

알고리즘

과제번호	01주차
날 짜	2018.09.19
학 번	201302395
이 름	류경빈

Insertion Sort (삽입 정렬)

```
public static int[] insertionSort(int[] arr){
    for (int i = 1; i < arr.length; i++){
        int key = arr[i];
        int standard = i-1;

        while (standard >= 0 && key < arr[standard]){
            arr[standard+1] = arr[standard];
            standard--;
        }
        arr[standard+1] = key;
    }
    return arr;
}
```

- Insertion Sort : 정렬되지 않은 배열의 원소들의 1번째 원소를 key 값으로 지정해 0번째 원소와 비교를 시작으로 계속해서 나머지 원소들과 비교해 순서대로 정렬한다.
- 시간복잡도 : Insertion Sort는 최악의 경우 모든 원소를 하나씩 모두 비교할 경우 $O(n^2)$ 이다. 하지만 모든 원소가 이미 정렬되어 있는 경우에는 한번씩 밖에 비교하지 않아 시간 복잡도는 $O(n)$ 이다. 하지만 Big-O notation은 최악의 경우이기 때문에 삽입정렬의 시간 복잡도는 $O(n^2)$ 이다.

Binary Search Insertion Sort (이진 탐색 삽입 정렬)

```
public static int[] binaryInsertionSort(int[] arr){
    int key, high, low, mid;
    for (int i = 1; i < arr.length; i++){
        key = arr[i];
        low = 0;
        high = i-1;

        while (low <= high){
            mid = (low + high)/2;
            if (arr[mid] < key){
                low = mid+1;
            }
            else {
                high = mid-1;
            }
        }
        for (int j = i-1; j > low-1; j--){
            arr[j+1] = arr[j];
        }
        arr[low] = key;
    }
    return arr;
}
```

- Binary Search Insertion Sort : 삽입 정렬을 실행할 때 비교의 횟수를 줄이기 위해 이진탐색을 이용해서 정렬한다. 중간 값을 mid로 받아와 해당 값이 클 경우 우측에, 작을 경우 좌측에 정렬시켜 정렬된 배열을 만든다.
- 시간복잡도 : 이진탐색의 탐색 시간은 $O(\log n)$ 으로 줄일 수 있지만 전체 알고리즘의 최악의 경우 Big-O notation은 똑같이 $O(n^2)$ 이 된다.

Insertion Sort 와 Binary Search Insertion Sort 비교

	Insertion Sort	Binary Search Insertion Sort
O-notation	$O(n^2)$	$O(n^2)$
Ω -notation	$\Omega(n)$	$\Omega(\log n)$
Θ -notation	$\Theta(n^2)$	$\Theta(n^2)$

Insertion Sort 와 Binary Search Insertion Sort 성능분석

```
int[] nums_insertion = Arrays.stream(splitedStr).mapToInt(Integer::parseInt).toArray();
int[] nums_binary = Arrays.stream(splitedStr).mapToInt(Integer::parseInt).toArray();

long start_insertion = System.nanoTime();

int[] insertionSortedArray = insertionSort(nums_insertion);

long end_insertion = System.nanoTime();

long start_binary = System.nanoTime();

int[] binaryInsertionSortedArray = binaryInsertionSort(nums_binary);

long end_binary = System.nanoTime();
Double time_binary = Double.valueOf(end_binary) - Double.valueOf(start_binary);
Double time_insertion = Double.valueOf(end_insertion) - Double.valueOf(start_insertion);

System.out.println("InsertionSort Run Time : " + time_insertion/1000000);
System.out.println("binaryInsertionSort Run Time : " + time_binary/1000000);
```

데이터 50개	Insertion Sort	Binary Search Insertion Sort
1차	0.018242 sec	0.022906 sec
2차	0.025836 sec	0.028893 sec
3차	0.023072 sec	0.028435 sec
4차	0.02589 sec	0.033505 sec
5차	0.028927 sec	0.029162 sec
평균	0.0243934 sec	0.0285802 sec

- 데이터가 고정적으로 50개 있을 경우 Insertion Sort가 Binary Search Insertion Sort보다 좋은 성능을 낸다. 둘 다 같은 $O(n^2)$ 의 평균적인 성능을 가지고 있더라도 경우에 따라서 Insertion Sort가 더 좋은 평균 속도를 나타낸다.

Insertion Sort 와 Binary Search Insertion Sort 성능분석

데이터 개수에 따른 Run Time 비교

데이터 50개	InsertionSort Run Time : 0.026492 binaryInsertionSort Run Time : 0.033779
데이터 100개	InsertionSort Run Time : 0.185425 binaryInsertionSort Run Time : 0.137159
데이터 200개	InsertionSort Run Time : 0.389768 binaryInsertionSort Run Time : 0.361627
데이터 1000개	InsertionSort Run Time : 4.539838 binaryInsertionSort Run Time : 3.95004
데이터 10000개	InsertionSort Run Time : 36.141657 binaryInsertionSort Run Time : 27.208319

- 데이터의 개수가 적을 때는 insertion Sort와 Binary Search Insertion Sort와 비교가 되지 않지만, 데이터의 개수가 많아질수록 실행 시간이 Binary Search Insertion Sort가 더 빠르다.
- 이론적으로 Insertion Sort와 Binary Search Insertion Sort가 모두 $\Theta(n^2)$ 의 성능으로 Average 속도를 가지고 있지만 이진탐색으로 이루어질 경우 반씩 잘라서 비교해가는 과정이 $\log n$ 만큼 시간을 절약하기 때문에 데이터 값이 클수록 Binary Search Insertion Sort가 더 빠르게 정렬한다.

Merge Sort (합병 정렬)

```
public static int[] mergeSort(int[] arr, int l, int r) {
    if(l<r) {
        count++;
        int mid = (l+r)/2; // mid 계산
        mergeSort(arr, l, mid); // merge left 전반부 정렬
        mergeSort(arr, mid+1, r); // merge right 후반부 정렬
        merge(arr, l, mid, r); // 합 arr의 l 부터 mid 까지, mid+1에서 r 까지
    }
    return arr;
}

public static void merge(int arr[], int l, int mid, int r) {
    int i = l;
    int j = mid+1; // merge right 후반부 첫 번째 원소
    int k = l; // temp의 시작점을 뜻
    int temp[] = new int[arr.length];
    int count = 0;

    while(i<=mid && j<=r) { // merge right와 left 배열이 끝날 때 까지를 의미 i값이 mid 값 즉 merge left의 값 끝까지를 뜻함..
        if(arr[i] < arr[j]) { // merge left와 right 값을 비교해서 더 작은 값을 가지는 배열의 원소를 temp에 저장한다.
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }
    // merge left, right 둘 중 하나의 배열의 원소를 모두 탐색했을 경우 넘어간다.
    while(i<=mid) // merge left 배열의 원소들이 남았을 경우
        temp[k++] = arr[i++];
    while(j<=r) // merge right 배열의 원소들이 남았을 경우
        temp[k++] = arr[j++];
    for(int index=l; index<k; index++) {
        arr[index] = temp[index]; // temp 배열을 arr에 복사중..
    }
}
```

- Merge Sort : Recursive로 만들어진 함수인 mergeSort 메소드는 정렬되지 않은 배열의 원소들을 계속해서 1/2로 나누어 한 개의 원소를 가지는 배열이 될 때까지 반복한다. 그렇게 된 배열의 원소들을 비교해서 순서대로 합병하며 정렬한다.
- mergeSort 함수가 시작할 때 count 값을 증가 시켜 해당 메소드 함수가 실행되는 횟수를 구한다. 그리고 생성되는 txt파일에 마지막으로 함께 출력한다.

1381,1469,1492,1646,2047,2560,2598,2937,6176,6615,7966,8110,8155,8401,10171,10696,10816,11712,11891,12833,14236,14327,15667,16012,16372,16590,16756,18444,18829,19454,19563,20144,21239,21346,21400,22342,22691,22750,22890,25177,25449,27539,28300,28317,28678,29256,30065,30682,31904,32180 49