11. Sorting

2017010698
수학과 오서영

목차

Sorting (분류) -> 실행 방법에 따른 분류

비교식 정렬(comparative sort)

: 비교하고자 하는 각 키 값들을 한 번에 두 개씩 비교하여 교환하는 방식으로 정렬을 실행

분산식 정렬(distribute sort)

: 키 값을 기준으로 하여 자료를 여러 개의 부분 집합으로 분해하고, 각 부분 집합을 정렬함으로써 전체를 정렬하는 방식

Sorting (분류) -> 정렬 장소에 따른 분류

내부 정렬(internal sort)

: 정렬할 자료를 메인 메모리에 올려서 정렬하는 방식
-> 정렬 속도가 빠르지만
정렬할 수 있는 자료의 양이 메인 메모리의 용량에 따라 제한

(ex) 교환 방식(Selection, Bubble, Quick),
삽입 방식(Insertion, Shell), 분배 방식(Radix),
병합 방식(2-way 병합, n-way 병합), 선택 방식(Heap, Tree)

외부 정렬(external sort)

: 정렬할 자료를 보조 기억장치에서 정렬하는 방식
-> 내부 정렬보다 속도는 떨어지지만
내부 정렬로 처리할 수 없는 대용량 자료에 대한 정렬이 가능

(ex) 병합 방식(2-way 병합, n-way 병합)

1. Bubble Sort (버블 정렬)

Bubble Sort

: 인접한 두 개의 원소를 비교하여 자리를 교환
-> 인접한 2개의 레코드를 비교하여 크기가 순서대로
되어 있지 않으면 서로 교환

1회전을 수행하고 나면 가장 큰 자료가 맨 뒤로 이동하므로
2회전에서는 맨 끝에 있는 자료는 정렬에서 **제외**
-> 1회전 수행할 때마다
정렬에서 제외되는 데이터가 하나씩 늘어난다.

장점 : 구현이 매우 간단
단점 : 일반적으로 자료의 교환 작업(SWAP)이
자료의 이동 작업(MOVE)보다 더 복잡하기 때문에
단순성에도 불구하고 거의 쓰이지 않는다.

1. Bubble Sort (버블 정렬)

버블정렬 C언어 코드

```
void bubble_sort(int list[], int n){
    int i, j, temp;

    for (i=n-1; i>0; i--){
    // 0 ~ (i-1)까지 반복
        for (j=0; j<i; j++){
        // j번째와 j+1번째의 요소가 크기 순이 아니면 교환
            if (list[j]<list[j+1]){
               temp = list[j];
               list[j] = list[j+1];
               list[j+1] = temp;       }    }  }}
```

2. Insertion Sort (삽입 정렬)

Insertion Sort

: 자료 배열의 모든 요소를 앞에서부터 차례대로
이미 정렬된 배열 부분과 비교 하여,
자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘
-> 손안의 카드를 정렬하는 방법과 유사.

1. 부분집합 **S** : 정렬된 앞 부분의 원소들
2. 부분집합 **U** : 아직 정렬되지 않은 나머지 원소들
3. 정렬되지 않은 부분집합 **U**의 원소를 하나씩 꺼내서 이미 정렬되어 있는 부분집합 **S**의 마지막 원소부터 비교하면서 위치를 찾아 삽입한다.
4. 삽입 정렬을 반복하면서 부분집합 **S**의 원소는 하나씩 늘리고 부분집합 **U**의 원소는 하나씩 감소하게 된다.
5. 부분집합 **U**가 공집합이 되면 정렬이 완성된다.

2. Insertion Sort (삽입 정렬)

장점 : 대부분 위 레코드가 이미 정렬되어 있는 경우에 매우 효율적일 수 있다
단점 : 레코드 수가 많고 레코드 크기가 클 경우에 적합하지 않다.

삽입정렬 C언어 코드

```
void insertion_sort(int list[], int n){
    int i, j, key;
    // 인덱스 0은 이미 정렬된 것으로 볼 수 있다.
    for (i=1; i<n; i++) {
        key = list[i]; // 현재 삽입될 숫자인 i번째 정수를 key 변수로 복사
// 현재 정렬된 배열은 i-1까지이므로 i-1번째부터 역순으로 조사한다.
// j 값은 음수가 아니어야 되고
// key 값보다 정렬된 배열에 있는 값이 크면 j번째를 j+1번째로 이동
        for (j=i-1; j>=0 && list[j]>key; j--){
            list[j+1] = list[j]; // 레코드의 오른쪽으로 이동     }
        list[j+1] = key;  }}
```

3. Selection Sort (선택 정렬)

Selection Sort

: 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘

1. 전체 원소 중에서 가장 작은 원소를 찾아 선택하여 첫 번째 원소와 자리를 교환한다.
2. 그다음 두 번째로 작은 원소를 찾아서 선택하여 두 번째 원소와 자리를 교환한다.
3. 그다음에는 세 번째로 작은 원소를 찾아 선택하여 세 번째 원소와 자리를 교환한다.
4. 이 과정을 반복

장점 : 자료 이동 횟수가 미리 결정됨
단점 : 값이 같은 레코드가 있는 경우, 상대적 위치가 변경 될 수 있다.

3. Selection Sort (선택 정렬)

선택 정렬 C언어 코드

```
void selection_sort(int list[], int n){
    int i, j, least, temp;
// 마지막 숫자는 자동으로 정렬되기 때문에 (숫자 개수-1) 만큼 반복한다.
    for (i=0; i<n-1; i++){
        least = i;    // 최솟값을 탐색한다.
        for (j=i+1; j<n; j++){
            if (list[j]<list[least])
                least = j;    }
// 최솟값이 자기 자신이면 자료 이동을 하지 않는다.
        if (i != least){
            SWAP(list[i], list[least], temp);    }  }}
```

4. Quick Sort (퀵 정렬)

Quick Sort

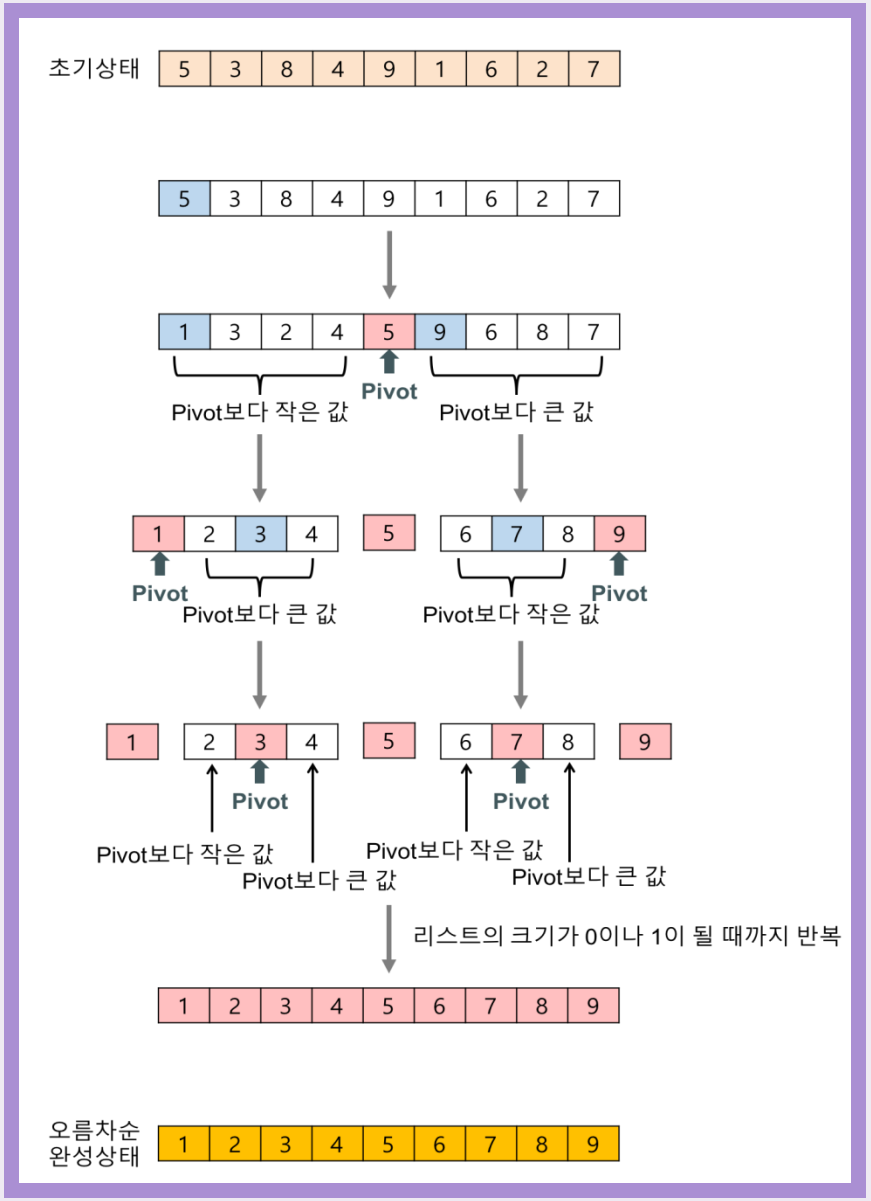
: 정렬할 전체 원소에 대해서 정렬을 수행하지 않고,
기준 값을 중심으로 왼쪽 부분 집합과 오른쪽 부분 집합으로
분할하여 정렬하는 방법

1. 왼쪽 부분 집합에는 기준 값보다 작은 원소들을 이동시키고, 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킨다.
2. **기준 값** : 피봇 (pivot), 일반적으로 전체 원소 중에서 가운데에 위치한 원소를 선택한다.
3. **분할(divide)** : 정렬할 자료들을 기준 값을 중심으로 2개의 부분 집합으로 분할한다.
4. **정복(conquer)** : 부분집합의 원소들 중에서 기준 값보다 작은 원소들은 왼쪽 부분집합으로, 기준 값보다 큰 원소들은 오른쪽 부분집합으로 정렬한다. 부분집합의 크기가 1이하로 충분히 작지 않으면 순환 호출을 이용하여 다시 분할한다.

4. Quick Sort (퀵 정렬)

장점 : 속도가 빠르다,
추가 메모리 공간 필요 X

단점 : 정렬된 리스트의 경우
오히려 시간이 더 오래걸림



5. Merge Sort (병합 정렬)

Merge Sort

: 2개 이상의 자료를 오름차순이나 내림차순으로 재배열

병합 정렬 방법의 종류

- **2-way 병합** : 2개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법
- **n-way 병합** : n 개의 정렬된 자료의 집합을 결합하여 하나의 집합으로 만드는 병합 방법

2-way 병합

1. 분할(divide) : 입력 자료를 같은 크기의 부분집합 2개로 분할
2. 정복(conquer) : 부분집합의 원소들을 정렬한다.
부분집합의 크기가 충분히 작지 않으면 순환 호출을 이용하여 다시 분할 정복 기법을 적용
3. 결합(combine) : 정렬된 부분집합들을 하나의 집합으로 결합
4. 1, 2, 3의 과정을 반복 수행하면서 정렬을 완성시킨다.

6. Heap Sort (힙 정렬)

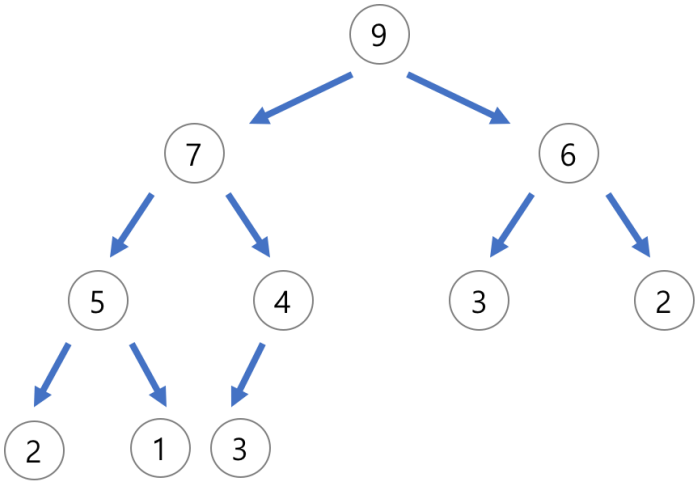
Heap Sort

: 최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법
-> 내림차순 정렬을 위해서는 최대 힙을 구성하고, 오름차순 정렬을 위해서는 최소 힙을 구성하면 된다.

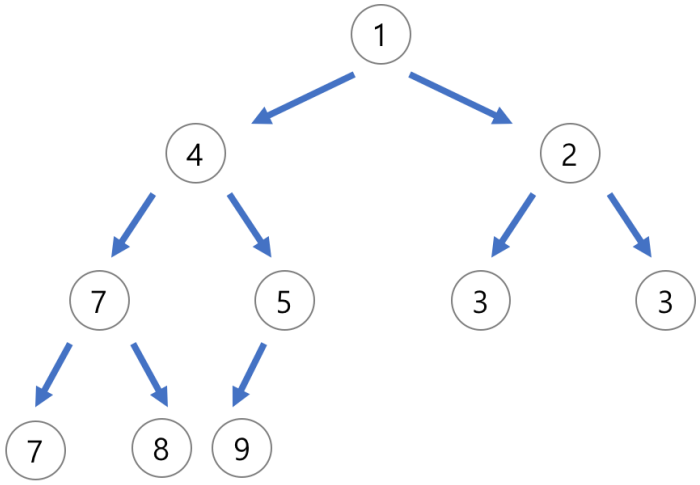
1. n 개의 노드에 대한 완전 이진 트리를 구성한다.
이때 루트 노드부터 부노드, 왼쪽 자노드,
오른쪽 자노드 순으로 구성한다.
2. 최대 힙을 구성한다. 최대 힙이란 부노드가 자노드보다
큰 트리를 말하는데, 단말 노드를 자노드로 가진
부노드로부터 구성하며 아래부터 루트까지 올라오며
순차적으로 만들어 갈 수 있다.
3. 가장 큰 수(루트에 위치)를 가장 작은 수와 교환한다.
4. 2와 3의 과정을 반복한다.

6. Heap Sort (힙 정렬)

장점 : 속도가 빠르다,
가장 큰 값 몇 개만
필요할 때 유용하다



-최대 힙(max heap)-



-최소 힙(min heap)-

7. Radix Sort

Radix Sort

: 원소의 키값을 나타내는 기수를 이용한 정렬 방법
-> 정렬할 원소의 키값에 해당하는 버킷(bucket)에 원소를 분배하였다가 버킷의 순서대로 원소는 꺼내는 방법을 반복
(ex) 원소의 키를 표현하는 기수만큼의 버킷을 사용.
-> 10진수로 표현된 키값을 가진 원소들을 정렬할 때에는 0부터 9까지의 10개의 버킷 사용

1. 키값의 일의 자리에 대해서 기수 정렬을 수행
2. 다음 단계에서는 키값의 십의 자리에 대해서 정렬을 수행
3. 그리고 그다음 단계에서는 백의 자리에 대해서 정렬을 수행
4. 1, 2, 3에서 진행되었던 것처럼 자릿수만큼 반복하여 정렬을 수행한다.

8. Shell Sort (셸 정렬)

Shell Sort

: 삽입 정렬 보완
-> 삽입정렬의 최대 문제점 : 요소들이 삽입될 때,
만약 삽입되어야 할 위치가 현재 위치에서 상당히 멀리 떨어진
곳이라면 많은 이동을 해야만 제자리로 갈 수 있다.
삽입 정렬과 다르게 셸 정렬은 전체의 리스트를
한 번에 정렬하지 않는다.

1. 먼저 정렬해야 할 리스트를 일정한 기준에 따라 분류
2. 연속적이지 않은 여러 개의 부분 리스트를 생성
3. 각 부분 리스트를 삽입 정렬을 이용하여 정렬
4. 모든 부분 리스트가 정렬되면 다시 전체 리스트를 더 적은 개수의 부분 리스트로 만든 후에 알고리즘을 반복
5. 위의 과정을 부분 리스트의 개수가 1이 될 때까지 반복

장점 : 삽입 정렬보다 더욱 빠르게 실행된다.
알고리즘이 간단하다.

8. Shell Sort (셸 정렬)

셸 정렬 C언어 코드

```
void 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];
// 현재 삽입될 숫자인 i번째 정수를 key 변수로 복사
// 현재 정렬된 배열은 i-gap까지이므로 i-gap번째부터 역순으로 조사한다.
// j 값은 first 이상이어야 하고
// key 값보다 정렬된 배열에 있는 값이 크면 j번째를 j+gap번째로 이동
        for (j=i-gap; j>=first && list[j]>key; j=j-gap){
            list[j+gap] = list[j];  }  // 레코드를 gap만큼 오른쪽으로 이동
            list[j+gap] = key;  }}
// 셸 정렬
void shell_sort(int list[], int n){
    int i, gap;
    for (gap=n/2; gap>0; gap=gap/2){
        if ((gap%2) == 0)(       gap++; // gap을 홀수로 만든다.      )
// 부분 리스트의 개수는 gap과 같다.
        for (i=0; i<gap; i++){     // 부분 리스트에 대한 삽입 정렬 수행
            inc_insertion_sort(list, i, n-1, gap);     }  }}
```

알고리즘 성능

$O(n^2)$: Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Quick Sort

O(n log n) : Heap Sort, Merge Sort

O(kn) : Radix Sort (k는 자릿수)

Reference

[1] 정렬 알고리즘 기초,
https://medium.com/@joongwon/%EC%A0%95%EB%A0%AC-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-%EA%B8%B0%EC%B4%88-805391cb088e

[2] [알고리즘] 퀵 정렬(quick sort)이란,
https://gmlwjd9405.github.io/2018/05/10/algorithm-quick-sort.html