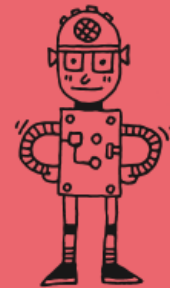


모두의 알고리즘 with 파이썬

컴퓨팅 사고를 위한 기초 알고리즘



셋째 마당 탐색과 정렬

문제07 순차 탐색

문제08 선택 정렬

문제09 삽입 정렬

문제10 병합 정렬

문제11 퀵 정렬

문제12 이분 탐색

문제 07 순차 탐색

주어진 리스트에 특정한 값이 있는지 찾아 그 위치를 돌려주는 알고리즘을 만들어 보세요.

리스트에 찾는 값이 없다면 -1을 돌려줍니다.

- 리스트에 있는 첫 번째 자료부터 하나씩 비교하면서 같은 값이 나오면 그 위치를 결과로 돌려줌
- 리스트 끝까지 찾아도 같은 값이 나오지 않으면 -1을 돌려주면 됨
- 순차 탐색(sequential search): 리스트 안에 있는 원소를 하나씩 순차적으로 비교하면서 탐색

1

순차 탐색으로 특정 값의 위치 찾기

- 순차 탐색 알고리즘을 이용하여 주어진 리스트 [17, 92, 18, 33, 58, 5, 33, 42]에서 특정 값(18, 33, 900)을 찾아서 해당 위치 번호를 돌려주는 프로그램
- 프로그램 7-1 순차 탐색 알고리즘

▼ 예제 소스 p07-1-search.py

```
# 리스트에서 특정 숫자 위치 찾기
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾으면 그 값의 위치, 찾지 못하면 -1

def search_list(a, x):
    n = len(a) # 입력 크기 n
```



순차 탐색으로 특정 값의 위치 찾기



```
for i in range(0, n): # 리스트 a의 모든 값을 차례로
    if x == a[i]:      # x 값과 비교하여
        return i      # 같으면 위치 돌려줍니다.

return -1 # 끝까지 비교해서 없으면 -1을 돌려줍니다.
```

```
v = [17, 92, 18, 33, 58, 7, 33, 42]
```

```
print(search_list(v, 18)) # 2(순서상 세 번째지만, 위치 번호는 2)
```

```
print(search_list(v, 33)) # 3(33은 리스트에 두 번 나오지만 처음 나온 위치만 출력)
```

```
print(search_list(v, 900)) # -1(900은 리스트에 없음)
```

1 순차 탐색으로 특정 값의 위치 찾기

- 실행 결과

```
2  
3  
-1
```

1 순차 탐색으로 특정 값의 위치 찾기

- 주어진 리스트 v에서 18을 순차 탐색으로 어떻게 찾을까?

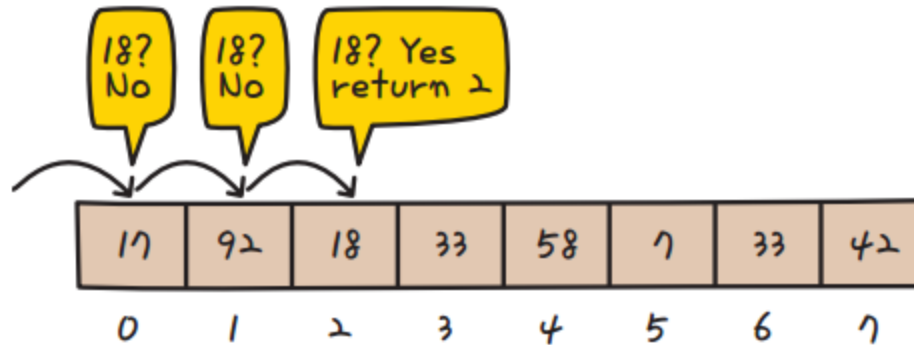


그림 7-1 리스트에서 18을 순차 탐색으로 찾는 과정

- 첫 번째 값(위치 번호는 0)인 17부터 차례로 비교하면서 18을 찾으면 해당 위치 번호인 2를 돌려줌(return i)

1 순차 탐색으로 특정 값의 위치 찾기

- 900과 같이 리스트에 없는 자료를 입력으로 넣을 경우 어떻게 될까?

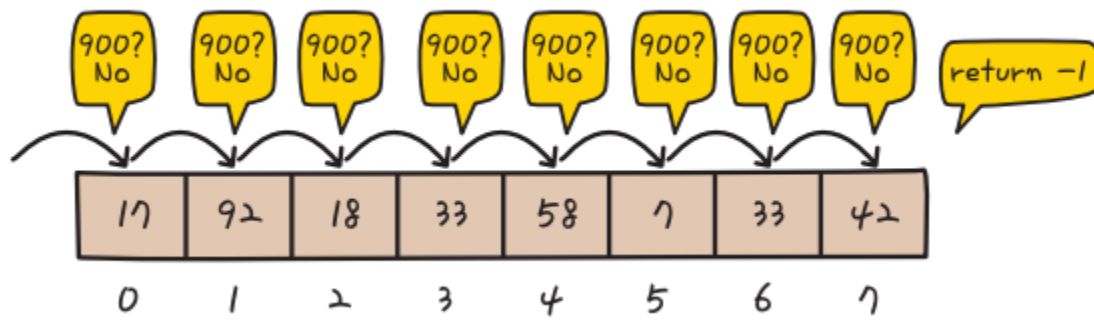


그림 7-2 리스트에서 900(없는 자료)을 순차 탐색으로 찾는 과정

- 리스트의 끝까지 차례로 비교해도 900과 같은 값이 없으므로 -1을 돌려줌 (return -1)

알고리즘 분석

- 순차 탐색 알고리즘으로 원하는 값을 찾으려면 비교를 몇 번 해야 할까?
- 경우에 따라 다름, 찾는 값이 리스트의 맨 앞에 있다면 단 한 번만 비교해도 결과를 얻을 수 있지만 찾는 값이 리스트의 마지막에 있거나 아예 없다면 리스트의 끝까지 하나하나 비교해야 함
- 경우에 따라 계산 횟수가 다를 때는 최선의 경우, 평균적인 경우, 최악의 경우로 나누어 각각 계산 복잡도를 생각해 보기도 함
- 최악의 경우를 분석하면 어떤 경우라도 그보다는 빨리 계산할 수 있을 것
- 따라서 보수적인 관점에서 이 알고리즘을 최악의 경우로 분석해 보면 비교가 최대 n 번 필요하고 계산 복잡도는 $O(n)$

연습 문제

- 7-1 프로그램 7-1은 리스트에 찾는 값이 여러 개 있더라도 첫 번째 위치만 결과로 돌려줍니다. 찾는 값이 나오는 모든 위치를 리스트로 돌려주는 탐색 알고리즘을 만들어 보세요. 찾는 값이 리스트에 없다면 빈 리스트인 []를 돌려줍니다.
- 7-2 연습 문제 7-1 프로그램의 계산 복잡도는 무엇일까요?

연습 문제

- 7-3 다음과 같이 학생 번호와 이름이 리스트로 주어졌을 때 학생 번호를 입력하면 학생 번호에 해당하는 이름을 순차 탐색으로 찾아 돌려주는 함수를 만들어 보세요. 해당하는 학생 번호가 없으면 물음표(?)를 돌려줍니다. 참고로 학생 번호가 39번이면 "Justin", 14번이면 "John"을 돌려줍니다.

```
stu_no = [39, 14, 67, 105]  
stu_name = ["Justin", "John", "Mike", "Summer"]
```

연습 문제 풀이

부록 A

■ 7-1 리스트에서 특정 숫자의 위치를 전부 찾기

▼ 예제 소스 e07-1-searchall.py

```
# 리스트에서 특정 숫자 위치 전부 찾기
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾는 값의 위치 번호가 담긴 리스트, 찾는 값이 없으면 빈 리스트 []

def search_list(a, x):
    n = len(a) # 입력 크기 n
    result = [] # 새 리스트를 만들어 결과값을 저장
    for i in range(0, n): # 리스트 a의 모든 값을 차례로
        if x == a[i]: # x값과 비교하여
            result.append(i) # 같으면 위치 번호를 결과 리스트에 추가

    return result # 만들어진 결과 리스트를 돌려줌
```



연습 문제 풀이

부록 A



```
v = [17, 92, 18, 33, 58, 7, 33, 42]
print(search_list(v, 18)) # [2] (순서상 세 번째지만, 위치 번호는 2)
print(search_list(v, 33)) # [3, 6] (33은 리스트에 두 번 나옴)
print(search_list(v, 900)) # [] (900은 리스트에 없음)
```

■ 실행 결과

```
[2]
[3, 6]
[]
```

연습 문제 풀이

부록 A

- 7-2 프로그램 7-1의 계산 복잡도

$O(n)$

- 연습 문제 7-1 프로그램은 찾는 값이 탐색 중간에 나오더라도 탐색을 멈추지 않고 혹시 더 있을 자료 값을 찾기 위해 끝까지 탐색을 해야 함
- 따라서 어떤 경우에도 비교가 n 번 필요

연습 문제 풀이

부록 A

7-3 학생 번호에 해당하는 학생 이름 찾기

▼ 예제 소스 e07-3-getname.py

```
# 학생 번호에 해당하는 학생 이름 찾기
# 입력: 학생 번호 리스트 s_no, 학생 이름 리스트 s_name, 찾는 학생 번호 find_no
# 출력: 해당하는 학생 이름, 학생 이름이 없으면 물음표 "?"

def get_name(s_no, s_name, find_no):
    n = len(s_no) # 입력 크기 n
    for i in range(0, n):
        if find_no == s_no[i]: # 학생 번호가 찾는 학생 번호와 같으면
            return s_name[i] # 해당하는 학생 이름을 결과로 반환

    return "?" # 자료를 다 뒤져서 못 찾았으면 물음표 반환
```



연습 문제 풀이

부록 A



```
sample_no = [39, 14, 67, 105]
sample_name = ["Justin", "John", "Mike", "Summer"]
print(get_name(sample_no, sample_name, 105))
print(get_name(sample_no, sample_name, 777))
```

■ 실행 결과

```
Summer
?
```

문제 08 선택 정렬

주어진 리스트 안의 자료를 작은 수부터 큰 수 순서로 배열하는 정렬 알고리즘을 만들어 보세요.

- 정렬(sort): 자료를 크기 순서대로 맞춰 일렬로 나열하는 것

리스트에 들어 있는 숫자를 크기 순으로 나열하는 정렬 알고리즘의 입출력

- 문제: 리스트 안에 있는 자료를 순서대로 배열하기
- 입력: 정렬할 리스트(예: [35, 9, 2, 85, 17])
- 출력: 순서대로 정렬된 리스트(예: [2, 9, 17, 35, 85])

1 선택 정렬로 줄 세우기

- 운동장에 모인 학생을 키 순서에 맞춰 일렬로 줄 세우는 방법
 - 1) 학생 열 명이 모여 있는 운동장에 선생님이 등장
 - 2) 선생님은 학생들을 둘러보며 키가 가장 작은 사람을 찾음. 키가 가장 작은 학생으로 '선택'된 민준이가 불러 나와 맨 앞에 섬. 민준이가 나갔으므로 이제 학생은 아홉 명 남았음
 - 3) 이번에는 선생님이 학생 아홉 명 중 키가 가장 작은 성진이를 선택. 선택된 성진이가 불러 나와 민준이 뒤로 줄을 섬
 - 4) 이처럼 남아 있는 학생 중에서 키가 가장 작은 학생을 한 명씩 뽑아 줄에 세우는 과정을 반복하면 모든 학생이 키 순서에 맞춰 줄을 서게 됨

1 선택 정렬로 줄 세우기

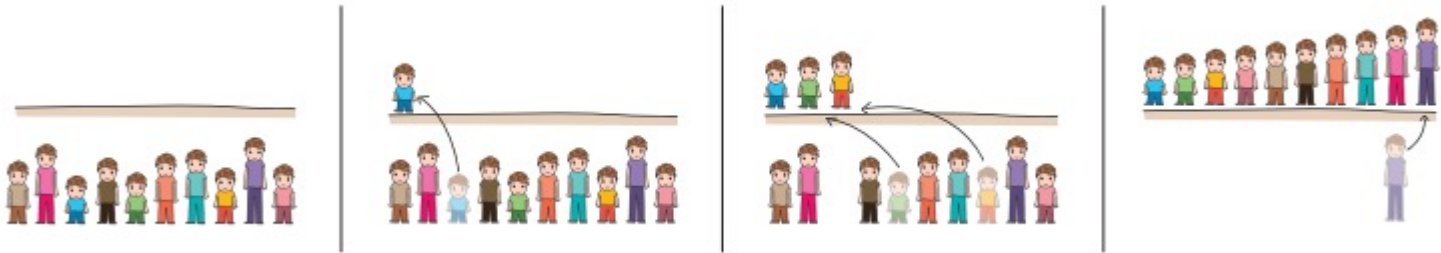


그림 8-1 선택 정렬

- '키 순서로 줄 세우기'는 대표적인 정렬 문제의 예
→ '학생의 키라는 자료 값을 작은 것부터 큰 순서로 나열하라'
- 쉽게 설명한 정렬 알고리즘: 정렬 원리를 이해하기 위한 참고용 프로그램
- 일반적인 정렬 알고리즘: 정렬 알고리즘을 정식으로 구현한 프로그램

2 쉽게 설명한 선택 정렬 알고리즘

■ 프로그램 8-1 쉽게 설명한 선택 정렬 알고리즘

▼ 예제 소스 p08-1-ssort.py

```
# 입력: 리스트 a,  
# 출력: 정렬된 새 리스트  
# 주어진 리스트에서 최솟값의 위치를 돌려주는 함수  
def find_min_idx(a):  
    n = len(a)  
    min_idx = 0  
    for i in range(1, n):  
        if a[i] < a[min_idx]:  
            min_idx = i  
    return min_idx  
  
def sel_sort(a):  
    result = [] # 새 리스트를 만들어 정렬된 값을 저장  
    while a: # 주어진 리스트에 값이 남아있는 동안 계속  
        min_idx = find_min_idx(a) # 리스트에 남아 있는 값 중 최솟값의 위치  
        value = a.pop(min_idx) # 찾은 최솟값을 빼내어 value에 저장  
        result.append(value) # value를 결과 리스트 끝에 추가  
    return result  
  
d = [2, 4, 5, 1, 3]  
print(sel_sort(d))
```

쉽게 설명한 선택 정렬 알고리즘

- 프로그램을 차근차근 읽어 보면 앞에서 설명한 줄 서기 원리가 잘 녹아 있음
- 1) 리스트 a에 아직 자료가 남아 있다면 → while a:
- 2) 남은 자료 중에서 최솟값의 위치를 찾음 → min_idx = find_min_idx(a)
- 3) 찾은 최솟값을 리스트 a에서 빼내어 value에 저장 → value = a.pop(min_idx)
- 4) Value를 result 리스트의 맨 끝에 추가 → result.append(value)
- 5) 1번 과정으로 돌아가 자료가 없을 때까지 반복

쉽게 설명한 선택 정렬 알고리즘

- 입력으로 주어진 리스트 [2, 4, 5, 1, 3]을 정렬하는 과정

① 시작

$\alpha = [2\ 4\ 5\ 1\ 3] \rightarrow$ 뽕표 생각

$result = []$

② α 리스트의 최솟값인 1을 α 에서 빼내어 $result$ 에 추가합니다.

$\alpha = [2\ 4\ 5\ 3]$

$result = [1]$

③ α 에 남아 있는 값 중 최솟값인 2를 α 에서 빼내어 $result$ 에 추가합니다.

$\alpha = [4\ 5\ 3]$

$result = [1\ 2]$

쉽게 설명한 선택 정렬 알고리즘

④ a에 남아 있는 값 중 최솟값인 3을 같은 방법으로 옮깁니다.

a = [4 5]

result = [1 2 3]

⑤ a에 남아 있는 값 중 최솟값인 4를 같은 방법으로 옮깁니다.

a = [5]

result = [1 2 3 4]

⑥ a에 남아 있는 값 중 최솟값인 5를 같은 방법으로 옮깁니다.

a = []

result = [1 2 3 4 5]

⑦ a가 비어 있으므로 종료합니다.

result = [1 2 3 4 5] → 최종 결과

3 일반적인 선택 정렬 알고리즘

- '일반적인 선택 정렬 알고리즘'은 입력으로 주어진 리스트 a 안에서 직접 자료의 위치를 바꾸면서 정렬시키는 프로그램
- 리스트 a에서 자료를 하나씩 빼낸 후 다시 result에 넣는 방식인 '쉽게 설명한 선택 정렬 알고리즘'보다 더 효율적으로 정렬할 수 있음

3 일반적인 선택 정렬 알고리즘

■ 프로그램 8-2 일반적인 선택 정렬 알고리즘

▼ 예제 소스 p08-2-ssort.py

```
# 선택 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def sel_sort(a):
    n = len(a)
    for i in range(0, n - 1): # 0부터 n-2까지 반복
        # i번 위치부터 끝까지 자료 값 중 최솟값의 위치를 찾음
        min_idx = i
        for j in range(i + 1, n):
            if a[j] < a[min_idx]:
                min_idx = j
        # 찾은 최솟값을 i번 위치로
        a[i], a[min_idx] = a[min_idx], a[i]

d = [2, 4, 5, 1, 3]
sel_sort(d)
print(d)
```

■ 실행 결과

[1, 2, 3, 4, 5]

3 일반적인 선택 정렬 알고리즘

파이썬에서 두 자료 값 서로 바꾸기

- 리스트 안에서 두 자료 값의 위치를 서로 바꾸는 데 다음과 같은 문장 사용

```
a[i], a[min_idx] = a[min_idx], a[i]
```

- 파이썬에서 두 변수의 값을 서로 바꾸려면 다음과 같이 쉼표를 이용해 변수를 뒤집어 표현 $\rightarrow x, y = y, x$

```
>>> x = 1
>>> y = 2
>>> x, y = y, x
>>> x
2
>>> y
1
```

알고리즘 분석

- 자료를 크기 순서로 정렬하려면 반드시 두 수의 크기를 비교해야 함
- 따라서 정렬 알고리즘의 계산 복잡도는 보통 비교 횟수를 기준으로 따짐
- 선택 정렬의 비교 방법은 문제 3의 동명이인 찾기에서 살펴본, 리스트 안의 자료를 한 번씩 비교하는 방법과 거의 같음
- 이 알고리즘은 비교를 총 $\frac{n(n-1)}{2}$ 번 해야 하며 계산 복잡도가 $O(n^2)$
- 비교 횟수가 입력 크기의 제곱에 비례
- 입력 크기가 커지면 커질수록 정렬하는 데 시간이 굉장히 오래 걸림

연습 문제

- 8-1 일반적인 선택 정렬 알고리즘을 사용해서 리스트 [2, 4, 5, 1, 3]을 정렬하는 과정을 적어 보세요.
- 8-2 프로그램 8-1과 8-2의 정렬 알고리즘은 숫자를 작은 수에서 큰 수 순서로 나열하는 오름차순 정렬이었습니다. 이 알고리즘을 큰 수에서 작은 수 순서로 나열하는 내림차순 정렬로 바꾸려면 프로그램의 어느 부분을 바꿔야 할까요?

오름차순과 내림차순 정렬 예

- 오름차순 정렬의 예: 가나다순, 쇼핑몰에서 낮은 가격순으로 보기
- 내림차순 정렬의 예: 시험 점수로 등수 구하기, 최신 뉴스부터 보기

연습 문제 풀이

부록 A

- 8-1 선택 정렬 과정
- 일반적인 선택 정렬은 처리할 대상 범위에서 최솟값을 찾아 그 값과 범위의 맨 앞에 있는 값을 서로 바꾸는 과정을 반복
- 이 과정이 한 번 끝날 때마다 범위 안의 맨 앞에 있는 값은 정렬이 끝난 것이므로 정렬 대상 범위에서 제외
- 이미 정렬이 끝난 부분과 앞으로 처리될 대상 범위 사이에 세로 선(|)을 넣어 구분

연습 문제 풀이

부록 A

| 2 4 5 1 3 ← 시작, 전체 리스트인 2, 4, 5, 1, 3을 대상으로 최솟값을 찾습니다.

| 1 4 5 2 3 ← 최솟값 1을 대상의 가장 왼쪽 값인 2와 바꿉니다.

1 | 4 5 2 3 ← 1을 대상에서 제외하고 4, 5, 2, 3에서 최솟값을 찾습니다.

1 | 2 5 4 3 ← 4, 5, 2, 3 중 최솟값인 2를 4와 바꿉니다.

1 2 | 5 4 3 ← 2를 대상에서 제외하고 5, 4, 3에서 최솟값을 찾습니다.

1 2 | 3 4 5 ← 5, 4, 3 중 최솟값인 3을 5와 바꿉니다.

1 2 3 | 4 5 ← 3을 대상에서 제외하고 4, 5에서 최솟값을 찾습니다.

1 2 3 | 4 5 ← 최솟값 4를 4와 바꿉니다(변화 없음).

1 2 3 4 | 5 ← 4를 대상에서 제외합니다. 자료가 5 하나만 남았으므로 종료합니다.

1 2 3 4 5 | ← 최종 결과

연습 문제 풀이

부록 A

정렬 중간 결과 출력하기

- 함수 반복 부분에 `print(a)` 추가 → 정렬 과정의 중간 결과 쉽게 확인 가능

```
def sel_sort(a):  
    n = len(a)  
    for i in range(0, n - 1):  
        min_idx = i  
        for j in range(i + 1, n):  
            if a[j] < a[min_idx]:  
                min_idx = j  
            a[i], a[min_idx] = a[min_idx], a[i]  
            print(a) # 정렬 과정 출력하기  
  
d = [2, 4, 5, 1, 3]  
sel_sort(d)  
print(d)
```

연습 문제 풀이

부록 A

- 8-2 내림차순 선택 정렬
- 오름차순 선택 정렬에서 최솟값 대신 최댓값을 선택하면 내림차순 정렬 (큰 수에서 작은 수로 나열)이 됨
- 비교 부등호 방향을 작다(<)에서 크다(>)로 바꾸기만 해도 내림차순 정렬 프로그램이 됨
- 변수 이름의 의미를 맞추려고 변수 `min_idx` 를 `max_idx`로 바꿈

연습 문제 풀이

부록 A

▼ 예제 소스 e08-2-ssort.py

```
# 내림차순 선택 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def sel_sort(a):
    n = len(a)
    for i in range(0, n - 1):
        max_idx = i # 최솟값(min) 대신 최댓값(max)을 찾아야 함
        for j in range(i + 1, n):
            if a[j] > a[max_idx]: # 부등호 방향 뒤집기
                max_idx = j
        a[i], a[max_idx] = a[max_idx], a[i]

d = [2, 4, 5, 1, 3]
sel_sort(d)
print(d)
```



연습 문제 풀이

부록 A

- 실행 결과

[5, 4, 3, 2, 1]

문제 09 삽입 정렬

리스트 안의 자료를 작은 수부터 큰 수 순서로 배열하는 정렬 알고리즘을 만들어 보세요.

- 문제 8과 같지만 이번에는 삽입 정렬(Insertion sort)으로 문제를 풀어 보자

1 삽입 정렬로 줄 세우기

- 문제를 풀기 전에 줄 서기부터 시작해 보자
- 1) 학생이 열 명 모인 운동장에 선생님이 등장
- 2) 선생님은 열 명 중 제일 앞에 있던 승규에게 나와서 줄을 서라고 함
승규가 나갔으니 이제 학생이 아홉 명 남음
- 3) 이번에는 선생님이 준호에게 키를 맞춰 줄을 서라고 함
준호는 이미 줄 선 승규보다 자신이 키가 작은 것을 확인하고 승규 앞에 섬
- 4) 남은 여덟 명 중 이번에는 민성이가 뽀빠 줄을 섬
민성이는 준호보다 크고 승규보다는 작으므로 준호와 승규 사이에 공간을 만들어 줄을 섬(삽입)

1 삽입 정렬로 줄 세우기

- 5) 마찬가지로 남은 학생을 한 명씩 뽑아 이미 줄을 선 학생 사이사이에 키를 맞춰 끼워 넣는 일을 반복
- 마지막 남은 학생까지 뽑아서 줄을 세우면 모든 학생이 제자리에 줄 서게 됨

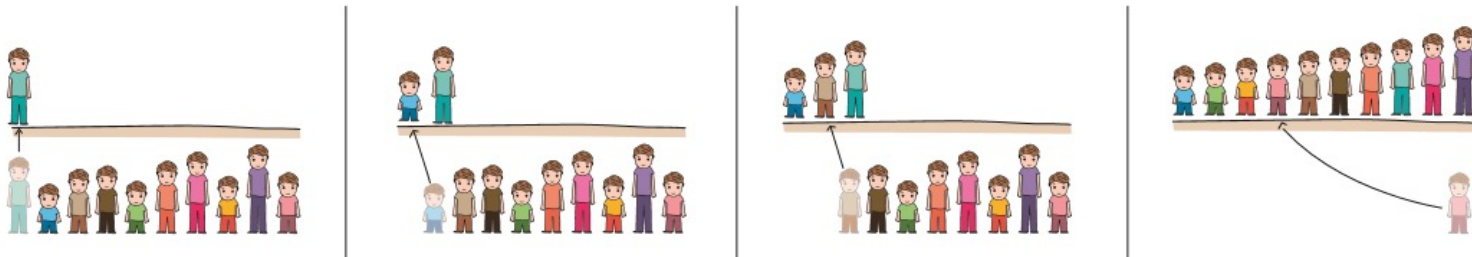


그림 9-1 삽입 정렬

쉽게 설명한 삽입 정렬 알고리즘

■ 프로그램 9-1 쉽게 설명한 삽입 알고리즘

▼ 예제 소스 p09-1-isort.py

```
# 쉽게 설명한 삽입 정렬
# 입력: 리스트 a
# 출력: 정렬된 새 리스트

# 리스트 r에서 v가 들어가야 할 위치를 돌려주는 함수
def find_ins_idx(r, v):
    # 이미 정렬된 리스트 r의 자료를 앞에서부터 차례로 확인하여
    for i in range(0, len(r)):
        # v 값보다 i번 위치에 있는 자료 값이 크면
        # v가 그 값 바로 앞에 놓여야 정렬 순서가 유지됨
        if v < r[i]:
            return i
```



쉽게 설명한 삽입 정렬 알고리즘



```
# 적절한 위치를 못 찾았을 때는  
# v가 r의 모든 자료보다 크다는 뜻이므로 맨 뒤에 삽입  
return len(r)
```

```
def ins_sort(a):  
    result = [] # 새 리스트를 만들어 정렬된 값을 저장  
    while a:   # 기존 리스트에 값이 남아 있는 동안 반복  
        value = a.pop(0) # 기존 리스트에서 한 개를 꺼냄  
        ins_idx = find_ins_idx(result, value) # 꺼낸 값이 들어갈 적당한 위치 찾기  
        result.insert(ins_idx, value) # 찾은 위치에 값 삽입(이후 값은 한 칸씩 밀려남)  
    return result
```

```
d = [2, 4, 5, 1, 3]  
print(ins_sort(d))
```

쉽게 설명한 삽입 정렬 알고리즘

- 실행 결과

```
[1, 2, 3, 4, 5]
```

쉽게 설명한 삽입 정렬 알고리즘

- 프로그램이 동작하는 원리

- 1) 리스트 a에 아직 자료가 남아 있다면 → while a:
- 2) 남은 자료 중에 맨 앞의 값을 뽑아냄 → `value = a.pop(0)`
- 3) 그 값이 result의 어느 위치에 들어가면 적당할지 알아냄
→ `ins_idx = find_ins_idx(result, value)`
- 4) 3번 과정에서 찾아낸 위치에 뽑아낸 값을 삽입
→ `result.insert(ins_idx, value)`
- 5) 1번 과정으로 돌아가 자료가 없을 때까지 반복

쉽게 설명한 삽입 정렬 알고리즘

- 입력으로 주어진 리스트 [2, 4, 5, 1, 3]이 정렬되는 과정

① 시작

$a = [2\ 4\ 5\ 1\ 3]$

$result = []$

② a 에서 2째 빼서 현재 비어 있는 $result$ 에 넣습니다.

$a = [4\ 5\ 1\ 3]$

$result = [2]$

③ a 에서 4째 빼서 $result$ 의 2 뒤에 넣습니다($2 < 4$).

$a = [5\ 1\ 3]$

$result = [2\ 4]$

2 쉽게 설명한 삽입 정렬 알고리즘

④ a에서 5를 빼서 result의 맨 뒤에 넣습니다($4 < 5$).

a = [1 3]

result = [2 4 5]

⑤ a에서 1을 빼서 result의 맨 앞에 넣습니다($1 < 2$).

a = [3]

result = [1 2 4 5]

⑥ a에서 마지막 값인 3을 빼서 result의 2와 4 사이에 넣습니다($2 < 3 < 4$).

a = []

result = [1 2 3 4 5]

⑦ a가 비어 있으므로 종료합니다.

result = [1 2 3 4 5] → 최종 결과

쉽게 설명한 삽입 정렬 알고리즘

- 프로그램 중간에 print 문을 적절히 추가하면 알고리즘이 진행되는 과정을 확인할 수 있어 알고리즘의 동작 원리를 파악하는 데 큰 도움이 됨
- ins_sort() 함수의 result.insert(ins_idx, value) 바로 다음 줄에 print(a, result)를 추가하면 다음과 같은 결과를 얻을 수 있음

```
[4, 5, 1, 3] [2]
```

```
[5, 1, 3] [2, 4]
```

```
[1, 3] [2, 4, 5]
```

```
[3] [1, 2, 4, 5]
```

```
[] [1, 2, 3, 4, 5]
```

3 일반적인 삽입 정렬 알고리즘

■ 프로그램 9-2 일반적인 삽입 알고리즘

▼ 예제 소스 p09-2-isort.py

```
# 삽입 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def ins_sort(a):
    n = len(a)
    for i in range(1, n): # 1부터 n-1까지
        # i번 위치의 값을 key로 저장
        key = a[i]
        # j를 i 바로 왼쪽 위치로 저장
        j = i - 1
```



3 일반적인 삽입 정렬 알고리즘



```
# 리스트의 j번 위치에 있는 값과 key를 비교해 key가 삽입될 적절한 위치를 찾음
while j >= 0 and a[j] > key:
    a[j + 1] = a[j] # 삽입할 공간이 생기도록 값을 오른쪽으로 한 칸 이동
    j -= 1
a[j + 1] = key # 찾은 삽입 위치에 key를 저장
```

```
d = [2, 4, 5, 1, 3]
ins_sort(d)
print(d)
```

■ 실행 결과

```
[1, 2, 3, 4, 5]
```

알고리즘 분석

- 삽입 정렬 알고리즘의 입력으로 이미 정렬이 끝난 리스트, 예를 들어 [1, 2, 3, 4, 5]와 같은 리스트를 넣어 주면 $O(n)$ 의 계산 복잡도로 정렬을 마칠 수 있음
→ 이런 경우는 특별한 경우
- 일반적인 입력일 때 삽입 정렬의 계산 복잡도는 선택 정렬과 같은 $O(n^2)$
- 따라서 선택 정렬과 마찬가지로 정렬할 입력 크기가 크면 정렬하는 데 시간이 굉장히 오래 걸림

연습 문제

- 9-1 일반적인 삽입 정렬 알고리즘을 사용해서 리스트 [2, 4, 5, 1, 3]을 정렬하는 과정을 적어 보세요.
- 9-2 문제 9에서 설명한 정렬 알고리즘은 숫자를 작은 수에서 큰 수 순서로 나열하는 오름차순 정렬이었습니다. 이를 큰 수에서 작은 수 순서로 나열하는 내림차순 정렬로 바꾸려면 프로그램의 어느 부분을 바꿔야 할까요?

연습 문제 풀이

부록 A

- 9-1 삽입 정렬 과정
- 일반적인 삽입 정렬은 처리할 대상 범위에 있는 맨 앞 값을 적절한 위치에 넣는 과정을 반복
- 이 과정이 한 번 끝날 때마다 대상 범위에 있는 맨 앞의 값이 제 위치를 찾아 가므로 정렬 대상 범위는 하나씩 줄어듦
- 이미 정렬이 끝난 부분과 앞으로 처리될 대상 범위 사이에 세로 선(|)을 넣어 구분

연습 문제 풀이

부록 A

| 2 4 5 1 3 ← 시작

2 | 4 5 1 3 ← 맨 앞에 있는 2는 옮기지 않아도 됩니다.

2 | 4 5 1 3 ← 4의 위치를 맞춥니다. 2 바로 다음이므로 위치가 변하지 않습니다.

2 4 | 5 1 3 ← 대상 범위를 하나 줄입니다.

2 4 | 5 1 3 ← 5의 위치를 맞춥니다. 4 바로 다음이므로 이번에도 위치가 그대로입니다.

2 4 5 | 1 3 ← 대상 범위를 하나 줄입니다.

1 2 4 | 5 3 ← 1의 위치를 맞춥니다. 1은 2, 4, 5보다 작으므로 이 값들을 한 칸씩 오른쪽으로 옮긴 다음
비어 있는 공간에 1을 넣습니다.

1 2 4 5 | 3 ← 대상 범위를 하나 줄입니다.

1 2 3 4 | 5 ← 마지막으로 3의 위치를 맞춥니다. 3은 4, 5보다 작으므로 4와 5를 한 칸씩 오른쪽으로 옮긴 다음
비어 있는 공간에 3을 넣습니다.

1 2 3 4 5 | ← 대상 범위를 하나 줄입니다. 더는 자료가 없으므로 종료합니다(최종 결과).

연습 문제 풀이

부록 A

- 9-2 내림차순 삽입 정렬
- 오름차순 정렬에서 키(key)를 비교하는 부분($a[j] > \text{key}$)의 부등호를 반대로 하면 내림차순 정렬 프로그램이 됨

▼ 예제 소스 e09-2-isort.py

```
# 내림차순 삽입 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def ins_sort(a):
    n = len(a)
```



연습 문제 풀이

부록 A



```
for i in range(1, n):  
    key = a[i]  
    j = i - 1  
    while j >= 0 and a[j] < key: # 부등호 방향 뒤집기  
        a[j + 1] = a[j]  
        j -= 1  
    a[j + 1] = key
```

```
d = [2, 4, 5, 1, 3]
```

```
ins_sort(d)
```

```
print(d)
```



연습 문제 풀이

부록 A

- 실행 결과

[5, 4, 3, 2, 1]

문제 10 병합 정렬

리스트 안의 자료를 작은 수부터 큰 수 순서로 배열하는 정렬 알고리즘을 만들어 보세요.

- 재귀 호출을 사용해 정렬 문제를 더 빠르게 풀어 보자

1 병합 정렬로 줄 세우기

- 줄 세우기를 통해 병합 정렬(Merge sort) 과정을 생각해 보자
- 1) 학생들에게 일일이 지시하는 것이 귀찮아진 선생님은 학생들이 알아서 줄을 설 수 있는 방법이 없을지 고민. 열 명이나 되는 학생들에게 동시에 알아서 줄을 서라고 하면 너무 소란스러울 것 같아서, 다섯 명씩 두 조로 나누어 그 안에서 키 순서로 줄을 서라고 시킴
- 2) 이제 선생님 앞에는 키 순서대로 정렬된 두 줄(중간 결과 줄)이 있음
- 3) 선생님은 각 줄의 맨 앞에 있는 두 학생 중에 키가 더 작은 민수를 뽑아 최종 결과 줄에 세움. 그리고 다시 각 중간 결과 줄의 맨 앞에 있는 두 학생을 비교해 더 작은 학생을 최종 결과 줄의 민수 뒤에 세움

1 병합 정렬로 줄 세우기

4) 이 과정을 반복하다가 중간 결과 줄 하나가 사라지면 나머지 중간 결과 줄에 있는 사람을 전부 최종 결과 줄에 세움

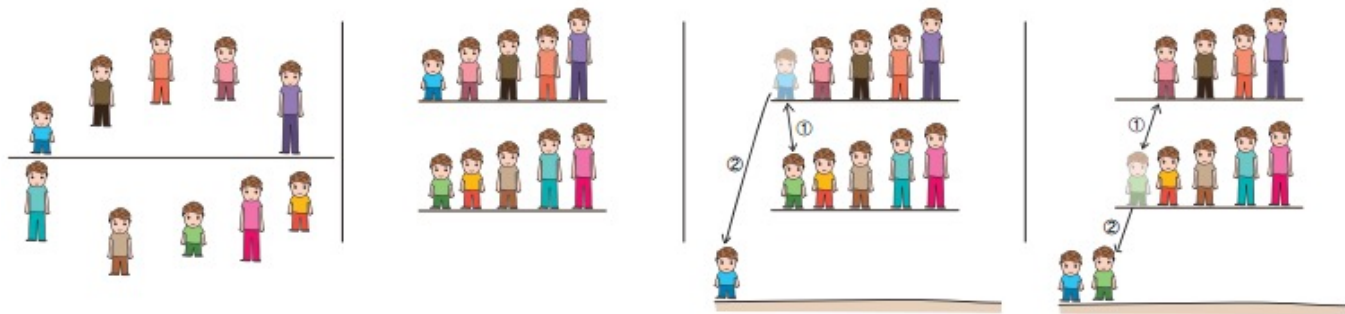


그림 10-1 병합 정렬

쉽게 설명한 병합 정렬 알고리즘

■ 프로그램 10-1 쉽게 설명한 병합 정렬 알고리즘

▼ 예제 소스 p10-1-msort.py

```
# 쉽게 설명한 병합 정렬
# 입력: 리스트 a
# 출력: 정렬된 새 리스트

def merge_sort(a):
    n = len(a)
    # 종료 조건: 정렬할 리스트의 자료 개수가 한 개 이하이면 정렬할 필요 없음
    if n <= 1:
        return a
```



쉽게 설명한 병합 정렬 알고리즘



```
# 그룹을 나누어 각각 병합 정렬을 호출하는 과정
mid = n // 2 # 중간을 기준으로 두 그룹으로 나눔
g1 = merge_sort(a[:mid]) # 재귀 호출로 첫 번째 그룹을 정렬
g2 = merge_sort(a[mid:]) # 재귀 호출로 두 번째 그룹을 정렬
# 두 그룹을 하나로 병합
result = [] # 두 그룹을 합쳐 만들 최종 결과
while g1 and g2: # 두 그룹에 모두 자료가 남아 있는 동안 반복
    if g1[0] < g2[0]: # 두 그룹의 맨 앞 자료 값을 비교
        # g1의 값이 더 작으면 그 값을 빼내어 결과로 추가
        result.append(g1.pop(0))
    else:
        # g2의 값이 더 작으면 그 값을 빼내어 결과로 추가
        result.append(g2.pop(0))
```



쉽게 설명한 병합 정렬 알고리즘



```
# 아직 남아 있는 자료들을 결과에 추가
# g1과 g2 중 이미 빈 것은 while을 바로 지나감
while g1:
    result.append(g1.pop(0))
while g2:
    result.append(g2.pop(0))
return result
```

```
d = [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]
print(merge_sort(d))
```

■ 실행 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

쉽게 설명한 병합 정렬 알고리즘

- 병합 정렬 함수의 첫 부분이 바로 종료 조건

```
n = len(a)
if n <= 1:
    return a
```

- 입력으로 주어진 리스트 a의 크기가 1 이하이면, 즉 자료가 한 개뿐이거나
아예 비어 있다면 정렬할 필요가 없음
→ 입력 리스트를 그대로 돌려주면서 재귀 호출을 끝냄

쉽게 설명한 병합 정렬 알고리즘

- 전체 리스트를 절반으로 나눠 각각 재귀 호출로 병합 정렬하는 부분

```
mid = n // 2 # 중간을 기준으로 두 그룹으로 나눔  
g1 = merge_sort(a[:mid]) # 재귀 호출로 첫 번째 그룹을 정렬  
g2 = merge_sort(a[mid:]) # 재귀 호출로 두 번째 그룹을 정렬
```

- 리스트의 자료 개수가 홀수일 때는 어떻게 절반으로 나눌까?
- $n // 2$ 는 리스트의 길이 n 을 2로 나눈 몫
→ n 이 5와 같은 홀수라면 $n // 2$ 는 2가 됨
- 즉, 자료가 두 개인 그룹과 세 개인 그룹으로 나눔

쉽게 설명한 병합 정렬 알고리즘

- `a[:mid]`는 리스트 `a`의 0번 위치부터 `mid` 위치 직전까지의 자료를 복사해서 새 리스트를 만드는 문장
- `a[mid:]`는 리스트 `a`의 `mid` 위치부터 끝까지의 자료를 복사해서 새 리스트를 만드는 문장

```
>>> a = [1, 2, 3, 4, 5]
>>> mid = len(a) // 2
>>> mid
2
>>> a[:mid]
[1, 2]
>>> a[mid:]
[3, 4, 5]
```

쉽게 설명한 병합 정렬 알고리즘

■ 리스트 [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]를 병합 정렬하는 과정

① 숫자 열 개를 두 그룹($g1$, $g2$)으로 나눕니다.

$g1$: [6 8 3 9 10]

$g2$: [1 2 4 7 5]

② 두 그룹을 각각 정렬합니다(재귀 호출 부분이므로 이 부분은 뒤에서 설명합니다. 일단 각 그룹을 정렬해 봅니다).

$g1$: [3 6 8 9 10]

$g2$: [1 2 4 5 7]

③ 이제 두 그룹을 합쳐 다시 한 그룹으로 만들겠습니다(병합).

두 그룹의 첫 번째 값을 비교하여 작은 값을 빼내 결과 리스트에 넣습니다. $g1$ 의 첫 번째 값은 3, $g2$ 의 첫 번째 값은 1이므로 1을 빼내 결과 리스트(result)에 넣습니다.

$g1$: [3 6 8 9 10]

$g2$: [2 4 5 7]

result: [1]

쉽게 설명한 병합 정렬 알고리즘

④ 두 그룹의 첫 번째 값을 비교하여 작은 값을 배내 결과 리스트에 넣는 과정을 반복합니다. 이번에는 g2의 2가 뽑혀 정렬됩니다.

g1: [3 6 8 9 10]

g2: [4 5 7]

result: [1 2]

⑤ 이번에는 g1의 3이 뽑혀 정렬됩니다.

g1: [6 8 9 10]

g2: [4 5 7]

result: [1 2 3]

⑥ 이 과정을 반복하면 다음과 같이 한 그룹의 자료가 다 빠져나가 비어 있게 됩니다.

g1: [8 9 10]

g2: []

result: [1 2 3 4 5 6 7]

쉽게 설명한 병합 정렬 알고리즘

⑦ g2에는 자료가 없으므로 비교할 필요 없이 g1에 남아 있는 값을 전부 result로 옮기면 정렬이 끝납니다.

g1: []

g2: []

result: [1 2 3 4 5 6 7 8 9 10]

이 방법을 병합 정렬이라 부르는 이유

- 이미 정렬된 두 그룹을 맨 앞에서부터 비교하면서 하나로 합치는 '병합 (merge)' 과정이 정렬 알고리즘의 핵심이기 때문(③~⑦번 과정)

3 병합 정렬에서의 재귀 호출

- 병합 정렬에서 ②번 과정을 보면 두 그룹으로 나눈 자료를 각각 정렬함
- 나누어진 그룹은 어떤 정렬 알고리즘으로 정렬하는 걸까? → 병합 정렬
- 병합 정렬을 하는 과정에서 나누어진 리스트를 다시 두 번의 병합 정렬로 정렬하는 것
- 이는 문제 8에서 살펴본, 원반이 n 개인 하노이의 탑 문제를 풀기 위해 원반이 $n-1$ 개인 하노이의 탑 문제를 재귀 호출하는 것과 비슷함
- 어떤 문제를 푸는 과정 안에서 다시 그 문제를 푸는 것이 바로 재귀 호출

3 병합 정렬에서의 재귀 호출

재귀 호출의 세 가지 요건

- 1) 함수 안에서 자기 자신을 다시 호출함
- 2) 재귀 호출할 때 인자로 주어지는 입력 크기가 작아짐
- 3) 특정 종료 조건이 만족되면 재귀 호출을 종료함

- 병합 정렬은 자료 열 개를 정렬하기 위해 자료를 다섯 개씩 두 그룹으로 나누어 병합 정렬 함수를 재귀 호출함 → 요건 1, 2는 쉽게 확인할 수 있음
- 종료 조건은 어떨까?
- 병합 정렬의 입력 리스트에 자료가 한 개뿐이거나 아예 자료가 없을 때는 정렬할 필요 없음 → 입력 리스트를 그대로 돌려 주면서 재귀 호출을 끝냄

4 일반적인 병합 정렬 알고리즘

■ 프로그램 10-2 일반적인 병합 정렬 알고리즘


▼ 예제 소스 p10-2-msort.py

```
# 병합 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)


def merge_sort(a):
    n = len(a)
    # 종료 조건: 정렬할 리스트의 자료 개수가 한 개 이하이면 정렬할 필요가 없음
    if n <= 1:
        return
```



4 일반적인 병합 정렬 알고리즘



```
# 그룹을 나누어 각각 병합 정렬을 호출하는 과정
mid = n // 2 # 중간을 기준으로 두 그룹으로 나눔
g1 = a[:mid]
g2 = a[mid:]
merge_sort(g1) # 재귀 호출로 첫 번째 그룹을 정렬
merge_sort(g2) # 재귀 호출로 두 번째 그룹을 정렬
# 두 그룹을 하나로 병합
i1 = 0
i2 = 0
ia = 0
while i1 < len(g1) and i2 < len(g2):
    if g1[i1] < g2[i2]:
        a[ia] = g1[i1]
        i1 += 1
        ia += 1
```



4 일반적인 병합 정렬 알고리즘



```
else:
    a[ia] = g2[i2]
    i2 += 1
    ia += 1
# 아직 남아 있는 자료들을 결과에 추가
while i1 < len(g1):
    a[ia] = g1[i1]
    i1 += 1
    ia += 1
while i2 < len(g2):
    a[ia] = g2[i2]
    i2 += 1
    ia += 1
```

```
d = [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]
merge_sort(d)
print(d)
```

4 일반적인 병합 정렬 알고리즘

- 실행 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- 프로그램 10-1과 정렬 원리는 같지만, return 값이 없고 입력 리스트 안의 자료 순서를 직접 바꾼다는 차이가 있음

알고리즘 분석

분할 정복(divide and conquer)

- 병합 정렬은 주어진 문제를 절반으로 나눈 다음 각각을 재귀 호출로 풀어 가는 방식
- 큰 문제를 작은 문제로 나눠서(분할하여) 푸는(정복하는) 방법을 알고리즘 설계 기법에서는 '분할 정복(divide and conquer)'이라고 부름
- 입력 크기가 커서 풀기 어려웠던 문제도 반복해서 잘게 나누다 보면 굉장히 쉬운 문제(종료 조건)가 되는 원리를 이용한 것
- 분할 정복은 잘 활용하면 계산 복잡도가 더 낮은 효율적인 알고리즘을 만드는 데 도움이 됨

알고리즘 분석

- 분할 정복을 이용한 병합 정렬의 계산 복잡도는 $O(n \cdot \log n)$
- 선택 정렬이나 삽입 정렬의 계산 복잡도 $O(n^2)$ 보다 낮음
- 따라서 정렬해야 할 자료의 개수가 많을수록 병합 정렬이 선택 정렬이나 삽입 정렬보다 훨씬 더 빠른 정렬 성능을 발휘
- 예를 들어 대한민국 국민 오천만 명을 생년월일 순서로 정렬한다고 가정
- 입력 크기가 $n=50,000,000$ 일 때 n^2 은 2,500조이고 $n \cdot \log n$ 은 약 13억
- 2,500조는 13억보다 무려 200만 배 정도 큰 숫자
→ $O(n^2)$ 정렬 알고리즘과 $O(n \cdot \log n)$ 정렬 알고리즘의 계산 시간이 얼마나 많이 차이 나는지 짐작할 수 있음

연습 문제

- 10-1 프로그램 10-2에서 다룬 정렬 알고리즘은 숫자를 작은 수에서 큰 수 순서로 나열하는 오름차순 정렬이었습니다. 오름차순 정렬을 큰 수에서 작은 수 순서로 나열하는 내림차순 정렬로 바꾸려면 프로그램의 어느 부분을 바꿔야 할까요?

■ 10-1 내림차순 병합 정렬

▼ 예제 소스 e10-1-msort.py

```
# 내림차순 병합 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def merge_sort(a):
    n = len(a)
    # 종료 조건: 정렬할 리스트의 자료 개수가 한 개 이하이면 정렬할 필요가 없음
    if n <= 1:
        return
```



연습 문제 풀이

부록 A

그룹을 나누어 각각 병합 정렬을 호출하는 과정

```
mid = n // 2
```

```
g1 = a[:mid]
```

```
g2 = a[mid:]
```

```
merge_sort(g1)
```

```
merge_sort(g2)
```

두 그룹을 합치는 과정(병합)

```
i1 = 0
```

```
i2 = 0
```

```
ia = 0
```

```
while i1 < len(g1) and i2 < len(g2):
```

```
    if g1[i1] > g2[i2]: # 부등호 방향 뒤집기
```

```
        a[ia] = g1[i1]
```

```
        i1 += 1
```

```
        ia += 1
```


연습 문제 풀이

부록 A



```
else:
    a[ia] = g2[i2]
    i2 += 1
    ia += 1
while i1 < len(g1):
    a[ia] = g1[i1]
    i1 += 1
    ia += 1
while i2 < len(g2):
    a[ia] = g2[i2]
    i2 += 1
    ia += 1

d = [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]
merge_sort(d)
print(d)
```

연습 문제 풀이

부록 A

- 실행 결과

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- 오름차순 정렬에서 값을 비교하는 부분($g1[i1] < g2[i2]$)의 부등호 방향을 반대로 하면 내림차순 정렬 프로그램이 됨

문제 11 퀵 정렬

리스트 안의 자료를 작은 수부터 큰 수 순서로 배열하는 정렬 알고리즘을 만들어 보세요.

- 퀵 정렬은 '그룹을 둘로 나눠 재귀 호출'하는 방식은 병합 정렬과 같음
- 하지만 그룹을 나눌 때 미리 기준과 비교해서 나눈다는 점이 다름
- 즉, 먼저 기준과 비교해서 그룹을 나눈 후 각각 재귀 호출하여 합치는 방식

1 퀵 정렬로 줄 세우기

- 1) 학생들에게 일일이 지시하는 것이 귀찮아진 선생님은 학생들이 알아서 줄을 서는 방법이 없을지 고민. 그렇다고 열 명이나 되는 학생들에게 한 번에 알아서 줄을 서라고 하면 소란스러울 것 같아조를 나누려고 함
- 2) 열 명 중에 기준이 될 사람을 한 명 뽑음. 기준으로 뽑은 태호를 줄 가운데 세운 다음 태호보다 키가 작은 학생은 태호 앞에, 태호보다 큰 학생은 태호 뒤에 서게 함(학생들은 태호하고만 키를 비교하면 됨)
- 3) 기준인 태호는 가만히 있고, 태호보다 키가 작은 학생은 작은 학생들끼리, 큰 학생은 큰 학생들끼리 다시 키 순서대로 줄을 서면 줄 서기가 끝남

1 퀵 정렬로 줄 세우기



그림 11-1 퀵 정렬

2 쉽게 설명한 퀵 정렬 알고리즘

- 프로그램 11-1 쉽게 설명한 퀵 정렬 알고리즘

▼ 예제 소스 p11-1-qsort.py

```
# 쉽게 설명한 퀵 정렬
# 입력: 리스트 a
# 출력: 정렬된 새 리스트

def quick_sort(a):
    n = len(a)
    # 종료 조건: 정렬할 리스트의 자료 개수가 한 개 이하이면 정렬할 필요가 없음
    if n <= 1:
        return a
```



쉽게 설명한 퀵 정렬 알고리즘



```
# 기준 값을 정하고 기준에 맞춰 그룹을 나누는 과정
pivot = a[-1] # 편의상 리스트의 마지막 값을 기준 값으로 정함
g1 = [] # 그룹 1: 기준 값보다 작은 값을 담은 리스트
g2 = [] # 그룹 2: 기준 값보다 큰 값을 담은 리스트
for i in range(0, n - 1): # 마지막 값은 기준 값이므로 제외
    if a[i] < pivot: # 기준 값과 비교
        g1.append(a[i]) # 작으면 g1에 추가
    else:
        g2.append(a[i]) # 크면 g2에 추가
# 각 그룹에 대해 재귀 호출로 퀵 정렬을 한 후
# 기준 값과 합쳐 하나의 리스트로 결과값 반환
return quick_sort(g1) + [pivot] + quick_sort(g2)
```

```
d = [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]
```

```
print(quick_sort(d))
```


쉽게 설명한 퀵 정렬 알고리즘

- 실행 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

쉽게 설명한 퀵 정렬 알고리즘

- 퀵 정렬 함수 `quick_sort()`는 재귀 호출 함수
→ 병합 정렬과 마찬가지로 첫 부분에 종료 조건이 명시되어 있음
- 입력으로 주어진 리스트 `a`의 크기가 1 이하이면, 즉 자료가 한 개뿐이거나
아예 비어 있다면 정렬할 필요가 없으므로 입력 리스트를 그대로 돌려주
면서 재귀 호출을 끝냄

```
n = len(a)
if n <= 1:
    return a
```

2 쉽게 설명한 퀵 정렬 알고리즘

- 퀵 정렬에서는 그룹을 나누기 위한 기준 값(pivot)이 필요
- 프로그램 11-1에서는 주어진 리스트의 맨 마지막 값을 기준 값으로 사용

```
pivot = a[-1]
```

- 다음 문장은 g1을 퀵 정렬한 결과에 기준 값과 g2를 퀵 정렬한 결과를 이어 붙여 새로운 리스트를 만들어 돌려주는 문장

```
return quick_sort(g1) + [pivot] + quick_sort(g2)
```

쉽게 설명한 퀵 정렬 알고리즘

- 두 개 이상의 리스트를 더하기로 연결하면 각 리스트 안의 자료를 순서대로 포함하는 새 리스트를 만들 수 있음

```
>>> [1, 2] + [3] + [4, 5]  
[1, 2, 3, 4, 5]
```

쉽게 설명한 퀵 정렬 알고리즘

- 리스트 [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]를 퀵 정렬하는 과정

① 리스트에서 기준 값을 하나 정합니다. 이 책에서는 편의상 정렬할 리스트의 맨 마지막 값을 기준으로 정하였습니다.

[6 8 3 9 10 1 2 4 7 5]의 기준 값: 5

② 기준 값보다 작은 값을 저장할 리스트로 g1, 큰 값을 저장할 리스트로 g2를 만듭니다.

③ 리스트에 있는 자료들을 기준 값인 5와 차례로 비교하여 5보다 작은 값을 g1, 큰 값을 g2에 넣습니다. 예를 들어 6은 5보다 크므로 g2에 넣고, 그 다음 값인 8도 5보다 크므로 g2, 3은 5보다 작으므로 g1에 넣습니다.

기준 값: 5

g1: [3 1 2 4]

g2: [6 8 9 10 7]

2 쉽게 설명한 퀵 정렬 알고리즘

④ 재귀 호출을 이용하여 $g1$ 을 정렬합니다. 함수 안에서 퀵 정렬을 재귀 호출하면서 문제를 풀어 $[1\ 2\ 3\ 4]$ 를 결과로 돌려줍니다.

⑤ 재귀 호출을 이용하여 $g2$ 를 정렬합니다. 마찬가지로 퀵 정렬로 문제를 풀어 $[6\ 7\ 8\ 9\ 10]$ 을 결과로 돌려줍니다.

⑥ 이제 $g1$ 에는 '기준보다 작은 값들'이 정렬되어 있고 $g2$ 에는 '기준보다 큰 값들'이 정렬되어 있습니다. 따라서 $g1$, 기준 값, $g2$ 를 순서대로 이어 붙이면 정렬이 완료됩니다.
 $[1\ 2\ 3\ 4] + [5] + [6\ 7\ 8\ 9\ 10] \rightarrow [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10]$

⑦ 최종 결과: $[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10]$

3 일반적인 퀵 정렬 알고리즘

- 프로그램 11-2 일반적인 퀵 정렬 알고리즘

▼ 예제 소스 p11-2-qsort.py

```
# 퀵 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)
# 리스트 a의 어디부터(start) 어디까지(end)가 정렬 대상인지
# 범위를 지정하여 정렬하는 재귀 호출 함수
def quick_sort_sub(a, start, end):
    # 종료 조건: 정렬 대상이 1개 이하면 정렬할 필요 없음
    if end - start <= 0:
        return
    # 기준 값을 정하고 기준 값에 맞춰 리스트 안에서 각 자료의 위치를 맞춤
    # [기준 값보다 작은 값들, 기준 값, 기준 값보다 큰 값들]
    pivot = a[end]    # 편의상 리스트의 마지막 값을 기준 값으로 정함
    i = start
```



3 일반적인 퀵 정렬 알고리즘



```
for j in range(start, end):
    if a[j] <= pivot:
        a[i], a[j] = a[j], a[i]
        i += 1
a[i], a[end] = a[end], a[i]
# 재귀 호출 부분
quick_sort_sub(a, start, i - 1) # 기준 값보다 작은 그룹을 재귀 호출로 다시 정렬
quick_sort_sub(a, i + 1, end)  # 기준 값보다 큰 그룹을 재귀 호출로 다시 정렬

# 리스트 전체(0 ~ len(a)-1)를 대상으로 재귀 호출 함수 호출
def quick_sort(a):
    quick_sort_sub(a, 0, len(a) - 1)

d = [6, 8, 3, 9, 10, 1, 2, 4, 7, 5]
quick_sort(d)
print(d)
```


3 일반적인 퀵 정렬 알고리즘

- 실행 결과

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

기준 값의 중요성

- 줄 세우기에서 선생님이 기준으로 정한 학생이 하필이면 열 명 중 키가 가장 작은 학생이었다면 어떻게 될까?
- 기준보다 작은 그룹($g1$)에는 학생이 한 명도 없고, 기준보다 큰 그룹($g2$)에는 나머지 학생이 모두 모인 상황이 됨
- 이렇게 되면 그룹을 둘로 나눈 의미가 없어져 퀵 정렬의 효율이 낮아짐
- 따라서 퀵 정렬에서는 '좋은 기준'을 정하는 것이 정렬의 효율성을 가늠하므로 굉장히 중요

4 기준 값의 중요성

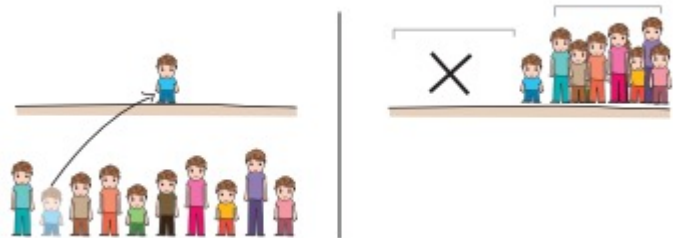


그림 11-2 기준을 잘못 정한 예 ①: 가장 작은 학생을 기준으로 잡은 경우

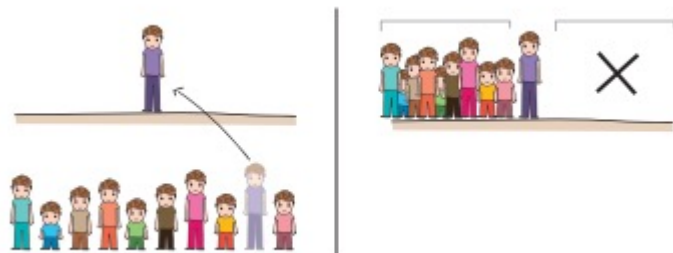


그림 11-3 기준을 잘못 정한 예 ②: 가장 큰 학생을 기준으로 잡은 경우

알고리즘 분석

- 퀵 정렬의 계산 복잡도는 최악의 경우 선택 정렬이나 삽입 정렬과 같은 $O(n^2)$ 이지만, 평균적일 때는 병합 정렬과 같은 $O(n \cdot \log n)$
- 최악의 경우란 그림 11-2나 11-3과 같이 기준을 잘못 정하여 그룹이 제대로 나뉘지 않았을 때
- 하지만 다행히도 좋은 기준 값을 정하는 알고리즘에 관해 서는 이미 많이 연구가 되어 있기 때문에 퀵 정렬은 대부분의 경우 $O(n \cdot \log n)$ 으로 정렬을 마칠 수 있음

연습 문제

- 11-1 지금까지 배운 네 가지 정렬 알고리즘 말고도 훨씬 많은 정렬 알고리즘이 있습니다. 그 중 하나인 거품 정렬(Bubble sort)을 줄 서기로 비유하면 다음과 같습니다. 다음 과정을 읽고 리스트 [2, 4, 5, 1, 3]이 정렬되는 과정을 알고리즘으로 적어 보세요.

연습 문제

- 거품 정렬(Bubble sort)을 줄 서기로 비유한 것
 - 1) 일단 학생들을 아무렇게나 일렬로 줄을 세움
 - 2) 선생님이 맨 앞에서부터 뒤로 이동하면서 이웃한 앞뒤 학생의 키를 서로 비교. 앞에 있는 학생의 키가 바로 뒤에 있는 학생보다 크면 두 학생의 자리를 서로 바꿈
 - 3) 선생님은 계속 뒤로 이동하면서 이웃한 앞뒤 학생의 키를 비교해서 필요하면 앞뒤 학생의 위치를 서로 바꿈
 - 4) 모든 학생이 키 순서대로 줄 설 때까지 이 과정을 반복(줄의 끝까지 확인하는 동안 자리를 바꾼 적이 한 번도 없으면 모든 학생이 순서대로 줄을 선 것)

연습 문제

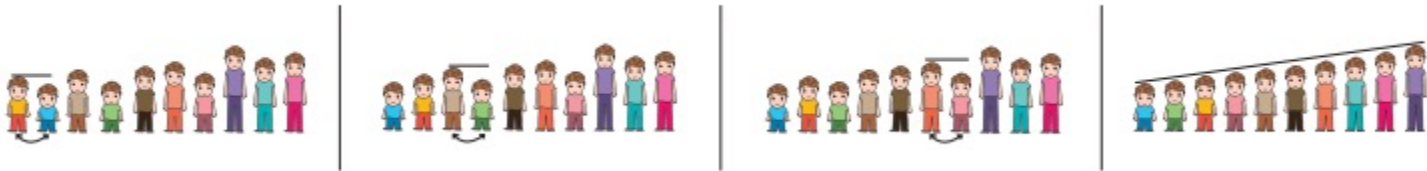


그림 11-4 거품 정렬

■ 11-1 거품 정렬

[2 4 5 1 3] ← 2 < 4이므로 그대로 둡니다.

[2 4 5 1 3] ← 4 < 5이므로 그대로 둡니다.

[2 4 5 1 3] ← 5 > 1이므로 5와 1의 위치를 서로 바꿉니다.

[2 4 1 5 3] ← 5 > 3이므로 5와 3의 위치를 서로 바꿉니다.

[2 4 1 3 5] ← 다시 앞에서부터 반복, 2 < 4이므로 그대로 둡니다.

[2 4 1 3 5] ← 4 > 1이므로 서로 위치를 바꿉니다.

[2 1 4 3 5] ← 4 > 3이므로 서로 위치를 바꿉니다.

[2 1 3 4 5] ← 4 < 5이므로 그대로 둡니다.

[2 1 3 4 5] ← 다시 앞에서부터 반복, 2 > 1이므로 서로 위치를 바꿉니다.

[1 2 3 4 5] ← 더는 바꿀 것이 없으므로 정렬을 마칩니다(최종 결과).

■ 11-1 거품 정렬

▼ 예제 소스 e11-1-bsort.py

```
# 거품 정렬
# 입력: 리스트 a
# 출력: 없음(입력으로 주어진 a가 정렬됨)

def bubble_sort(a):
    n = len(a)
    while True: # 정렬이 완료될 때까지 계속 수행
        changed = False # 자료를 앞뒤로 바꾸었는지 여부
        # 자료를 훑어 보면서 뒤집힌 자료가 있으면 바꾸고 바뀌었다고 표시
        for i in range(0, n - 1):
            if a[i] > a[i + 1]: # 앞이 뒤보다 크면
                print(a)      # 정렬 과정 출력(참고용)
```



연습 문제 풀이

부록 A



```
a[i], a[i + 1] = a[i + 1], a[i] # 두 자료의 위치를 맞바꿈
changed = True # 자료가 앞뒤로 바뀌었음을 기록
# 자료를 한 번 훑어보는 동안 바뀐 적이 없다면 정렬이 완성된 것이므로 종료
# 바뀐 적이 있으면 다시 앞에서부터 비교 반복
if changed == False:
    return
```

```
d = [2, 4, 5, 1, 3]
bubble_sort(d)
print(d)
```

연습 문제 풀이

부록 A

■ 실행 결과

[2, 4, 5, 1, 3]

[2, 4, 1, 5, 3]

[2, 4, 1, 3, 5]

[2, 1, 4, 3, 5]

[2, 1, 3, 4, 5]

[1, 2, 3, 4, 5]

연습 문제 풀이

부록 A

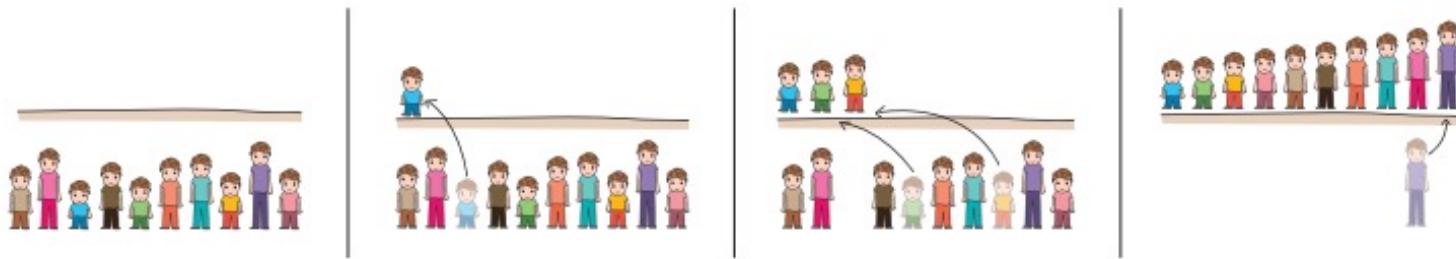
거품 정렬의 특징

- 거품 정렬의 입력으로 이미 정렬된 리스트가 주어졌을 때는 리스트를 한 번 훑어보는 동안 바꿀 자료가 없으므로 바로 정렬이 종료됨
- 즉, 최선의 경우 계산 복잡도는 $O(n)$
- 단, 이미 정렬된 리스트가 아닌 일반적인 입력에 대한 거품 정렬의 계산 복잡도는 $O(n^2)$
- 하지만 거품 정렬은 자료 위치를 서로 바꾸는 횟수가 선택 정렬이나 삽입 정렬보다 더 많기 때문에 실제로 더 느리게 동작한다는 단점이 있음

정렬 알고리즘 요약

한눈에 보는 정렬 알고리즘

[1] 선택 정렬

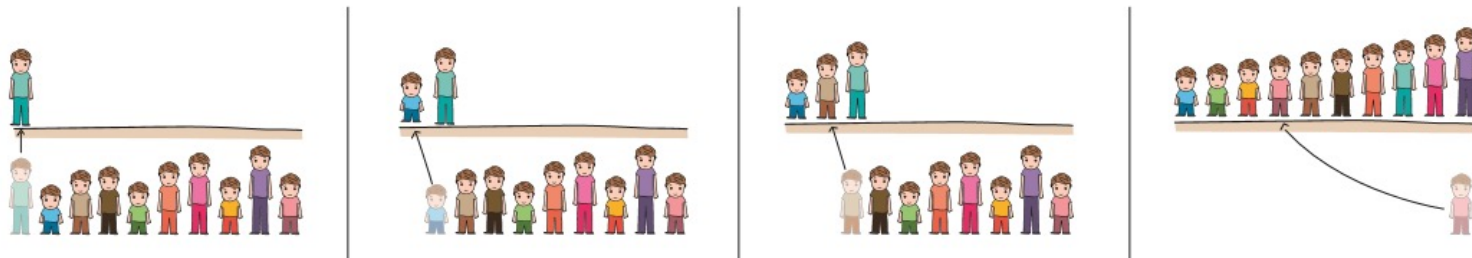


- 동작 원리: 남은 자료 중에 최솟값을 뽑아 차례로 배치
- 계산 복잡도: $O(n^2)$

정렬 알고리즘 요약

한눈에 보는 정렬 알고리즘

[2] 삽입 정렬

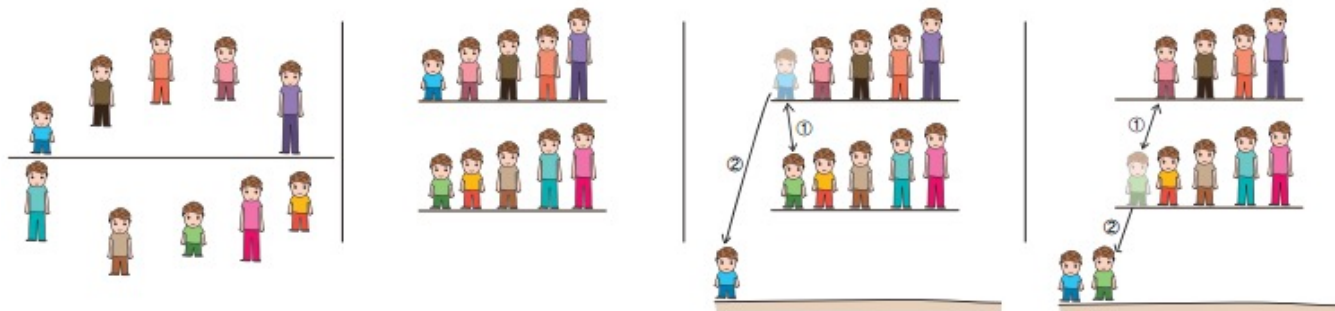


- 동작 원리: 자료를 하나씩 적절한 위치에 삽입
- 계산 복잡도: 보통의 경우 $O(n^2)$

정렬 알고리즘 요약

한눈에 보는 정렬 알고리즘

[3] 병합 정렬

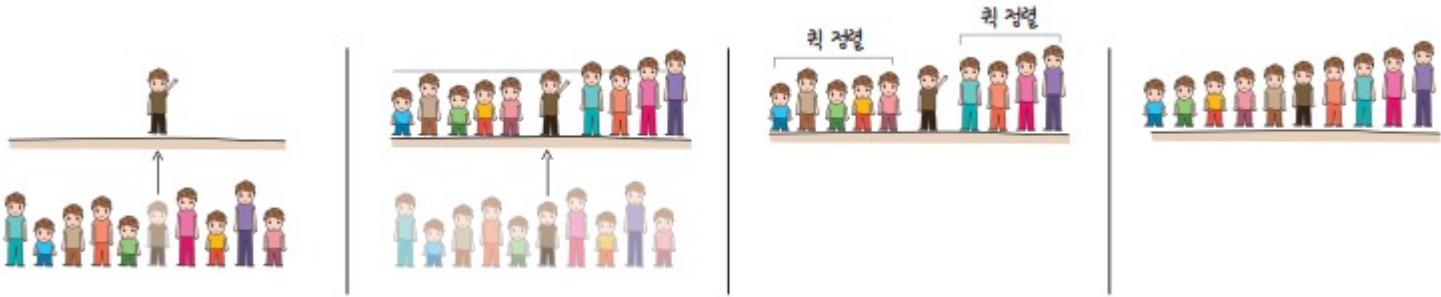


- 동작 원리: 그룹 나누기 → 그룹별로 각각 정렬(재귀 호출) → 병합
- 계산 복잡도: $O(n \cdot \log n)$

정렬 알고리즘 요약

한눈에 보는 정렬 알고리즘

[4] 퀵 정렬

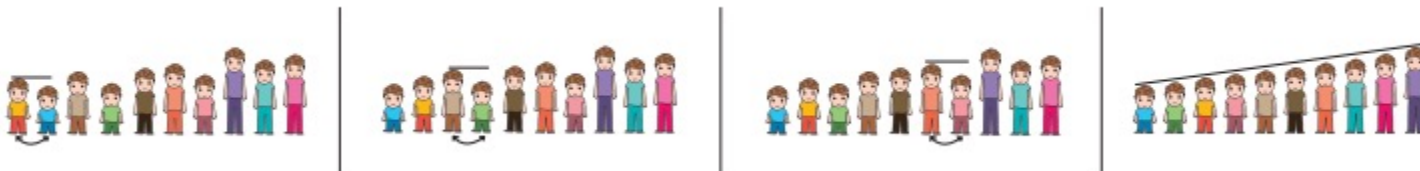


- 동작 원리: 기준 선택 → 기준에 맞춰 그룹 나누기 → 그룹별로 각각 정렬 (재귀 호출)
- 계산 복잡도: 보통의 경우 $O(n \cdot \log n)$

정렬 알고리즘 요약

한눈에 보는 정렬 알고리즘

[5] 거품 정렬



- 동작 원리: 앞뒤로 이웃한 자료를 비교 → 크기가 뒤집힌 경우 서로 위치를 바꿈
- 계산 복잡도: 보통의 경우 $O(n^2)$

파이썬의 정렬 기능

파이썬의 정렬

- 파이썬, 자바, C#과 같은 최신 컴퓨터 프로그래밍 언어는 대부분 정렬 기능 내장
- 파이썬에서는 `sort()` 혹은 `sorted()` 함수를 이용하면 리스트 쉽게 정렬 가능

```
>>> sorted([5, 2, 3, 1, 4])  
[1, 2, 3, 4, 5]
```

```
>>> a = [5, 2, 3, 1, 4]  
>>> a.sort()  
>>> a  
[1, 2, 3, 4, 5]
```

파이썬의 정렬 기능

- `sorted()` 함수는 인자로 리스트를 주면 그 리스트를 정렬한 리스트를 새로 만들어 돌려줌
- 반면에 `sort()` 함수는 새 리스트를 따로 만들지 않고 정렬 대상이 되는 리스트 자체의 순서를 바꿔 줌
- 그렇다면 파이썬은 실제로 어떤 정렬 알고리즘으로 정렬을 하는 걸까?
→ 팀소트(Timsort)라는 알고리즘을 이용해 정렬
- 팀소트는 우리가 이미 배운 병합 정렬과 삽입 정렬의 아이디어를 적절하게 섞어 만든 새로운 정렬 알고리즘으로 평균 계산 복잡도는 $O(n \cdot \log n)$

문제 12 이분 탐색

자료가 크기 순서대로 정렬된 리스트에서 특정한 값이 있는지 찾아 그 위치를 돌려주는 알고리즘을 만들어 보세요. 리스트에 찾는 값이 없으면 -1을 돌려줍니다.

- 문제 7과 같지만 이번에는 리스트의 자료가 순서대로 정렬되어 있으므로 훨씬 더 빠르게 탐색할 수 있음
- 이분 탐색(Binary search)의 이분(二分)은 '둘로 나눈다'는 뜻
- 탐색할 자료를 둘로 나누어 찾는 값이 있을 법한 곳만 탐색하기 때문에 자료를 하나하나 찾아야 하는 순차 탐색보다 원하는 자료를 훨씬 빨리 찾을 수 있음

1 일상생활 속의 탐색 문제

- 두꺼운 책을 한 권 앞에 두고 특정한 쪽 수(예를 들어 618쪽)를 찾는 과정
 - 1) 우선 책의 중간쯤을 펼쳐 쪽 수를 보니 520쪽
 - 2) 찾고자 하는 쪽 수가 펼친 쪽 수보다 크므로($618 > 520$) 펼친 곳의 앞쪽은 더 이상 찾을 필요가 없음
 - 3) 현재 펼친 곳에서 뒤쪽으로 적당해 보이는 곳을 다시 펼치니 710쪽
 - 4) 찾고자 하는 쪽 수가 펼친 쪽 수보다 작으므로($618 < 710$) 펼친 곳의 뒤쪽은 더 이상 찾을 필요가 없음
 - 5) 이번에는 다시 앞쪽으로 책을 펼쳤더니 613쪽이 나옴

1 일상생활 속의 탐색 문제

- 6) 찾으려는 쪽 수와 가까운 쪽까지 왔으니 이제 쪽을 한 장 한 장 뒤로 넘김
- 7) 원하는 618쪽이 나오면 탐색을 멈춤



그림 12-1 책에서 원하는 쪽을 찾는 과정

1 일상생활 속의 탐색 문제

- 1~5번 과정, 즉 책을 적당히 펼쳐 쪽을 비교한 다음에 찾고자 하는 쪽이 있을 방향(앞인지 뒤인지)으로만 다시 탐색하는 과정 → 이분 탐색
- 찾으려는 쪽이 몇 쪽 남지 않았을 때 한 장씩 넘기면서 찾는 과정은 이미 문제 7에 서 배운 적이 있는 순차 탐색과 비슷함
- 책에서 특정한 쪽을 찾을 때 우리가 이분 탐색을 할 수 있었던 이유는?
- 모든 책의 쪽 수가 1부터 빠짐없이 차례로 커지고 있었기 때문
- 즉, 책의 쪽 번호가 이미 정렬되어 있으므로 특정 쪽의 앞쪽을 찾아보아야 할지 뒤쪽을 찾아보아야 할지 바로 알 수 있음

1 일상생활 속의 탐색 문제

- 굉장히 큰 호텔에서 원하는 방의 호수를 찾는 것 역시 탐색 문제

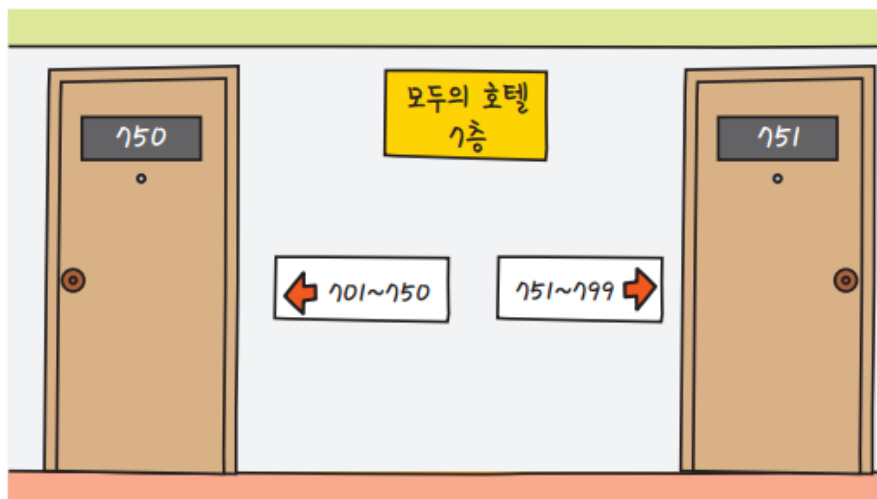


그림 12-2 호텔 엘리베이터에서 내리면 볼 수 있는 방 번호 표지판

1 일상생활 속의 탐색 문제

- 743호를 찾으려면 엘리베이터로 7층까지 올라간 다음 표지판을 보고
오른쪽에 있는 방(751~799호)은 무시하고 왼쪽 복도로 걸어 감
→ 이분 탐색의 원리를 이용하여 탐색할 범위를 절반으로 줄인 예
- 왼쪽 복도로 걸어가면서 방문에 붙은 방 번호를 743과 하나하나 비교
→ 순차 탐색에 해당

이분 탐색 알고리즘

- 정렬된 리스트에서 특정 값을 찾으려면 어떻게 해야 할까?
 - 리스트: [1, 4, 9, 16, 25, 36, 49, 64, 81]
 - 찾는 값: 36
-
- 1) 먼저 전체 리스트의 중간 위치를 찾음. 위치 번호 4가 리스트의 중간 위치이고, 중간 위치 값은 25
 - 2) 찾는 값 36과 중간 위치 값을 비교. $36 > 25$ 이므로 36이 리스트 안에 있다면 반드시 25의 오른쪽에 있어야 함. 즉, 리스트에서 25보다 오른쪽에 있는 값만 대상으로 생각하면 됨

2 이분 탐색 알고리즘

- 3) 이제 [36, 49, 64, 81] 리스트에서 중간 위치를 찾음. 이 경우 49와 64의 한가운데가 중간 위치가 되는데, 두 자료 중 앞에 있는 값인 49를 중간 위치 값으로 뽑음
- 4) 찾는 값 36과 중간 위치 값 49를 비교. $36 < 49$ 이므로 찾는 값 36은 처음에 비교한 값인 25보다는 오른쪽에 있고 49보다는 왼쪽에 있음
- 5) '25보다 오른쪽에 있고 49보다 왼쪽에 있는 값'은 한 개뿐이므로 위치 번호 5의 36이 중간 위치 값임
- 6) 찾는 값 36이 중간 위치 값과 같으므로 위치 번호 5를 결과값으로 돌려 주고 종료

2 이분 탐색 알고리즘

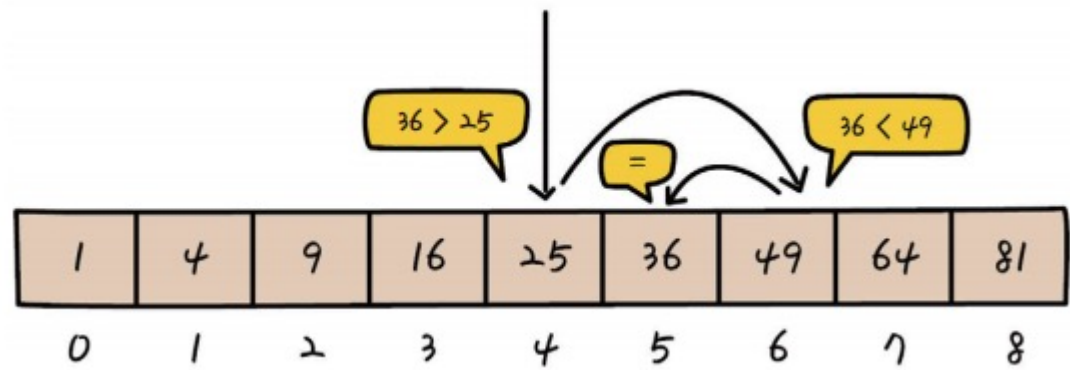


그림 12-3 이분 탐색으로 36을 찾는 과정

2 이분 탐색 알고리즘

- 이분 탐색 알고리즘 정리

- 1) 중간 위치를 찾음
- 2) 찾는 값과 중간 위치 값을 비교
- 3) 같다면 원하는 값을 찾은 것이므로 위치 번호를 결과값으로 돌려줌
- 4) 찾는 값이 중간 위치 값보다 크다면 중간 위치의 오른쪽을 대상으로 다시 탐색(1번 과정부터 반복)
- 5) 찾는 값이 중간 위치 값보다 작다면 중간 위치의 왼쪽을 대상으로 다시 탐색(1번 과정부터 반복)

2 이분 탐색 알고리즘

- 자료의 중간부터 시작해 찾을 값이 더 크면 오른쪽으로, 작으면 왼쪽으로 점프하며 자료를 찾음
- 점프할 때마다 점프 거리는 절반씩 줄어듦

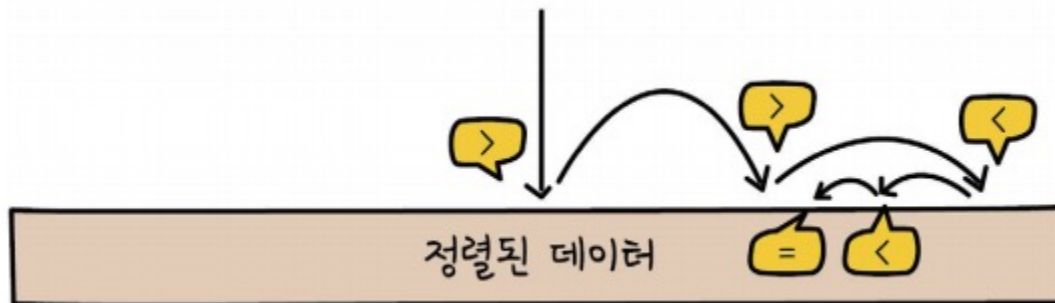


그림 12-4 이분 탐색의 과정

2 이분 탐색 알고리즘

■ 프로그램 12-1 이분 탐색 알고리즘

▼ 예제 소스 p12-1-bsearch.py

```
# 리스트에서 특정 숫자 위치 찾기(이분 탐색)
# 입력: 리스트 a, 찾는 값 x
# 출력: 찾으면 그 값의 위치, 찾지 못하면 -1
```

```
def binary_search(a, x):
```

```
    # 탐색할 범위를 저장하는 변수 start, end
```

```
    # 리스트 전체를 범위로 탐색 시작(0 ~ len(a)-1)
```

```
    start = 0
```

```
    end = len(a) - 1
```

```
    while start <= end: # 탐색할 범위가 남아 있는 동안 반복
```

```
        mid = (start + end) // 2 # 탐색 범위의 중간 위치
```



이분 탐색 알고리즘



```
if x == a[mid]: # 발견!
    return mid
elif x > a[mid]: # 찾는 값이 더 크면 오른쪽으로 범위를 좁혀 계속 탐색
    start = mid + 1
else:           # 찾는 값이 더 작으면 왼쪽으로 범위를 좁혀 계속 탐색
    end = mid - 1
```

```
return -1 # 찾지 못했을 때
```

```
d = [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
print(binary_search(d, 36))
```

```
print(binary_search(d, 50))
```

2 이분 탐색 알고리즘

- 실행 결과

```
5  
-1
```

3 알고리즘 분석

- 이분 탐색은 값을 비교할 때마다 찾는 값이 있을 범위를 절반씩 좁히면서 탐색하는 효율적인 탐색 알고리즘
- 자료가 천 개 있을 때 원하는 자료를 찾는다고 생각해 보자
- 순차 탐색은 최악의 경우에 자료 천 개와 모두 비교해야 함
- 이분 탐색은 최악의 경우에도 자료 열 개와 비교하면 탐색을 마칠 수 있음
($\log_2 1,000 \approx 9.966$)

3 알고리즘 분석

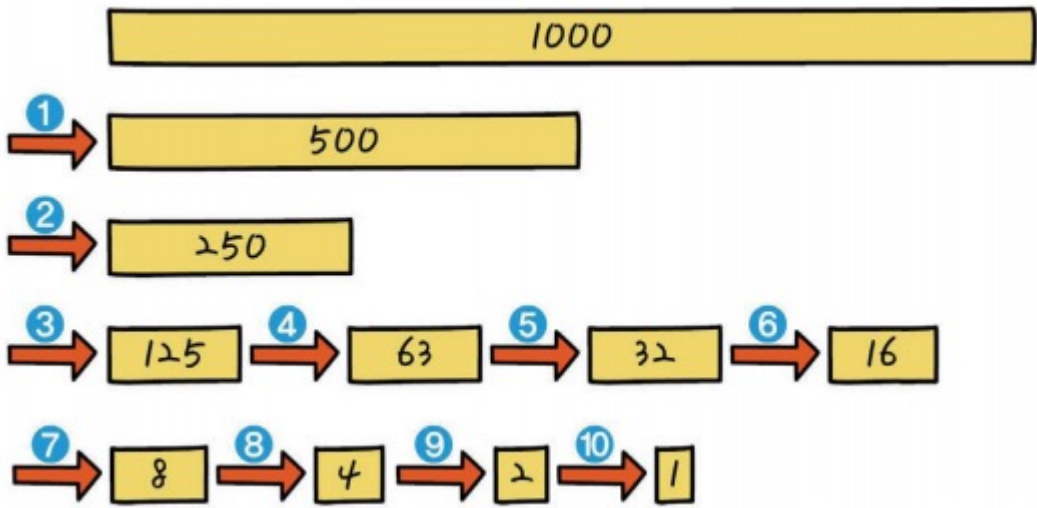


그림 12-5 이분 탐색으로 탐색 범위를 1까지 좁혀 가는 과정

알고리즘 분석

- 이분 탐색의 계산 복잡도는 $O(\log n)$
→ 순차 탐색의 계산 복잡도인 $O(n)$ 보다 훨씬 더 효율적
- 대한민국 전 국민 오천만 명 중에서 주민등록번호로 한 명을 찾는다면?
- 순차 탐색으로는 최악의 경우 주민등록번호가 같은지 오천만 번 비교해야 함(찾는 사람이 자료의 맨 마지막에 있을 때)
- 이분 탐색으로는 최악의 경우라도 26명의 주민등록번호와 같은지 혹은 큰지를 비교하면 원하는 사람을 찾을 수 있음($\log_2 50,000,000 \approx 25.575$)

연습 문제

- 12-1 다음 과정을 참고하여 재귀 호출을 사용한 이분 탐색 알고리즘을 만들어 보세요.
 - ① 주어진 탐색 대상이 비어 있다면 탐색할 필요가 없습니다(종료 조건).
 - ② 찾는 값과 주어진 탐색 대상의 중간 위치 값을 비교합니다.
 - ③ 찾는 값과 중간 위치 값이 같다면 결과값으로 중간 위치 값을 돌려줍니다.
 - ④ 찾는 값이 중간 위치 값보다 크다면 중간 위치의 오른쪽을 대상으로 이분 탐색 함수를 재귀 호출합니다.
 - ⑤ 찾는 값이 중간 위치 값보다 작다면 중간 위치의 왼쪽을 대상으로 이분 탐색 함수를 재귀 호출합니다.

연습 문제 풀이

부록 A

12-1 재귀 호출을 이용한 이분 탐색

▼ 예제 소스 e12-1-bsearch.py

```
# 리스트에서 특정 숫자 위치 찾기(이분 탐색과 재귀 호출)
# 입력: 리스트 a, 찾는 값 x
# 출력: 특정 숫자를 찾으면 그 값의 위치, 찾지 못하면 -1

# 리스트 a의 어디부터(start) 어디까지(end)가 탐색 범위인지 지정하여
# 그 범위 안에서 x를 찾는 재귀 함수
def binary_search_sub(a, x, start, end):
    # 종료 조건: 남은 탐색 범위가 비었으면 종료
    if start > end:
        return -1
```





```
mid = (start + end) // 2 # 탐색 범위의 중간 위치
if x == a[mid]: # 발견!
    return mid
elif x > a[mid]: # 찾는 값이 더 크면 중간을 기준으로 오른쪽 값을 대상으로 재귀 호출
    return binary_search_sub(a, x, mid + 1, end)
else: # 찾는 값이 더 작으면 중간을 기준으로 왼쪽 값을 대상으로 재귀 호출
    return binary_search_sub(a, x, start, mid - 1)

return -1 # 찾지 못했을 때

# 리스트 전체(0 ~ len(a)-1)를 대상으로 재귀 호출 함수 호출
def binary_search(a, x):
    return binary_search_sub(a, x, 0, len(a) - 1)
```



연습 문제 풀이

부록 A



```
d = [1, 4, 9, 16, 25, 36, 49, 64, 81]
print(binary_search(d, 36))
print(binary_search(d, 50))
```

■ 실행 결과

```
5
-1
```

계산 복잡도 비교

계산 복잡도 비교

- 대문자 O 표기법을 계산이 간단한 것에서 복잡한 것 순으로 정리

① $O(1)$: 입력 크기 n 과 계산 복잡도가 무관할 때

예) 계산 공식 $n(n+1)/2$ 를 이용한 1부터 n 까지의 합(문제 1)

② $O(\log n)$: 입력 크기 n 의 로그 값에 비례하여 계산 복잡도가 증가할 때

예) 이분 탐색(문제 12)







③ $O(n)$: 입력 크기 n 에 비례하여 계산 복잡도가 증가할 때

예) 최댓값 찾기(문제 2), 순차 탐색(문제 7)

계산 복잡도 비교

- ④ $O(n \cdot \log n)$: 입력 크기 n 과 로그 n 값의 곱에 비례하여 계산 복잡도가 증가할 때
예) 병합 정렬(문제 10), 퀵 정렬(문제 11)
- ⑤ $O(n^2)$: 입력 크기 n 의 제곱에 비례하여 계산 복잡도가 증가할 때
예) 선택 정렬(문제 8), 삽입 정렬(문제 9)
- ⑥ $O(2^n)$: 입력 크기가 n 일 때 2의 n 제곱 값에 비례하여 계산 복잡도가 증가할 때
예) 하노이의 탑(문제 6)

계산 복잡도 비교

	$O(1)$	$O(\log n)$	$O(n)$	$O(n \cdot \log n)$	$O(n^2)$	$O(2^n)$
그래프						
$n = 10$	1	3.3	10	32.2	100	1024
$n = 100$	1	6.6	100	664.4	10000	1267650... (31자리 숫자)
$n = 10000$	1	13.3	10000	132877.1	100000000	1995063... (3011자리 숫자)