

12

CHAPTER

정렬 고급

학습목표

- 성능을 향상시킬 수 있는 정렬 방식을 학습한다.
- 고급 정렬을 활용한 응용 프로그래밍을 작성한다.

SECTION 00 생활 속 자료구조와 알고리즘

SECTION 01 고급 정렬 알고리즘의 원리와 구현

SECTION 02 고급 정렬 알고리즘의 응용

연습문제

응용예제



Section 00 생활 속 자료구조와 알고리즘

- 적은 인원을 이름 순서대로 줄을 세우는 것은 어렵지 않다.
하지만 여름철 해수욕장에 모인 많은 인원의 이름을 순서대로 정렬하는 것은 상당히 오랜 시간이 걸린다.
그래서 정렬 방식 중에서도 빠르게 정렬되는 방법을 채택해서 사용해야 한다.



Section 01 고급 정렬 알고리즘의 원리와 구현

■ 버블 정렬(Bubble Sort)

■ 버블 정렬의 개념

- 첫 번째 값부터 시작해서 바로 앞뒤 데이터를 비교하여 큰 것은 뒤로 보내는 방법을 사용하는 정렬
- 정렬을 위해 비교하는 모양이 거품(Bubble)처럼 생긴 것에서 이름 유래



그림 12-1 버블 정렬의 기본 개념

■ 버블 정렬의 구현

- 4개 데이터를 버블 정렬하는 예



(a) 버블 정렬할 데이터



(b) 3회 사이클

그림 12-2 버블 정렬 예

Section 01 고급 정렬 알고리즘의 원리와 구현

- 사이클1에서는 데이터 4개 중 앞부터 2개씩을 비교해서 큰 것을 뒤로 보냄

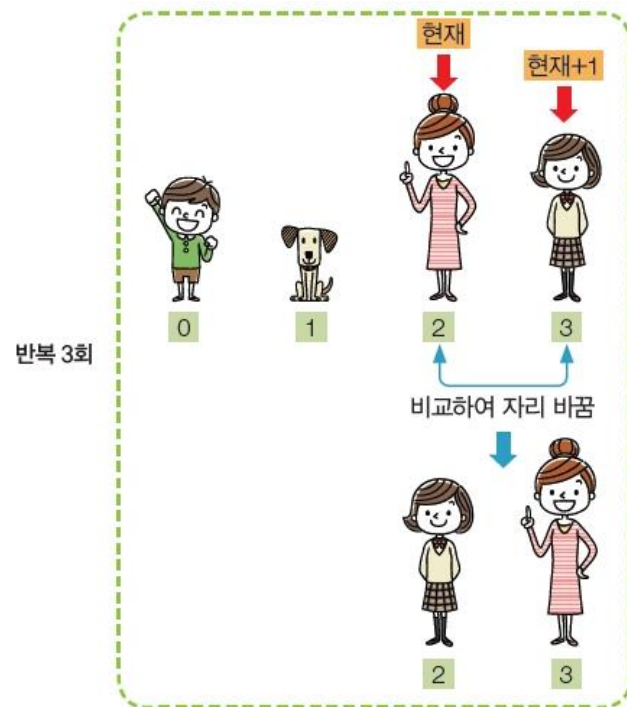
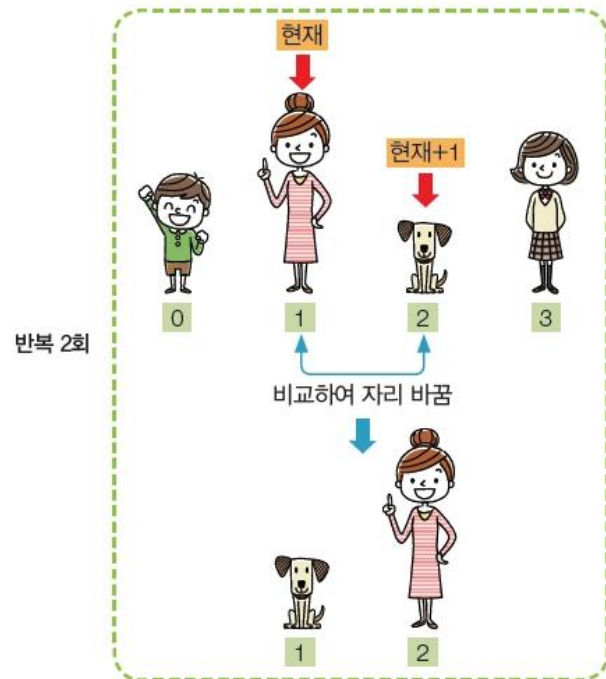
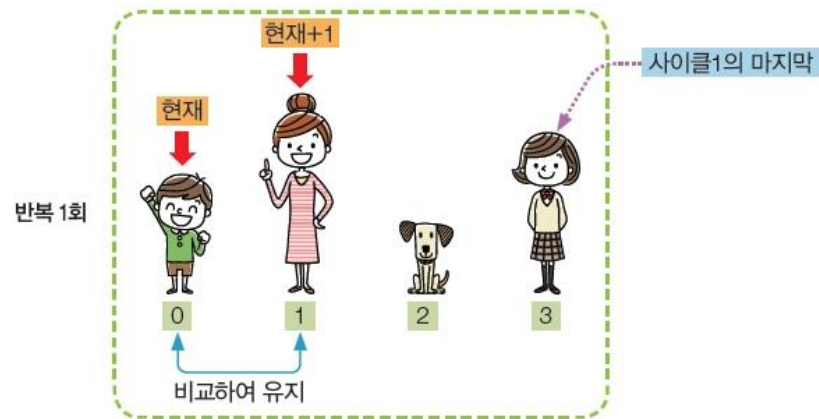


그림 12-3 버블 정렬 예 : 사이클1

Section 01 고급 정렬 알고리즘의 원리와 구현

- 사이클2에서는 데이터 3개 중 앞부터 2개씩을 비교해서 큰 것을 뒤로 보냄

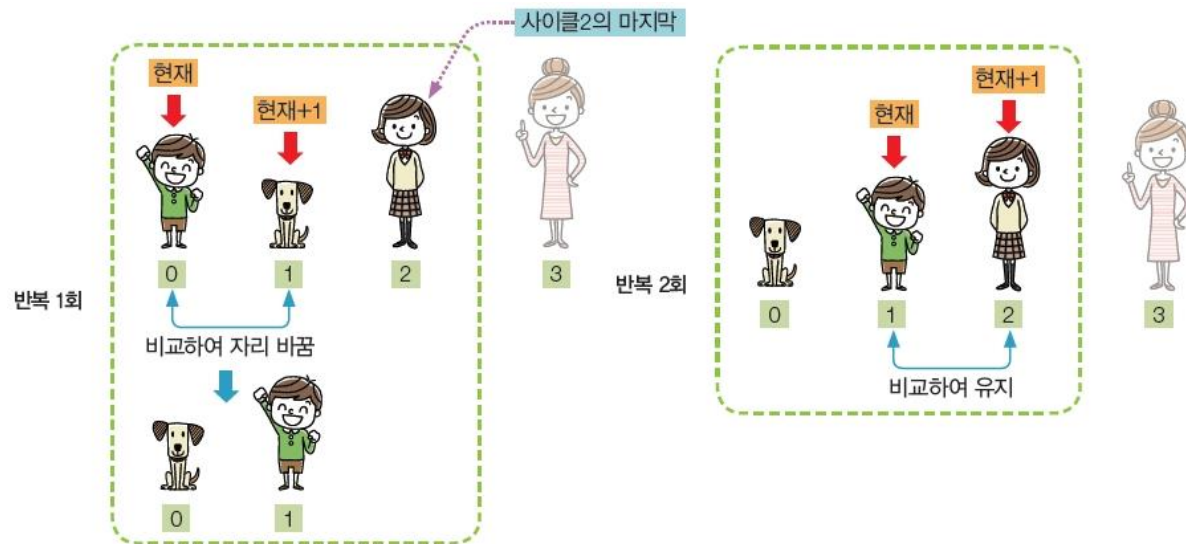


그림 12-4 버블 정렬 예 : 사이클2

- 사이클3에서는 데이터 2개 중 앞부터 2개씩 비교해서 큰 것을 뒤로 보냄

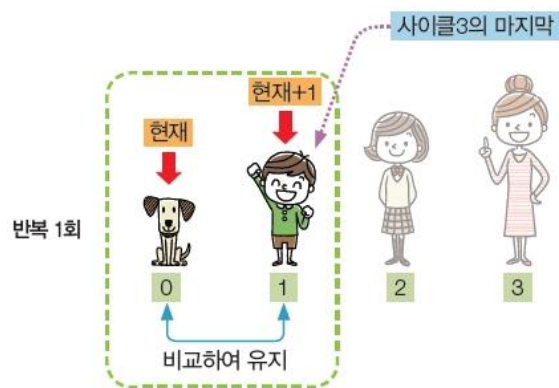


그림 12-5 버블 정렬 예 : 사이클3

Section 01 고급 정렬 알고리즘의 원리와 구현

- 모든 사이클 완료 후 정렬된 결과



그림 12-6 정렬 후 데이터

Code12-01.py 버블 정렬 구현

```
1  ## 클래스와 함수 선언 부분 ##
2  def BubbleSort(ary) :
3      n = len(ary)
4      for end in range(n-1, 0, -1) :
5          for cur in range(0, end) :
6              if (ary[cur] > ary[cur+1]) :
7                  ary[cur], ary[cur+1] = ary[cur+1], ary[cur]
8      return ary
9
10 ## 전역 변수 선언 부분 ##
11 dataAry = [188, 162, 168, 120, 50, 150, 177, 105]
12
13 ## 메인 코드 부분 ##
14 print('정렬 전 -->', dataAry)
15 dataAry = BubbleSort(dataAry)
16 print('정렬 후 -->', dataAry)
```

실행 결과

정렬 전 --> [188, 162, 168, 120, 50, 150, 177, 105]

정렬 후 --> [50, 105, 120, 150, 162, 168, 177, 188]

Section 01 고급 정렬 알고리즘의 원리와 구현

■ 버블 정렬 성능과 특이점

- 버블 정렬도 연산 수는 $O(n^2)$
- 기존에 배열이 어느 정도 정렬되어 있다면 연산 수가 급격히 줄어듦
- 데이터 1개를 제외하고 대부분 정렬되어 있는 배열의 경우



그림 12-7 대부분의 데이터가 어느 정도 정렬된 초기 상태

- 버블 정렬 방법에 따라 사이클1을 완료하면 정렬이 완료됨

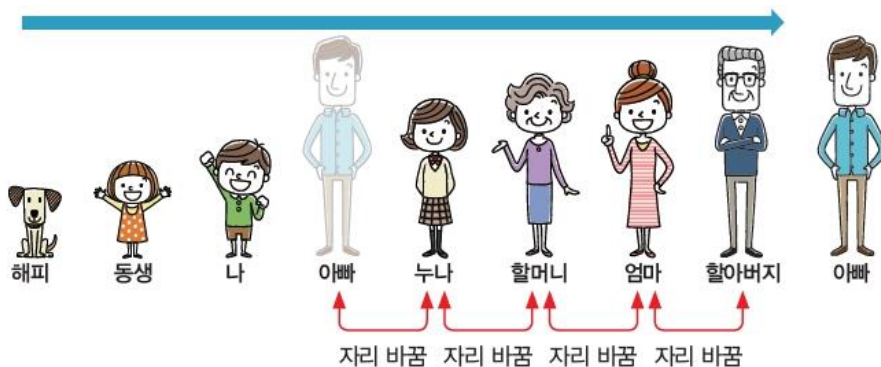


그림 12-8 사이클1의 작동과 결과(4회 자리 바꿈 발생)

Section 01 고급 정렬 알고리즘의 원리와 구현

- 이미 정렬된 상태이므로 사이클2에서는 자리 바꿈이 발생되지 않음



그림 12-9 사이클2의 작동과 결과(자리 바꿈 발생하지 않음)

Section 01 고급 정렬 알고리즘의 원리와 구현

Code12-02.py 개선된 버블 정렬의 구현

```
1  ## 클래스와 함수 선언 부분 ##
2  def bubbleSort(ary) :
3      n = len(ary)
4      for end in range(n-1, 0, -1) :
5          changeYN = False
6          print('#사이클-->', ary)
7          for cur in range(0, end) :
8              if (ary[cur] > ary[cur+1]) :
9                  ary[cur], ary[cur+1] = ary[cur+1], ary[cur]
10                 changeYN = True
11             if not changeYN :
12                 break
13         return ary
14
15  ## 전역 변수 선언 부분 ##
16  dataAry = [50, 105, 120, 188, 150, 162, 168, 177]
17
18  ## 메인 코드 부분 ##
19  print('정렬 전 -->', dataAry)
20  dataAry = bubbleSort(dataAry)
21  print('정렬 후 -->', dataAry)
```

실행 결과

```
정렬 전 --> [50, 105, 120, 188, 150, 162, 168, 177]
#사이클--> [50, 105, 120, 188, 150, 162, 168, 177]
#사이클--> [50, 105, 120, 150, 162, 168, 177, 188]
정렬 후 --> [50, 105, 120, 150, 162, 168, 177, 188]
```

Section 01 고급 정렬 알고리즘의 원리와 구현

SELF STUDY 12-1

Code12-02.py를 수정해서 랜덤하게 0과 200 사이의 숫자 20개를 생성한 후 내림차순으로 정렬하도록 코드를 작성하자. 그리고 몇 번 만에 정렬했는지 횟수를 출력하자.

실행 결과

정렬 전 → [158, 20, 119, 39, 66, 41, 124, 151, 168, 154]

정렬 후 → [20, 39, 41, 66, 119, 124, 151, 154, 158, 168]

35회로 정렬 완료

Section 01 고급 정렬 알고리즘의 원리와 구현

■ 퀵 정렬(Quick Sort)

■ 퀵 정렬의 개념

- 기준을 하나 뽑은 후 기준보다 작은 그룹과 큰 그룹을 나누어 다시 각 그룹으로 정렬하는 방법



그림 12-10 퀵 정렬의 개념

■ 퀵 정렬 구현(가족을 정렬하는 경우)

- 일반적으로 중간에 위치한 데이터를 기준으로 선정



그림 12-11 퀵 정렬 전 초기 상태

Section 01 고급 정렬 알고리즘의 원리와 구현

- 할머니를 기준으로 선정한 후 할머니보다 작은 가족은 왼쪽 그룹, 큰 가족은 오른쪽 그룹으로 보냄

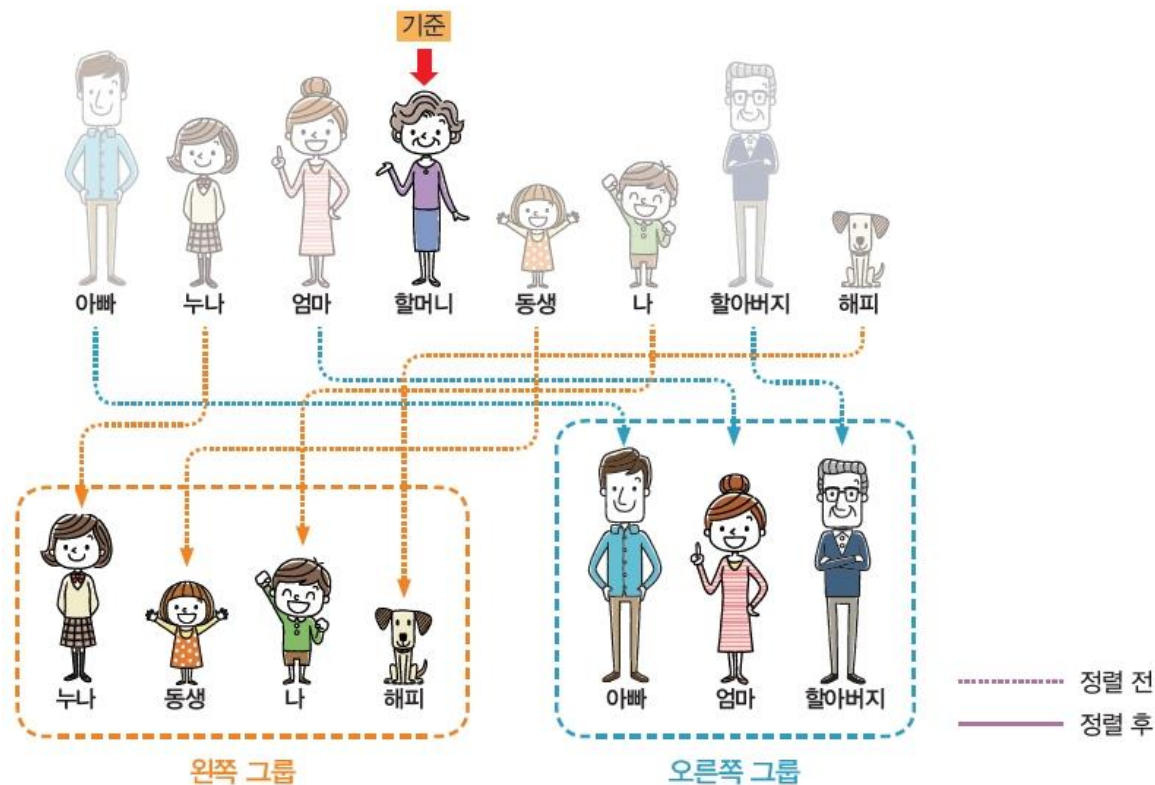


그림 12-12 퀵 정렬 1단계

Section 01 고급 정렬 알고리즘의 원리와 구현

- 나뉜 두 그룹에서 왼쪽 그룹 정렬 예



그림 12-13 퀵 정렬 2단계: 왼쪽 그룹 정렬

Section 01 고급 정렬 알고리즘의 원리와 구현

- 나뉜 두 그룹에서 오른쪽 그룹 정렬 예

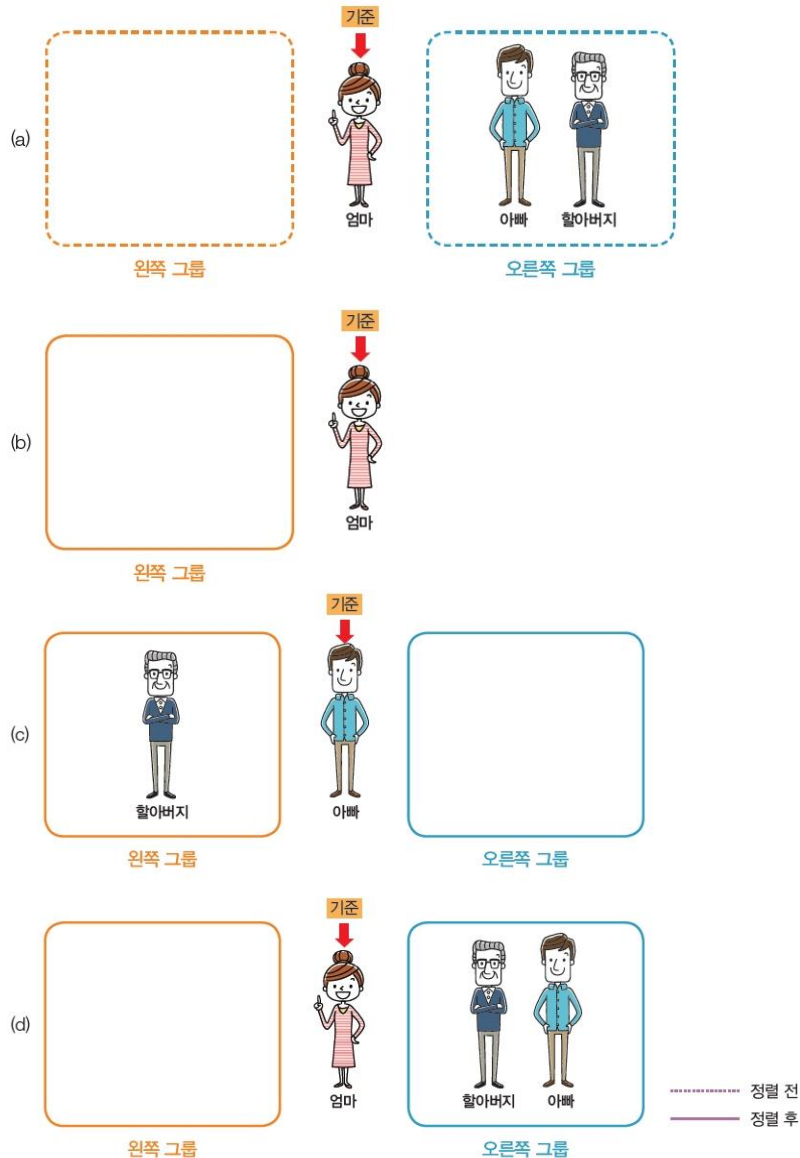


그림 12-14 퀵 정렬 3단계: 오른쪽 그룹 정렬

Section 01 고급 정렬 알고리즘의 원리와 구현

- 전체 데이터가 모두 정렬된 최종 상태

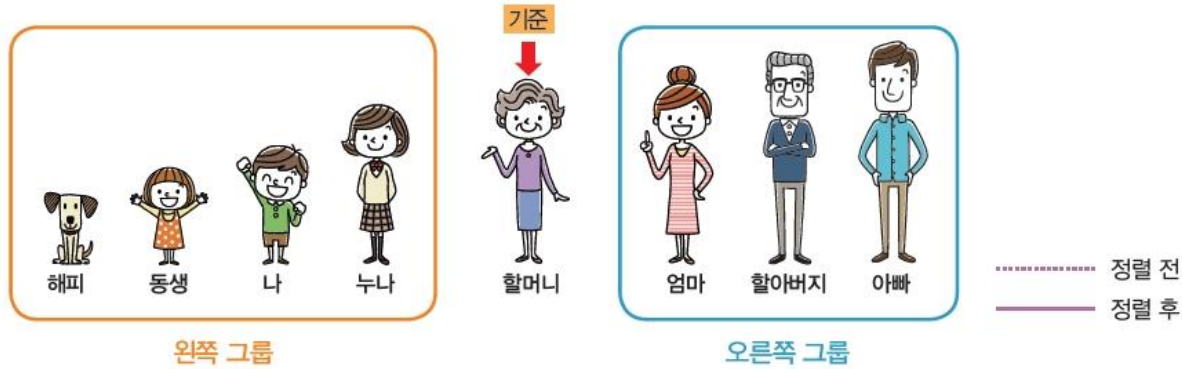


그림 12-15 퀵 정렬 완료

Section 01 고급 정렬 알고리즘의 원리와 구현

Code12-03.py 퀵 정렬의 간단한 구현

```
1  ## 클래스와 함수 선언 부분 ##
2  def quickSort(ary) :
3
4      n = len(ary)
5      if n <= 1 :                # 정렬할 리스트의 개수가 1개 이하면
6          return ary
7
8      pivot = ary[n // 2]        # 기준 값을 중간값으로 지정
9      leftAry, rightAry = [], []
10
11     for num in ary :
12         if num < pivot :
13             leftAry.append(num)
14         elif num > pivot :
15             rightAry.append(num)
16
17     return quickSort(leftAry) + [pivot] + quickSort(rightAry)
18
19 ## 전역 변수 선언 부분 ##
20 dataAry = [188, 150, 168, 162, 105, 120, 177, 50]
21
22 ## 메인 코드 부분 ##
23 print('정렬 전 -->', dataAry)
24 dataAry = quickSort(dataAry)
25 print('정렬 후 -->', dataAry)
```

실행 결과

정렬 전 --> [188, 150, 168, 162, 105, 120, 177, 50]

정렬 후 --> [50, 105, 120, 150, 162, 168, 177, 188]

Section 01 고급 정렬 알고리즘의 원리와 구현

SELF STUDY 12-2

Code12-03.py를 수정해서 랜덤하게 0과 200 사이의 숫자 20개를 생성한 후 내림차순으로 정렬하도록 코드를 작성하자. 그리고 몇 번 만에 정렬했는지 횟수를 출력하자.

실행 결과

정렬 전 --> [31, 49, 175, 33, 103, 76, 63, 151, 166, 25]

정렬 후 --> [175, 166, 151, 103, 76, 63, 49, 33, 31, 25]

25회로 정렬 완료

Section 01 고급 정렬 알고리즘의 원리와 구현

▪ 중복 값을 고려한 퀵 정렬

Code12-04.py 퀵 정렬의 간단한 구현(중복된 값을 고려)

```
1  ## 클래스와 함수 선언 부분 ##
2  def quickSort(ary) :
3      n = len(ary)
4      if n <= 1 :                # 정렬할 리스트 개수가 1개 이하이면
5          return ary
6
7      pivot = ary[n // 2]        # 기준 값을 중간 값으로 지정
8      leftAry, midAry, rightAry = [], [], []
9
10     for num in ary :
11         if num < pivot :
12             leftAry.append(num)
13         elif num > pivot :
14             rightAry.append(num)
15         else :
16             midAry.append(num)
17
18     return quickSort(leftAry) + midAry + quickSort(rightAry)
19
20 ## 전역 변수 선언 부분 ##
21 dataAry = [120, 120, 188, 150, 168, 50, 50, 162, 105, 120, 177, 50]
22
23 ## 메인 코드 부분 ##
24 print('정렬 전 -->', dataAry)
25 dataAry = quickSort(dataAry)
26 print('정렬 후 -->', dataAry)
```

실행 결과

정렬 전 --> [120, 120, 188, 150, 168, 50, 50, 162, 105, 120, 177, 50]

정렬 후 --> [50, 50, 50, 105, 120, 120, 120, 150, 162, 168, 177, 188]

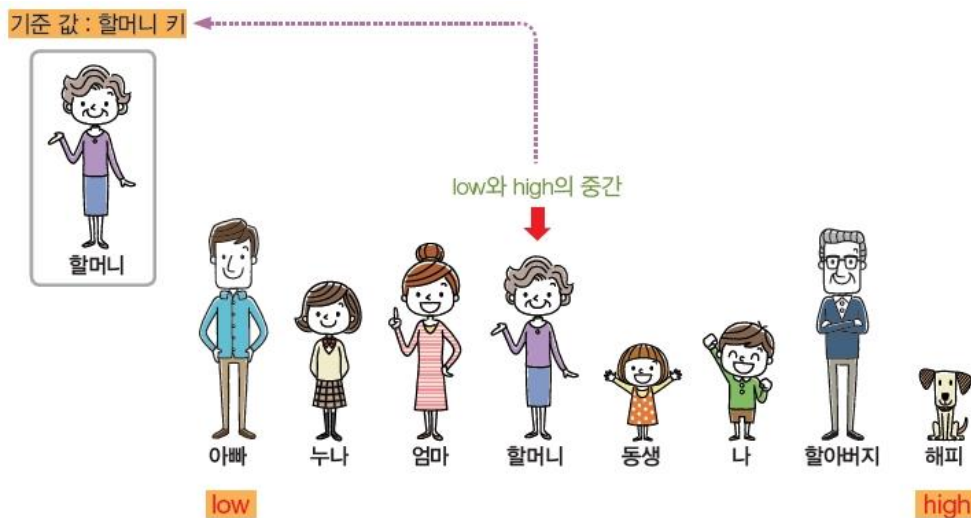
Section 01 고급 정렬 알고리즘의 원리와 구현

■ 퀵 정렬의 일반 구현

1 start(시작점)에 있는 아빠 위치를 low로, end(끝점)에 있는 해피 위치를 high로 둔다.



2 low와 high의 중간 위치에 있는 할머니 키를 기준 값으로 설정한다.



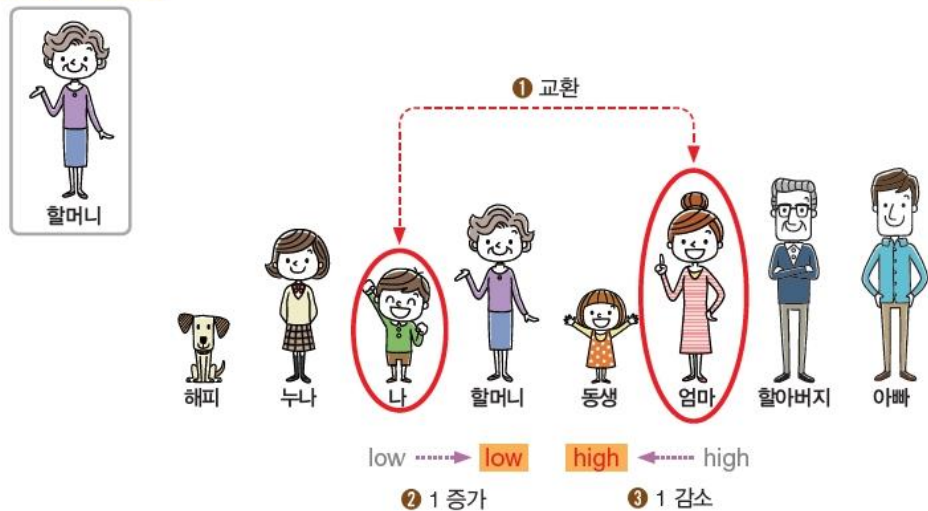
Section 01 고급 정렬 알고리즘의 원리와 구현

3 low가 high보다 크거나 같아질 때까지 1~3과 같이 반복한다.

1 기준 값 : 할머니 키

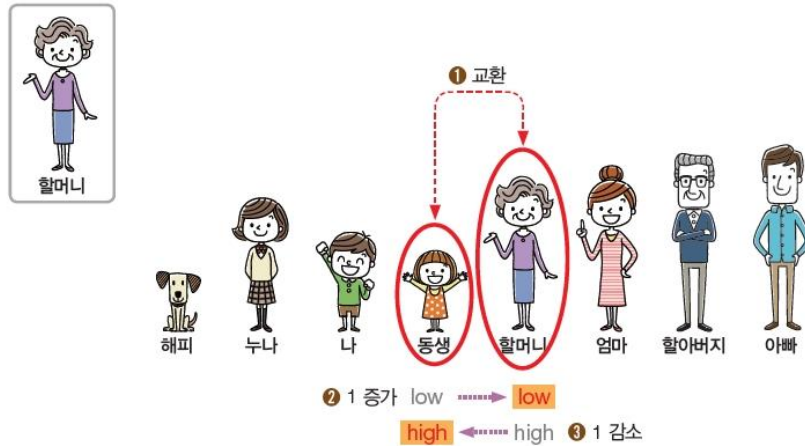


2 기준 값 : 할머니 키

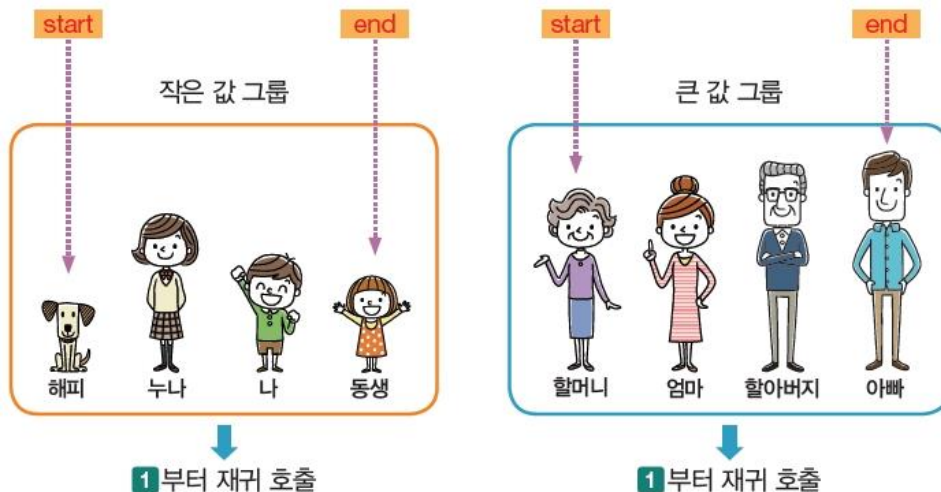


Section 01 고급 정렬 알고리즘의 원리와 구현

3 기준 값 : 할머니 키



4 low 위치를 중심 위치(mid)로 하면 왼쪽에는 작은 값들이, 오른쪽에는 큰 값들이 들어갔다. 즉, 두 그룹으로 분리되었다. 왼쪽 작은 값 그룹과 오른쪽 큰 값 그룹에 대해 1 ~ 3을 재귀적으로 진행한다.



Section 01 고급 정렬 알고리즘의 원리와 구현

Code12-05.py 퀵 정렬의 일반적인 구현

```
1  ## 클래스와 함수 선언 부분 ##
2  def qSort(arr, start, end) :
3      if end <= start :
4          return
5
6      1 { low = start
7        high = end
8
9      2 pivot = arr[(low + high) // 2] # 작은 값은 왼쪽, 큰 값은 오른쪽으로 분리
10     while low <= high :
11         while arr[low] < pivot :
12             low += 1
13         while arr[high] > pivot : 3의 1
14             high -= 1
15         if low <= high :
16             arr[low], arr[high] = arr[high], arr[low] 3의 2 및 3
17             low, high = low + 1, high - 1
18
19     { mid = low
20     4 { qSort(arr, start, mid-1)
21       qSort(arr, mid, end)
22
23
24  def quickSort(ary) :
25      qSort(ary, 0, len(ary)-1)
26
```

Section 01 고급 정렬 알고리즘의 원리와 구현

```
27 ## 전역 변수 선언 부분 ##
28 dataAry = [188, 150, 168, 162, 105, 120, 177, 50]
29
30 ## 메인 코드 부분 ##
31 print('정렬 전 -->', dataAry)
32 quickSort(dataAry)
33 print('정렬 후 -->', dataAry)
```

실행 결과

정렬 전 --> [188, 150, 168, 162, 105, 120, 177, 50]

정렬 후 --> [50, 105, 120, 150, 162, 168, 177, 188]

■ 퀵 정렬 성능과 특이점

- 퀵 정렬도 가장 나쁜 경우에는 연산 수는 $O(n^2)$ 이 되지만 평균적으로 $O(n \log n)$ 의 연산 수를 가짐
- 다른 정렬에 비해서 상당히 빠른 속도이며, 특히 정렬할 데이터 양이 많을수록 다른 정렬보다 매우 우수한 성능을 냄
- 예 : n 이 데이터 100만 개를 정렬한다면, 퀵 정렬은 $100만 * \log 100만 = \text{약 } 2000\text{만의 연산 횟수로 처리됨}$

Section 02 고급 정렬 알고리즘의 응용

■ 컬러 이미지를 흑백 이미지로 만들기

■ 컬러와 흑백 색상 표현 이해하기

- Red, Green, Blue 색이 어떻게 조합되어 표현되는지 2×2픽셀의 이미지로 가정

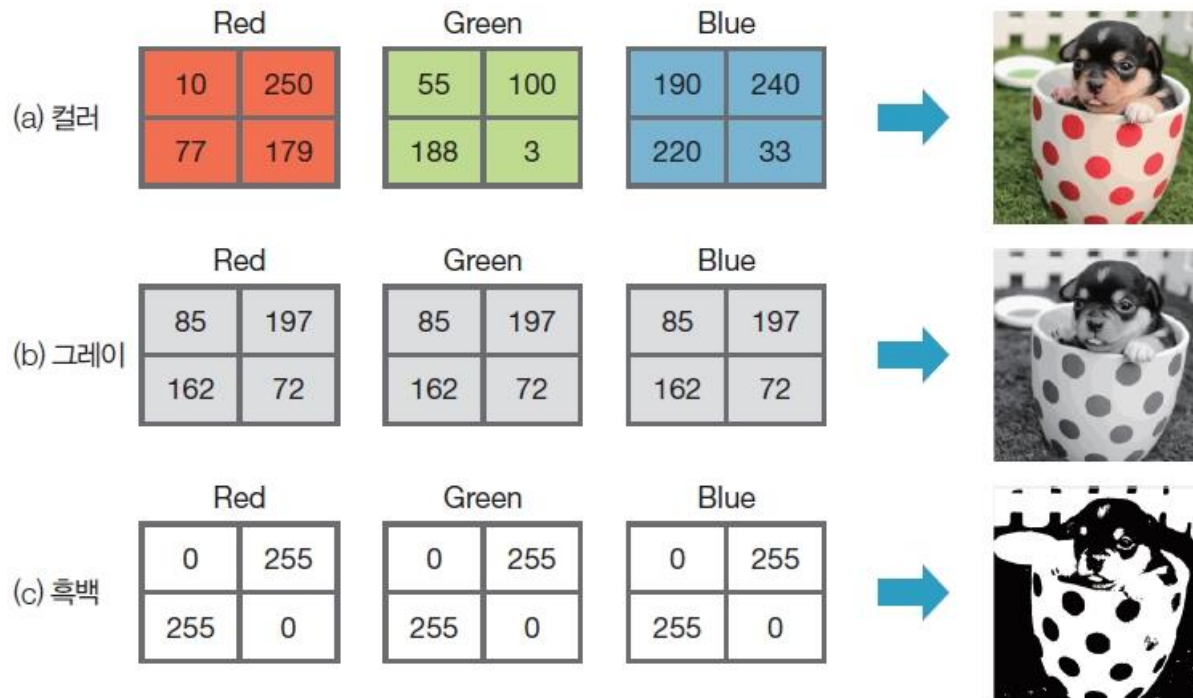


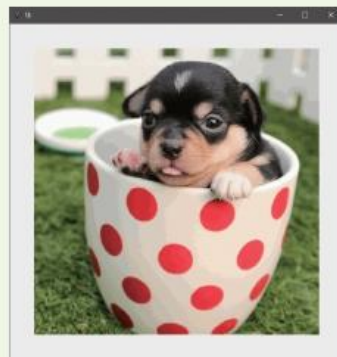
그림 12-16 컬러, 그레이, 흑백 이미지의 색상 표현

Section 02 고급 정렬 알고리즘의 응용

■ 파이썬으로 이미지 출력하기

Code12-06.py GIF 파일의 간단한 화면 출력

```
1 from tkinter import *
2
3 window = Tk()
4 window.geometry("600x600")
5
6 photo = PhotoImage(file = 'pet01.gif')
7
8 paper = Label(window, image=photo)
9 paper.pack(expand=1, anchor=CENTER)
10 window.mainloop()
```



■ 컬러 이미지를 1차원 배열로 만들기

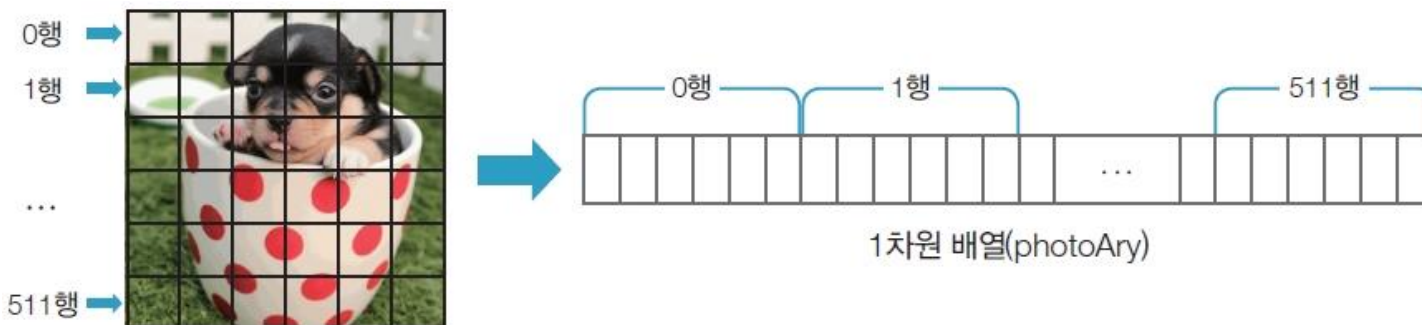


그림 12-17 2차원 이미지를 1차원 배열로 변환

Section 02 고급 정렬 알고리즘의 응용

- 1차원 배열 하나에 저장할 때는 세 점을 합한 평균으로 저장함

```
❶ r, g, b = photo.get(i, k)
❷ value = (r + g + b) // 3
❸ photoAry.append(value)
```

Code12-07.py GIF 파일을 1차원 배열로 저장

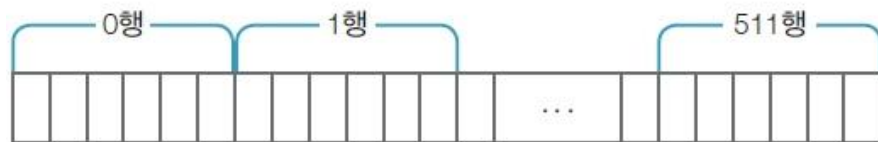
```
1  from tkinter import *
2
3  window = Tk()
4  window.geometry("600x600")
5  photo = PhotoImage(file = 'pet01.gif')
6
7  photoAry = []
8  h = photo.height()
9  w = photo.width()
10 for i in range(h) :
11     for k in range(w) :
12         r, g, b = photo.get(i, k)
13         value = (r + g + b) // 3
14         photoAry.append(value)
15
16 # 이 부분에 필요한 내용을 추가
17
18
19 paper = Label(window, image=photo)
20 paper.pack(expand=1, anchor=CENTER)
21 window.mainloop()
```



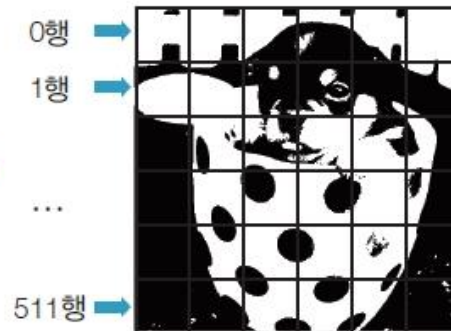
Section 02 고급 정렬 알고리즘의 응용

■ 흑백 이미지로 만들기

```
if photoAry[i] <= 127 :  
    photoAry[i] = 0  
else :  
    photoAry[i] = 255
```



0과 255 값으로만 구성된 1차원 배열



512×512 픽셀의 그림

그림 12-18 1차원 배열을 2차원 이미지로 전환

Section 02 고급 정렬 알고리즘의 응용

Code12-08.py 1차원 배열을 흑백 값으로 변환 후 화면 출력

```
... # 생략(Code12-07.py의 1~14행과 동일)
15
16 for i in range(len(photoAry)) :
17     if photoAry[i] <= 127 :
18         photoAry[i] = 0
19     else :
20         photoAry[i] = 255
21
22 pos = 0
23 for i in range(h) :
24     for k in range(w) :
25         r = g = b = photoAry[pos]
26         pos += 1
27         photo.put("#%02x%02x%02x" % (r, g, b), (i, k))
28
29 paper = Label(window, image=photo)
30 paper.pack(expand=1, anchor=CENTER)
31 window.mainloop()
```



Section 02 고급 정렬 알고리즘의 응용

- 퀵 정렬로 중앙값 계산하기
 - 어두운 이미지를 기준 값 127로 정해서 변환한 예

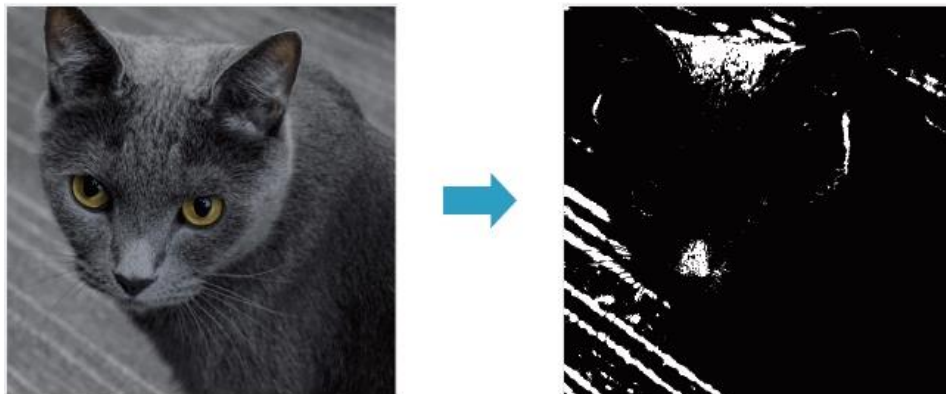


그림 12-19 어두운 이미지를 127을 기준으로 흑백으로 변환한 예

Code12-09.py 퀵 정렬을 이용한 중앙값을 찾아서 처리

```
1 from tkinter import *
2
3 ## 클래스와 함수 선언 부분 ##
4 def qSort(arr, start, end) :
5     if end <= start :
6         return
7
8     low = start
9     high = end
10
```



Section 02 고급 정렬 알고리즘의 응용

```
11 pivot = arr[(low + high) // 2]      # 작은 값은 왼쪽, 큰 값은 오른쪽으로 분리
12 while low <= high :
13     while arr[low] < pivot :
14         low += 1
15     while arr[high] > pivot :
16         high -= 1
17     if low <= high :
18         arr[low], arr[high] = arr[high], arr[low]
19         low, high = low + 1, high - 1
20
21 mid = low
22
23 qSort(arr, start, mid-1)
24 qSort(arr, mid, end)
25
26 def quickSort(ary) :
27     qSort(ary, 0, len(ary)-1)
28
29 ## 메인 코드 부분 ##
30 window = Tk()
31 window.geometry("600x600")
32 photo = PhotoImage(file = 'pet02.gif')
33
34 photoAry = []
35 h = photo.height()
36 w = photo.width()
37 for i in range(h) :
38     for k in range(w) :
39         r, g, b = photo.get(i, k)
```

Section 02 고급 정렬 알고리즘의 응용

```
40     value = (r + g + b) // 3
41     photoAry.append(value)
42
43     dataAry = photoAry[:]
44     quickSort(dataAry)
45     midValue = dataAry[h*w // 2]
46
47     for i in range(len(photoAry)) :
48         if photoAry[i] <= midValue :
49             photoAry[i] = 0
50         else :
51             photoAry[i] = 255
52
53     pos = 0
54     for i in range(h) :
55         for k in range(w) :
56             r = g = b = photoAry[pos]
57             pos += 1
58             photo.put("#%02x%02x%02x" % (r, g, b), (i, k))
59
60     paper = Label(window, image=photo)
61     paper.pack(expand=1, anchor=CENTER)
62     window.mainloop()
```

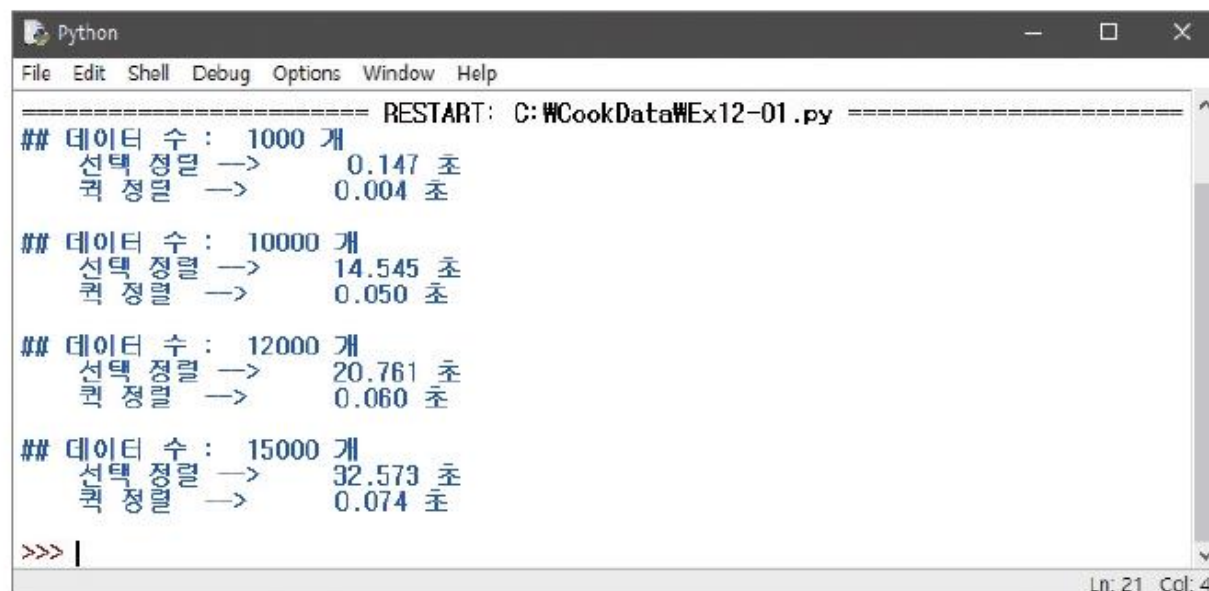
응용예제 01 선택 정렬과 퀵 정렬의 성능 비교하기

난이도 ★☆☆☆☆

예제 설명

선택 정렬은 $O(n^2)$ 의 연산 횟수를, 퀵 정렬은 평균 $O(n \log n)$ 의 연산 횟수를 갖는다. 정렬할 데이터양에 따라서 두 정렬 방식의 시간 차이를 비교해 본다. 데이터는 1000, 10000, 12000, 15000개를 정렬한다. 실행 결과는 컴퓨터의 성능에 따라서 달리 나올 수 있지만, 선택 정렬은 개수가 많아질수록 시간이 급격히 증가하는 것은 동일하게 확인할 수 있다. 퀵 정렬은 개수가 많아져도 짧은 시간에 정렬 가능하다.

실행 결과



```
Python
File Edit Shell Debug Options Window Help
===== RESTART: C:\WookData\WEx12-01.py =====
## 데이터 수 : 1000 개
  선택 정렬 -> 0.147 초
  퀵 정렬 -> 0.004 초

## 데이터 수 : 10000 개
  선택 정렬 -> 14.545 초
  퀵 정렬 -> 0.050 초

## 데이터 수 : 12000 개
  선택 정렬 -> 20.761 초
  퀵 정렬 -> 0.060 초

## 데이터 수 : 15000 개
  선택 정렬 -> 32.573 초
  퀵 정렬 -> 0.074 초

>>> |
```

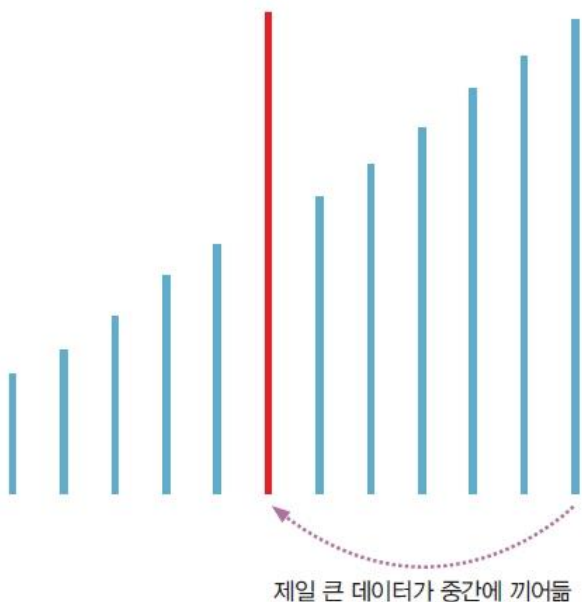
Ln: 21 Col: 4

응용예제 02 이미 정렬된 줄에 끼어들기

난이도 ★★☆☆☆

예제 설명

배열이 이미 정렬된 상태에서 맨 마지막 값이 아무 위치에나 끼어들 때, 다시 정렬하는 효율적인 방법을 생각해 보자. 일반적인 상태에서 성능이 뛰어난 퀵 정렬과 대부분의 데이터가 정렬된 상태에서 효율적인 버블 정렬을 사용하여 이런 특수한 상태에 더 효율적인 알고리즘을 예상해 본다.



실행 결과

```
Python
File Edit Shell Debug Options Window Help
===== RESTART: C:\CookData\WEx12-02.py =====
# 데이터 개수 --> 1000000
# 끼어들 위치 --> 994718
다시 정렬 시간(버블 정렬) --> 0.944 초
다시 정렬 시간(퀵 정렬) --> 5.818 초
>>>
```