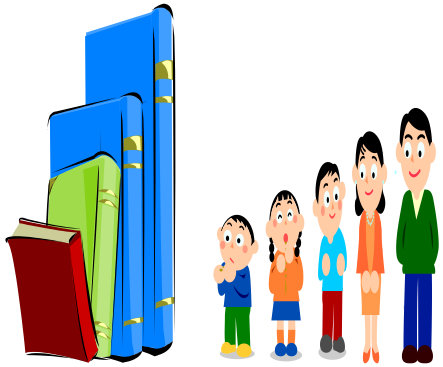
9장: 정렬

- 목차
  - 1. 정렬이란?
  - 2. 선택 정렬
  - 3. 삽입 정렬
  - 4. 버블 정렬
  - 5. 쉘 정렬
  - 6. 합병 정렬
  - 7. 퀵 정렬
  - 8. 기수 정렬
  - 9. 정렬 알고리즘의 비교

1. 정렬이란?

- 정렬은 물건을 크기순으로 오름차순이나 내림차순으로 나열하는 것



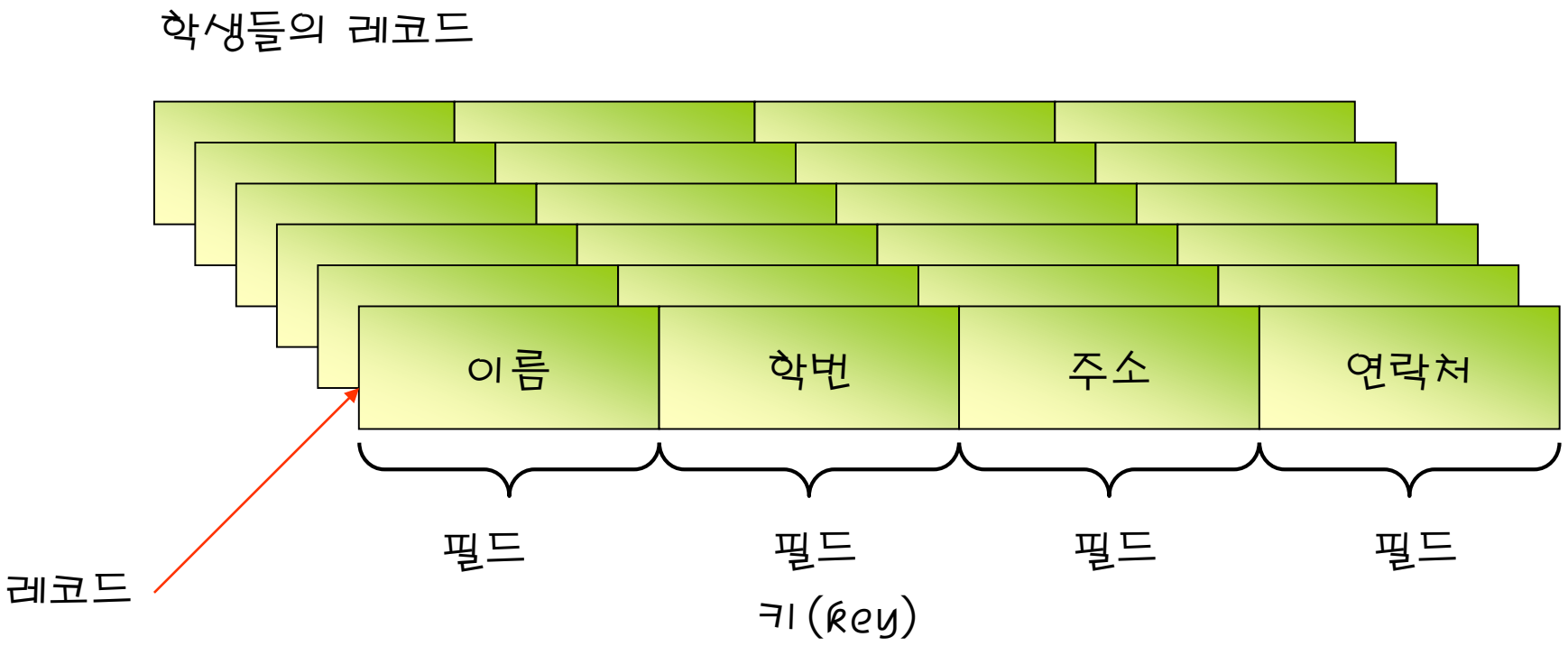
- 정렬은 컴퓨터 공학분야에서 가장 기본적이고 중요한 알고리즘중의 하나임

- 정렬은 자료 탐색에 있어서 필수적이다.

(예) 만약 사전에서 단어들이 정렬이 안되어 있다면?

비교	제조사	모델명	요약설명	최저가↓	업체수	출시
☐	ROLLEI	D-41com	410만화소(0.56")/1.8"LCD/3배줌/연사/CF카드	320,000	4	02년
☐	카시오	QV-R40	413만화소(0.56")/1.6"LCD/3배줌/동영상/히스토그램/앨범기능/SD,MMC카드	344,000	73	03년
☐	파나소닉	DMC-LC43	423만화소(0.4")/1.5"LCD/3배줌/동영상+녹음/연사/SD,MMC카드	348,000	36	03년
☐	현대	DC-4311	400만화소(0.56")/1.6"LCD/3배줌/동영상/SD,MMC카드	350,000	7	03년
☐	삼성테크윈	Digimax420	410만화소(0.56")/1.5"LCD/3배줌/동영상+녹음/음성메모/한글/SD카드	353,000	47	03년
☐	니콘	Coolpix4300	413만화소(0.56")/1.5"LCD/3배줌/동영상/연사/CF카드HOT4	356,800	79	02년
☐	올림푸스	뮤-20 Digital	423만화소(0.4")/1.5"LCD/3배줌/동영상/연사/생활방수/xD카드	359,000	63	03년
☐	코닥	LS-443(Dock포함)	420만화소/1.8"LCD/3배줌/동영상+녹음/SD,MMC카드/Dock시스템	365,000	39	02년
☐	올림푸스	C-450Z	423만화소(0.4")/1.8"LCD/3배줌/동영상/연사/xD카드	366,000	98	03년
☐	올림푸스	X-1	430만화소/1.5"LCD/3배줌/동영상/연사/xD카드	367,000	19	03년
☐	미놀타	DiMAGE-F100	413만화소(0.56")/1.5"LCD/3배줌/동영상+녹음/음성메모/동체추적AF/연사/SD,MMC카드	373,000	18	02년
☐	삼성테크윈	Digimax410	410만화소(0.56")/1.6"LCD/3배줌/동영상+녹음/음성메모/한글/CF카드	374,000	4	02년

정렬의 단위



정렬 알고리즘의 개요

- 많은 정렬 알고리즘 존재
  - 단순하지만 비효율적인 방법
    - 삽입정렬, 선택정렬, 버블정렬 등
  - 복잡하지만 효율적인 방법
    - 퀵정렬, 히프정렬, 합병정렬, 기수정렬 등

- 모든 경우에 최적인 알고리즘은 없음
- 응용에 맞추어 선택
- 정렬 알고리즘의 평가
  - 비교횟수
  - 이동횟수

2. 선택정렬(selection sort)

- 선택정렬(selection sort): 정렬이 안된 숫자들중에서 최소값을 선택하여 배열의 첫번째 요소와 교환
- 왼쪽 리스트에는 정렬이 완료된 숫자, 오른쪽에는 정렬되지 않은 숫자들이 있음
- 선택정렬은 오른쪽 리스트에서 제일 작은 숫자를 선택하여 왼쪽 리스트로 이동

왼쪽 리스트	오른쪽 리스트	설 명
	5, 3, 8, 1, 2, 7	초기 상태
1	5, 3, 8, 2, 7	1 선택
1, 2	5, 3, 8, 7	2 선택
1, 2, 3	5, 8, 7	3 선택
1, 2, 3, 5	8, 7	5 선택
1, 2, 3, 5, 7	8	7 선택
1, 2, 3, 5, 7, 8		8 선택

선택 정렬 유사 코드

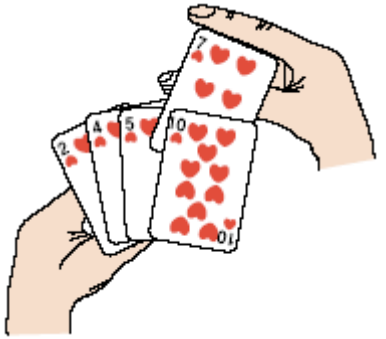
```
selection_sort(A, n)

for i←0 to n-2 do
          least ← A[i], A[i+1],..., A[n-1] 중에서 가장 작은 값의 인덱스;
          A[i]와 A[least]의 교환;
          i++;
```

선택 정렬 분석

- 비교 횟수: 두개의 for 루프(내/외부 루프)를 사용
- 비교 시간 복잡도: $O(n^2)$
- 교환 횟수: 외부 루프의 실행횟수 * 이동 횟수 = 3(n-1)
- 선택 정렬의 장점
  - 자료의 이동 횟수가 미리 결정
- 선택 정렬의 단점
  - 이동 횟수가 상당히 크다
  - 자료가 정렬된 경우 불필요한 이동 발생 (if문 추가로 해결 가능)
  - https://visualgo.net/en/sorting

3. 삽입정렬(insertion sort)

- 삽입정렬은 정렬되어 있는 부분에 새로운 레코드를 적절한 위치에 삽입하는 과정을 반복



- 삽입정렬의 과정

삽입정렬 유사코드

```
Algorithm InsertionSort(list, n):
        Input: n개의 정수를 저장하고 있는 배열 list
        Output: 정렬된 배열 list
        for i←1 to n-1 do
        {
              정렬된 리스트에서 list[i]보다 더 큰요소들이동;
               list[i]를 정렬된 리스트의 적절한 위치에 삽입;
               i++;
        }
```

삽입정렬 프로그램

```
// 삽입정렬
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;
   }
}
```

- 많은 이동-> 레코드가 클 경우 불리
- 안정된 정렬방법
- 이미 정렬되어 있으면 효율적

삽입정렬 복잡도 분석

- 최상의 경우: 이미 정렬되어 있는 경우
  - 비교: n-1번
  - 시간복잡도: O(n)
- 최악의 경우: 역순으로 정렬
  - 모든 단계에서 앞에 놓인 자료 전부 이동
  - 비교: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - 이동: $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$
- 삽입정렬은 비교적 많은 데이터의 이동을 포함하나 대부분의 데이터가 정렬되어 있는 경우 매우 효율적

4. 버블정렬(bubble sort)

- 인접한 레코드가 순서대로 되어 있지 않으면 교환
- 전체가 정렬될때까지 비교/교환계속

버블정렬 유사코드

```
BubbleSort(A, n)

for i←n-1 to 1 do
   for j←0 to i-1 do
          j와 j+1번째의 요소가 크기순이 아니면 교환
          j++;
   i--;
```

버블정렬 C코드

```
#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 분석

- 비교횟수
  - 버블정렬의 비교횟수는 최상, 평균, 최악의 어떠한 경우에도 항상 일정

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

- 이동횟수
  - 최악: 역순정렬 이동 = 3*비교횟수
  - 최상: 이미정렬 이동 = 0
  - 평균: $O(n^2)$
  - 애플릿 확인

5. 쉘정렬(Shell sort)

- 삽입정렬이 어느 정도 정렬된 배열에 대해서는 대단히 빠른 것에 착안한 방법
- 쉘정렬은 삽입 정렬보다 빠르다.
- 삽입 정렬의 최대 문제점은 요소들이 이동할 때, 이웃한 위치로만 이동
- 쉘정렬에서는 요소들이 멀리 떨어진 위치로도 이동할 수 있다.
- 전체 리스트를 일정 간격의 부분 리스트로 나누고 부분 리스트를 정렬: 간격은 n/2로 나누고 짝수이면 1을 더함
- 간격(gap)은 점점 줄어들게 된다.

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

쉘정렬 분석

- 장점
  - 삽입 정렬에 비해서 자료 교환을 위해 비교적 큰 거리를 이동할 수 있다
  - 부분 파일이 어느 정도 정렬 된 상태이기 때문에 기본적으로 삽입 정렬의 장점(거의 정렬된 데이터의 정렬이 빠르게 되는점)을 극대화 시킬 수 있다.

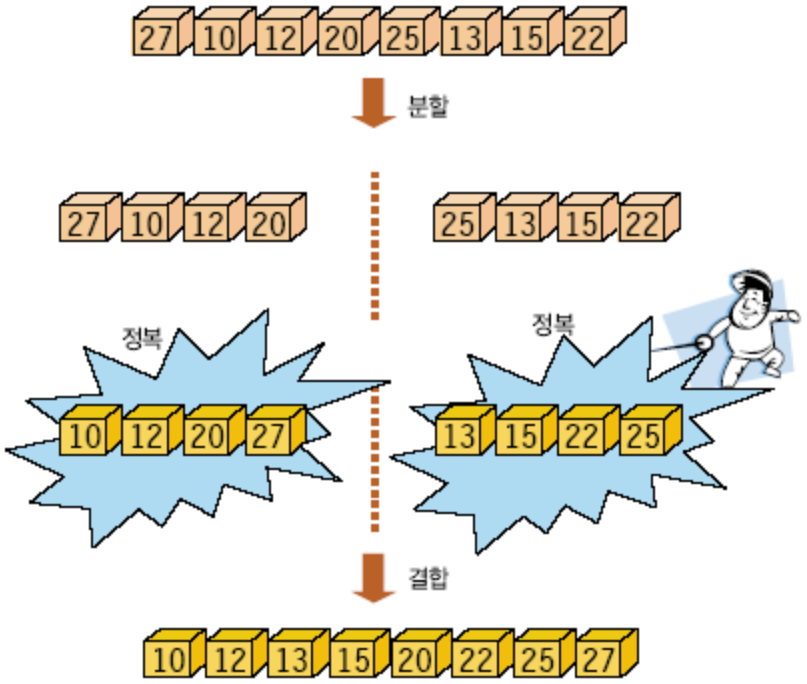
- 평균적인 경우 시간복잡도는 $O(n^{1.5})$

쉘정렬 프로그램

```
// 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 ) {
          if( (gap%2) == 0 ) gap++;
          for(i=0;i<gap;i++)            // 부분 리스트의 개수는 gap
                    inc_insertion_sort(list, i, n-1, gap);
    }
}
```

6. 합병정렬

- 합병정렬은 리스트를 두개로 나누어, 각각을 정렬한 다음, 다시 하나로 합치는 방법
- 합병정렬은 분할정복기법에 바탕



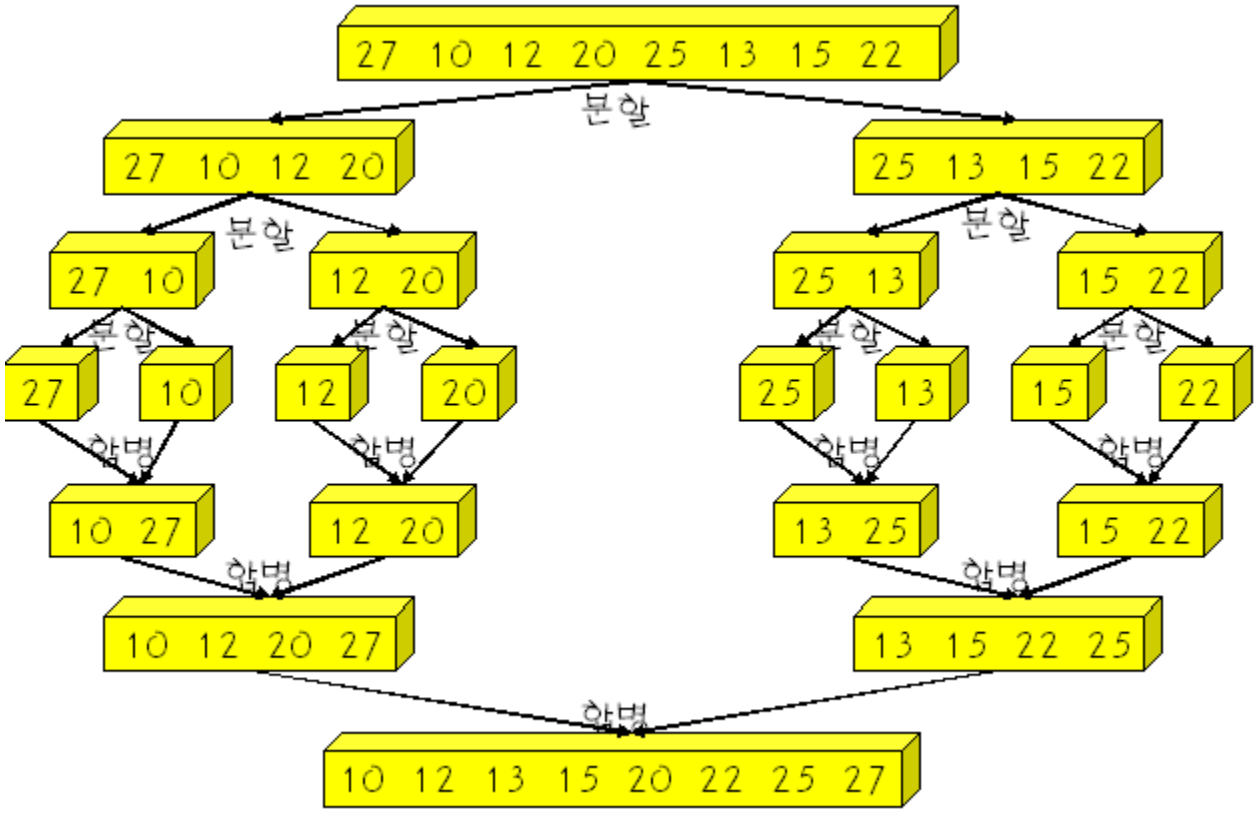
분할정복법

- 분할정복법(divide and conquer)은 문제를 작은 2개의 문제로 분리하고 각각을 해결한 다음, 결과를 모아서 원래의 문제를 해결하는 전략.
- 분리된 문제가 아직도 해결하기 어렵다면, 즉 충분히 작지 않다면 분할정복방법을 다시 적용한다. 이는 재귀호출을 이용하여 구현.

```
1. 분할(Divide): 배열을 같은 크기의 2개의 부분 배열로 분할한다.
2. 정복(Conquer): 부분배열을 정렬한다. 부분배열의 크기가 충분히 작지 않으면 재
   귀호출을 이용하여 다시 분할정복기법을 적용한다.
3. 결합(Combine): 정렬된 부분배열을 하나의 배열에 통합한다.
```

```
입력파일: 27 10 12 20 25 13 15 22

1.분할(Divide): 배열을 27 10 12 20 과 25 13 15 22의 2개의 부분배열로 분리
2.정복(Conquer): 부분배열을 정렬하여 10 12 20 27 과 13 15 22 25를 얻는다.
3.결합(Combine): 부분배열을 통합하여 10 12 13 15 20 22 25 27을 얻는다.
```

합병정렬의 전체과정


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

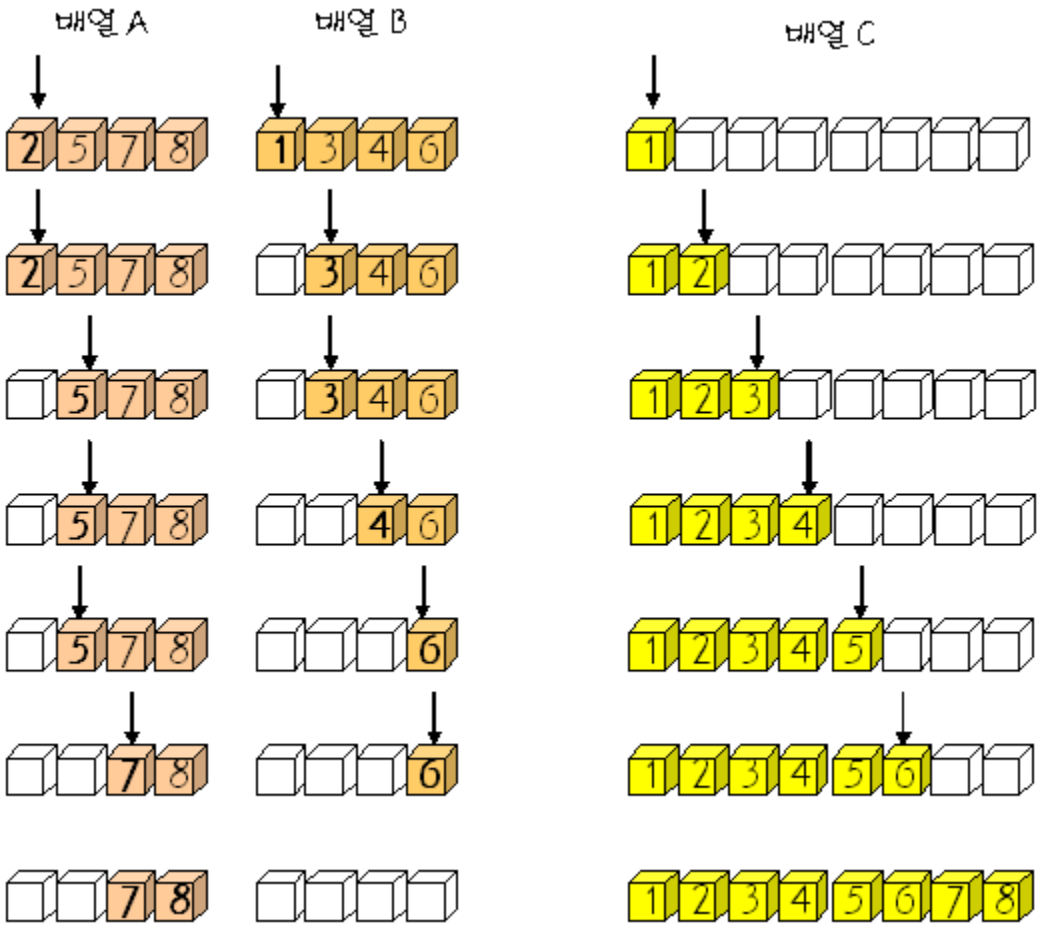
합병정렬의 유사코드

```
merge_sort(list, left, right)

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

- 합병정렬은 하나의 리스트를 두 개의 균등한 크기로 분할하고 분할된 부분 리스트를 정렬한 다음, 두 개의 정렬된 부분 리스트를 합하여 전체가 정렬된 리스트를 얻고자 하는 것

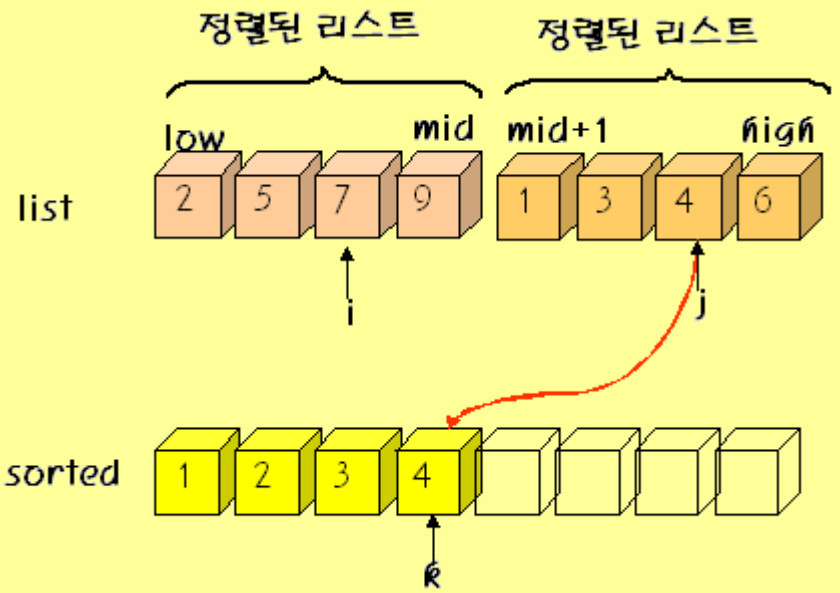
- 합병정렬에서 실제로 정렬이 이루어지는 시점은 2개의 리스트를 합병하는 단계이다.

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, last):
// 2개의 인접한 배열 list[left..mid]와 list[mid+1..right]를 합병

b1←left;
e1←mid;
b2←mid+1;
e2←right;
sorted 배열을 생성;
index←0;
while b1≤e1 and b2≤e2 do
    if(list[b1]<list[b2])
          then
            sorted[index]←list[b1];
            b1++;
            index++;
          else
            sorted[index]←list[b2];
            b2++;
            index++;
요소가 남아있는 부분배열을 sorted로 복사한다;
sorted를 list로 복사한다;
```



합병정렬의 분석

- 비교횟수
  - 합병정렬은 크기 n인 리스트를 정확히 균등 분배하므로 퀵정렬의 이상적인 경우와 마찬가지로 정확히 $\log_2 n$개의 패스를 가진다. 각 패스에서 리스트의 모든 레코드 n개를 비교하여 합병하므로 n 번의 비교 연산이 수행된다.
  - 따라서 합병정렬은 최적, 평균, 최악의 경우 모두 큰 차이 없이 $n \log_2 n$ 번의 비교를 수행하므로 $O(n \log_2 n)$의 복잡도를 가지는 알고리즘이다. 합병정렬은 안정적이며 데이터의 초기 분산 순서에 영향을 덜 받는다.
- 이동횟수
  - 배열을 이용하는 합병정렬은 레코드의 이동이 각 패스에서 2n번 발생하므로 전체 레코드의 이동은 $2n \log_2 n$ 번 발생한다. 이는 레코드의 크기가 큰 경우에는 매우 큰 시간적 낭비를 초래한다.
  - 그러나 레코드를 연결 리스트로 구성하여 합병 정렬할 경우, 링크 인덱스만 변경되므로 데이터의 이동은 무시할 수 있을 정도로 작아진다.
  - 따라서 크기가 큰 레코드를 정렬할 경우, 연결 리스트를 이용하는 합병정렬은 퀵정렬을 포함한 다른 어떤 정렬 방법보다 매우 효율적이다.

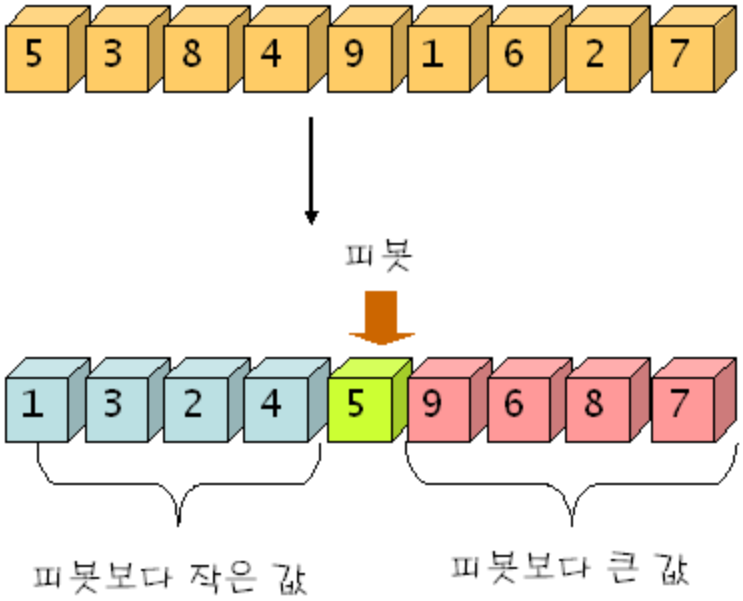
합병정렬의 분석

- 장점
  - 안정적인 정렬방법으로 데이터의 분포에 영향을 덜 받는다. (즉 입력 데이터가 무엇이던지 간에 정렬되는 시간은 동일)
  - 만약 링크드 리스트를 이용하면 합병정렬은 퀵정렬을 포함한 다른 어떤 정렬방법보다도 효율적 일수 있음

- 단점
  - 임시 배열이 필요함
  - 만약 레코드의 크기가 큰 경우에는 이동 횟수가 많으므로 매우 큰 시간적 낭비를 초래(단 링크드 리스트로 구성할 경우 링크 인덱스만 변경하면 되므로 데이터 이동이 무시할 정도로 작아짐)

7. 퀵 정렬(quick sort)

- 평균적으로 가장 빠른 정렬 방법
- 분할정복법 사용
- 퀵정렬은 전체 리스트를 2개의 부분리스트로 분할하고, 각각의 부분리스트를 다시 퀵정렬로 정렬



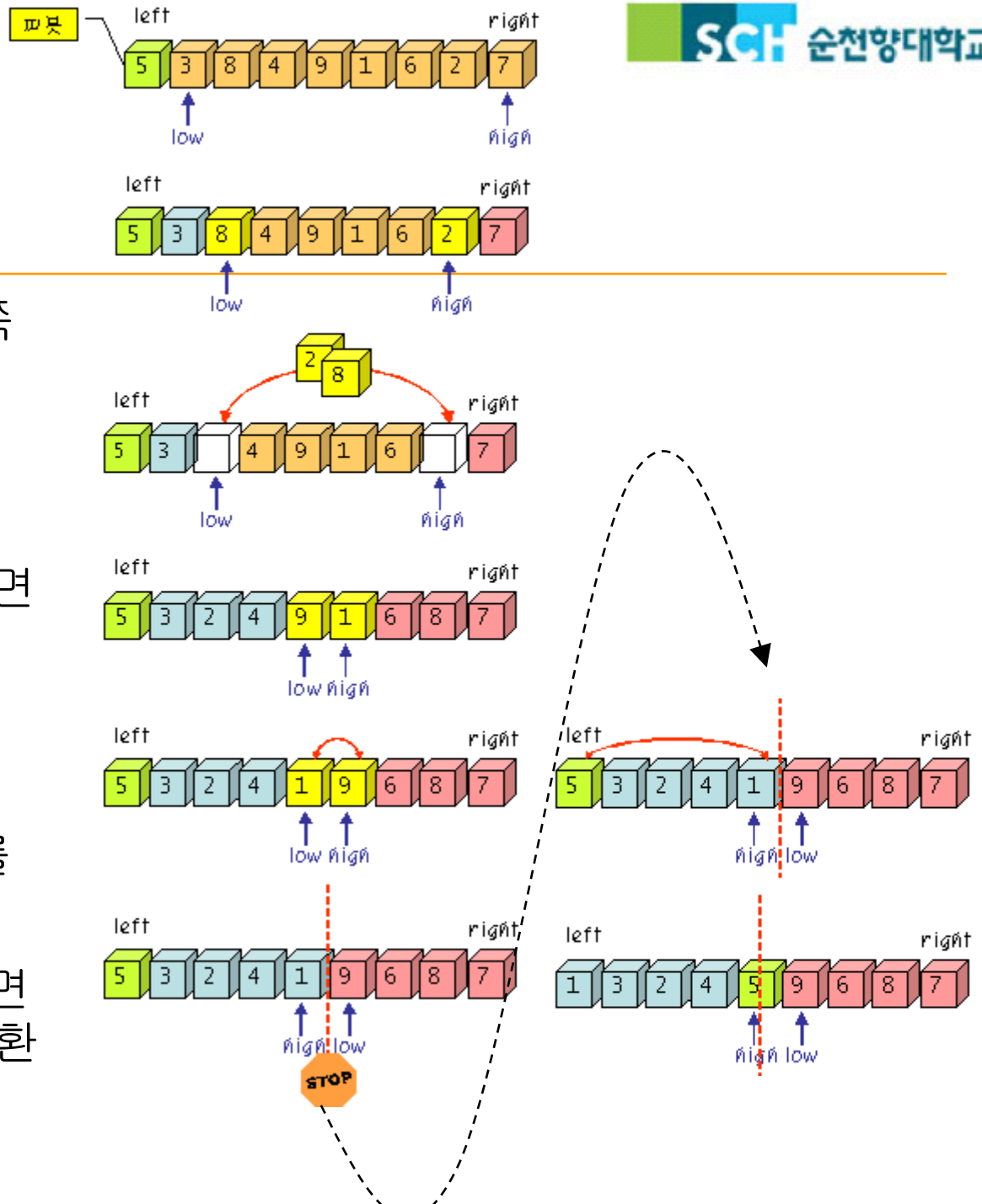
퀵 정렬 알고리즘

```
1.   void quick_sort(int list[], int left, int right)
2.   {
3.     if(left<right){
4.        int q=partition(list, left, right);
5.        quick_sort(list, left, q-1);
6.        quick_sort(list, q+1, right);
7.      }
8.   }
```

- 3 정렬할 범위가 2개 이상의 데이터이면
- 4 partition 함수를 호출하여 피벗을 기준으로 2개의 리스트로 분할한다. partition 함수의 반환값은 피봇의 위치가 된다.
- 5 left에서 피봇 위치 바로 앞까지를 대상으로 순환 호출한다(피봇은 제외된다).
- 6 피봇 위치 바로 다음부터 right까지를 대상으로 순환 호출한다(피봇은 제외된다).

partition 함수

- 피봇(pivot): 가장 왼쪽 숫자라고 가정
- 두개의 변수 low와 high를 사용한다.
- low는 피봇보다 작으면 통과, 크면 정지
- high는 피봇보다 크면 통과, 작으면 정지
- 정지된 위치의 숫자를 교환
- low와 high가 교차하면 종료: high와 pivot 교환

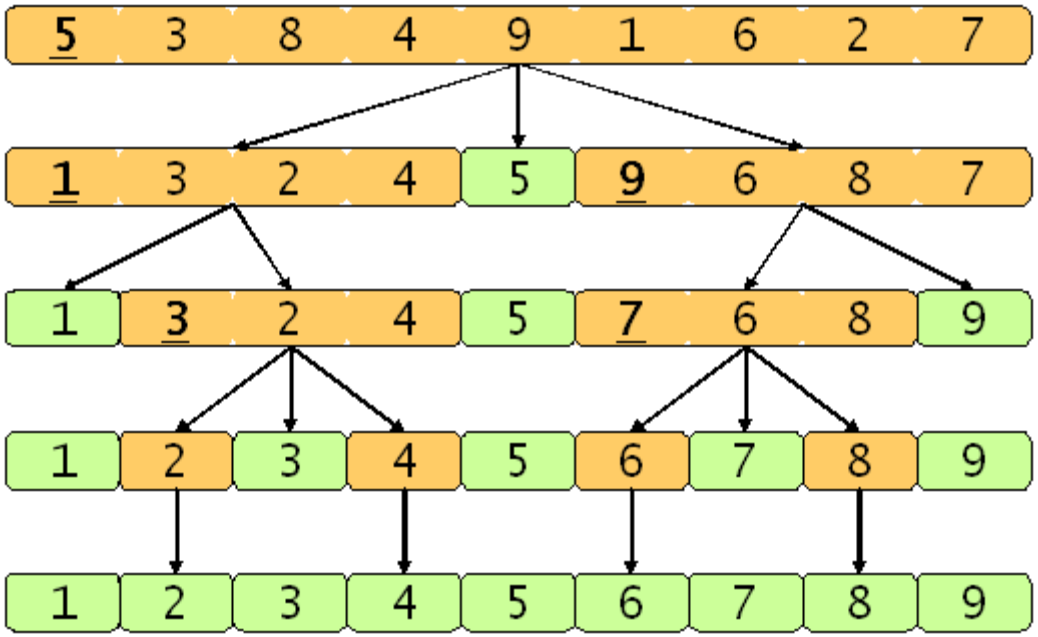


partition 함수 구현

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

   low = left;
   high = right+1;
   pivot = list[left];
   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);
   } while(low<high);

   SWAP(list[left], list[high], temp);
   return high;
}
```

- low는 left+1에서 출발
- high는 right에서 출발
- 정렬할 리스트의 가장 왼쪽 데이터를 피봇으로 선택
- low와 high가 교차할때 까지 진행
- list[low]가 pivot보다 작으면 계속 low 증가
- list[high]가 pivot보다 크면 계속 high 증가
- low와 high가 교차하지 않았으면 list[low]와 list[high]를 교환
- low와 high가 교차하면 반복 종료
- 피봇을 중앙에 위치
- 피봇의 위치 반환

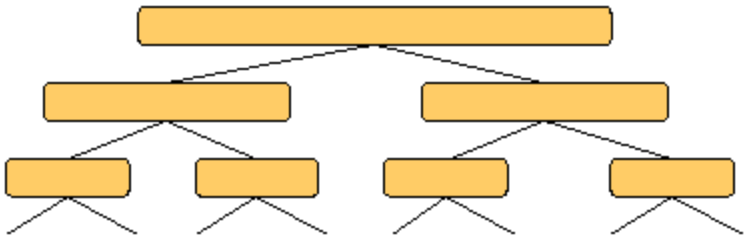
퀵 정렬 전체과정



- 밑줄친 숫자: 피봇

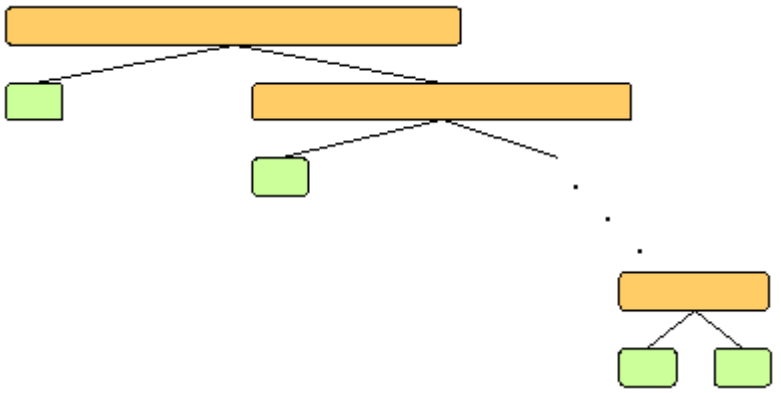
퀵 정렬의 분석

- 최상의 경우:
  - 거의 균등한 리스트로 분할되는 경우
  - 패스 수: $\log_2 n$
    - 2->1
    - 4->2
    - 8->3
    - ...
    - n->$\log_2 n$
  - 각 패스안에서의 비교횟수: n
  - 총비교횟수: n $\log_2 n$
  - 총이동횟수: 비교횟수에 비하여 무시가능
  - O(n $\log_2 n$)



퀵 정렬의 분석(cont.)

- 최악의 경우:
  - 불균등한 리스트로 분할되는 경우
  - 패스 수: n
  - 각 패스안에서의 비교횟수: n
  - 총비교횟수: n^2
  - 총이동횟수: 비교횟수에 비하여 무시가능
  - (예) 정렬된 리스트가 주어졌을 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

퀵 정렬의 분석(cont.)

- 퀵 정렬의 평균 시간 복잡도: $O(n \log_2 n)$
- 특히 $O(n \log_2 n)$의 다른 정렬 알고리즘보다 빠름
  - 불필요한 데이터의 이동을 줄임
  - 먼 거리의 데이터 교환 가능
  - 한번 결정된 피벗들은 추후 연산에서 제외
- 퀵 정렬의 장점: 속도가 빠르고 추가 메모리를 요구하지 않음
- 퀵 정렬의 단점: 이미 정렬된 리스트에 대해서는 수행시간이 오래 걸림
  - 단점을 해결하기 위해 리스트의 왼쪽 데이터를 선택 하는 대신 몇 개의 데이터 중에서 중간값을 피벗으로 선택

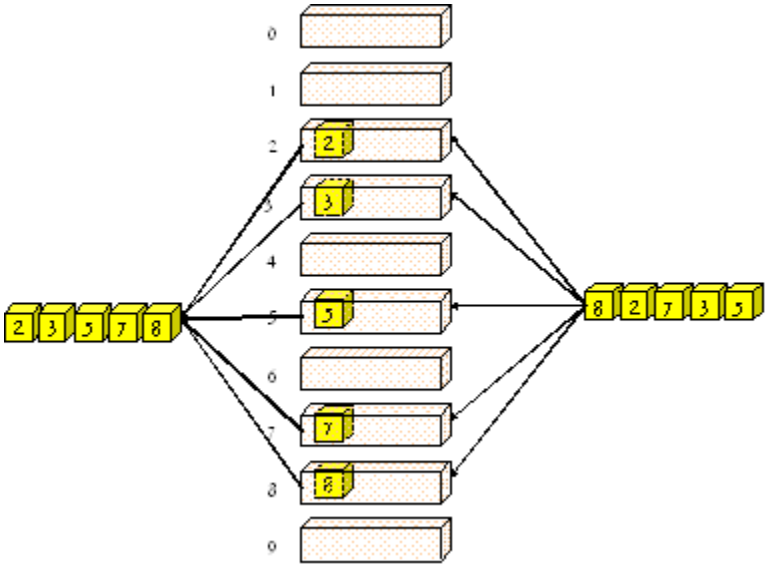
8. 기수정렬(Radix Sort)

- 이때까지의 정렬 방법들은 모두 레코드들을 비교하여 정렬
- 기수 정렬은 레코드를 비교하지 않고도 정렬하는 방법
- 기수 정렬(radix sort)은 $O(n\log_2 n)$ 이라는 이론적인 하한선을 깰 수 있는 유일한 방법이다.
- 기수 정렬은 O(kn) 의 시간 복잡도를 가지는데 대부분 k<4 이하
- 기수 정렬의 단점은 정렬할 수 있는 레코드의 타입이 한정: 레코드의 키들이 동일한 길이를 가지는 숫자나 문자열로 구성

- (예) 한자리수의 기수정렬

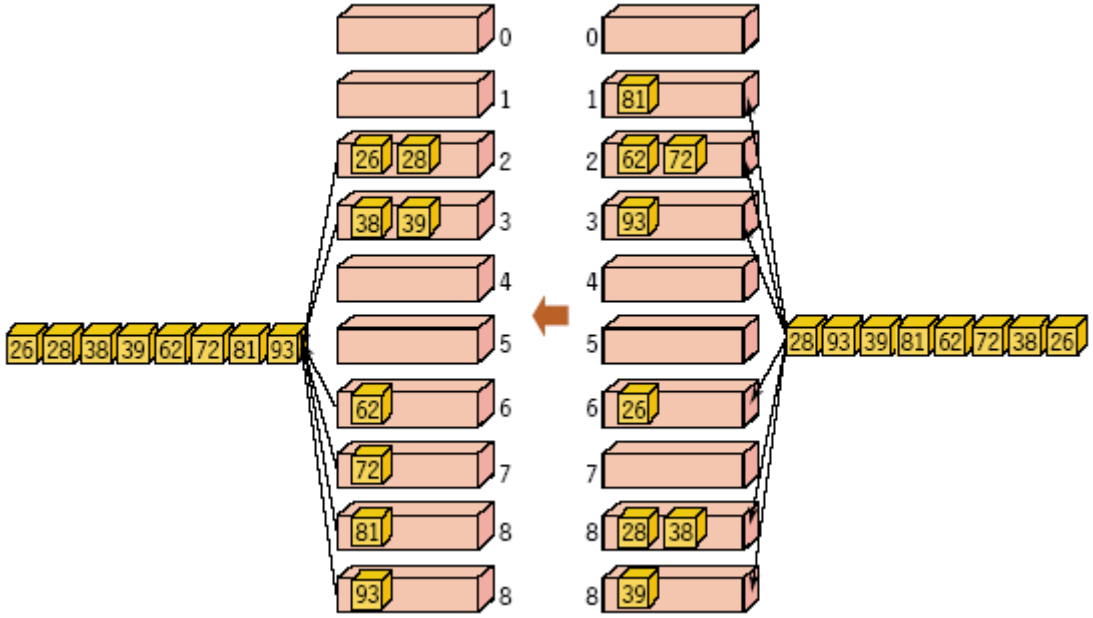
  (8, 2, 7, 3, 5)

- 단순히 자리수에 따라 버킷에 넣었다가 꺼내면 정렬됨



기수정렬

- 만약 2자리수이면? (28, 93, 39, 81, 62, 72, 38, 26)
- 낮은 자리수로 먼저 분류한다음, 순서대로 읽어서 다시 높은 자리수로 분류하면 된다.



기수정렬 알고리즘

```
RadixSort(list, n):


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

- 버킷은 큐로 구현
- 버킷의 개수는 키의 표현 방법과 밀접한 관계
  - 이진법을 사용한다면 버킷은 2개.
  - 알파벳 문자를 사용한다면 버킷은 26개
  - 십진법을 사용한다면 버킷은 10개
  - 32비트의 정수의 경우, 8비트씩 나누어 기수정렬의 개념을 적용한다면 ->필요한 상자의 수는 256개가 된다. 대신에 필요한 패스의 수는 4개로 십진수 표현보다 줄어든다.

기수 정렬의 분석

- 32비트의 컴퓨터의 경우 대개 10개정도의 자리수 안에서 표현되므로 일반적으로 k는 n에 비하여 아주 작은 수가 되므로 O(n)이라고 하여도 무리가 없다
- 기수 정렬의 장점: 다른 정렬 방법에 비해 비교적 빠른 수행 시간안에 정렬 가능
- 기수 정렬의 단점: 기수 정렬은 사용되는 키값이 숫자로 표현 되어야만 적용 가능 (예를 들면 실수나 한글 한자등으로 이루어진 키 값은 많은 버킷이 필요하게 되므로 기수 정렬이 불가능)

9. 정렬 알고리즘의 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	$O(dn)$	$O(dn)$	$O(dn)$