CSE2010 자료구조론

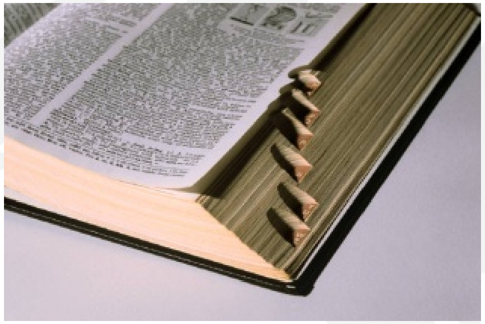
Week 12: Sorting

ICT융합학부 조용우

정렬(sorting)이란?

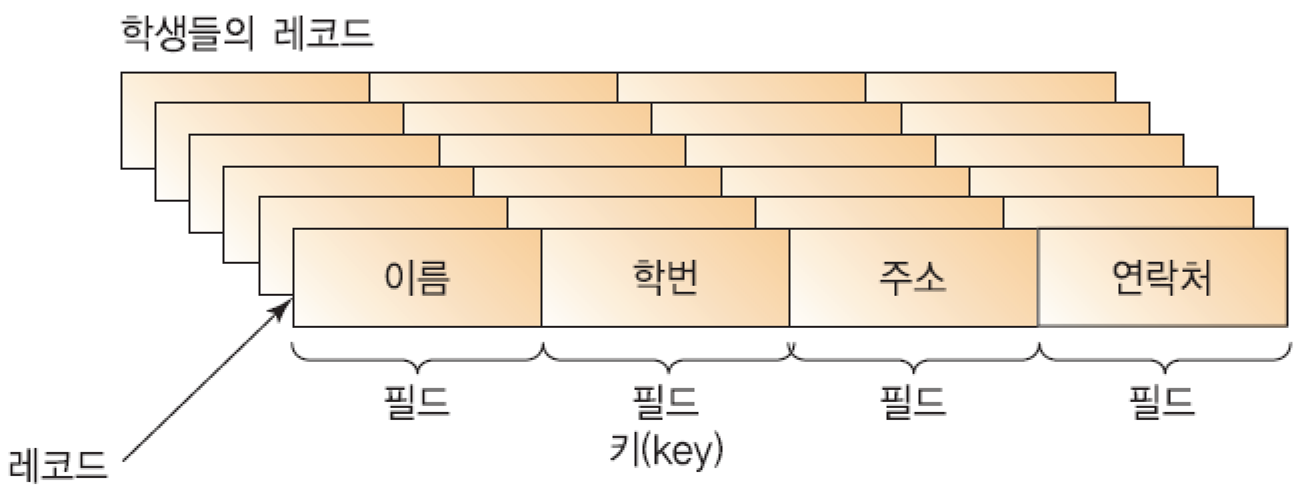
- 크기 순으로 오름차순이나 내림차순으로 나열하는 것
  - 정렬은 컴퓨터공학을 포함한 모든 과학기술분야에서 가장 기본적이고 중요한 알고리즘 중 하나

- 정렬은 자료 탐색에 있어서 필수적
  - (예) 만약 영어사전에서 단어들이 알파벳 순으로 정렬되어 있지 않다면?



정렬의 대상

- 일반적으로 정렬시켜야 될 대상은 레코드(record)
  - 레코드는 필드(field)라는 보다 작은 단위로 구성
  - 키(key) 필드: 레코드와 레코드를 구별



정렬 알고리즘(1)

- 모든 경우에 최적인 정렬 알고리즘은 없음
- 각 응용 분야에 적합한 정렬 방법 사용해야 함
  - 레코드 수의 많고 적음
  - 레코드 크기의 크고 작음
  - Key의 특성(문자, 정수, 실수 등)
  - 메모리 내부/외부 정렬
- 정렬 알고리즘의 평가 기준
  - 비교 횟수의 많고 적음
  - 이동 횟수의 많고 적음

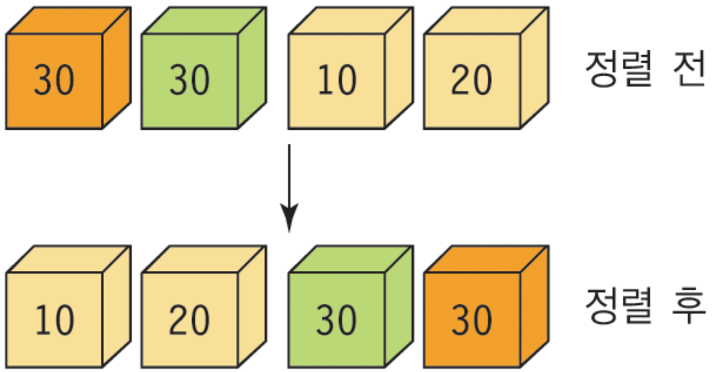
정렬 알고리즘(2)

- 단순하지만 비효율적인 방법: 삽입 정렬, 선택 정렬, 버블 정렬 등
- 복잡하지만 효율적인 방법: 퀵 정렬, 히프(heap) 정렬, 합병 정렬, 기수 정렬 등

- 내부 정렬(internal sorting): 모든 데이터가 주기억장치에 저장된 상태에서 정렬
- 외부 정렬(external sorting): 외부 기억장치에 대부분의 데이터가 있고 일부만 주기억장치에 저장된 상태에서 정렬

정렬 알고리즘의 안정성

- 정렬 알고리즘의 안정성(stability)
  - 입력 데이터에 동일한 키 값을 갖는 레코드가 여러 개일 경우, 동일한 키 값을 갖는 레코드들의 상대적인 위치가 정렬 후에도 바뀌지 않음
  - 안정성이 낮은 정렬의 예



정렬의 안정성이 중요한 경우, 안정성을 추구하는 삽입 정렬, 합병 정렬 등을 사용!

선택정렬(Selection Sort)

- 선택 정렬의 원리
  - 정렬된 왼쪽 리스트와 정렬 안된 오른쪽 리스트 가정
  - 초기에는 왼쪽 리스트는 비어 있고, 정렬할 숫자들은 모두 오른쪽 리스트에 존재
  - 오른쪽 리스트에서 최소값 선택하여 오른쪽 리스트의 첫번째 수와 교환
    - 왼쪽 리스트 크기 1 증가
    - 오른쪽 리스트 크기 1 감소
  - 오른쪽 리스트가 소진되면 정렬 완료

왼쪽 리스트	오른쪽 리스트	설명
()	(5,3,8,1,2,7)	초기 상태
(1)	(5,3,8,2,7)	1선택
(1,2)	(5,3,8,7)	2선택
(1,2,3)	(5,8,7)	3선택
(1,2,3,5)	(8,7)	5선택
(1,2,3,5,7)	(8)	7선택
(1,2,3,5,7,8)	()	8선택

선택정렬 알고리즘(1)

- 제자리 정렬(in-place sorting)
  - 입력 배열 이외에 추가적인 공간을 사용하지 않음
  - 입력 배열에서 최소값 발견 후, 이 최소값을 배열의 첫 번째 요소와 교환
  - 첫 번째 요소를 제외한 나머지 요소들 중에서 가장 작은 값 선택하고 이를 두 번째 요소와 교환
  - …

선택정렬 알고리즘(2)

```
selection_sort(A, n)

for i←0 to n-2 do // A[0]~A[n-2]까지 정렬되어 있으면, 이미 A[n-1]이 가장
                  // 큰 값이므로 n-1까지 정렬할 필요 없음
    least ← A[i], A[i+1],..., A[n-1] 중에서 가장 작은 값의 인덱스;
    A[i]와 A[least]의 교환;
    i++;
```

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void selection_sort(int list[], int n)
{   int i, j, least, temp;
    for(i=0; i<n-1; i++) {
        least = i;
        for(j=i+1; j<n; j++)        // 최소값 탐색
            if(list[j]<list[least]) least = j;
        SWAP(list[i], list[least], temp);
    }
}
```

선택정렬 실행 프로그램

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10000
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
int list[MAX_SIZE];
int n;

void selection_sort(int list[], int n)
{
  //...
}

void main()
{
        int i;
        n = MAX_SIZE;
        for(i=0; i<n; i++)                 // 난수 생성 및 출력
                list[i] = rand()%n;        // 난수 발생 범위 0~n

        selection_sort(list, n);  // 선택 정렬 호출

        for(i=0; i<n; i++)                 // 정렬 결과 출력
                printf("%d\n", list[i]);
}
```

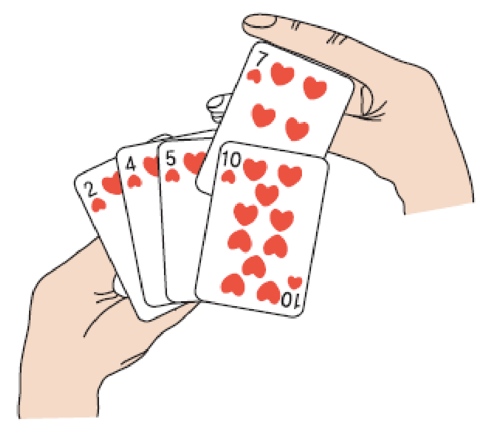
선택정렬 복잡도 분석

- 시간 복잡도: O(n^2)
  - 2개의 for 루프
- 안정성을 만족하지 않음
  - 값이 같은 레코드가 있을 경우, 상대적인 위치 변경 가능

^ 같거나 작은 것을 골라내는 코드라면 안정성이 없지만,
| 작은 것을 골라내는 코드라면 안정성 넘친다 !

삽입정렬(Insertion Sort)

- 손안의 카드를 정렬하는 방법과 유사 **<- 적절한 위치에 삽입한다**
  - 새로운 카드를 기존의 정렬된 카드 사이의 올바른 자리를 찾아 삽입
- 삽입정렬
  - 정렬되어 있는 리스트에 새로운 레코드를 올바른 위치에 삽입하는 과정 반복



삽입정렬 과정


5 3 8 1 2 7
초기 상태
5 3 8 1 2 7
3을 삽입
3 5 8 1 2 7
8은 이미 제자리에
3 5 8 1 2 7
1을 삽입
1 3 5 8 2 7
2를 삽입
1 2 3 5 8 7
7을 삽입
1 2 3 5 7 8
정렬 완료

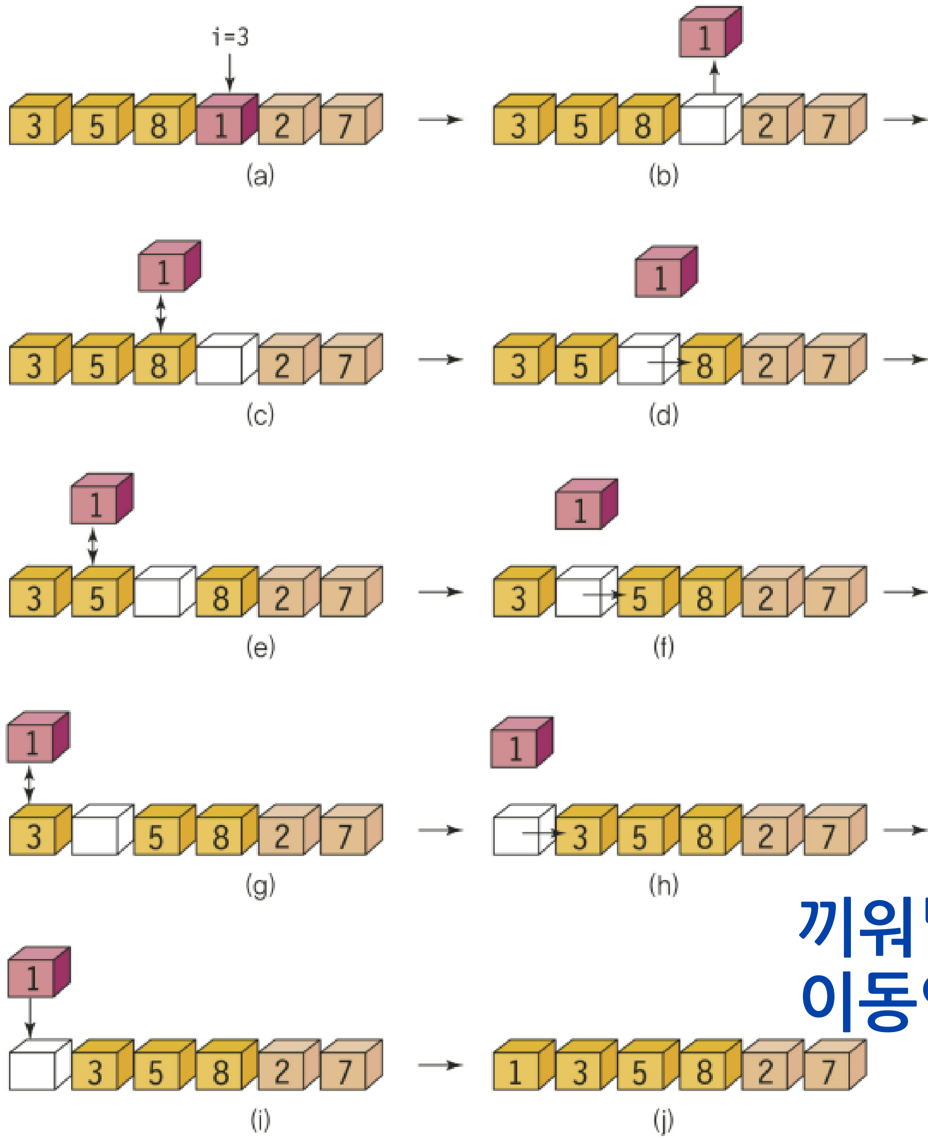
삽입정렬 알고리즘

```
insertion_sort(A, n)

1. for i ← 1 to n-1 do
2.     key ← A[i];
3.     j ← i-1;
4.     while j≥0 and A[j]>key do
5.             A[j+1] ← A[j];
6.             j ← j-1;
7.      A[j+1] ← key
```

- 1. 인덱스 1부터 시작
- 2. 현재 삽입될 숫자인 i번째 정수를 key 변수로 복사
- 3. 현재 정렬된 배열은 i-1까지 이므로, i-1번째부터 역순으로 조사
- 4. j값이 음수가 아니어야 되고 key값보다 정렬된 배열에 있는 값이 크면 수행
- 5. j번째를 j+1번째로 이동
- 6. j를 하나 감소
- 7. j번째 정수가 key보다 작으므로 j+1번째가 key값이 들어갈 위치

삽입정렬 알고리즘 수행 예



끼워넣을 때 마다
이동연산이 발생함 ㅠ

삽입정렬 코드

```
void insertion_sort(int list[], int n)
{
   int i, j, key;
   for(i=1; i<n; i++){
     key = list[i];
     for(j=i-1; j>=0 && list[j]>key; j--)
          list[j+1] = list[j];          // 레코드의 오른쪽 이동
     list[j+1] = key;
   }
}
```

삽입정렬 복잡도 분석

- 최선의 경우 O(n): 이미 정렬되어 있는 경우
  - 비교: *n-1* 번
- 최악의 경우 $O(n^2)$: 역순으로 정렬되어 있는 경우
  - 모든 단계에서 앞에 놓인 자료 전부 이동
- 평균의 경우 $O(n^2)$

- 특징
  - 많은 이동 필요 -> 레코드가 클 경우 불리
  - 안정된 정렬 방법
  - 대부분 정렬되어 있으면 매우 효율적

버블정렬(Bubble Sort)

- 인접한 2개의 레코드를 비교하여 순서대로 되어 있지 않으면 서로 교환
- 이러한 비교-교환 과정을 리스트의 왼쪽 끝에서 오른쪽 끝까지 반복(스캔)

버블정렬 알고리즘

```
BubbleSort(A, n)

for i←n-1 to 1 do
   for j←0 to i-1 do //하나의 스캔 과정
           j와 j+1번째의 요소가 크기 순이 아니면 교환
           j++;
   i--;
```

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void bubble_sort(int list[], int n)
{
   int i, j, temp;
   for(i=n-1; i>0; i--){
      for(j=0; j<i; j++)     // 앞뒤의 레코드를 비교한 후 교체
          if(list[j]>list[j+1])
               SWAP(list[j], list[j+1], temp);
   }
}
```

버블정렬 복잡도

- 비교 횟수(최상, 평균, 최악의 경우 모두 일정)

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- 이동 횟수
  - 역순으로 정렬된 경우(최악의 경우) :
    - 이동 횟수 = 3 * 비교 횟수
  - 이미 정렬된 경우(최선의 경우) :
    - 이동 횟수 = 0
  - 평균의 경우 : O(n^2)

이동 연산이 너무 많아져~

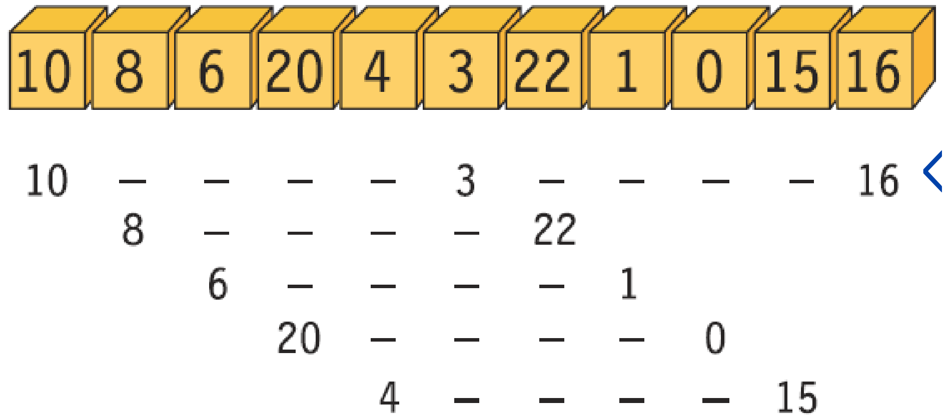
- 레코드의 이동 과다
  - 이동 연산은 비교 연산 보다 더 많은 시간이 소요됨

셸정렬(Shell Sort)

- 삽입정렬이 어느 정도 정렬된 리스트에서 대단히 빠른 것에 착안
  - 삽입 정렬은 요소들이 이웃한 위치로만 이동하므로, 많은 이동에 의해서만 요소가 제자리를 찾아감
  - 요소들이 멀리 떨어진 위치로 이동할 수 있게 하면, 보다 적게 이동하여 제자리 찾을 수 있음
- 전체 리스트를 일정 간격의 부분 리스트로 나눔
  - 나뉘어진 각각의 부분 리스트를 삽입정렬 함
- 간격을 줄임
  - 부분 리스트의 수는 더 작아짐
  - 각 부분 리스트의 크기는 더 커짐
- 간격이 1이 될 때 까지 부분 리스트의 삽입 정렬 과정 반복
- 평균적인 경우의 시간복잡도 : O($n^{1.5}$)

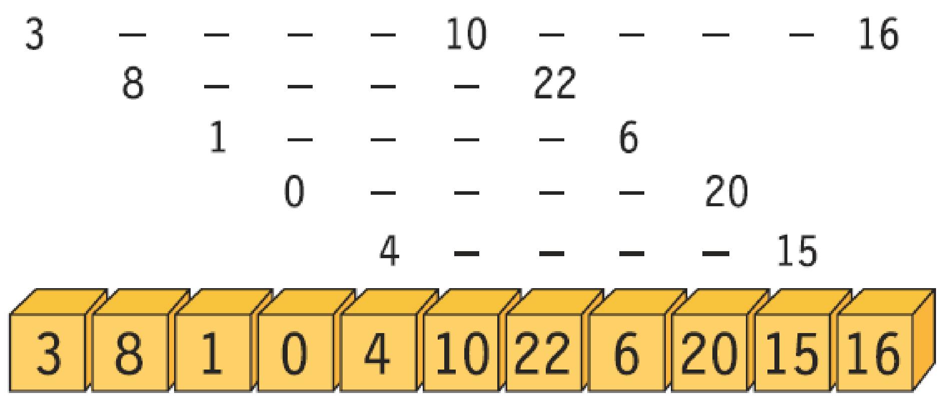
이동 연산이 줄고
크기가 줄어 들고
어느 정도 정렬됨

셀정렬 예(1)



<- 세개만으로 삽입정렬함

(a) 간격 5로 만들어진 부분 리스트



(b) 부분 리스트들이 정렬된 후의 리스트

셸정렬 예(2)

입력 배열	10	8	6	20	4	3	22	1	0	15	16
간격 5일때의 부분 리스트	10					3					16
		8					22				
			6					1			
				20					0		
					4					15	
부분 리스트 정렬 후	3					10					16
		8					22				
			1					6			
				0					20		
					4					15	
간격 5 정렬 후의 전체 배열	3	8	1	0	4	10	22	6	20	15	16
간격 3일때의 부분 리스트	3			0			22			15	
		8			4			6			16
			1			10			20		
부분 리스트 정렬 후	0			3			15			22	
		4			6			8			16
			1			10			20		
간격 3 정렬 후의 전체 배열	0	4	1	3	6	10	15	8	20	22	16
간격 1 정렬 후의 전체 배열	0	1	3	4	6	8	10	15	16	20	22

삽입정렬을 반복함

셸정렬 구현

```
// gap 만큼 떨어진 요소들을 삽입 정렬
// 정렬의 범위는 first에서 last
inc_insertion_sort(int list[], int first, int last, int gap)
{
        int i, j, key;
        for(i=first+gap; i<=last; i=i+gap){
          key = list[i];
          for(j=i-gap; j>=first && key<list[j];j=j-gap)
                  list[j+gap]=list[j];
          list[j+gap]=key;
        }
}

void shell_sort( int list[], int n )                    // n = size
{
        int i, gap;
        for( gap=n/2; gap>0; gap = gap/2 ) { // 처음에 gap 간격을 n/2로 하고, 각 패스마다 간격
                                              // 을 절반씩 줄이는 방식 이용
          if( (gap%2) == 0 ) gap++;           // gap이 짝수이면 1을 더함
          for(i=0;i<gap;i++)                  // 부분 리스트의 개수는 gap
                  inc_insertion_sort(list, i, n-1, gap);
        }
}
```

셸정렬 복잡도

- 셸 정렬의 장점
  - 불연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치 교환으로 제자리 찾을 가능성 증대
  - 부분 리스트가 점진적으로 정렬된 상태가 되므로 삽입 정렬 속도 증가

멀리 이동할 확률이 낮아지고
원래의 자리에 있을 확률이 높아짐

- 시간적 복잡도
  - 최악의 경우 O(n^2)
  - 평균적인 경우 O($n^{1.5}$)

합병정렬(Merge Sort)

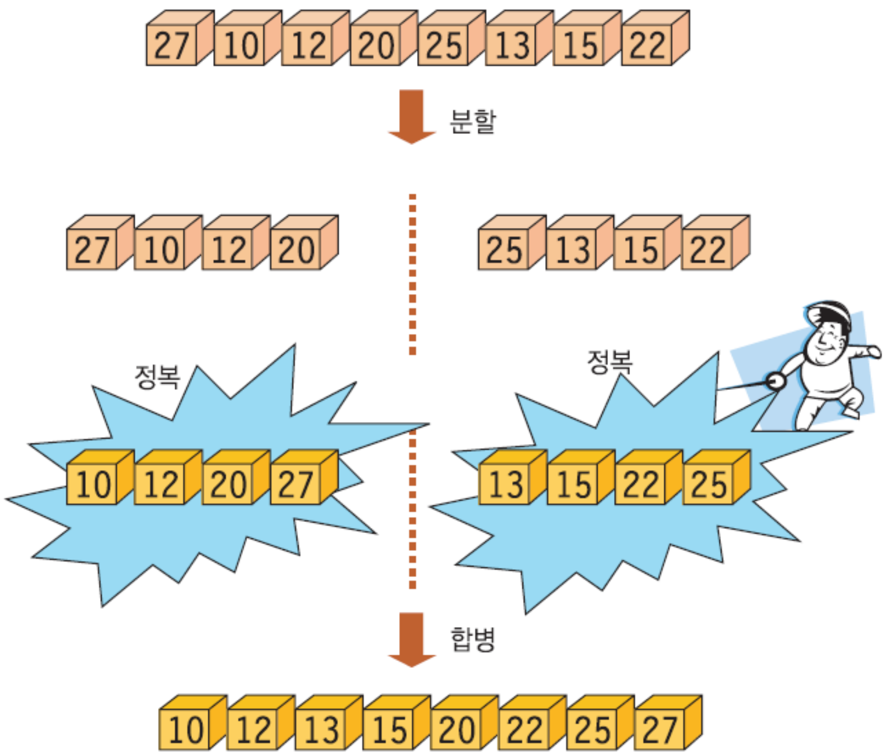
- 리스트를 두 개의 균등한 크기로 분할하고 분할된 부분리스트를 정렬
- 정렬된 두 개의 부분 리스트를 합하여 전체 리스트를 정렬함
- 분할 정복(divide and conquer) 방법 사용
  - 문제를 보다 작은 2개의 문제로 분리하고 각 문제를 해결한 다음,결과를 모아서 원래의 문제를 해결하는 전략
  - 분리된 문제가 아직도 해결하기 어렵다면(즉 충분히 작지 않다면) 분할정복방법을 다시 적용(재귀호출 이용)

1.분할(Divide): 배열을 같은 크기의 2개의 부분 배열로 분할
2.정복(Conquer): 부분배열을 정렬한다. 부분배열의 크기가 충분히 작지 않으면 재귀호출을 이용하여 다시 분할정복기법 적용
3.결합(Combine): 정렬된 부분배열을 하나의 배열에 통합

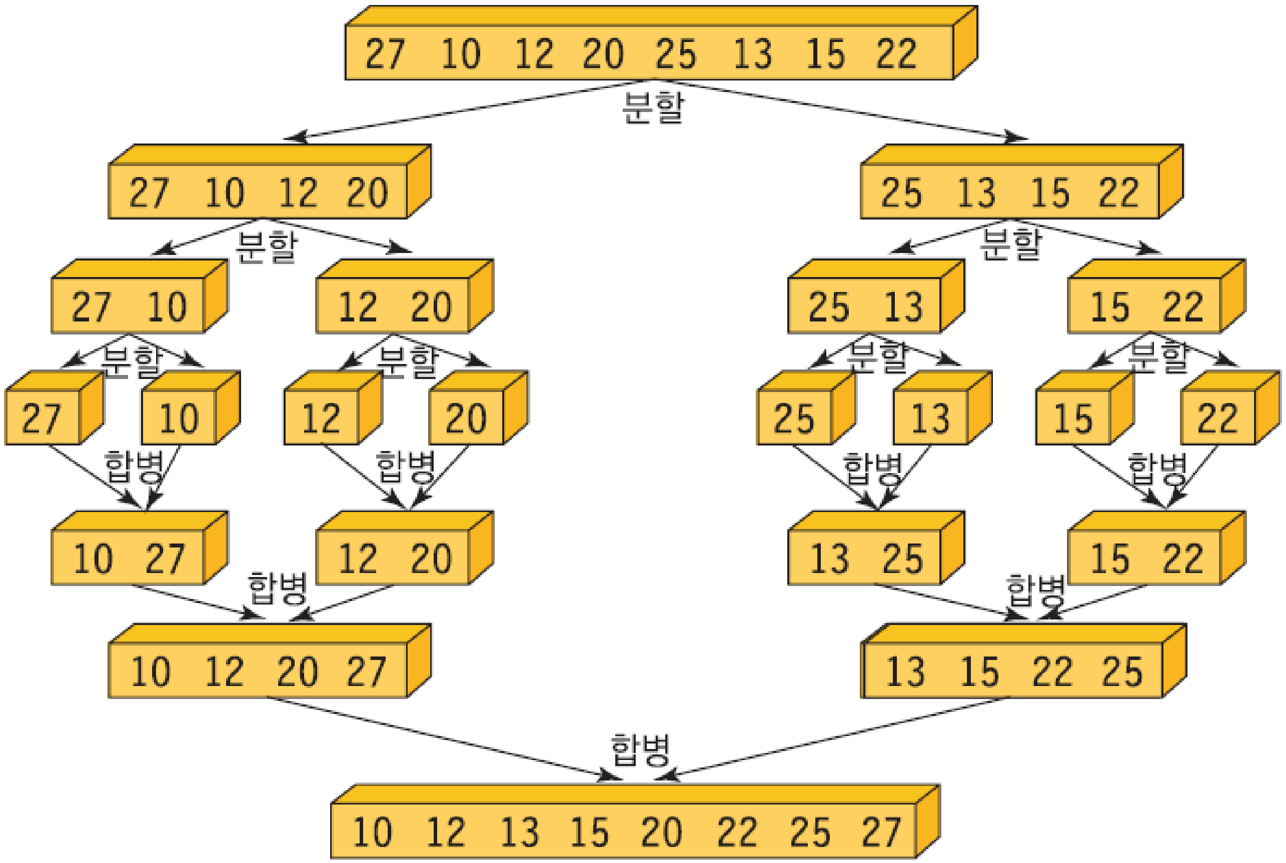
합병정렬 아이디어: D & C

입력: (27 10 12 20 25 13 15 22)

1.분할(Divide) : 전체 배열을 (27 10 12 20), (25 13 15 22) 2개 부분배열로 분리
2.정복(Conquer): 각 부분배열 정렬 (10 12 20 27), (13 15 22 25)
3.결합(Combine): 2개의 정렬된 부분배열 통합 (10 12 13 15 20 22 25 27)



합병정렬 과정



27 10 12 20 25 13 15 22
분할
27 10 12 20
25 13 15 22
분할
분할
27 10
12 20
25 13
15 22
분할
분할
분할
분할
27
10
12
20
25
13
15
22
합병
합병
합병
합병
10 27
12 20
13 25
15 22
합병
합병
10 12 20 27
13 15 22 25
합병
10 12 13 15 20 22 25 27

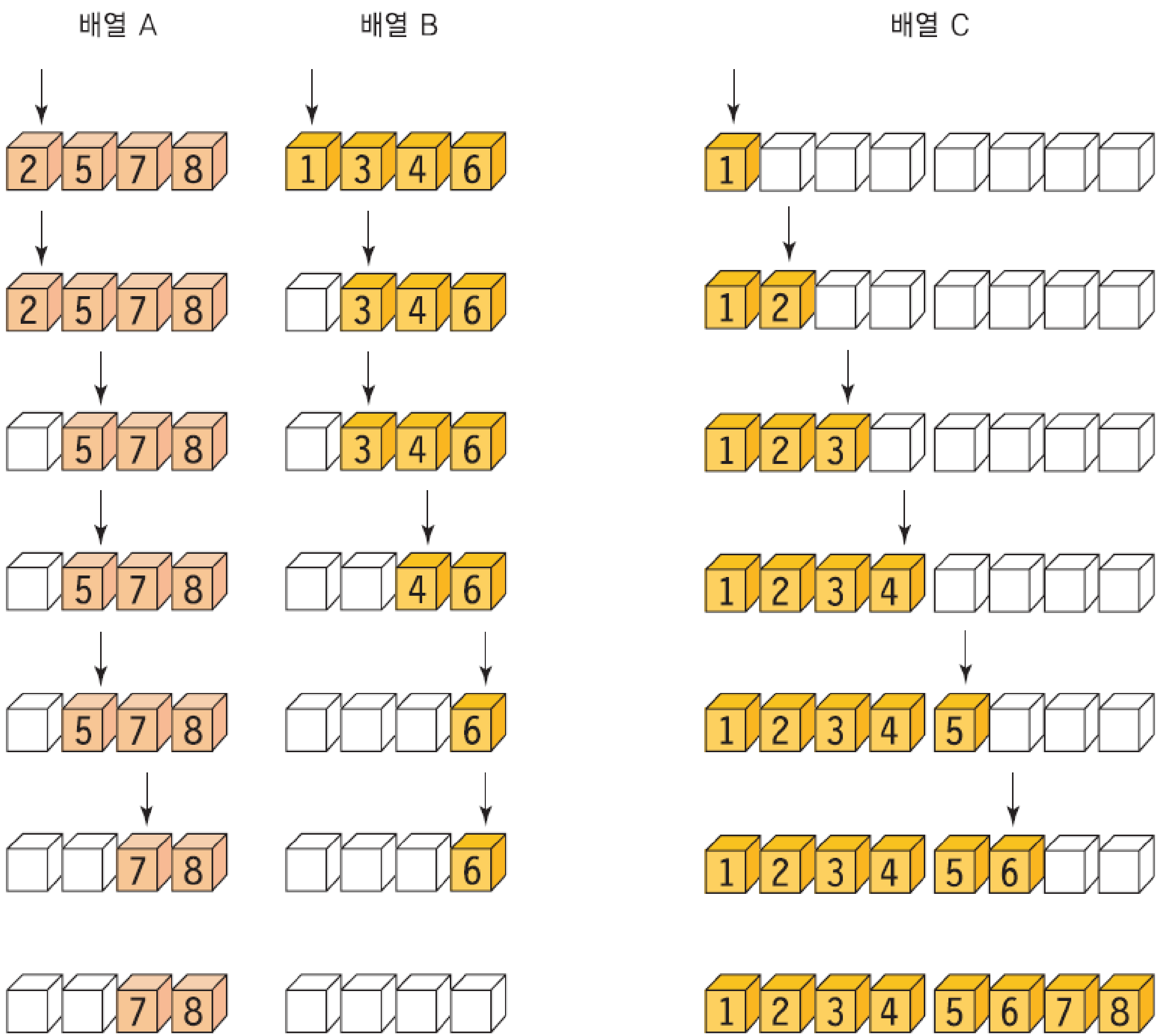
합병정렬 알고리즘

```
merge_sort(list, left, right)

1. if left < right
2.    mid = (left+right)/2;
3.    merge_sort(list, left, mid);
4.    merge_sort(list, mid+1, right);
5.    merge(list, left, mid, right);
```

1. 만약 나누어진 구간의 크기가 1이상이면
2. 중간 위치 계산
3. 왼쪽 부분 배열 정렬: merge_sort 함수 순환 호출
4. 오른쪽 부분 배열을 정렬: merge_sort 함수 순환 호출
5. 정렬된 2개의 부분 배열을 통합하여 하나의 정렬된 배열 만듦

합병과정



배열 A
배열 B
배열 C
2 5 7 8
1 3 4 6
1
2 5 7 8
3 4 6
1 2
5 7 8
3 4 6
1 2 3
5 7 8
4 6
1 2 3 4
5 7 8
6
1 2 3 4 5
7 8
6
1 2 3 4 5 6
7 8
1 2 3 4 5 6 7 8

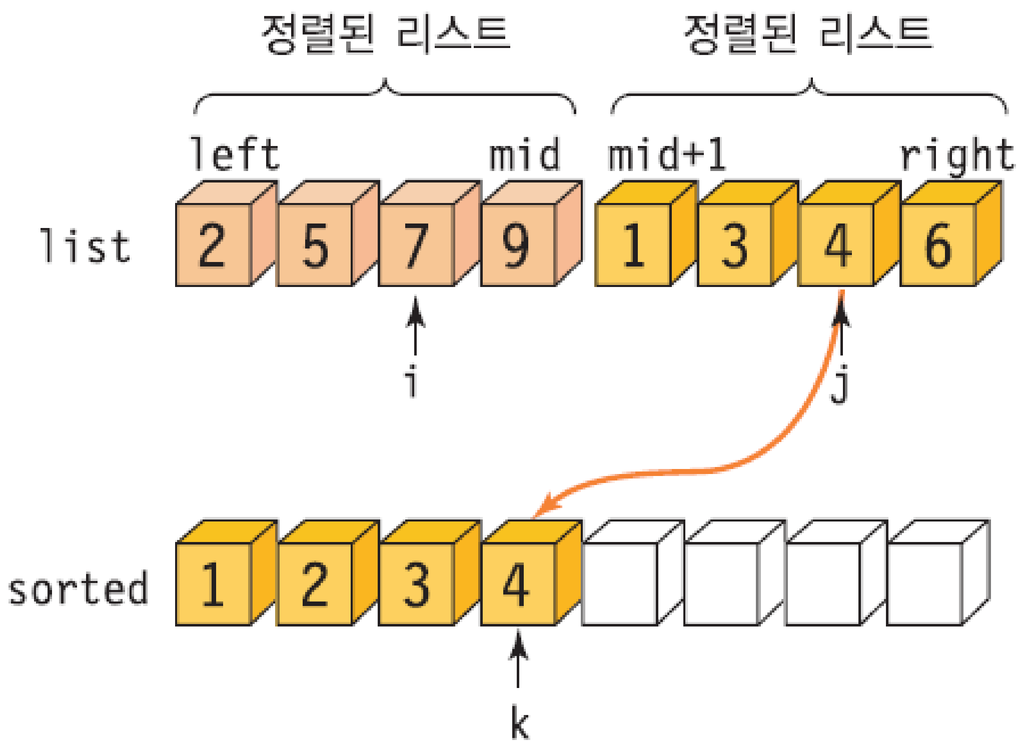
합병 알고리즘

```
merge(list, left, mid, right)
// 2개의 인접한 배열 list[left..mid]와 list[mid+1..right]를 합병
i←left;
j←mid+1;
k←left;
sorted 배열 생성; //합병된 리스트를 임시로 저장하기 위한 배열
while i≤left and j≤right do
      if(list[i]<list[j])
        then
          sorted[k]←list[i];
          k++;
          i++;
        else
          sorted[k]←list[j];
          k++;
          j++;

요소가 남아있는 부분배열을 sorted로 복사한다;
sorted를 list로 복사한다;
```

합병 중간 상태

배열 list에 있는 두 개의 list를 배열 sorted에 합병하는 세부 과정



합병정렬 코드(1)

```
void merge_sort(int list[], int left, int right)
{
        int mid;
        if(left<right)
        {
                mid = (left+right)/2;                   // 리스트의 균등분할
                merge_sort(list, left, mid);        // 부분리스트 정렬
                merge_sort(list, mid+1, right); //부분리스트 정렬
                merge(list, left, mid, right);      // 합병
        }
}
```

합병정렬 코드(2)

```
int sorted[MAX_SIZE]; // 추가 공간이 필요
// i는 정렬된 왼쪽리스트에 대한 인덱스
// j는 정렬된 오른쪽리스트에 대한 인덱스
// k는 정렬될 리스트에 대한 인덱스
void merge(int list[], int left, int mid, int right)
{
        int i, j, k, l;
        i=left; j=mid+1; k=left;
        // 분할 정렬된 list의 합병
        while(i<=mid && j<=right){
              if(list[i]<=list[j]) sorted[k++] = list[i++];
              else sorted[k++] = list[j++];
        }
        if(i>mid)    // 남아 있는 레코드의 일괄 복사
              for(l=j; l<=right; l++)
                      sorted[k++] = list[l];
        else   // 남아 있는 레코드의 일괄 복사
              for(l=i; l<=mid; l++)
                      sorted[k++] = list[l];
        // 배열 sorted[]의 리스트를 배열 list[]로 복사
        for(l=left; l<=right; l++)
                list[l] = sorted[l];
}
```

합병정렬 복잡도(1)

- 합병 정렬은 순환 호출 구조로 되어있음
- 레코드 개수 n이 2의 거듭제곱이라 가정하면($n=2^k$) 순환 호출의 깊이는 k가 됨, 여기서 $k=\log_2 n$
- Merge 함수에서 비교연산과 이동연산 수행
    - 순환 호출 깊이만큼 합병 연산 필요함
- 비교 연산
    - 크기 n인 리스트를 정확히 균등 분배하므로 log(n)개의 패스
    - 각 패스에서 리스트의 모든 레코드 n개를 비교하므로 n번의 비교 연산, 합병단계가 $k=\log_2 n$만큼 있으므로 총 비교연산은 $n\log_2 n$

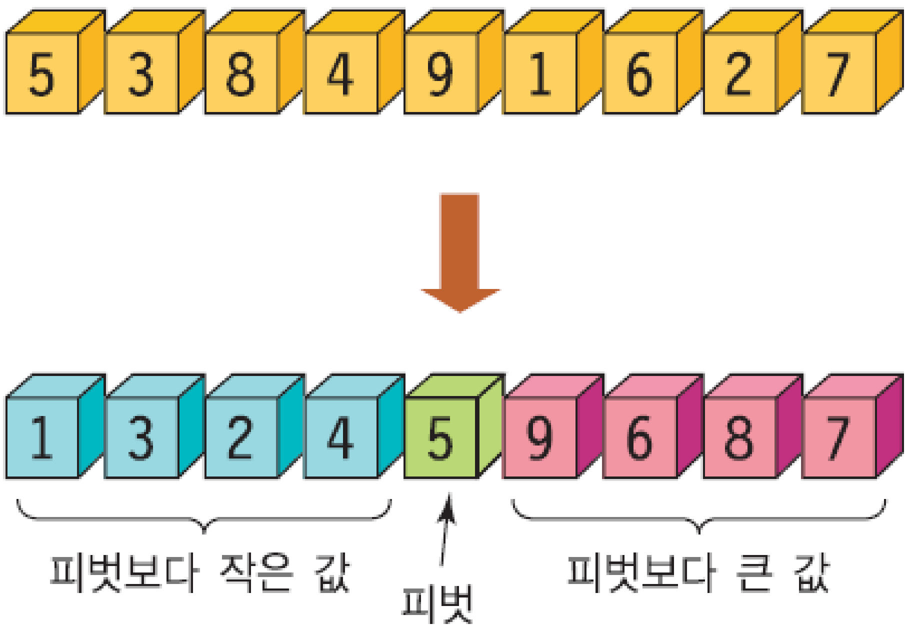
합병정렬 복잡도(2)

- 이동 연산
  - 레코드의 이동이 각 패스에서 2n번 발생하므로 전체 레코드의 이동은 2n*log(n)번 발생
    - 하나의 합병 단계에서 임시 배열에 복사했다가 다시 가져와야 하기 때문에, 이동 연산은 총 부분 배열에 들어 있는 요소의 개수가 n인 경우, 총 2n번 발생함
    - 총 $\log_2 n$개의 합병 단계가 필요하므로 총 2n*log(n)번 이동 발생
  - 레코드의 크기가 큰 경우에는 매우 큰 시간적 낭비 초래
  - 레코드를 연결 리스트로 구성하여 합병 정렬할 경우,
    - 링크 인덱스만 변경되므로 데이터의 이동은 무시할 수 있을 정도로 작아짐
    - 따라서 크기가 큰 레코드를 정렬할 경우, 다른 어떤 정렬 방법보다 매우 효율적

- 최적, 평균, 최악의 경우 큰 차이 없이 $\boxed{O(n*\log(n))}$의 복잡도
- 안정적이며 데이터의 초기 분포 정도에 영향을 덜 받음

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

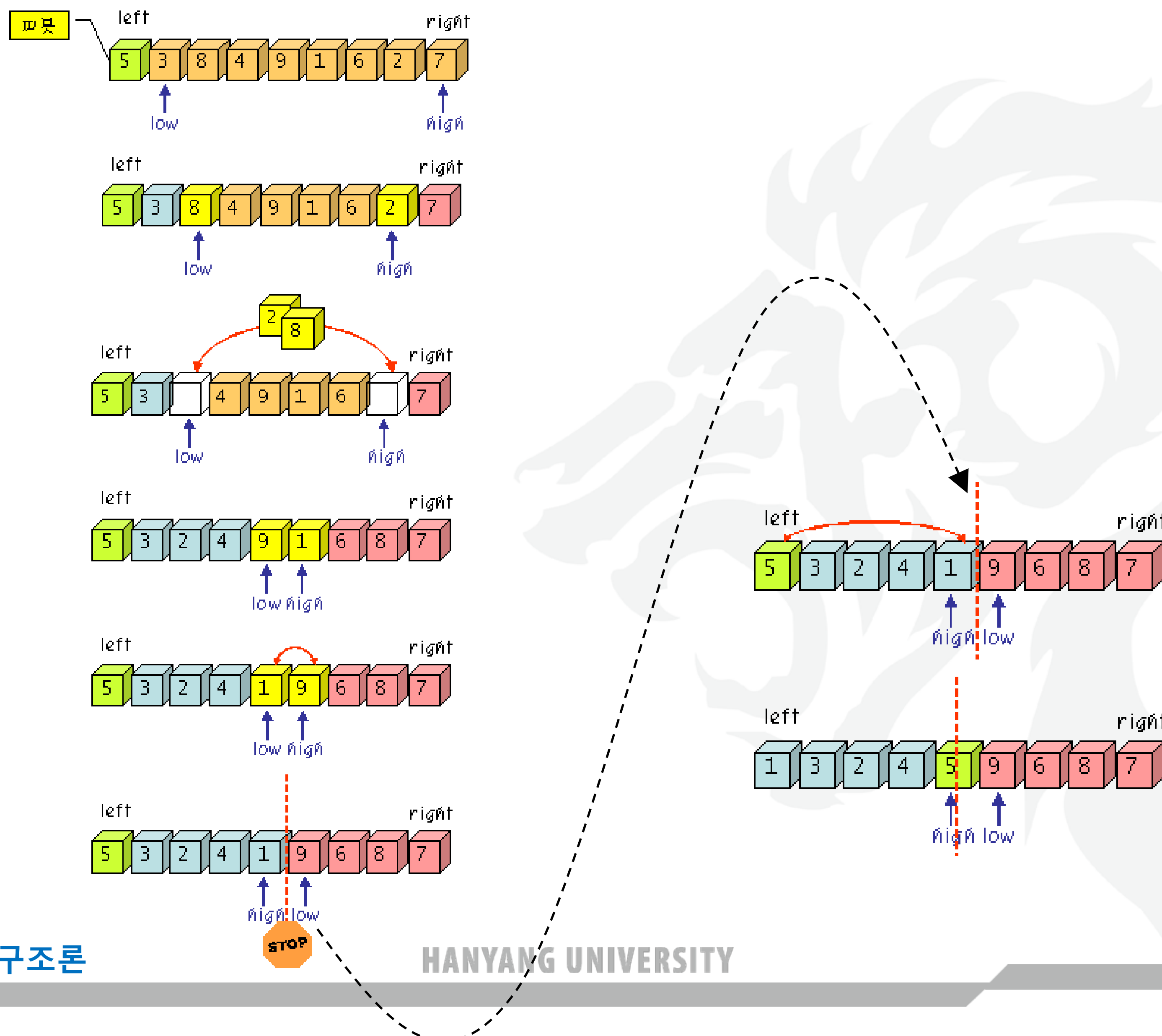
```
void quick_sort(int list[], int left, int right)
{
1.    if(left<right){
2.          int q=partition(list, left, right); //가장 중요한 함수
3.          quick_sort(list, left, q-1);
4.          quick_sort(list, q+1, right);
      }
}
```

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 까지의리스트를, 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


피봇
left
right
5 3 8 4 9 1 6 2 7
low
high
5 3 8 4 9 1 6 2 7
2
8
5 3 4 9 1 6 7
5 3 2 4 9 1 6 8 7
5 3 2 4 1 9 6 8 7
5 3 2 4 1 9 6 8 7
STOP
1 3 2 4 5 9 6 8 7

분할 함수

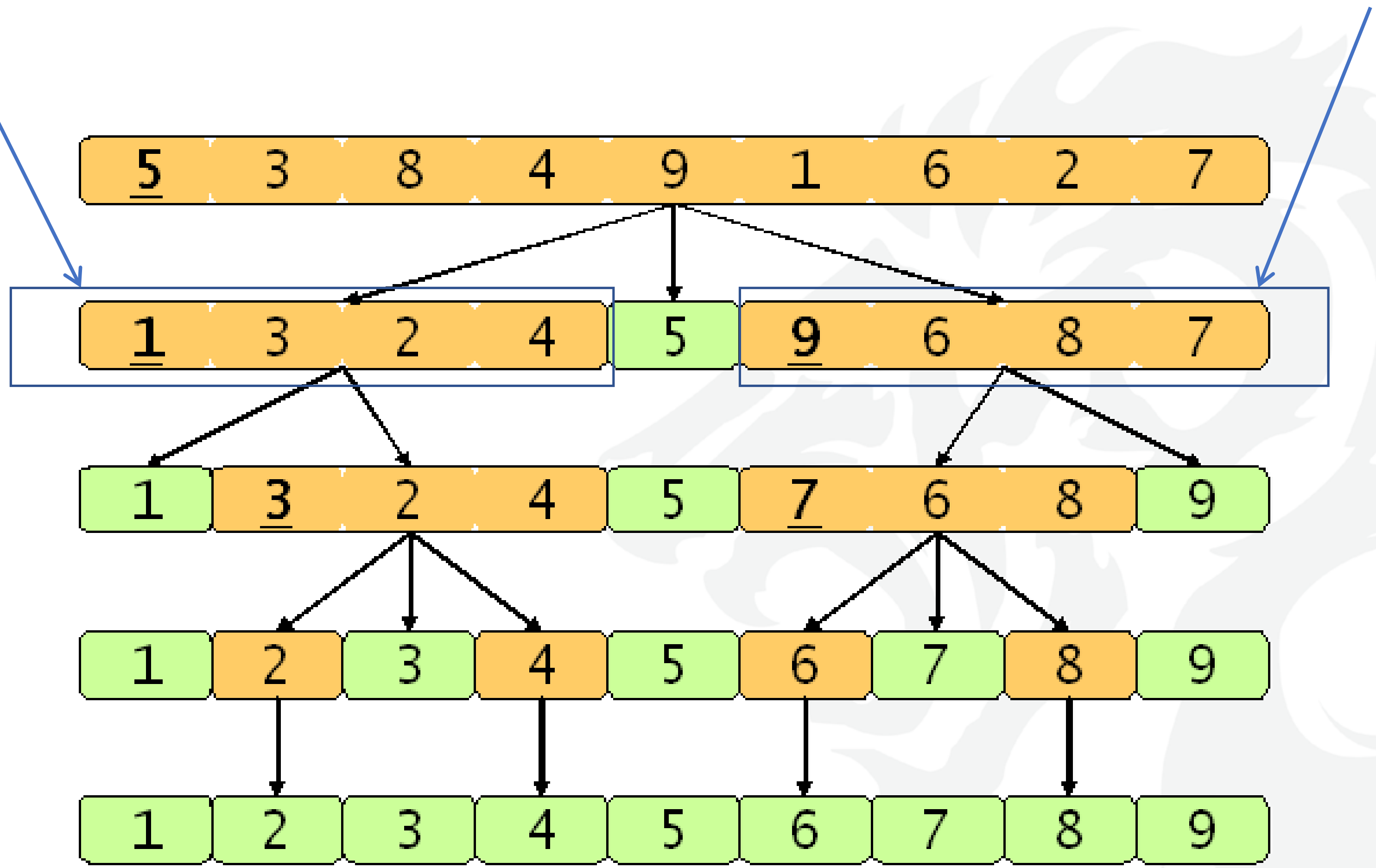
```
int partition(int list[], int left, int right)
{
      int pivot, temp;
      int low,high;

      low = left; //(do-while문에서 증가시키므로, low=left+1로 출발)
      high = right+1; //(do-while문에서 감소시키므로, high=right로 출발)
      pivot = list[left]; //정렬할 리스트의 가장 왼쪽 데이터를 피벗
      do {
            do
                  low++;
            while(low<=right &&list[low]<pivot);
            do
                  high--;
            while(high>=left && list[high]>pivot);
            if(low<high) SWAP(list[low], list[high], temp); //low와 high가 아직 교차 하지 않았으면
      } while(low<high);

      SWAP(list[left], list[high], temp);
      return high; //피벗의 위치 반환
}
```

퀵정렬 전체 과정

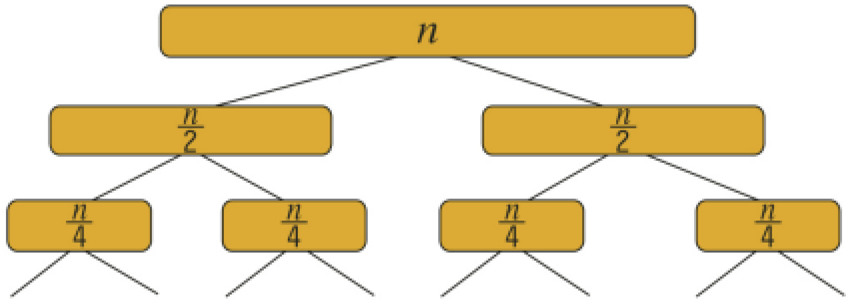
피벗을 제외하고, 왼쪽리스트(1 3 2 4), 오른쪽 리스트(9 6 8 7) 독립적으로 퀵 정렬



- 밑줄 친 숫자: 피벗

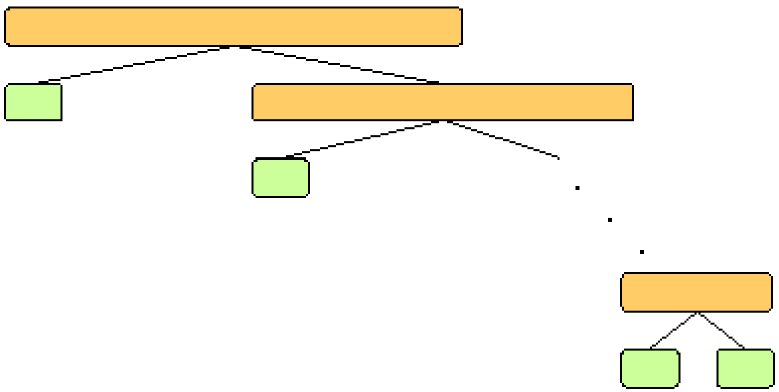
퀵정렬 복잡도 분석(1)

- 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
```

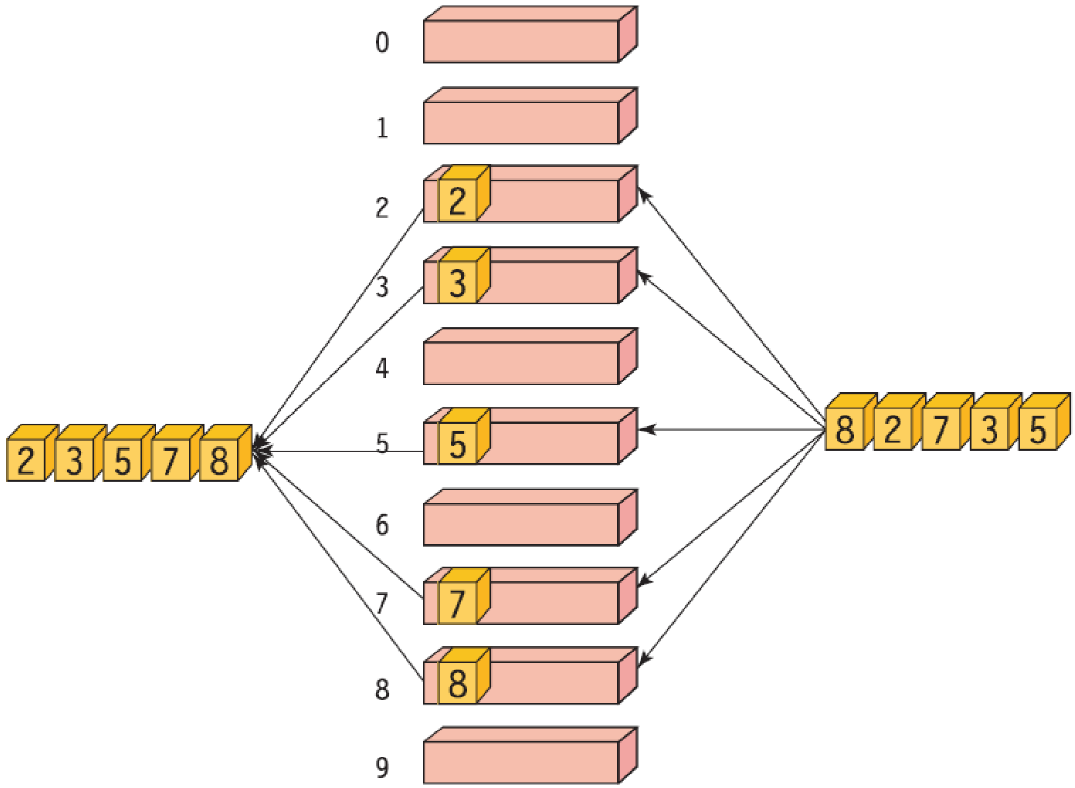
- 중간값(medium)을 피벗으로 선택하면 불균등 분할 완화 가능

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

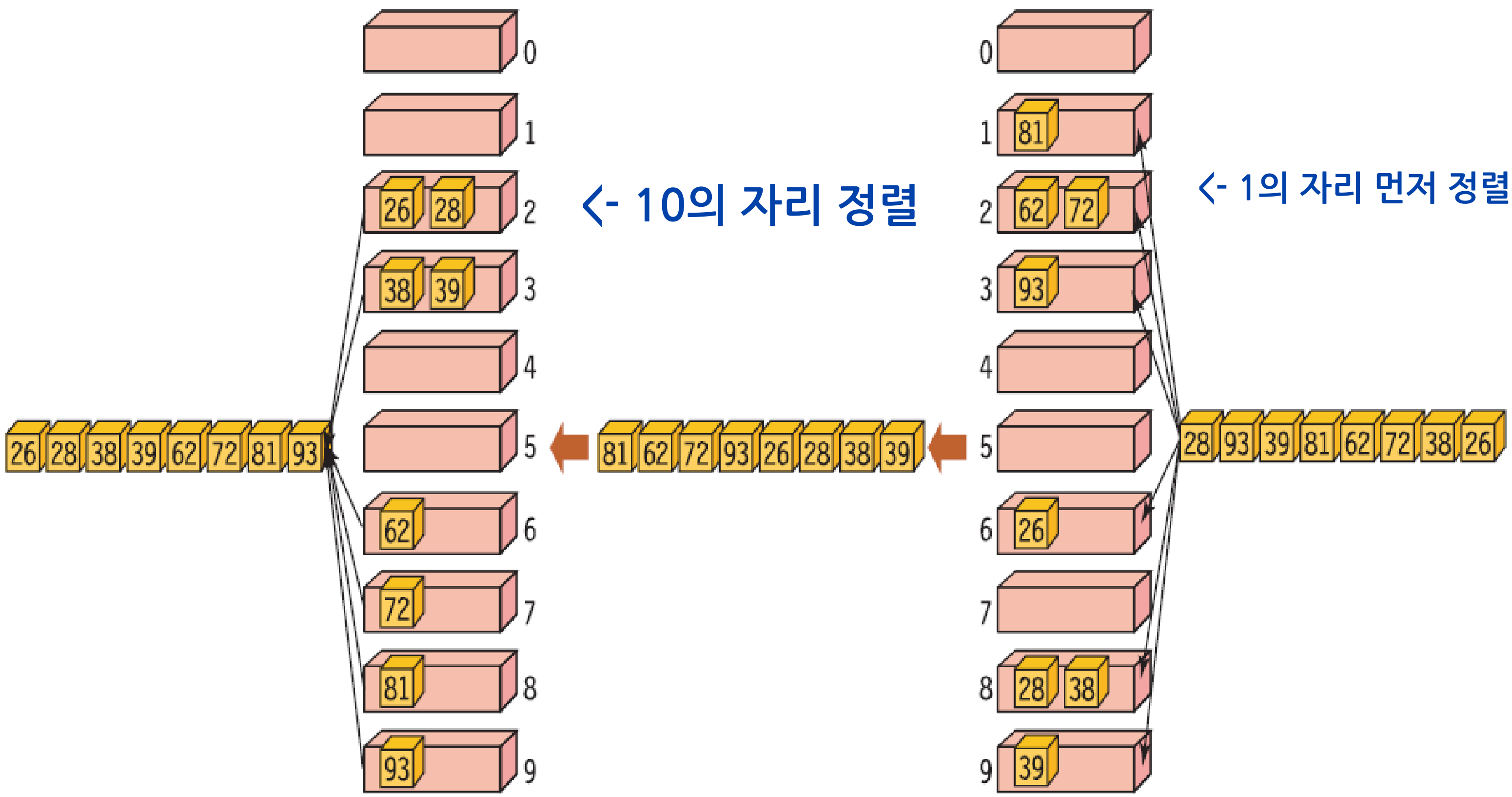
기수정렬 예(1)

- 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



10의 자리가 같은 경우, 1의 자리의 정렬이 먼저 일어나기 때문에
10의 자리로 정렬을 시작할때, 순서대로 이미 정렬이 되어 있음
<- 낮은 자리부터 분류를 하는 이유

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 코드

```
#define BUCKETS 10
#define DIGITS 4
void radix_sort(int list[], int n)
{
        int i, b, d, factor=1;
        QueueType queues[BUCKETS];

        for(b=0;b<BUCKETS;b++) init(&queues[b]);      // 큐들의 초기화

        for(d=0; d<DIGITS; d++){
                for(i=0;i<n;i++)                          // 데이터들을 자리수에 따라 큐에 입력
                    enqueue( &queues[(list[i]/factor)%10], list[i]);

                for(b=i=0;b<BUCKETS;b++)                  // 버켓에서 꺼내어 list로 합침
                    while( !is_empty(&queues[b]) )
                        list[i++] = dequeue(&queues[b]);
                factor *= 10;                          // 그 다음 자리수로 간다.
        }
}
```

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
- 실수, 한글, 한자로 이루어진 키는 정렬 못함

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