퀵 정렬
(이해)

- Divide and conquer에 근거
- 5개 변수의 의미
  - left 정렬대상의 가장 왼쪽 지점
  - right 정렬대상의 가장 오른쪽 지점
  - pivot 중심축의 의미
  - low 피벗을 제외한 가장 왼쪽에 위치한 지점
  - high 피벗을 제외한 가장 오른쪽에 위치한 지점

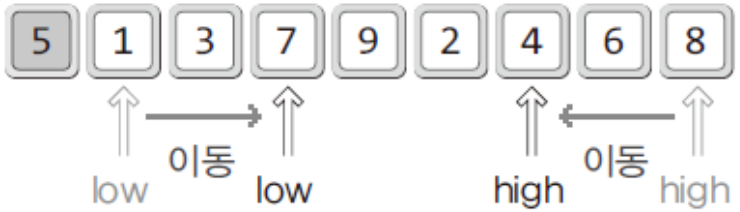
퀵 정렬
(이해)

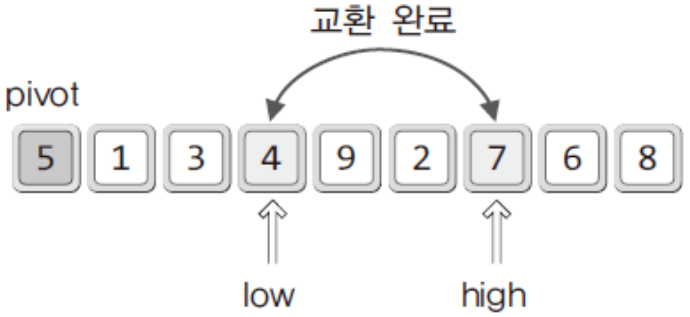
◆ 오름차순 정렬

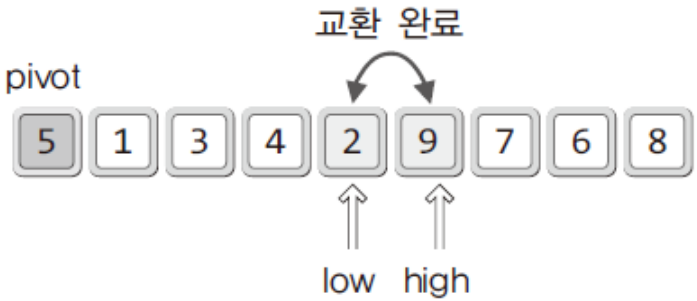
(1) pivot 선택 후, low와 high의 이동 (low위치에 pivot보다 큰 수, high 위치에 pivot보다 작은 수가 올 때까지)
(2) low와 high 위치의 데이터를 교환
(3) low, high 위치가 역전될 때까지 (1),(2)를 반복
(4) pivot과 high 위치의 데이터를 교환
(5) pivot 하나만 정렬된 상태가 됨
(6) pivot 기준으로 왼쪽과 오른쪽을 다시 퀵 정렬

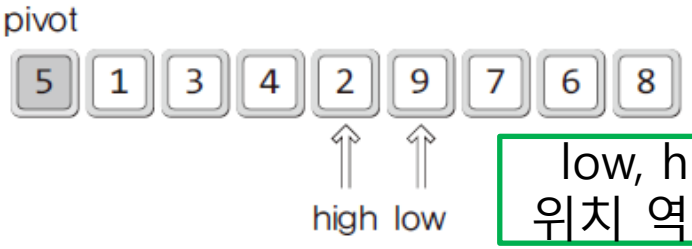
401-404p

퀵 정렬
(이해)

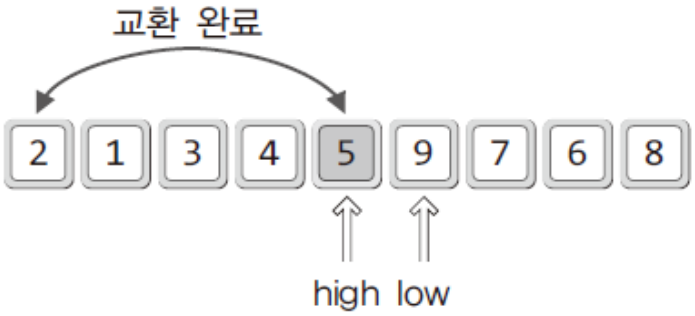


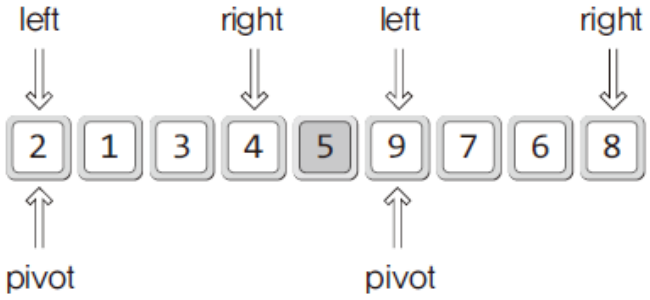






low, high
위치 역전됨





퀵 정렬
(구현)

```
#include <stdio.h>

void Swap(int arr[], int idx1, int idx2)
{
	int temp = arr[idx1];
	arr[idx1] = arr[idx2];
	arr[idx2] = temp;
}
```

405p

퀵 정렬
(구현)

```
int Partition(int arr[], int left, int right) {
    int pivot = arr[left];
    int low = left + 1;
    int high = right;

    while (low <= high) {
        while(pivot >= arr[low] && low <= right)
            low++;
        while(pivot <= arr[high] && high >= (left+1))
            high--;
        if (low <= high)
            Swap(arr, low, high);
    }
    Swap(arr, left, high);
    return high;
}
```

피벗의 위치는 가장 왼쪽!

교차되지 않은 동안 반복

조건에 유의!

교차되지 않은 상태라면 high, low위치의 값을 Swap

high, low가 교차된 후, 피벗과 high 위치의 값을 swap

옮겨진 피벗의 위치 정보 반환

퀵 정렬
(구현)

```
void QuickSort(int arr[], int left, int right) {
    if (left <= right) {
        int pivot = Partition(arr, left, right);
        QuickSort(arr, left, pivot - 1);
        QuickSort(arr, pivot + 1, right);
    }
}
int main(void) {
    int arr[7] = {3, 2, 4, 1, 7, 6, 5};
    //int arr[3] = { 3, 3, 3 };
    int len = sizeof(arr) / sizeof(int), i;

    QuickSort(arr, 0, sizeof(arr) / sizeof(int) - 1);
    for (i = 0; i < len; i++)
        printf("%d ", arr[i]);
    printf("₩n");
    return 0;
}
```

둘로 나눠서 왼쪽 영역과 오른쪽 영역을 각각 정렬

퀵 정렬
(구현:피벗 선택에 대한 논의)

- 피벗 : 가급적 중간에 해당하는 값이 선택되어야 좋은 성능을 보임.
- 개선방법 : 정렬 대상에서 3개의 값을 추출하여 그 중간 없을 pivot으로 선택.

pivot

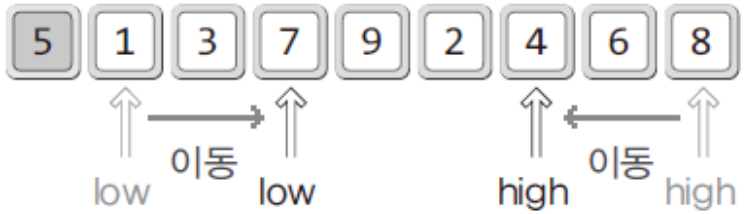
- 정렬이 되어 있고
- 피벗이 정렬 대상의 한쪽 끝에 치우치는 경우

* 최악의 효율

pivot

- 데이터가 불규칙적으로 나열되어 있고
- 피벗이 중간에 해당 하는 값에 가깝게 선택이 되면

* 최상의 효율



- 정렬과정에서 선택되는 피벗의 수가 적을수록

* 최상의 효율

퀵 정렬
(성능평가)

◆ 비교 연산 횟수
- $O(n\log_2 n)$
- 최악의 경우 $O(n^2)$ 이지만 다른 $O(n\log_2 n)$ 인 알고리즘보다 빠른 것으로 알려져 있음.

◆ 퀵 정렬의 장점
- 데이터 이동 횟수가 적음.
- 별도의 메모리 공간이 필요 없음.

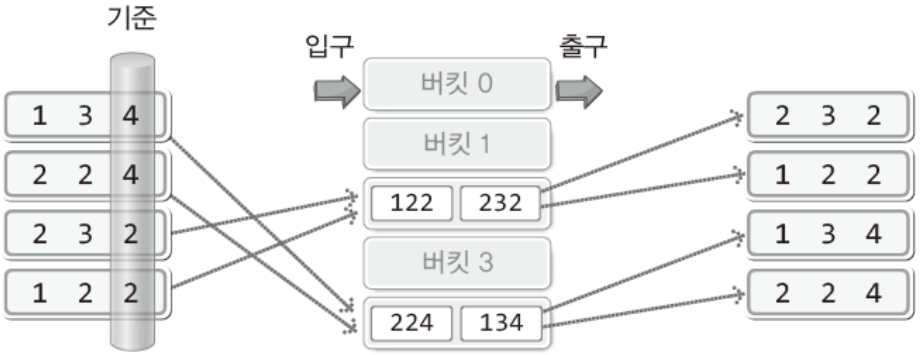
412-413p

기수 정렬
(이해)

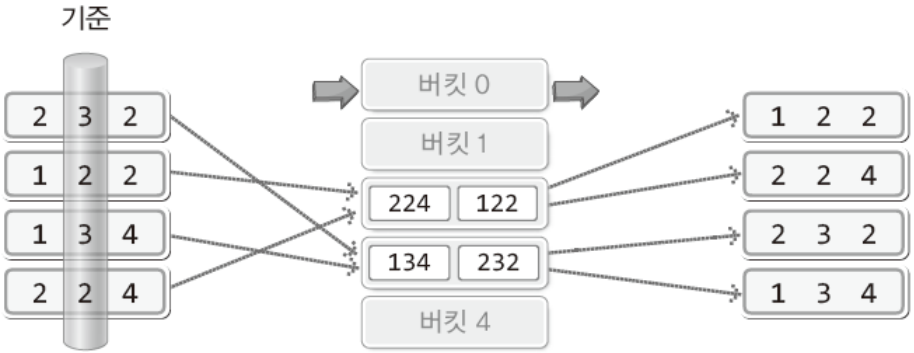
- 특징
  - 정렬순서의 앞서고 뒤섬을 비교하지 않는다.
  - 정렬 알고리즘의 한계로 알려진 O(nlog2n)을 뛰어 넘을 수 있다.
  - 적용할 수 있는 대상이 매우 제한적이다. 길이가 동일한 데이터들의 정렬에 용이하다!
- 용어
  - 기수(radix): 주어진 데이터를 구성하는 기본 요소(기호)
  - 버킷(bucket): 기수의 수에 해당하는 만큼의 버킷을 활용한다.
- 방법
  - 단순히 버킷에 넣고 빼면 됨.
  - 즉 정렬의 과정에서 데이터간 비교가 발생하지 않음.

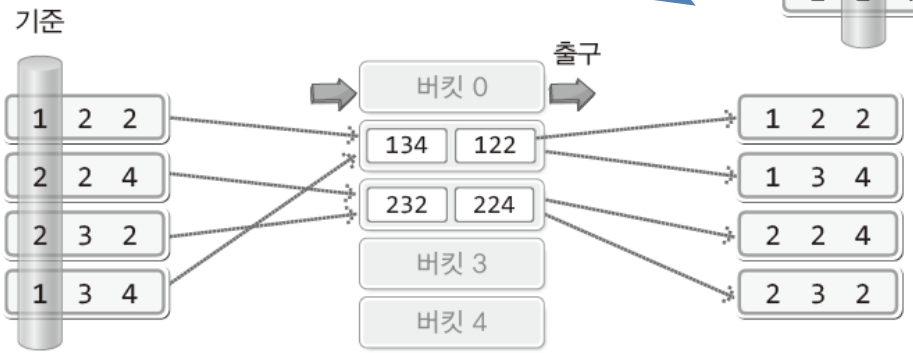
LSD(Least Significant Digit)기수 정렬 (이해)

414-416p



LSD이므로
3번째 자리 기준으로 버켓에 넣었다가 빼고,
2번째 자리 기준으로 버켓에 넣었다가 빼고, …



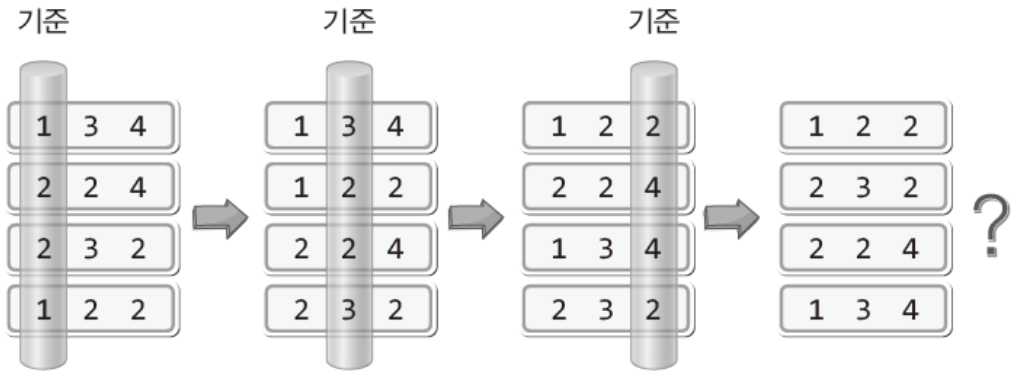


MSD 까지 진행을 해야 값의 우선순위대로 정렬이 완성됨.

[단점] 중간에는 대략의 정렬된 결과도 얻을 수 없음.

MSD(Most Significant Digit)기수 정렬 (이해)

416-417p



정렬의 기준 선정 방향만을 바꾸어서 기수 정렬을 진행한 결과!

MSD 방식은 점진적으로 정렬이 완성되어 가는 방식.

따라서 중간 중간에 정렬이 완료된 데이터는 더 이상의 정렬과정을 진행하지 않아야 한다.

이를 위해 중간에 데이터를 점검해야 함.

LSD 기수정렬
(구현)

- 양의 정수라면 길이에 상관 없이 정렬 대상에 포함시키기 위한 간단한 알고리즘
  - NUM으로부터 첫 번째 자리 숫자 추출 — NUM / 1 % 10
  - NUM으로부터 두 번째 자리 숫자 추출 — NUM / 10 % 10
  - NUM으로부터 세 번째 자리 숫자 추출 — NUM / 100 % 10

- 위와 같이 하면 아래와 같이 자리수가 다른 숫자를 최고 자리수까지 비교할 때 MSD쪽 digit을 0으로 간주하므로 편리.
  - 42, 715 를 정렬

LSD 기수정렬
(연결리스트 기반 큐 활용한 구현)

418-420p

```
#include <stdio.h>
#define TRUE    1
#define FALSE   0
typedef int Data;
typedef struct _node {
        Data data;
        struct _node * next;
} Node;
typedef struct _lQueue {
        Node * front;
        Node * rear;
} LQueue;
typedef LQueue Queue;
void QueueInit(Queue * pq) {
        pq->front = NULL;
        pq->rear = NULL;
}
int QIsEmpty(Queue * pq) {
        if (pq->front == NULL)
                return TRUE;
        else
                return FALSE;
}
```

LSD 기수정렬
(연결리스트 기반 큐 활용한 구현)

```
void Enqueue(Queue * pq, Data data) {
	Node * newNode = (Node*)malloc(sizeof(Node));
	newNode->next = NULL;
	newNode->data = data;

	if (QIsEmpty(pq)) {
		pq->front = newNode;
		pq->rear = newNode;
	}
	else {
		pq->rear->next = newNode;
		pq->rear = newNode;
	}
}
```

LSD 기수정렬
(연결리스트 기반 큐 활용한 구현)

```
Data Dequeue(Queue * pq) {
        Node * delNode;
        Data retData;
        if (QIsEmpty(pq)) {
                printf("Queue Memory Error!");
                exit(-1);
        }
        delNode = pq->front;
        retData = delNode->data;
        pq->front = pq->front->next;
        free(delNode);
        return retData;
}
Data QPeek(Queue * pq) {
        if (QIsEmpty(pq)) {
                printf("Queue Memory Error!");
                exit(-1);
        }
        return pq->front->data;
}
```

418-420p

LSD 기수정렬
(연결리스트 기반 큐 활용한 구현)

```
#define BUCKET_NUM          10
void RadixSort(int arr[], int num, int maxLen) {
    Queue buckets[BUCKET_NUM];
    int bi, pos, di, divfac = 1, radix;
    for (bi = 0; bi < BUCKET_NUM; bi++)
        QueueInit(&buckets[bi]);
    for (pos = 0; pos < maxLen; pos++) {
        for (di = 0; di < num; di++) {
            radix = (arr[di] / divfac) % 10;
            Enqueue(&buckets[radix], arr[di]);
        }
        for (bi = 0, di = 0; bi < BUCKET_NUM; bi++) {
            while (!QIsEmpty(&buckets[bi]))
                arr[di++] = Dequeue(&buckets[bi]);
        }
        divfac *= 10;
    }
}
```

가장 긴 데이터의 길이

총 10개의 버킷 초기화

가장 긴 데이터의 길이만큼 반복

정렬 대상의 수만큼 반복

N번째 자리의 숫자 추출. 추출한 숫자를 근거로 데이터 버킷에 저장.

버킷 수만큼 반복

버킷에 저장된 것 순서대로 다 꺼내서 다시 arr에 저장

N번째 자리의 숫자 추출을 위한 피제수의 증가

LSD 기수정렬
(연결리스트 기반 큐 활용한 구현)

```
int main(void)
{
	int arr[7] = { 13, 212, 14, 7141, 10987, 6, 15 };

	int len = sizeof(arr) / sizeof(int);
	int i;

	RadixSort(arr, len, 5);

	for (i = 0; i < len; i++)
		printf("%d ", arr[i]);

	printf("₩n");
	return 0;
}
```

LSD 기수정렬
(성능평가)

◆ 핵심 연산

- 비교 연산이 핵심이 아님.
- 삽입/추출 빈도수를 핵심으로 간주.
  maxLen x num
- 정렬대상의 수가 n이고, 모든 정렬대상의 길이를 l 이라 할 때 시간 복잡도에 대한 기수정렬의 빅-오는…
  $O(ln) \rightarrow O(n)$