CHAP 9. 정렬(sort)

"*The Art of Computer Programming*" Donald Knuth

- Knuth : 컴퓨터 프로그래밍의 철학과 기술을 예술적 수준으로 승화시킨, 알고리즘을 집대성한 대가

"정렬은 전통적으로 비즈니스 데이터 처리과정에서 사용되었지만, 모든 프로그래머들이 다양한 상황에서 사용하기 위해서 반드시 기억해야 하는 기초적인 도구이다."

정렬이란?

- 정렬은 물건을 크기 순으로 나열하는 것
  : 오름차순 또는 내림차순

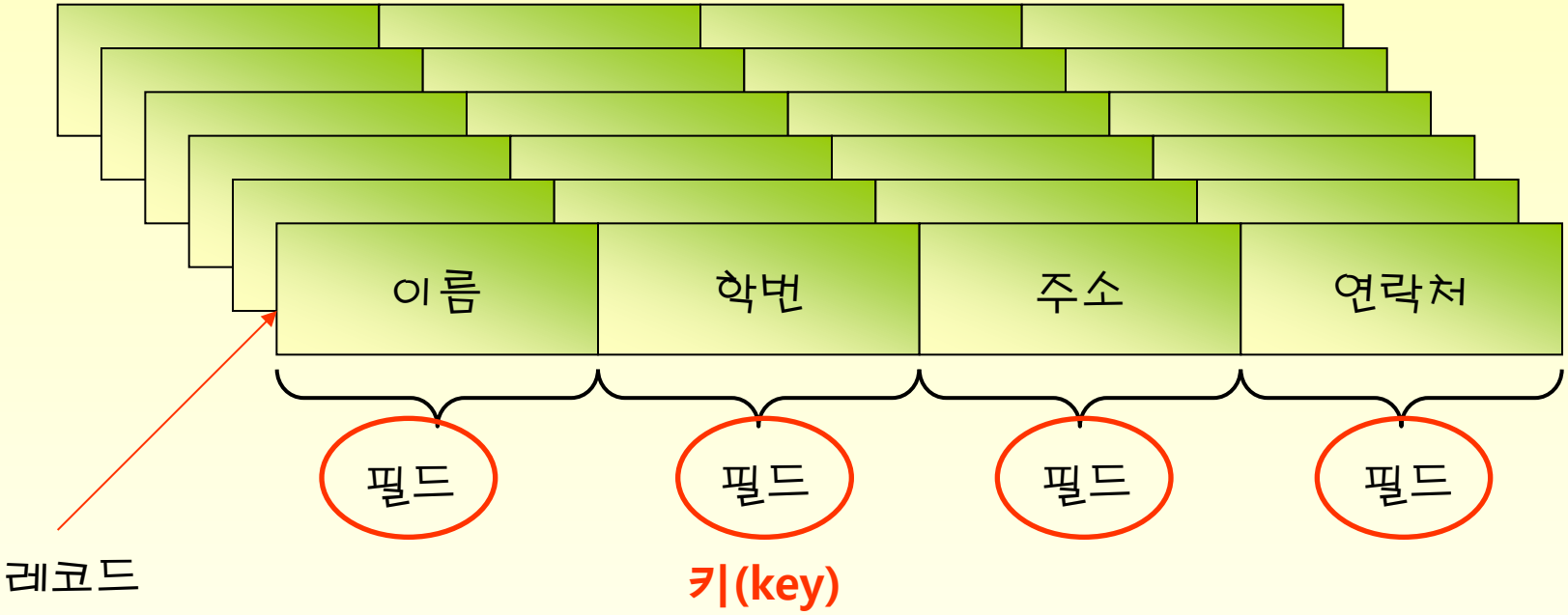
- 정렬은 컴퓨터 분야에서 가장 많이 사용, 기본적이고 중요한 알고리즘 중의 하나

- 정렬은 자료 탐색에 있어서 필수기능
  (예) 만약 사전에서 단어들이 정렬이 되어 있지 않다면?
  (예) 포털사이트에서 검색한 웹문서들이 중요도 또는 검색빈도 또는 신뢰도로 정렬되어 제시

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

<예> 학생들의 레코드, 데이터 크기, 입력 시간 등을 기준함



정렬 알고리즘의 개요

- 정렬의 기본 개념
  - 임의로 입력된 데이터들은 원하는 크기 순서이 아님 ➔ 역 순서쌍이 존재
  - 정렬 : 역 순서쌍들의 존재를 검색하여 정 순서쌍으로 재배치하는 동작

    5,4,3,2,1 입력에는 몇 개의 역 순서쌍이 존재할까?

    ➔ 몇 번의 역 순서쌍 수정이 필요할까?

    ➔ 이런 입력에는 어떤 방법이 가장 효과적인 정렬을 제공할까?

- 주로 사용하는 주요 정렬 알고리즘
  - 삽입정렬(향상→쉘 정렬), 선택정렬(향상→세이크 정렬), 버블정렬, 퀵정렬, 히프정렬, 합병정렬, 기수정렬 등

정렬 알고리즘의 개요

- 정렬 알고리즘 선택방법
  - 모든 경우(언제나)의 입력에 항상 최적의 성능을 보여주는 알고리즘은 없음
  - 그러므로 주어진 문제에 대한 분석을 통하여 그 상황에 맞는 방법을 선택해야 함
  - 입력 예 : 1 2 3 4 5
    5 4 3 2 1
    3 4 5 1 2

- 정렬 알고리즘의 성능평가 기준 : 시간, 공간 성능

  1) 시간 성능
  - 비교 : 크기 비교(if문) 횟수
  - 이동 : 자리 바꿈(배정문) 횟수

  2) 공간 성능
  - 입력자료 저장공간 n + 추가공간 α

응용분야의 경우에 적합한 정렬방법을 선택해서 사용해야 함

- **레코드 수의 많고 적음에 따른 선택** : 정렬 속도가 빠른 정렬을 선택
- **레코드 크기의 크고 작음에 따른 선택** : 레코드 위치 교환이 적은 정렬을 선택
- **Key의 특성(문자, 정수, 실수 등)에 따른 선택** : 큰 차이 없음
- **메모리 내부정렬 또는 외부정렬 여부에 따른 선택** : 상황에 따라 선택

정렬 알고리즘들의 평가 기준

- **비교 횟수의 많고 적음을 기준** : 훨씬 더 중요한 기준
- **이동 횟수의 많고 적음을 기준**

정렬 알고리즘 분류

- **삽입법** : 삽입정렬, 쉘정렬 등
- **교환법** : 버블정렬, 퀵정렬 등
- **선택법** : 선택정렬, 쉐이크정렬 등
- **분할법** : 합병정렬 등

(1) 선택정렬(selection sort)

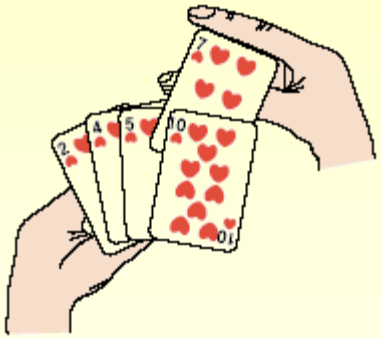
- **selection sort :**
  - 입력 데이터들을 배열에 저장
  - 그 중에서 최소값을 선택하여 첫 번째 요소와 위치를 교환(역순서쌍 제거)
    또는 최대값을 선택하여 맨 끝의 요소와 위치를 교환(역순서쌍 제거)
  - 방금 배치한 데이터(첫번째 최소값)를 제외하고 그 이후부터 반복

```
selection_sort(A, n)
{
    for  i←0  to  n-1  do
        smallest ← A[i] ~ A[n-1] 중에서 가장 작은 값;
        A[i]와 smallest 위치교환;
        i++; // 앞으로 재배치된 값은 제외하고 그 이후부터 반복
}
```

- 분석 (데이터 개수 n)
  - 시간복잡도
    - n개 데이터 중에서 최소값을 찾는데 걸리는 시간 : O(n)
    - for 문에서 반복하는 횟수 : O(n-1)
    - 전체 시간복잡도 : $O(n^2)$ ➔ 최선/최악/평균 성능이 모두 동일
  - 입력된 순서에 대한 안정성(?)을 만족하지 않는다.

(2) 삽입정렬(insertion sort)

- 삽입정렬
  - 더 작은 값이 뒤쪽에 위치해 있다면, 적정한 위치에 끼워넣는 방식
  - 또는 부분적으로 정렬되어 있는 데이터들 사이에 적절한 위치에 삽입하는 과정



삽입정렬의 과정

- 첫번째 데이터 뒤에 위치한 데이터들에 대하여 크기 비교
- 크기가 역순이면, 더 작은 데이터를 찾아서 앞으로 계속 비교
- 적절한 자기 위치가 결정되면 그 위치로 삽입(데이터 이동이 필요함)

삽입정렬 프로그램

```
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];   // 오른쪽으로 위치 이동하여 자리 비워주기
        list[j+1] = key;     // 비워진 그 자리에 저장하기
    }
}
```

- **삽입과정에서 데이터의 이동이 많아짐**
  - ➔ **데이터가 포함된 레코드 크기가 클수록 이동에 따른 시간 손해가 큼**
- **입력순서가 안정된 정렬방법**
- **이미 정렬되어 있거나 부분적으로 정렬된 입력일 때, 더욱 효율적**

삽입정렬 복잡도 분석

- **Best-Case** : 데이터들이 이미 정렬되어 있는 경우
  - 비교횟수 : *n-1*번　　이동횟수 : *2(n-1)*번
  - 시간성능 : O(*n*)
- **Worst-Case** : 데이터들이 역순으로 정렬되어 있는 경우
  - 프로그램에서 삽입을 위해 앞에 놓인 자료를 전부 이동해야 하는 경우가 발생
    - 비교 횟수

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

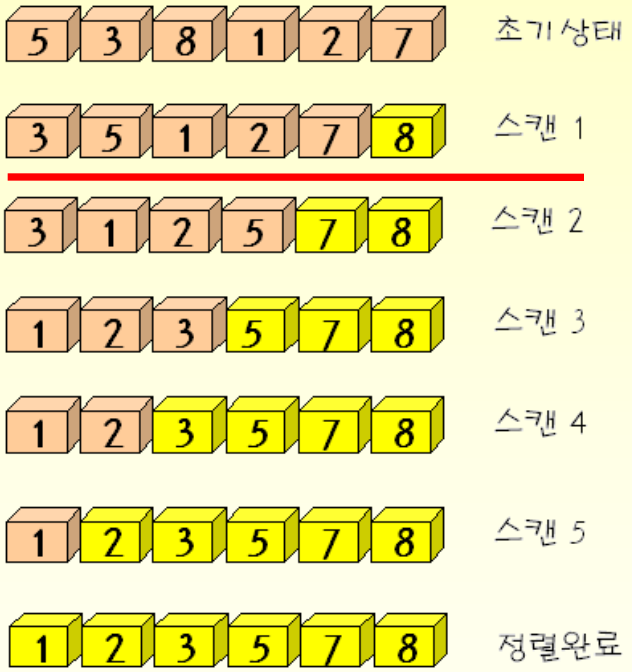
    - 이동 횟수

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

  - 시간성능 : O(n^2)
- **Average-Case** : 일반적인 입력에 대한 정렬 성능
  - 시간성능 : O($n^{1.7}$)

(3) 버블정렬(bubble sort)

- 앞뒤로 위치한 데이터 쌍이 역순서 쌍이면 서로 맞교환(아니면 패스)
- 전체가 모두 정렬될 때까지 비교/교환 동작을 계속 반복(n번 비교*n번 반복)



버블정렬 유사(Pseudo)코드

```
BubbleSort(A, n)   // 알고리즘
{
   for i←n-1 to 1 do
       for j←0 to i-1 do {
           j와 j+1번째의 값이 역순이면 위치 맞교환
           j++;  i--;
       }
}
```

```
void bubble_sort(int list[], int n)   // 프로그램
{
   int i, j, temp;
   for(i=n-1; i>0; i--){
      for(j=0; j<i; j++)
           if (list[j]>list[j+1]) // 역순서쌍이면 맞교환
              { temp = list[j]; list[j] = list[j+1], 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
  - worst-case : 역순으로 정렬되어 있으면, 이동횟수 = 2배 * 비교횟수 O(n^2)
  - best-case : 이미 정렬되어 있으면 비교횟수 O(n^2), 이동횟수 = 0번
  - average-case : 이동횟수 = 비교횟수 = O(n^2)

선택, 삽입, 버블 정렬 비교

	선택						삽입						버블					
	5	2	6	3	1	4	5	2	6	3	1	4	5	2	6	3	1	4
1R																		
2R																		
3R																		
4R																		
5R																		

(4) 쉘 정렬(Shell sort)

- **삽입정렬을 약간 변형시킨 방법**
  - 데이터 일부가 정렬된 상태로 입력될 때, 삽입정렬이 대단히 빠름에 착안한 방법
  - 삽입법을 사용하는 셀 정렬은 (원조 격인)삽입 정렬보다 빠르다.
- **삽입 정렬의 문제점 : 자리 이동할 때, 거의 이웃한 위치로만 이동**
- **쉘정렬 아이디어 : 이웃한 값끼리 비교하지 말고, 멀리 떨어진 값끼리 비교**
  ➔ 간격 gap 지정이 필요함
- **방법 :**
  - 전체 리스트를 간격 gap 만큼 떨어진 데이터들을 모아서 부분 리스트 만들기
    ➔ 부분 리스트의 데이터끼리 삽입정렬을 수행하여 부분 리스트를 정렬해놓기
  - 간격 gap은 처음에는 n/2, 다음 반복에서는 n/4, n/8, ..., 1로 줄여가면서 반복

[예] 삽입정렬 vs. 쉘정렬 비교

2 8 10 21 33 1 4 6 11 16 5 3

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

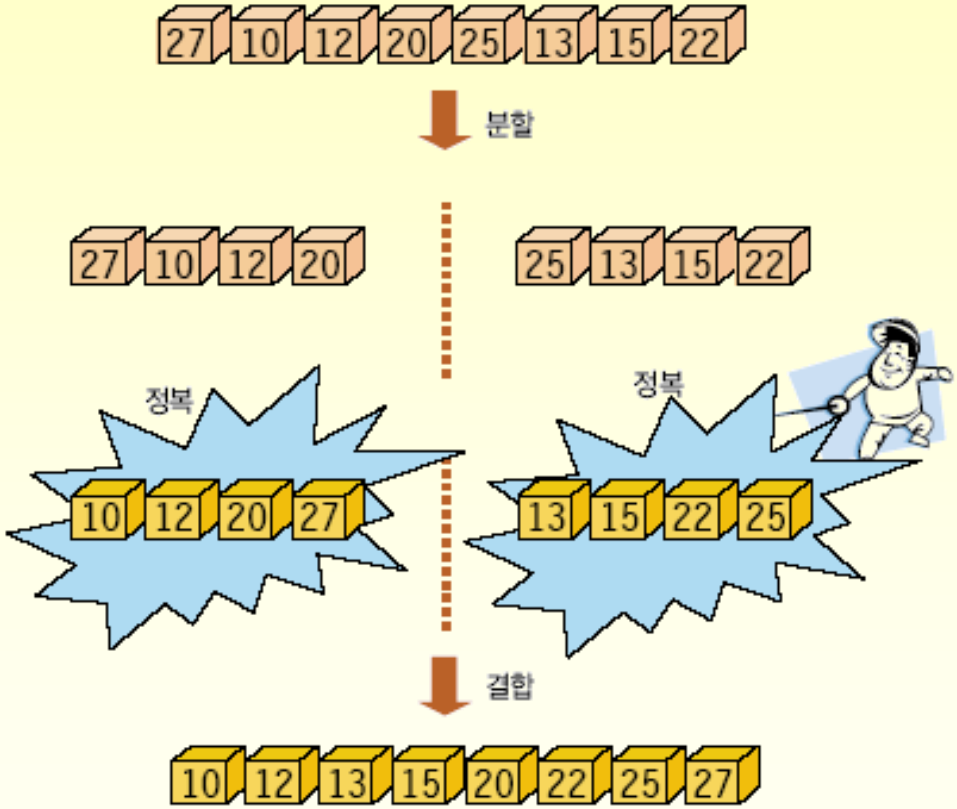
```
// gap 만큼 떨어진 요소들을 삽입 정렬, 정렬의 범위는 first에서 last
void  inc_sort(int list[], int first, int last, int gap)
{
     int  i, j, key;
     for (i=first+gap; i<=last; i=i+gap){
          key = list[i];
          for(j=i-gap; j>=first && key<list[j];j=j-gap)
               list[j+gap]=list[j];
          list[j+gap]=key;
     }
}
void  shell_sort( int list[], int n )   // n = 입력크기
{
     int  i, gap;
     for( gap=n/2; gap>1; gap = gap/2 ) {
               if( (gap%2) == 0 ) gap++;
               for(i=0;i<gap;i++)    // 부분 리스트의 개수는 gap ➔ gap개수만큼 반복
                    inc_sort(list, i, n-1, gap);
     }
}
```

- 프로그램의 정확성 점검
  - 간격(gap)은 for문에서 gap* ½ 크기로 점점 감소 ➔ 1 이하가 되면 exit

- 시간성능 점검
  - worst-case : $O(n^2)$
  - Average-case : $O(n^{1.5})$ ➔ 평균 성능이 삽입정렬($O(n^{1.7})$)보다 약간 빠름

[5] 합병정렬

- 리스트를 2개로 나누어, 부분 리스트를 각각 정렬한 다음, 다시 하나로 합치는 방법
- 합병정렬은 분할정복 기법에 기반한 정렬 알고리즘



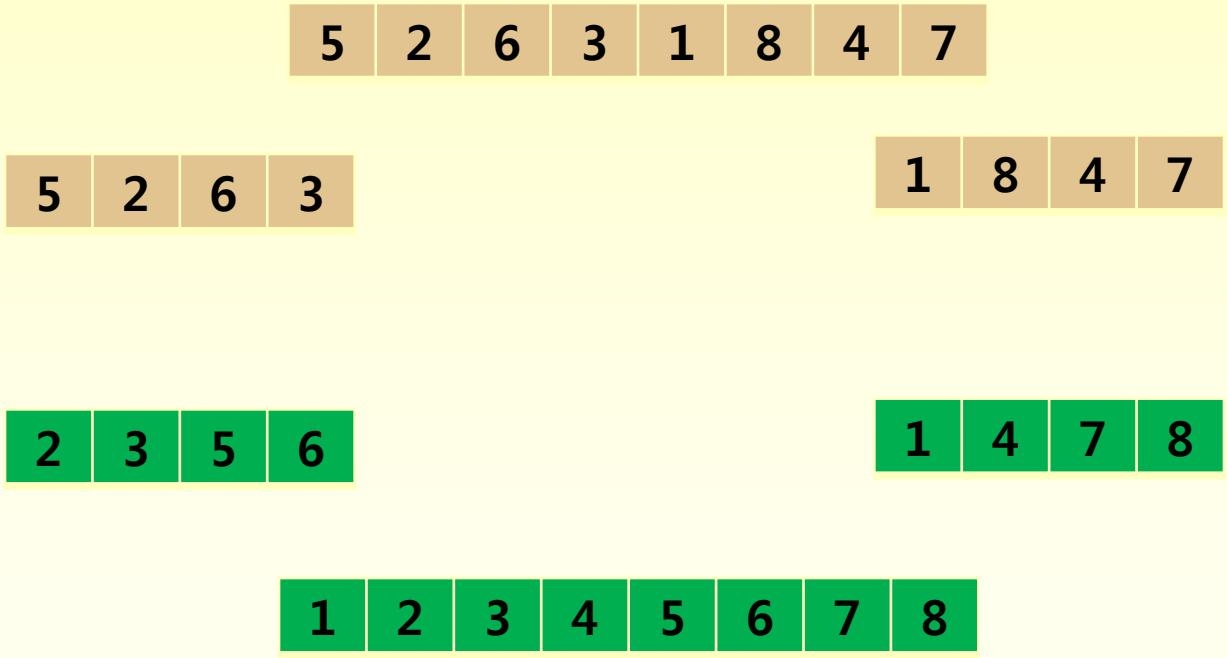
합병정렬에 사용한 분할정복법

- **분할정복법(divide and conquer)**
  - 주어진 문제를 한번에 해결할 수 없을 때, 작은 문제로 나누면 해결 가능
  - (분할) 주어진 문제를 서로 연관성이 없는 작은 문제들로 분리
  - (정복) 최소 크기로 분리된 부분 문제들을 각각 해결
  - (합병) 선택사항으로, 부분 결과들을 모아서 원래의 문제를 해결

- **분할정복법 적용과정**
  - 모든 부분 문제에 대하여 동일한 해결책을 적용해야 하므로 대상만 다를 뿐, 방법은 같다 ➔ 재귀호출(Recursion, 순환법)을 적용하면 강력한 성능
  - 2단계 까지만 사용하는 문제 예 : 이진검색
  - 3단계를 모두 사용하는 문제 예 : 합병정렬

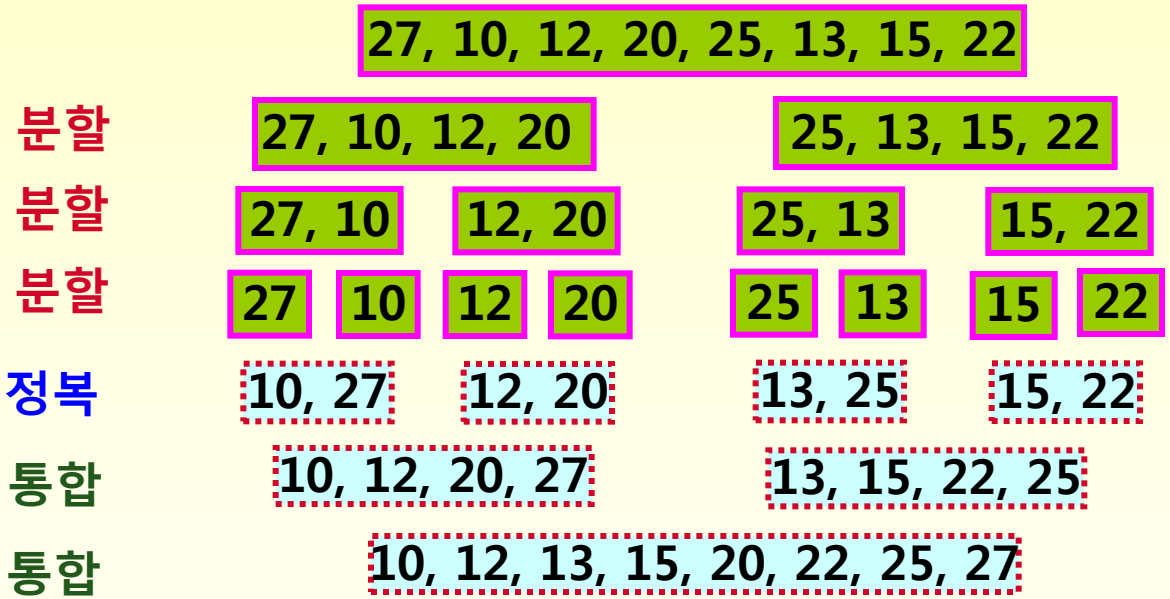
합병정렬에 사용한 분할정복법

1. 분할(Divide) (필수) 배열을 같은 크기의 2개의 부분 배열로 분할한다.
2. 정복(Conquer) (필수) 부분배열을 정렬한다. 부분배열 크기가 충분히 작지 않으면 재귀 호출을 이용하여 다시 적용한다.
3. 결합(Combine) (선택) 정렬된 부분배열을 하나의 배열에 통합한다.

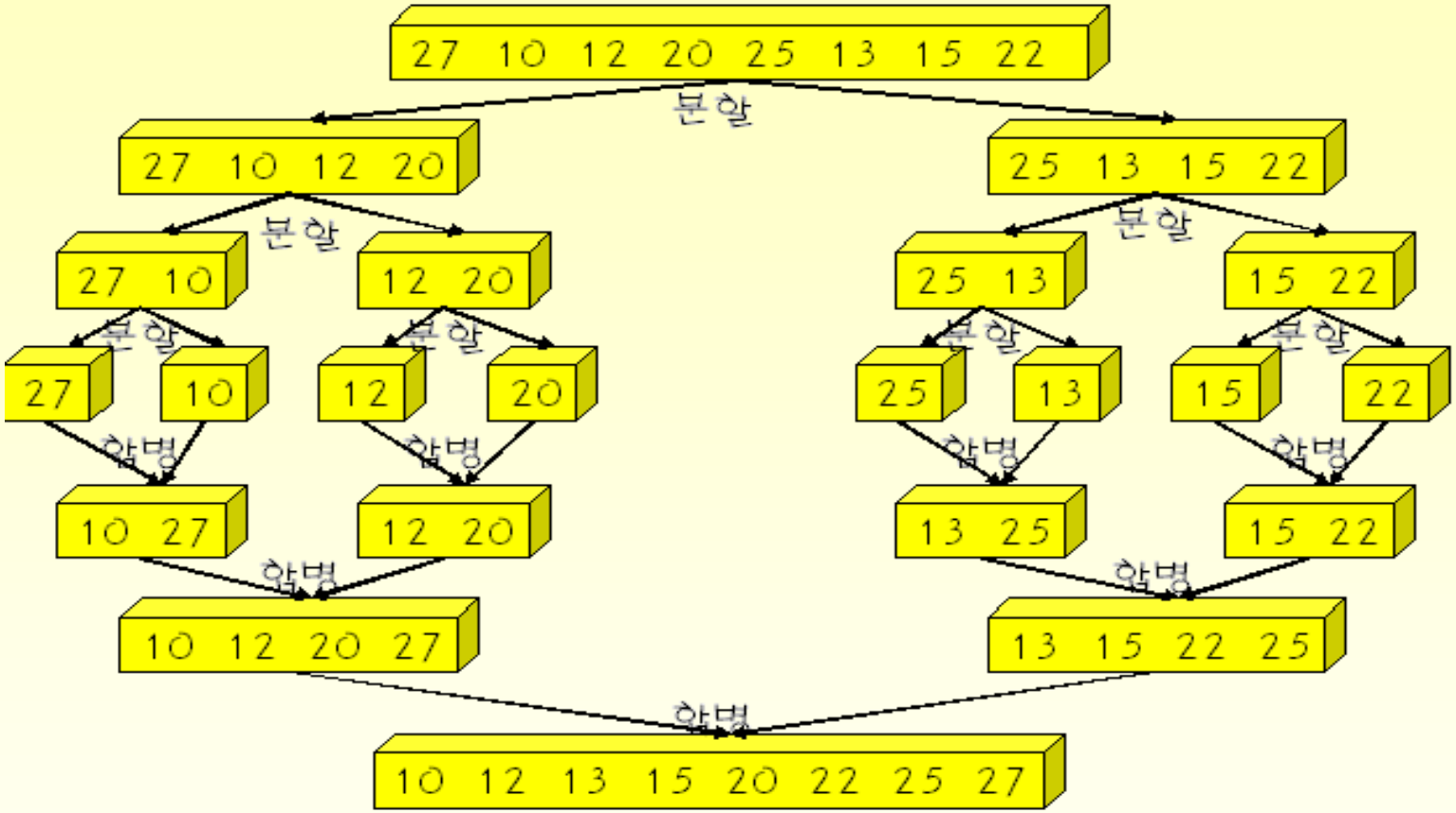


합병정렬의 전체과정

- 문제: *n*개의 정수를 비내림차순으로 정렬하시오.
- 입력: 정수 *n*, 크기가 *n*인 배열 S[1..n]
- 출력: 비내림차순으로 정렬된 배열 S[1..n]



합병정렬의 전체과정

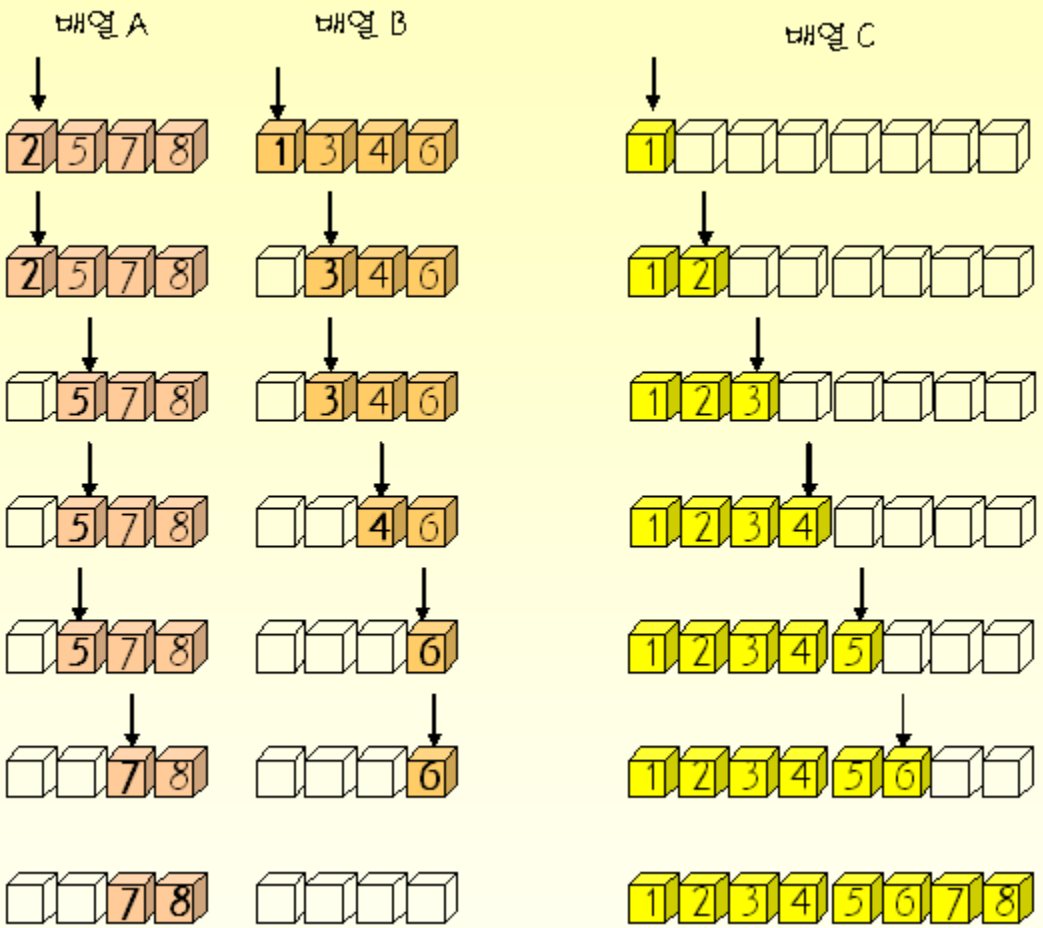

27 10 12 20 25 13 15 22
분할
27 10 12 20
25 13 15 22
분할
분할
27 10
12 20
25 13
15 22
분할
분할
분할
분할
27
10
12
20
25
13
15
22
합병
합병
합병
합병
10 27
12 20
13 25
15 22
합병
합병
10 12 20 27
13 15 22 25
합병
10 12 13 15 20 22 25 27

합병정렬 알고리즘

```
merge_sort(list, left, right)
{
    if  (left < right) {
        mid = (left+right)/2;              //중간 위치 선정
        merge_sort(list, left, mid);       //왼쪽 사이드 정렬
        merge_sort(list, mid+1, right);    //오른쪽 사이드 정렬
        merge(list, left, mid, right);     // 양쪽 사이드 합병
    }
}
```

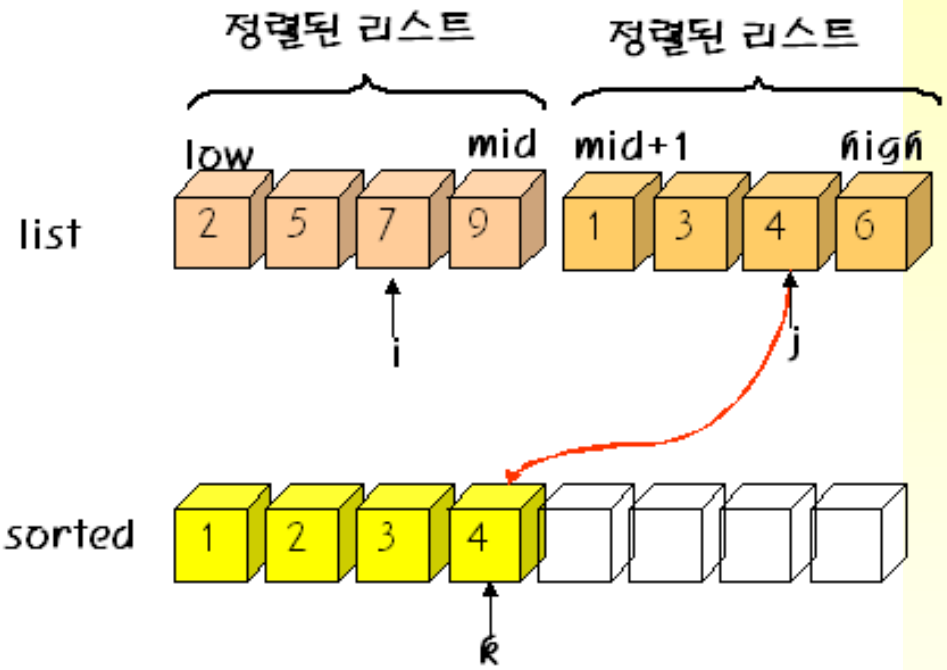
- **합병정렬은** 하나의 리스트를 2개의 균등한 크기로 분할하고 분할된 부분 리스트를 정렬한 다음, 2개의 정렬된 부분 리스트를 병합하여 정렬된 전체 리스트를 얻고자 하는 방법
- **합병정렬에서 실제로 정렬이 이루어지는 시점 : 리스트를 합병하는 단계**

합병과정



합병 알고리즘

```
merge(list, left, mid, last)   // 2개의 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])
          sorted[index]←list[b1];
          b1++;
          index++;
     else
          sorted[index]←list[b2];
          b2++;
          index++;
  요소가 남아있는 부분배열을 sorted로 복사한다;
  sorted를 list로 복사한다;
}
```



합병정렬의 분석

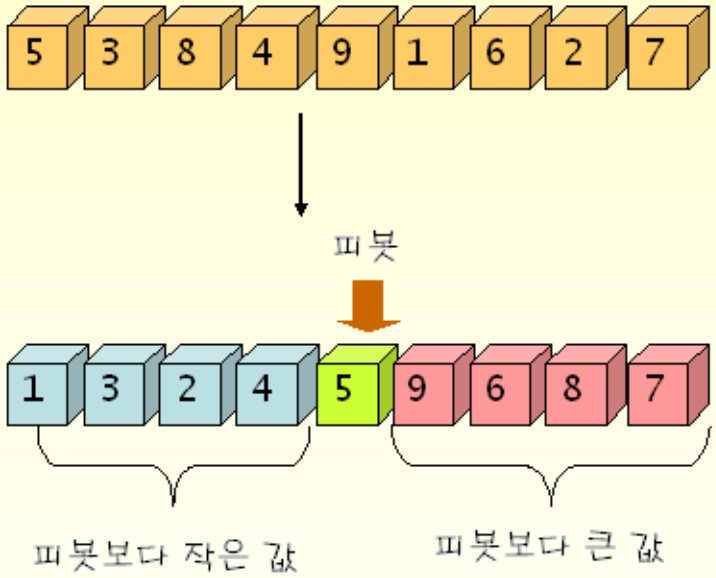
- 비교횟수 기준
  - 합병정렬은 크기 n인 리스트를 균등하게 분배하므로 반복은 log$n$ 번의 패스 수행
  - 각 패스에서 부분 리스트의 데이터를 비교하여 합병하므로 n번의 비교 연산 수행
  - 따라서 합병정렬은 최적, 평균, 최악의 경우는 동일하게 $n\log n$ 번의 비교 수행 ➔ 따라서 항상 O($n\log n$)의 시간 성능을 제공하는 알고리즘이며, 동작은 항상 안정적

- 이동횟수 기준
  - 크기 비교를 하여 위치교환하는 횟수는 각 패스에서 2n번 발생 ➔ 전체 데이터의 이동은 2n * 패스 logn번 발생 ➔ 레코드가 큰 경우에는 시간적 낭비 가능함
  - But, 레코드 크기가 크고 많은 파일을 정렬할 경우, 연결 리스트로 구성한 합병정렬은 어떤 정렬 방법보다 매우 빠르고 효율적이다.

합병정렬의 동작 예

(6) 퀵정렬(quick sort)

- 평균 성능이 가장 빠른 정렬 방법(최악 성능은 가장 느린 정렬방법)
- 합병정렬과 마찬가지로, 분할정복법을 사용함
- 전체 리스트를 2개의 리스트로 분할하고, 각 부분리스트를 순환적으로 퀵 정렬 수행



퀵정렬 알고리즘

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

3번 문장에서, 정렬할 범위가 2개 이상의 데이터이면

4번 문장에서, partition 함수를 호출하여 피벗을 기준으로 2개의 리스트로 분할한다.

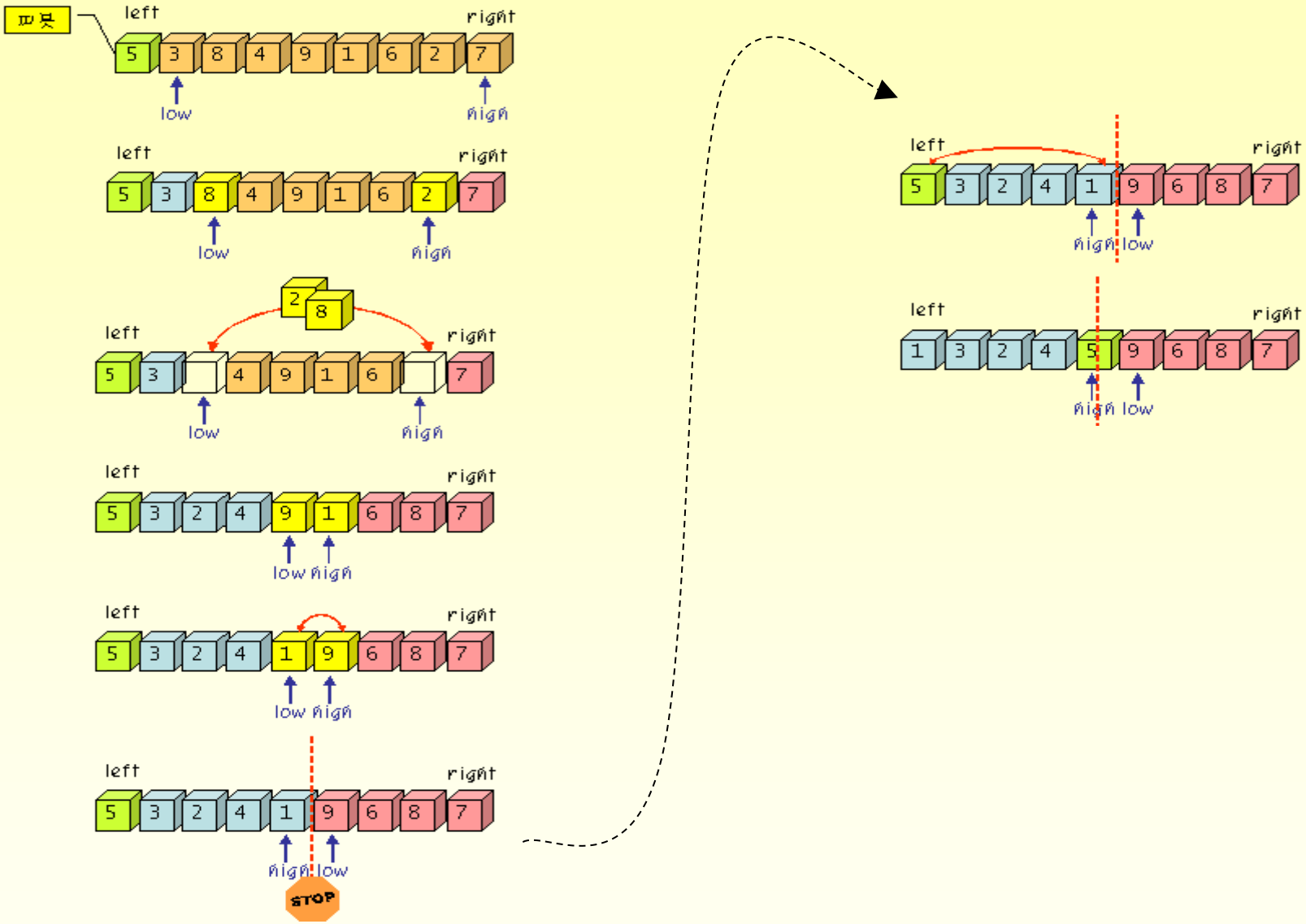
partition 함수에서 반환되는 정수는 피벗의 배열 첨자

5번 문장에서, left에서 피벗 바로 앞까지의 데이터를 대상으로 순환 호출한다(피벗은 제외).

6번 문장에서, 피벗 다음부터 right까지의 데이터를 대상으로 순환 호출한다(피벗은 제외).

partition 함수

- 피봇(pivot) : 기준되는 숫자(임의로 정하거나, 첫 번째 값을 선택)
- 2개의 배열 첨자 low와 high를 사용
  - 배열[low]가 피봇 값보다 작으면 low는 +1, 크면 정지
  - 배열[high]가 피봇 값보다 크면 high는 -1, 작으면 정지
  - 정지된 위치의 숫자(배열[low], 배열[high])를 맞교환
- 배열 첨자 low가 high보다 커지면, 더 이상 역순서쌍이 없으므로 종료

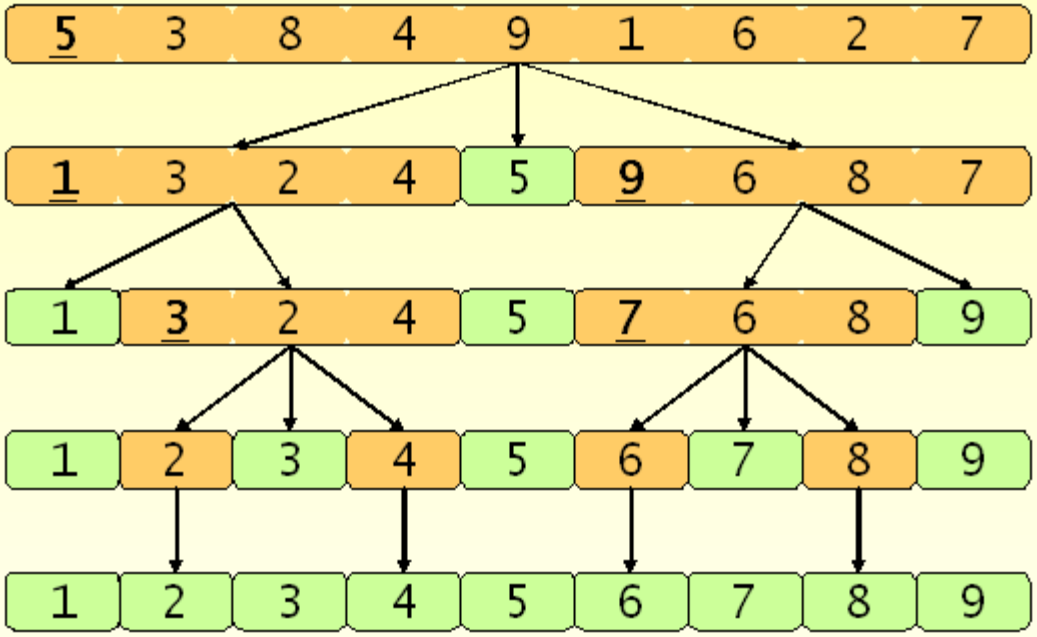

피봇
left
right
5 3 8 4 9 1 6 2 7
low
high
left
right
5 3 8 4 9 1 6 2 7
low
high
2 8
left
right
5 3 4 9 1 6 7
low
high
left
right
5 3 2 4 9 1 6 8 7
low high
left
right
5 3 2 4 1 9 6 8 7
low high
left
right
5 3 2 4 1 9 6 8 7
high low
STOP
left
right
5 3 2 4 1 9 6 8 7
high low
left
right
1 3 2 4 5 9 6 8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

partition 함수 구현

```
int  partition(int  list[], int  left, int  right)
{
     int  pivot, temp, low, high;
     low = left;  high = right+1;
     pivot = list[left];
     do {  while (low<=right &&list[low]<pivot) { low++; };
           while (high>=left && list[high]>pivot) { high--; };
           if(low<high) // 배열 값들을 맞교환
              { temp = list[low]; list[low] = list[high]; list[high] = temp); }
     } while (low<high);
     // list[pivot] ← → list[low] 맞교환
     temp = list[low]; list[low] = list[pivot]; list[pivot] = temp); }
     return high;
}
```

퀵정렬 전체과정

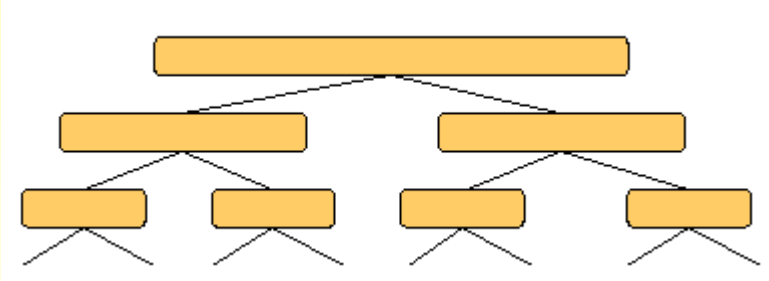
밑줄 친 숫자: 피봇



퀵정렬의 분석

- **Best Case** : partition() 함수에서 매번 균등한 리스트로 분할된다면...
  - 패스 수 : $\log^2 n$

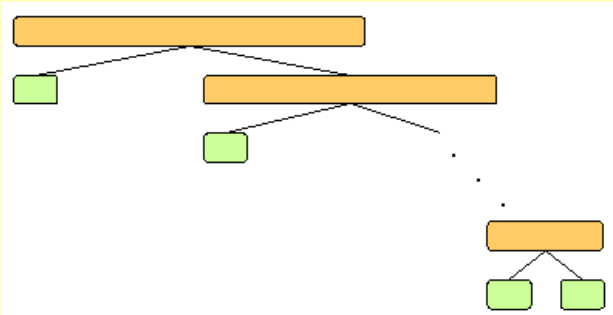
    n=2 → 1번, n=4 →2번, n=8 → 3번, ... , n → $\log^2 n$번 패스 수행



  - 각 패스에서의 비교횟수 : partition() 함수의 비교횟수 : n-1번
  - 총 비교횟수: (n-1) * $\log_2 n$ = $(n-1)\log_2 n$ - 매우 우수함
  - 총 이동횟수: 비교횟수에 비하여 무시가능

퀵정렬의 분석(cont.)

- **Worst Case** : 불균등한 리스트로 분할되는 경우
  - 패스 수 : n



  - 각 패스 안에서의 비교횟수: n-1
  - 총 비교횟수: (n-1)*n - 함수호출 횟수이므로 다른 정렬방법보다 긴 시간이 소요됨
  - 총 이동횟수: 비교횟수에 비하여 무시가능

(예) 정렬된 리스트가 주어졌을 경우, 오히려 최악의 입력상태로 동작함

```
(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
```

퀵정렬의 분석(cont.)

- **Average Case** : 균등 분할과 불균등 분할이 섞여서 수행됨
  - 패스 수 : 1.38 * $\log_2 n$번
  - 각 패스 안에서의 비교횟수: n-1
  - 총 비교횟수: $(n-1) * 1.38\log^2 n = O(n\log_2 n)$ - 다른 정렬방법보다 빠름
  - 그러나, 모든 입력에 대한 시간 성능은 최악의 경우를 우선 고려하므로 회피됨

(7) 힙정렬(Heap Sort)

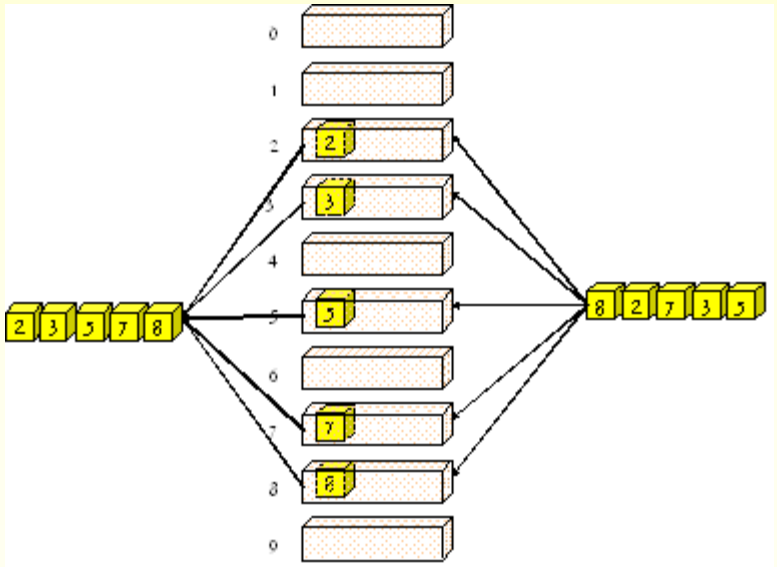
- 입력된 데이터들을 힙 구조에 저장
- 정렬 방법
  - 오름차순 정렬 : Min-Heap으로 구성(루트에는 최소값이 저장되어 있음)
  - 내림차순 정렬 : Max-Heap으로 구성(루트에는 최대값이 저장되어 있음)

- 성능
  - heap 구성하는 시간 : 1개 입력할 때마다 평균 $O(\log_2 n)$ 이므로,
    n개 입력하면 $O(n\log_2 n)$
  - heap 출력하는 시간 : 1개 출력할 때마다 평균 $O(\log_2 n)$ 이므로,
    n개 출력하면 $O(n\log_2 n)$

(8) 기수정렬(Radix Sort)

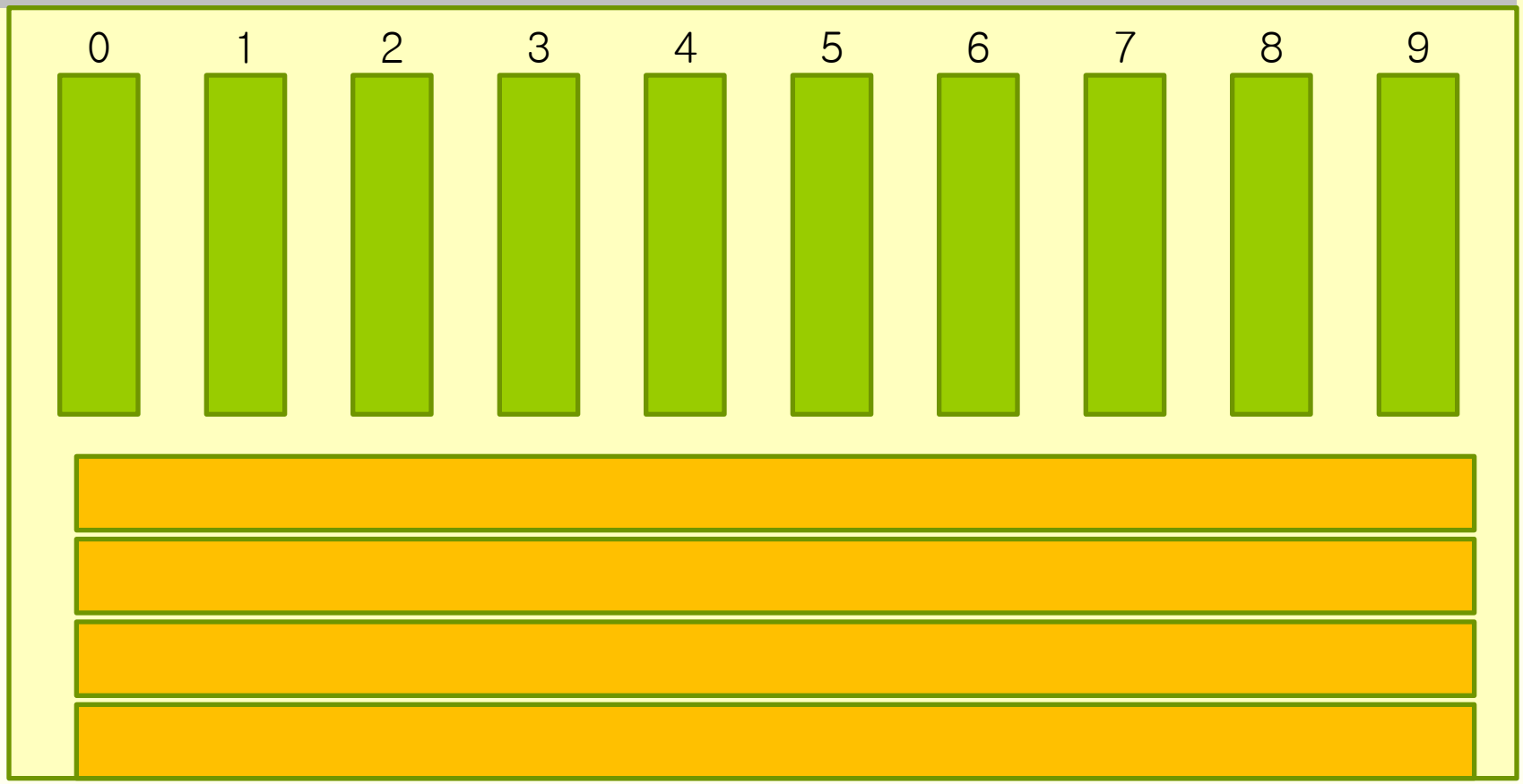
- 이때까지의 정렬 방법들은 모두 레코드들을 비교하여 정렬하였지만, 기수 정렬은 레코드의 값을 비교하지 않고 정렬하는 방법
- 기수 정렬(radix sort)은 $O(n\log_2 n)$ 이라는 정렬의 이론적인 하한선을 깰 수 있는 유일한 방법이 될 수도 있음
- 기수 정렬은 O(k*n) 의 시간 복잡도를 가지는데, 일반적으로 k는 4~10
- 기수 정렬의 단점 : 정렬할 수 있는 데이터의 종류는 제한적 ➔ 레코드 키값이 동일한 길이를 가지는 숫자나 문자열로 구성

(예) 한자리수의 기수정렬
(8, 2, 7, 3, 5)
: 단순히 자리수에 따라 버킷에 넣었다가 꺼내면 정렬됨



기수정렬

128 93 390 81 623 72 38 2613



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.

정렬 알고리즘의 비교

알고리즘	최선	평균	최선
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	$O(dn)$	$O(dn)$	$O(dn)$

- **각 정렬 알고리즘을 적용할 입력상태를 미리 분석할 수 있다면?**
  가장 빠른 정렬방법을 적용할 수 있음