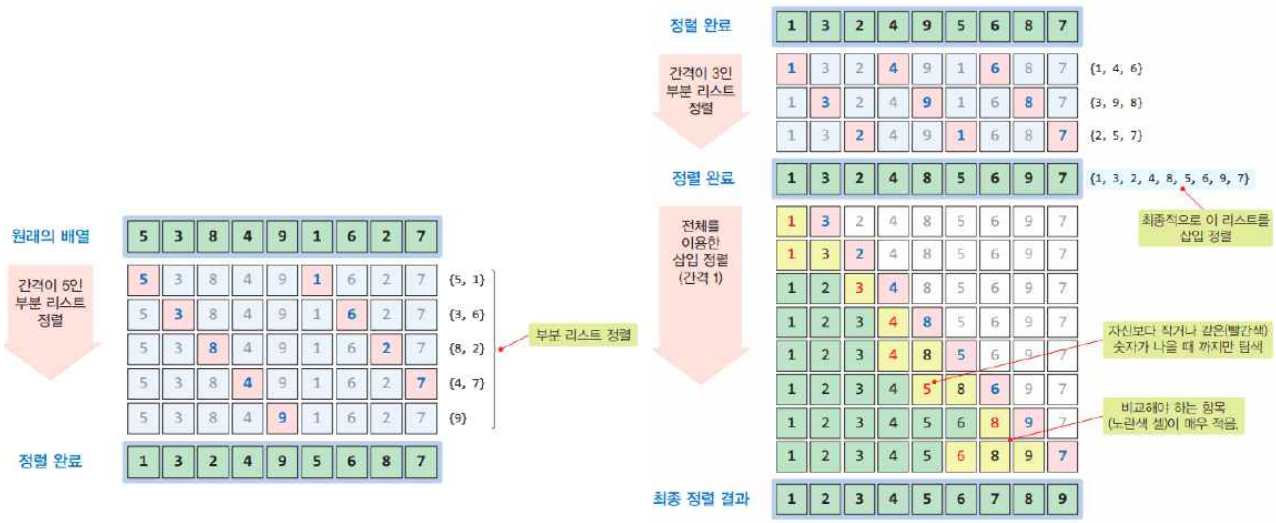
13차시

```
## 다양한 정렬 알고리즘
# 간단한 정렬 알고리즘의 특징 O(N^2)
# 선택정렬 : 입력의 크기에 따라 자료 이동 횟수가 결정
# 삽입 정렬 : 레코드의 많은 이동이 필요, 대부분의 레코드가 이미 정렬되어 있는 경우에는 효율적
# 버블 정렬 : 가장 간단한 알고리즘

# 효율적인 알고리즘들
# 셸 정렬 : 삽입 정렬 개념을 개선한 방법
# 합 정렬 : 제자리 정렬로 구현하는 방법
# 병합 정렬 : 연속적인 분할과 병합을 이용
# 퀵 정렬, 이중피벗 퀵 정렬 : 피벗을 이용한 정렬
# 기수, 카운팅 정렬 : 분배를 이용해 정렬, 킷값에 제한이 있음

## 셸정렬

## 기본 아이디어
# 삽입 정렬은 어느 정도 정렬된 리스트에서 대단히 빠르지만 요소들이 이수한 위치로만 이동하므로 많은 이동 발생
# 요소들이 멀리 떨어진 위치로 이동할 수 있게 한다면 보다 적게 이동하여 제자리를 찾을 수 있음

## 알고리즘
# 리스트를 일정 간격의 부분 리스트로 나눔 - 나뉘어진 각각의 부분 리스트를 삽입 정렬 함
# 간격을 줄임 - 부분 리스트의 수는 더 작아지고, 각 부분 리스트는 더 커짐
# 간격이 1이 될 때까지 이 과정을 반복

## 장점
# 불연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치교환으로 제자리 찾을 가능성 증대
# 부분 리스트 점신적으로 정렬된 상태가 되므로 삽입 정렬 속도 증가

## 시간 복잡도
# 최악의 경우 : O(n^2)
# 평균적인 경우 : O(n^1.5)
```



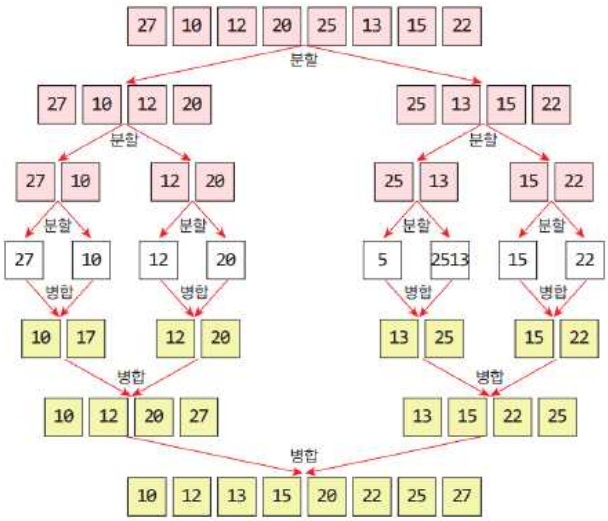
```
def shell_sort(A):
    n = len(A)
    gap = n//2
    print('shell_')

    while gap>0:
        print('\nwhile 시작')
        print('원래 gap : ', gap)
        if (gap % 2) == 0 :
            gap += 1
        print('정정한 gap : ', gap)
        for i in range(gap):
            print('\n    sortGapInsertion(A, %d, %d, %d)'%(i, n-1, gap), ' 함수 시작')
            sortGapInsertion(A, i, n-1, gap)
            print('    sortGapInsertion(A, %d, %d, %d)'%(i, n-1, gap), ' 함수 종료')
        print('\n    Gap=',gap,A)
        gap = gap//2

def sortGapInsertion(A, first, last, gap):
    print('        range(%d, %d, %d) for문 시작'%(first+gap, last+1, gap))
    for i in range(first+gap, last+1, gap):
        key = A[i]
        j = i - gap
        print('            key : ',A[i],' , j : ',j)
        while j >= first and key < A[j]:
            print('                whlie문 통과')
            A[j + gap] = A[j]
            j = j - gap
        print('            A[',j + gap,'] = ',key,' 으로 재정의')
        A[j + gap] = key

if __name__ == "__main__":
    list = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print('data : ', list)
    print('\nshell 정렬 시작')

    shell_sort(list)
    print('\nshell : ', list)
```

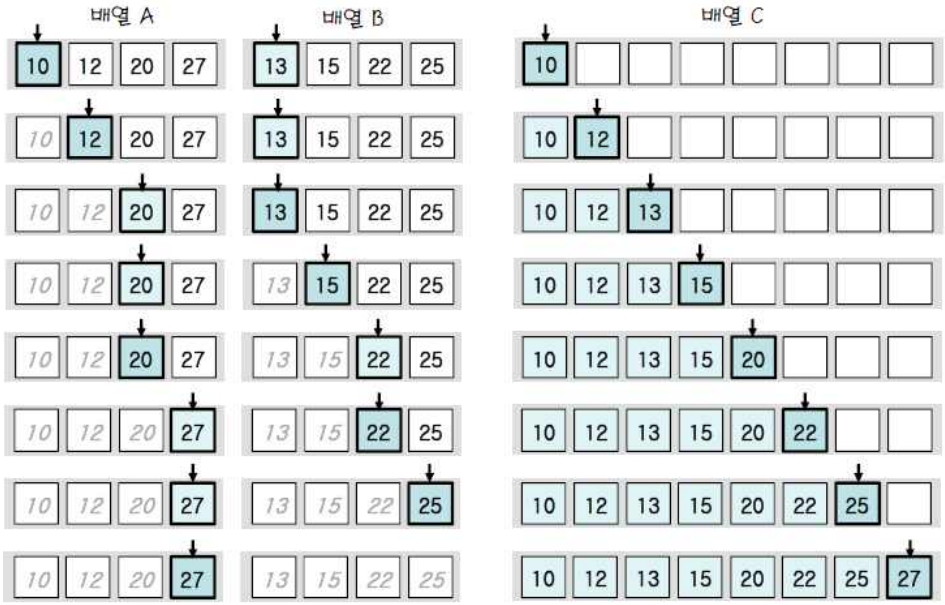
```
data :  [5, 3, 8, 4, 9, 1, 6, 2, 7]

shell 정렬 시작
shell_

while 시작
원래 gap :  4
정정한 gap :  5

   sortGapInsertion(A, 0, 8, 5)  함수 시작
      range(5, 9, 5) for문 시작
         key :  1   , j :  0
            whlie문 통과
         A[ 0 ] =  1   으로 재정의
   sortGapInsertion(A, 0, 8, 5)  함수 종료

   sortGapInsertion(A, 1, 8, 5)  함수 시작
      range(6, 9, 5) for문 시작
         key :  6   , j :  1
         A[ 6 ] =  6   으로 재정의
   sortGapInsertion(A, 1, 8, 5)  함수 종료

   sortGapInsertion(A, 2, 8, 5)  함수 시작
      range(7, 9, 5) for문 시작
         key :  2   , j :  2
            whlie문 통과
         A[ 2 ] =  2   으로 재정의
   sortGapInsertion(A, 2, 8, 5)  함수 종료

   sortGapInsertion(A, 3, 8, 5)  함수 시작
      range(8, 9, 5) for문 시작
         key :  7   , j :  3
         A[ 8 ] =  7   으로 재정의
   sortGapInsertion(A, 3, 8, 5)  함수 종료

   sortGapInsertion(A, 4, 8, 5)  함수 시작
      range(9, 9, 5) for문 시작
   sortGapInsertion(A, 4, 8, 5)  함수 종료

   Gap= 5 [1, 3, 2, 4, 9, 5, 6, 8, 7]

while 시작
원래 gap :  2
정정한 gap :  3

   sortGapInsertion(A, 0, 8, 3)  함수 시작
      range(3, 9, 3) for문 시작
         key :  4   , j :  0
         A[ 3 ] =  4   으로 재정의
         key :  6   , j :  3
         A[ 6 ] =  6   으로 재정의
   sortGapInsertion(A, 0, 8, 3)  함수 종료

   sortGapInsertion(A, 1, 8, 3)  함수 시작
      range(4, 9, 3) for문 시작
         key :  9   , j :  1
         A[ 4 ] =  9   으로 재정의
         key :  8   , j :  4
            whlie문 통과
         A[ 4 ] =  8   으로 재정의
   sortGapInsertion(A, 1, 8, 3)  함수 종료

   sortGapInsertion(A, 2, 8, 3)  함수 시작
      range(5, 9, 3) for문 시작
         key :  5   , j :  2
         A[ 5 ] =  5   으로 재정의
         key :  7   , j :  5
         A[ 8 ] =  7   으로 재정의
   sortGapInsertion(A, 2, 8, 3)  함수 종료

   Gap= 3 [1, 3, 2, 4, 8, 5, 6, 9, 7]

while 시작
원래 gap :  1
정정한 gap :  1

   sortGapInsertion(A, 0, 8, 1)  함수 시작
      range(1, 9, 1) for문 시작
         key :  3   , j :  0
         A[ 1 ] =  3   으로 재정의
         key :  2   , j :  1
            whlie문 통과
         A[ 1 ] =  2   으로 재정의
         key :  4   , j :  2
         A[ 3 ] =  4   으로 재정의
         key :  8   , j :  3
         A[ 4 ] =  8   으로 재정의
         key :  5   , j :  4
            whlie문 통과
         A[ 4 ] =  5   으로 재정의
         key :  6   , j :  5
            whlie문 통과
         A[ 5 ] =  6   으로 재정의
         key :  9   , j :  6
         A[ 7 ] =  9   으로 재정의
         key :  7   , j :  7
            whlie문 통과
            whlie문 통과
         A[ 6 ] =  7   으로 재정의
   sortGapInsertion(A, 0, 8, 1)  함수 종료

   Gap= 1 [1, 2, 3, 4, 5, 6, 7, 8, 9]

shell :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
'''
```

```
## 힙 정렬
# 힘 클래스를 이용한 정렬
# 추가적인 메모리를 필요로 함

## 시간복잡도
# O(n log n)


from MaxHeap import MaxHeap # 최대힙 클래스

def heapSort(data):
    heap = MaxHeap() # 최대 힙 사용
    for n in data: # 모든 항목들을 힙에 넣음
        heap.insert(n)

    print('heap에 대이터 삽입한 결과 : ', end='')
    heap.display()
    for i in range(1, len(data)+1): # 1, 2, 3, ... , n
        data[-i] = heap.delete() # 앤 뒤에서 앞으로 -1, -2, ...
        print(data[-i],'를 제거한 다음의 ',end='')
        heap.display()

if __name__ == "__main__":
    dataL = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print('data       :', dataL)
    heapSort(dataL)
    print('HeapSort :', dataL)

'''
data       : [5, 3, 8, 4, 9, 1, 6, 2, 7]
heap에 대이터 삽입한 결과 : 힙트리:  [9, 8, 6, 7, 4, 1, 5, 2, 3]
9 를 제거한 다음의 힙트리:  [8, 7, 6, 3, 4, 1, 5, 2]
8 를 제거한 다음의 힙트리:  [7, 4, 6, 3, 2, 1, 5]
7 를 제거한 다음의 힙트리:  [6, 4, 5, 3, 2, 1]
6 를 제거한 다음의 힙트리:  [5, 4, 1, 3, 2]
5 를 제거한 다음의 힙트리:  [4, 3, 1, 2]
4 를 제거한 다음의 힙트리:  [3, 2, 1]
3 를 제거한 다음의 힙트리:  [2, 1]
2 를 제거한 다음의 힙트리:  [1]
1 를 제거한 다음의 힙트리:  []
HeapSort : [1, 2, 3, 4, 5, 6, 7, 8, 9
'''
```

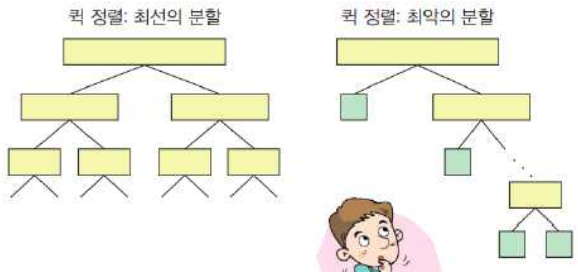
```
## 제자리 정렬로 구현한 힙 정렬

## 알고리즘
# 1단계 : 리스트를 최대힙으로 만듦
# 2단계 : 최대힙을 정렬된 리스트로 만듦
# Heapify : 최대힙을 만드는 과정

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l<n and arr[i] < arr[l]:
        largest = l
    if r<n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)
    print("i=",0, arr)
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)
        print("i=",i, arr)
    print()

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
        print("i= ",i, arr)

if __name__ == "__main__":
    dataL = [5, 3, 8, 4, 9, 1, 6, 2, 7]

    heapSort(dataL)

    print('HeapSort : ',dataL)
'''
i= 0 [5, 3, 8, 4, 9, 1, 6, 2, 7]
i= 4 [5, 3, 8, 4, 9, 1, 6, 2, 7]
i= 3 [5, 3, 8, 7, 9, 1, 6, 2, 4]
i= 2 [5, 3, 8, 7, 9, 1, 6, 2, 4]
i= 1 [5, 9, 8, 7, 3, 1, 6, 2, 4]
i= 0 [9, 7, 8, 5, 3, 1, 6, 2, 4]

i=  8 [8, 7, 6, 5, 3, 1, 4, 2, 9]
i=  7 [7, 5, 6, 2, 3, 1, 4, 8, 9]
i=  6 [6, 5, 4, 2, 3, 1, 7, 8, 9]
i=  5 [5, 3, 4, 2, 1, 6, 7, 8, 9]
i=  4 [4, 3, 1, 2, 5, 6, 7, 8, 9]
i=  3 [3, 2, 1, 4, 5, 6, 7, 8, 9]
i=  2 [2, 1, 3, 4, 5, 6, 7, 8, 9]
i=  1 [1, 2, 3, 4, 5, 6, 7, 8, 9]
HeapSort :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
'''
```

```
## 병합 정렬

## 분할 정복 방법
# 문제를 보다 작은 2개의 문제로 분리하고 각 문제를 해결한 다음, 결과를 모아서 원래의 문제를 해결하려는 전략

## 시간 복잡도
## 비교횟수
# 크기 n인 리스트를 균등 분배하므로 log(n)개의 패스
# 각 패스에서 레코드 n개를 비교, n번 비교 연산
## 이동횟수
# 각 패스에서 2n의 이동 발생 -> 전체 이동 2n * log(n)
## 시간복잡도
# O(n log n)

## 분석
# 효율적인 알고리즘
# 최적, 평균, 최악의 경우에도 동일한 시간에 정렬
# 추가적인 메모리가 필요

def merge_sort(A, left, right):
    if left < right:
        mid = (left+right) // 2
        merge_sort(A, left, mid)
        merge_sort(A, mid+1, right)
        merge(A, left, mid, right)

def merge(A, left, mid, right):
    global sorted
    k = left
    i = left
    j = mid + 1
    while i <= mid and j <= right:
        if A[i] <= A[j]:
            sorted[k] = A[i]
            i, k= i+1, k+1
        else:
            sorted[k] = A[j]
            j, k = j+1, k+1

    if i > mid:
        sorted[k : k+right-j+1] = A[j:right+1]
    else:
        sorted[k:k+mid-i+1] = A[i:mid+1]
    A[left:right+1] = sorted[left:right+1]

if __name__ == "__main__":
    sorted = []
    data = [10,12,20,27,13,15,22,25]
    sorted = [0]*len(data)
    print('      data : ', data)
    merge_sort(data, 0, len(data)-1)
    print('MergeSort : ', data)
'''
     data :  [10, 12, 20, 27, 13, 15, 22, 25]
MergeSort :  [10, 12, 13, 15, 20, 22, 25, 27]
'''
```

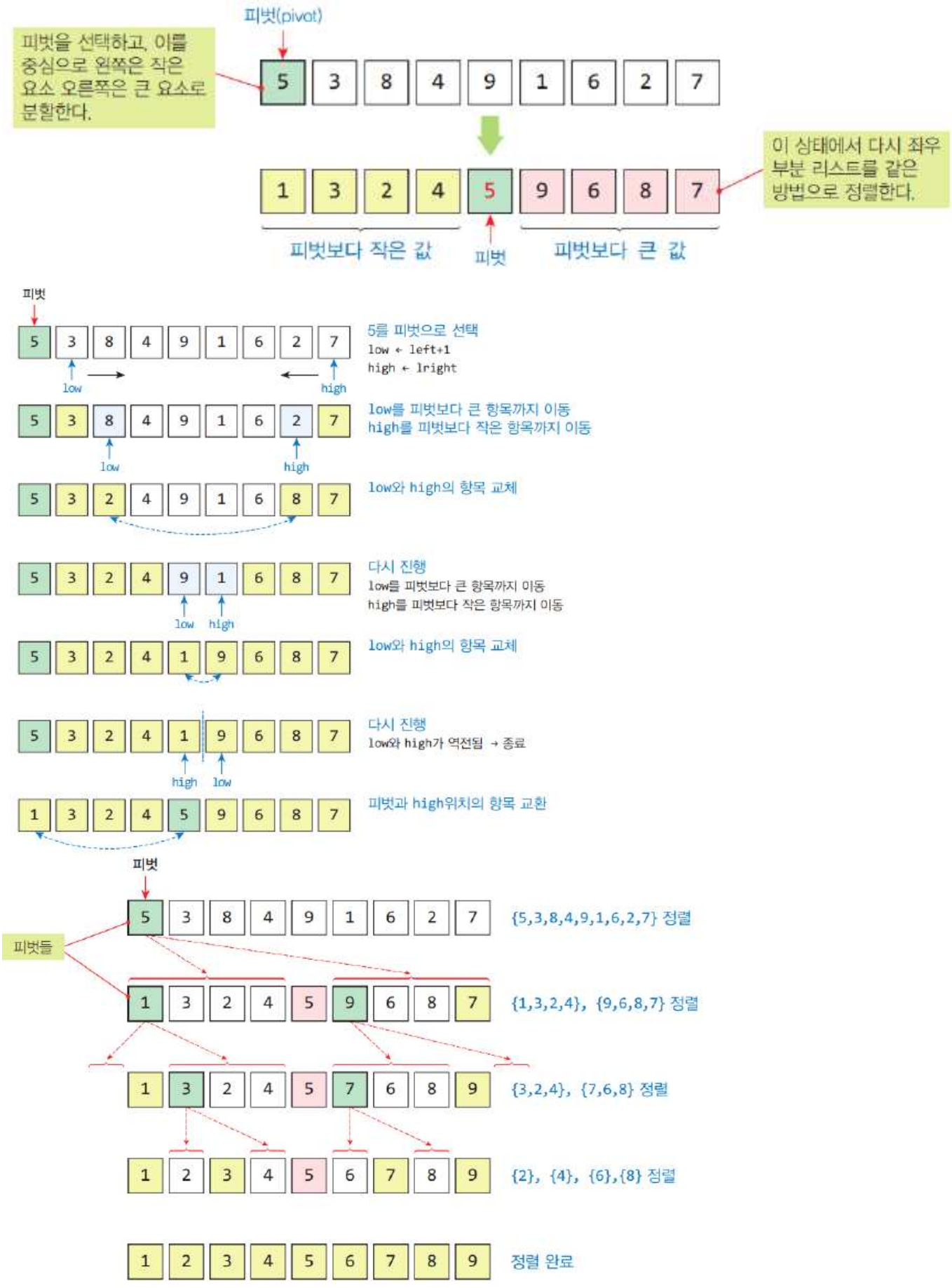

27 10 12 20 25 13 15 22
분할
27 10 12 20
25 13 15 22
분할
27 10
12 20
25 13
15 22
분할
27
10
12
20
5
2513
15
22
병합
10 17
12 20
13 25
15 22
병합
10 12 20 27
13 15 22 25
병합
10 12 13 15 20 22 25 27


배열 A
배열 B
배열 C

```
## 퀵 정렬 -- 코드는 넘어간다.
# 분할 정복법 사용
# 리스트를 2개의 부분 리스트로 비 균등 분할
# 각각의 부분리스트를 다시 퀵 정렬함(순환 호출)

## 복잡도 분석
# 최선의 경우
# 균등 분할
# 패스 수 : log n
# 복잡도 : O(n log n)

# 최악의 경우
# 이미 정렬된 리스트
# 패스 수 : n
# 복잡도 : O(n^2)
```

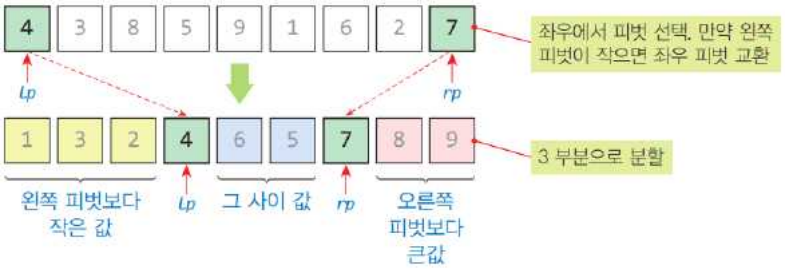



```
def quick_sort(A, left, right):
    if left < right:
        q = partition(A, left, right)
        quick_sort(A, left, q-1)
        quick_sort(A, q+1, right)

def partition(A, left, right):
    low = left + 1
    high = right
    pivot = A[left]
    while (low <= high):
        while low <= right and A[low] < pivot:
            low += 1
        while high >= left and A[high] > pivot:
            high -= 1

        if low < high:
            A[low], A[high] = A[high], A[low]

    A[left], A[high] = A[high], A[left]
    return high

if __name__ == "__main__":
    data = [5, 3, 8, 4, 9, 1, 6, 2, 7]
    print('      data : ', data)
    quick_sort(data, 0, len(data)-1)
    print('QuickSort : ', data)

'''
      data :  [5, 3, 8, 4, 9, 1, 6, 2, 7]
QuickSort :  [1, 2, 3, 4, 5, 6, 7, 8, 9]
'''
```


피벗(pivot)
피벗을 선택하고, 이를 중심으로 왼쪽은 작은 요소 오른쪽은 큰 요소로 분할한다.
5 3 8 4 9 1 6 2 7
1 3 2 4 5 9 6 8 7
이 상태에서 다시 좌우 부분 리스트를 같은 방법으로 정렬한다.
피벗보다 작은 값
피벗
피벗보다 큰 값
피벗
low
high
5를 피벗으로 선택
low ← left+1
high ← lright
low를 피벗보다 큰 항목까지 이동
high를 피벗보다 작은 항목까지 이동
low와 high의 항목 교체
다시 진행
low를 피벗보다 큰 항목까지 이동
high를 피벗보다 작은 항목까지 이동
low와 high의 항목 교체
다시 진행
low와 high가 역전됨 → 종료
피벗과 high위치의 항목 교환
피벗
피벗들
{5,3,8,4,9,1,6,2,7} 정렬
{1,3,2,4}, {9,6,8,7} 정렬
{3,2,4}, {7,6,8} 정렬
{2}, {4}, {6},{8} 정렬
정렬 완료

이중 피벗 퀵 정렬

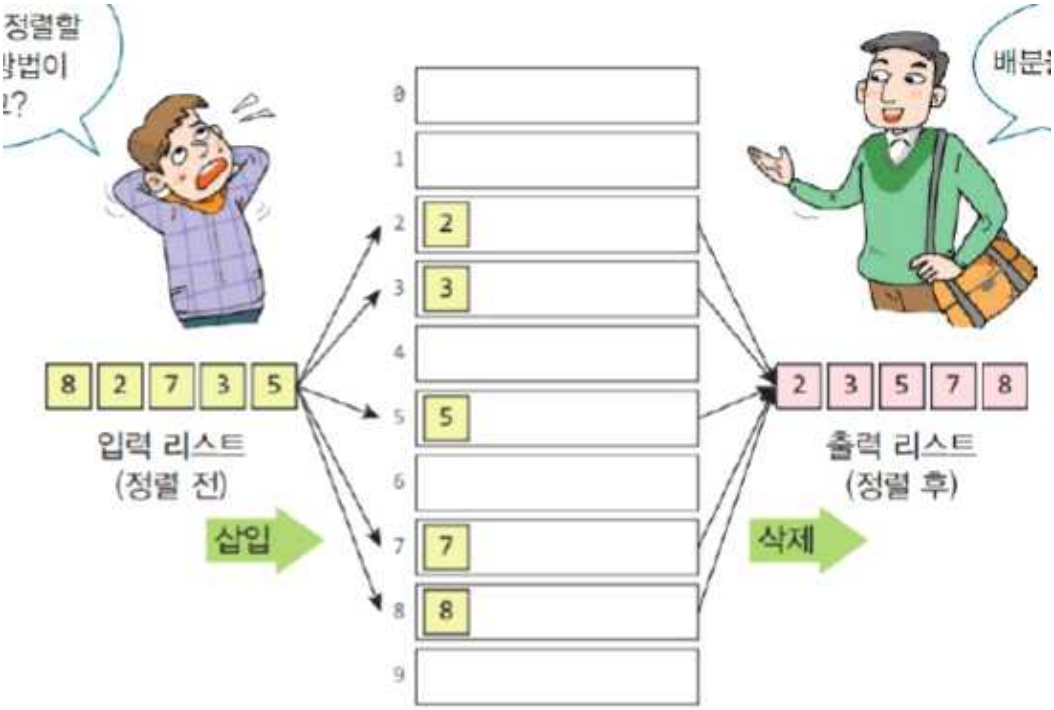
2개의 피벗을 사용하는 퀵 정렬

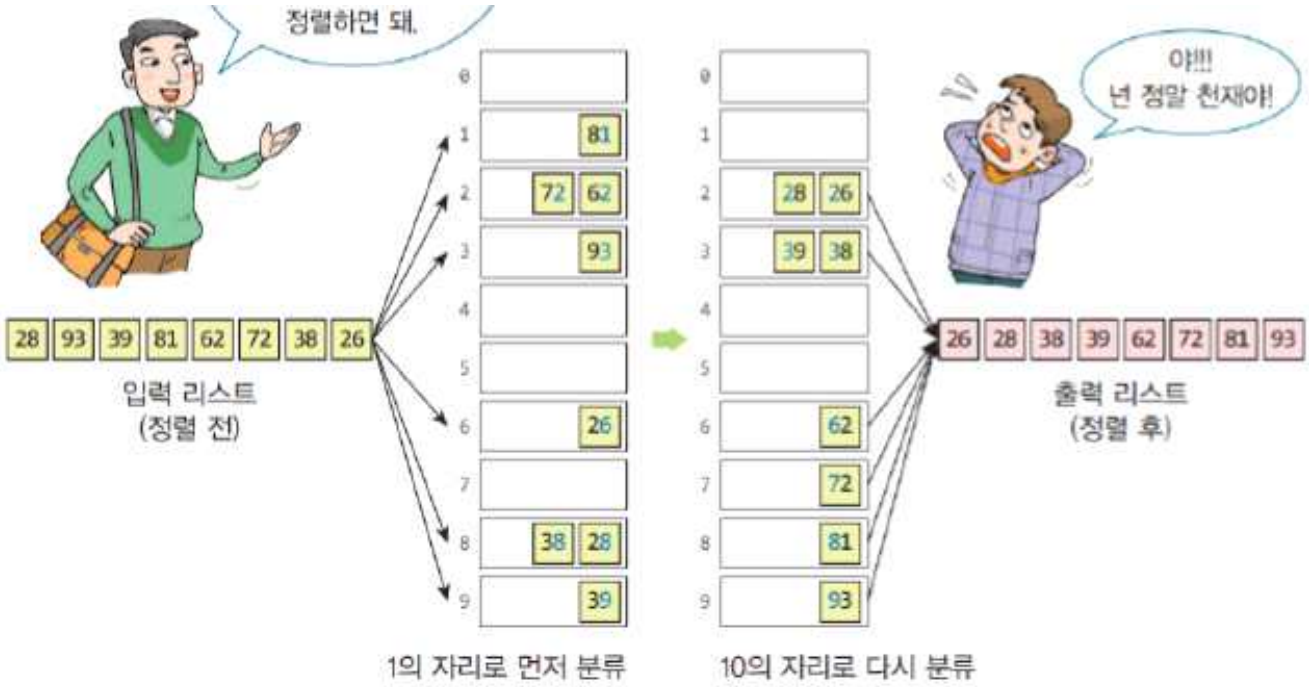



```
def dp_quick_sort(A, low, high) :
    if low < high :
        lp, rp = partitionDP(A, low, high)      # 좌우 피벗의 인덱스를 반환받음
        dp_quick_sort(A, low, lp-1)             # low ~ lp-1 정렬
        dp_quick_sort(A, lp+1, rp-1)            # lp+1 ~ rp-1 정렬
        dp_quick_sort(A, rp+1, high)            # rp+1 ~ high 정렬
```

```
## 기수정렬
# 레코드를 비교하지 않고 분배하여 정렬 수행
# 비교에 의한 정렬의 하한인 O(n log n)보다 좋을 수 있음
# 시간복잡도 : O(dn), 대부분 d<10 이하

## 아이디어
# 단순히 자리수에 따라 숫자를 bucket에 넣었다가 꺼내면 정렬됨
# 두자리수 이상 기수는 낮은 자릿수 분류하고 순서대로 읽으면서 높은 자리수를 분류를 진행하면 된다.

## 버킷(큐)의 개수는 키의 표현 방법과 밀접한 관계
# 이진법을 사용한다면 버킷은 2개
# 알파벳 문자를 사용한다면 버킷은 26개
# 십진법을 사용한다면 버킷은 10개

## n개의 레코드 d개의 자릿수 키의 기수 정렬
# 메인 루프는 자릿수 d번 반복
# 큐에 n개 레코드 입력 수행

## 단점
# 정렬할 수 있는 레코드의 타입 한정
# 정수나 단순 문자(알파벳 등)이어야만 함
# 실수, 한글, 한자로 이루어진 키는 정렬 못함

import random
from queue import Queue

def radix_sort(A):
    queues = []
    for i in range(BUCKETS):
        queues.append(Queue())

    n = len(A)
    factor = 1
    for d in range(DIGITS):
        for i in range(n):
            queues[ ( A[i]//factor )%10 ].put(A[i])
        i = 0
        for b in range(BUCKETS):
            while not queues[b].empty():
                A[i] = queues[b].get()
                i += 1

        factor *= 10
        print("step", d+1, A)

if __name__ == "__main__":
    BUCKETS = 10 # data 수
    DIGITS = 4 # 각 숫자의 자리수
    data = [] # data : [4279, 663, 2976, 9632, 92, 5073, 500, 4141, 8017, 6680]
    for i in range(10):
        data.append(random.randint(1,9999))
    print(" data :", data)
    print("\nRadixSort Start")

    radix_sort(data)
    print("\nRadixSort:", data)
```

```
'''
 data : [4279, 663, 2976, 9632, 92, 5073, 500, 4141, 8017, 6680]

RadixSort Start
step 1 [500, 6680, 4141, 9632, 92, 663, 5073, 2976, 8017, 4279]
step 2 [500, 8017, 9632, 4141, 663, 5073, 2976, 4279, 6680, 92]
step 3 [8017, 5073, 92, 4141, 4279, 500, 9632, 663, 6680, 2976]
step 4 [92, 500, 663, 2976, 4141, 4279, 5073, 6680, 8017, 9632]

RadixSort: [92, 500, 663, 2976, 4141, 4279, 5073, 6680, 8017, 9632]
'''
```





❖ 하이브리드 정렬 알고리즘의 예

➢ 팀 정렬: 파이썬의 기본 정렬 알고리즘으로 사용

❖ 정렬 알고리즘들의 성능 비교

알고리즘	최선	평균	최악
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
버블 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
셸 정렬	$O(n)$	$O(n^{1.5})$	$O(n^2)$
힙 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
병합 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
퀵 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
이중피벗 퀵 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
기수 정렬	$O(dn)$	$O(dn)$	$O(dn)$
카운팅 정렬	$O(k+n)$	$O(k+n)$	$O(k+n)$
팀 정렬	$O(n)$	$O(n \log_2 n)$	$O(n \log_2 n)$