

재귀와 반복 - 정렬

Recursion & Iteration - Sorting

컴퓨터과학의 고전적인 문제 중의 하나인 정렬(sorting)은 순서를 매길 수 있는 데이터를 정한 순서로 나열하는 문제이다. 정렬 문제가 초기에서부터 주목을 받은 이유는 정렬된 데이터의 처리가 그렇지 않은 데이터의 처리보다 훨씬 더 쉽고 효율적이기 때문이다. 그래서 오래전부터 정렬문제를 푸는 다양한 알고리즘이 고안되어 사용되고 있다.

이 장에서는 복수의 데이터를 나란히 모아서 보존할 수 있는 구조인 순서열의 표현방식과 연산, 의미를 먼저 공부한다. 정렬할 데이터는 순서열의 일종인 리스트로 모아 쓰지만, 이 기회에 다른 종류의 순서열인 튜플 및 문자열과 비교하면서 차이점도 함께 익힌다. 그리고 널리 알려진 상이한 정렬 알고리즘을 몇 가지 공부하고, 임의 원소의 리스트(예: [3,5,4,2])를 인수로 받아서 오름차순으로 정렬한 리스트(예: [2,3,4,5])를 결과로 내주는 함수를 알고리즘 별로 작성한다.

1. 순서열

순서열(sequence)은 여러 개의 데이터를 순서있게 나열해 모아놓은 데이터 구조를 통칭한다. Python 언어에서 기본적으로 제공하는 순서열은 리스트list, 튜플tuple, 정수범위range가 있는데, The Python Standard Library의 4.6. Sequence Types에 문법과 의미가 자세히 설명되어 있다. 이미 배운 문자열string도 순서열의 일종으로 문자를 차례로 나열한 문자순서열이다. 순서열은 문자열과 같은 방식으로 위치번호index가 지정되어 있다. 순서열은 크게 수정가능mutable 순서열과 수정불가능immutable 순서열로 나눌 수 있다. 리스트는 수정가능 순서열이고, 튜플과 정수범위는 수정불가능 순서열이다. (문자열은 수정불가능하다고 이미 공부했다.) 이 절에서는 Python 순서열의 표현방식과 공통점, 차이점을 알아보고, 공통으로 사용하는 순서열 연산을 공부한다.

리스트

리스트list는 유일하게 수정가능한 순서열이다. 주로 같은 타입의 원소를 나열하는데 많이 쓰지만, 굳이 원소가 모두 같을 필요는 없다. 그러면 리스트의 표현방식과 연산을 공부해보자.

리스트는 다음과 같이 대괄호로 둘러싸아 표현한다. 리스트가 만들어지면 오른쪽 아래 그림과 같은 형식의 메모리셀에 원소가 차례대로 저장된 리스트가 메모리에 생긴다고 상상하면 된다.

```
>>> odds = [1,3,5,7,9]
```

위와 같이 이름을 지정하면 다음과 같이 이름을 불러 참조할 수 있다.

```
>>> odds
[1, 3, 5, 7, 9]
```



문자열과 마찬가지로 0으로 시작하는 위치번호가 지정되어 있으며 다음과 같이 위치번호로 해당 원소를 볼 수 있다.

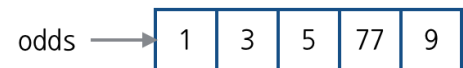
```
>>> odds[1]
3
>>> odds[-2]
7
>>> odds[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

위치번호가 범위를 벗어나면 `IndexError` 오류가 발생하므로 참조 범위를 벗어나면 참조하지 못하도록 사전에 막을 수 있어야 한다.

리스트는 수정가능하므로 다음과 같이 지정문으로 특정 위치의 원소를 수정할 수 있다.

```
>>> odds[3] = 77
```

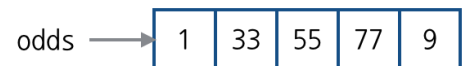
실행 결과 메모리의 상태는 오른쪽과 같다.



7은 지워지고 그 자리에 77이 저장되었다.

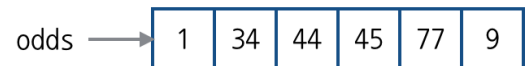
범위를 지정하여 수정할 수도 있다. 다음 지정문의 실행 결과 메모리의 상태는 오른쪽과 같다.

```
>>> odds[1:3] = [33,55]
```



수정하는 범위의 길이가 같은 필요는 없다. 다음 지정문의 실행 결과 메모리의 상태는 오른쪽과 같다.

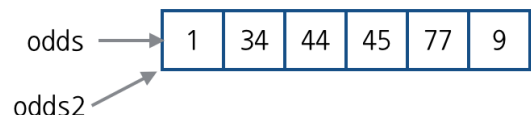
```
>>> odds[1:3] = [34,44,45]
```



다음과 같이 이미 있는 리스트에 다른 이름을 붙일 수 있다.

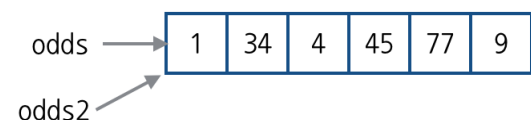
```
>>> odds2 = odds
```

이 지정문의 실행결과와 오른쪽과 같다. 두 이름이 같은 리스트를 가리킨다. 따라서 둘 중 하나의 이름으로 수정을 하



고, 다른 이름이 가리키는 리스트는 바뀌지 않았으리라는 착각을 하면 곤란하다. 즉, 다음과 같이 실행해 보면 이 현상을 관찰할 수 있다.

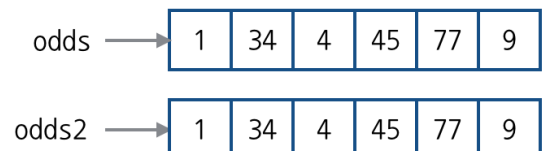
```
>>> odds2
[1, 34, 44, 45, 77, 9]
>>> odds2[2] = 4
>>> odds2
[1, 34, 4, 45, 77, 9]
>>> odds
[1, 34, 4, 45, 77, 9]
```



리스트를 복제하려면 다음과 같이 해야 한다.

```
>>> odds2 = odds[:]
```

그러면 오른쪽과 같이 된다.



튜플

이미 공부한 적인 있는 순서열인 튜플tuple은 다음과 같이 괄호로 둘러싸아 표현한다. 튜플은 일단 만들어 놓으면 내부의 내용을 수정불가능immutable한 점이 리스트와 다르다.

```
>>> t = ('컴퓨터과학', 1, '짱')
```

```
>>> t[2]
```

```
'짱'
```

```
>>> t[:2]
```

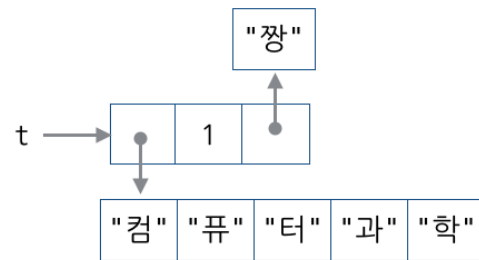
```
('컴퓨터과학', 1)
```

```
>>> t[2] = '짱'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```



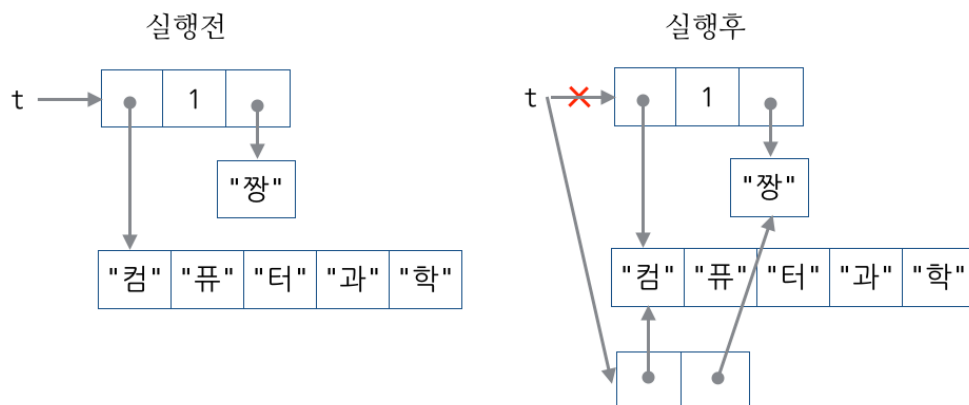
튜플은 수정불가능하므로 지정문을 써서 수정하려 하면 오류가 발생한다.

다음과 같이 하면 수정한 것 같아 보이지만, 사실은 수정한 것이 아니라 아래 그림에서 볼 수 있듯이 원래 튜플의 일부를 참조하여 새 튜플을 만들고 같은 이름을 붙인 것 뿐이다.

```
>>> t = t[:1] + t[2:]
```

```
>>> t
```

```
('컴퓨터과학', '짱')
```



이전에 `t`가 가리키던 튜플은 지워진 것이 아니고, 위의 지정문이 실행된 이후 변수 `t`는 새로 만든 튜플을 가리키므로 더 이상 접근할 수 없을 뿐이다. 이와 같은 성질때문에 수정이 잦은 데이터는 리스트에 보관하는 것이 효율적일 수 있다.

문자열

앞에서 공부한 문자열string도 튜플과 마찬가지로 일단 만들면 수정불가능immutable하다.

```
>>> s = "컴퓨터과학"
```

```
>>> s
```

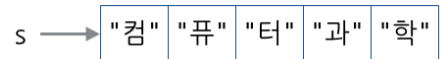
```
'컴퓨터과학'
```

```
>>> s[3] = '공'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item assignment
```

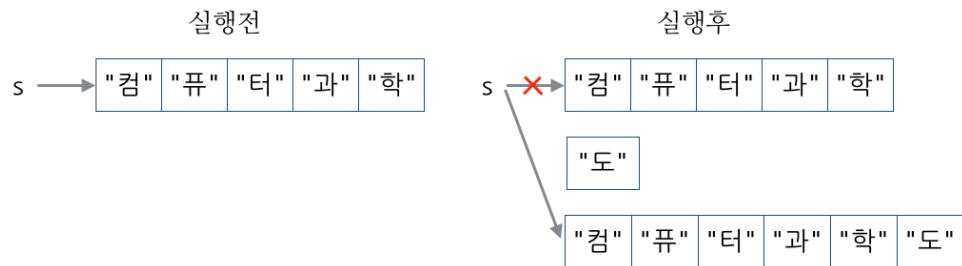


다음과 같이 하면 수정한 것 같아 보이지만, 사실은 수정한 것이 아니라 내부적으로는 아래 그림과 같이 새로 문자열을 만들고 같은 이름을 붙인 것 뿐임을 명심하자.

```
>>> s = s + '도'
```

```
>>> s
```

```
'컴퓨터과학도'
```



다시말하면, 여기서도
이전에 `s`가 가리키던 문자열은 지워진 것이 아닙니다.

위의 지정문이 실행된 이후 변수 `s`는 새로 만든 문자열을 가리킨다.

정수범위

정수범위range는 정수가 일정 간격으로 나열된 순서열이다. `range(n)`은 정수 0부터 `n-1`까지의 간격 1의 정수범위 순서열을 나타낸다. 즉, `range(5)`은 0, 1, 2, 3, 4 순서열을 나타낸다.

```
>>> r = range(5)
```

```
>>> r
```

```
range(0, 5)
```

```
>>> r[2]
```

```
2
```

정수범위도 수정불가능하다.

```
>>> r[2] = 3
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'range' object does not support item assignment
```

특정 정수가 정수범위에 있는지는 다음과 같이 `in` 논리연산으로 검사할 수 있다.

```
>>> 3 in r
```

```
True
```

```
>>> 5 in r
```

```
False
```

`range(m,n)`은 정수 `m`부터 `n-1`까지의 간격 1의 정수범위 순서열을 나타낸다. 즉, `range(3,10)`은 3, 4, 5, 6, 7, 8, 9 순서열을 나타낸다.

```
>>> r = range(3,10)
>>> r
range(3, 10)
>>> r[1]
4
>>> 6 in r
True
>>> 2 in r
False
```

정수범위 시퀀스의 간격을 1이 아닌 정수로 지정하려면 셋째 인수를 추가한다. 즉, `range(m,n,k)`는 정수 `m`부터 `n-1`까지의 간격 `k`의 정수범위 순서열을 나타낸다. `range(3,11,2)`은 3, 5, 7, 9를 나타내고, `range(10,3,-1)`은 10, 9, 8, 7, 6, 5, 4를 나타낸다.

```
>>> r = range(3,11,2)
>>> r[2]
7
>>> 7 in r
True
>>> 6 in r
False
```

정수범위 시퀀스는 `for` 반복문에서 많이 사용한다. 이제 `for` 반복문을 공부해보자.

반복문 : for

지금까지 공부한 프로그램에서 반복 계산이 필요한 경우 모두 `while` 문을 사용했다. 길이가 고정된 문자열을 구성하는 문자를 하나씩 차례로 반복 계산하는 경우 `for` 문을 사용하면 간편하다.

`for` 문의 문법과 의미는 다음과 같다.

문법	<code>for <변수이름> in <순서열> :</code> <몸체>
의미	<변수이름>을 <code>x</code> 라고 하고, <순서열>을 <code>s</code> 라고 하면, 다음은 차례로 실행한다. <code>s[0]</code> 를 변수 <code>x</code> 로 지정하고 <몸체>를 실행한다. <code>s[1]</code> 를 변수 <code>x</code> 로 지정하고 <몸체>를 실행한다. <code>s[2]</code> 를 변수 <code>x</code> 로 지정하고 <몸체>를 실행한다. ... <code>s[len(s)-1]</code> 를 변수 <code>x</code> 로 지정하고 <몸체>를 실행한다.

반복하는 동안 매번 x 변수의 값으로 순서열의 원소가 앞에서부터 하나씩 차례로 지정된다. 이와 같은 형태의 for 반복문에서 <몸체>는 정확하게 <순서열>의 길이의 횟수만큼 기본적으로 반복한다. 그렇지만 도중에 return 문을 만나거나 break 문을 만나 도중에 반복을 중단하고 빠져나갈 수도 있다.

다음 코드를 이해한 다음, 실행하여 이해한 대로 결과가 나타나는지 확인하자.

```
1 for i in range(5):
2     print(i)
```

```
1 for i in range(3,10):
2     print(i)
```

```
1 for i in range(10,3,-3):
2     for j in range(3,11,3):
3         print(i,j)
```

연습문제

앞 장에서 (지난 주에) 공부한 아래 함수의 while 반복문을 for 반복문으로 재작성해보자.

1. countdown
2. sigma2
3. sumrange
4. fact
5. power
6. mult

순서열 연산 (공용)

다음은 모든 순서열에 공통으로 사용하는 연산자를 모아놓은 것이다. 하나씩 의미를 이해해보자. 여기서 s 는 순서열, x 는 원소, i, j, k 는 위치번호, n 은 정수를 나타낸다.

연산	의미	예외
$x \text{ in } s$	x 가 s 에 있으면 True, 없으면 False	
$x \text{ not in } s$	x 가 s 에 없으면 True, 있으면 False	
$s[i]$	s 의 i 위치에 있는 원소	
$s[i:j]$	i 위치에서 j 위치까지의 순서열 조각 (i 위치의 원소 포함, j 위치의 원소 제외)	
$s[i:j:k]$	i 위치에서 j 위치까지의 k 간격으로 띄운 순서열 조각 (i 위치의 원소 포함, j 위치의 원소 제외)	
$\text{len}(s)$	s 의 길이	
$\text{min}(s)$	s 에서 가장 작은 원소	
$\text{max}(s)$	s 에서 가장 큰 원소	
$s.\text{index}(x)$	s 에서 가장 앞에 나오는 원소 x 의 위치번호	
$s.\text{index}(x,i)$	s 의 i 위치에서 시작하여 가장 앞에 나오는 원소 x 의 위치번호	

연산	의미	예외
<code>s.index(x,i,j)</code>	s의 i 위치로부터 j 위치 전까지에서 가장 앞에 나오는 원소 x의 위치번호	
<code>s.count(x)</code>	s에서 원소 x의 빈도수	
<code>s + t</code>	s와 t 나란히 붙이기	정수범위 순서 열에서는 적용 불가
<code>s * n</code>	s를 n번 반복하여 나란히 붙이기	
<code>n * s</code>	s를 n번 반복하여 나란히 붙이기	

리스트의 귀납 정의

리스트도 앞에서 공부한 자연수와 비슷하게 귀납법으로 다음과 같이 정의할 수 있다.

(1)	기초 basis	빈리스트 []은 리스트이다.
(2)	귀납 induction	x가 임의의 원소이고 L이 임의의 리스트이면, [x] + L 도 리스트이다. 여기서, x는 <u>선두원소</u> head, L은 <u>후미리스트</u> tail라고 한다.
(3)	그 외에 다른 리스트는 없다.	

자연수와 유사한 방식으로, 리스트를 귀납 구조에 맞추어 빈리스트(기초)와 그렇지 않은 리스트(귀납)와 같이 둘로 나누어 사고할 수 있다. 빈 리스트는 종료조건과 관련이 있고, 비어 있지 않은 리스트는 반복 조건과 관련이 있다. 비어 있지 않은 리스트는 선두원소와 후미리스트로 쪼개서 사고한다. 앞으로 리스트를 다루는 재귀 함수는 모두 이 구조를 이용하니 이 귀납구조를 확실히 숙지하고 넘어가자.

2. 선택정렬 Selection Sort

알고리즘

정수 리스트 s를 정렬하려면

- (반복조건) $s \neq []$,
 - s에서 가장 작은 수를 찾아서 `smallest`라고 하고, 이를 s에서 제거
 - s를 정렬
 - `smallest`와 s를 차례로 나란히 붙여서 내줌
- (종료조건) $s == []$, 정렬할 필요가 없으므로 []를 그대로 내줌

구현

리스트 s에서 가장 작은 수를 찾기 위해서 순서열 연산 `min(s)`를 사용한다. 그리고 리스트 s에서 원소 x를 하나 찾아 제거하려면 표준 라이브러리에서 제공하는 리스트의 메소드 `remove`를 사용한다.

연산	의미	비고
<code>s.remove(x)</code>	s에서 가장 앞에 나오는 원소 x를 제거한다.	x가 s에 없으면 <code>ValueError</code> 가 발생한다.

이 알고리즘은 다음과 같이 재귀함수로 정의할 수 있다.

```

1 def ssort0(s):
2     if s != []:
3         smallest = min(s)
4         s.remove(smallest)
5         return [smallest] + ssort0(s)
6     else:
7         return []

```

예제를 가지고 실행추적하여 정렬이 어떻게 하는지 살펴보자.

```

ssort0([3,5,4,2])
=> [2] + ssort0([3,5,4])
=> [2] + [3] + ssort0([5,4])
=> [2] + [3] + [4] + ssort0([5])
=> [2] + [3] + [4] + [5] + ssort0([])
=> [2] + [3] + [4] + [5] + []
= [2,3,4,5]

```

주의

`s.remove(x)`를 호출하면, 가장 앞에 나오는 원소 `x`가 제거된 새로운 리스트를 만들고 이를 변수 `s`가 가리킨다. 그런데 호출 결과로 제거된 리스트를 리턴해주는 건 아님을 주의하자. 즉 다음과 같이 코딩하면 오류이다.

```

1 def ssort0err(s):
2     if s != []:
3         smallest = min(s)
4         return [smallest] + ssort0err(s.remove(smallest))
5     else:
6         return []

```

위에서 짰 재귀 함수를 다음 틀에 맞춰 꼬리재귀 형태로 변환해보자.

```

1 def ssort1(s):
2     def loop(s,ss):
3         if s != []:
4             smallest = min(s)
5             s.remove(smallest)
6             return loop(s,ss+[smallest])
7         else:
8             return ss
9     return loop(s,[])

```

여기서도 `ss`는 가장 작은 정수부터 지금까지 정렬이 완료된 리스트를 가지고 다니는 추가 파라미터이다. `loop` 재귀호출을 할 때마다 `s`에서 가장 작은 수는 제거되고, 그 수는 `ss`의 맨 뒤에 추가된다. 리스트

의 맨 뒤에 원소를 추가하는 기능은 리스트의 메소드 `append`로 다음 표와 같이 표준 라이브러리에 이미 준비되어 있어 그냥 사용하면 된다.

연산	의미	비고
<code>s.append(x)</code>	s의 맨 뒤에 x를 붙인다.	

`append`를 써서 위 꼬리재귀 함수를 다시 짜면 다음과 같다.

```

1 def ssort2(s):
2     def loop(s,ss):
3         if s != []:
4             smallest = min(s)
5             s.remove(smallest)
6             ss.append(smallest)
7             return loop(s,ss)
8         else:
9             return ss
10    return loop(s,[])

```

`while` 반복문으로 정리하면 최종 코드는 다음과 같다.

```

1 def ssort3(s):
2     ss = []
3     while s != []:
4         smallest = min(s)
5         s.remove(smallest)
6         ss.append(smallest)
7     return ss

```

3. 삽입정렬 Insertion Sort

알고리즘

정수 리스트 s 를 정렬하려면

- (반복조건) $s \neq []$,
 - s 의 후미리스트인 $s[1:]$ 를 정렬하여 ss 라고 함
 - 정렬된 리스트 ss 의 적당한 위치에 s 의 선두원소인 $s[0]$ 를 끼워서 내줌
- (종료조건) $s = []$, 정렬할 필요가 없으므로 그대로 내줌

구현

정렬된 리스트 ss 의 적절한 위치에 정수 x 를 끼워서 정렬된 리스트를 내주는 함수 $\text{insert}(x, ss)$ 가 있다고 가정하면, 이 알고리즘은 다음과 같이 재귀함수로 정의할 수 있다.

```

1 def isort0(s):
2     if s != []:
3         return insert(s[0], isort0(s[1:]))
4     else:
5         return []

```

insert 함수가 잘 작동한다고 가정하고, 위의 예제를 가지고 실행추적해보면 다음과 같다.

```

isort0([3,5,4,2])
=> insert(3, isort0([5,4,2]))
=> insert(3, insert(5, isort0([4,2])))
=> insert(3, insert(5, insert(4, isort0([2])))
=> insert(3, insert(5, insert(4, insert(2, isort0([]))))
=> insert(3, insert(5, insert(4, insert(2, []))))
=> insert(3, insert(5, insert(4, [2])))
=> insert(3, insert(5, [2,4]))
=> insert(3, [2,4,5])
=> [2,3,4,5]

```

그러면 이제 insert 함수를 구현해보자.

정렬된 리스트에 임의의 원소 제자리에 끼워넣기

정수 x 와 정렬된 정수 리스트 ss 를 인수로 받아 x 를 ss 의 제 위치에 끼워넣어 정렬된 리스트를 내주는 함수 insert 는 다음과 같이 실행되어야 한다.

```

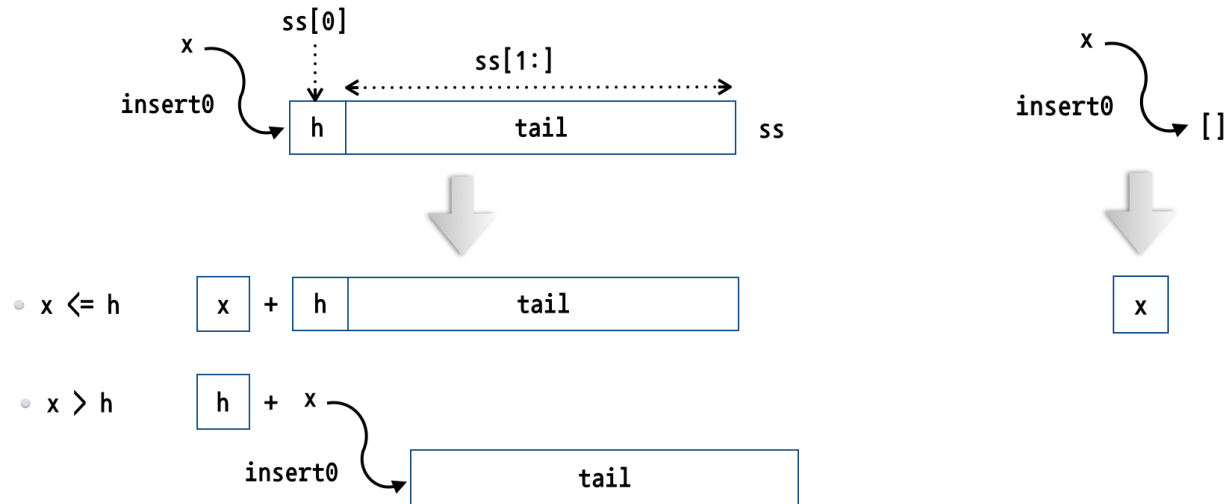
insert(1, [2,4,5,7,8]) => [1, 2, 4, 5, 7, 8]
insert(6, [2,4,5,7,8]) => [2, 4, 5, 6, 7, 8]
insert(9, [2,4,5,7,8]) => [2, 4, 5, 7, 8, 9]

```

재귀 함수 insert0 부터 짜보자.

재귀 함수 설계 전략 (사례를 통한 구상)

• 정렬된 리스트 ss 의 선두원소보다 작은 수 끼워넣기	$insert0(1,[2,4,5]) \Rightarrow [1] + [2,4,5]$
• ss 의 선두원소보다 큰 수 끼워넣기	$insert0(3,[2,4,5]) \Rightarrow [2] + insert0(3,[4,5])$
• 빈 리스트에 끼워넣기	$insert0(1,[]) \Rightarrow [1]$

 $insert0(x, ss)$ 알고리즘

정수 x 를 정렬된 리스트 ss 의 제 위치에 끼워넣으려면 다음과 같이 2가지 경우로 나누어 처리한다.

- ss 가 빈 리스트가 아닌 경우
 - ss 를 선두원소 $ss[0]$ 와 후미리스트 $ss[1:]$ 로 나눈다.
 - x 가 선두원소 $ss[0]$ 보다 작거나 같으면, x 를 ss 의 앞에 붙인다.
 - x 가 선두원소 $ss[0]$ 보다 크면, x 를 후미리스트 $ss[1:]$ 의 제 위치에 끼워넣고 그 앞에 선두원소 $ss[0]$ 를 붙인다.
- ss 가 빈 리스트인 경우, 그냥 x 만 가지고 리스트를 만든다.

다음 $insert0$ 함수 호출의 실행추적을 각각 따라가면서 알고리즘을 이해하자.

```
insert0(1,[2,4,5,7,8])
=> [1, 2, 4, 5, 7, 8]
```

```
insert0(6,[2,4,5,7,8])
=> [2] + insert0(6,[4,5,7,8])
=> [2] + [4] + insert0(6,[5,7,8])
=> [2] + [4] + [5] + insert0(6,[7,8])
=> [2] + [4] + [5] + [6] + [7,8]
=> ...
=> [2,4,5,6,7,8]
```

```
insert0(9,[])
=> [9]
```

4. 합병정렬 Merge Sort

알고리즘

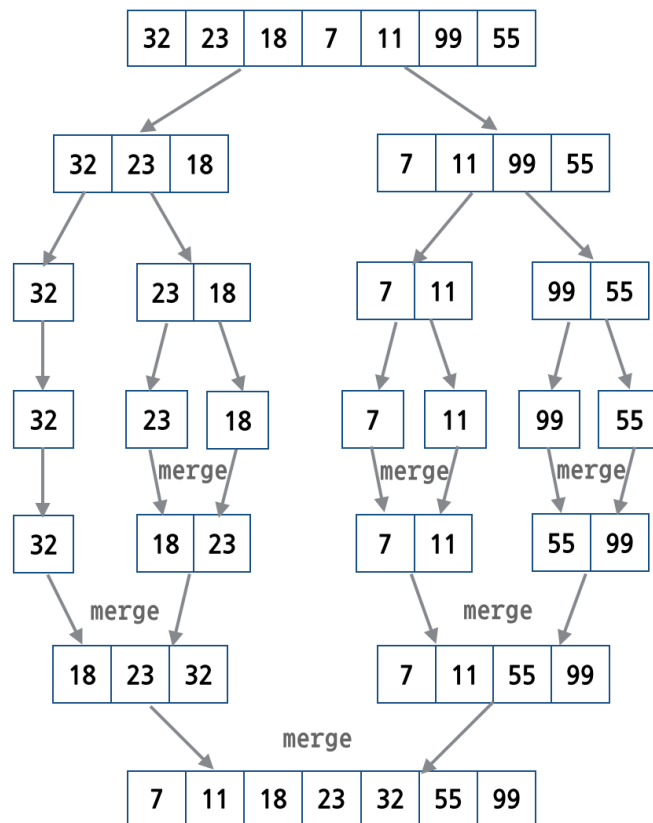
정수 리스트 s 를 정렬하려면,

- $\text{len}(s) > 1$, s 를 반으로 쪼개서, 따로 재귀로 정렬을 완료하고, 두 정렬된 리스트를 앞에서부터 차례로 훑어가며 가장 작은 수를 먼저 선택하는 방식으로 하나로 합병(merge)한다.
- $\text{len}(s) \leq 1$, 정렬할 필요가 없으므로 그대로 내준다.

예를 들어, $[32, 55, 18, 7, 11, 99, 23]$ 을 합병정렬 알고리즘으로 정렬하면 다음의 절차를 거친다.

- $[32, 55, 18]$ 과 $[7, 11, 99, 23]$ 으로 반으로 쪼개서, 각각 합병정렬한다.
- 정렬된 두 리스트 $[18, 32, 55]$ 와 $[7, 11, 23, 99]$ 를 앞에서부터 차례로 훑어 가며 가장 작은 수를 먼저 선택하는 방식으로 하나로 합병하여 $[7, 11, 18, 23, 32, 55, 99]$ 를 내준다.

이 합병정렬 과정을 도식화하면 다음과 같다.



구현

이 합병정렬 알고리즘을 재귀함수로 짜면 다음과 같다.

```

1 def msort(s):
2     if len(s) > 1:
3         mid = len(s) // 2
4         return merge(msort(s[:mid]), msort(s[mid:]))
5     else:
6         return s

```

이제 두 정렬된 리스트를 합치는 merge 함수를 설계해보자.

- 두 리스트 모두 정렬되어 있으므로, 각 리스트의 맨 앞에 있는 두 수 중에서 작은 수를 하나 빼내어 정렬된 리스트를 만들어나간다.
- 이 과정을 두 리스트 중에서 하나가 소진될 때까지 계속 반복한다.
- 둘 중에 하나가 소진되면 나머지를 정렬된 리스트 뒤에 붙인다.

이제 두 정렬된 리스트 left와 right를 합치는 재귀함수 merge0 함수를 먼저 짜보자. 반복조건을 먼저 생각해보자.

```

left와 right 양쪽에 최소한 원소가 하나 있음
= left도 비어있지 않고 right도 비어있지 않음
= left != [] and right != []
= not (left == []) and not (right == [])
= not (left == [] or right == [])

```

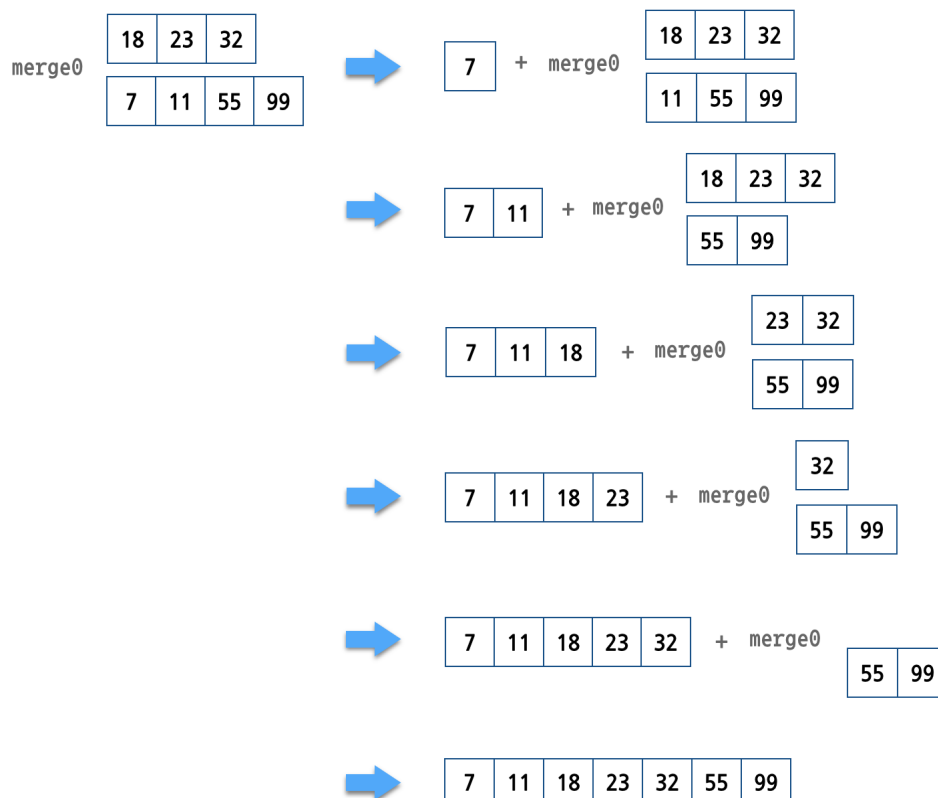
다음은 merge0 재귀 함수이다.

```

1 def merge0(left,right):
2     if not (left == [] or right == []):
3         if left[0] <= right[0]:
4             return [left[0]] + merge0(left[1:],right)
5         else:
6             return [right[0]] + merge0(left,right[1:])
7     else:
8         return left + right

```

이 함수를 위의 마지막 합병 사례를 가지고 실행추적해보면 다음과 같다.



5. 퀵정렬 Quicksort

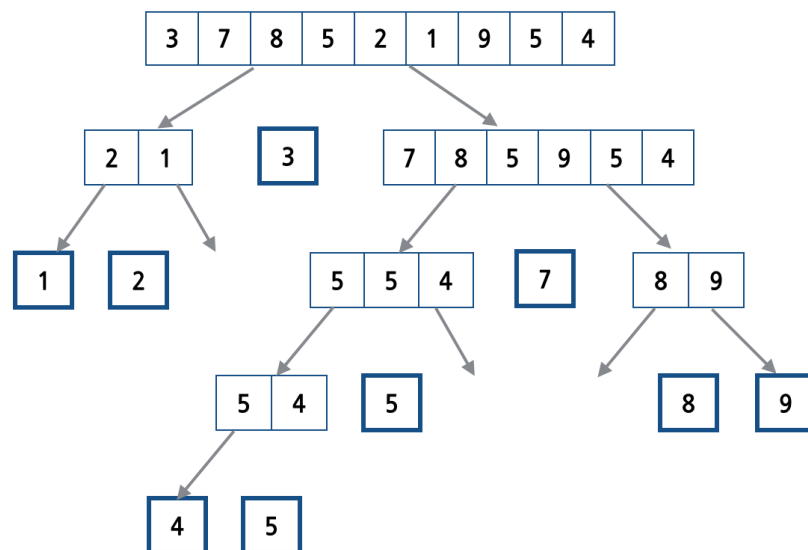
알고리즘

퀵 정렬은 영국의 유명한 컴퓨터과학자 C.A.R. Hoare가 60년대에 고안한 정렬 알고리즘이다. 이 알고리즘은 합병정렬과 비슷하나 대상 리스트를 무조건 반으로 쪼개는 대신 피벗(pivot)이라고 하는 기준값을 선택하고 기준값보다 작은 값만 들어 있는 리스트와 큰 값만 들어 있는 리스트로 나눌 수, 재귀적으로 정렬한다. 합병정렬은 무조건 반으로 나누어 재귀적으로 정렬하므로 추후에 합병 절차가 필요하지만, 퀵정렬은 피벗 값을 중심으로 작은 값과 큰 값을 따로 끼리끼리 나누는 수고를 미리 하고 재귀적으로 정렬하므로 뒤에 합병할 필요가 없다.

정수 리스트 s 를 퀵정렬하는 함수 `qsort`는 다음과 같이 짤다.

- $\text{len}(s) > 1$,
 - 기준으로 사용할 피벗값 `pivot`을 하나 고른다. 편의상 맨 앞에 있는 수를 고르기로 한다. (사실 피벗값을 잘 골라서 리스트가 항상 반으로 쪼개지는 경우, 퀵정렬의 성능이 가장 좋으므로 피벗값을 고르는 작업은 사실 중요하다.)
 - `pivot`을 기준으로 작은 작은 수는 왼쪽 리스트에, 큰 수는 오른쪽 리스트로 옮긴다.
 - 왼쪽 리스트와 오른쪽 리스트를 각각 재귀로 정렬하고, 가운데에 `pivot` 값을 끼운다.
- $\text{len}(s) \leq 1$, 정렬할 필요가 없으므로 그대로 내줌

예를 들어, $[3, 7, 8, 5, 2, 1, 9, 5, 4]$ 를 퀵정렬하면 다음 그림과 같은 절차를 정렬한다.



다음 프로그램은 이 전략을 그대로 적용하여 재귀함수로 작성한 `qsort` 함수이다.

```

1 def qsort(s):
2     if len(s) > 1:
3         pivot = s[0]
4         (left, right) = partition(pivot, s[1:])
5         return qsort(left) + [pivot] + qsort(right)
6     else:
7         return s

```

피벗값을 중심으로 리스트를 나누는 작업은 함수 `partition`을 만들어 하게 한다. `partition` 함수는 피벗값 `pivot`과 리스트 `s`를 인수로 받아서, `pivot`을 중심으로 작은 수가 들어있는 리스트와 큰수가 들어있는 리스트를 튜플로 내준다. 위에서는 이를 각각 `left`와 `right` 변수로 지정하였다.

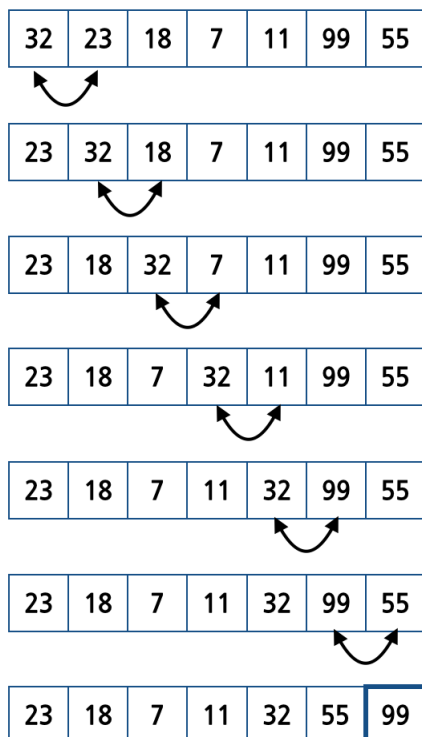
`partition` 함수는 다음과 같이 `for` 반복문을 사용하여 간단히 작성할 수 있다.

```
1 def partition(pivot,s):
2     left, right = [], []
3     for x in s:
4         if x <= pivot:
5             left.append(x)
6         else:
7             right.append(x)
8     return left, right
```

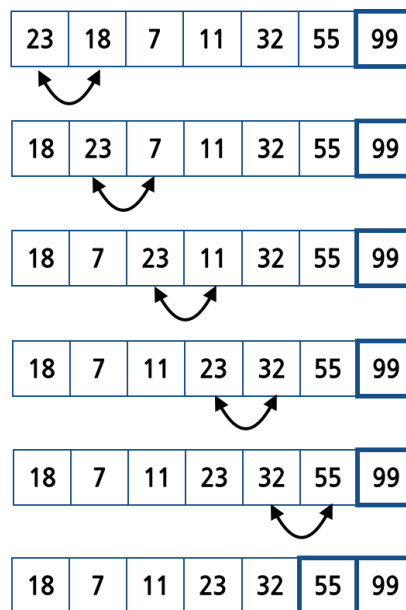
6. 버블정렬 Bubble sort

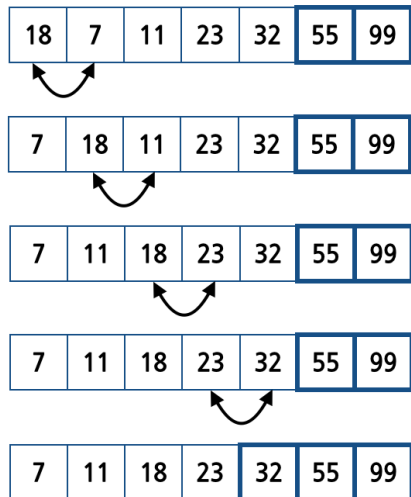
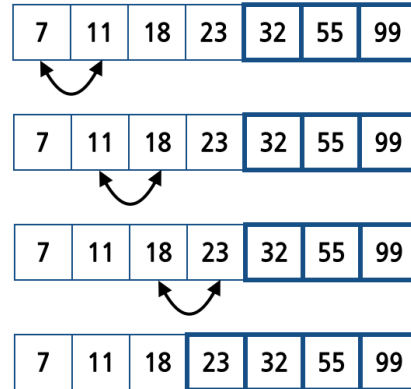
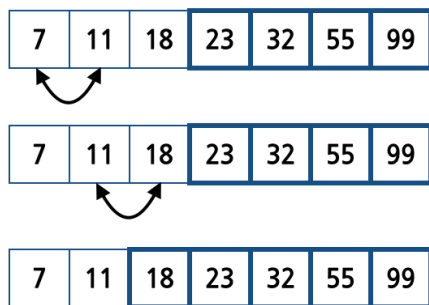
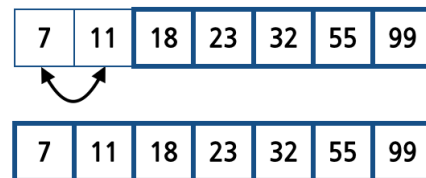
버블정렬은 리스트에서 인접한 두 수를 비교하여 순서가 뒤바뀐 경우 이를 교환하여 정렬하는 방법이다. 아래의 사례를 따라가보면 어떻게 정렬하는지 감이 잡힐 것이다.

1st loop



2nd loop



3rd loop4th loop5th loop6th loop

버블정렬을 잘 살펴보면 리스트 내부에서 수의 교환이 이루어진다. 따라서 추가공간이 전혀 필요없다. 이와 같이 추가공간을 사용하지 않고 자체적으로 값을 교환하여 정렬하는 것을 **제자리정렬** in-place sort 이라고 한다. 버블정렬을 제외하고 지금까지 공부한 정렬은 모두 제자리정렬이 아니었다.