< 소팅 알고리즘 보고서 >

이 세상에는 수 많은 정보들이 존재하고 이를 활용하기 위해서는 적절한 자료구조와 알고리즘이 필요하다.

자료구조란 자료를 효율적으로 저장하고 관리하기 위한 데이터 구조, 그리고 알고리즘이란 주어진 문제를 효율적으로 풀기 위한 방법이다.

사실 우리 주위엔 정보를 정렬하는 것이 필수적인 분야들이 많다. 사람들은 자료를 입렬할 때 무작위로 입력하는 경우가 대부분입이다. 따라서 우리는 '사람들은 정렬을 하지 않고 자료를 입력한다'는 가정을 가지고 있어야 한다. 대한민국의 수많은 중,고등학생의 정보가 정렬 없이 너저분하게 널려 있는 모습, 그리고 선생님이 내 이름을 찾으려 낑낑대는 모습을 상상할 수 있다.

이렇게 정렬 알고리즘의 필요성은 수 많은 정보들이 존재하고 만들어지고 있는 현대 사회에서 그 필요성이 더 가중되고 있다. 주어진 데이터들을 순서대로 정렬하고 싶을 때, 다양한 알고리즘이 존재한다. 오름차순으로 정렬할지, 내림차순으로 정렬할지에 대한 분류부터 시작하여 시간복잡도와 공간복잡도의 차이까지 고려하게 되면 매우 다양한 정렬 방법이 존재하게 된다.

그 중에서 가장 대표적인 5 가지 정렬 선택정렬, 삽입정렬, 병합정렬, 퀵정렬, 힙정렬에 대한 고찰을 해보려고 한다.

각 정렬 방법은 그 알고리즘이 다르고 각각의 장단점이 존재한다. 나는 최종적으로 각 정렬방법의 시간적 효율과 공간적 효율에 대해 정리해보고, 각각의 한계점과 개선점까지도 알아보고자 하였다. 그에 앞서, 각각 정렬이 어떤 알고리즘을 바탕으로 주어진 데이터들을 정렬하는 지를 살펴보도록 하자.

선택정렬은 말 그대로 매 순간에서의 선택을 통해 정렬을 구현하는 알고리즘이다. 다시 말하자면, 데이터들이 주어졌을 때 가장 큰 수나 가장 작은 수를 기준으로 잡아서 첫 번째 데이터부터 시작하여 뒤로 가면서 각각 값들을 비교하여 오름차순 또는 내림차순으로 정렬하는 방법이다.

삽입정렬은 데이터들을 순서대로 비교해가면서 자신의 값보다 크거나

작은 것이 발견되면 그 지점을 기준으로 앞에 삽입하거나 뒤에 삽입하는 방식으로 정렬을 하는 방법이다. 각각 삽입을 할 시에는, 넣고자 하는 지점의 원래 값을 잠시 다른 곳에 저장해두고 삽입을 진행한 후 나머지 데이터들을 한 칸씩 뒤로 밀어주거나 한 칸씩 앞으로 밀어주는 방식을 통해 나머지 데이터의 손실이 없도록 정렬을 하면 된다.

병합정렬은 데이터들을 하나단위의 원소로 쪼개어 다시 순서대로 붙이는 과정을 통해 정렬하는 방법이다. 이는 재귀함수를 이용하는 것이 유용하다. 먼저 데이터를 하나하나 쪼개는 방식은 데이터의 가운데를 정하여 이의 앞부분과 뒷부분을 나누는 방법인데, 이것은 재귀함수를 이용하여 terminal 조건을 데이터의 길이 (원소의 개수)가 1 개일 때까지로 정해주면 논리적으로 진행할 수 있다. 붙이는 과정, Merging, 에서는 서로 다른 한 쌍의 데이터가 만나 붙는데, 새로운 길이 (두 데이터 길이의 합)의 데이터베이스를 만들고 그것을 각각 두 데이터로부터 나오는 정렬된 데이터들을 비교하면서 순차적으로 데이터를 넣어주면 된다. 이러한 방식으로 진행하면 결과적으로 데이터를 모두 쪼개고 다시 순차적으로 붙이는 방법을 통해 데이터를 정렬할 수 있게 된다.

퀵 정렬은 pivot (중심점) 을 지정하여 비교하는 방법이다. 주어진 데이터의 초기 pivot 을 임의로 잡고 그것과 크기를 비교하여 pivot 앞 부분을 pivot 에 위치한 값보다 항상 작거나 크도록, 뒷 부분을 pivot 에 위치한 값보다 항상 크거나 작도록 위치를 조정해주어 오름차순 또는 내림차순으로 정렬해주는 방법이다. pivot 으로 한 지점이 잡히고, 그곳을 기준으로 데이터의 pivot 앞 부분이 pivot 값보다 모두 작고 뒷 부분이 pivot 값보다 모두 크면, 그 pivot 값에 있는 데이터는 그 위치가 최종 오름차순에서의 위치라고 할 수 있게 되는 것이다. 따라서 pivot 값을 지정하고 앞과 뒤의 데이터를 확인하는 방식을 통해 정렬을 하면 되므로, 재귀함수를 쓰면 유용하고 이 때의 terminal 조건은 pivot 을 정하여 나누고자 하는 주어진 데이터의 크기가 1 일때가 된다.

힙정렬을 알기 전에 먼저 힙이라는 자료구조에 대해 알아야한다. 히프는

여러 값들 중에서 가장 큰 값이나 가장 작은 값을 빠르게 찾아내도록 만들어진 자료구조로서 완전 이진트리의 형태이다. 부모 노드의 키 값이 자식 노드의 값보다 항상 크거나 같은 완전 이진트리를 최대히프라고 하고, 부모 노드의 키 값이 자식 노드의 키 값보다 항상 작거나 같은 완전 이진트리를 최소히프라고 한다. 힙정렬은 최대히프와 최소히프를 모두 이용할 수 있는데, 그 방법을 보여주겠다.

■ 서론

소팅이란 순서 없이 배열된 자료를 특정 기준에 따라 오름차순 또는 내림차순으로 자료들을 재배치하는 것이다. 이러한 소팅 방법에는 여러가지가 있지만 그 중 대표적인 5 가지를 직접 구현해보고 비교 분석 해보았다.

■ 본론

1) 소팅 알고리즘 소스코드 구현

I. Insertion Sort

```
class InsertionSort
{
    public static void sort(int array[])
    {
        for(int i=1;i<array.length;i++)
        {
            int j=i;
            while((j>0 && array[j-1]>array[j]))
            {
                int temp=array[j-1];
                array[j-1] = array[j];
                array[j] = temp;
                j--;
            }
        }
    }
}
```

Insertion Sort는 이름 그대로 배열의 요소들을 차례대로 적절한 위치에 삽입해서 정렬하는 방법이다. For문의 시작에는 Array[1]의 값을 적절히 정렬해준다. Array[0]과 Array[1]의 크기를 비교해서 Array[0]의 크기가 Array[1]보다 크면 두 값의 위치를 바꿔준다. 즉, index가 1인데까지는 모두 정렬되게 되는 것이다. 다시 for문으로 돌아와 이번에는 Array[2]까지

고려하여 index가 작은 요소의 크기가 크면 위치를 바꿔준다. 결국 index가 2인데까지는 모두 정렬되게 되는 것이다. 이런 식으로 for문의 i가 array.length-1일때까지 반복되고 모든 요소가 정렬되게 된다.

II. Selection Sort

```
public class SelectionSort
{
    public static void sort(int a[])
    {
        for(int i=0;i<a.length;i++)
        {
            int min = a[i];
            int mIndex = i;
            for(int j=i;j<a.length;j++)
            {
                if(min>a[j]){min = a[j]; mIndex = j;}
            }
            int t = a[i];
            a[i] = min;
            a[mIndex] = t;
        }
    }
}
```

Selection Sort는 가장 작은 값을 선택해서 정렬하는 것을 반복하는 방법이다. 처음에는 배열의 모든 요소들을 비교해서 가장 작은 값을 min으로 설정하고 그 값을 index 0과 교체한다. 그 다음에는 index 0의 요소를 제외하고 가장 작은 값은 min에 설정하고 그 값을 index 1과 교체한다. 이런 식으로 index array.length-1까지 반복해서 정렬한다.

III. Merge Sort

```
class MergeSort
{
    public static void sort(int[] array)
    {
        sort(array,0,array.length-1);
    }

    private static void sort(int[] array, int start, int end)
    {
        if(start == end)
            return;
        else
        {
            //나누는 작업
            int middle = (start + end)/2;

            //recursion
            sort(array,start,middle);
            sort(array,middle+1,end);

            //temp에 나눠졌다 병합될 array를 복사하는 작업
            int[] temp = new int[end-start+1];
            for(int i=start;i<=middle;i++)
                temp[i-start] = array[i];
            for(int i=middle+1;i<=end;i++)
                temp[temp.length-(i-middle)] = array[i];

            //merge 작업
            int array1 = 0;
            int array2 = end-start;
            for(int i=start;i<=end;i++)
            {
                if(temp[array1]<temp[array2])
                    array[i] = temp[array1++];
                else
                    array[i] = temp[array2--];
            }
        }
    }
}
```

Merge Sort는 한 배열을 반반씩 나누어 각각 정렬한 후, 다시 합쳐서 하나의 정렬된 리스트로 병합하는 방법이다. 하나로 되어있는 배열을 동일한 크기의 배열 두개로 나눠준다. 재귀호출을 통해 요소가 2개인 배열이 될 때까지 배열을 반복해서 나눠준다. 분할이 완료되었으면 2개 요소의 크기를 비교해서 정렬해주고 정렬된 각각의 배열들을 병합해준다. 병합할때는 병합되는 2개의 배열을 각각의 index 순서대로 크기를 비교하며 새로운 배열에 정렬시킨다.

IV. Quick Sort

```java
public class QuickSort
{
    public static void sort(int[] arr)
    {
        Quick(arr,0,arr.length-1);
    }

    public static void Quick(int[] arr, int start, int end)
    {
        if(start<end)
        {
        int pivot = Partician(arr,start,end);
        Quick(arr,start,pivot-1);
        Quick(arr,pivot+1,end);
        }
    }

    public static int Partician(int[] arr,int start, int end)
    {
        int left = start+1;
        int right = end;
        int pivot = start;

        while(left<right)
        {
            while(arr[pivot]>arr[left]&&left<end)
                left++;
            while(arr[pivot]<=arr[right]&&right>=left)
                right--;
            if(left < right)
            {
                int temp = arr[left];
                arr[left] = arr[right];
                arr[right] = temp;
            }
        }
        int temp = arr[right];
        arr[right] = arr[start];
        arr[start] = temp;

        return right;
    }
}
```

Quick Sort 는 전체 배열을 2 개의 부분으로 나누고 각각의 부분을 다시 Quick Sort 해서 정렬하는 방법이다. Pivot 값을 우선 임의로 설정한 후 피벗을 기준으로 피벗보다 작은 값은 왼쪽에 큰 값은 오른쪽에 정렬한다. 그러면 피벗값은 적절히 정렬되게 되고 피벗의 왼쪽 배열과 피벗의 오른쪽 배열 각각에 재귀호출을 실행해주면 모두 정렬되게 된다.

V. Heap Sort

```java
public class HeapSort
{
    public static void sort(int[] arr)
    {
        heapSort(arr);
    }

    public static void heapSort(int[] arr)
    {
        int[] temp = new int[arr.length+1];
        temp[0]=arr.length;
        for(int i=1;i<=arr.length;i++)
        {
            temp[i]=arr[i-1];
            upheap(temp,i);
        }
        for(int i=0;i<arr.length;i++)
        {
            arr[i]=temp[1];
            temp[1]=temp[temp[0]-i];
            downheap(temp,1,temp[0]-i);
        }
    }

    public static void upheap(int[] array, int branch)
    {
        if(branch==1) return;
        if(array[branch]<array[branch/2])
            swap(array,branch,branch/2);
        upheap(array,branch/2);
    }

    public static void downheap(int[] array, int branch, int end)
    {
        if(branch>end/2) return;
        if(branch*2<end && array[branch]>array[branch*2])
            swap(array,branch,branch*2);
        if(branch*2+1<end && array[branch]>array[branch*2+1])
            swap(array,branch,branch*2+1);
        downheap(array,branch*2,end);
        downheap(array,branch*2+1,end);
    }

    public static void swap(int[] a, int i, int j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

Heap Sort 는 트리구조를 이용하여 정렬하는 방법이다. 배열의 요소들을 차례대로 트리구조로 나열한뒤 upheap 을 한번 해준다. 그러면 가장 큰 원소가 맨 위에 위치하게 되고 그 원소를 원래 배열의 맨 앞에 위치시켜주고 가장 아래의 원소를 가장 위에 위치하게 한다. 그런 뒤 downheap 을 해주면 다시 가장 큰 원소가 맨위 에 위치하고 이를 같은 방법으로 수행해주면 배열이 정렬되게 한다. 그런데 여기서 upheap 이란 아래부터 부모노드와 자식노드의 크기를 비교해서 부모노드가 자식노드보다 항상 크게 해주는 작업이고 downheap 이란 위에서부터 부모노드와 자식노드의 크기를 비교해서 부모노드가 자식노드보다 항상 크게 해주는 작업이다.

2) Sorting Algorithm 성능 비교

각 Sorting Algorithm 간의 성능 비교는 시간 복잡도를 구해보고 직접 실행시켜 정렬되는 시간을 기준으로 비교했다.

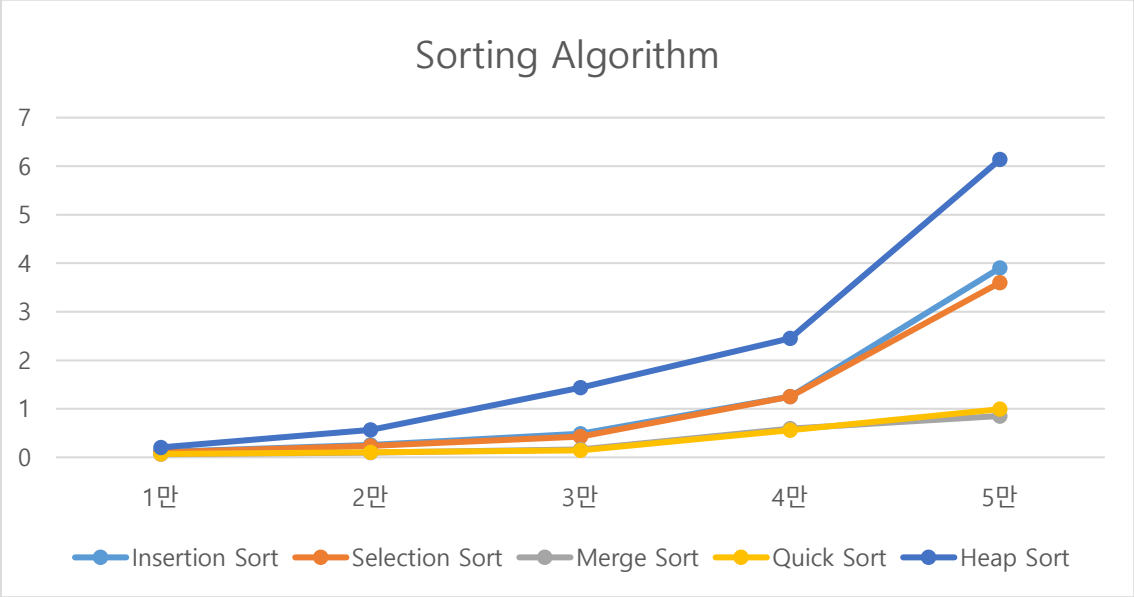
알고리즘	시간 복잡도
Insertion Sort	O(n^2)
Selection Sort	O(n^2)
Merge Sort	O(nlogn)
Quick Sort	O(nlogn)
Heap Sort	O(nlogn)

1)Insertion Sort 와 Selection Sort 는 일반적으로 n 개의 원소에 대해 n 번씩 비교해야하므로 n^2 이고 2) Merge Sort, Quick Sort, Heap Sort 는 2 개씩으로 나누는 작업 logn 에 대하여 n 개의 원소를 비교해야하므로 nlogn 이다.

코드를 직접 실행시켜보고 걸리는 시간을 측정해보았다. System.currentTimeMillis(); 메소드를 사용해서 정렬 전 시간과 정렬 후 시간의 차이를 구하는 방식으로 했다.

알고리즘	10000 개 정렬(초)	20000 개 정렬(초)
Insertion Sort	0.111 초	0.256 초
Selection Sort	0.103 초	0.233 초
Merge Sort	0.064 초	0.095 초
Quick Sort	0.069 초	0.101 초
Heap Sort	0.205 초	0.563 초
30000 개 정렬(초)	40000 개 정렬(초)	50000 개 정렬(초)
0.484 초	1.249 초	3.903 초
0.428 초	1.250 초	3.601 초
0.163 초	0.591 초	0.852 초
0.141 초	0.557 초	0.992 초
1.434 초	2.453 초	6.138 초

■ 결론



결과값을 엑셀에 입력하여 그래프를 그려봤더니 다음과 같았다. Binary 기법을 사용하는 Merge Sort, Quick Sort, Heap Sort 는 Heap Sort 를 제외하고는 예상대로 비교적 성능이 좋았다. O(n^2)의 시간 복잡도를 가진 Insertion Sort 와 Selection Sort 는 예상대로의 점점 빨리 증가하는 그래프가 나타났다.

시간복잡도에 따르면 Heap Sort 도 뛰어난 성능을 보일 것으로 예상되었다. 하지만 Heap Sort 는 배열의 개수가 늘어날수록 효율이 급격히 나빠진다는 것을 발견하였고, 예상과 다르게 덜 효율적이라는 결과가 나왔는다. 이는 Heap Sort 는 트리구조를 따로 저장할 공간이 필요하고 그 곳에 힙구조를 만든 후 다시 빼내는 작업을 해서 인것으로 보인다.

이를 개선하기 위해서는 트리구조를 조금 덜 복잡하게 만든다면 즉, 자식노드의 갯수를 2 보다 크게한다면 조금 더 성능을 높일 수 있지 않을까라고 생각했다.

결론적으로, 앞의 실행결과에 따르면 5 가지 Sorting Algorithm 중 랜덤으로 주어진 데이터의 수가 커질수록 가장 효율적인 것은 Merge Sort 이고, 전반적인 효율성은 Merge Sort 와 Quick Sort 가 비교적 좋다는 것을 알게되었다.