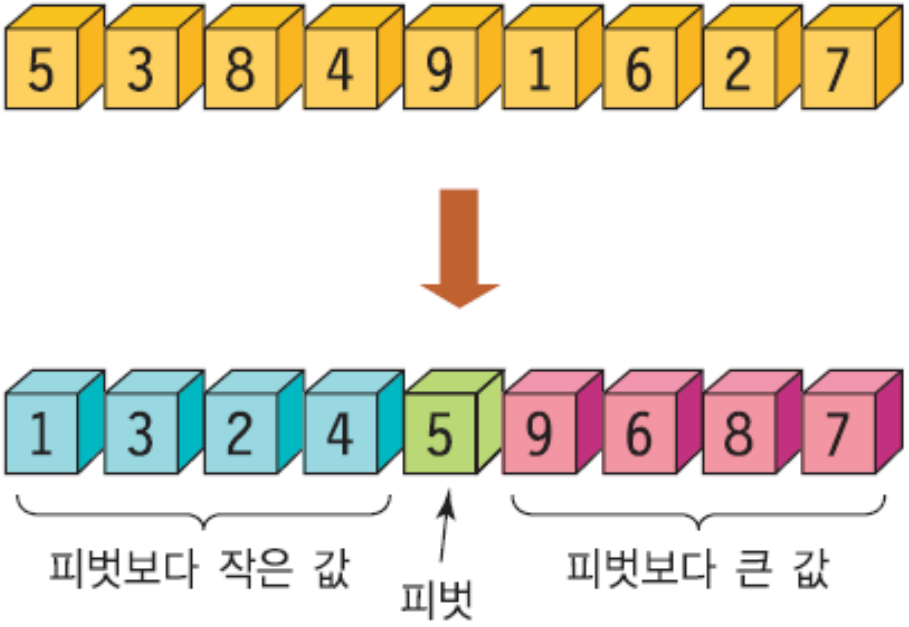
CSE2010 자료구조론

Week 12: Sorting 3

ICT융합학부 한진영

퀵정렬(Quick Sort)

- 평균적으로 가장 빠른 정렬 방법
- 분할 정복법(Divide & Conquer) 사용
- 리스트를 2개의 부분리스트로 비균등(pivot을 기준으로) 분할하고, 각각의 부분리스트를 다시 퀵 정렬함(재귀호출)



퀵정렬 알고리즘

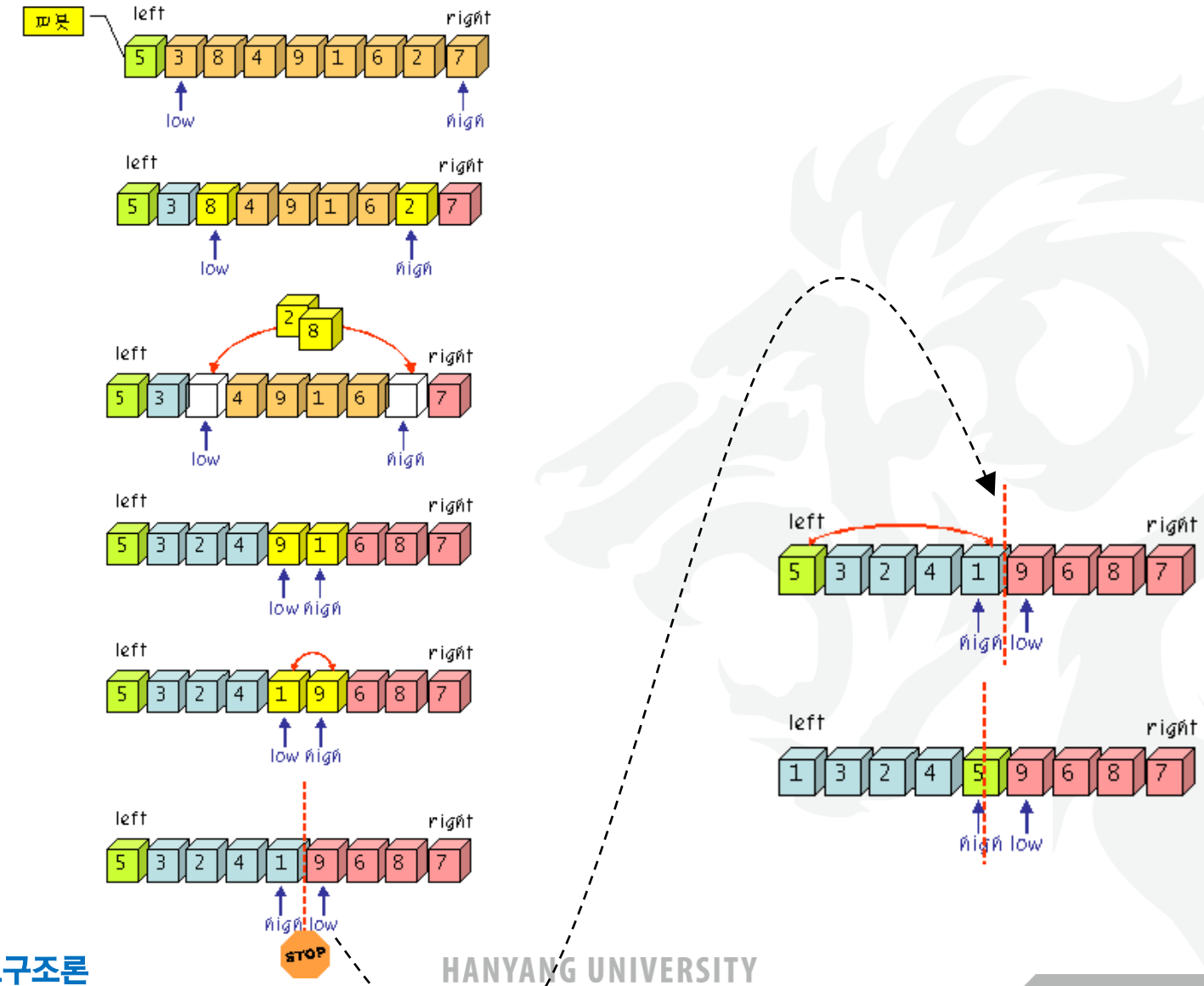
```
void quick_sort(int list[], int left, int right)
{
1.    if(left<right){
2.          int q=partition(list, left, right); //가장 중요한 함수
3.          quick_sort(list, left, q-1);
4.          quick_sort(list, q+1, right);
      }
}
```

1. 정렬할 범위가 2개 이상의 데이터이면
2. partition 함수 호출로 피벗을 기준으로 2개의 리스트로 분할 partition 함수의 반환 값이 피벗의 위치
3. left에서 피벗 바로 앞까지를 대상으로 순환호출(피벗 제외)
4. 피벗 바로 다음부터 right까지를 대상으로 순환호출(피벗 제외)

퀵정렬: 분할(Partition)

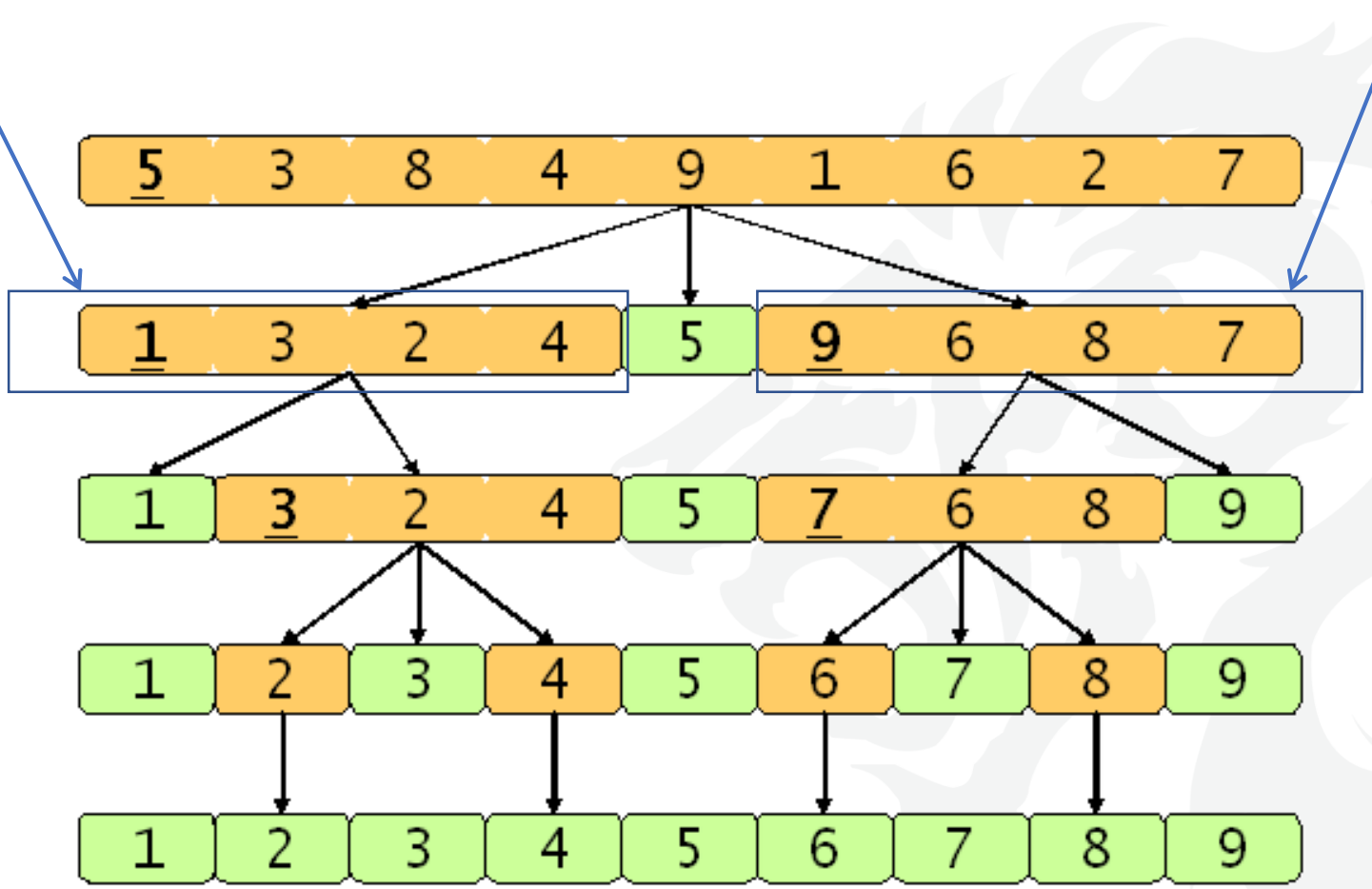
- Partition 함수
  - 데이터가 들어 있는 배열 list의 left부터 right 까지의 리스트를 피벗을 기준으로 2개의 부분 리스트로 나눔
  - 피벗보다 작은 데이터는 무조건 왼쪽, 큰 데이터는 모두 오른쪽으로 옮겨짐
- Partition 알고리즘
  - 피벗(pivot): 가장 왼쪽 숫자(입력 리스트의 첫 번째 데이터) 라고 가정
  - 두 개의 인덱스 변수 low(왼쪽 부분리스트 만드는데 사용)와 high(오른쪽 부분리스트 만드는데 사용)를 사용
  - Low(왼쪽에서 오른쪽으로 탐색): 피벗보다 작으면 통과, 크면 정지
  - High(오른쪽에서 왼쪽으로 탐색): 피벗보다 크면 통과, 작으면 정지
  - 정지된 위치의 숫자를 교환
  - low와 high가 교차하면 종료

분할 과정



퀵정렬 전체 과정

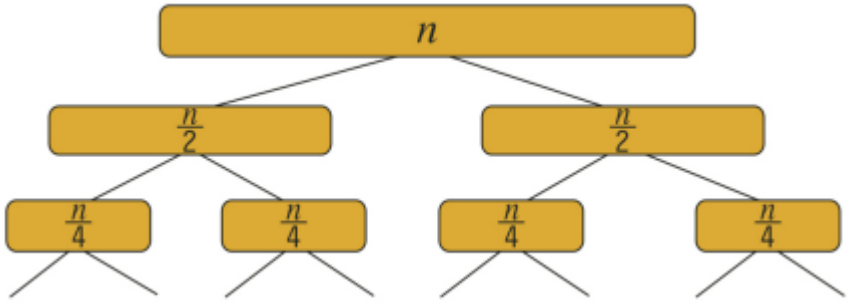
피벗을 제외하고, 왼쪽리스트(1 3 2 4), 오른쪽 리스트(9 6 8 7) 독립적으로 퀵 정렬



- 밑줄 친 숫자: 피벗

퀵정렬 복잡도 분석(1)

- 최선의 경우(거의 균등한 리스트로 분할되는 경우)
  - 패스 수: log(n)
    - 2->1
    - 4->2
    - 8->3
    - …
    - n->log(n)

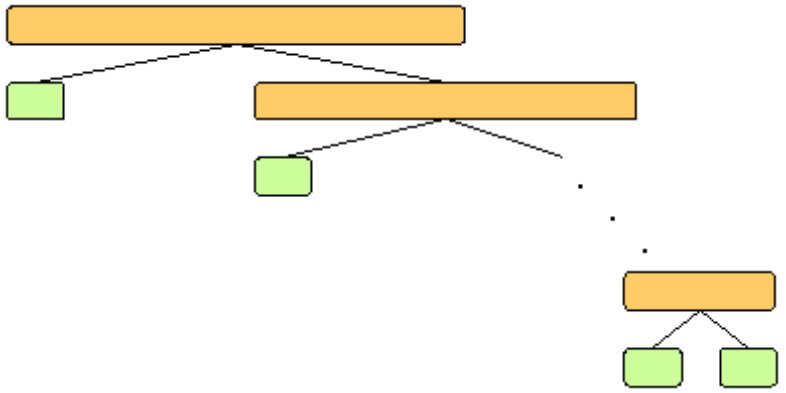


  - 각 패스 안에서의 비교횟수: n
  - 총 비교횟수: n*log(n)
  - 총 이동횟수: 비교횟수에 비하여 적으므로 무시 가능

퀵정렬 복잡도 분석(2)

- 최악의 경우(극도로 불균등한 리스트로 분할되는 경우)
  - 패스 수: n
  - 각 패스 안에서의 비교횟수: n
  - 총 비교횟수: n^2
  - 총 이동횟수: 무시 가능
  - 예: 이미 정렬된 리스트를 정렬할 경우

```
(1 2  3  4  5  6  7  8  9)
1 (2  3  4  5  6  7  8  9)
1  2 (3  4  5  6  7  8  9)
1  2  3 (4  5  6  7  8  9)
1  2  3  4 (5  6  7  8  9)
                    ...
1  2  3  4  5  6  7  8  9
```



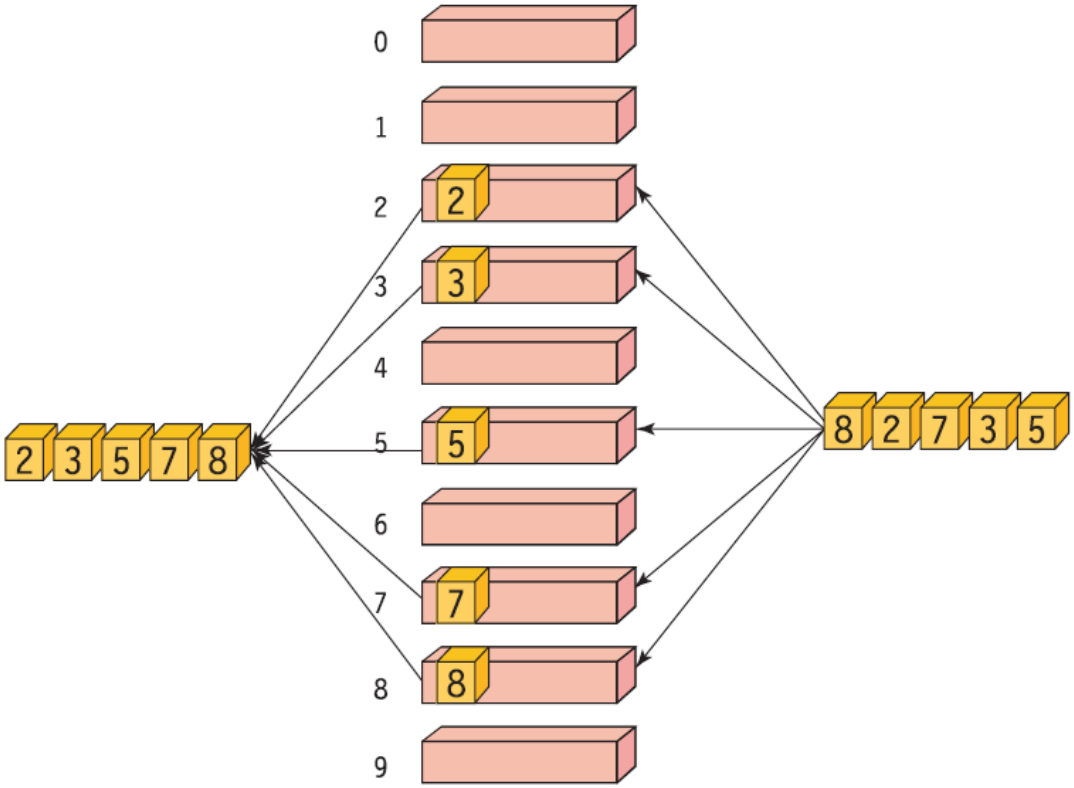
- 중간값(medium)을 피벗으로 선택하면 불균등 분할 완화 가능

기수정렬(Radix Sort)

- 대부분의 정렬 방법들은 레코드들을 비교함으로써 정렬 수행
- 기수 정렬(radix sort)은 레코드를 비교하지 않고 정렬 수행
  - 비교에 의한 정렬의 하한인 O(n*log(n)) 보다 좋을 수 있음
  - 기수 정렬은 O(dn) 의 시간적복잡도를 가짐(대부분 d<10 이하)
- 기수 정렬의 단점
  - 정렬할 수 있는 레코드의 타입 한정(실수, 한글, 한자 등은 정렬 불가)
    - 즉, 레코드의 키들이 동일한 길이를 가지는 숫자나 단순 문자(알파벳 등)이어야만 함
  - 추가적인 메모리 필요

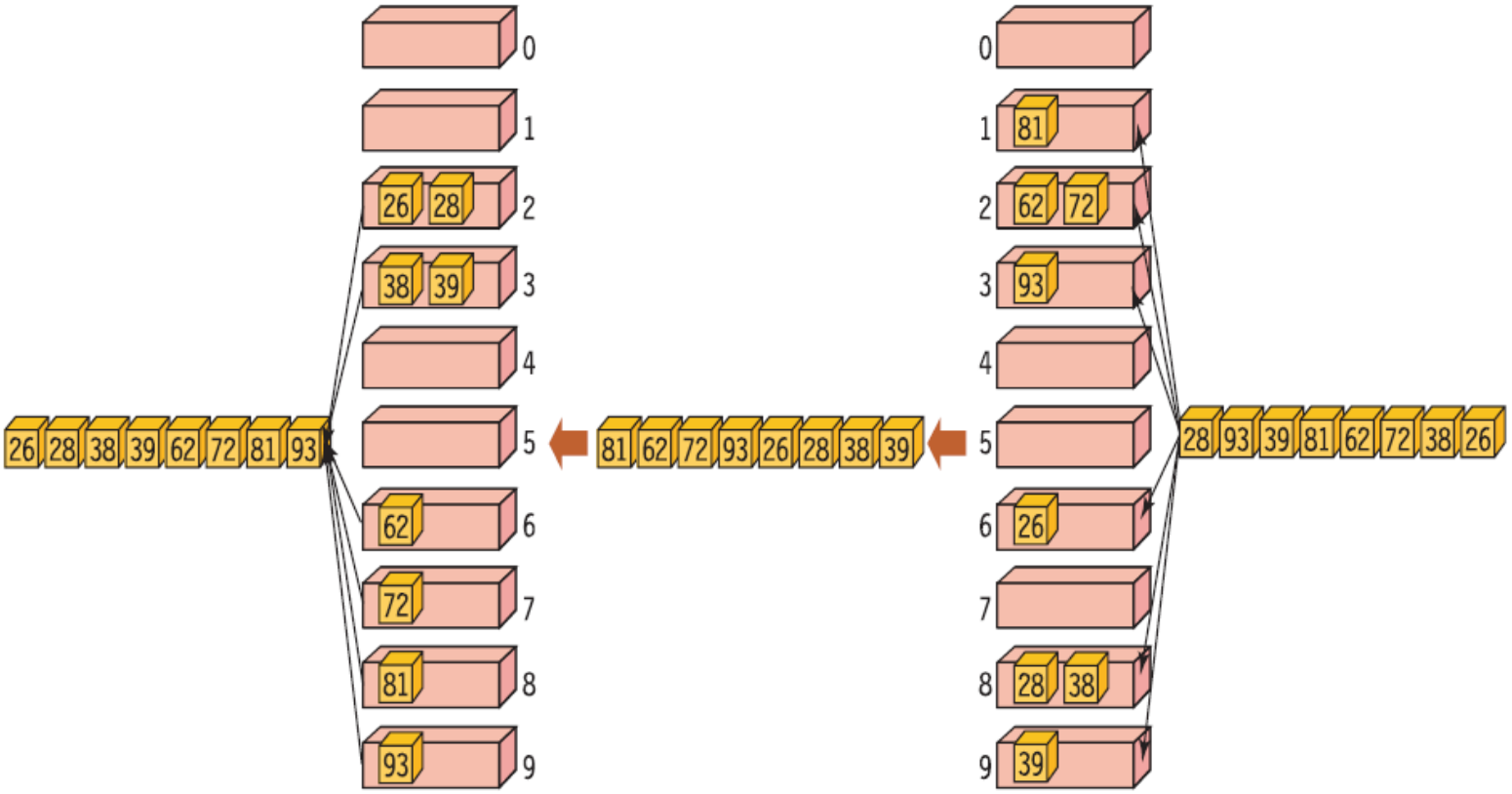
기수정렬 예(1)

- 예: 한자리수 (8, 2, 7, 3, 5)의 기수정렬
- 단순히 자리수에 따라 버켓(bucket)에 넣었다가 꺼내면 정렬됨



기수정렬 예(2)

- 만약 2자리수이면? (28, 93, 39, 81, 62, 72, 38, 26)
- 낮은 자리수로 먼저 분류한 다음, 순서대로 읽어서 다시 높은 자리수로 분류



기수정렬 알고리즘

```
RadixSort(list, n):

for d←LSD의 위치 to MSD의 위치 do
{
  d번째 자릿수에 따라 0번부터 9번 버켓에 넣는다.
  버켓에서 숫자들을 순차적으로 읽어서 하나의 리스트로 합친다.
  d++;
}
```

- 버켓은 큐로 구현
- 버켓의 개수는 키의 표현 방법과 밀접한 관계
  - 이진법을 사용한다면 버켓은 2개
  - 알파벳 문자를 사용한다면 버켓은 26개
  - 십진법을 사용한다면 버켓은 10개
- 예: 32비트의 정수의 경우, 8비트씩 나누면 -> 버켓은 256개로 늘어남. 대신 필요한 패스의 수는 4로 줄어듦.

기수정렬 복잡도 분석

- n개의 레코드, d개의 자릿수로 이루어진 키를 기수 정렬할 경우
  - 메인 루프는 자릿수 d번 반복
  - 큐에 n개 레코드 입력 수행
- O(dn) 의 시간적 복잡도
  - 키의 자릿수 d는 10 이하의 작은 수이므로 빠른 정렬임

정렬 알고리즘 비교

알고리즘	최선	평균	최악
삽입 정렬	$O(n)$	$O(n^2)$	$O(n^2)$
선택 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
버블 정렬	$O(n^2)$	$O(n^2)$	$O(n^2)$
쉘 정렬	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
퀵 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
히프 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
합병 정렬	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
기수 정렬	$O(dn)$	$O(dn)$	$O(dn)$

정렬 알고리즘 실험 예: 정수 60,000개

알고리즘	실행 시간(단위:sec)
삽입 정렬	7.438
선택 정렬	10.842
버블 정렬	22.894
쉘 정렬	0.056
히프 정렬	0.034
합병 정렬	0.026
퀵 정렬	0.014

Week 12: Sorting 3