DATA
STRUCTURES
USING C

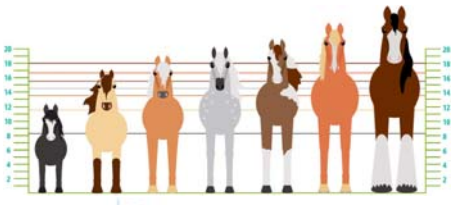
- 정렬의 개념을 이해한다.
- 각 정렬 알고리즘의 동작 원리를 이해한다.
- 각 정렬 알고리즘의 장점과 단점을 이해한다.
- 각 정렬 알고리즘의 효율성을 이해한다.
- 각 정렬 알고리즘의 구현 방법을 이해한다.

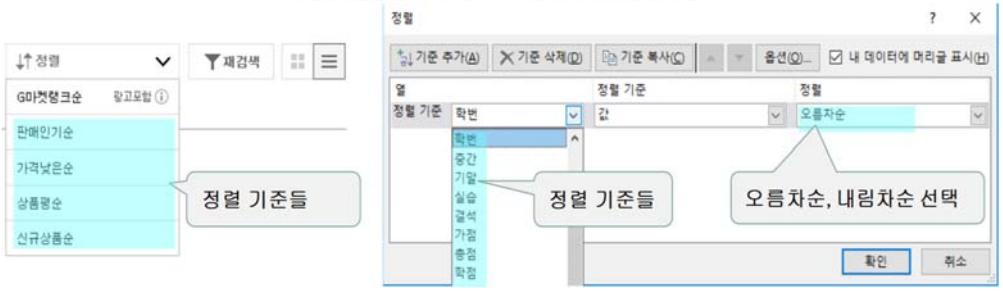
CHAPTER 13

정렬

13.1 정렬이란?

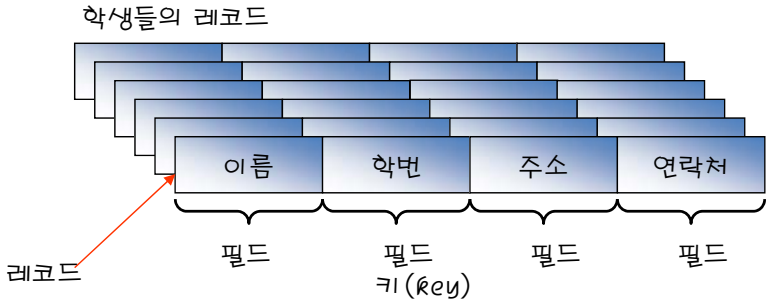
- 물건을 크기순으로 오름/내림차순으로 나열하는 것
  - 가장 기본적이고 중요한 알고리즘
  - (예) 사전에서 단어들이 알파벳 순으로 정렬되어 있지 않다면?





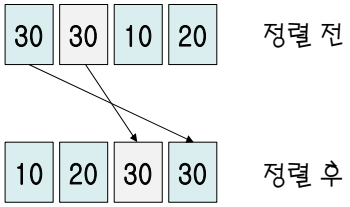
정렬의 대상

- 일반적으로 정렬시켜야 될 대상은 레코드(record)
  - 레코드는 다시 필드(field)라는 보다 작은 단위로 구성됨



정렬 알고리즘 종류

- 모든 경우에 대해 최적인 정렬 알고리즘은 없음
  ➔ 해당 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
- 단순하지만 비효율적인 방법 : 삽입, 선택, 버블정렬 등
- 복잡하지만 효율적인 방법: 퀵, 힙, 병합, 기수정렬 등
- 정렬 알고리즘의 안정성(stability): 삽입, 버블, 병합

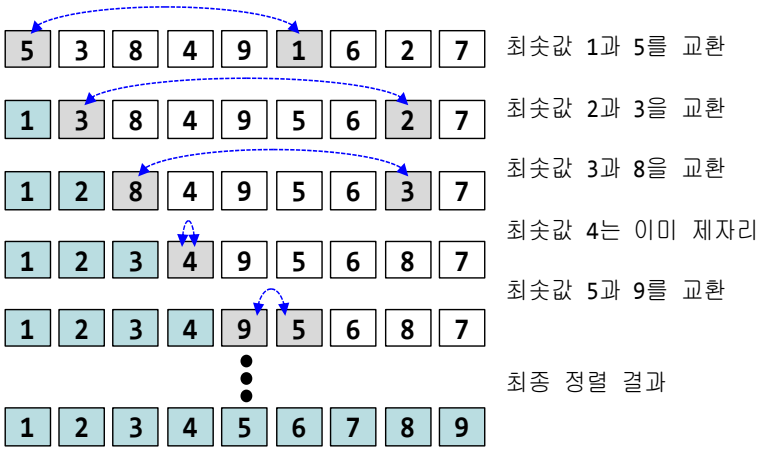


13.2 선택정렬(selection sort)

- 정렬된 왼쪽 리스트와 정렬 안된 오른쪽 리스트를 가짐

왼쪽 리스트 (정렬된 리스트)	오른쪽 리스트 (정렬 안 된 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

선택정렬 알고리즘




```
selectionSort(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++;
```

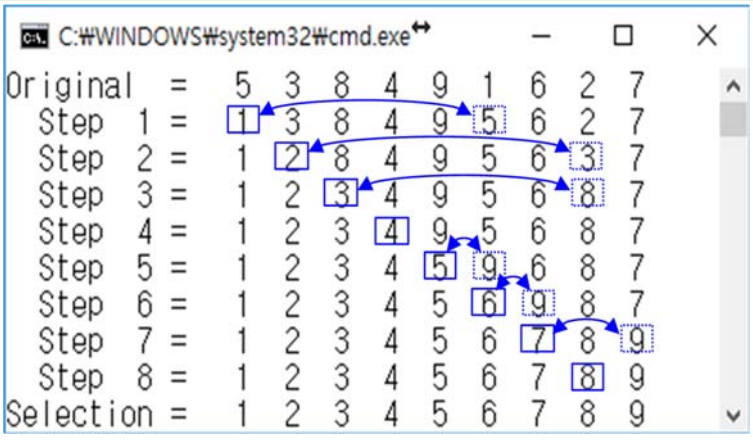
선택정렬 구현

- 구현 코드: int 배열 오름차순 정렬 함수

```
#define SWAP(x,y,t) ((t)=(x),(x)=(y),(y)=(t))

void selection_sort (int list[], int n)
{
  int i, j, least,tmp;
  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],tmp);
    printStep(list, n, i+1);
  }
}
```


전체 프로그램

```
void main()
{
  int    list[9] = { 5, 3, 8, 4, 9, 1, 6, 2, 7 };
  printArray( list, 9, "Original " );      // 정렬 전 배열 출력
  selection_sort( list, 9 );               // 선택 정렬 실행
  printArray( list, 9, "Selection" );      // 정렬 후 배열 출력
}
```



선택정렬 복잡도 분석

- 비교 횟수
  - $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 = O(n^2)$
- 이동 횟수
  - $3(n - 1)$
- 전체 시간 복잡도
  - $O(n^2)$
- 특징
  - 알고리즘이 매우 간단함
  - 효율적인 알고리즘은 아니며 안정성을 만족하지 않음

- 다음 자료에 대하여 “선택 정렬 " 을 사용하여 오름차순으로 정렬할 경우 PASS 3의 결과는?
  - 초기상태: 8, 3, 4, 9, 7

- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

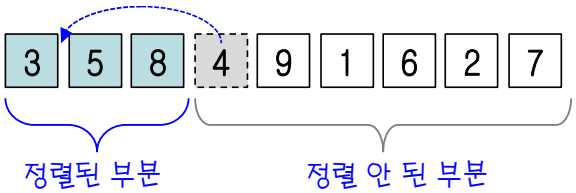
7	4	9	6	3	8	7	5

1. 선택 정렬

13.3 삽입정렬(insertion sort)

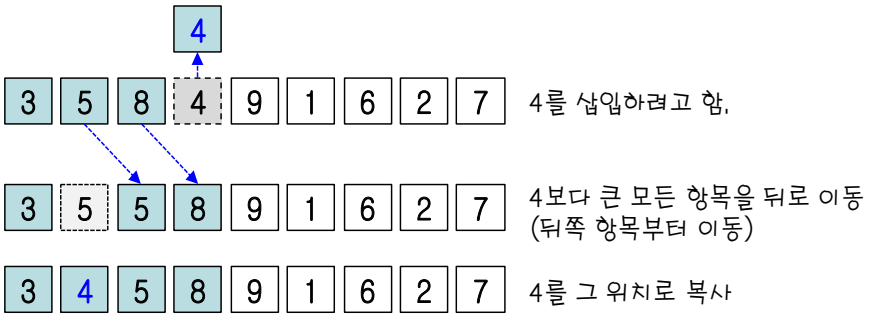
- 정렬되어 있는 부분에 새로운 레코드를 올바른 위치에 삽입하는 과정 반복

- 정렬된 부분과 안된 부분



한번의 삽입과정

- 항목들의 이동이 필요함



삽입정렬 알고리즘

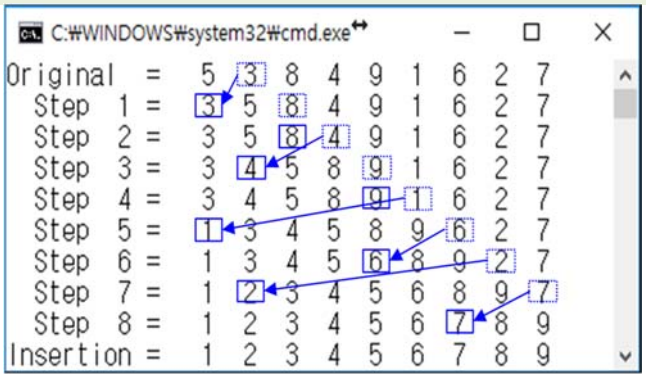
- 삽입정렬 과정 예

- 알고리즘 유사코드

```
insertionSort(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 구현

```
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;
                printStep(list, n, i);            // 중간 과정 출력용 문장
        }
}
```



삽입정렬 복잡도

- 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 $O(n^2)$
- 특징
  - 많은 이동 필요 → 레코드가 큰 경우 불리
  - 안정된 정렬방법
  - 대부분 정렬되어 있으면 매우 효율적

- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

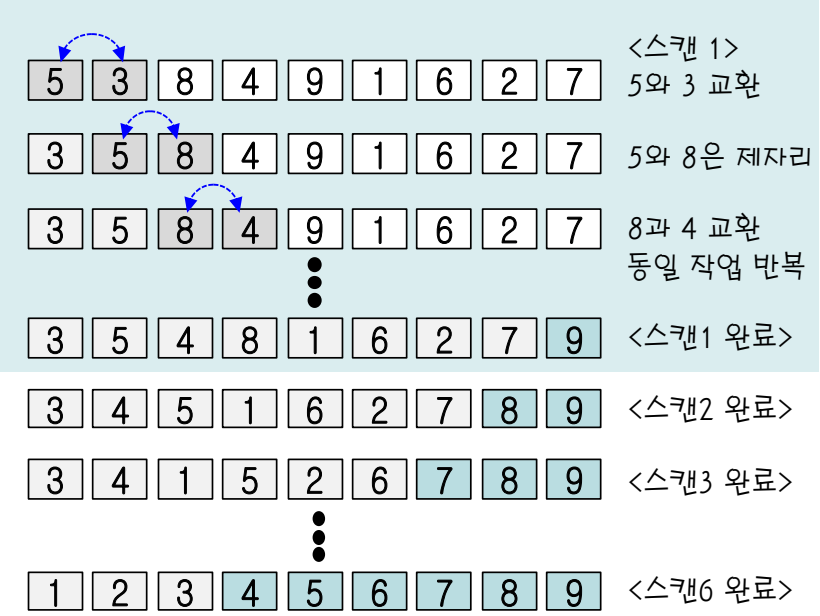
7	4	9	6	3	8	7	5

1. 삽입 정렬

13.4 버블정렬 (bubble sort)

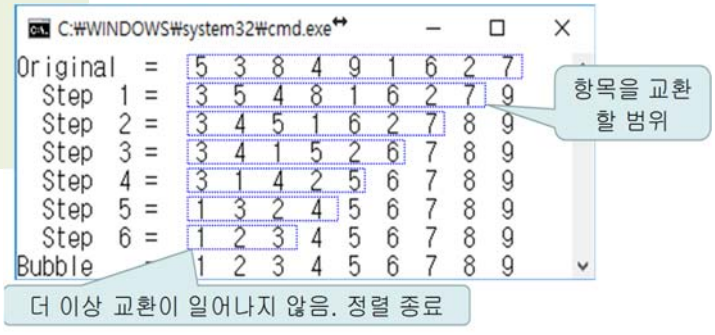
- 기본 전략
  - 인접한 2개의 레코드를 비교하여 순서대로 되어 있지 않으면 서로 교환
  - 이러한 비교-교환 과정을 리스트의 왼쪽 끝에서 오른쪽 끝까지 반복(스캔)
    - 한번의 스캔이 완료되면 리스트의 오른쪽 끝에 가장 큰 레코드가 이동함
  - 끝으로 이동한 레코드를 제외한 왼쪽 리스트에 대하여 스캔 과정을 반복함
    - 더 이상 교환이 일어나지 않을 때 까지 수행

버블정렬 진행과정



버블정렬 알고리즘

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



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(최악): 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

- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

7	4	9	6	3	8	7	5

1. 버블 정렬

13.5 함수 포인터를 사용한 정렬

- 오름차순/내림차순? → 비교 함수

```
int ascend (int x, int y) { return y - x; }
int descend(int x, int y) { return x - y; }
```

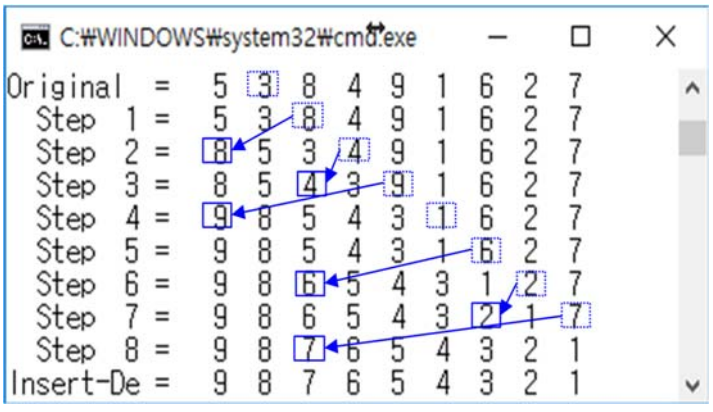
- 정렬 함수의 매개변수? → 함수 포인터

```
void insertion_sort_fn (int list[], int n, int (*f)(int,int))
{
        int i, j, key;
        for(i=1; i<n; i++){
                key = list[i];
                for(j=i-1 ; j>=0 && f(list[j],key) < 0 ;j--)
                        list[j+1] = list[j];
                        list[j+1] = key;
        }
}
```

- 함수 호출 방법? → 함수 이름을 전달

```
insertion_sort_fn( list, n, descend ); // 내림차순 삽입정렬 알고리즘
```


내림차순 삽입정렬 실행 결과



- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

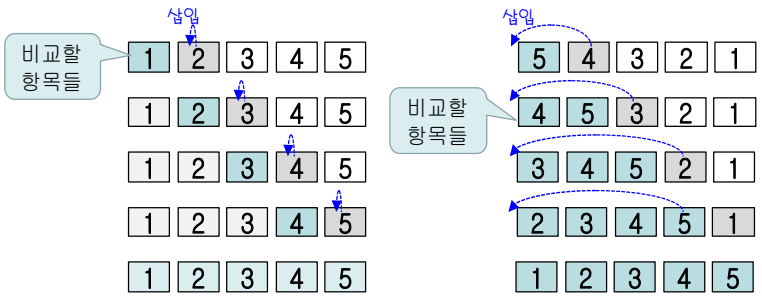
7	4	9	6	3	8	7	5

1. 삽입 정렬

13.6 셸정렬 (Shell sort)

- 기본 아이디어
  - 삽입정렬은 어느 정도 정렬된 리스트에서 대단히 빠르다!
  - 그러나 요소들이 이웃한 위치로만 이동➔많은 이동 발생
  - 요소들이 멀리 떨어진 위치로 이동할 수 있게 한다면

➔ 보다 적게 이동하여 제자리 찾을 수 있음



(a) 정렬된 배열의 삽입 정렬 (b)역순 정렬 배열의 삽입 정렬

셸정렬

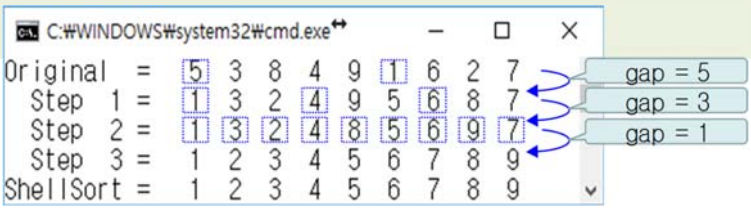
- 리스트를 일정 간격(gap)의 부분 리스트로 나눔
  - 나뉘어진 각각의 부분 리스트를 삽입정렬 함
- 간격을 줄임
  - 부분 리스트의 수는 더 작아지고, 각 부분 리스트는 더 커짐
- 간격이 1이 될 때까지 이 과정 반복

- 간격 k=5의 부분 리스트 정렬 전: {5,1} {3,6}, {8,2}, {9}
- 간격 k=5의 부분 리스트 정렬 후: {1,5} {3,6}, {2,8}, {9}
- Step1 완료: 1, 3, 2, 4, 9, 5, 6, 8, 7
- 간격 k=3의 부분 리스트 정렬 전: {1,4,6} {3,9,8}, {2,5,7}
- 간격 k=3의 부분 리스트 정렬 후: {1,4,6} {3,8,9}, {2,5,7}
- Step2 완료: 1, 3, 2, 4, 8, 5, 6, 9, 7
- 간격 k=1의 부분 리스트 정렬 전: {1,3,2,4,8,5,6,9,7}
- 간격 k=1의 부분 리스트 정렬 후: {1,2,3,4,5,6,7,8,9}
- Step3 완료: 1, 2, 3, 4, 5, 6, 7, 8, 9

입력 배열	5	3	8	4	9	1	6	2	7
간격 5일 때의 부분 리스트	5					1			
		3					6		
			8					2	
				4					7
					9				
Step1 완료	**1**	**3**	**2**	**4**	**9**	**5**	**6**	**8**	**7**
간격 3일 때의 부분 리스트	1			4			6		
		3			9			8	
			2			5			7
Step2 완료	**1**	**3**	**2**	**4**	**8**	**5**	**6**	**9**	**7**
간격 1일 때의 부분 리스트	1	3	2	4	8	5	6	9	7
간격 1 정렬: 1번 비교(1)	1	**3**							
2번 비교(3,1)	1	3	**2**						
1번 비교(3)			3	**4**					
1번 비교(4)				4	**8**				
2번 비교(8,4)				4	8	**5**			
2번 비교(8,5)					5	8	**6**		
1번 비교(8)							8	**9**	
3번 비교(9,8,7)						6	8	9	**7**
Step3 완료	**1**	**2**	**3**	**4**	**5**	**6**	**8**	**7**	**9**

셸정렬 구현

```
static void sortGapInsertion (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, count=0;
    for( gap=n/2; gap>0; gap = gap/2 ) {
        if( (gap%2) == 0 ) gap++;
        for( i=0 ; i<gap ; i++ )
            sortGapInsertion( list, i, n-1, gap );
    }
}
```



셸정렬 복잡도 분석

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

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

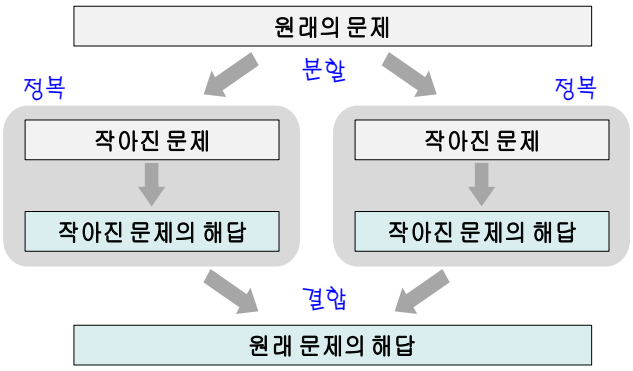
- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

7	4	9	6	3	8	7	5

1. 쉘 정렬

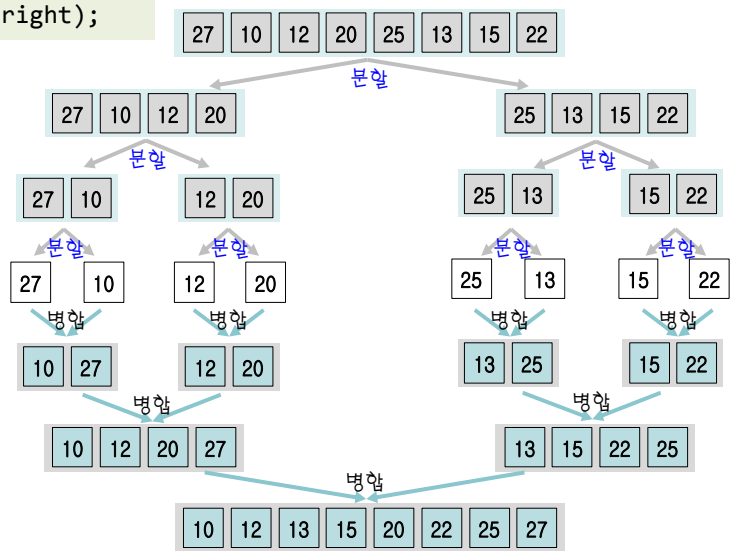
13.7 병합정렬 (Merge Sort)

- 리스트를 두 개의 부분 리스트로 분할하고 각각을 정렬
- 정렬된 부분 리스트를 합해 전체 리스트를 정렬
- 분할 정복(divide and conquer) 방법
  - 문제를 보다 작은 2개의 문제로 분리하고 각 문제를 해결한 다음, 결과를 모아서 원래의 문제를 해결하는 전략

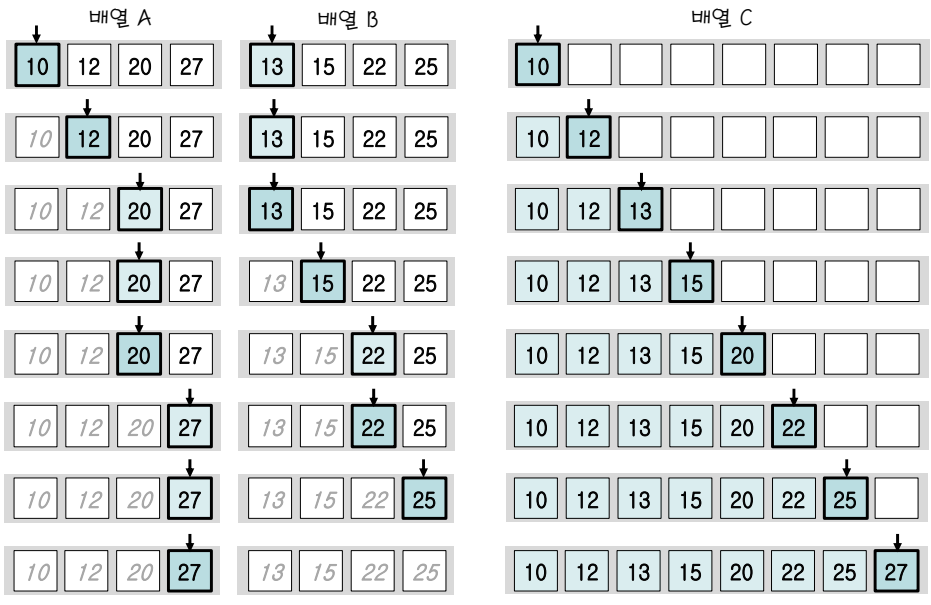


병합정렬 알고리즘

```
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);
```



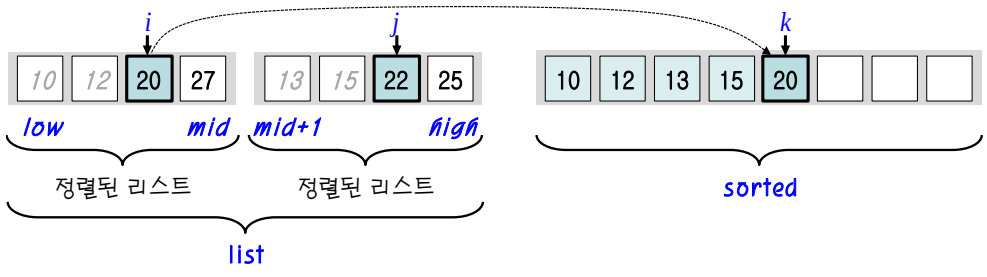
병합 과정



병합 알고리즘

```
merge(list, left, mid, last)
// 2개의 인접한 배열 list[left~mid]와 list[mid+1~right]를 병합
i←left;
e1←mid;
j←mid+1;
e2←right;
sorted 배열을 생성;
k←0;
while i≤e1 and j≤e2 do
  if(list[i]<list[j])
  then
    sorted[k++]←list[i++];
  else
    sorted[k++]←list[j++];
요소가 남아있는 부분배열을 sorted로 복사한다;
sorted를 list로 복사한다;
```


병합의 중간 상태



병합 정렬 프로그램

```
static void merge(int list[], int left, int mid, int right) {
    int i, j, k = left, l;
    static int sorted[MAX_SIZE];
    for( i=left, j=mid+1 ; i<=mid && j<=right ; )
        sorted[k++] = (list[i]<=list[j]) ? list[i++] : list[j++];
        if( i > mid )  // 남은 부분 복사
            for( l=j ; l<=right ; l++, k++ )
                sorted[k] = list[l];
        else
            for( l=i ; l<=mid ; l++, k++ )
                sorted[k] = list[l];
        for( l=left ; l<=right ; l++ )
                list[l] = sorted[l];
}
void merge_sort ( int list[], int left, int right ) {
        if( left<right ) {
                int mid = (left+right)/2;
                merge_sort(list, left, mid);
                merge_sort(list, mid+1, right);
                merge(list, left, mid, right);
        }
}
```


병합 정렬 복잡도 분석

- 복잡도 분석
  - 비교 횟수
    - 크기 n인 리스트를 균등 분배하므로 log(n)개의 패스
    - 각 패스에서 레코드 n개를 비교 ➔ n번 비교연산
  - 이동 횟수
    - 각 패스에서 2n번 이동 발생 ➔ 전체 이동: 2n*log(n)
  - 시간 복잡도= O(n*log(n))
    - 최적, 평균, 최악의 경우 큰 차이 없음
- 분석
  - 효율적인 알고리즘
  - 안정적이며 데이터의 초기 분산 순서에 영향을 덜 받음

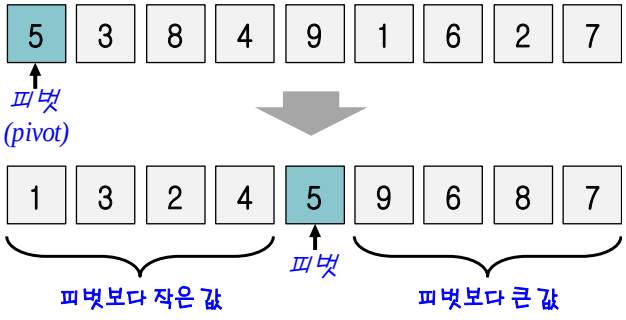
- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

71	49	92	55	38	82	72	53

1. 병합 정렬

13.8 퀵 정렬 (quick sort)

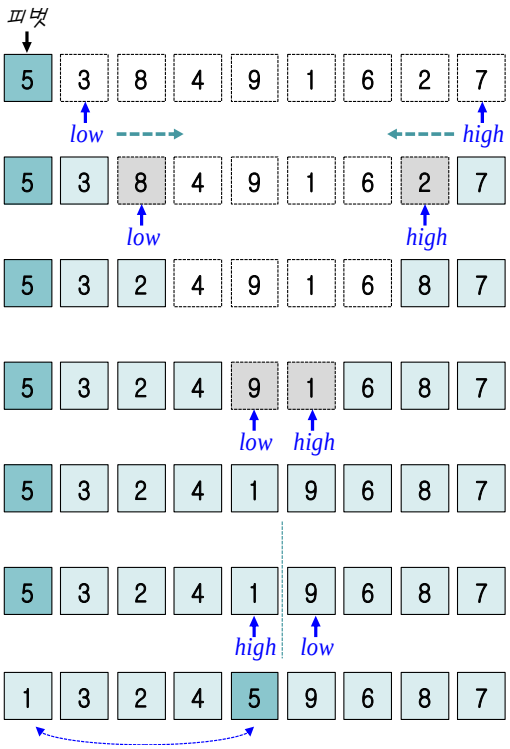
- 평균적으로 가장 빠른 정렬 방법
  - 분할 정복법 사용
  - 리스트를 2개의 부분리스트로 비균등 분할하고
  - 각각의 부분리스트를 다시 퀵 정렬함(순환 호출)



퀵정렬 알고리즘

```
// 퀵 정렬 : 배열의 left ~ right 항목들을 오름차순으로 정렬하는 함수
void quick_sort ( int list[], int left, int right )
{
    int q;
    if( left<right ){
        q = partition(list,left,right);
        quick_sort (list, left, q-1);
        quick_sort (list, q+1, right);
    }
}
```


분할 과정



5를 피벗으로 선택
low ← left+1
high ← right

low를 피벗보다 큰 항목까지 이동
high를 피벗보다 작은 항목까지 이동

low와 high의 항목 교체

다시 진행
low를 피벗보다 큰 항목까지 이동
high를 피벗보다 작은 항목까지 이동

low와 high의 항목 교체

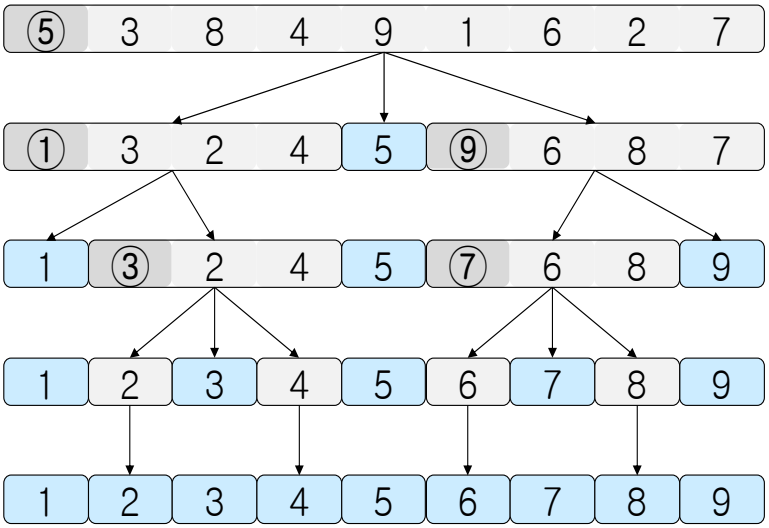
다시 진행
low와 high가 역전됨 → 종료

피벗과 high위치의 항목 교환

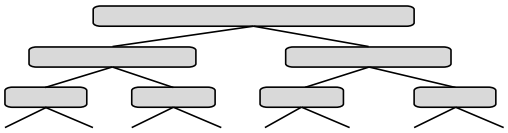
분할 함수 : partition()

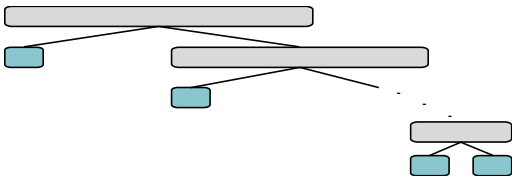
```
int partition ( int list[], int left, int right )
{
    int tmp;
    int low = left+1;
    int high = right;
    int pivot = list[left];        // 피벗 설정

    while(low < high ) {           // low와 high가 역전되지 않는 한 반복
        for ( ; low <=right && list[low] < pivot ; low++) ;
        for ( ; high>=left && list[high]> pivot ; high--) ;
        if( low < high )           // 선택된 두 레코드 교환
            SWAP(list[low],list[high],tmp);
    }
    SWAP(list[left],list[high],tmp); // high와 피벗 항목 교환
    return high;
}
```


퀵정렬 전체과정



퀵정렬 복잡도 분석





- 최선의 경우
  - 균등 분할
  - 패스 수: log(n)
  - 복잡도: O(nlog(n))

- 최악의 경우
  - 불균등 분할
    - (예) 이미 정렬된 리스트
  - 패스 수: n
  - 복잡도: $O(n^2)$
  - 피벗의 선택이 중요
    - 중간값(medium) 등

- 다음의 정렬기법을 이용하여 다음의 정수 배열을 오름차순으로 정렬하라. 각 단계에서의 배열의 내용을 나타내어라.

71	49	92	55	38	82	72	53

1. 퀵 정렬

- 다음과 같은 입력 배열을 퀵 정렬을 이용하여 정렬할 때, 피봇을 선택하는 방법을 다르게 하여 각 단계별 내용을 나타내어라.

1	2	3	4	5	6	7	8

1. 왼쪽 첫 번째 요소를 피봇으로 하는 방법
2. 왼쪽, 중간, 오른쪽 가운데 중간값(median of three) 방법

퀵정렬 라이브러리 함수

- C언어 표준 라이브러리 함수

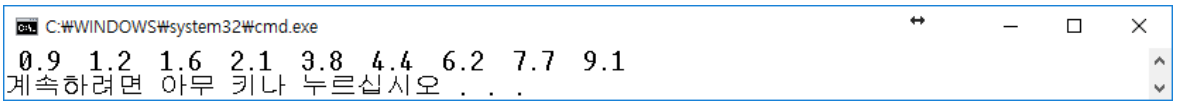
```
void qsort( void *base,
        int num,
        int width,
        int (*compare)(const void *, const void *)
);
```

  - base : 배열의 시작 주소
  - num : 배열 요소의 개수
  - width : 배열 요소 하나의 크기(바이트 단위).
  - compare : 비교 함수. 포인터를 통하여 두개의 요소를 비교하여 비교 결과를 정수로 반환한다. 사용자가 제공하여야 함.

반환값	설명
< 0	elem1이 elem2보다 작으면
0	elem1이 elem2과 같으면
> 0	elem1이 elem2보다 크면

퀵정렬 라이브러리 함수 사용 예

```
int compare( const void *arg1, const void *arg2 )
{
        if ( *(double *)arg1 > *(double *)arg2 ) return 1;
        else if( *(double *)arg1 < *(double *)arg2 ) return -1;
        else return 0;
}
int main()
{
        int i;
        double list[9] = {2.1, 0.9, 1.6, 3.8, 1.2, 4.4, 6.2, 9.1, 7.7};
        qsort( (void *)list, 9, sizeof(double), compare );
        for( i=0 ; i<9 ;i++ )
                printf("%4.1f ", list[i]);
}
```

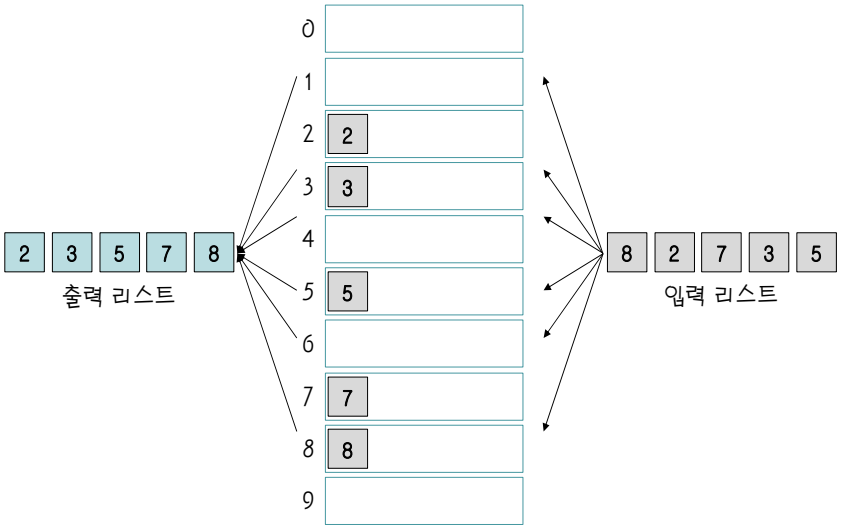


13.10 기수정렬(Radix Sort)

- 다른 정렬 방법들은 레코드를 비교하여 정렬 수행
- 기수 정렬은 레코드를 비교하지 않고 정렬 수행
  - 비교에 의한 정렬의 하한인 O(n*log(n)) 보다 좋을 수 있음
  - 기수 정렬은 O(dn) 의 시간 복잡도를 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

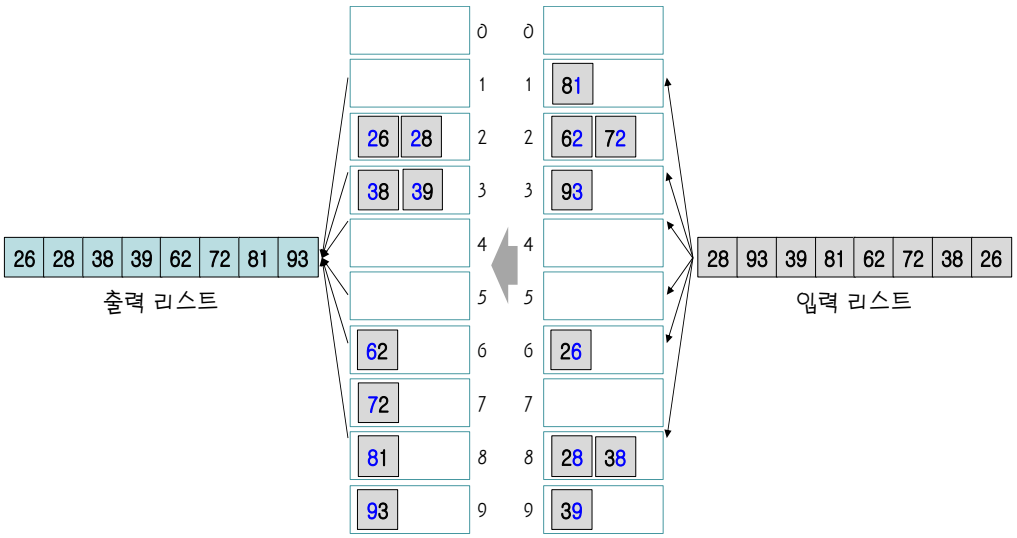
한 자릿수 기수정렬

- (8, 2, 7, 3, 5) 정렬의 예
  - 단순히 자리수에 따라 bucket에 넣었다가 꺼내면 정렬됨



두 자릿수 기수정렬

- (28, 93, 39, 81, 62, 72, 38, 26)
  - 낮은 자릿수 분류 → 순서대로 읽음 → 높은 자릿수 분류



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++;
```

  - 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

```
#define BUCKETS 10
#define DIGITS 4
void radixSort( int list[], int n )
{
    Queue queues[BUCKETS];
    int factor=1, i, b, d;
    for( i=0 ; i<BUCKETS ; i++ )
        init_queue( &queues[i] );
    for( d=0 ; d<DIGITS ; d++ ){
        for( i=0 ; i<n ; i++ )
            enqueue( &queues[(list[i]/factor)%10], list[i]);
        for( b=i=0 ; b<BUCKETS ; b++)
            while( !is_empty(&queues[b]) )
                list[i++] = dequeue( &queues[b] );
        factor *= 10;
    }
}
```

기수정렬 분석

- n개의 레코드, d개의 자릿수로 이루어진 키를 기수정렬
  - 메인 루프는 자릿수 d번 반복
  - 큐에 n개 레코드 입력 수행
- O(dn) 의 시간 복잡도
  - 키의 자릿수 d는 10 이하의 작은 수이므로 빠른 정렬임
- 실수, 한글, 한자로 이루어진 키는 정렬 못함

13.11 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	$O(n)$	$O(n^2)$	$O(n^2)$
셸 정렬	$O(n)$	$O(n^{1.5})$	$O(n^2)$
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	$O(dn)$	$O(dn)$	$O(dn)$