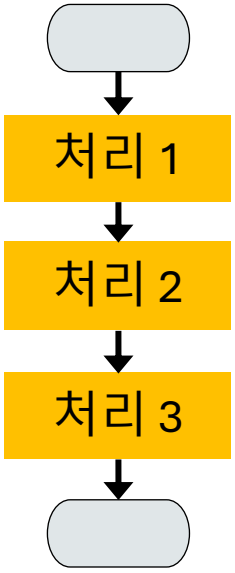
GEN1030

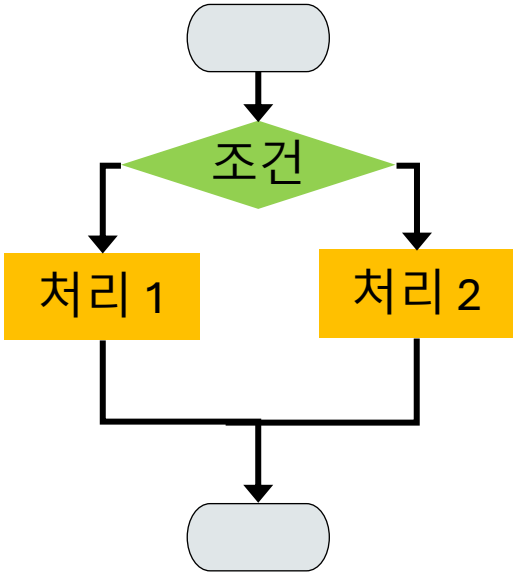
알고리즘

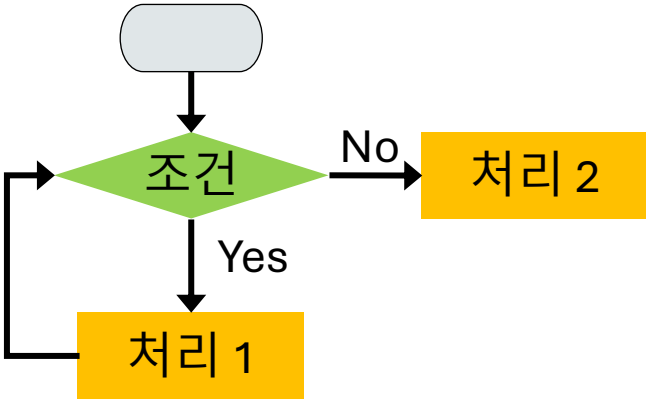
알고리즘 설계

- 알고리즘 설계
  - 가장 효율적인 문제 해결 방법을 찾아내는 과정

- 알고리즘 제어 구조 종류
  - 순차 구조: 명령 순서대로 하나씩 수행
  - 선택 구조: 조건의 결과에 따라 명령 선택해서 실행
  - 반복 구조: 같은 동작 여러 번 반복해 수행해야 할 때







알고리즘의 조건

- 입력(Input)
  - 입력되는 데이터가 0개 이상 (필수 입력이 없어도 됨)
- 출력(Output)
  - 하나 이상의 결과가 반드시 생성
- 명확성(Definiteness)
  - 각 단계 명령어는 모호하지 않고 명확해야 함
- 유한성(Finiteness)
  - 알고리즘은 반드시 종료되어야 함
- 유효성/실행 가능성(Effectiveness)
  - 알고리즘의 모든 명령은 실행 가능해야 함

알고리즘 분석

- 알고리즘 복잡도
  - 알고리즘의 효율성 나타냄
  - 입력 데이터의 양(n) 증가할 때 얼마나 더 많은 자원을 소모하는지 수학적으로 표현한 것
    - 입력이 커졌을 때 얼마나 빨리 실행되는지, 얼마나 많은 메모리 사용하는지
- 시간 복잡도(Time Complexity)
  - 실행 시간
- 공간 복잡도(Space Complexity)
  - 메모리 사용량

시간 복잡도(Time Complexity)

- 알고리즘이 실행되고 종료될 때까지 어느 정도의 시간이 필요한지 측정하는 방법
  - 실제 시간은 컴퓨터 사양에 따라 다름 → 연산 횟수로 계산
  - 입력 데이터 크기에 따라 실행시간이 어떻게 증가하는지

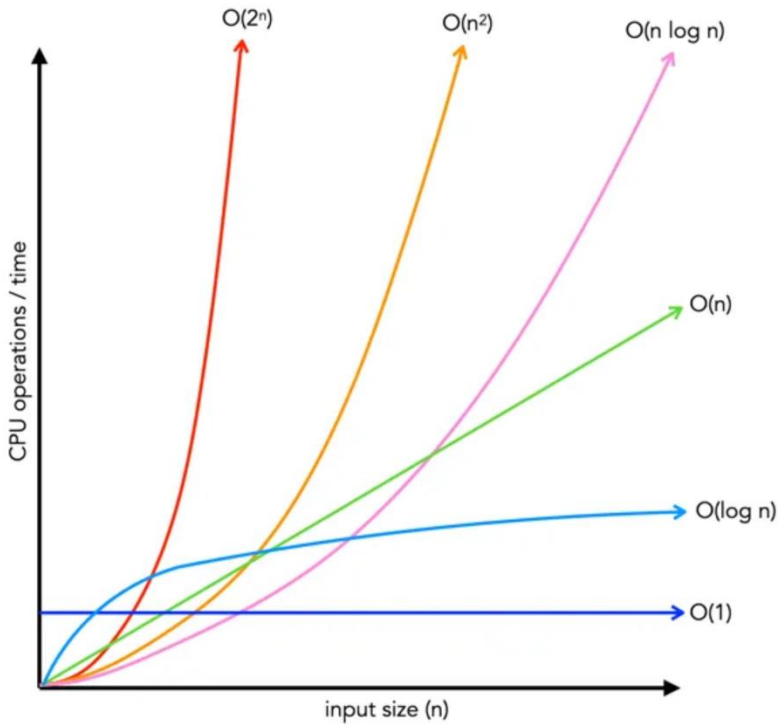
- **Big O 표기법(Notation)**
  - 복잡도를 근삿값으로 표현

```
sum = 0, array = [1, 2, 3, ..., n]
for x in array:
    sum += x
```

→ **O(n)**

  - 데이터 증가에 따른 성능 변화의 최악의 경우 나타냄
    - "아무리 느려도 이 정도 성능은 나옴"
  - 가장 차수가 높은 항(term)만을 표시
    - $f(x) = 4x^3 + 2x$ → Big O 표기법은 $O(x^3)$

Big O 표기법



알고리즘 분석에서 보통
$\log(n) = \log_2(n)$

- O(1)
  - 상수 시간
  - 데이터 개수와 무관하게 일정한 시간이 걸림
- O(log n)
  - 로그 시간
  - 실행할 때 마다 처리할 양이 반으로 줄어듦 (ex. Binary search)
- O(n)
  - 선형 시간
  - 데이터 양 만큼 연산 횟수가 늘어남
- O(n*logn)
  - 선형 로그 시간
  - 효율적인 정렬 알고리즘의 속도
- $O(n^2)$
  - 2차 시간
  - 연산 횟수가 n^2에 비례해서 증가

Why it matters?

- 입력 데이터 개수에 따른 시간 복잡도 증가율 비교

n	log(n)	n*log(n)	n^2
2	1	2	4
4	2	8	16
8	3	24	64
16	4	64	256
32	5	160	1024
64	6	384	4096
128	7	896	16384

“입력 커질수록 차이가 커짐”

공간 복잡도(Space Complexity)

- 알고리즘이 사용하는 메모리의 양
  - 입력 크기(n)에 따라 얼마나 증가하는지
  - 입력 데이터 외에 알고리즘 해결위해 '추가적으로' 얼마나 필요한가 (auxiliary space)

- Examples

```
array = [1, 2, ..., n]
new_array = []
for x in array:
    new_array.append(x * 2)
```

→ **O(n)**

```
array = [1, 2, ..., n]
sum = 0
for x in array:
    sum += x
```

→ **O(1)**

정렬 알고리즘

- 정렬(Sort)
  - 데이터를 일정한 규칙에 따라 나열하는 것

- 정렬 알고리즘(Sort Algorithm)
  - 주어진 데이터를 규칙에 따라 나열하는 알고리즘

  - 종류
    - 선택 정렬(Selection Sort)
    - 버블 정렬(Bubble Sort)
    - 삽입 정렬(Insertion Sort)
    - 병합 정렬(Merge Sort)
    - …

선택 정렬(Selection Sort)

- 데이터 중 가장 작은 데이터의 위치를 가장 앞에 있는 데이터 위치와 바꾸는 정렬 알고리즘

- 수행 과정
  1. 정렬되지 않은 데이터 중 최솟값을 찾는다
  2. 최솟값과 제일 앞에 위치한 값의 위치를 서로 바꾼다
  3. 교체 된 제일 앞 위치를 제외한 나머지에 대해 위 과정을 반복한다

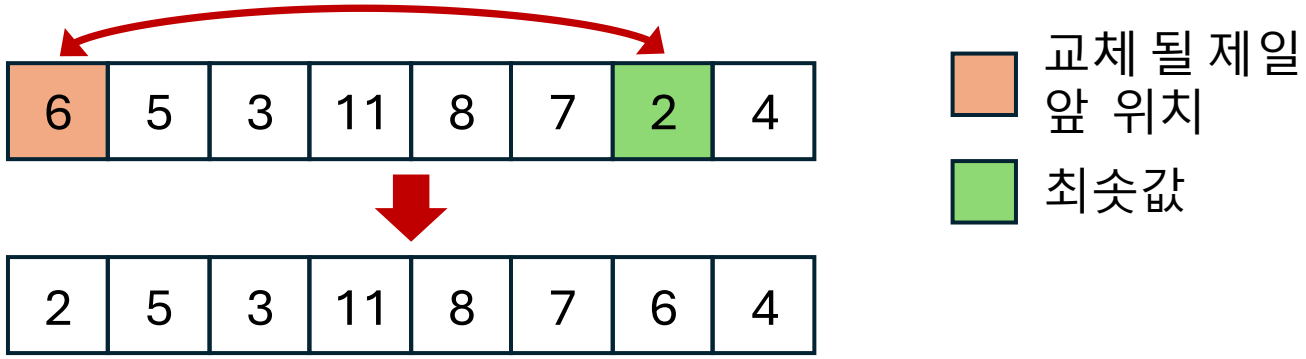
선택 정렬(Selection Sort)

- 수행 과정
  1. 정렬되지 않은 데이터 중 최솟값을 찾는다
  2. 최솟값과 제일 앞에 위치한 값의 위치를 서로 바꾼다
  3. 교체 된 제일 앞 위치를 제외한 나머지에 대해 위 과정을 반복한다

6	5	3	11	8	7	2	4

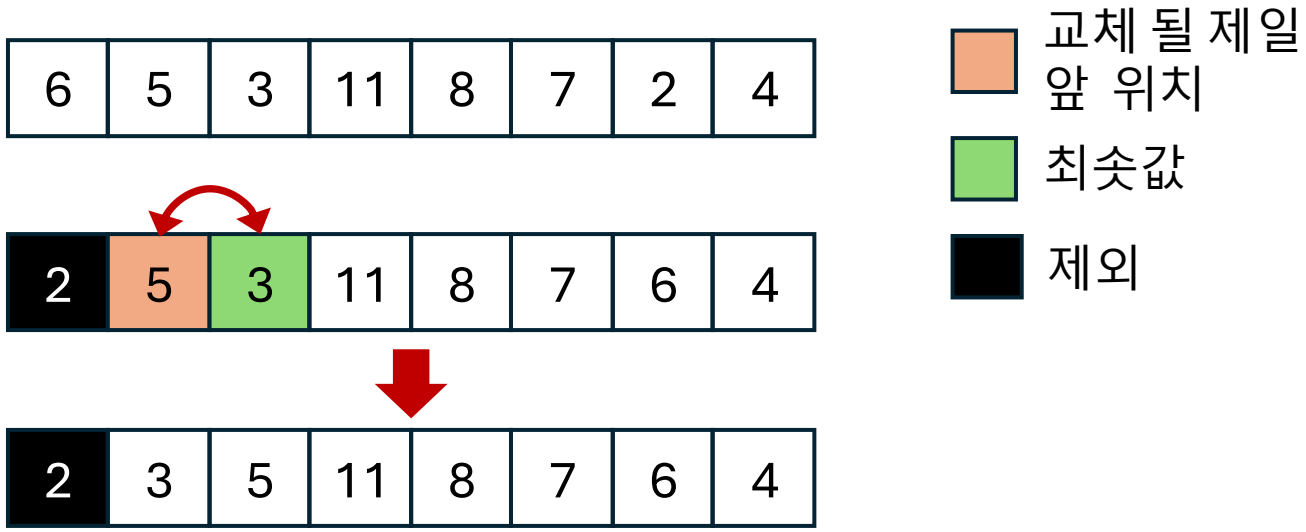
선택 정렬(Selection Sort)

- 수행 과정
  1. 정렬되지 않은 데이터 중 최솟값을 찾는다
  2. 최솟값과 제일 앞에 위치한 값의 위치를 서로 바꾼다
  3. 교체 된 제일 앞 위치를 제외한 나머지에 대해 위 과정을 반복한다



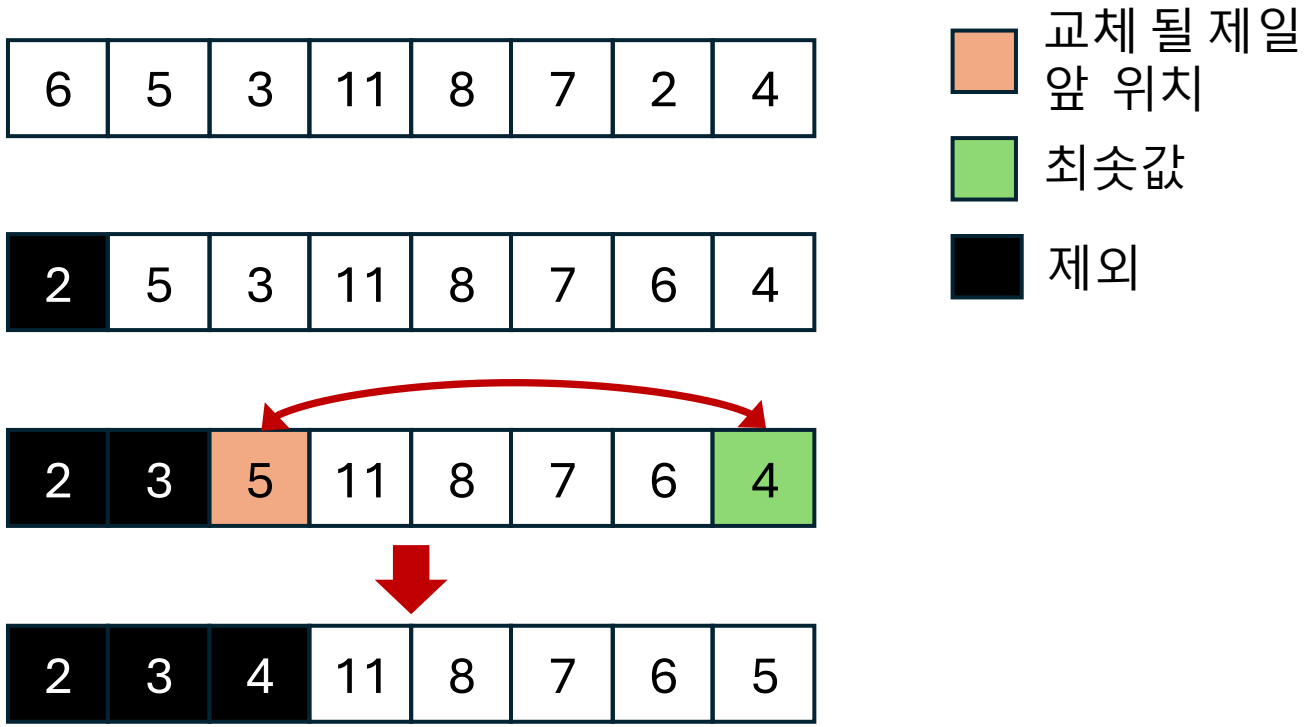
선택 정렬(Selection Sort)

- 수행 과정
  1. 정렬되지 않은 데이터 중 최솟값을 찾는다
  2. 최솟값과 제일 앞에 위치한 값의 위치를 서로 바꾼다
  3. 교체 된 제일 앞 위치를 제외한 나머지에 대해 위 과정을 반복한다



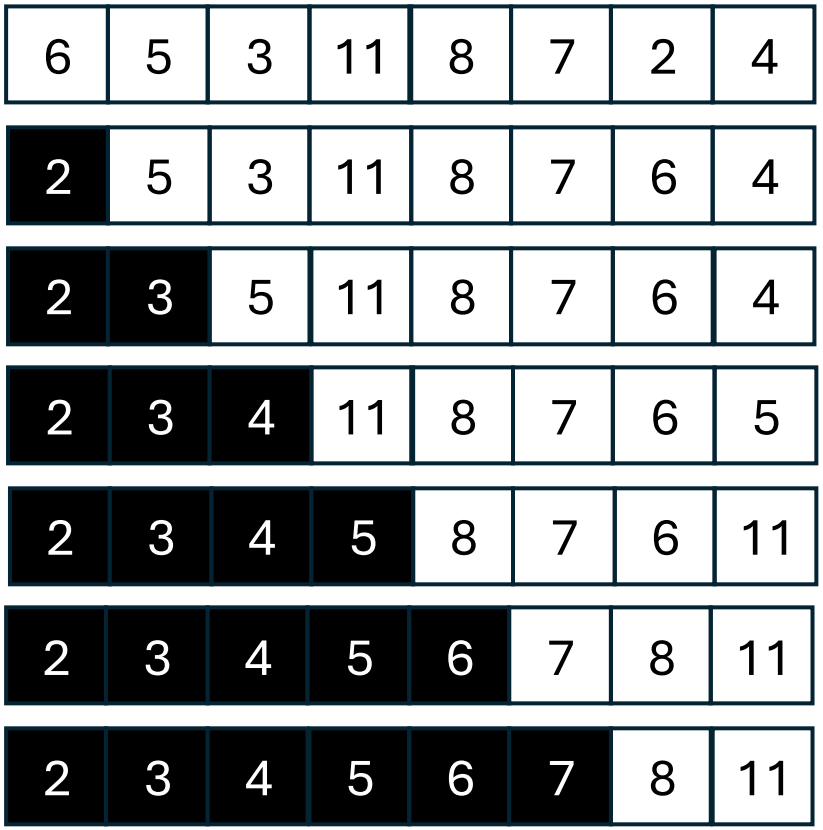
선택 정렬(Selection Sort)

- 수행 과정
  1. 정렬되지 않은 데이터 중 최솟값을 찾는다
  2. 최솟값과 제일 앞에 위치한 값의 위치를 서로 바꾼다
  3. 교체 된 제일 앞 위치를 제외한 나머지에 대해 위 과정을 반복한다



선택 정렬(Selection Sort) - Complexity

Space complexity **O(1)**



Time complexity **O(n^2)** ⬅ $\frac{(n-1)*n}{2}$

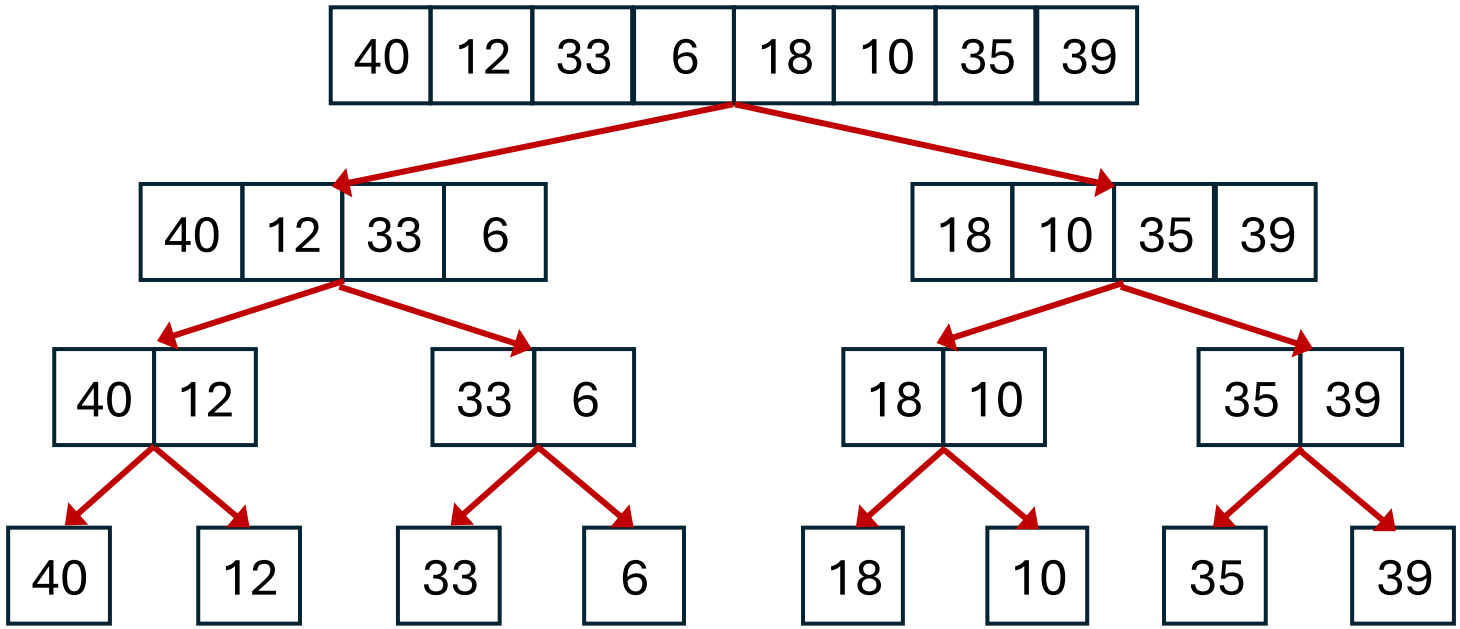
병합 정렬(Merge Sort)

- 정렬 된 여러 개의 자료 집합을 병합하여 하나의 정렬된 집합으로 만드는 정렬 방법

- 수행 과정
  - 분할(Divide): 자료들을 부분집합들로 분할한다
  - 정복(Conquer): 부분집합에 있는 원소를 정렬한다
  - 통합(Combine): 정렬된 부분집합들을 하나의 집합으로 결합한다

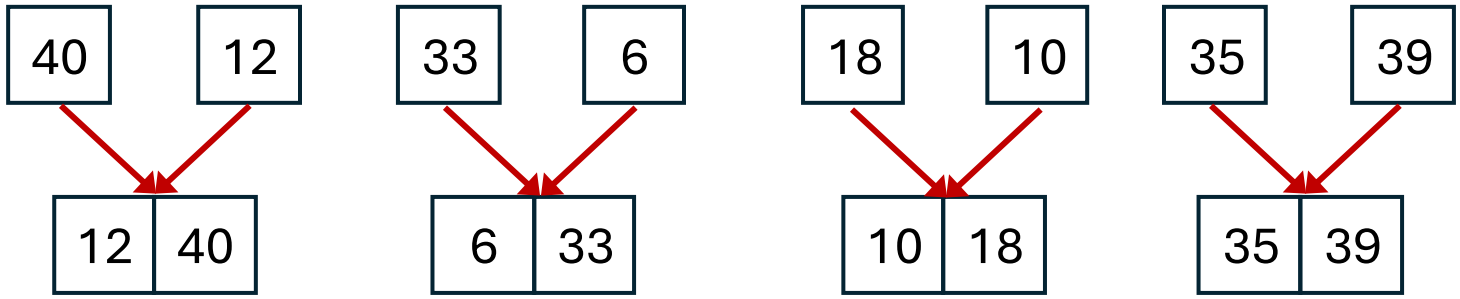
병합 정렬(Merge Sort)

- 수행 과정
  1. Divide: 전체 데이터 집합이 최소 원소 단위가 될 때까지 분할 작업을 반복



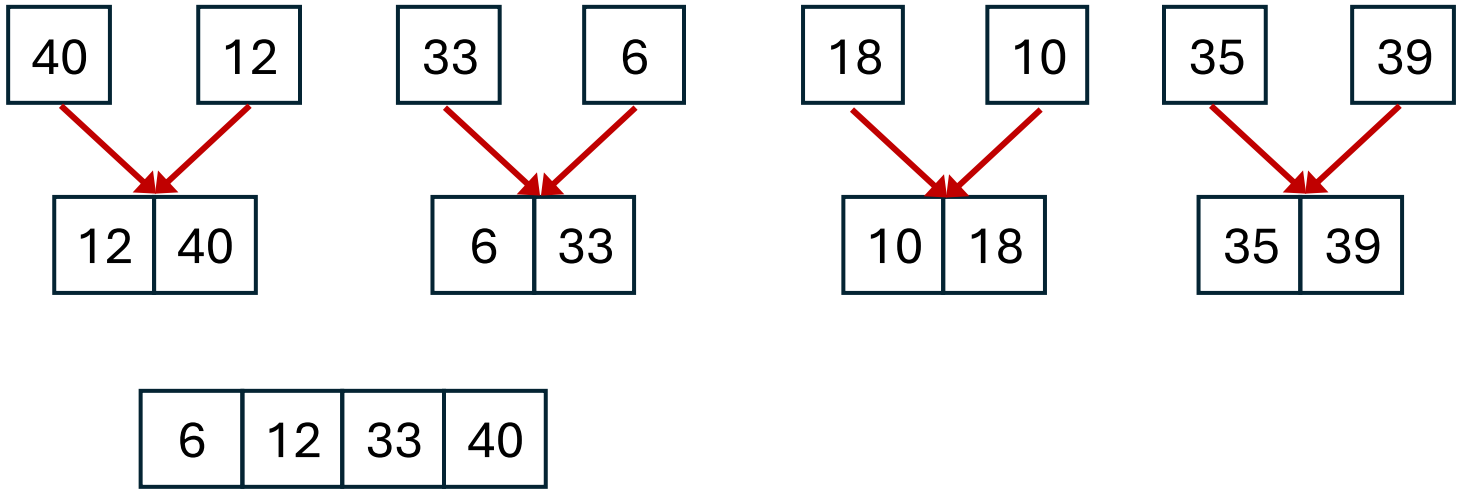
병합 정렬(Merge Sort)

- 수행 과정
  1. Conquer and Combine: 부분집합 두 개를 정렬하여 하나로 결합. 최종 하나의 집합이 남을 때까지 과정 반복.



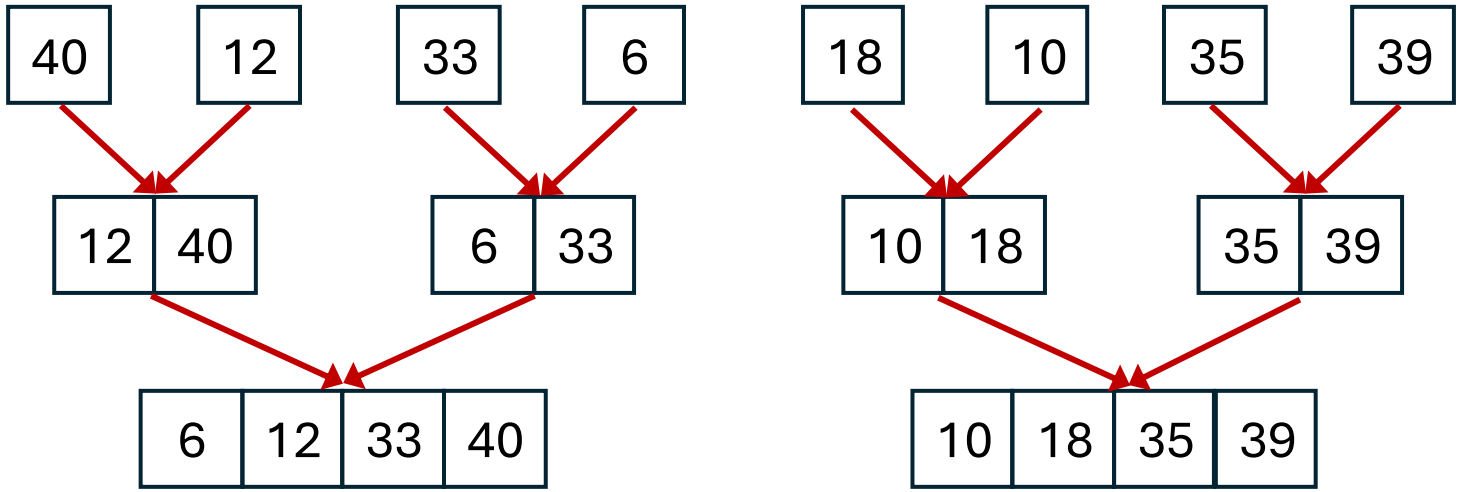
병합 정렬(Merge Sort)

- 수행 과정
  1. Conquer and Combine: 부분집합 두 개를 정렬하여 하나로 결합. 최종 하나의 집합이 남을 때까지 과정 반복.



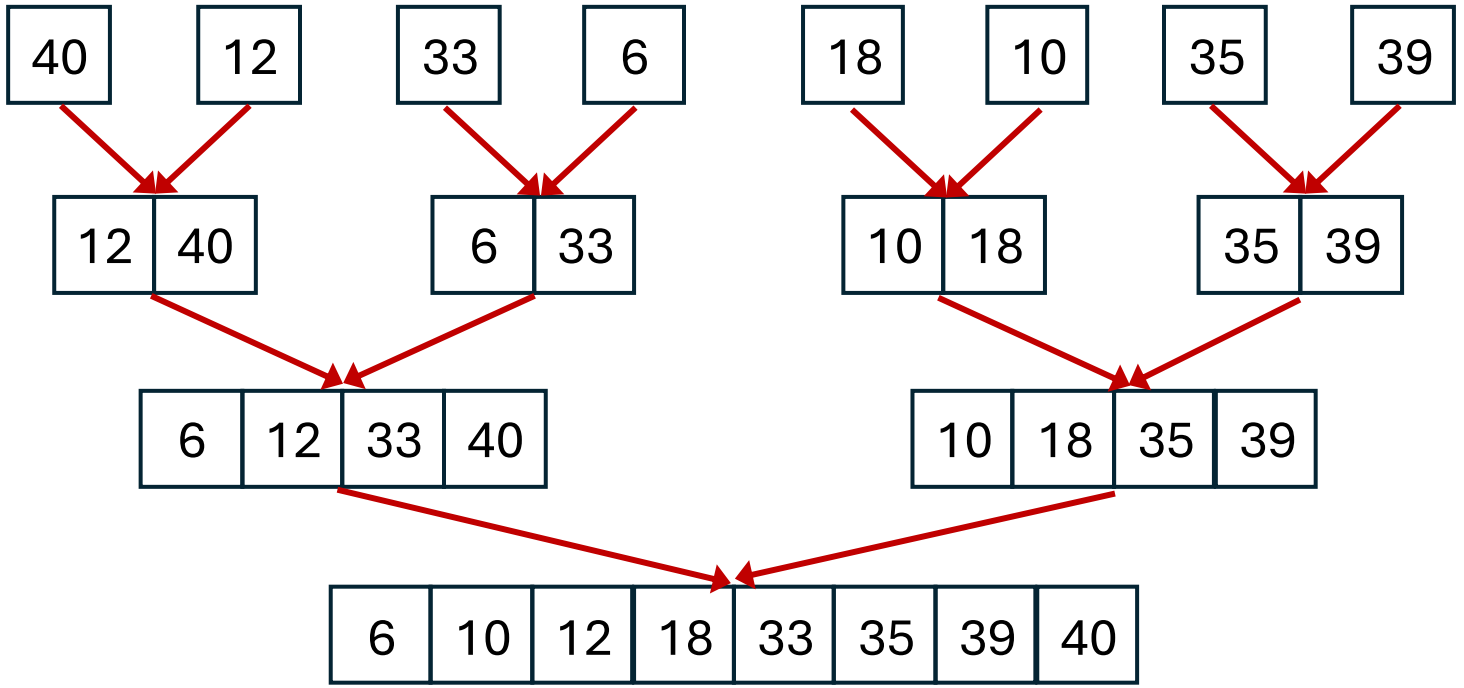
병합 정렬(Merge Sort)

- 수행 과정
  1. Conquer and Combine: 부분집합 두 개를 정렬하여 하나로 결합. 최종 하나의 집합이 남을 때까지 과정 반복.

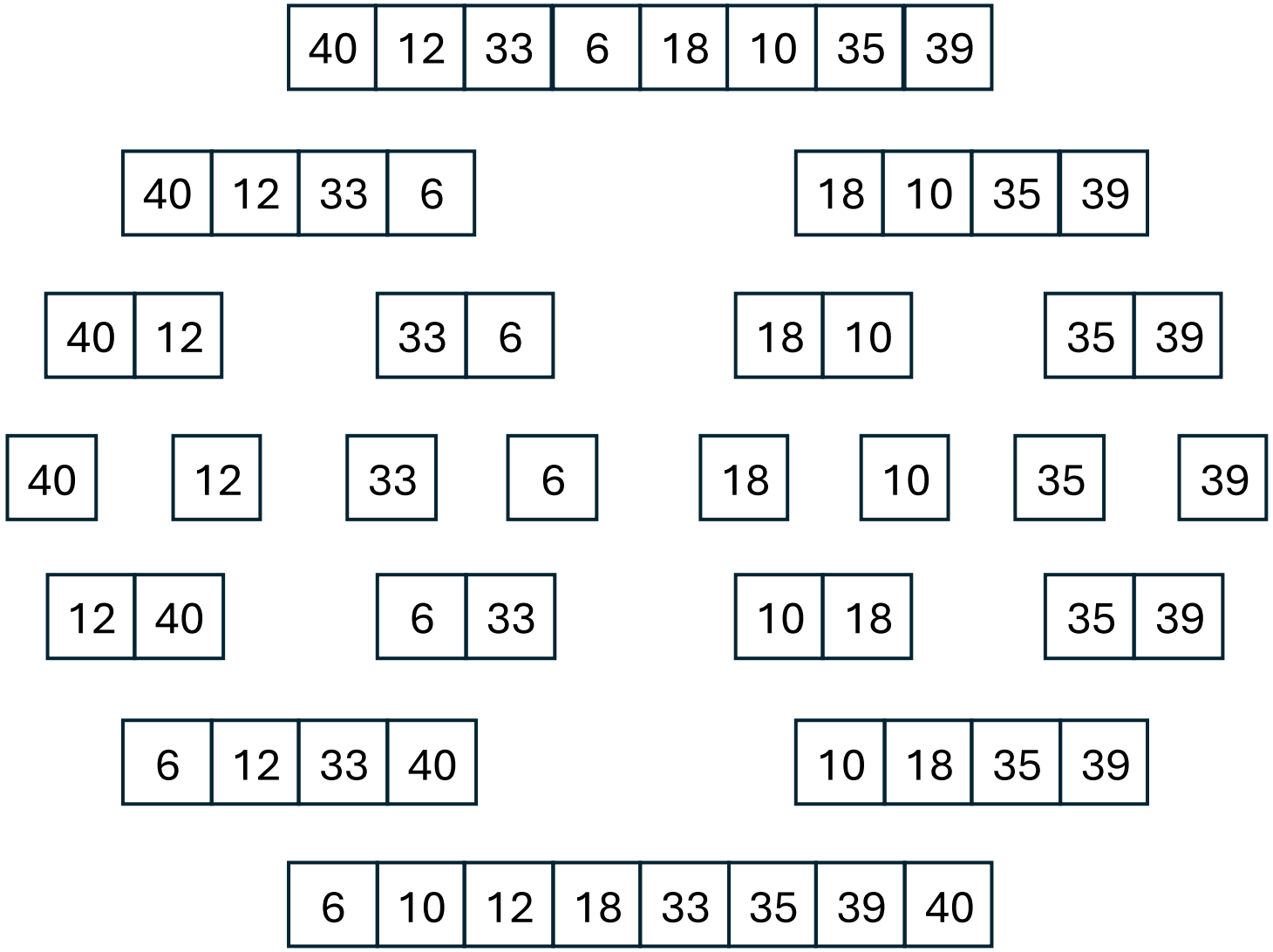


병합 정렬(Merge Sort)

- 수행 과정
  1. Conquer and Combine: 부분집합 두 개를 정렬하여 하나로 결합. 최종 하나의 집합이 남을 때까지 과정 반복.



병합 정렬(Merge Sort)



Time Complexity

O(nlogn)

검색 알고리즘

- 검색(Search)
  - 특정 데이터 집합에서 어떤 조건이나 성질을 만족하는 데이터를 찾는 것

- 검색 알고리즘(Search Algorithm)
  - 검색의 개념을 활용한 알고리즘

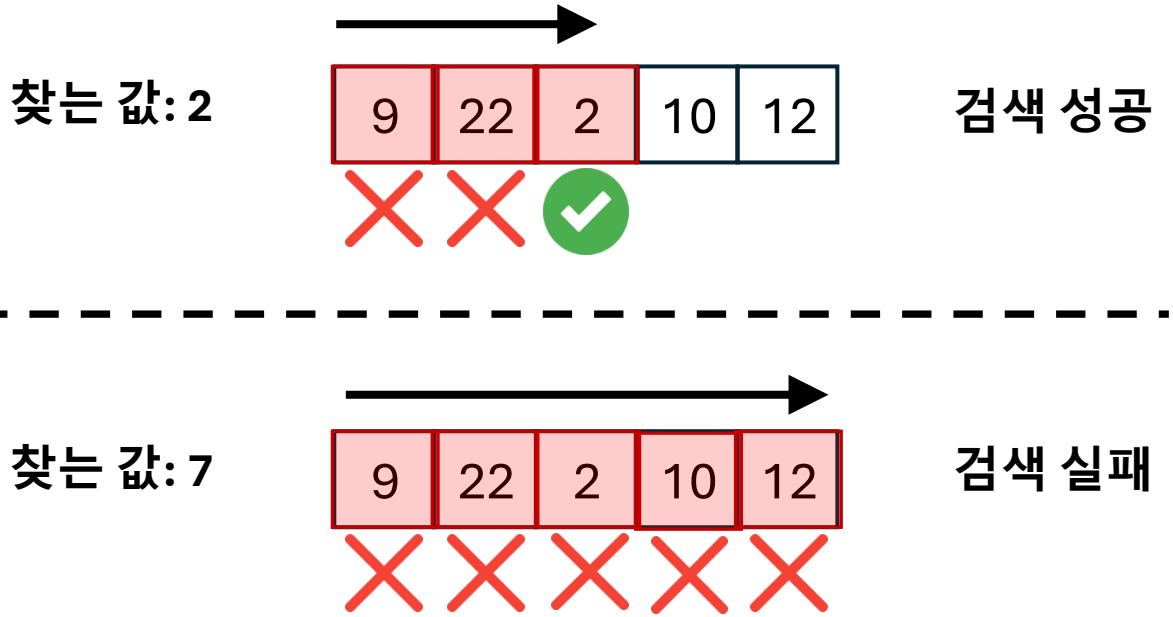
  - 종류
    - 순차 검색(Sequential Search)
    - 이진 검색(Binary Search)
    - 해싱(Hashing)
    - …

순차 검색(Sequential Search)

- 일렬로 나열된 데이터를 처음부터 끝까지 순서대로 검색하는 방법
- 배열과 같은 선형 자료구조에서 자주 쓰임
- 검색해야 하는 데이터 양에 따라 효율이 달라짐
  - 검색해야 할 데이터가 정렬된 상태인지 아닌지 등에 따라 검색 실패 판단 시점 달라짐

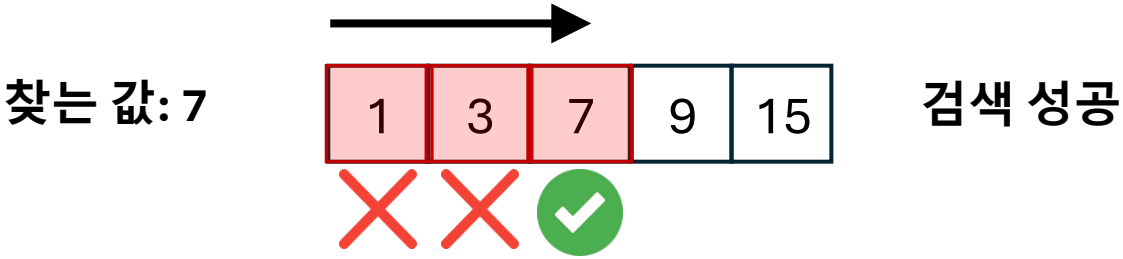
순차 검색(Sequential Search)

- **"정렬 되어 있지 않은" 데이터**
  - 첫번째 데이터부터 마지막 데이터까지 순서대로 비교 필요할 수 있음
  - 마지막 데이터까지 비교해도 찾는 값이 없으면 검색 실패



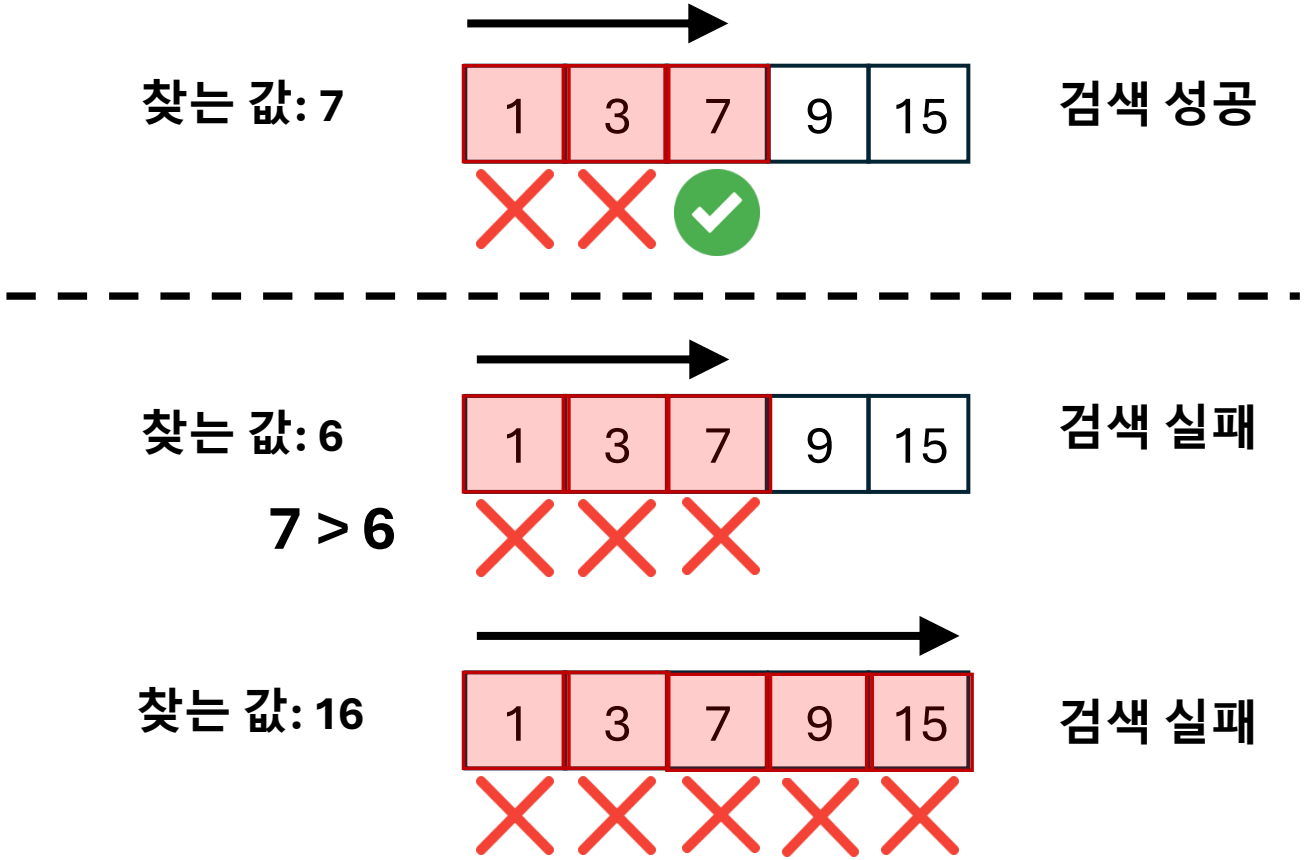
순차 검색(Sequential Search)

- **"정렬 되어 있는" 데이터**
  - 데이터 값이 찾는 값보다 크면 찾는 데이터가 없다는 뜻
    - 검색 실패 여부를 빠르게 파악 가능



순차 검색(Sequential Search)

- **"정렬 되어 있는" 데이터**
  - 데이터 값이 찾는 값보다 크면 찾는 데이터가 없다는 뜻
    - 검색 실패 여부를 빠르게 파악 가능



순차 검색(Sequential Search)

- **Time complexity**
  - Best case: 데이터가 맨 앞에 있음
    → O(1)
  - Worst case: 데이터가 맨 마지막에 있음 or 리스트에 없음
    → O(n)

- **Space complexity**
  - 추가 메모리 사용 없음
    → O(1)

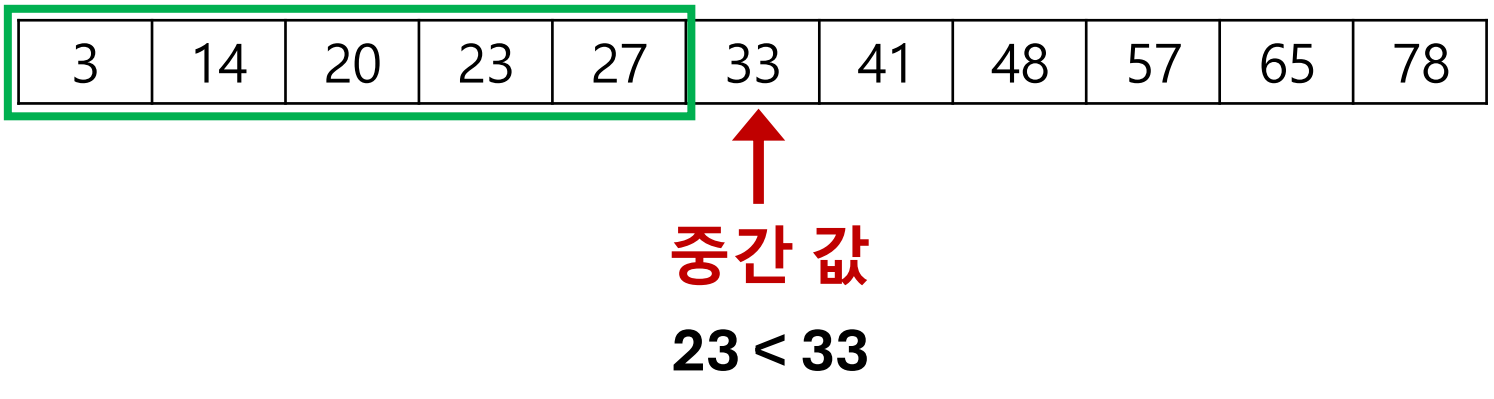
이진 검색(Binary Search)

- 정렬된 리스트에서 탐색 범위를 줄여 나가면서 검색 값을 찾는 알고리즘
  - **데이터가 정렬(오름차순 또는 내림차순)되어 있어야 함**
- 데이터 가운데에 위치한 항목을 검색 값과 비교
  - 검색 값이 더 크면 오른쪽 부분 검색
  - 검색 값이 더 작으면 왼쪽 부분 검색
- 원하는 값을 찾을 때까지 검색 범위를 줄여가며 반복
  - **검색 대상인 데이터 개수를 평균적으로 ½씩 줄임**
  - 데이터 양이 많을 때 활용하면 빠른 속도로 탐색 가능

이진 검색(Binary Search)

- Example

찾는 값: 23



이진 검색(Binary Search)

- Example

찾는 값: 23

이진 검색(Binary Search)

- Example

찾는 값: 23

중앙 위치?
- 홀수: 중앙
- 짝수: 왼쪽 공간에서 가장 큰 수

이진 검색(Binary Search)

- Example

찾는 값: 23

중앙 위치?
- 홀수: 중앙
- 짝수: 왼쪽 공간에서 가장 큰 수

이진 검색(Binary Search)

- **Time complexity**
  - 탐색 범위가 절반(1/2)씩 줄어들게 됨
  - n개 데이터가 1개가 남을때까지 반으로 나누는 횟수: $\log_2 n$
    → O(log n)

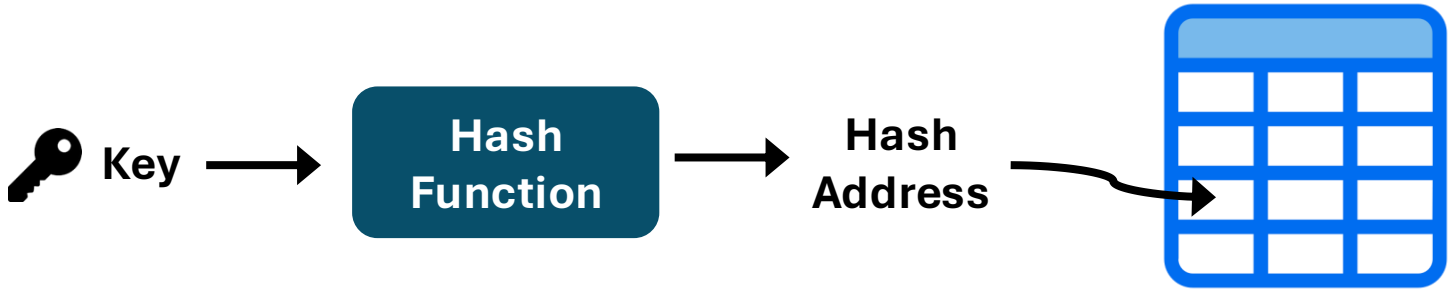
- **Space complexity**
  - O(1)

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] > target:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

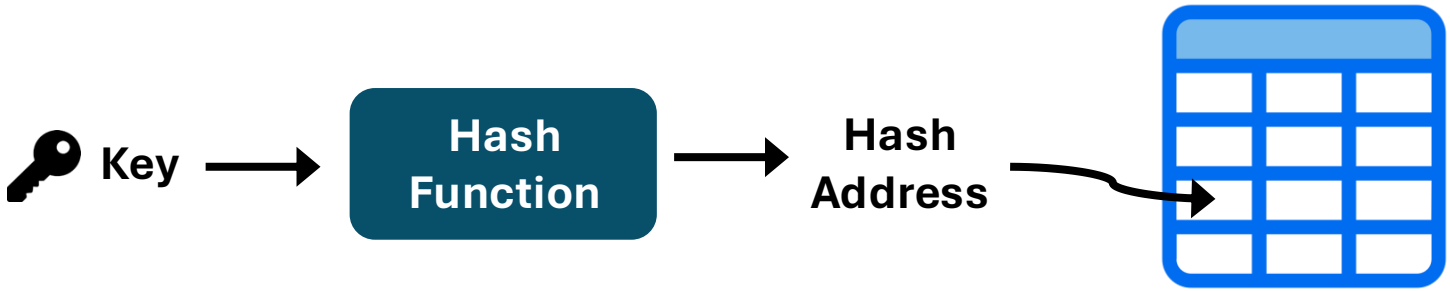
해싱(Hashing)

- 산술적인 연산을 이용해 키(Key)가 있는 위치를 계산하고, 그 위치를 찾는 방법
  - 데이터를 위치로 바로 매핑 - 특정 위치로 바로 이동
  - 키(key): 찾고자 하는 원래 값
- 해시 함수(Hash Function)
  - 키 값을 데이터 위치(ex. 인덱스, index)로 변환하는 함수
- 해시 테이블(Hash Table)
  - 해시 함수에 따라 계산 된 주소 위치에 데이터를 저장하거나 꺼낼 수 있는 테이블



해싱(Hashing)

- Example) 학번에 따라 학생 이름을 저장/검색



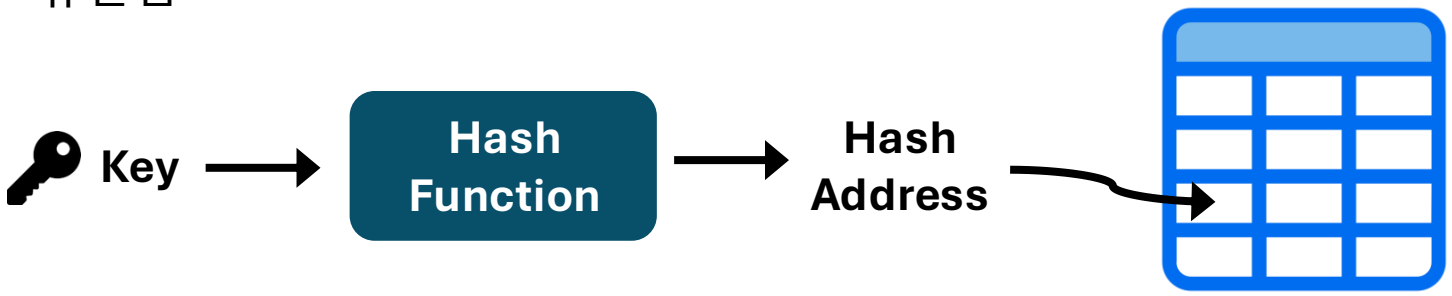
Student ID	Function	Address (202600001 % 100 =)
202600001	학번을 100으로 나눈 나머지 반환	1
202600012		12
202600053	ex) 123 % 100 = 23	53

Hash Address	이름
1	김OO
...	...
12	이OO
...	...
53	박OO

Time complexity? O(1)

해싱(Hashing)

- 해시 충돌(Hash Collision)
  - 데이터(Key)의 범위는 무한대 일 수 있지만 이를 저장하는 공간은 유한함



Student ID	Function	Student ID
202600001	학번을 100으로 나눈 나머지 반환	1
202600012		12
202600053		53
	ex) 123 % 100 = 23	
202601112		**12**

Hash Address	이름
1	김OO
...	...
12	이OO
...	...
53	박OO

Summary

- 알고리즘 복잡도
  - Time complexity
    - Big O Notation
  - Space complexity
- 정렬 알고리즘
  - Selection sort
  - Merge sort
- 검색 알고리즘
  - Sequential search
  - Binary search
  - Hashing