

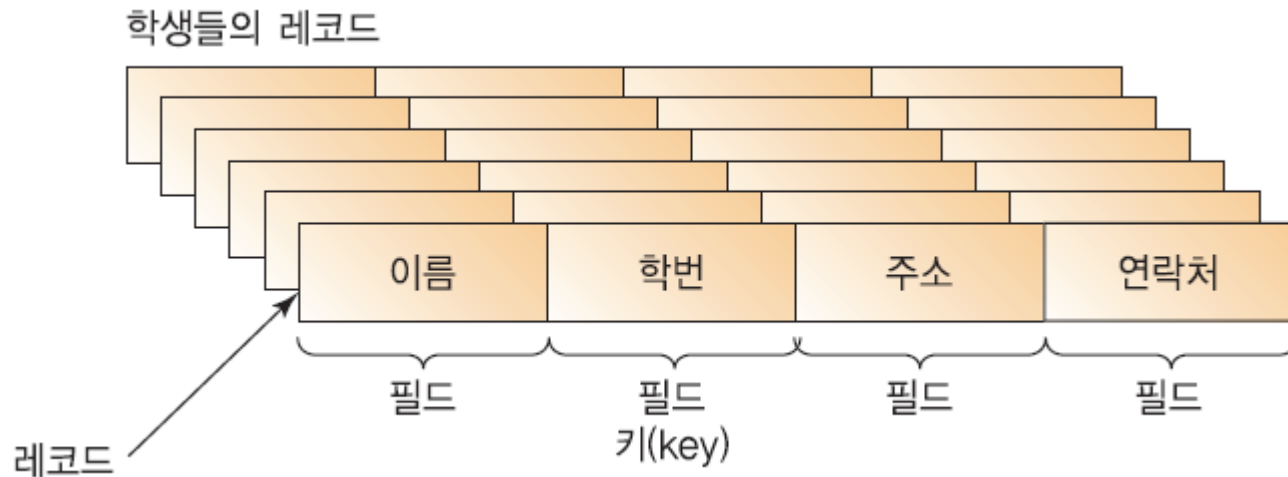
## 12장 정리

# 정렬이란?

- 정렬은 물건을 크기 순으로 오름차순이나 내림차순으로 나열하는 것
  - 정렬은 컴퓨터공학을 포함한 모든 과학기술 분야에서 가장 기본적이고 중요한 알고리즘 중 하나
  - 정렬은 자료 탐색에 있어서 필수적
- (예) 만약 영어사전에서 단어들이 알파벳 순으로 정렬되어 있지 않다면?



- 일반적으로 정렬시켜야 될 대상은 레코드(record)
- 레코드는 필드(field)라는 보다 작은 단위로 구성
- 키필드로 레코드와 레코드를 구별한다.





# 정렬 알고리즘 개요

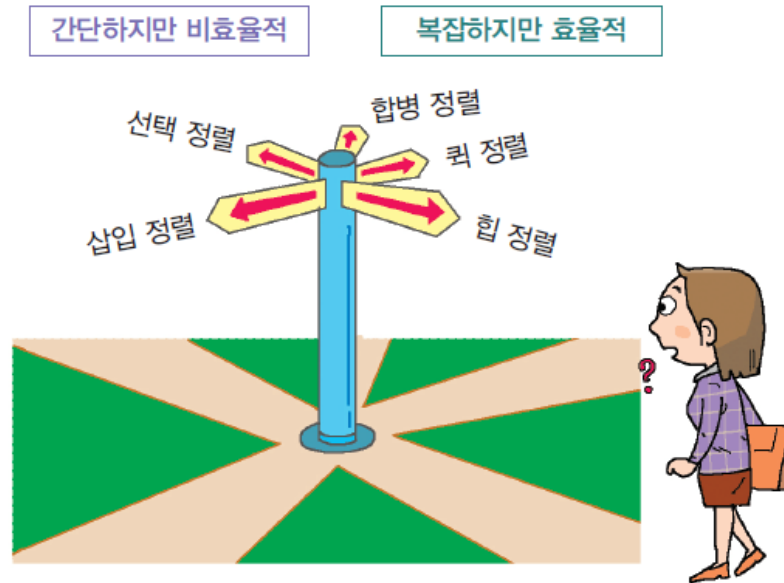
- 모든 경우에 최적인 정렬 알고리즘은 없음
- 각 응용 분야에 적합한 정렬 방법 사용해야 함
  - ▣ 레코드 수의 많고 적음
  - ▣ 레코드 크기의 크고 작음
  - ▣ **Key**의 특성(문자, 정수, 실수 등)
  - ▣ 메모리 내부/외부 정렬
- 정렬 알고리즘의 평가 기준
  - ▣ 비교 횟수의 많고 적음
  - ▣ 이동 횟수의 많고 적음





# 정렬 알고리즘 개요

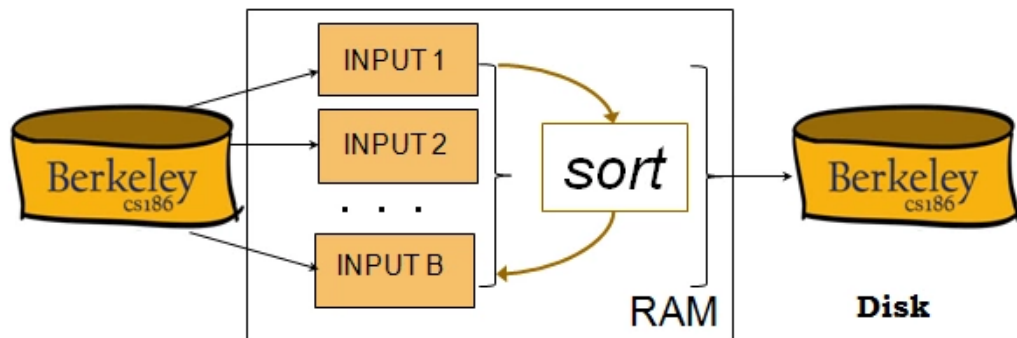
- 단순하지만 비효율적인 방법
  - ▣ 삽입정렬, 선택정렬, 버블정렬 등
- 복잡하지만 효율적인 방법
  - ▣ 퀵정렬, 힙정렬, 합병정렬, 기수정렬 등





# 정렬 알고리즘 개요

- 내부 정렬(internal sorting)
  - ▣ 모든 데이터가 주기억장치에 저장되어진 상태에서 정렬
- 외부 정렬(external sorting)
  - ▣ 외부기억장치에 대부분의 데이터가 있고 일부만 주기억장치에 저장된 상태에서 정렬



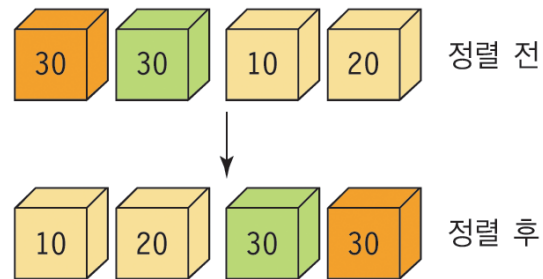
이미지 출처: [Cs186 Wiki - FANDOM](#)





# 정렬 알고리즘 개요

- 정렬 알고리즘의 안정성(stability)
  - ▣ 동일한 키 값을 갖는 레코드들의 상대적인 위치가 정렬 후에도 바뀌지 않음
  - ▣ 안정하지 않은 정렬의 예 =====>





# 선택정렬(selection sort)

- 정렬된 왼쪽 리스트와 정렬안된 오른쪽 리스트 가정
  - ▣ 초기에는 왼쪽 리스트는 비어 있고, 정렬할 숫자들은 모두 오른쪽 리스트에 존재

왼쪽 리스트	오른쪽 리스트	설명
( )	(5,3,8,1,2,7)	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







# 제자리 정렬





```
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++;
```





# 선택정렬 코드

```
#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++)           // 최솟값 탐색
            if(list[j]<list[least]) least = j;
        SWAP(list[i], list[least], temp);
    }
}
```





# 선택정렬 코드

```
//
int main(void)
{
    int i;
    n = MAX_SIZE;
    srand(time(NULL));
    for (i = 0; i<n; i++)           // 난수 생성 및 출력
        list[i] = rand() % 100; // 난수 발생 범위 0~99

    selection_sort(list, n); // 선택정렬 호출
    for (i = 0; i<n; i++)
        printf("%d ", list[i]);
    printf("\n");
    return 0;
}
```

16 24 25 38 48 64 87 90 93 96





# 선택정렬 복잡도 분석

- 비교 횟수
  - ▣  $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$
- 이동 횟수
  - ▣  $3(n - 1)$
- 전체 시간적 복잡도:  $O(n^2)$
- 안정성을 만족하지 않음





# 삽입정렬(insertion sort)

- 정렬되어 있는 부분에 새로운 레코드를 올린 위치에 삽입하는 과정 반복





# 삽입정렬(insertion sort)

## □ 삽입정렬 과정





# 사이저력 알고리즘

insertion\_sort(A, n)

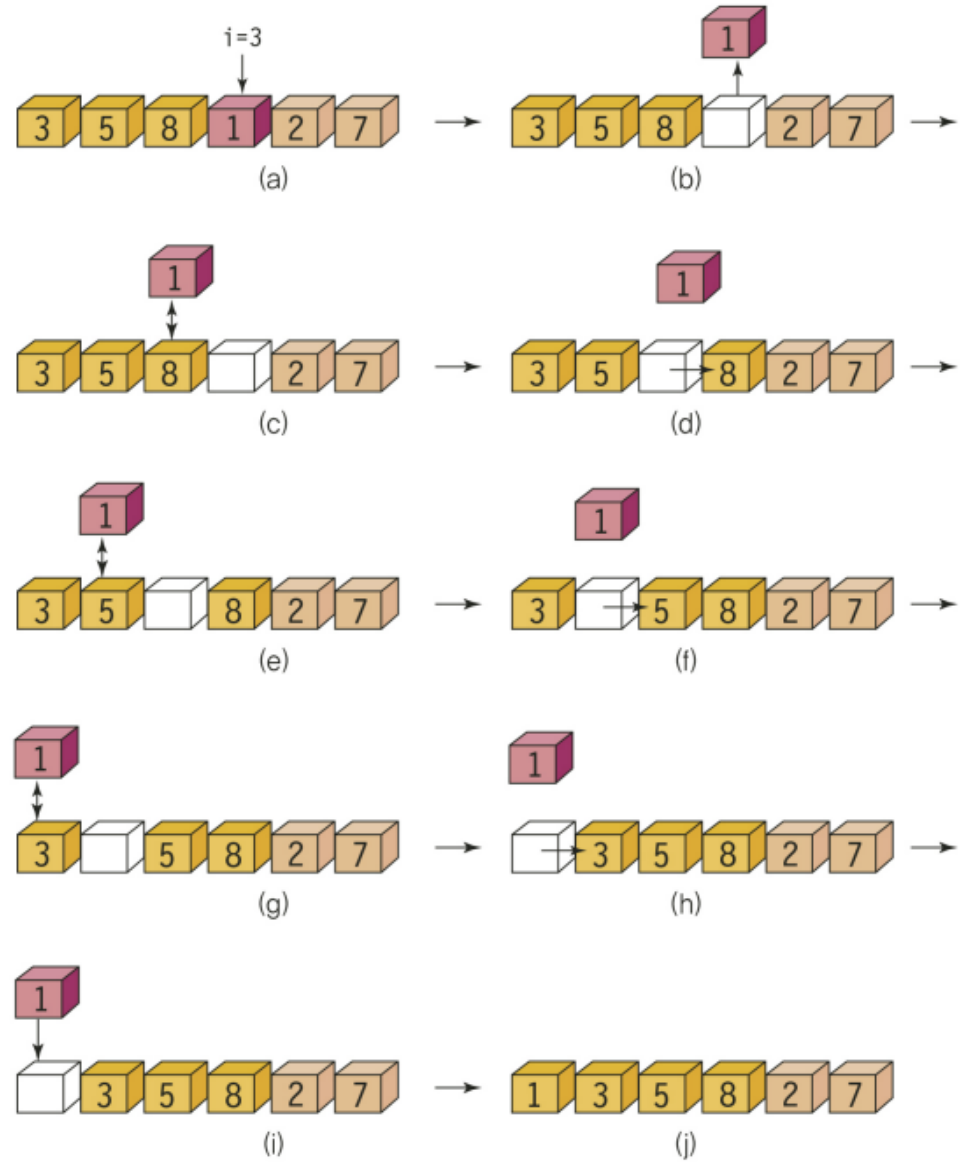
```
1. for i ← 1 to n-1 do
2.     key ← A[i];
3.     j ← i-1;
4.     while j ≥ 0 and A[j] > key do
5.         A[j+1] ← A[j];
6.         j ← j-1;
7.     A[j+1] ← key
```







# 삽입정렬(insertion sort)





# 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;  
    }  
}
```





# 삽입정렬 복잡도 분석

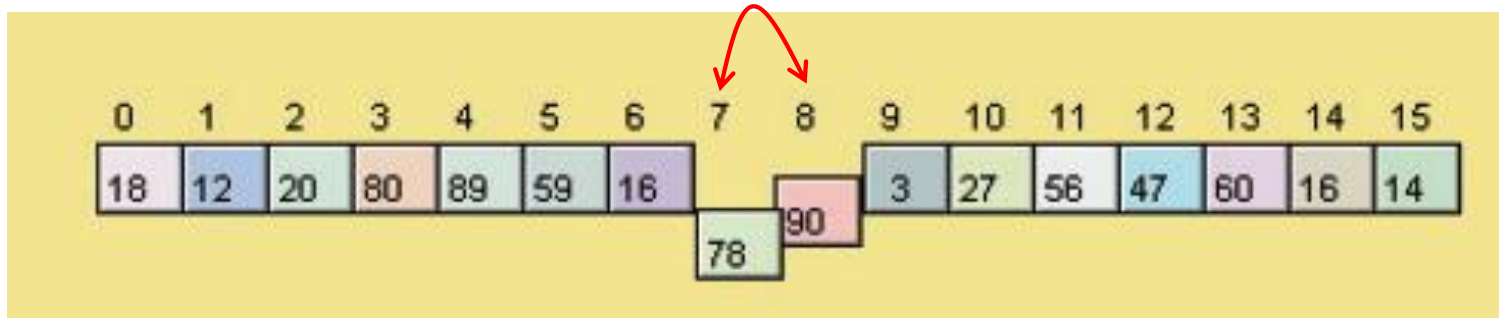
- 최선의 경우  $O(n)$  : 이미 정렬되어 있는 경우
  - 비교:  $n-1$  번
- 최악의 경우  $O(n^2)$  : 역순으로 정렬되어 있는 경우
  - 모든 단계에서 앞에 놓인 자료 전부 이동
  - 비교: 
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$
  - 이동: 
$$\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$$
- 평균의 경우  $O(n^2)$
- 많은 이동 필요 -> 레코드가 클 경우 불리
- 안정된 정렬방법
- 대부분 정렬되어 있으면 매우 효율적





# 버블정렬(bubble sort)

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





# 버블정렬(bubble sort)





# 버블정렬 알고리즘

BubbleSort(A, n)

for  $i \leftarrow n-1$  to 1 do

  for  $j \leftarrow 0$  to  $i-1$  do

$j$ 와  $j+1$ 번째의 요소가 크기 순이 아니면 교환

$j++$ ;

$i--$ ;





# 버블정렬 코드

```
#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 분석

- 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 :  $O(n^2)$

- 레코드의 이동 과다

- ▣ 이동연산은 비교연산 보다 더 많은 시간이 소요됨







# 셸 정렬(Shell sort)

- 삽입정렬이 어느 정도 정렬된 리스트에서 대단히 빠른 것에 착안
  - ▣ 삽입 정렬은 요소들이 이웃한 위치로만 이동하므로, 많은 이동에 의해서만 요소가 제자리를 찾아가감
  - ▣ 요소들이 멀리 떨어진 위치로 이동할 수 있게 하면, 보다 적게 이동하여 제자리 찾을 수 있음
  - ▣
- 전체 리스트를 일정 간격(gap)의 부분 리스트로 나눔
  - ▣ 나뉘어진 각각의 부분 리스트를 삽입정렬 함





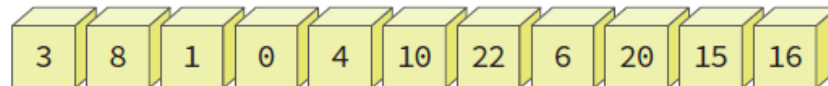
# 셸 정렬(Shell sort)



10	-	-	-	-	3	-	-	-	-	16
	8	-	-	-	-	22				
		6	-	-	-	-	1			
			20	-	-	-	-	0		
				4	-	-	-	-	15	

(a) 간격 5로 만들어진 부분 리스트

3	-	-	-	-	10	-	-	-	-	16
	8	-	-	-	-	22				
		1	-	-	-	-	6			
			0	-	-	-	-	20		
				4	-	-	-	-	15	



(b) 간격 5로 만들어진 부분 리스트





# 셸 정렬(Shell sort)

입력 배열	10	8	6	20	4	3	22	1	0	15	16
간격 5일 때의 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 정렬후의 전체 배열	3	8	1	0	4	10	22	6	20	15	16
간격 3일 때의 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





# 셸 정렬 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);
    }
}
```





# 셀 정렬 복잡도 분석

## □ 셀 정렬의 장점

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

## □ 시간적 복잡도

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





# 합병 정렬(merge sort)

1. 리스트를 두 개의 균등한 크기로 분할하고 분할된 부분 리스트를 정렬
2. 정렬된 두 개의 부분 리스트를 합하여 전체 리스트를 정렬함

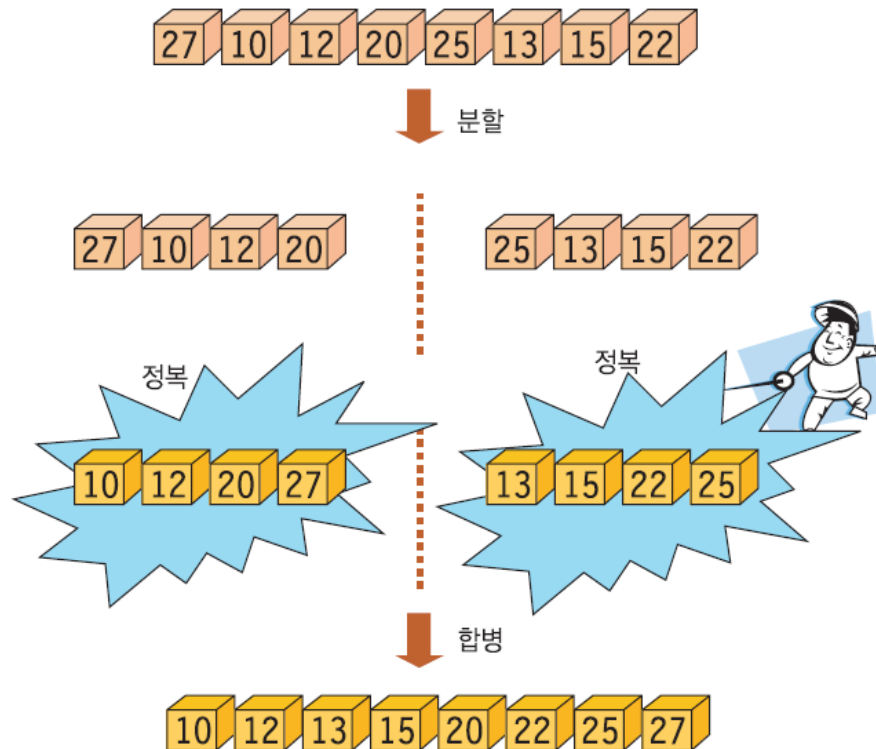


1. 분할(Divide) : 배열을 같은 크기의 2개의 부분 배열로 분할
2. 정복(Conquer): 부분배열을 정렬한다. 부분배열의 크기가 충분히 작지 않으면 재귀호출을 이용하여 다시 분할정복기법 적용
3. 결합(Combine): 정렬된 부분배열을 하나의 배열에 통합



입력파일: (27 10 12 20 25 13 15 22)

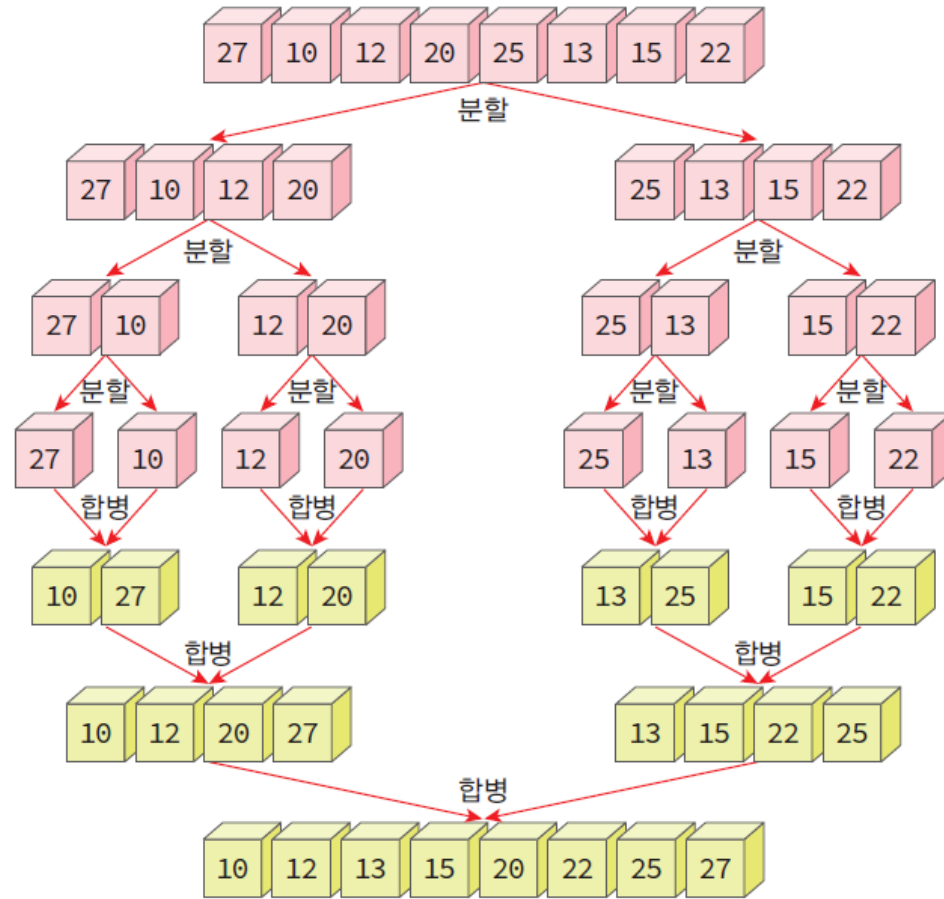
- 1.분할(Divide) : 전체 배열을 (27 10 12 20), (25 13 15 22) 2개 부분배열로 분리
- 2.정복(Conquer): 각 부분배열 정렬 (10 12 20 27), (13 15 22 25)
- 3.결합(Combine): 2개의 정렬된 부분배열 통합 (10 12 13 15 20 22 25 27)







# 합병정렬의 전체 과정





# 하버정려 알고리즘

```
merge_sort(list, left, right)
```

```
1 if left < right
```

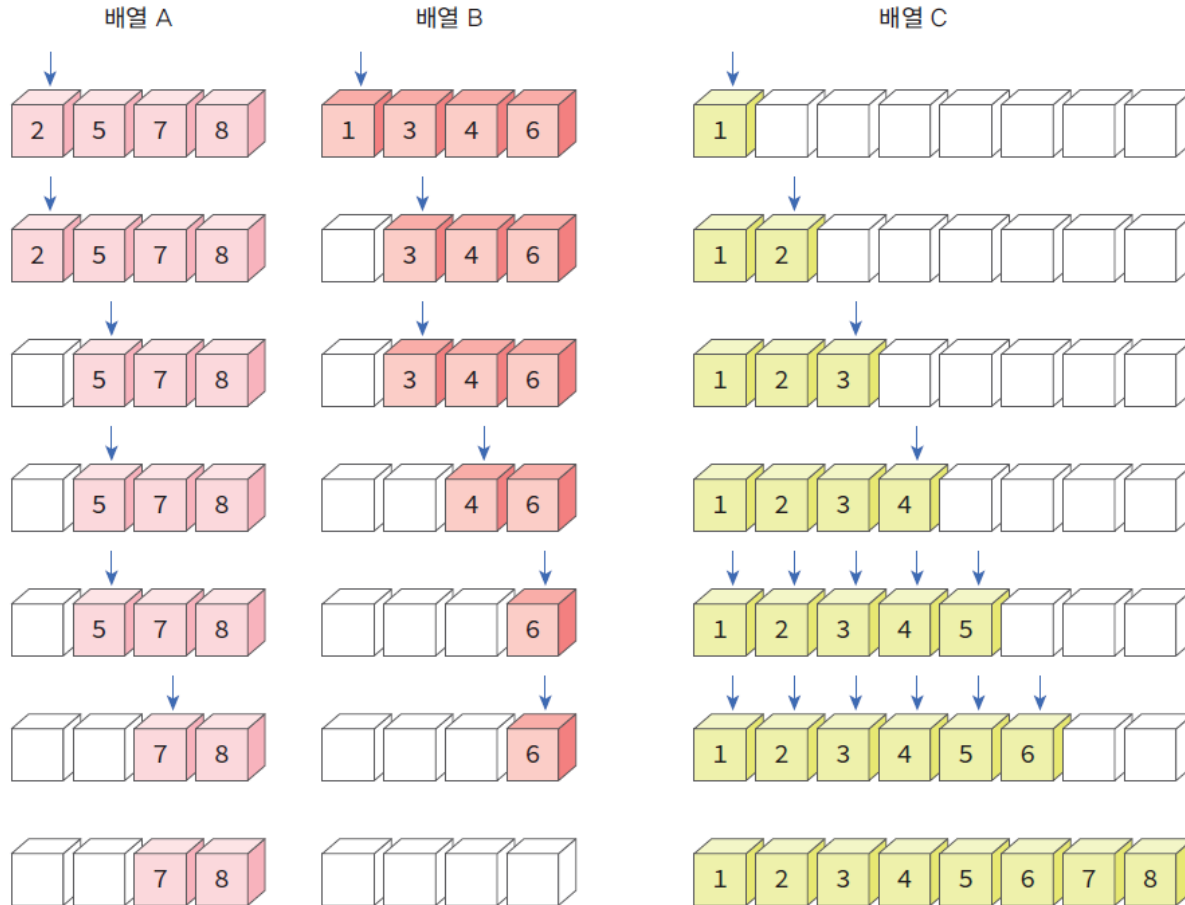
```
2     mid = (left+right)/2;
```

```
3     merge_sort(list, left, mid);
```

```
4     merge_sort(list, mid+1, right);
```

```
5     merge(list, left, mid, right);
```







# 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;
```

```
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++;
```

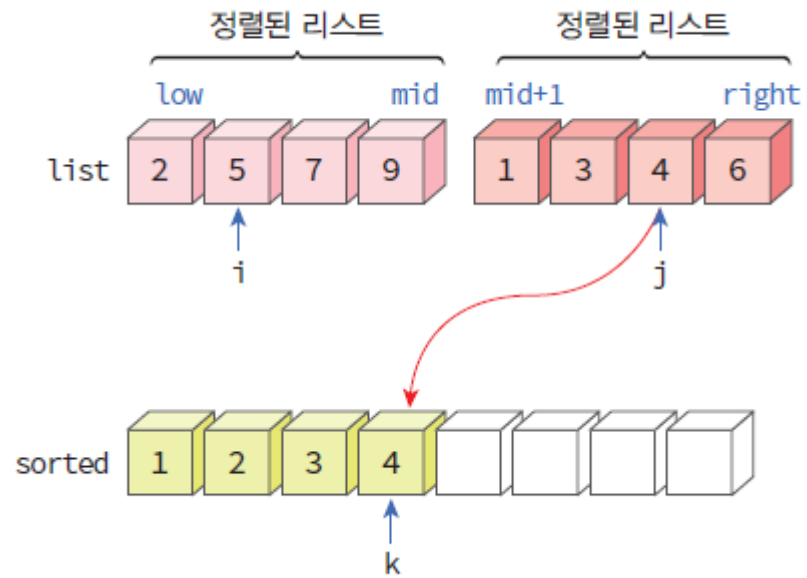
```
요소가 남아있는 부분배열을 sorted로 복사한다;
```

```
sorted를 list로 복사한다;
```





# 합병의 중간 상태



```
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];
}
```



```
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 복잡도 분석

## □ 비교 횟수

- ▣ 크기  $n$ 인 리스트를 정확히 균등 분배하므로  $\log(n)$  개의 패스
- ▣ 각 패스에서 리스트의 모든 레코드  $n$ 개를 비교하므로  $n$ 번의 비교 연산

## □ 이동 횟수

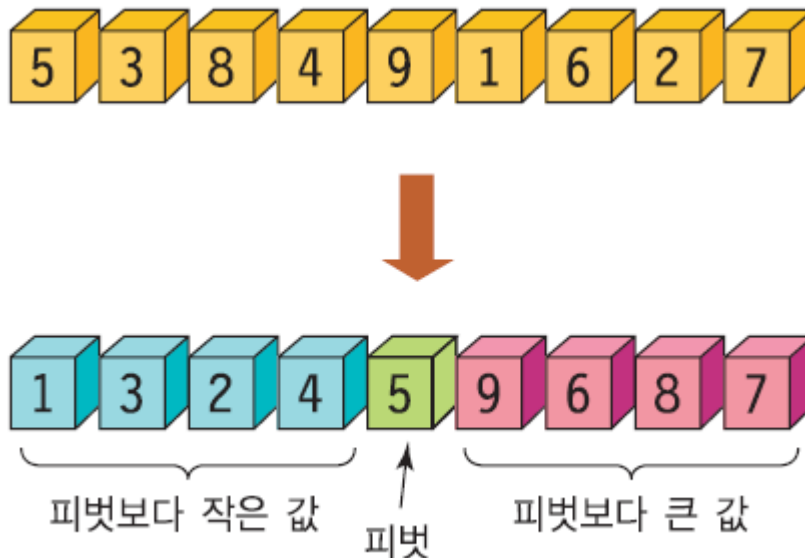
- ▣ 레코드의 이동이 각 패스에서  $2n$ 번 발생하므로 전체 레코드의 이동은  $2n \cdot \log(n)$ 번 발생
- ▣ 레코드의 크기가 큰 경우에는 매우 큰 시간적 낭비 초래
- ▣ 레코드를 연결 리스트로 구성하여 합병 정렬할 경우, 매우 효율적
- 최적, 평균, 최악의 경우 큰 차이 없이  $O(n \cdot \log(n))$ 의 복잡도
- 안정적이며 데이터의 초기 분산 순서에 영향을 덜 받음





# 퀵정렬(quick sort)

- 평균적으로 가장 빠른 정렬 방법
- 분할정복법 사용
- 리스트를 2개의 부분리스트로 비균등 분할하고, 각각의 부분리스트를 다시 퀵정렬함(재귀호출)

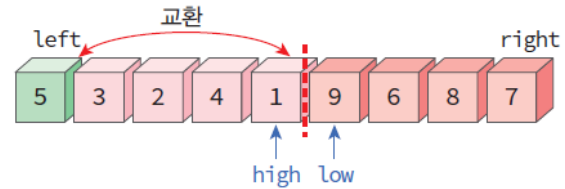
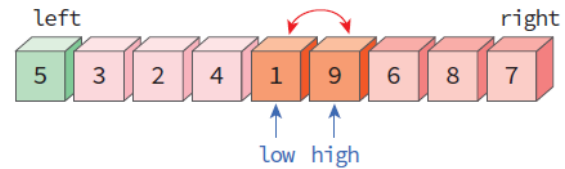
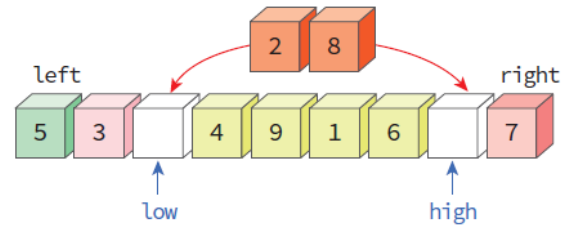
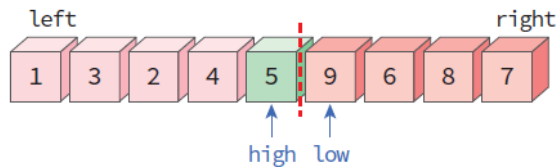
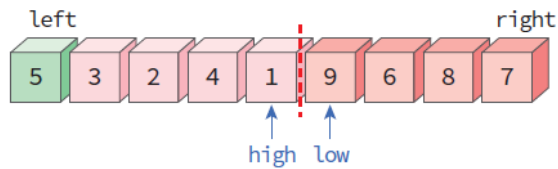
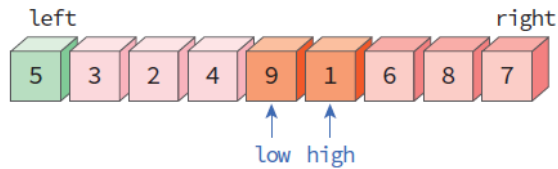
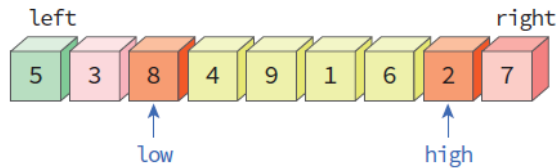
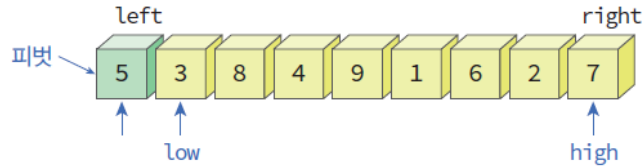




# 퀵 정렬 알고리즘

```
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. }
```







# partition 함수

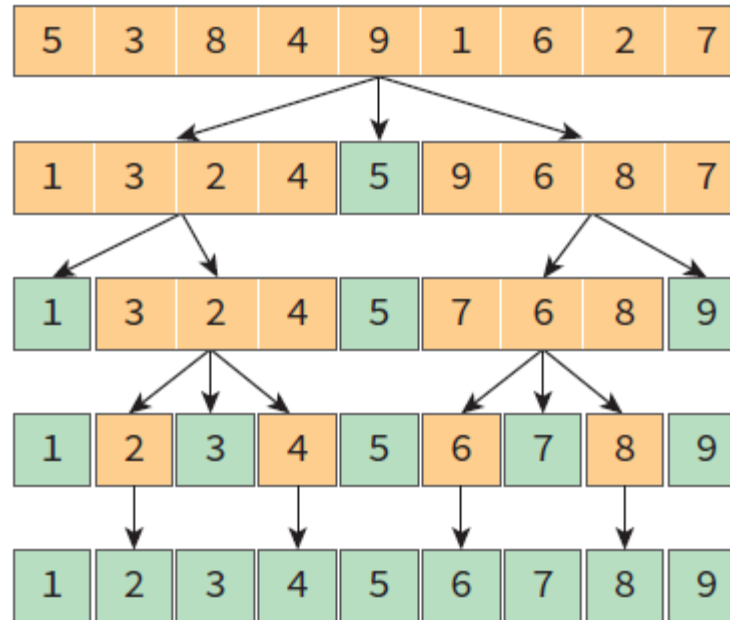
```
int partition(int list[], int left, int right)
{
    int pivot, temp;
    int low, high;

    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;
}
```



# 전체 과정





하△

```
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);
    }
}
//
int main(void)
{
    int i;
    n = MAX_SIZE;
    srand(time(NULL));
    for (i = 0; i < n; i++)          // 난수 생성 및 출력
        list[i] = rand() % 100;

    quick_sort(list, 0, n-1); // 퀵정렬 호출
    for (i = 0; i < n; i++)
        printf("%d ", list[i]);
    printf("\n");
    return 0;
}
```



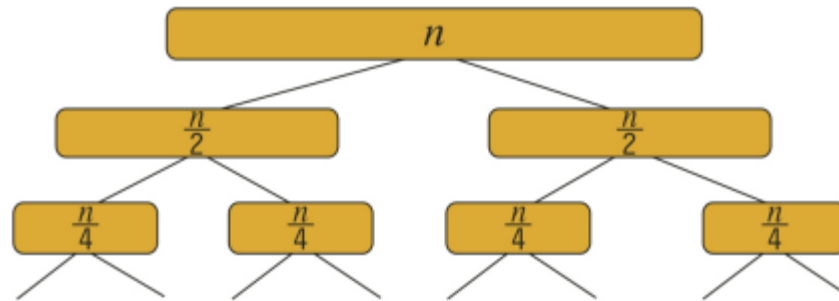


# 퀵 정렬 복잡도 분석

## □ 최선의 경우(거의 균등한 리스트로 분할되는 경우)

### ▣ 패스 수: $\log(n)$

- 2->1
- 4->2
- 8->3
- ...
- $n \rightarrow \log(n)$



### ▣ 각 패스 안에서의 비교횟수: $n$

### ▣ 총 비교횟수: $n * \log(n)$

### ▣ 총 이동횟수: 비교횟수에 비하여 적으므로 무시 가능







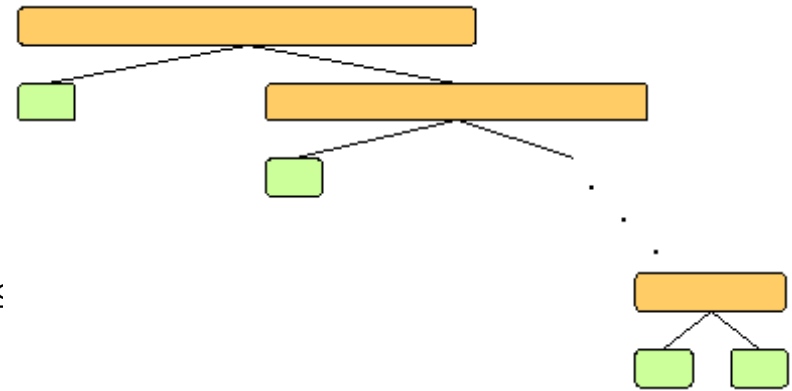
# 퀵정렬 복잡도 분석(cont.)

## □ 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



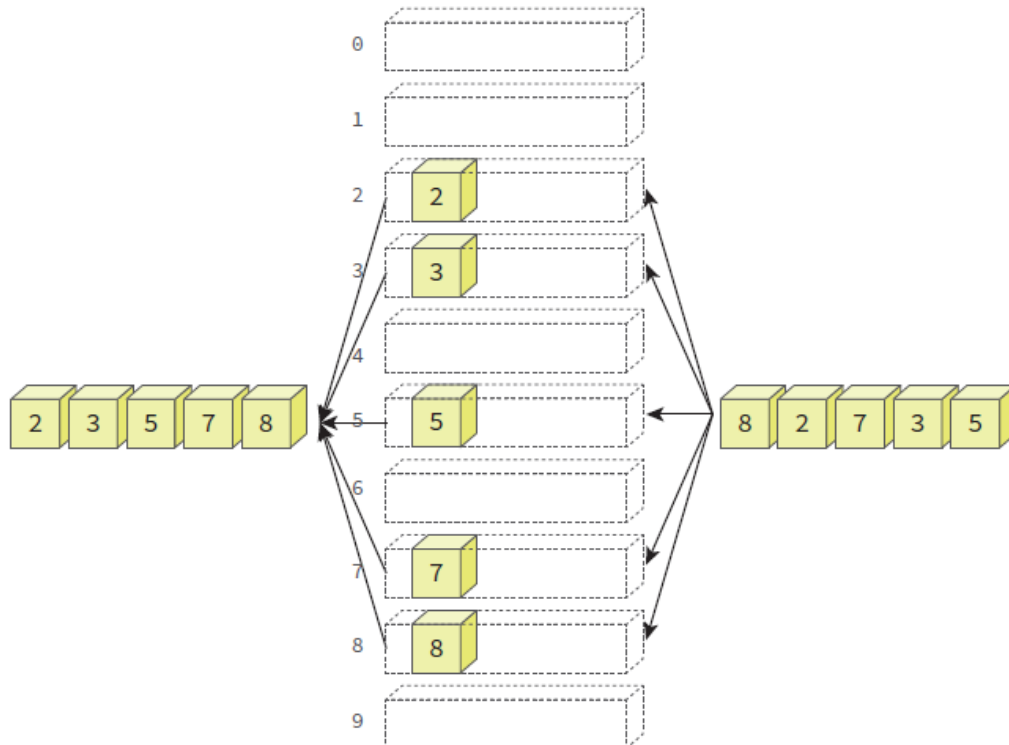


# 기수정렬(Radix Sort)

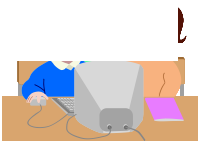
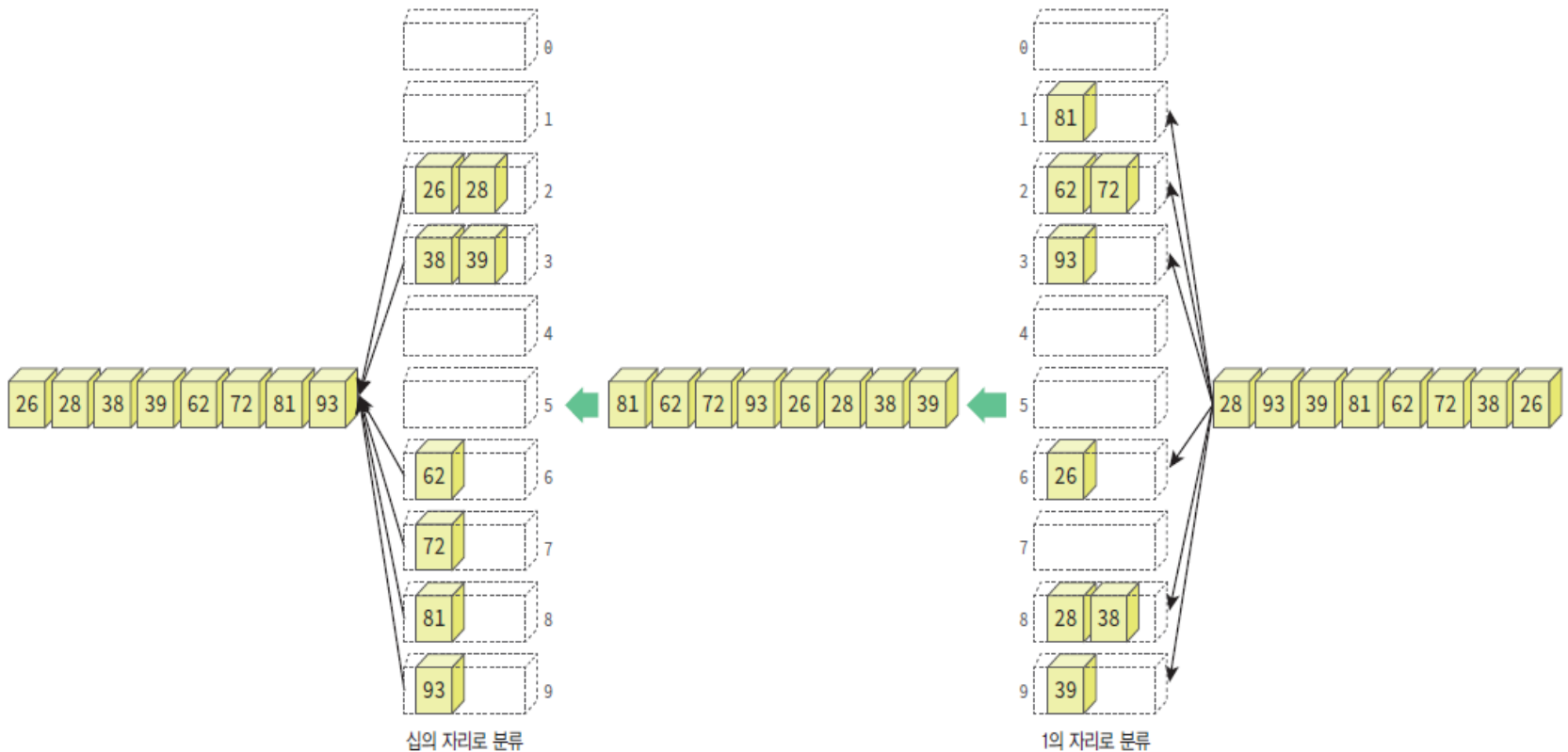
- 대부분의 정렬 방법들은 레코드들을 비교함으로써 정렬 수행
- 기수 정렬(**radix sort**)은 레코드를 비교하지 않고 정렬 수행
  - ▣ 비교에 의한 정렬의 하한인  $O(n \cdot \log(n))$  보다 좋을 수 있음
  - ▣ 기수 정렬은  $O(dn)$  의 시간적복잡도를 가짐(대부분  $d < 10$  이하)



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



- 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 버킷에 넣는다.
    버킷에서 숫자들을 순차적으로 읽어서 하나의 리스트로 합친다.
    d++;
}
```





# 기수정렬 알고리즘

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





# 기수정렬 프로그램

```
// 6장의 큐 소스를 여기에...
#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++) // 버킷에서 꺼내어 list로 합친다.
            while( !is_empty(&queues[b]) )
                list[i++] = dequeue(&queues[b]);
        factor *= 10; // 그 다음 자리수로 간다.
    }
}
```





# 기수정렬 프로그램

```
#define SIZE 10

int main(void)
{
    int list[SIZE];
    srand(time(NULL));
    for (int i = 0; i < SIZE; i++)          // 난수 생성 및 출력
        list[i] = rand() % 100;

    radix_sort(list, SIZE); // 기수정렬 호출
    for (int i = 0; i < SIZE; i++)
        printf("%d ", list[i]);
    printf("\n");
    return 0;
}
```







# 기수정렬 복잡도 분석

- $n$ 개의 레코드,  $d$ 개의 자릿수로 이루어진 키를 기수 정렬할 경우
  - ▣ 메인 루프는 자릿수  $d$ 번 반복
  - ▣ 큐에  $n$ 개 레코드 입력 수행
- $O(dn)$ 의 시간적 복잡도
  - ▣ 키의 자릿수  $d$ 는 10 이하의 작은 수이므로 빠른 정렬임
- 실수, 한글, 한자로 이루어진 키는 정렬 못함





# 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	$O(n^2)$	$O(n^2)$	$O(n^2)$
셸 정렬	$O(n)$	$O(n^{1.5})$	$O(n^{1.5})$
퀵 정렬	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
힙 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 알고리즘의 실험 예 (정수 60,000개)

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

