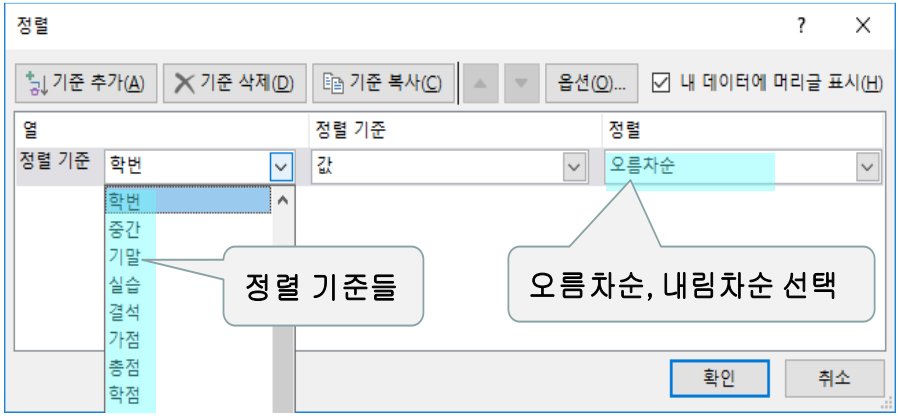
정렬

Sort

정렬

정렬이란?

- 물건을 크기순으로 오름/내림차순으로 나열하는 것
- 컴퓨터공학을 포함한 모든 과학기술 분야에서 가장 기본적이고 중요한 알고리즘 중 하나
- 자료 탐색에 있어 필수



정렬

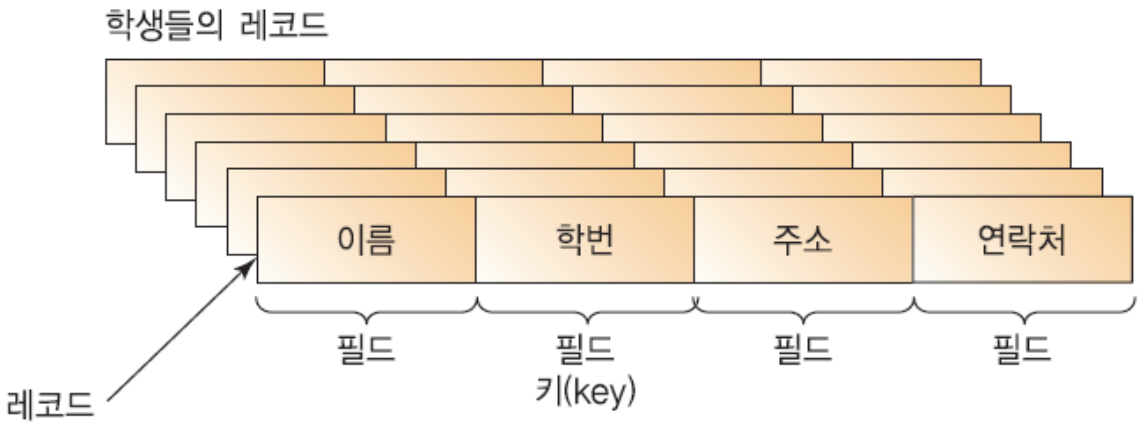
정렬 대상

- 일반적으로 정렬시켜야 될 대상은 레코드(record)
  [레코드는 필드(field)라는 보다 작은 단위로 구성]

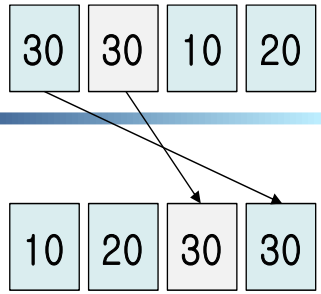
열 = 필드

	A	B	C
1	…	…	…
2	…	…	…
3	…	…	…
4	…	…	…

행 = 레코드



정렬

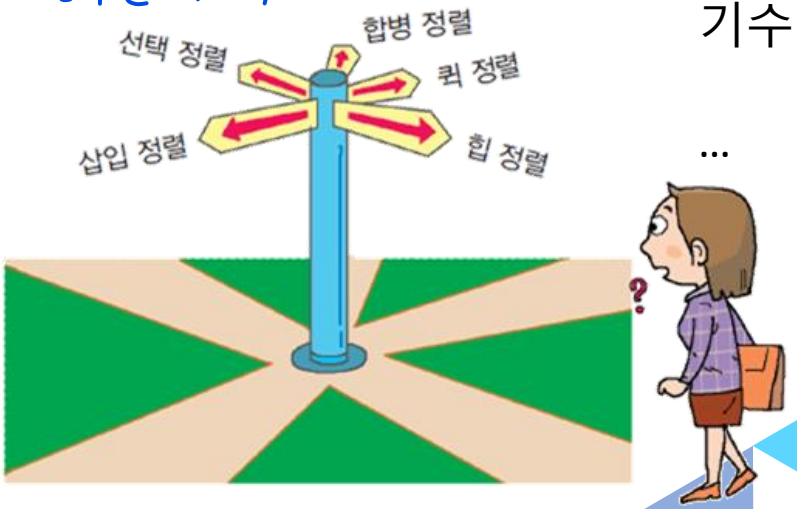


종류

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

삽입 정렬
선택 정렬
버블 정렬
} 단순

퀵 정렬
...
(우선순위 큐) 히프 정렬
합병 정렬
기수 정렬
...



정렬

선택 정렬 (selection sort)

- 기초적인 정렬 알고리즘 중 하나
- 정렬된 왼쪽 리스트와 정렬 안된 오른쪽 리스트를 가지는 형태
- 소요 시간 $\Theta(n^2)$ = (n-1) + (n-2) + … + 2 + 1

왼쪽 리스트 (정렬된 리스트)	오른쪽 리스트 (정렬 안 된 리스트)	설명
( )	(5,3,8,1,2,7) 가장 작은 것	초기상태
(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

정렬

선택 정렬 (selection sort)

큰 수 찾기 : 정렬된 부분 오른쪽
작은 수 찾기 : 정렬된 부분 왼쪽

1. 정렬할 배열이 주어진다.
2. 가장 큰 수를 찾는다. (또는 작은 수를 찾는다)
3. 큰 수를 맨 오른쪽 원소와 교환한다.
   (작은 수를 찾을 경우 맨 왼쪽 원소와 교환한다)
4. 맨 오른쪽 원소는 정렬에서 제외한다.
   (작은 수로 한 경우는 맨 왼쪽 원소는 정렬에서 제외)

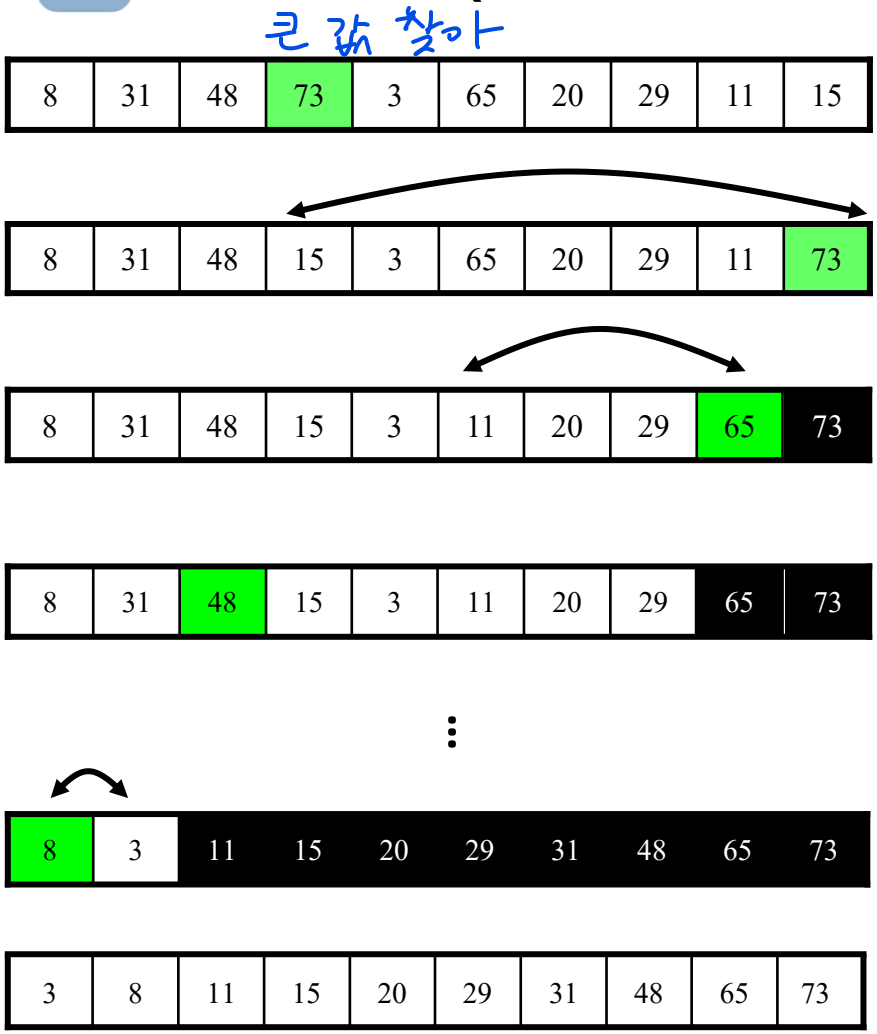
↓ 맨 왼쪽부터 탐색

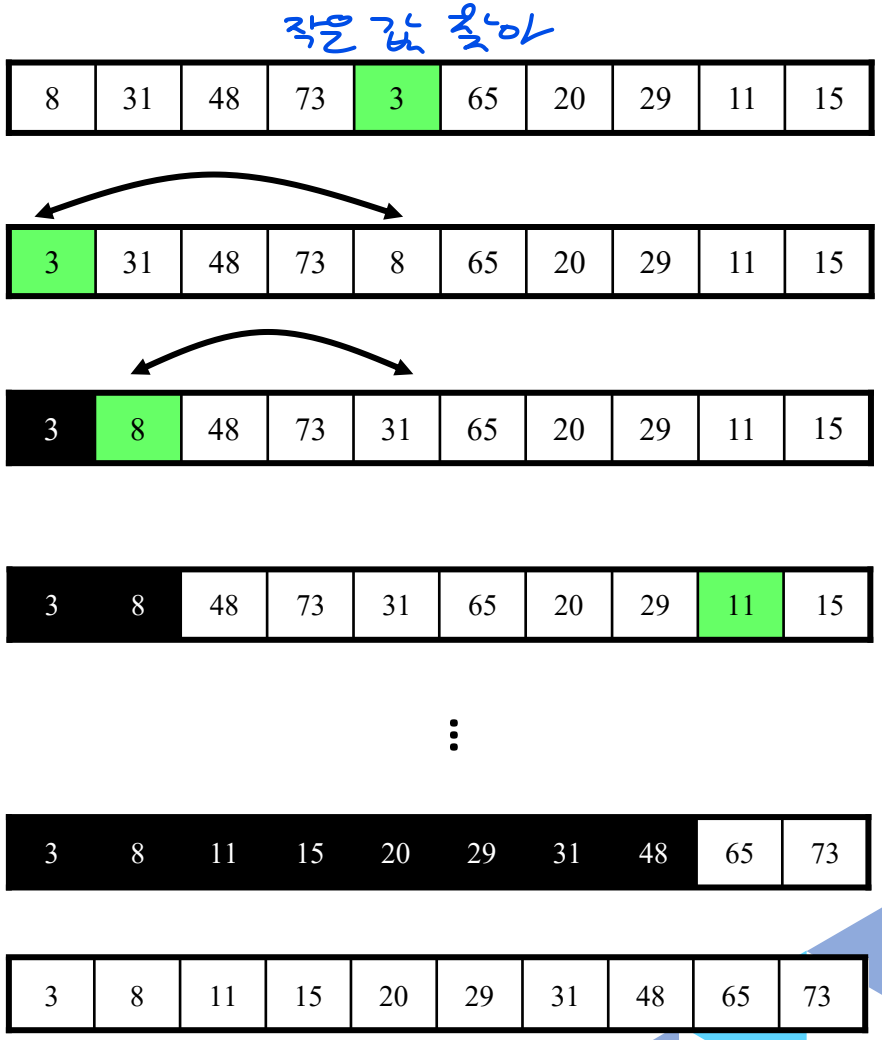
8	31	48	73	3	65	20	29	11	15

3	8	11	15	20	29	31	48	65	73

정렬

선택 정렬 (selection sort)





정렬

구현 방법 (선택 정렬 - 최소)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
int list[MAX_SIZE];
int n;
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++)
      if (list[j] < list[least])
        least = j;
    SWAP(list[i], list[least], temp);
  }
}
```

$O(n^2)$

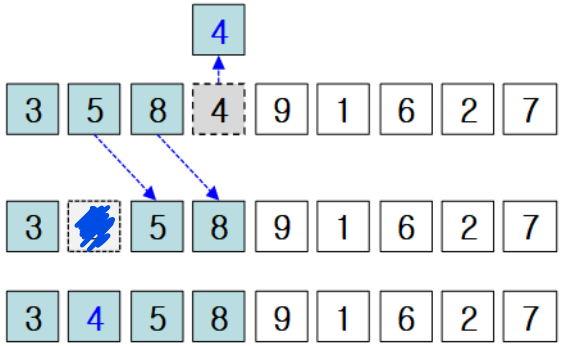
```c
int main(void){
  int i;
  n = MAX_SIZE;
  srand(time(NULL));
  for (i = 0; i<n; i++)
    list[i] = rand() % 100;
  selection_sort(list, n);
  for (i = 0; i<n; i++)
    printf("%d ", list[i]);
  printf("₩n");
  return 0;
}
```

정렬

삽입 정렬 (insert sort)

$O(n^2)$

- 정렬되어 있는 부분에 새로운 레코드를 올바른 위치에 삽입하는 과정 반복
- 항목 이동 필요



정렬

삽입 정렬 (insert sort)

정렬

구현 방법 (삽입 정렬)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
int list[MAX_SIZE];
int n;

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];
    list[j + 1] = key;
  }
}
```

$O(n^2)$

```c
int main(void){
  int i;
  n = MAX_SIZE;
  srand(time(NULL));
  for (i = 0; i<n; i++)
    list[i] = rand() % 100;
  insertion_sort(list, n);
  for (i = 0; i<n; i++)
    printf("%d ", list[i]);
  printf("₩n");
  return 0;
}
```

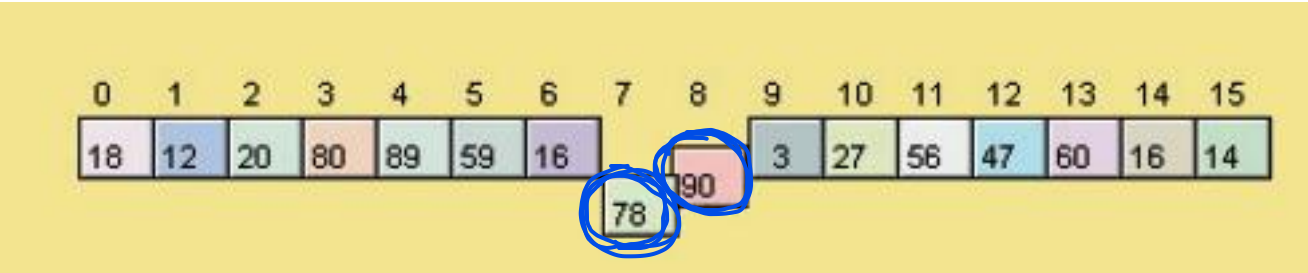
정렬

버블 정렬 (bubble sort)

- 인접한 2개의 레코드를 비교하여 순서대로 되어 있지 않으면 서로 교환

- 이러한 비교-교환 과정을 리스트의 왼쪽 끝에서 오른쪽 끝까지 반복(스캔)
  * 한번 스캔이 완료되면 리스트 오른쪽 끝에 가장 큰 레코드가 이동함

- 끝으로 이동한 레코드를 제외한 왼쪽 리스트에 대하여 위 과정 반복



정렬

버블 정렬 (bubble sort)

뒤에서 부터 정렬

정렬

구현 방법 (버블 정렬) 삽입 정렬처럼 '이동 많음'

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )

int list[MAX_SIZE];
int n;

void bubble_sort(int list[], int n){
  int i, j, temp;
  for (i = n - 1; i > 0; i--) {
    for (j = 0; j < i; j++)
      if (list[j] > list[j + 1])
        SWAP(list[j], list[j + 1], temp);
  }
}
```

```c
int main(void){
  int i;
  n = MAX_SIZE;
  srand(time(NULL));
  for (i = 0; i<n; i++)
    list[i] = rand() % 100;
  bubble_sort(list, n);
  for (i = 0; i<n; i++)
    printf("%d ", list[i]);
  printf("₩n");
  return 0;
}
```

정렬

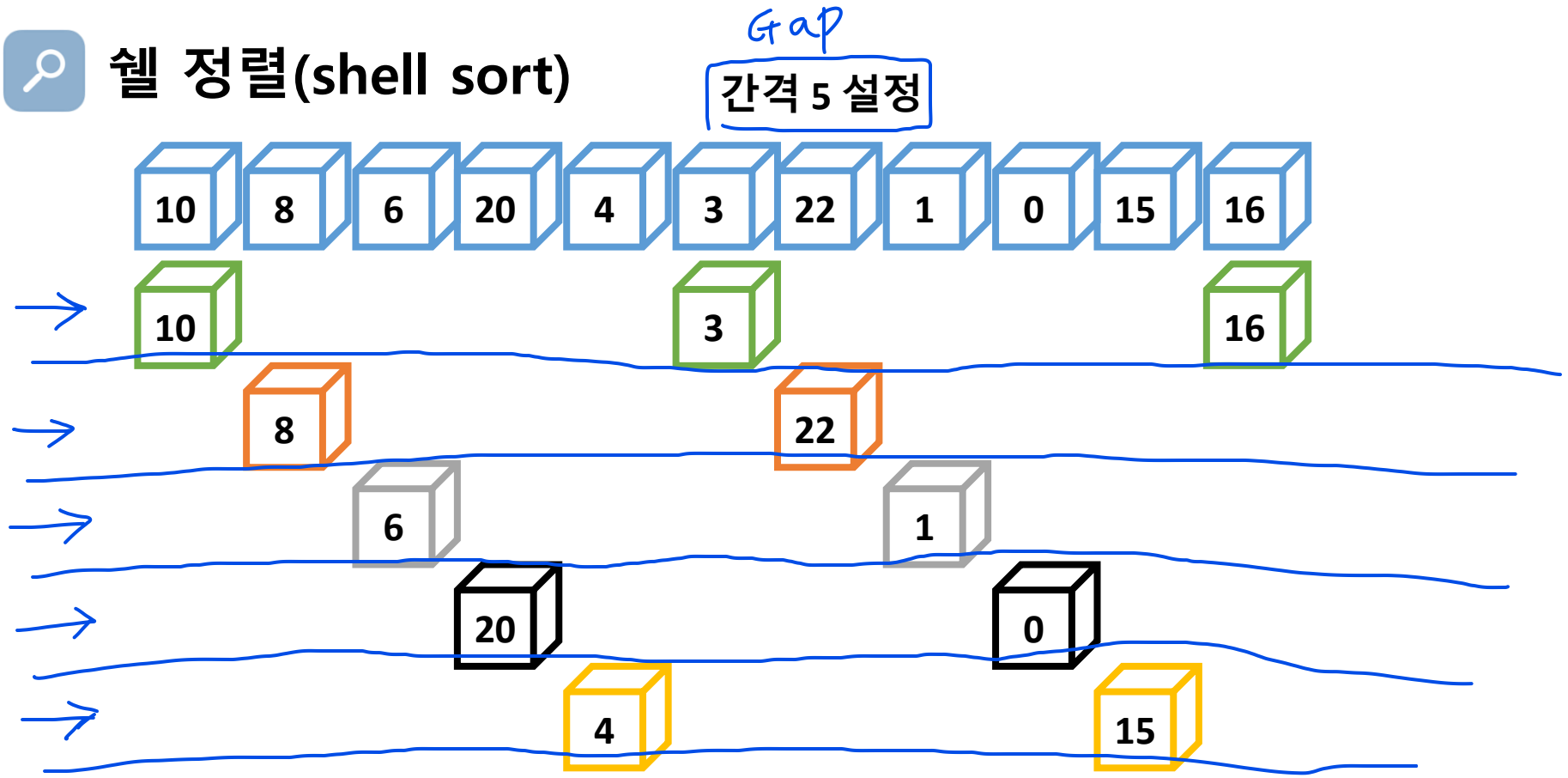
쉘 정렬(shell sort)

- 셸 정렬의 경우 “삽입정렬”이 어느 정도 정렬된 리스트에서는 대단히 빠른 것에 착안
  - 삽입 정렬은 요소들이 이웃한 위치로만 이동하므로, 많은 이동에 의해서만 요소가 제자리를 찾아 감
  - 요소들이 멀리 떨어진 위치로 이동할 수 있게 한다면 보다 적게 이동하여 제자리를 찾을 수 있음

- 과정
  - 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 반복

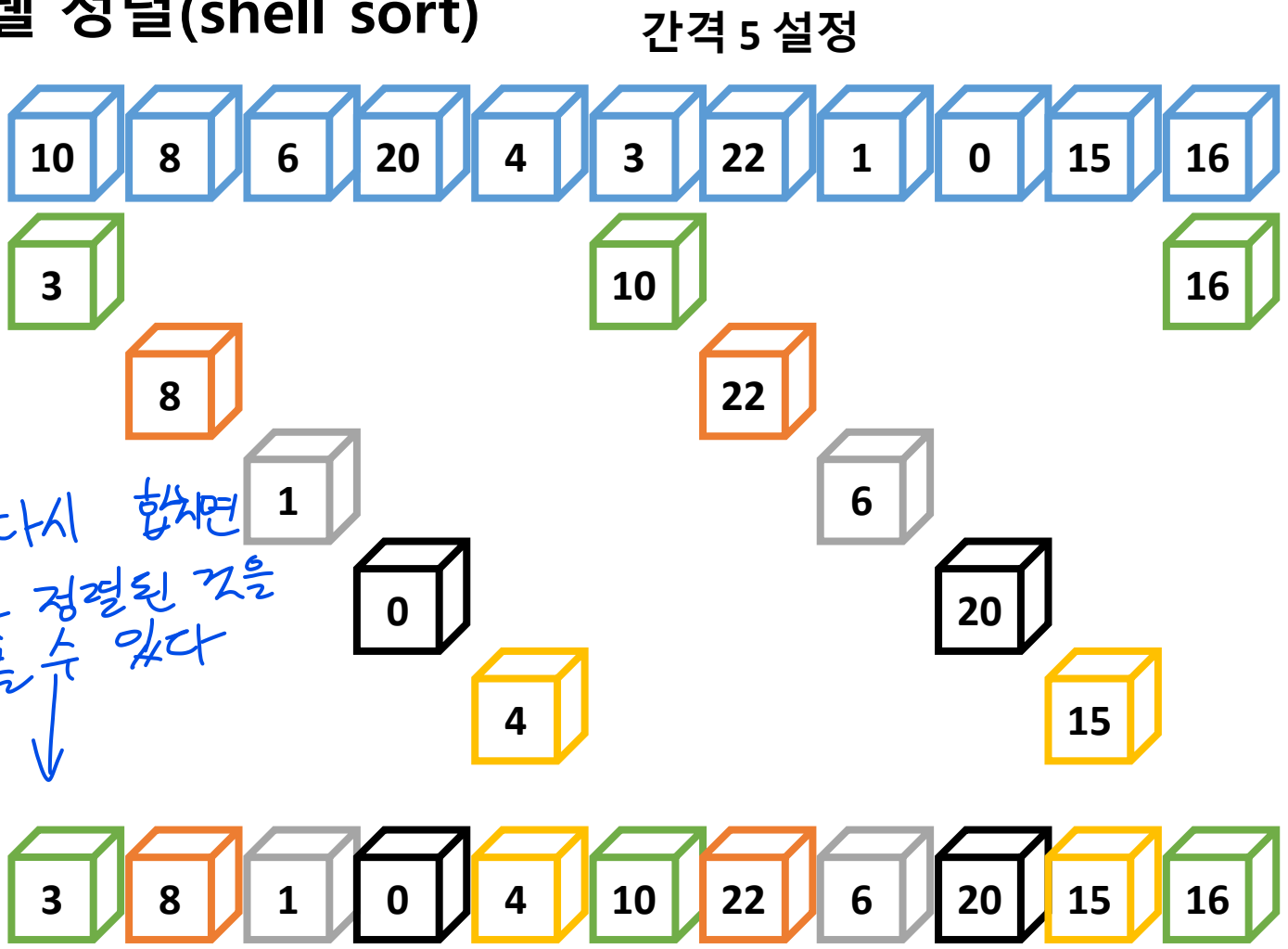
정렬

쉘 정렬(shell sort)



정렬

쉘 정렬(shell sort)

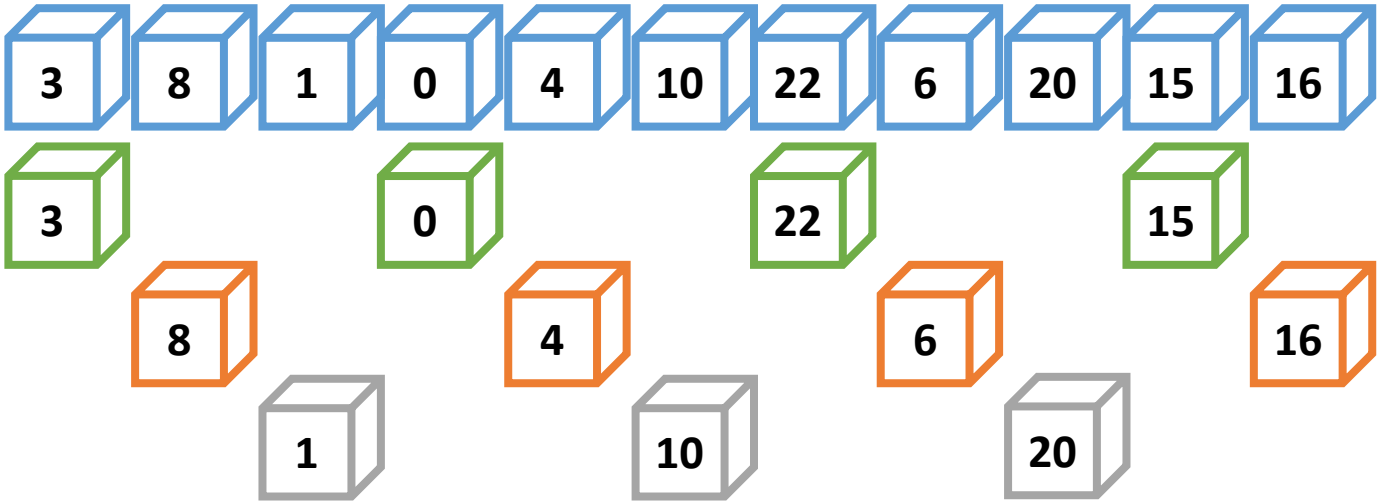


정렬

쉘 정렬(shell sort)

또 할거임

간격 3 설정



정렬

쉘 정렬(shell sort)

정렬

구현 방법 (쉘 정렬)

```
...
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){
  int i, gap;
  for (gap = n / 2; gap > 0; gap = gap / 2) {
    if ((gap % 2) == 0) gap++;
      for (i = 0; i < gap; i++)
        inc_insertion_sort(list, i, n - 1, gap);
  }
}
...
```

정렬

쉘 정렬(shell sort) 장점

- 불연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치 교환으로 제자리 찾을 가능성 증대
- 부분 리스트가 점진적으로 정렬된 상태가 되므로 삽입정렬 속도 증가

시간적 복잡도

최악의 경우 O(n^2)
평균적인 경우 O($n^{1.5}$)

정렬

합병 정렬(merge sort)

- 리스트를 2 개의 균등한 크기로 분할하고 분할된 부분 리스트를 정렬

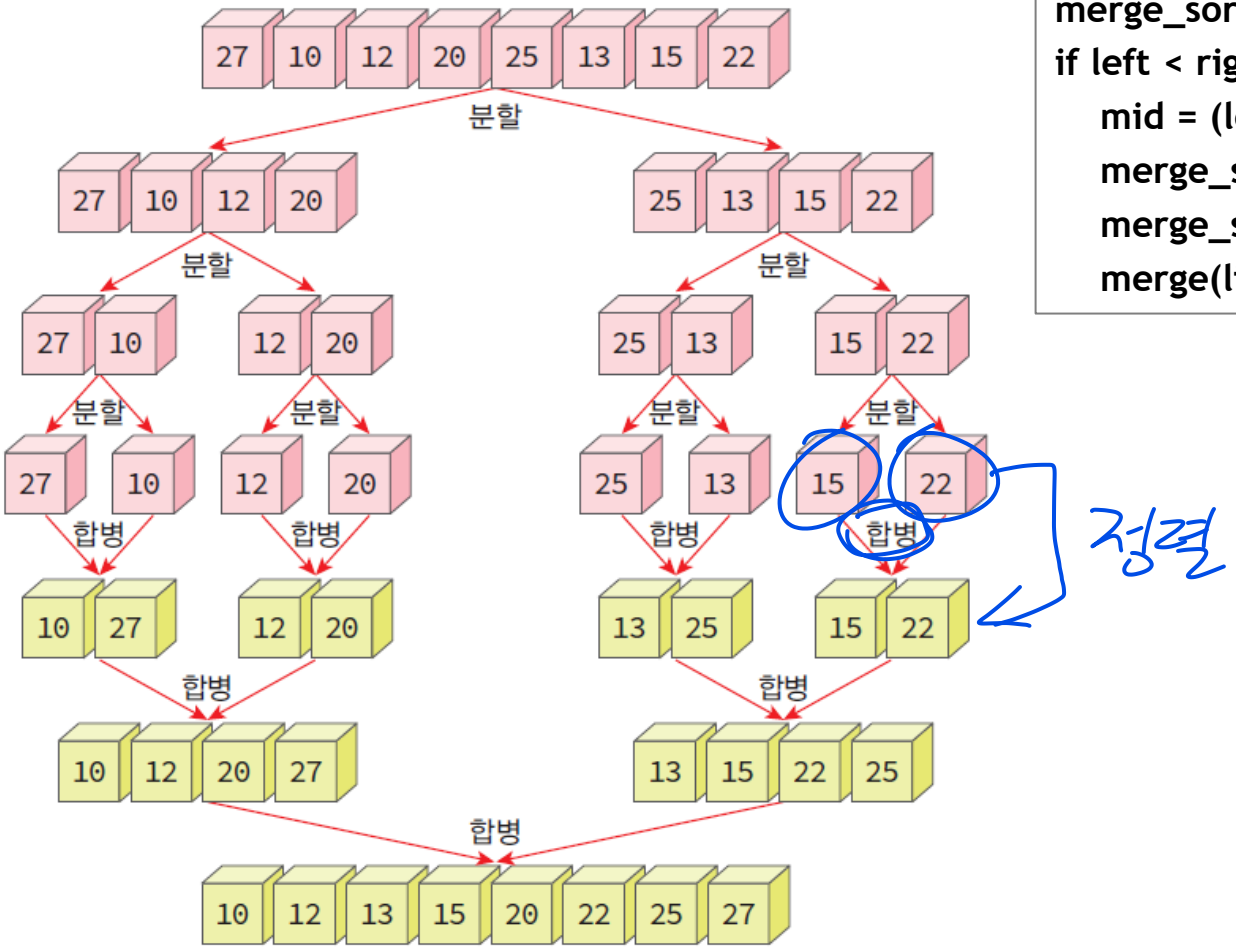
- 정렬된 두 개의 부분 리스트를 합하여 전체 리스트를 정렬

- 분할 정복 기법

꼬개기

정렬

합병 정렬(merge sort) – 전체 과정

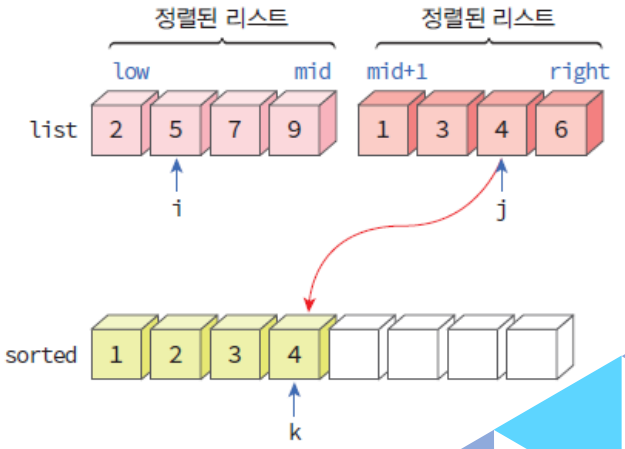



```
merge_sort(list, left, right)
if left < right
    mid = (left+right)/2;
    merge_sort(list, left, mid);
    merge_sort(list, mid+1, right);
    merge(list, left, mid, right);
```

정렬

합병 - 과정


```
merge(list, left, mid, right):
i←left;
j←mid+1;
k←left;
while i≤ mid  and j≤right do
 if(list[i]<list[j])
    then
      sorted[k]←list[i];
      k++;
      i++;
    else
      sorted[k]←list[j];
      k++;
      j++;
```



정렬

구현 방법 (합병 정렬)

```
...
void merge(int list[], int left, int mid, int right){
  int i, j, k, l;
  i = left;  j = mid + 1;  k = left;
  while (i <= mid && j <= right) {
    if (list[i] <= list[j])
      sorted[k++] = list[i++];
    else
      sorted[k++] = list[j++];
  }
  if (i > mid)
    for (l = j; l <= right; l++)
      sorted[k++] = list[l];
  else
    for (l = i; l <= mid; l++)
      sorted[k++] = list[l];
    for (l = left; l <= right; l++)
      list[l] = sorted[l];
}
```

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

정렬

합병 정렬(merge sort) 복잡도 분석

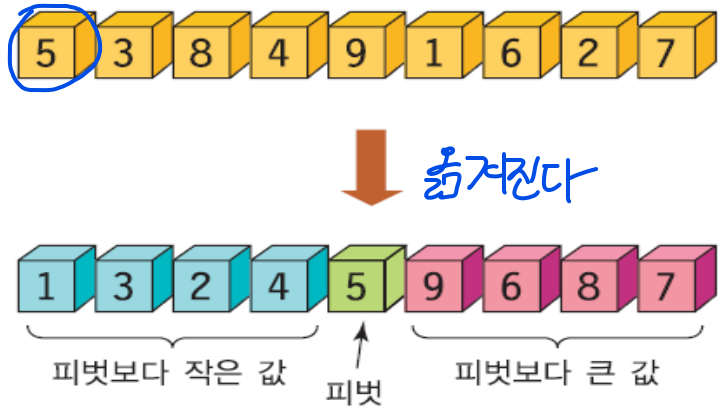
- 비교 횟수
  - 크기 n인 리스트를 정확히 균등 분배하므로 log(n)개 패스
  - 각 패스에서 리스트 모든 레코드 n개를 비교하므로 n번의 비교 연산
- 이동 횟수
  - 레코드의 이동이 각 패스에서 2n 발생하므로 전체 레코드의 이동은 2n*log(n)번 발생
  - 레코드 크기가 큰 경우에는 매우 큰 시간적 낭비 초래
  - 레코드를 연결 리스트로 구성하여 합병 정렬할 경우, 매우 효율적
- 최적, 평균, 최악의 경우 큰 차이 없이 O(n*log(n))의 복잡도
- 안정적이며 데이터 초기 분산 순서에 영향을 덜 받음

정렬

퀵 정렬(quick sort)

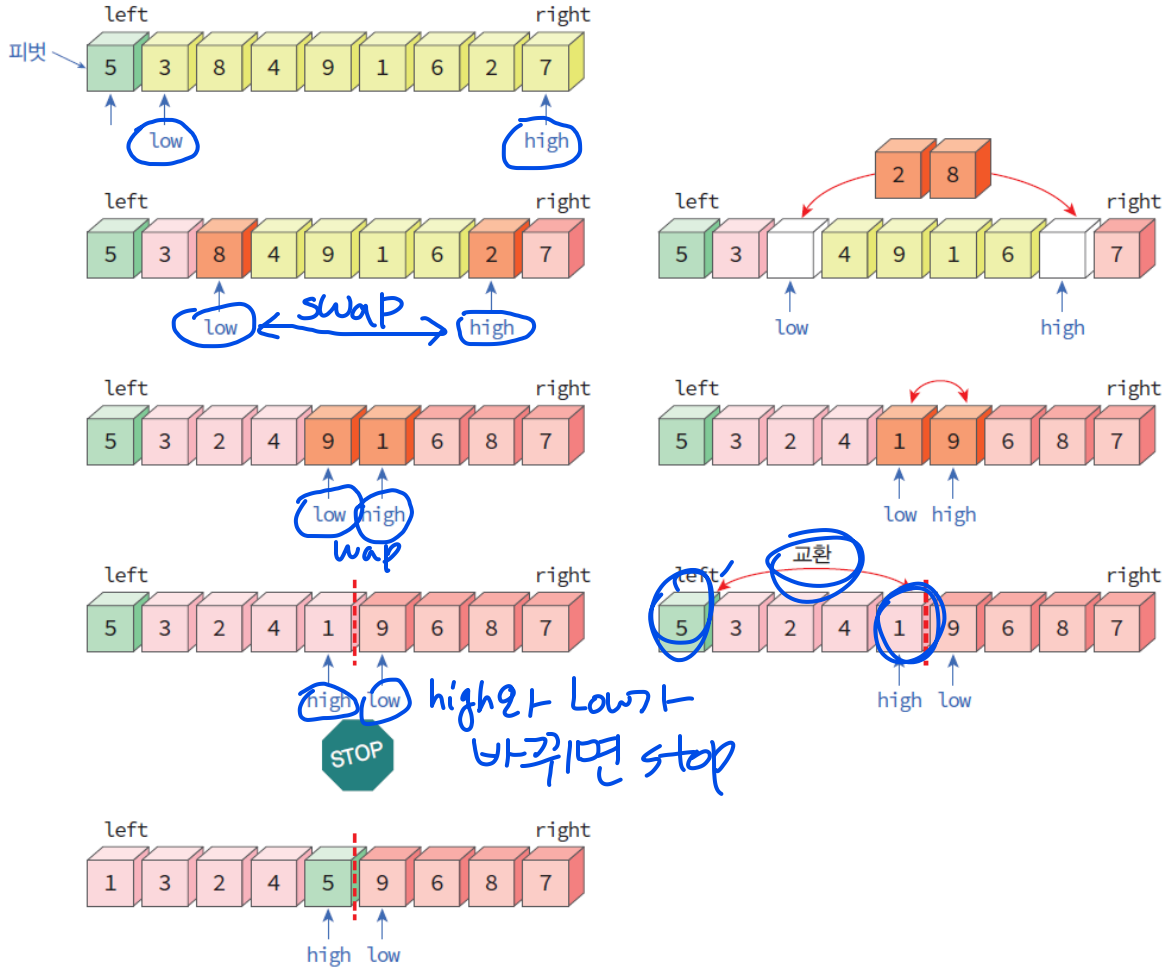
- 평균적으로 가장 빠른 정렬 방법
- 합병과 동일하게 분할 정복법 사용
- 리스트를 2개의 부분리스트로 비균등 분할하고, 각 부분리스트를 다시 퀵정렬 (재귀호출)

맨 앞이 피벗



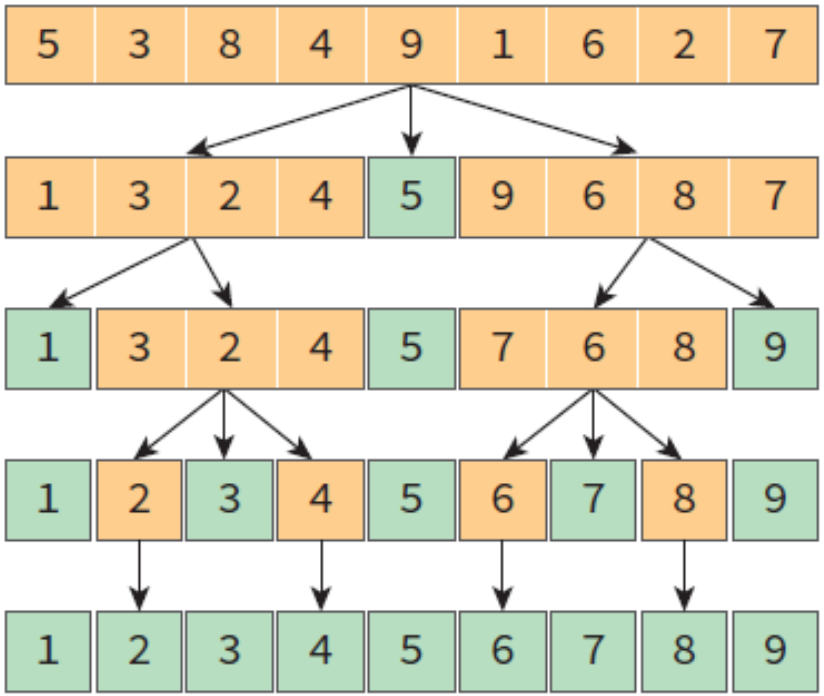
정렬

퀵 정렬(quick sort)



정렬

퀵 정렬(quick sort)



정렬

구현 방법 (퀵 정렬)

```
...
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 (list[low] < pivot);
    do
      high--;
    while (list[high] > pivot);
    if (low < high) SWAP(list[low], list[high], temp);
  } while (low < high);
  SWAP(list[left], list[high], temp);
  return high;
}
```

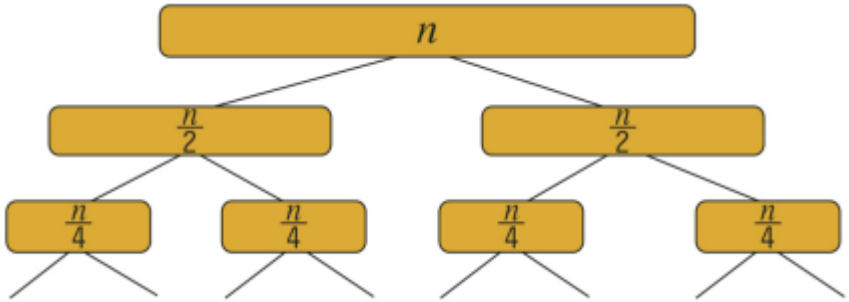
```
void quick_sort(int list[], int left, int
right){
      if (left < right) {
      int q = partition(list, left, right);
      quick_sort(list, left, q - 1);
      quick_sort(list, q + 1, right);
      }
}
```

정렬

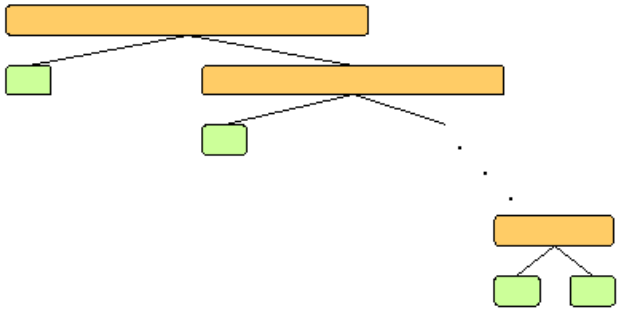
피벗을 어떻게 사용하는가에 따라 달라짐

퀵 정렬 복잡도 분석

- 최선의 경우와 최악의 경우
- 최선: 거의 균등한 리스트로 분할되는 경우, log(n)
  - 2 -> 1
  - 4 -> 2
  - 8 -> 3
  - ...
  - n->log(n)



- 최악: 극도로 불균등한 리스트로 분할 n^2 까지도 가능

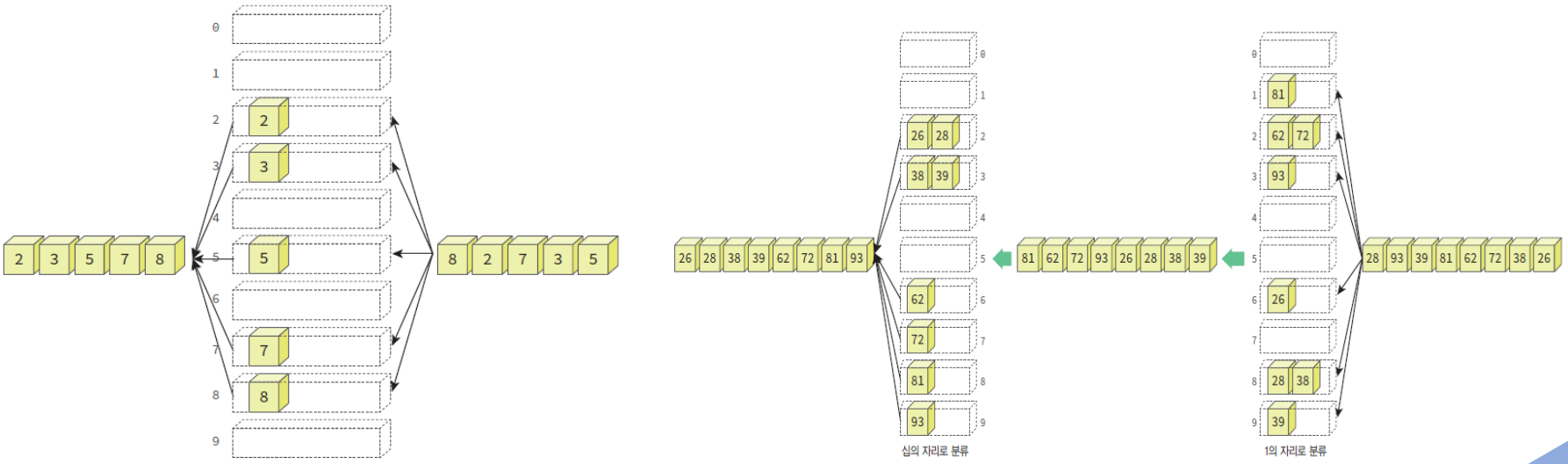


정렬

숫자의 자리수

기수 정렬(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)



정렬

기수 정렬(Radix sort)

```
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++;
    }
```

- 버켓 = 큐로 구현
- 버켓 개수는 키의 표현 방법과 밀접한 관계
  - 이진법을 사용한다면 버켓 = 2개
  - 알파벳 문자를 사용한다면 버켓은 26개
  - 십진법을 사용한다면 버켓은 10개

정렬

기수 정렬 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

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

정렬 알고리즘 실험 (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