알고리즘 문제풀이

6.정렬과알고리즘

Sorting Algorithm

{2, 1, 4, 3, 5}

{1, 2, 3, 4, 5}

정렬 알고리즘이란

원소들을 일정한 순서대로 열거하는 알고리즘 이다.

정렬 알고리즘을 사용할 때, 상황에 맞게 다음의 기준들로 사용할 알고리즘을 선정한다.

시간 복잡도 (소요되는 시간)
공간 복잡도 (메모리 사용량)

시간, 공간 복잡도는 Big-O 표기법으로 나타낼 수 있다.

또한 정렬되는 항목 외에 충분히 무시할 만한 저장공간만을 더 사용하는 정렬 알고리즘들을 제자리 정렬이라고 한다.

swap()

```
public static void swap(int[] arr, int idx1, int idx2) {
        int tmp = arr[idx1];
        arr[idx1] = arr[idx2];
        arr[idx2] = tmp;
}
```

배열의 두 인덱스의 원소를 교환하는 메소드

```
        tmp = a;
        a = b;
        b = tmp;
```

❗ Big-O 표기법

👉 시간 복잡도를 표기하는 방법

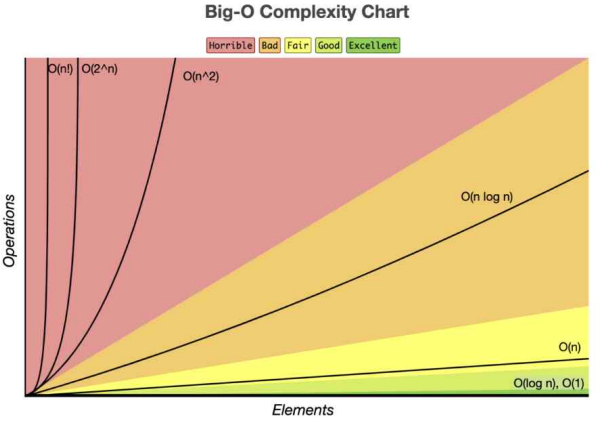
- Big-O(빅-오) ⇒ 상한 점근
- Big-Ω(빅-오메가) ⇒ 하한 점근
- Big-θ(빅-세타) ⇒ 그 둘의 평균
- 위 세 가지 표기법은 시간 복잡도를 각각 최악, 최선, 중간(평균)의 경우에 대하여 나타내는 방법이다.

👉 가장 자주 사용되는 표기법은?

- **빅오 표기법**은 **최악의 경우를 고려**하므로, 프로그램이 실행되는 과정에서 소요되는 **최악의 시간까지 고려**할 수 있기 때문이다.
- **"최소한 특정 시간 이상이 걸린다"** 혹은 **"이 정도 시간이 걸린다"**를 고려하는 것보다 **"이 정도 시간까지 걸릴 수 있다"**를 고려해야 그에 맞는 대응이 가능하다.

👉 Big-O 표기법의 종류

1. O(1)
2. O(n)
3. O(log n)
4. O(n2)
5. O(2n)



- O(1)는 일정한 복잡도(constant complexity)라고 하며, 입력값이 증가하더라도 시간이 늘어나지 않는다.비교와 교환이 모두 일어날 수 있기 때문에 코드는 단순하지만 성능은 좋지 않다.
- O(n)은 선형 복잡도(linear complexity)라고 부르며, 입력값이 증가함에 따라 시간 또한 같은 비율로 증가하는 것을 의미한다.
- O(log n)은 로그 복잡도(logarithmic complexity)라고 부르며, Big-O표기법 중 O(1) 다음으로 빠른 시간 복잡도를 가진다.
- O(n2)은 2차 복잡도(quadratic complexity)라고 부르며, 입력값이 증가함에 따라 시간이 n의 제곱수의 비율로 증가하는 것을 의미한다.
- O(2n)은 기하급수적 복잡도(exponential complexity)라고 부르며, Big-O 표기법 중 가장 느린 시간 복잡도를 가진다.

버블 정렬(Bubble Sort)

- 정렬 과정에서 거품이 수면으로 올라오는 모습과 흡사하여 지어진 이름이다.
- 여기의 움짤을 보면 왜 버블 정렬인지 이해가 된다.

- 비교와 교환이 모두 일어날 수 있기 때문에 코드는 단순하지만 성능은 좋지 않다.

```
public static void sortByBubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
            }
        }
    }
}
```

선택 정렬(Selection Sort)

- 맨 앞 인덱스부터 차례대로 들어갈 원소를 선택하여 정렬하는 알고리즘이다.

- 가장 작은 값을 선택해서 왼쪽(또는 오른쪽) 위치에 가져다 두는 방식의 알고리즘입니다.

- 교환 횟수는 O(n)으로 적지만, 비교는 모두 진행된다.
- 즉, 버블 정렬보다는 성능이 좋다.

```
public static void sortBySelectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        swap(arr, i, minIdx);
    }
}
```

삽입 정렬(Insertion Sort)

- 인덱스 1의 원소부터 앞 방향으로 들어갈 위치를 찾아 교환하는 정렬 알고리즘이다.
- 정렬이 되어 있는 배열의 경우 O(n)의 속도로 정렬되어 있을수록 성능이 좋다.

```
public static void sortByInsertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        int j = i - 1;
        while (j >= 0 && tmp < arr[j]) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = tmp;
    }
}
```

셸 정렬(Shell Sort)

- 삽입 정렬의 장점을 살리고 단점을 보완한 정렬 알고리즘이다.

- 삽입 정렬의 단점은 (n - 1)번째 인덱스 원소의 들어가야할 자리가 0번째 인덱스라면 많은 swap()을 해야하는 것이다.

- 단점을 보완하기 위하여 간격을 정하여 배열을 부분 배열들로 나누어 어느정도 정렬 시키고, 다시 간격을 줄여 정렬시키는 것을 반복한다.
- 여기의 영상을 한번 시청하면 이해가 빠를 것이다.

- 평균 시간 복잡도가 O(n^1.5)로 개선된다.

```
public static void sortByShellSort(int[] arr) {
   for (int h = arr.length / 2; h > 0; h /= 2) {
      for (int i = h; i < arr.length; i++) {
         int tmp = arr[i];
         int j = i - h;
         while (j >= 0 && arr[j] > tmp) {
            arr[j + h] = arr[j];
            j -= h;
         }
         arr[j + h] = tmp;
      }
   }
}
```

합병(병합) 정렬(Merge Sort)

- 분할 정복 알고리즘 중 하나이다.

- 분할,정복,병합을 모두 수행하는 알고리즘

- 배열의 길이가 1이 될 때까지 2개의 부분 배열로 분할한다.
- 분할이 완료됐으면 다시 2개의 부분 배열을 합병하고 정렬한다.
- 모든 부분 배열이 합병될 때 까지 반복한다.

- 시간 복잡도가 O(nlog n)으로 빠르지만, 아래 코드를 보면 tmpArr을 사용해야 돼서 제자리 정렬보다 O(n)만큼 추가적인 메모리가 사용되는 단점이 있다.
- 보통은 재귀함수로 구현하므로 이 것또한 메모리를 많이 사용하게 된다.

```
public static void sortByMergeSort(int[] arr) {
    int[] tmpArr = new int[arr.length];
    mergeSort(arr, tmpArr, 0, arr.length - 1);
}
public static void mergeSort(int[] arr, int[] tmpArr, int left, int right) {
    if (left < right) {
        int m = left + (right - left) / 2;
        mergeSort(arr, tmpArr, left, m);
        mergeSort(arr, tmpArr, m + 1, right);
        merge(arr, tmpArr, left, m, right);
    }
}
```

//다음페이지에 소스계속

```
public static void merge(int[] arr, int[] tmpArr, int left, int mid, int right) {
    for (int i = left; i <= right; i++) {
        tmpArr[i] = arr[i];
    }
    int part1 = left;
    int part2 = mid + 1;
    int index = left;
```

//다음페이지에 소스계속

```
    while (part1 <= mid && part2 <= right) {
        if (tmpArr[part1] <= tmpArr[part2]) {
            arr[index] = tmpArr[part1];              part1++;
        } else {
            arr[index] = tmpArr[part2];              part2++;
        }
        index++;
    }
    for (int i = 0; i <= mid - part1; i++) {
        arr[index + i] = tmpArr[part1 + i];
    }
}//end merge
```

힙 정렬(Heap Sort)

- 오름차 순 정렬일 때 최대힙을 사용하는 정렬이다. (내림차는 최소힙)
- 최대힙을 배열로 구현하면 0번째 인덱스가 가장 큰 수라는 점을 사용한다.

- 시간 복잡도가 O(nlog n)으로 합병정렬, 퀵정렬과 동일하지만 실상 성능은 더 낮게 나온다.

- 매번 루트에서 최대 값을 뺄 때마다 heapify()를 사용하여 다시 최대힙으로 만들어야 해서 그렇다.
- 필자가 구현한 코드로 볼 때는 메모리는 다른 두 정렬보다 적게 사용된다는 장점이 있다.

```
 public static void sortByHeapSort(int[] arr) {
    for (int i = arr.length / 2 - 1; i < arr.length; i++) {
        heapify(arr, i, arr.length - 1);
    }
    for (int i = arr.length - 1; i >= 0; i--) {
        swap(arr, 0, i);
        heapify(arr, 0, i - 1);
    }
}
```

//다음페이지에 소스계속

```
public static void heapify(int[] arr, int parentIdx, int lastIdx) {
    int leftChildIdx;
    int rightChildIdx;
    int largestIdx;
    while (parentIdx * 2 + 1 <= lastIdx) {
        leftChildIdx = (parentIdx * 2) + 1;
        rightChildIdx = (parentIdx * 2) + 2;
        largestIdx = parentIdx;
       //다음페이지에 소스계속
```

```
        if (arr[leftChildIdx] > arr[largestIdx]) {          largestIdx = leftChildIdx;
        }
        if (rightChildIdx <= lastIdx && arr[rightChildIdx] > arr[largestIdx]) {
            largestIdx = rightChildIdx;
        }
        if (largestIdx != parentIdx) {
            swap(arr, parentIdx, largestIdx);
            parentIdx = largestIdx;
        } else {
            break;
        }
    }
}
```

퀵 정렬(Quick Sort)

- 분할기준 피벗(pivot)을 사용한 정렬 알고리즘이며 합병 정렬과 같은 분할 정복 알고리즘이다.
- 합병 정렬은 일정한 부분 리스트로 분할하지만 퀵 정렬은 피벗이 들어갈 위치에 따라 불균형하다.
- 합병 정렬과 속도가 비슷하고 힙 정렬보다 빠르지만, 최악의 경우 O(n^2)만큼 걸린다는 점, 보통 재귀로 구현하기 때문에 메모리를 더 사용할 수 있다는 단점이 있다.
- 최악의 경우는 피봇을 최솟값이나 최댓값 으로 선택하여 부분 배열이 한쪽으로 계속 몰리는 경우이다.

```
public static void sortByQuickSort(int[] arr) {
    quickSort(arr, 0, arr.length - 1);
}
public static void quickSort(int[] arr, int left, int right) {
    int part = partition(arr, left, right);
    if (left < part - 1) {
        quickSort(arr, left, part - 1);
    }
    if (part < right) {
        quickSort(arr, part, right);
    }
}
```
//다음페이지에 소스계속

```
public static int partition(int[] arr, int left, int right) {
    int pivot = arr[(left + right) / 2];
    while (left <= right) {
        while (arr[left] < pivot) {          left++;       }
        while (arr[right] > pivot) {          right--;       }
        if (left <= right) {
            swap(arr, left, right);
            left++;
            right--;
        }
    }
    return left;
}
```

1. 선택 정렬

설명

N개이 숫자가 입력되면 오름차순으로 정렬하여 출력하는 프로그램을 작성하세요.

정렬하는 방법은 선택정렬입니다.

입력

첫 번째 줄에 자연수 N(1<=N<=100)이 주어집니다.

두 번째 줄에 N개의 자연수가 공백을 사이에 두고 입력됩니다. 각 자연수는 정수형 범위 안에 있습니다.

출력

오름차순으로 정렬된 수열을 출력합니다.

예시 입력 1

```
6
13 5 11 7 23 15
```

예시 출력 1

```
5 7 11 13 15 23
```

2. 버블 정렬

설명

N개이 숫자가 입력되면 오름차순으로 정렬하여 출력하는 프로그램을 작성하세요.

정렬하는 방법은 버블정렬입니다.

입력

첫 번째 줄에 자연수 N(1<=N<=100)이 주어집니다.

두 번째 줄에 N개의 자연수가 공백을 사이에 두고 입력됩니다. 각 자연수는 정수형 범위 안에 있습니다.

출력

오름차순으로 정렬된 수열을 출력합니다.

예시 입력 1

```
6
13 5 11 7 23 15
```

예시 출력 1

```
5 7 11 13 15 23
```

3. 삽입 정렬

설명

N개이 숫자가 입력되면 오름차순으로 정렬하여 출력하는 프로그램을 작성하세요.

정렬하는 방법은 삽입정렬입니다.

입력

첫 번째 줄에 자연수 N(1<=N<=100)이 주어집니다.

두 번째 줄에 N개의 자연수가 공백을 사이에 두고 입력됩니다. 각 자연수는 정수형 범위 안에 있습니다.

출력

오름차순으로 정렬된 수열을 출력합니다.

예시 입력 1

```
6
11 7 5 6 10 9
```

예시 출력 1

```
5 6 7 9 10 11
```