버블정렬(Bubble Sort)
```
def bubble_sort(A):
  n = len(A)
  for i in range(n-1):
    for j in range(0, n-i-1):
      if A[j] > A[j + 1] :
        A[j], A[j + 1] = A[j + 1], A[j]
```

선택정렬(Selection Sort)
```
def selection_sort(A):
  n=len(A)
  for i in range(n-1):
    least = i
    for j in range(i+1, n):
      if A[least] > A[j]:
        least = j
      A[i], A[least] = A[least], A[i]
```

삽입정렬(Insertion Sort)
```
def insertion_sort(A):
  n = len(A)
  for i in range(1, n):
    key = A[i]
    j = i-1
    while j >= 0 and key < A[j] :
      A[j + 1] = A[j]
      j -=  1
    A[j + 1] = key
```

병합정렬(Merge Sort)
```
def merge(A, left, mid, right):
  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]

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)
```

퀵 정렬(Quick Sort)
```
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

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

힙 정렬(Heap Sort)
```
def heapify(A, n, i):
  largest = i
  left = 2 * i + 1
  right = 2 * i + 2

  if left < n and A[largest] < A[left]:
    largest = left

  if right < n and A[largest] < A[right]:
    largest = right

  if largest != i:
    A[i], A[largest] = A[largest], A[i]
    heapify(A, n, largest)

def heap_sort(A):
  n = len(A)
  for i in range(n//2 – 1, -1 , -1):
    heapify(A, n, i)

  for i in range(n-1, 0, -1):
    A[i], A[0] = A[0], A[i]
    heapify(A, i, 0)
```

쉘 정렬(Shell Sort)
```
def shell_sort(A):
  gap = len(A) // 2

  while gap > 0:
    i = 0
    j = gap
    while j < len(A):
      if A[i] > A[j]:
        A[i],A[j] = A[j],A[i]

      i += 1
      j += 1
      k = i
      while k - gap > -1:
        if A[k - gap] > A[k]:
          A[k-gap],A[k] = A[k],A[k-gap]
        k -= 1
    gap //= 2
```

기수정렬(Radix Sort)
```
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])
    j = 0
    for b in range(BUCKETS):
      while not queues[b].empty():
        A[j] = queues[b].get()
        j += 1
    factor *= 10
```

카운팅정렬(Counting Sort)
```
def counting_sort(A):
  output = [0] * len(A)
  count = [0] * MAX_VAL

  for i in A:
    count[i] += 1

  for i in range(MAX_VAL):
    count[i] += count[i-1]

  for i in range(len(A)):
    output[count[A[i]] - 1] = A[i]
    count[A[i]] -= 1

  for i in range(len(A)):
    A[i] = output[i]i
```

버블정렬(Bubble Sort)

>> 장점

- 일단 구현이 쉽다. Bubble정렬은 인접한 값만 계쏙해서 비교하면 되는 방식으로 굉장히 구현이 쉬운 편이다.

- 코드 자체가 직관적이다.

- stable

>> 단점

- 굉장히 비효율적이다. 최악이든 최선이든 $O(N^2)$ 이라는 시간복잡도를 갖기 때문에 사실 알고리즘에서 효율적인 정렬방법으로 사용되지는 않는다.

선택정렬(Selection Sort)

>> 장점

- 선택정렬 또한 버블정렬과 마찬가지로 구현이 쉬운편에 속하는 정렬법이다.

- 정렬을 위한 비교 횟수는 많지만 실제로 교환하는 횟수는 적기 때문에 많은 교환이 일어나야 하는 자료상태에서 효율적으로 사용될 수 있다.

- 버블정렬과 비교했을 때, 똑같이 $O(N^2)$ 이라는 시간복잡도를 갖지만, 실제로 시간을 측정해보면 버블정렬에 비해서는 조금 더 빠른 정렬 방식이다.

>> 단점

- 선택정렬 또한 항상 $O(N^2)$ 이라는 시간복잡도를 갖기 때문에 시간이 오래걸리는 정렬 방식이다.

- not stable

삽입정렬(Insertion Sort)

>> 장점

- 최선의 경우(= 이미 오름차순으로 정렬되어 있는경우) O(N)이라는 엄청나게 빠른 효율성을 가지고 있다.

- 대부분의 레코드가 이미 정렬되어 있는 경우 효율적으로 사용될 수 있다는 점에서 다른 정렬 알고리즘(Shell Sort)의 일부로 사용될 만큼 좋은 정렬법이다

- stable

>> 단점

- 최악의 경우(= 리스트가 역으로 정렬된 경우) $O(N^2)$ 이라는 시간복잡도를 갖게된다. 즉, 데이터의 상태 및 데이터의 크기에 따라서 성능의 편차가 굉장히 심한 정렬법이다.

퀵 정렬(Quick Sort)

>> 장점

- 기준값에 의한 분할을 통해서 구현하는 정렬법으로써, 분할 과정에서 logN 이라는 시간이 걸리게되고 전체적으로 보게 되면NlogN 으로써 실행시간이 준수한 편이다.

- 최선의 경우 (= 리스트 분할이 항상 리스트의 가운데에서 이루어지는 상황)

>> 단점

- 기준값(Pivot)에 따라서 시간복잡도가 크게 달라진다. Pivot이 적당하게 이상적인 값을 선택했다면 NlogN의 시간복잡도를 갖지만, 최악으로 Pivot을 선택할 경우 (= 정렬된 리스트) $O(N^2)$ 이라는 시간복잡도를 갖게 된다.

병합정렬(Merge Sort)

>> 장점

- 퀵소팅과 비슷하게 원본 배열을 반씩 분할해가면서 정렬하는 정렬법으로써 분할 하는 과정에서 logN 만큼의 시간이 걸린다.

즉, 최종적으로 보게되면 NlogN 이 된다.

- 퀵소팅과 달리, Pivot을 설정하거나 그런 과정 없이 무조건 절반으로 분할하기 때문에 Pivot에 따라서 성능이 안좋아지거나 하는 경우가 없다. (= 최선, 평균, 최악의 경우를 나누어 생각할 필요가 없다). 따라서 항상 O(NlogN) 이라는 시간복잡도를 갖게된다. 이는 정렬법들 중에서 매우 준수한 수준이다.

- stable

>> 단점

- 장점만 본다면 퀵 보다는 무조건 병합정렬을 사용하는 것이 좋다고 생각할 수 있지만 가장 큰 단점은 '추가적인 메모리 필요' 이다. 병합정렬은 임시배열에 원본맵을 계속해서 옮겨주면서 정렬을 하는 방법이다.

즉, '추가적인 메모리를 할당할 수 없을 경우, 데이터가 최악으로 있다면 병합 vs 퀵 정렬법 중 무엇을 써야할까?' 라고 물으면 데이터가 최악인 것만 본다면 퀵보다는 병합정렬이 훨씬 빠르기때문에 병합정렬을 사용하는것이 많지만, 추가적인 메모리를 할당할 수 없다면 병합정렬은 사용할 수 없기 때문에 퀵을 사용해야 하는 것이다.

힙 정렬(Heap Sort)

>> 장점

- 추가적인 메모리를 필요로 하지 않으면서 항상 O(NlogN) 이라는 시간복잡도를 가지는 굉장히 정렬법들 중에서 효율적인 정렬법이라고 볼 수 있다. 퀵 정렬도 굉장히 효율적이라고 볼 수 있지만 최악의 경우 시간이 오래걸린다는 단점이 있지만 힙 정렬의 경우 항상 O(NlogN) 으로 보장된다는 장점이 있다.

>> 단점

- 하지만 이상적인 경우에 퀵정렬과 비교했을 때 똑같이 O(NlogN)이 나오긴 하지만 실제 시간을 측정해보면 퀵정렬보다 느리다고 한다. 즉, 데이터의 상태에 따라서 다른 정렬법들에 비해서 조금 느린 편이다.

또한, 안정성(Stable)을 보장받지 못한다는 단점이 있다.

-최악의 경우 (= pivot을 계속해서 최솟값 혹은 최댓값만 잡는 경우)

쉘 정렬(Shell Sort)

>> 장점

- 삽입정렬의 단점을 보완해서 만든 정렬법으로 삽입정렬도 성능이 뛰어난 편이지만 더 뛰어난 성능을 갖는 정렬법이다.

>> 단점

- 일정한 간격에 따라서 배열을 바라봐야 한다. 즉, 이 '간격'을 잘못 설정한다면 성능이 굉장히 안 좋아질수 있다.

기수정렬(Radix Sort)

>> 장점

- 정렬법들 중에서 O(N) 이라는 말도안되는 시간복잡도를 갖는 정렬법으로써 일단 엄청나게 빠르다.

- 정렬법에서 O(NlogN)을 깰 수 있는 방법은 없다고 알려져있는데, 그 방법을 깨는 유일한 방법이 기수정렬법이다.

>> 단점

- '버킷' 이라는 추가적인 메모리가 할당되어야 한다. 즉, 메모리가 엄청나게 여유롭다면 상관이 없겠지만 메모리가 생각보다 많이 소비되는 정렬법이다.

- 데이터 타입이 일정한 경우에만 가능하다. 기존에 정렬법들은 음수와 양수, 실수와 정수가 섞여 있더라도 비교를 하려면 할 수 있었지만, 기수정렬의 경우 양의 정수는 양의 정수끼리만, 음의 정수는 음의 정수끼리만 정렬이 가능하다.

- 즉, 엄청나게 빠른 대신에 구현을 위한 조건이 굉장히 많이 붙기 때문에 그렇게 많이 사용되는 방법 같지는 않다.

카운팅정렬(Counting Sort)

>> 장점

- 이 정렬법도 비교를 하지 않고 정렬하는 법으로 O(N) 이라는 시간복잡도를 갖게 된다.

- 일단, O(N)이라는 것 자체만으로도 정렬법 중에서 엄청나게 빠른 편에 속하고 이것이 장점으로 작용한다.

- 카운팅 정렬을 적용하기에 가장 좋은 입력은 나타날 수 있는 킷값이 일정한 개수로 제한되는 경우이다.

-입력 데이터가 크지 않은 일정한 범위의 값을 가진 정수라면 매우 효율적이다. 알고리즘의 모든 루프가 단일 루프이다.

>> 단점

- 숫자 갯수를 저장해야 될 별도의 공간, 또 결과를 저장할 별도의 공간 등 추가적인 메모리가 필요하다.

- 또한, 하나의 값 때문에 메모리의 낭비를 많이 하게 될 수도 있다. 예를 들면 다음과 같은 경우이다.

[1, 2, 3, 4, 5, 99999999999] 이 경우에는 99999999999 때문에 숫자의 갯수를 저장해야 될 배열의 크기가 최소 [99999999999] 보다는 커야 하고, 결과적으로 안 쓰는 낭비되는 인덱스들이 많이 발생하게 된다.

- 킷값이 실수라면 적용할 수 없다.

Sorting	장점	단점
버블 정렬	- 구현이 쉽다	- 시간이 오래 걸리고 비효율적이다.
선택 정렬	- 구현이 쉽다 - 비교하는 횟수에 비해 교환이 적게 일어난다.	- 시간이 오래 걸려서 비효율적이다.
퀵 정렬	- 실행시간 O(NlogN)으로 빠른 편이다	- Pivot에 의해서 성능의 차이가 심하다. - 최악의 경우 O(N^2)이 걸리게 된다.
힙 정렬	- 항상 O(NlogN)으로 빠른 편이다.	- 실제 시간이 다른 O(NlogN)이 정렬법들에 비해서 오래걸린다.
병합 정렬	- 항상 O(NlogN)으로 빠른 편이다.	- 추가적인 메모리 공간을 필요로 한다.
삽입 정렬	- 최선의 경우 O(N)으로 굉장히 빠른 편이다. - 성능이 좋아서 다른 정렬법에 일부로 사용됨.	- 최악의 경우O(N^2) 으로, 데이터의 상태에 따라서 성능 차이가 심하다.
쉘 정렬	- 삽입정렬보다 더 빠른 정렬법이다.	- 설정하는 '간격'에 따라서 성능 차이가 심하다.
기수 정렬	- O(N)이라는 말도 안 되는 속도를 갖는다.	- 추가적인 메모리가 '많이' 필요하다. - 데이터 타입이 일정해야 한다. - 구현을 위한 조건이 많이 붙는다.
카운팅 정렬	- O(N)이라는 말도 안 되는 속도를 갖는다.	- 추가적인 메모리 공간이 필요하다. - 일부 값 때문에 메모리의 낭비가 심해질 수 있다.

Sorting	최악의 경우(Worst)	일반(평균)적인 경우(Average)	최선의 경우(Best)
버블 정렬	$O(N^2)$	$O(N^2)$	$O(N^2)$
선택 정렬	$O(N^2)$	$O(N^2)$	$O(N^2)$
퀵 정렬	$O(N^2)$	$O(NlogN)$	$O(NlogN)$
힙 정렬	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
병합 정렬	$O(NlogN)$	$O(NlogN)$	$O(NlogN)$
삽입 정렬	$O(N^2)$	$O(N^2)$	$O(N)$
쉘 정렬	$O(N^2)$	$O(N^{1.3,1.5})$	$O(N)$
기수 정렬	$O(N)$	$O(N)$	$O(N)$
카운팅 정렬	$O(N)$	$O(N)$	$O(N)$