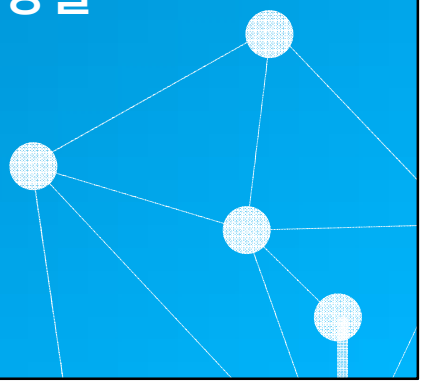
파이썬
자료구조

CHAPTER 12

고급 정렬



12장. 학습 목표

- 셸 정렬
- 힙 정렬 (Heapify 알고리즘)
- 병합 정렬
- 퀵 정렬
- 이중피벗 퀵 정렬
- 기수 정렬
- 카운팅 정렬
- 정렬 알고리즘의 성능 비교

12.1 다양한 정렬 알고리즘

- 간단한 정렬 알고리즘의 특징 $O(n^2)$
  - 선택 정렬: 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
  - 힙 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.

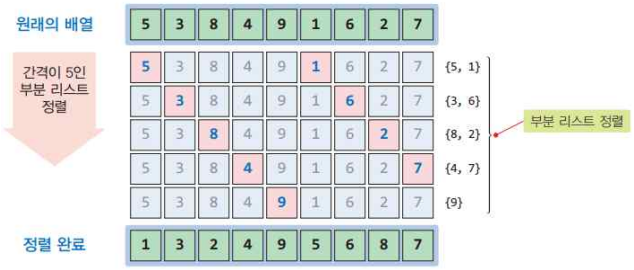
12.2 셸 정렬

- 기본 아이디어
  - 삽입정렬은 어느 정도 정렬된 리스트에서 대단히 빠르다!
  - 그러나 요소들이 이웃한 위치로만 이동➔많은 이동 발생
  - 요소들이 멀리 떨어진 위치로 이동할 수 있게 한다면

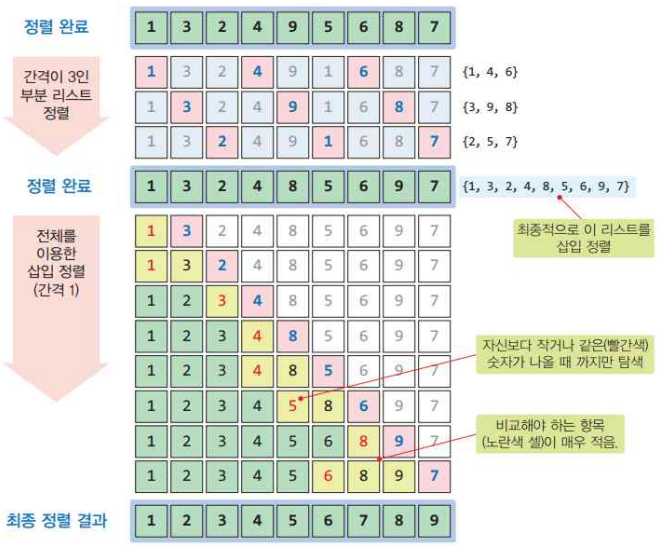
  ➔ 보다 적게 이동하여 제자리 찾을 수 있음

셸 정렬

- 리스트를 일정 간격(gap)의 부분 리스트로 나눔
  - 나뉘어진 각각의 부분 리스트를 삽입정렬 함
- 간격을 줄임
  - 부분 리스트의 수는 더 작아지고, 각 부분 리스트는 더 커짐
- 간격이 1이 될 때까지 이 과정 반복



셸 정렬



셸 정렬 알고리즘

```
def shell_sort(A) :                          # 셸 정렬 알고리즘
    n = len(A)
    gap = n//2                               # 최초의 gap:리스트 크기의 절반
    while gap > 0 :
        if (gap % 2) == 0 : gap += 1         # gap이 짝수이면 1을 더함
        for i in range(gap) :
            sortGapInsertion(A, i, n - 1, gap)
        print('    Gap=', gap, A)             # 중간 결과 출력용
        gap = gap//2
```

```
def sortGapInsertion(A, first, last, gap) :
    for i in range(first+gap, last+1, gap) :
        key = A[i]
        j = i - gap
        while j >= first and key<A[j] :      # 삽입 위치를 찾음
            A[j + gap] = A[j]                # 항목 이동
            j = j - gap
        A[j + gap] = key                     # 최종 위치에 삽입
```

```
C:₩WINDOWS₩system32₩cmd.exe
    Gap= 5 [1, 3, 2, 4, 9, 5, 6, 8, 7]
    Gap= 3 [1, 3, 2, 4, 8, 5, 6, 9, 7]
    Gap= 1 [1, 2, 3, 4, 5, 6, 7, 8, 9]
Shell      : [1, 2, 3, 4, 5, 6, 7, 8, 9]
```


셸 정렬 분석

- 장점
  - 불연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치교환으로 제자리 찾을 가능성 증대
  - 부분 리스트가 점진적으로 정렬된 상태가 되므로 삽입정렬 속도 증가
- 시간 복잡도
  - 최악의 경우: $O(n^2)$
  - 평균적인 경우: $O(n^{1.5})$

12.3 힙 정렬

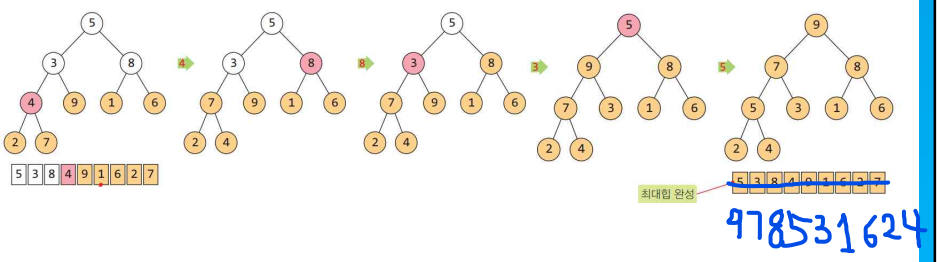
- 힙 클래스를 이용한 정렬
  - $O(n \log n)$
  - 추가적인 메모리를 필요로 함

```
def heapSort(data):
    heap = MaxHeap()                    # 최대 힙 사용 (8.3절)
    for n in data :                     # 모든 항목들을 힙에 넣음
        heap.insert(n)

    for i in range(1,len(data)+1):      # 1, 2, ..., n
        data[-i] = heap.delete()        # 맨 뒤에서 앞으로: -1, -2, ... -n
```


제자리 정렬로 구현한 힙 정렬

- 단계1: 리스트를 최대힙으로 만듦
- 단계2: 최대힙을 정렬된 리스트로 만듦

- Heapify: 최대 힙을 만드는 과정



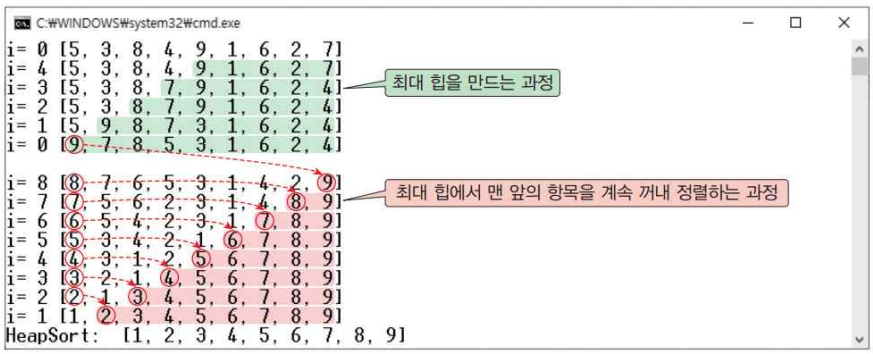
Heapify 알고리즘

```
def heapify(arr, n, i):
    largest = i       # i번째가 가장 크다고 하자.
    l = 2 * i + 1     # 왼쪽 자식: left = 2*i + 1 (배열 0번을 사용함)
    r = 2 * i + 2     # 오른쪽 자식: right = 2*i + 2 (배열 0번을 사용함)

    if l < n and arr[i] < arr[l]: largest = l              # 교환조건 검사
    if r < n and arr[largest] < arr[r]: largest = r        # 교환조건 검사
       if largest != i:                                    # 교환이 필요하면
          arr[i],arr[largest] = arr[largest],arr[i]        # 교환
          heapify(arr, n, largest)                      # 순환적으로 자식노드로 내려감
```


힙 정렬 알고리즘

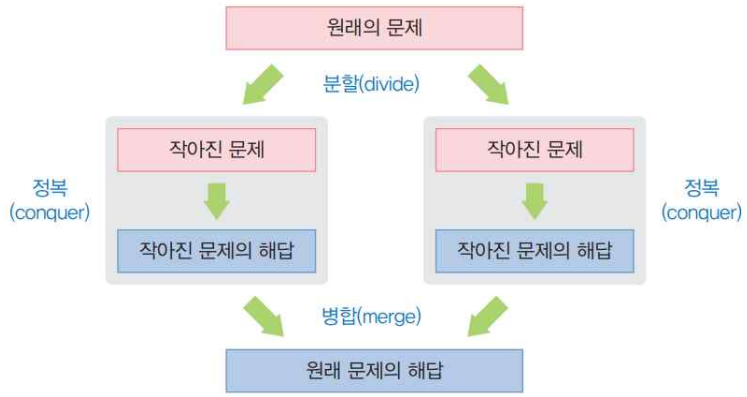
```
def heapSort(arr):
    n = len(arr)                    정렬되지 않은 배열 → 최대 힙
    print("i=", 0, arr)             # 중간결과 출력용
    for i in range(n//2, -1, -1):   # 최대 힙을 만듦: i: n//2, ..., 1, 0
        heapify(arr, n, i)          # heap 조건을 맞춤(downheap)
        print("i=", i, arr)         # 중간결과 출력용
    print()                         # 중간결과 출력용
                                    최대 힙 → 정렬된 배열
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # 루트를 뒤쪽으로 옮김. 교체
        heapify(arr, i, 0)          # heap 조건을 맞춤(downheap)
        print("i=", i, arr)         # 중간결과 출력용
```

실행 결과



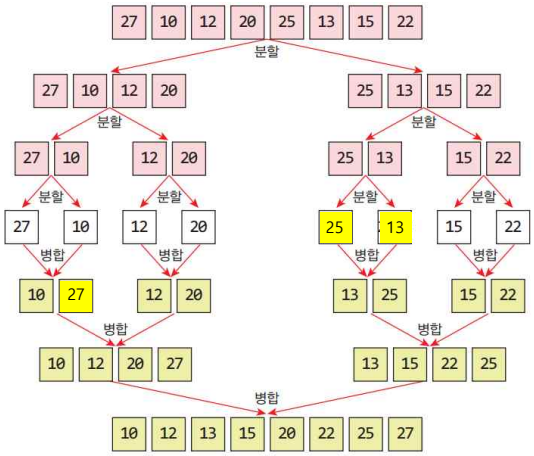
12.4 병합 정렬

- 분할 정복(divide and conquer) 방법
  - 문제를 보다 작은 2개의 문제로 분리하고 각 문제를 해결한 다음, 결과를 모아서 원래의 문제를 해결하는 전략



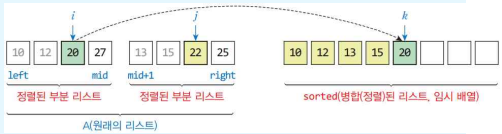
병합 정렬 알고리즘

```
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)
```

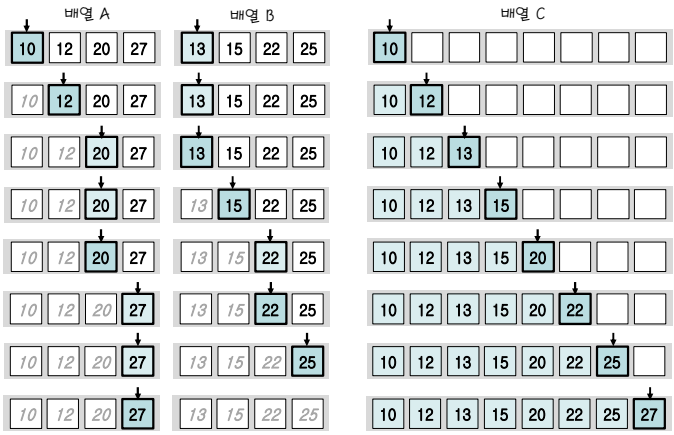


병합 알고리즘

```
def merge(A, left, mid, right) :
    global sorted                   # 병합을 위한 추가적인 배열
    k = left                        # 배열 C(정렬될 리스트)의 인덱스
    i = left                        # 배열 A의 인덱스
    j = mid + 1                     # 배열 B의 인덱스
    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 :                                # 한쪽에 남아 있는 레코드의 일괄 복사
        sorted[k:k+right-j+1] = A[j:right+1]    # 리스트의 슬라이싱 이용
    else :
        sorted[k:k+mid-i+1] = A[i:mid+1]        # 리스트의 슬라이싱 이용
    A[left:right+1] = sorted[left:right+1]      # sorted를 원래 배열 A에 복사
```



병합 과정



병합 정렬 복잡도 분석

- 시간 복잡도
  - 비교 횟수
    - 크기 n인 리스트를 균등 분배하므로 log(n)개의 패스
    - 각 패스에서 레코드 n개를 비교 ➔ n번 비교연산
  - 이동 횟수
    - 각 패스에서 2n번 이동 발생 ➔ 전체 이동: $2n * \log(n)$
  - 시간 복잡도= $O(n \log n)$

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

12.5 퀵 정렬

- 분할 정복법 사용
  - 리스트를 2개의 부분리스트로 비 균등 분할
  - 각각의 부분리스트를 다시 퀵 정렬함(순환 호출)

퀵 정렬 알고리즘

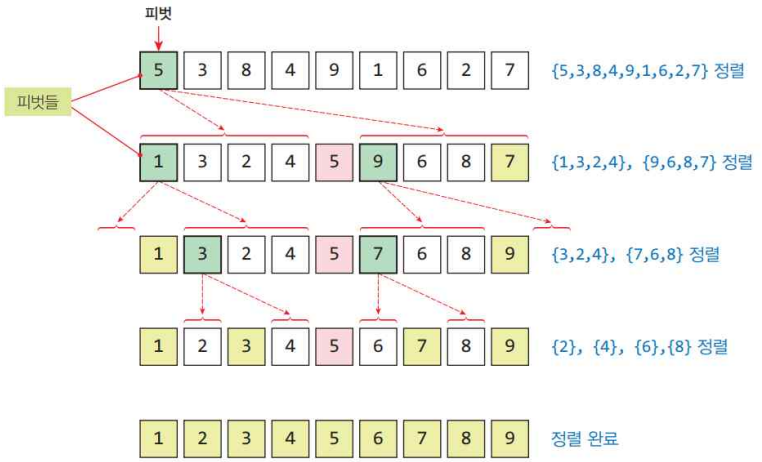
```
def quick_sort(A, left, right) :
    if left<right :                    # 정렬 범위가 2개 이상인 경우
        q = partition(A, left, right)  # 좌우로 분할
        quick_sort(A, left, q - 1)     # 왼쪽 부분리스트를 퀵 정렬
        quick_sort(A, q + 1, right)    # 오른쪽 부분리스트를 퀵 정렬
```

분할 과정

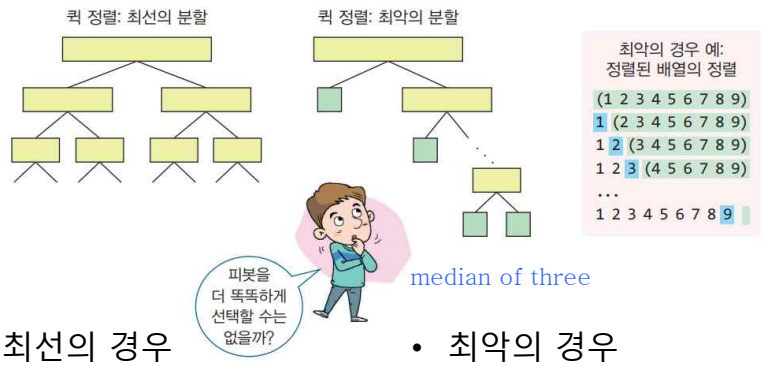
분할 함수 : partition()

```
def partition(A, left, right) :
    low = left + 1                          # 왼쪽 부분 리스트의 인덱스 (증가방향)
    high = right                            # 오른쪽 부분 리스트의 인덱스 (감소방향)
    pivot = A[left]                         # 피벗 설정
    while (low <= high) :                   # low와 high가 역전되지 않는 한 반복
        while low <= right and A[low] < pivot : low += 1
        while high >= left and A[high]> pivot : high-= 1

        if low < high :                     # 선택된 두 레코드 교환
            A[low], A[high] = A[high], A[low]

    A[left], A[high] = A[high], A[left]     # 마지막으로 high와 피벗 항목 교환
    return high                             # 피벗의 위치 반환
```

퀵 정렬 전체과정



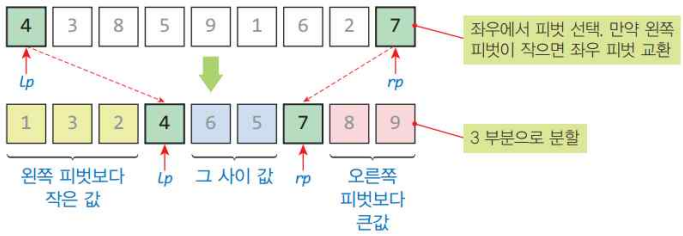
퀵정렬 복잡도 분석



- 최선의 경우
  - 균등 분할
  - 패스 수: log n
  - 복잡도: O(n log n)
- 최악의 경우
  - 이미 정렬된 리스트
  - 패스 수: n
  - 복잡도: $O(n^2)$

12.6 이중피벗 퀵 정렬

- 2개의 피벗을 사용하는 퀵 정렬



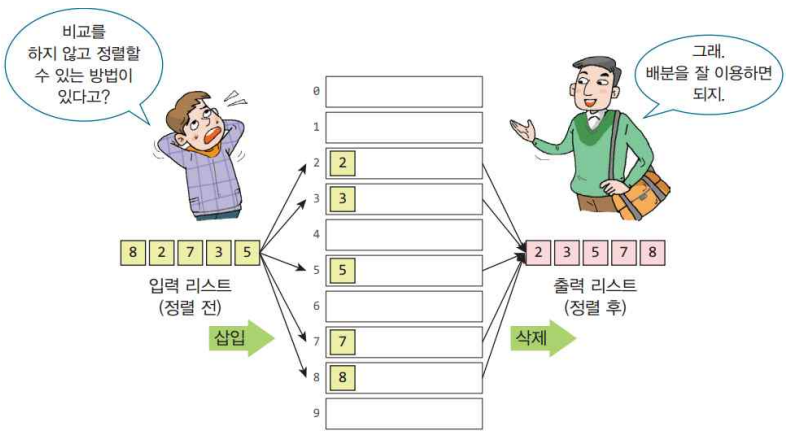

```
def dp_quick_sort(A, low, high) :
    if low < high :
        lp, rp = partitionDP(A, low, high)      # 좌우 피벗의 인덱스를 반환받음
        dp_quick_sort(A, low, lp-1)             # low ~ lp-1 정렬
        dp_quick_sort(A, lp+1, rp-1)            # lp+1 ~ rp-1 정렬
        dp_quick_sort(A, rp+1, high)          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


이중피벗 퀵 정렬의 분할 과정 예

12.7 기수 정렬

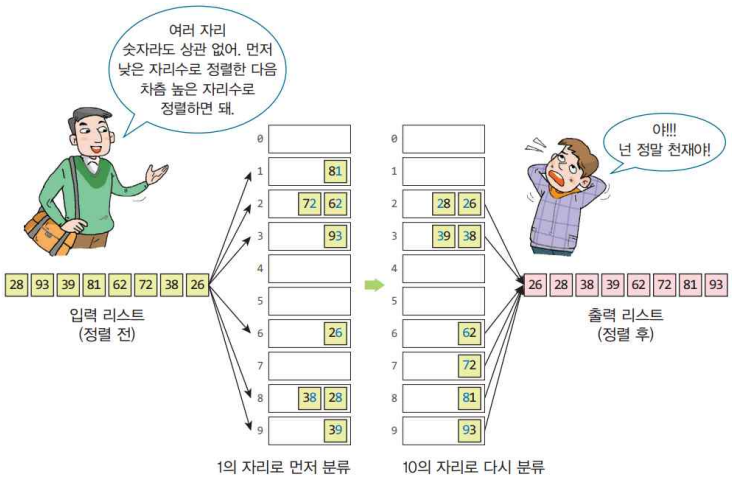
- 레코드를 비교하지 않고 분배하여 정렬 수행
  - 비교에 의한 정렬의 하한인 $O(n \log n)$ 보다 좋을 수 있음
  - 시간복잡도: $O(dn)$, 대부분 d<10 이하
- 아이디어
  - 단순히 자리수에 따라 숫자를 bucket에 넣었다가 꺼내면 정렬됨
- 단점
  - 정렬할 수 있는 레코드의 타입 한정
  - 정수나 단순 문자(알파벳 등)이어야만 함

기수 정렬의 기본 아이디어: 한 자리수



두 자릿수 기수 정렬

- 낮은 자릿수 분류 → 순서대로 읽음 → 높은 자릿수 분류

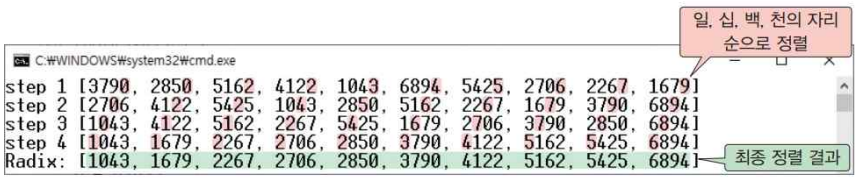


기수 정렬 알고리즘

```
def radix_sort(A) :
    queues = []                                  # 큐의 리스트
    for i in range(BUCKETS) :
        queues.append(Queue())                   # BUCKETS개의 큐 사용

    n = len(A)
    factor = 1                                   # 1의 자리부터 시작
    for d in range(DIGITS) :                     # 모든 자리에 대해
        for i in range(n) :                      # 자릿수에 따라 큐에 삽입
            queues[(A[i]//factor) % 10].put(A[i])  # 숫자를 삽입
        i = 0
        for b in range(BUCKETS) :                # 버킷에서 꺼내어 원래의 리스트로
            while not queues[b].empty() :        # b번째 큐가 비어있지 않는 동안.
                A[i] = queues[b].get()           # 원소를 꺼내 리스트에 저장
                i += 1
        factor *= 10                             # 그 다음 자리수로 간다.
        print("step", d+1, A)                    # 중간 과정 출력용 문장
```

테스트 프로그램

```
import random
BUCKETS = 10
DIGITS  = 4
data = []
for i in range(10) :
    data.append(random.randint(1,9999))        # 1~9999사이의 숫자 10개 생성
radix_sort(data)                               # 기수 정렬
print("Radix: ", data)                         # 결과 출력
```



기수 정렬 분석

- 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
- 시간 복잡도: $O(dn)$, 대부분 d<10 이하
- 실수, 한글, 한자로 이루어진 키는 정렬 못함

12.8 카운팅 정렬

- 분배 기반 정렬
- 일정한 범위를 가진 정수의 정렬에 효과적
- 예) [1, 4, 1, 2, 7, 5, 2] 정렬

인덱스	0	1	2	3	4	5	6	7	8	9
Count	0	2	2	0	1	1	0	1	0	0

인덱스	0	1	2	3	4	5	6	7	8	9
Count	0	2	4	4	5	6	6	7	7	7

인덱스	0	1	2	3	4	5	6
정렬 후	1	1	2	2	4	5	7

카운팅 정렬 알고리즘

```
MAX_VAL = 1000
def counting_sort(A):
    output = [0] * MAX_VAL          # 정렬 결과를 저장
    count  = [0] * MAX_VAL          # 각 숫자의 빈도를 저장

    for i in A:                     # 각 숫자별 빈도를 계산
        count[i] += 1

    for i in range(MAX_VAL):        # count[i]가 출력 배열에서
        count[i] += count[i-1]      # 해당 숫자의 위치가 되도록 수정

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

    for i in range(len(A)):         # 정렬 결과를 원래 배열에 복사
        A[i] = output[i]
```

12.9 정렬 알고리즘의 성능 비교

- 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	$O(n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

12장 연습문제, 실습문제

감사합니다!